Proceedings of the
Prague Stringology Conference 2008

Edited by Jan Holub and Jan Žďárek

Department of Computer Science
and Engineering

September 2008

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

Proceedings of the Prague Stringology Conference 2008
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 Fax: +420-2-2492-3325
Printed by Česká technika – Naklatelství ČVUT, Thákurova 550/1, Praha 6, 160 41, Czech Republic

© Czech Technical University in Prague, Czech Republic, 2008

ISBN 978-80-01-04145-1

Conference Organization

Program Committee

Amihood Amir	(Bar-Ilan University, Israel)
Gabriela Andrejková	(P. J. Šafárik University, Slovakia)
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)
William F. Smyth	(McMaster University, Canada)
Bruce W. Watson	(Technische Universiteit Eindhoven, Netherlands)

Organizing Committee

Miroslav Balík
Jan Holub
Bořivoj Melichar
Ladislav Vagner
Michal Voráček
Jan Žďárek

External Referees

Pavlos Antoniou
Miroslav Balík
Jérémie Bourdon
Arturo Carpi
Joseph Chan
Loek Cleophas
Jackie Daykin
Lucian Ilie
Inuka Jayasekera
Arnaud Lefebvre
Spiros Michalakopoulos
Laurent Mouchard
Elise Prieur
Tinus Strauss
Niko Valimaki

Preface

The proceedings in your hands contains the papers presented in the Prague Stringology Conference 2008 (PSC'08) which was held at the Department of Computer Science and Engineering of the Czech Technical University in Prague, Czech Republic, on September 1–3, 2008. 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 the road coloring problem.

The Prague Stringology Conference has a long tradition. PSC'08 is the thirteenth 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 2001–2006 preceded this conference. The proceedings of these workshops and 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 series of stringology conferences was interrupted in 2007 when the members of the Prague Stringology Club were honoured to organize Conference on Implementation and Application of Automata 2007 (CIAA2007).

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, sequences, and trees with emphasis on finite automata theory. The first event organized by the Prague Stringology Club was the workshop PSCW'96 featuring only a handful of 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 and related areas, 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'08 as well as the reviewers. Special thanks go 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'08. 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 2008

Jan Holub

Table of Contents

Invited Talk

Contributed Talks

The Road Coloring and Černy Conjecture

Avraham N. Trahtman

Bar-Ilan University, Dep. of Math., 52900, Ramat Gan, Israel
trakht@macs.biu.ac.il
http://www.cs.biu.ac.il/~trakht

Abstract. A synchronizing word of a deterministic automaton is a word in the alphabet of colors (considered as letters) of its edges that maps the automaton to a single state. A coloring of edges of a directed graph is synchronizing if the coloring turns the graph into a deterministic finite automaton possessing a synchronizing word.
The road coloring problem is the problem of synchronizing coloring of a directed finite strongly connected graph with constant outdegree of all its vertices if the greatest common divisor of lengths of all its cycles is one. The problem was posed by Adler, Goodwyn and Weiss over 30 years ago and evoked noticeable interest among the specialists in the theory of graphs, deterministic automata and symbolic dynamics.
The positive solution of the road coloring problem is presented.
Some consequences on the length of the synchronizing word are discussed.

Keywords: road coloring problem, graph, deterministic finite automaton, synchronization

Introduction

The road coloring problem originates in [2] and was stated explicitly in [1] for a strongly connected directed finite graph with constant outdegree of all its vertices where the greatest common divisor (gcd) of lengths of all its cycles is one. The edges of the graph are unlabelled. The task is to find a labelling of the edges that turns the graph into a deterministic finite automaton possessing a synchronizing word. So the road coloring problem is connected with the problem of existence of synchronizing word for deterministic complete finite automaton.

The condition on gcd is necessary [1], [6]. It can be replaced by the equivalent property that there does not exist a partition of the set of vertices on subsets V_1, V_2, ..., $V_{k+1} = V_1$ $(k > 1)$ such that every edge which begins in V_i has its end in V_{i+1} [6], [20]. The outdegree of the vertex can be considered also as the size of an alphabet where the letters denote colors.

The road coloring problem is important in automata theory: a synchronizing coloring makes the behavior of an automaton resistant against input errors since, after detection of an error, a synchronizing word can reset the automaton back to its original state, as if no error had occurred. The problem appeared first in the context of symbolic dynamics and is important also in this area.

Together with the Černy conjecture [22], [24], the road coloring problem belongs to the most fascinating problems in the theory of finite automata. The problem was discussed even in "Wikipedia" – the popular Internet Encyclopedia. However, at the same time it was considered as a "notorious open problem" [18], [6] and "unfeasible" [13]. For some positive results in this area see [4], [5], [11], [12], [13], [15], [16], [20], [21].

Avraham N. Trahtman: The Road Coloring and Černy Conjecture, pp. 1–12.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

The road coloring conjecture is settled in the affirmative: A finite strong digraph with constant outdegree has a synchronizing coloring if and only if the greatest common divisor of the lengths of its cycles is 1.

The concept of a stable pair of states [6], [16] of Culik, Karhumaki and Kari with corresponding results and consequences is used below. The first version of our paper had also used results from [11]. However, we are now able to simplify the proof using idea from [3], [25] and [26].

A problem of the minimal length of synchronizing word, best known as Černy's conjecture, was raised independently by distinct authors. Jan Černy found in 1964 [7] n-state complete DFA with shortest synchronizing word of length $(n-1)^2$ for alphabet size $q = 2$. He conjectured that it is an upper bound for the length of the shortest synchronizing word for any n-state complete DFA. The best known upper bound is now equal to $(n^3 - n)/6$ [10], [17]. The conjecture holds true for a lot of automata, but in general the problem still remains open. Moreover, the examples of automata with shortest synchronizing word of length $(n-1)^2$ are infrequent. After the sequence found by Černy and example of Černy, Piricka and Rosenauerova [8] of 1971 for $q = 2$, the next such example was found by Kari [16] only in 2001 for $n = 6$ and $q = 2$. Roman [23] had found an analogous example for $n = 5$ and $q = 3$ in 2004. There are no examples of automata for the time being such that the length of the shortest synchronizing word is greater than $(n-1)^2$.

We use a new efficient algorithm for finding a synchronizing word. The known algorithm of Eppstein [9] finds a synchronizing word for n-state DFA in $O(n^3 + n^2 q)$ time. The actual running time of our algorithm ($O(n^2 q)$) on a lot of examples proved to be less than in the case of $O(n^3 q)$ time complexity (the worst case). It gives a chance to extend noticeably the class of considered DFA.

The program had studied all automata with strongly connected transition graph of size $n \leq 10$ for $q = 2$, of size $n \leq 8$ for $q \leq 3$ and of size $n \leq 7$ for $q \leq 4$. All known together with some new examples of DFA with shortest synchronizing word of length $(n-1)^2$ from this class of automata were obtained. So all examples of DFA with shortest synchronizing word of length $(n-1)^2$ in this area are known for today. The size of the alphabet of the examples is two or three. The contradictory examples for the Černy conjecture do not exist in this class of automata. Moreover, the program does not find examples of DFA with reset word of length $(n-1)^2$ for $n > 4$ as well as for $q > 3$. No such examples exist also for alphabet of size four if $n \leq 7$ and of size three if $n \leq 8$.

All examples on the Černy border $(n-1)^2$ except one have loops and therefore by some recoloring have shortest synchronizing word of length not greater than $n-1$. It supports the conjecture that by some coloring every synchronizing automaton has synchronizing word of length less than $(n-1)^2$.

Preliminaries

A finite directed strongly connected graph with constant outdegree of all its vertices where the gcd of lengths of all its cycles is one will be called *AGW graph* as aroused by Adler, Goodwyn and Weiss.

The bold letters will denote the vertices of a graph (the states of an automaton).

If there exists a path in an automaton from the state $\mathbf{p}$ to the state $\mathbf{q}$ and the edges of the path are consecutively labelled by $\sigma_1, ..., \sigma_k$, then for $s = \sigma_1 ... \sigma_k \in \Sigma^+$ let us write $\mathbf{q} = \mathbf{p}s$ and $\mathbf{p} \succeq \mathbf{r}$.

Let Ps be the map of the subset P of states of an automaton by help of $s \in \Sigma^+$ and let Ps^{-1} be the maximal set of states Q such that $Qs \subseteq P$. For the transition graph Γ of an automaton let Γs denote the map of the set of states of the automaton.

$|P|$ – the size of the subset P of states from an automaton (of vertices from a graph).

A word $s \in \Sigma^+$ is called a *synchronizing* (or*2-reset*) word of the automaton with transition graph Γ if $|\Gamma s| = 1$.

A coloring of a directed finite graph is *synchronizing* if the coloring turns the graph into a deterministic finite automaton possessing a synchronizing word.

A pair of distinct states $\mathbf{p}, \mathbf{q}$ of an automaton (of vertices of the transition graph) will be called *synchronizing* if $\mathbf{p}s = \mathbf{q}s$ for some $s \in \Sigma^+$. In the opposite case, if for any s $\mathbf{p}s \neq \mathbf{q}s$, we call the pair *deadlock*.

A synchronizing pair of states $\mathbf{p}$, $\mathbf{q}$ of an automaton is called *stable* if for any word u the pair $\mathbf{p}u, \mathbf{q}u$ is also synchronizing [6], [16].

We call the set of all outgoing edges of a vertex a *bunch* if all these edges are incoming edges of only one vertex.

The subset of states (of vertices of the transition graph Γ) of maximal size such that every pair of states from the set is deadlock will be called an *F-clique*.

A state [a vertex] $\mathbf{r}$ is called *sink* of an automaton [of a graph] if $\mathbf{p} \succeq \mathbf{r}$ for all states $\mathbf{p}$.

The direct product Γ^2 of two copies of the graph Γ over the alphabet Σ consists of vertices $(\mathbf{p}, \mathbf{r})$ and edges $(\mathbf{p}, \mathbf{r}) \rightarrow (\mathbf{p}\sigma, \mathbf{r}\sigma)$ labelled by σ. Here $\mathbf{p}, \mathbf{r} \in \Gamma$, $\sigma \in \Sigma$.

1 Some properties of F-clique

The road coloring problem was formulated for *AGW* graphs [1] and only such graphs are considered below. We exclude from the consideration also the primitive cases of graphs with loops and of only one color [1], [20].

Let us recall that a binary relation ρ on the set of the states of an automaton is called congruence if ρ is equivalence and for any word u from $\mathbf{p}\ \rho\ \mathbf{q}$ follows $\mathbf{p}u\ \rho\ \mathbf{q}u$. Let us formulate an important result from [16] in the following form:

Theorem 1. *[16] Let us consider a coloring of AGW graph Γ. Stability of states is a binary relation on the set of states of the obtained automaton; denote this relation by ρ. Then ρ is a congruence relation, Γ/ρ presents an AGW graph and synchronizing coloring of Γ/ρ implies synchronizing recoloring of Γ.*

Lemma 2. *Let F be F-clique via some coloring of AGW graph Γ. For any word s the set Fs is also an F-clique and any state [vertex] $\mathbf{p}$ belongs to some F-clique.*

Proof. Any pair $\mathbf{p}$, $\mathbf{q}$ from an F-clique F is a deadlock. To be deadlock is a stable binary relation, therefore for any word s the pair $\mathbf{p}s$, $\mathbf{q}s$ from Fs also is a deadlock. So all pairs from Fs are deadlocks.

For any $\mathbf{r}$ from a strongly connected graph Γ, there exists a word u such that $\mathbf{r} = \mathbf{p}u$ for $\mathbf{p}$ from the F-clique F, whence $\mathbf{r}$ belongs to the F-clique Fu.

Lemma 3. *Let A and B ($|A| > 1$) be distinct F-cliques via some coloring without stable pairs of the AGW graph Γ. Then $|A| - |A \cap B| = |B| - |A \cap B| > 1$.*

Proof. Let us assume the contrary: $|A| - |A \cap B| = 1$. By the definition of F-clique, $|A| = |B|$ and $|B| - |A \cap B| = 1$, too.

The pair of states $\mathbf{p} \in A \setminus B$ and $\mathbf{q} \in B \setminus A$ is not stable. Therefore for some word s the pair $(\mathbf{p}s, \mathbf{q}s)$ is a deadlock. Any pair of states from the F-clique A and from the F-clique B as well as from F-cliques As and Bs is a deadlock. So any pair of states from the set $(A \cup B)s$ is a deadlock.

One has $|(A \cup B)s| = |A| + 1 > |A|$ in spite of maximality of the size of F-clique A among the sets of states such that every pair of its states is deadlock.

Lemma 4. *Let some vertex of AGW graph Γ have two incoming bunches. Then any coloring of Γ has a stable couple.*

Proof. If a vertex $\mathbf{p}$ has two incoming bunches from vertices $\mathbf{q}$ and $\mathbf{r}$, then the couple $\mathbf{q}$, $\mathbf{r}$ is stable for any coloring because $\mathbf{q}\alpha = \mathbf{r}\alpha = \mathbf{p}$ for any letter (color) $\alpha \in \Sigma$.

2 The spanning subgraph of cycles and trees with maximal number of edges in the cycles

Définition 1 *Let us call a subgraph S of the AGW graph Γ a* spanning subgraph *of Γ if to S belong all vertices of Γ and exactly one outgoing edge of every vertex.*

A maximal subtree of the spanning subgraph S with root on a cycle from S and having no common edges with cycles from S is called a tree *of S.*

The length of path from a vertex $\mathbf{p}$ through the edges of the tree of the spanning set S to the root of the tree is called the level *of $\mathbf{p}$ in S.*

Remark 5. Any spanning subgraph S consists of disjoint cycles and trees with roots on cycles; any tree and cycle of S is defined identically, the level of the vertex from cycle is zero, the vertices of trees except root have positive level, the vertex of maximal positive level has no incoming edge from S.

Lemma 6. *Let L be a set of vertices of level l from some tree of the spanning subgraph S of AGW graph Γ and let all edges of S have a color α by some coloring of Γ. Then for any F-clique F of the coloring holds $|F \cap L| \leq 1$.*

Proof. Some power of α synchronizes all states of given level of the tree and maps them into the root. Any couple of states from an F-clique could not be synchronized and therefore could not belong to L.

Lemma 7. *Let AGW graph Γ have a spanning subgraph R of only disjoint cycles (without trees). Then Γ also has another spanning subgraph with exactly one vertex of maximal positive level.*

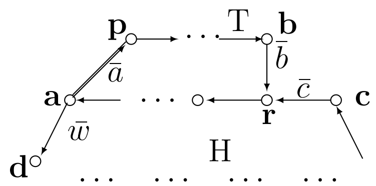
Proof. The spanning subgraph R has only cycles and therefore the levels of all vertices are equal to zero. In view of gcd =1 in the strongly connected graph Γ, not all edges belong to a bunch. Therefore there exist two edges $u = \mathbf{p} \to \mathbf{q} \notin R$ and $v = \mathbf{p} \to \mathbf{s} \in R$ with common first vertex $\mathbf{p}$ but such that $\mathbf{q} \neq \mathbf{s}$. Let us replace the edge $v = \mathbf{p} \to \mathbf{s}$ from R by u. Then only the vertex $\mathbf{s}$ has maximal level $L > 0$ in the new spanning subgraph.

Lemma 8. *Let any vertex of an AGW graph Γ have no two incoming bunches. Then Γ has a spanning subgraph such that all its vertices of maximal positive level belong to one non-trivial tree.*

Proof. Let us consider a spanning subgraph R with a maximal number of vertices [edges] in its cycles. In view of Lemma 7, suppose that R has non-trivial trees and let $L > 0$ be the maximal value of the level of a vertex.

Further consideration is necessary only if at least two vertices of level L belong to distinct trees of R with distinct roots.

Let us consider a tree T from R with vertex $\mathbf{p}$ of maximal level L and edge $\bar{b}$ from vertex $\mathbf{b}$ to the tree root $\mathbf{r} \in T$ on the path of length L from $\mathbf{p}$. Let the root $\mathbf{r}$ belong to the cycle H of R with the edge $\bar{c} = \mathbf{c} \to \mathbf{r} \in H$. There exists also an edge $\bar{a} = \mathbf{a} \to \mathbf{p}$ that does not belong to R because Γ is strongly connected and $\mathbf{p}$ has no incoming edge from R.



Let us consider the path from $\mathbf{p}$ to $\mathbf{r}$ of maximal length L in T. Our aim is to extend the maximal level of the vertex on the extension of the tree T much more than the maximal level of vertex of other trees from R. We plan to use the following three changes:

1) replace the edge $\bar{w}$ from R with first vertex $\mathbf{a}$ by the edge $\bar{a} = \mathbf{a} \to \mathbf{p}$,

2) replace the edge $\bar{b}$ from R by some other outgoing edge of the vertex $\mathbf{b}$,

3) replace the edge $\bar{c}$ from R by some other outgoing edge of the vertex $\mathbf{c}$.

If one of the ways does not succeed let us go to the next assuming the situation in which the previous way fails and excluding the successfully studied cases. So we diminish the considered domain. We can use sometimes two changes together. Let us begin with

1) Suppose first $\mathbf{a} \notin H$. If $\mathbf{a}$ belongs to a path in T from $\mathbf{p}$ to $\mathbf{r}$ then a new cycle with part of the path and edge $\mathbf{a} \to \mathbf{p}$ is added to R extending the number of vertices in its cycles in spite of the choice of R. In opposite case the level of $\mathbf{a}$ in the new spanning subgraph is $L+1$ and the vertex $\mathbf{r}$ is a root of the new tree containing all vertices of maximal level (in particular, the vertex $\mathbf{a}$ or its ancestors in R).

So let us assume $\mathbf{a} \in H$ and suppose $\bar{w} = \mathbf{a} \to \mathbf{d} \in H$. In this case the vertices $\mathbf{p}$, $\mathbf{r}$ and $\mathbf{a}$ belong to a cycle H_1 with new edge $\bar{a}$ of a new spanning subgraph R_1. So we have the cycle $H_1 \in R_1$ instead of $H \in R$. If the length of path from $\mathbf{r}$ to $\mathbf{a}$ in H is r_1 then H_1 has length $L + r_1 + 1$. A path to $\mathbf{r}$ from the vertex $\mathbf{d}$ of the cycle H remains in R_1. Suppose its length is r_2. So the length of the cycle H is $r_1 + r_2 + 1$. The length of the cycle H_1 is not greater than the length of H because the spanning subgraph R has maximal number of edges in its cycles. So $r_1 + r_2 + 1 \geq L + r_1 + 1$, whence $r_2 \geq L$. If $r_2 > L$, then the length r_2 of the path from $\mathbf{d}$ to $\mathbf{r}$ in a tree of R_1 (and the level of $\mathbf{d}$) is greater than L and the level of $\mathbf{d}$ (or of some other ancestor of $\mathbf{r}$ in a tree from R_1) is the desired unique maximal level.

So assume for further consideration $L = r_2$ and $\mathbf{a} \in H$. Analogously, for any vertex of maximal level L with root in the cycle H and incoming edge from a vertex $\mathbf{a}_1$ the proof can be reduced to the case $\mathbf{a}_1 \in H$ and $L = r_2$ for the corresponding new value of r_2.

2) Suppose the set of outgoing edges of the vertex $\mathbf{b}$ is not a bunch. So one can replace in R the edge $\bar{b}$ from the vertex $\mathbf{b}$ by an edge $\bar{v}$ from $\mathbf{b}$ to a vertex $\mathbf{v} \neq \mathbf{r}$.

The vertex $\mathbf{v}$ could not belong to T because in this case a new cycle is added to R and therefore a new spanning subgraph has a number of vertices in the cycles greater than in R.

If the vertex $\mathbf{v}$ belongs to another tree of R but not to cycle, then T is a part of a new tree T_1 with a new root of a new spanning subgraph R_1 and the path from $\mathbf{p}$ to the new root is extended. So only the tree T_1 has states of new maximal level.

If $\mathbf{v}$ belongs to some cycle $H_2 \neq H$ from R, then together with replacing $\bar{b}$ by $\bar{v}$, we replace also the edge $\bar{w}$ by $\bar{a}$. So we extend the path from $\mathbf{p}$ to the new root $\mathbf{v}$ at least by the edge $\bar{a} = \mathbf{a} \rightarrow \mathbf{p}$ and by almost all edges of H. Therefore the new maximal level $L_1 > L$ has either the vertex $\mathbf{d}$ or its ancestors from the old spanning subgraph R.

Now there remains only the case when $\mathbf{v}$ belongs to the cycle H. The vertex $\mathbf{p}$ also has level L in new tree T_1 with root $\mathbf{v}$. The only difference between T and T_1 (just as between R and R_1) is the root and the incoming edge of the root. The new spanning subgraph R_1 has also a maximal number of vertices in cycles just as R. Let r_3 be the length of the path from $\mathbf{d}$ to the new root $\mathbf{v} \in H$.

For the spanning subgraph R_1, one can obtain $L = r_3$ just as it was done on the step 1) for R. From $\mathbf{v} \neq \mathbf{r}$ follows $r_3 \neq r_2$, though $L = r_3$ and $L = r_2$.

So for further consideration suppose that the set of outgoing edges of the vertex $\mathbf{b}$ is a bunch to $\mathbf{r}$.

3) The set of outgoing edges of the vertex $\mathbf{c}$ is not a bunch to $\mathbf{r}$ because $\mathbf{r}$ has another bunch from $\mathbf{b}$.

Let us replace in R the edge $\bar{c}$ by an edge $\bar{u} = \mathbf{c} \rightarrow \mathbf{u}$ such that $\mathbf{u} \neq \mathbf{r}$. The vertex $\mathbf{u}$ could not belong to the tree T because in this case the cycle H is replaced by a cycle with all vertices from H and some vertices of T whence its length is greater than $|H|$. Therefore the new spanning subgraph has a number of vertices in its cycles greater than in spanning subgraph R in spite of the choice of R.

So remains the case $\mathbf{u} \notin T$. Then the tree T is a part of a new tree with a new root and the path from $\mathbf{p}$ to the new root is extended at least by a part of H from the former root $\mathbf{r}$. The new level of $\mathbf{p}$ therefore is maximal and greater than the level of any vertex in some another tree.

Thus anyway there exists a spanning subgraph with vertices of maximal level in one non-trivial tree.

Theorem 9. *Any AGW graph Γ has a coloring with stable couple.*

Proof. By Lemma 4, in the case of vertex with two incoming bunches Γ has a coloring with stable couples. In opposite case, by Lemma 8, Γ has a spanning subgraph R such that the vertices of maximal positive level L belong to one tree of R.

Let us give to the edges of R the color α and denote by C the set of all vertices from the cycles of R. Then let us color the remaining edges of Γ by other colors arbitrarily.

By Lemma 2, in a strongly connected graph Γ for every word s and F-clique F of size $|F| > 1$, the set Fs also is an F-clique of the same size and for any state $\mathbf{p}$ there exists an F-clique F such that $\mathbf{p} \in F$.

In particular, some F has non-empty intersection with the set N of vertices of maximal level L. The set N belongs to one tree, whence by Lemma 6 this intersection has only one vertex. The word α^{L-1} maps F on an F-clique F_1 of size $|F|$. One has $|F_1 \setminus C| = 1$ because the sequence of edges of color α from any tree of R leads to the root of the tree, the root belongs to a cycle colored by α from C and only for the set

N with vertices of maximal level holds $N\alpha^{L-1} \not\subseteq C$. So $|N\alpha^{L-1} \cap F_1| = |F_1 \setminus C| = 1$ and $|C \cap F_1| = |F_1| - 1$.

Let the integer m be a common multiple of the lengths of all considered cycles from C colored by α. So for any $\mathbf{p}$ from C as well as from $F_1 \cap C$ holds $\mathbf{p}\alpha^m = \mathbf{p}$. Therefore for an F-clique $F_2 = F_1\alpha^m$ holds $F_2 \subseteq C$ and $C \cap F_1 = F_1 \cap F_2$.

Thus two F-cliques F_1 and F_2 of size $|F_1| > 1$ have $|F_1| - 1$ common vertices. So $|F_1 \setminus (F_1 \cap F_2)| = 1$. Consequently, in view of Lemma 3, there exists a stable couple in the considered coloring.

Theorem 10. *Every AGW graph Γ has synchronizing coloring.*

The proof follows from Theorems 9 and 1.

3 Some auxiliary properties

Lemma 11. *Suppose $\mathbf{p} \notin \Gamma s$. Then $\mathbf{p} \notin \Gamma us$ for any word u.*

Proof follows from $\Gamma u \subseteq \Gamma$.

Lemma 12. *Suppose $\mathbf{p} \notin \Gamma s$ for a word s and a state $\mathbf{p}$ of transition graph Γ of DFA.*

Then there exist two minimal integer k and r such that $\mathbf{p}s^k = \mathbf{p}s^{k+r}$. The pair of states $\mathbf{p}, \mathbf{p}s^r$ has 2-reset word s^k and for every $i < k$ the pair of states $\mathbf{p}s^i, \mathbf{p}s^{r+i}$ has 2-reset word s^{k-i}. The word s^k is a 2-reset word for at least k different pairs of states.
In the case $r = 1$, the word s^k maps the set of states $\mathbf{p}, \mathbf{p}s, ..., \mathbf{p}s^k$ on $\mathbf{p}s^k$.

Proof. The sequence $\mathbf{p}s, \mathbf{p}s^2, ..., \mathbf{p}s^t, ...$ is finite and belongs to Γs. Therefore such k and r exist. Two states $\mathbf{p}s^i$ and $\mathbf{p}s^{r+i}$ are mapped by the power s^{k-i} on $\mathbf{p}s^k = \mathbf{p}s^{k+r}$ as well as the states $\mathbf{p}$ and $\mathbf{p}s^r$ are mapped by the power s^k on $\mathbf{p}s^k$. All states $\mathbf{p}s^i$ are distinct for $i \leq k$, whence the word s^k unites at least k distinct pairs of states.

In the case $r = 1$, $\mathbf{p}s^k = \mathbf{p}s^j s^k$ for any j. All states $\mathbf{p}s^i$ are distinct for $0 \geq i \leq k$, whence the word s^k unites in this case at least $k + 1$ distinct states.

Lemma 13. *Suppose $\mathbf{r}\alpha = \mathbf{t}\alpha$ for a letter α and two distinct states $\mathbf{r}$, $\mathbf{t}$ of transition graph Γ of DFA and let the states $\mathbf{r}$ and $\mathbf{r}\alpha$ be consecutive states of a cycle C of Γ.*

Then there exists a word s of length of the cycle C such that $\mathbf{r}s = \mathbf{r}$ and $|\Gamma s| < |\Gamma|$. For some state $\mathbf{p} \in \Gamma \setminus \Gamma s$ there exists a minimal integer k such that $\mathbf{p}s^k = \mathbf{p}s^{k+1}$. The pair of states $\mathbf{p}, \mathbf{p}s^k$ has 2-reset word s^k and for every $i < k$ the pair of states $\mathbf{p}s^i, \mathbf{p}s^k$ has 2-reset word s^{k-i}. The word s^k unites at least $k + 1$ distinct states.

Proof. A word s with first letter α can be obtained from consecutive letters on the edges of the cycle C. Therefore $|s|$ is equal to the length of the cycle and $\mathbf{r}s = \mathbf{r}$. $|\Gamma s| < |\Gamma|$ follows from $\mathbf{r}\alpha = \mathbf{t}\alpha$.

From $\mathbf{r}s = \mathbf{r} \neq \mathbf{t}$ and $\mathbf{r}\alpha = \mathbf{t}\alpha$ follows that $\mathbf{t}s = \mathbf{r} \neq \mathbf{t}$, whence $\mathbf{r} = \mathbf{t}s^i \neq \mathbf{t}$ for any integer i. In the case $\mathbf{t} \in \Gamma \setminus \Gamma s$ suppose $\mathbf{p} = \mathbf{t}$, and so the state $\mathbf{p}$ is defined.

In opposite case the state $\mathbf{t}$ has by mapping s some preimage $\mathbf{t}s^{-1}$ and in view of $\mathbf{t}s^i \neq \mathbf{t}$ for all i there exists an integer k (only one) such that the state $\mathbf{t}s^{-k}$ belongs to $\Gamma \setminus \Gamma s$. Now suppose $\mathbf{p} = \mathbf{t}s^{-k}$. One has $\mathbf{p}s^k = \mathbf{p}s^{k+1} = \mathbf{r}$ for $\mathbf{p}$ from $\Gamma \setminus \Gamma s$.

So the pair of states $\mathbf{p}, \mathbf{p}s^k$ has 2-reset word s^k and for every $i < k$ the pair of states $\mathbf{p}s^i, \mathbf{p}s^k$ has 2-reset word s^{k-i}. The states $\mathbf{p}s^i$ for $i \leq k$ and $\mathbf{p}$ are distinct because k is unique. The word s^k maps all these states on the state $\mathbf{r}$.

Lemma 14. *Let Γ be strongly connected graph of synchronizing automaton with transition semigroup S. Suppose $\Gamma a = \Gamma b$ for reset words a and b. Then $a = b$. Any reset word is an idempotent.*

Proof. The elements a and b from S induce equal mappings on the set of states of Γ. S can be embedded into the semigroup of all functions on the set of states under composition. Therefore $a = b$ in S. $\Gamma a = \Gamma a^2$, whence $a = a^2$ for any reset word a and the element $a \in S$ is an idempotent.

4 Synchronizing Algorithms

The following help construction was supposed by Eppstein [9]. Let us keep for any pair of states $\mathbf{r}, \mathbf{p}$ the first letter α of the minimal 2-reset word w of the pair together with the length of the word w. The second letter of w is the first letter of the analogical word of the pair of states $\mathbf{r}\alpha, \mathbf{p}\alpha$. Therefore the 2-reset word w of minimal length can be restored on this way. The time and space complexity of this preprocessing is $O(n^2)$ [9] for n-state automaton.

4.1 Checking synchronizability

A help algorithm with $O(n^2q)$ time complexity in the worst case verifies whether or not a given n-state DFA of alphabet size q is synchronizing. The algorithm follows [9]. Our modification of the algorithm finds first all SCC of the graph (the first-depth search is a linear) and then checks the minimal SCC Γ_s of sink states of the graph (if exists). If there is no sink state then the automaton is not synchronizing. Exactly one sink state implies synchronizability. The time and space complexity of the algorithm in both these cases are linear.

Let us consider the graph Γ_s with at least two sink states. The next step is the consideration of Γ_s^2. We unite any pair of states $(\mathbf{p}, \mathbf{r})$ and $(\mathbf{r}, \mathbf{p})$, all states $(\mathbf{r}, \mathbf{r})$ are united in one state $(0, 0)$. Then let us mark sink state $(0, 0)$ and all ancestors of $(0, 0)$ using the first-depth search on the reverse of the obtained graph G. The graph Γ is synchronizing if any node of G will be marked.

4.2 An efficient algorithm for reset word

An efficient semigroup algorithm, essential improvement of the algorithm [9], based on the properties of transition semigroup and inspired mostly by results from the previous section plays a central role in the program.

We consider the square Γ^2 and the reverse graph I of Γ. The graph I is not deterministic for synchronizing graph Γ. Suppose that the graph Γ is synchronizing, all sink states are found on the stage of checking of the synchronizability, the graph Γ^2 and the reverse graph I were build.

Let us find by help of the reverse graph I for any pair of states $\mathbf{r}, \mathbf{p}$ from Γ^2 the first letter of the minimal 2-reset word w of the pair and the length of w [9]. So for any pair $\mathbf{r}, \mathbf{p}$ can be restored a 2-reset word w of minimal length.

Let us order the set of states $(\mathbf{r}, \mathbf{p})$ according to the length of the word w. The ordering can be made linear in the size of the set in the following way:

Let us find first the number c_i of all states $(\mathbf{r}, \mathbf{p})$ with given length i of minimal 2-reset word for any i, then adjust the intervals of size c_i for to place the pairs and then allocate in every interval the pairs with common length. It needs $O(n^2)$ time.

We use also a complementary idea for to reorder the pairs of states. If a word w unites at least two states let us find the number of states united by powers of w and use this value for *complementary* order.

The important part of the preprocessing supposed by Eppstein was the computing of the mapping Γw of the graph Γ induced by the minimal 2-reset word w of the pair of states $\mathbf{r}, \mathbf{p}$. This stage begins from the shortest words w and therefore is linear for any considered pair of states $\mathbf{r}, \mathbf{p}$. Nevertheless, the time complexity of the stage is $O(n^3)$. For to avoid the extremes of this step, our algorithm stops on linear number of pairs. The obtained set G of 2-reset words is considered as a set of generators of some subsemigroup from A and will be marked together with corresponding pairs of states. The time complexity of this step is therefore $O(n^2)$. Let us reorder G in the *complementary* order and use the mapping of the graph induced by powers of generators.

Let Γ_i be consecutive images of the graph $\Gamma = \Gamma_0$ such that for $w_i \in A$ holds $\Gamma_i w_{i+1} = \Gamma_{i+1}$ and $|\Gamma_i| > |\Gamma_{i+1}|$. Let A_i be a semigroup generated by the set w_1, ... w_i. Let us check pairs of states corresponding to the words from G. If the pair belongs to Γ_i then the corresponding minimal reset word w_{i+1} together with its powers may be used for to find the image Γ_{i+1}.

In the case no minimal 2-reset word of a pair from Γ_i was marked, let us consider the products of marked words. If some product unites a pairs of states of Γ_i, then let us use the mapping, mark the product of words and the pair of states. Let us notice that on this step are considered not all marked pairs. The number of considered products must be linear in the size of Γ. The product of two mappings can be found in linear time. Therefore the time complexity of this stage is $O(nk)$ for the defect k of the mapping of Γ_i.

If two considered stages still do not find a reset word, then the new generator must be added to considered subsemigroup A_i. Let us take a pair of states $\mathbf{r}, \mathbf{p}$ from Γ_i with reset word w_i. Suppose $w_i = u_i v_i$ such that the word v_i was marked. Then the mapping w_i can be found in $n|u_i|$ time. Let us notice that only on this step the time complexity may by greater than quadratic.

Lemma 15. *Let Γ_i be consecutive images of the graph $\Gamma = \Gamma_0$ such that for v_i from semigroup A $\Gamma_i v_{i+1} = \Gamma_{i+1}$, $|\Gamma_i| > |\Gamma_{i+1}|$ and $|\Gamma_s| = 1$ for some integer s. Let A_i be a semigroup generated by the set w_1, ... w_i such that $w_i = u_i v_i$ is a reset word for some pair of states from Γ_{i-1} and v_i is a marked element of the subsemigroup A_{i-1}.*

Then the considered algorithm has $max(O(|\Gamma|^2 q), O(|\Gamma||u_1...u_s|)$ time complexity.

Proof. The time complexity of the step of the building of Γ^2 is $O(|\Gamma|^2 q)$. So $O(|\Gamma|^2 q)$ is a lower bound for the complexity of the considered algorithm.

Let the set w_1, ... w_i generate A_i. The creation of the mapping w_i needs $|\Gamma||u_i| + 1$ steps because for the marked element v_i the mapping is known.

The element will be marked and used only if it is either a generator from A_i or a product of two marked elements. With a marked semigroup element will be associated the mapping of Γ defined by the element. The finding of the mapping of the product of two elements with known images is linear in the size of the graph.

We repeat the process with the obtained image Γ_i. The defect of the mapping is growing on every step. After not over than $|\Gamma| - 1$ steps Γ will be synchronized.

As for complexity of the algorithm, let us notice that the length of the synchronizing word found by the algorithm was less than n^2 in all considered cases. The stage of adding of new generators was used only in a small number of cases, only some percents of considered automata. The number of generators of the semigroup A is usually small. For instance, for Černy graphs there are only two generators. Therefore the time complexity of the algorithm is $O(n^2q)$ in majority of cases and the algorithm can be considered as subquadratic.

4.3 An algorithm for reset word of minimal length

A straightforward algorithm for finding synchronizing word of minimal length is used by the program on its last stage. The algorithm is not polynomial in the most worst case (the finding of the synchronizing word of minimal length is NP-hard [9], [19]). The size of the transition semigroup is in general not polynomial in the size of the transition graph. The program for search of minimal reset word uses this algorithm relatively rare.

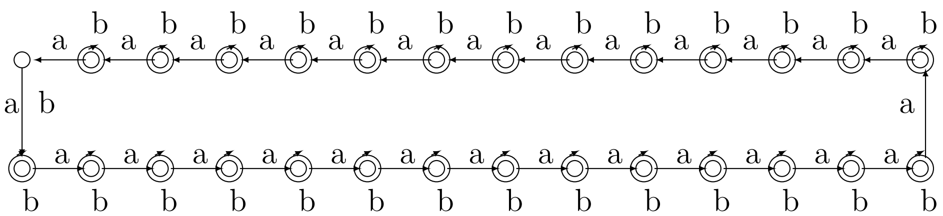
We find mappings of the graph of the automaton induced by the letters of the alphabet of the labels. Mappings with the same set of states are identified. It essentially simplifies the process. Distinct mappings are saved. For this aim, any two mappings must to be compared, so we have $O(s(s-1)/2)$ steps for s mappings.
The mappings correspond to semigroup elements. With any mapping let us connect a previous mapping and the letter that creates the mapping. On this way, the path on the graph of the automaton can be constructed. The time complexity of the considered procedure is $O(nqs^2)$ with $O(ns)$ space complexity.

Proposition 16. *The algorithm finds a list of all words (elements of transition semigroup) of length k where k is growing. The first synchronizing word of the list has minimal length.*

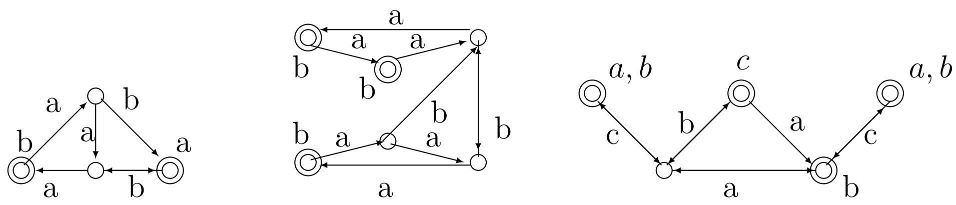
5 Experimental data

The considered synchronization algorithms were used in a program for search of automata with minimal reset word of relatively great length. The program has investigated all complete DFA for $n \leq 10$, $q = 2$ and for $n \leq 7$, $q \leq 4$.
An automaton with k states outside sink SCC A of the transition graph can be mapped on A by word of length not greater than $k(k-1)/2$. Therefore only automata with strongly connected transition graphs need investigation. The graphs with synchronizing proper subgraph obtained by moving off letters from the alphabet are omitted too. In particular, there are no synchronizing 3-state automata for $q \geq 3$ such that by removing any letter the obtained automata are not synchronizing. Therefore such automata are not studied and in the table below for $n = 3$ appears zero.

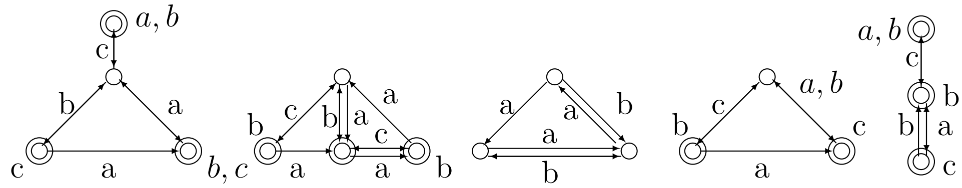
The known n-state automata with minimal reset word of length $(n-1)^2$ are presented by sequence of Černy [7] (here n=28):



by automata supposed by Černy, Piricka and Rosenauerova [8], by Kari [16] and Roman [23].



Our program has found five new following examples on the border $(n-1)^2$.



The corresponding reset words of minimal length are: *abcacabca*, *acbaaacba*, *baab*, *acba*, *bacb*. All considered algorithms have found the same reset word for every example. The size of the transition semigroup found by the package TESTAS is 148, 180, 24, 27 and 27 correspondingly.

There are no contradictory examples for the Černy conjecture in considered class of automata. Moreover, there are no new examples of automata with reset word of length $(n-1)^2$ for $n > 4$ and $q > 3$ in this class. And what is more, the examples with minimal length of reset word disappear even for values near the Černy bound $(n-1)^2$ with growth of the size of the automaton and of the size of the alphabet. The following table displays this noteworthy trend for the maximum of lengths of minimal reset words of length less than $(n-1)^2$. By $*$ are denoted here non-isomorphic automata having minimal reset words of length $(n-1)^2$ that do not belong to Černy sequence.

size of the automaton	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10
$(n-1)^2$	4	9	16	25	36	49	64	81
max length, 2 letters	$3*$	$8*$	15	$23*$	32	44	58	74
max length, 3 letters	$0**$	$8**$	$15*$	23	31	≤ 44	–	–
max length, 4 letters	0	8	15	22	30	–	–	–

The gap between $(n-1)^2$ and the maximum of considered length of the minimal reset word grows with n and q. This gap supports the following funny

Conjecture *The set of n-state DFA with minimal reset word of length not less than $(n-1)^2$ contains only the sequence of Černy and the eight automata mentioned above, three of size 3, three of size 4, one of size 5 and one of size 6.*

and also

Conjecture *Any AGW graph has coloring with minimal reset word of length less than $(n-1)^2$.*

References

1. R.L. Adler, L.W. Goodwyn, B. Weiss: *Equivalence of topological Markov shifts*, Israel J. of Math. 27(1977), 49–63.
2. R.L. Adler, B. Weiss: *Similarity of automorphisms of the torus*, Memoirs of the Amer. Math. Soc., Providence, RI, 98(1970).
3. M.P. B'eal, D. Perrin: *A quadratic algorithm for road coloring.* arXiv:0803.0726v2 [cs.DM].
4. G. Budzban, A. Mukherjea: *A semigroup approach to the Road Coloring Problem*, Probability on Algebraic Structures. Contemporary Mathematics, 261(2000), 195–207.

5. A. Carbone: *Cycles of relatively prime length and the road coloring problem*, Israel J. of Math., 123(2001), 303–316.
6. K. Culik II, J. Karhumaki, J. Kari: *A note on synchronized automata and Road Coloring Problem*, Developments in Language Theory (5th Int. Conf., Vienna, 2001), Lecture Notes in Computer Science, 2295(2002), 175–185.
7. J.Černy: Poznamka k homogenym eksperimentom s konechnymi automatami, Math.-Fyz. Čas., 14(1964) 208–215.
8. J. Černy, A. Piricka, B. Rosenauerova: On directable automata, Kybernetika 7(1971), 289–298.
9. D. Eppstein: Reset sequences for monotonic automata. SIAM J. Comput., 19(1990), 500–510.
10. P. Frankl: An extremal problem for two families of sets, Eur. J. Comb., 3(1982), 125–127.
11. J. Friedman: *On the road coloring problem*, Proc. of the Amer. Math. Soc. 110(1990), 1133–1135.
12. E. Gocka, W. Kirchherr, E. Schmeichel: *A note on the road-coloring conjecture.* Ars Combin. 49(1998), 265–270.
13. R. Hegde, K. Jain: *Min-Max theorem about the Road Coloring Conjecture* EuroComb 2005, DMTCS proc., AE, 2005, 279–284.
14. P.M. Higgins: The range order of a product of I-transformation from a finite full transformation semigroup, Semigroup Forum, 37(1988), 31–36.
15. N. Jonoska, S. Suen: *Monocyclic decomposition of graphs and the road coloring problem*, Congressum numerantium, 110(1995), 201–209.
16. J. Kari: *Synchronizing finite automata on Eulerian digraphs*, Springer, Lect. Notes in Comp. Sci., 2136(2001), 432–438.
17. A.A. Kljachko, I.K. Rystsov, M.A. Spivak: An extremely combinatorial problem connected with the bound on the length of a recurrent word in an automata. Kybernetika. 2(1987), 16–25.
18. D. Lind, B. Marcus: *An Introduction of Symbolic Dynamics and Coding*, Cambridge Univ. Press, 1995.
19. A. Mateescu, A. Salomaa: *Many-Valued Truth Functions, Černy's Conjecture and Road Coloring*, Bull. of Eur. Ass. for TCS, 68(1999), 134–148.
20. G.L. O'Brien: *The road coloring problem*, Isr. J. of Math., 39(1981), 145–154.
21. D. Perrin, M.P. Schützenberger: *Synchronizing prefix codes and automata, and the road coloring problem*, In Symbolic Dynamics and Appl., Contemp. Math., 135(1992), 295–318.
22. J.E. Pin: *On two combinatorial problems arising from automata theory*, Annals of Discrete Math., 17(1983), 535–548.
23. A. Roman: *Synchronization of finite automaton. Computations for different alphabet sizes*, Workshop on words and automata. S-Petersburg. 2006.
24. A.N. Trahtman: *Notable trends concerning the synchronization of graphs and automata*, CTW06, El. Notes in Discrete Math., 25(2006), 173-175.
25. M.V. Volkov: A private letter.
26. W.H. Wheeler: A note on Trakhtman's proof of the road coloring theorem. Submitted.

Dynamic Burrows-Wheeler Transform

Mikaël Salson[1⋆], Thierry Lecroq[1], Martine Léonard[1], and Laurent Mouchard[1,2]

[1] LITIS EA 4108, University of Rouen, 76821 Mont Saint Aignan Cedex, France
[2] Algorithm Design Group, Department of Computer Science, King's College London, Strand, London WC2R 2LS, England
Laurent.Mouchard@univ-rouen.fr

Abstract. The Burrows-Wheeler Transform is a building block for many text compression applications and self-index data structures. It reorders the letters of a text T to obtain a new text $bwt(T)$ which can be better compressed. This forward transform has been intensively studied over the years, but a major problem still remains: $bwt(T)$ has to be entirely recomputed whenever T is modified. In this article, we are considering standard edit operations (insertion, deletion, substitution of a letter or a factor) that are transforming a text T into T'. We are studying the impact of these edit operations on $bwt(T)$ and are presenting an algorithm that converts $bwt(T)$ into $bwt(T')$. Moreover, we show that we can use this algorithm for converting the suffix array of T into the suffix array of T'. Even if the theoretical worst-case time complexity is $O(|T|)$, the experiments we conducted indicate that it performs really well in practice.

1 Introduction

Data compression plays an important role in computer science. Its main goal is to reduce the normal consumption of data storage (one can easily store a large selection of books on a single USB key or CD). Nowadays, one of its main interests is to save network bandwith, enabling fast access to large distant resources, permitting the development of services such as Video On Demand or WebTV broadcasting over DSL [2]. While efficient image, video or sound compressions are traditionnally achieved using lossy algorithms, text compression only tolerates lossless algorithms, as no letter of the text should be omitted.

Some of the most popular lossless text compression tools, such as bzip, 7Z or winzip, are using a preprocessing engine that reorders the letters of the original text and eases the compression, paving the way for Run-Length Encoding, entropy encoding or Prediction by Partial Matching methods [4,3]. This preprocessor, the Burrows-Wheeler Transform [1], is a very interesting block-sorting algorithm: conceptually speaking, it is very close to the suffix array proposed in [17,12] and has been proved to be a particular case of the Gessel-Reutenauer transforms [5].
Due to its intrinsic structure and its similarity with the suffix array, it has been also used for advanced compressed index structures [8,9] that authorize approximate pattern matching, and therefore can be used by search engines.

The Burrows-Wheeler Transform of a text T of length n, $bwt(T)$, is often obtained from the fitting suffix array. Its construction is based on the construction of the suffix array, usually performed in $O(n)$-time [19]. Storing the intermediate suffix array is still one of the main technological bottlenecks, as it requires $\Omega(n \log n)$ bits, while storing $bwt(T)$ and T only require $O(n \log \sigma)$ bits, where σ is the size of the alphabet.

Even if this transform has been intensively studied over the years [10], one essential problem still remains: $bwt(T)$ has to be totally reconstructed as soon as the text T is

⋆ Funded by the French Ministry of Research – Grant 26962-2007

Mikaël Salson, Thierry Lecroq, Martine Léonard, Laurent Mouchard: Dynamic Burrows-Wheeler Transform, pp. 13–25.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

altered. Although some authors already addressed the issue of maintaining an index for a dynamic text [6,7,14], their answer cannot be fully applied to the Burrows-Wheeler Transform.

In this article, we are considering the usual edit operations (insertion, deletion, substitution of a letter or a factor) that are transforming T into T'. We are studying their impact on $bwt(T)$ and are presenting an algorithm for converting $bwt(T)$ into $bwt(T')$. Moreover, we show that we can use this algorithm for changing the suffix array of T into the suffix array of T'.

The article is organized as follows: in section 2 we introduce the Burrows-Wheeler Transform and all associated vocabulary and structures and state the formal problem we are facing. In section 3, we present a detailed explanation of the proposed algorithm when considering an insertion. We then extend the algorithm to handle the other edit operations, exhibiting their respective complexities. In section 4, we expose our results and compare them with the theoretical assumptions and finally in section 5 we conclude and draw perspectives.

2 Preliminaries

Let the text $T = T[0\,..\,n]$ be a word of length $n+1$ over a finite ordered alphabet Σ of size σ. Mimicking the suffix tree and suffix array structures, we are considering here that the rightmost letter of T is a sentinel letter \$. This letter has been added to the alphabet Σ and is smaller than any other letter of Σ.
A factor starting at position i and ending at position j is denoted by $T[i\,..\,j]$ and a single letter is denoted by $T[i]$ (or T_i to facilitate the reading). We add that when $i > j$, $T[i\,..\,j]$ is the empty word. The cyclic shift of order i of the text T is $T^{[i]} = T[i\,..\,n]T[0\,..\,i-1]$ for a given $0 \leq i \leq n$.

Remark 1. $T_i = T^{[(i+1) \bmod |T|]}[n]$ that will be simply denoted by $T_n^{[(i+1) \bmod |T|]}$ thereafter.

The Burrows-Wheeler Transform of T, denoted $bwt(T)$, is the text of length $n+1$ corresponding to the last column L of the conceptual matrix whose rows are the lexicographically sorted $T^{[i]}$ (see Fig. 1b).Note that F, the first column of this matrix, is sorted, so can be trivially deduced from L, and that in Fig. 1c, π is the fitting sort function.

$$T = \overset{0}{\mathrm{G}}\overset{1}{\mathrm{T}}\overset{2}{\mathrm{C}}\overset{3}{\mathrm{T}}\overset{4}{\$}$$

$$\begin{array}{c|ccccc|}
\cline{2-6}
T^{[0]} & G & T & C & T & \$ \\
T^{[1]} & T & C & T & \$ & G \\
T^{[2]} & C & T & \$ & G & T \\
T^{[3]} & T & \$ & G & T & C \\
T^{[4]} & \$ & G & T & C & T \\
\cline{2-6}
\end{array}
\quad \xrightarrow{\text{lexicographic sort}} \quad
\begin{array}{c|ccccc|}
\cline{2-6}
T^{[4]} & \$ & G & T & C & T \\
T^{[2]} & C & T & \$ & G & T \\
T^{[0]} & G & T & C & T & \$ \\
T^{[3]} & T & \$ & G & T & C \\
T^{[1]} & T & C & T & \$ & G \\
\cline{2-6}
\multicolumn{1}{c}{} & F & & & & \multicolumn{1}{c}{L}
\end{array}
\quad \xrightarrow{\text{fitting sort function}} \quad
\begin{array}{c|c}
i & \pi(i) \\
\hline
0 & 4 \\
1 & 2 \\
2 & 0 \\
3 & 3 \\
4 & 1
\end{array}$$

(a) Cyclic shifts (b) Sorted cyclic shifts (c) π

Figure 1. $bwt(\mathrm{GTCT\$}) = L =$ **TT\$CG**

Remark 2. We can observe that π corresponds to the suffix array of T, SA confirming the adjacency between L (letters) and SA (integers). Moreover, we simply have $L[i] = T[(SA[i]-1) \mod |T|]$, meaning we can deduce L from SA.

Combining Remarks 1 and 2, one can easily recover the original word T when considering both columns L and π. We know that: $T_0{=}T_4^{[1]}$, $T_1{=}T_4^{[2]}$, $T_2{=}T_4^{[3]}$, $T_3{=}T_4^{[4]}$ and $T_4{=}T_4^{[0]}$. The orders of the cyclic shifts are $(1,2,3,4,0)$ in the $\pi(i)$-column, that is $(4,1,3,0,2)$ in the i-column and finally (G, T, C, T, \$) in the L-column. We obtain T=GTCT\$.
Similarly, a right-to-left reconstruction of T will use sequence $(0,4,3,2,1)$, that is $(2,0,3,1,4)$ in the i-column and finally (\$, T, C, T, G) in the L-column. Reading this sequence from right to left, we obtain T=GTCT\$.

We clearly know how to progress in the $\pi(i)$-column, if we consider a value j in this column, its predecessor is $(j-1) \mod 5$. Starting with $j=0$, we obtain the sequence $(0,4,3,2,1)$. We have now to study how to progress in the i-column. Considering a value j in this column, the corresponding value in $\pi(i)$-column is obviously $\pi(j)$. Its predecessor in $\pi(i)$-column is $(\pi(j)-1 \mod 5)$ and finally the associated value back in the i-column is $\pi^{-1}((\pi(j)-1) \mod 5)$.

i	$\pi(i)$	$(\pi(i)-1) \mod 5$	$\pi^{-1}((\pi(i)-1) \mod 5)$
0	4	3	3
1	2	1	4
2	0	4	0
3	3	2	1
4	1	0	2

Using this formula, we obtain a permutation 0→3, 3→1, 1→4, 4→2, 2→0. We have to start with i such that $\pi(i)$ =0, that is $i=2$, corresponding to $(2,0,3,1,4)$ in the i-column and subsequently (\$, T, C, T, G) in the L-column. Reading this sequence from right to left, we obtain T=GTCT\$.

This function is of crucial importance, since it creates a link between two consecutive elements of L or more precisely between an element of L and its equivalent in F, as described in Fig. 2. It has been shown [8] that this function, which creates a table LF of size $n+1$, can be computed using only L and the functions $rank_c(U,i)$ that return the number of c in $U[0\,..\,i]$.

i	F	L	LF
0	\$	T	3
1	C	T	4
2	G	\$	0
3	T	C	1
4	T	G	2

$$T = \overset{0}{\mathrm{G}}\overset{1}{\mathrm{T}}\overset{2}{\mathrm{C}}\overset{3}{\mathrm{T}}\overset{4}{\$}$$

The second T in L ($rank_{\mathrm{T}}(L,1)$=2) is linked to the second T in F ($rank_{\mathrm{T}}(F,4)$=2). This specific T occurs at position 1 in the text. $LF[1]$=4 so $L[1]$=T is immediately preceded by $L[LF[1]]$=$L[4]$=G.

Figure 2. LF: Establishing a relation between L and F

Remark 3. Without the added sentinel letter \$, LF can not be necessarily determined from $bwt(T)$, *e.g.* T=AAA. It is clear that F and L would be both equal to AAA and that $rank_{\mathrm{A}}(L,i) = rank_{\mathrm{A}}(F,i)$ for all $0 \le i < 3$, annihilating all possible relation between consecutive elements of L.

To cut a long story short, LF provides a convenient way of navigating between cyclic shifts of order i and $i-1$ and will be intensively used in this article.

We already explained that L is conceptually very close to SA, with a simple forward transform from the former to the latter. It follows that most of the algorithms constructing L are using the existing $O(n)$-time (theoretical) algorithms that build SA [19] and are applying the forward transform afterwards. Storing SA is still the main technological bottleneck, as it requires $\Omega(n \log n)$ bits while L and T only require $O(n \log \sigma)$ bits. Such a requirement prevents large texts to be encoded, even if a recent promising result [15] authorizes large texts to be processed by computing the suffix array, a block at a time.

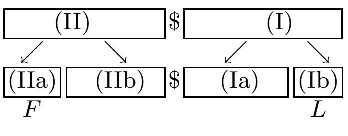
Nevertheless, L is a text that accepts no direct modification: a simple transformation of T into T' traditionally leads to the computation of its Burrows-Wheeler Transform, L', from scratch. Our goal is to study how L is affected when standard edit operations (insertion, deletion or substitution of a block of letters) are applied to T. Based on these observations, we are presenting an algorithm for transforming L into L' with only a very limited extra space and prove its correctness.

3 A Four-stage Algorithm for Updating L

We start by conducting a complete study on how an edit operation, transforming T into T', is impacting L (either directly or implicitly). To illustrate this study, we are considering the simple case consisting of the insertion of a single letter. Based on this study, we propose a four-stage algorithm for transforming L into L'. We are conducting a parallel study for F, which is required for the construction of L'. In order to do so, we are maintaining a two-column matrix gathering F and L. Each row contains the F and L values corresponding to a given cyclic shift (as described in Fig. 2). At the end of the process, L is equal to $bwt(T')$. Finally, we extend our approach to the insertion of a factor, and explain how we can consider substitutions and deletions.

In order to study the impact the insertion of a single letter has, we have first to recall that L' strongly depends on the ranking of all cyclic shifts of T'. We thus have to study how the insertion of a letter is modifying the cyclic shifts. Assume we are inserting a letter c at position i in T. Depending on the cyclic shift we are considering, we can formalize these four cases, remembering that T_n=\$, by:

$$T'^{[j]} = \begin{cases} T[j-1\,..\,n-1]\ \$\ T[0\,..\,i-1]\ c\ T[i\,..\,j-2] & \text{if } i+1 < j \le n+1 \quad \text{(Ia)} \\ T[i\,..\,n-1]\ \$\ T[0\,..\,i-1]\ c & \text{if } j = i+1 \quad \text{(Ib)} \\ c\ T[i\,..\,n-1]\ \$\ T[0\,..\,i-1] & \text{if } j = i \quad \text{(IIa)} \\ T[j\,..\,i-1] c\ T[i\,..\,n-1]\ \$\ T[0\,..\,j-1] & \text{if } 0 \le j < i \quad \text{(IIb)} \end{cases}$$



c appears: (I) right to \$, (II) left to \$.
That means:
c appears: (Ia) between \$ and L, (Ib) in L.
c appears: (IIa) in F, (IIb) between F and \$.

Figure 3. All possible locations of c in $T'^{[j]}$ after the insertion

3.1 Cyclic Shifts of Order $j > i$ (I)

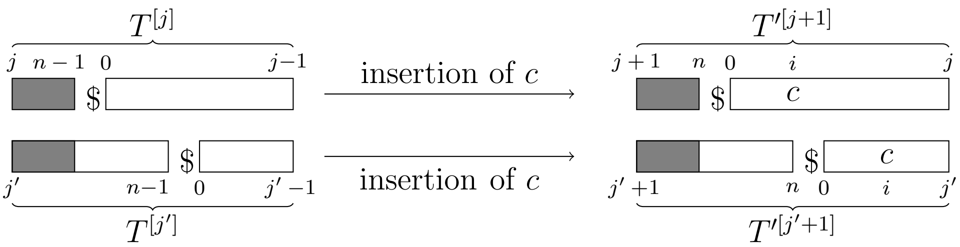
In this section, we are considering all cyclic shifts associated with positions in T that are strictly greater than i. We show that the two stages (Ia) and (Ib) are not modifying the respective ranking of the corresponding cyclic shifts.

From Fig. 3 (Ia), $T'^{[j+1]} = T[j\,..\,n-1]\$T[0\,..\,i-1]cT[i\,..\,j-1]$, $\forall j \geq i$ meaning that $T'^{[j+1]}$ and $T^{[j]}$ are sharing a common prefix $T[j\,..\,n-1]\$T[0\,..\,i-1]$.

Lemma 4. *Inserting a letter c at position i in T has no effect on the respective ranking of cylic shifts whose orders are strictly greater than i. That is, for all $j \geq i$ and $j' \geq i$, we have $T^{[j]} < T^{[j']} \iff T'^{[j+1]} < T'^{[j'+1]}$.*

Proof. In order to prove this lemma, we have to prove that the relative lexicographical rank of two cyclic shifts, of orders strictly greater than i is the same before and after the insertion.

Assume without loss of generality that $j > j'$ and $T^{[j]} < T^{[j']}$.
We know that for every $k < |T|$, $T^{[j]}[0\,..\,k] \leq T^{[j']}[0\,..\,k]$. The prefix of $T^{[j]}$ ending before the sentinel letter \$ is of length $n-j < |T|$, and therefore $T^{[j]}[0\,..\,n-j-1] \leq T^{[j']}[0\,..\,n-j-1]$. That is, $T[j\,..\,n-1] \leq T[j'\,..\,j'+n-j-1]$ (grey rectangles below). Moreover \$, the smallest letter of Σ, occurs only once in T. The fact that $T[j+n-j]$ is equal to \$ induces $T[j'+n-j] \neq \$$, and is therefore strictly greater than \$. It follows that $T^{[j]}[0\,..\,n-j] < T^{[j']}[0\,..\,n-j]$.



Since $T'[j+1\,..\,n]\$ = T[j\,..\,n-1]\$$ and $T'[j'+1\,..\,n+j'-j+1] = T[j'\,..\,n+j'-j]$, we have $T'[j+1\,..\,n]\$ < T'[j'+1\,..\,n+j'-j+1]$. So $T'[j+1\,..\,n]\$u < T'[j'+1\,..\,n+j'-j+1]v$, for all texts u, v over Σ. Finally, $T^{[j]} < T^{[j']} {:} T'^{[j+1]} < T'^{[j'+1]}$.

The proof of $T'^{[j+1]} < T'^{[j'+1]} {:} T^{[j]} < T^{[j']}$ is done in a similar way.

Remark 5. This lemma can be generalized to the insertion of a factor of length k by considering $T'^{[j+k]} < T'^{[j'+k]}$ instead of $T'^{[j+1]} < T'^{[j'+1]}$.

Cyclic Shifts of Order $j > i+1$: (Ia) c between \$ and L It follows, from Lemma 4, that the ranking of all cyclic shifts $T'^{[j+1]}$ is identical to the ranking of all cyclic shifts $T^{[j]}$. In the rows corresponding to $T'^{[j]}$, F and L are unchanged.

Cyclic Shift of Order $i+1$: (Ib) c in $L \rightarrow$ Modification of L The respective ranking of this cyclic shift with respect to the cyclic shifts of greater order is preserved. Since c is inserted at position i, it follows that $T'^{[i+1]} = T^{[i]}c$. These two cyclic shifts are sharing a common prefix $T^{[i]}$. In the row corresponding to $T'^{[i+1]}$, F is unchanged while L, which was equal to T_{i-1}, is now equal to c.
We find the position of $T'^{[i+1]}$ by using a subsampling of π (see [9,16]) and computing k such that $\pi(k)=i$.

Insertion of **G** at position i=**2** in T

$$T = \overset{0}{\mathrm{C}}\overset{1}{\mathrm{T}}\overset{2}{\mathrm{C}}\overset{3}{\mathrm{T}}\overset{4}{\mathrm{G}}\overset{5}{\mathrm{C}}\overset{6}{\$} \rightarrow T' = \overset{0}{\mathrm{C}}\overset{1}{\mathrm{T}}\overset{2}{\mathbf{G}}\overset{3}{\mathrm{C}}\overset{4}{\mathrm{T}}\overset{5}{\mathrm{G}}\overset{6}{\mathrm{C}}\overset{7}{\$}$$

(Ia): no modification.

(Ib): $T^{[i]}$ is at position k=3 ($\pi(3)$=**2**), $L[3] \leftarrow$**G**.

After stage (Ib), we have: one G in F and two Gs in L, two Ts in F and one T in L.

π	F	L		F	L
6	\$	C		\$	C
5	C	G		C	G
0	C	\$		C	\$
i= **2**	C	T	$\xrightarrow{\text{(Ib)}}$	C	**G**
4	G	T		G	T
1	T	C		T	C
3	T	C		T	C

3.2 Cyclic Shifts of Order $j \leq i$

Cyclic Shift of Order i: (IIa) c in F $\rightarrow$ Insertion of a new row After considering the cyclic shift $T'^{[i+1]}$ that ends with the added letter c, we now have to consider the brand new cyclic shift that starts with the added c, that is $T'^{[i]} = cT^{[i]} = cT[i\,..\,n-1]\$T[0\,..\,i-1]$ which ends with T_{i-1}. Since $T'^{[i+1]}$ is located at position k, $T'^{[i]}$ has to be inserted in the table at position $LF[k]$ (derived from the function $rank_c(L,k)$).

Insertion of **G** at position i=**2** in T

T=CTCTGC\$ (positions 0 1 2 3 4 5 6) $\rightarrow$ T'=CT**G**CTGC\$ (positions 0 1 2 3 4 5 6 7)

(IIa): $T'^{[i]}$ is inserted in the table at position $LF[k]$.

For this inserted row F=c=**G** and L=T_{i-1}=**T**.

$T'^{[i+1]}$ finishes with a G which is the second G in L.

$T'^{[i]}$ begins with this G which has to be the second G in F.

After stage (IIa), we have: two Gs in F and two Gs in L, two Ts in F and two Ts in L.

F	L
\$	C
C	G
C	\$
C	**G**
G	T
T	C
T	C

$\xrightarrow{\text{(IIa)}}$

F	L
\$	C
C	G
C	\$
C	G
G	T
G	**T**
T	C
T	C

Cyclic Shifts of Order $j < i$: (IIb) c between F and \$ $\rightarrow$ Reordering So far, the L-value of one row has been updated (Ib) and one new row has been inserted (IIa). However, cyclic shifts $T'^{[j]}$, for any $j < i$, may have a different lexicographical rank than $T^{[j]}$ (e.g. AAG\$ < AG\$A but ATAG\$ > AG\$AT). Consequently, some rows corresponding to those cyclic shifts may be moved.

To know which rows have to move, we compare the position of $T^{[j]}$ with the computed position of $T'^{[j]}$, from $j = i-1$ downto 0, until these two positions are equal. The position of $T^{[j]}$ is obtained from $T^{[j+1]}$ and the LF-table we updated while considering $T^{[j+1]}$ (UpdateLF in the algorithm). The position of $T'^{[j]}$ is obtained from $T'^{[j+1]}$ and the current LF-table.
When these two positions are different, the row corresponding to $T^{[j]}$ is moved to the computed position of $T'^{[j]}$ (MoveRow in the algorithm).

We give the pseudocode of the reordering step. The function *index* returns the position of a cyclic shift in the matrix.

```
Reorder(L, i)
  j ← index(T^[i−1])           ▷ Gives the position of T^[i−1]
  j′ ← LF[index(T′^[i])]       ▷ Gives the computed position of T′^[i−1]
  while j ≠ j′ do
        new_j ← LF[j]
        MoveRow(j, j′)
        UpdateLF(j′, new_j)
        j ← new_j
        j′ ← LF[j′]
```

We now prove that the algorithm Reorder is correct: it ends as soon as all the cyclic shifts of T' are sorted. In the following lemma, we denote by C a succinct representation of F. Since the letters of the text are lexicographically sorted in F, we only need to store the number of times each letter appears in the text. Thus, $C[c]$ is defined as the number of letters in the text strictly lower than c, e.g. when F = \$AAACCGGT, $C[\$] = 0$ and $C[\text{G}] = 6$.

Lemma 6. *$\forall j < i$, $\forall j' > j$, $T'^{[j]} < T'^{[j']} \Leftarrow: index(T'^{[j]}) < index(T'^{[j']})$, after the iteration considering $T^{[j]}$, in* Reorder.

Proof. We prove the lemma recursively for any $j \leq i+1$.

From the previous lemma, $\forall j' \geq i+1$ we have $T'^{[i+1]} < T'^{[j']} \Leftarrow: T^{[i]} < T^{[j'-1]}$. Obviously, the property we want to prove is true for any j, on the text T and the original BWT. Thus $T'^{[i+1]} < T'^{[j']} \Leftarrow: index(T^{[i]}) < index(T^{[j'-1]})$. Neither $T'^{[i+1]}$ nor $T'^{[j']}$ have been moved in the algorithm. Thus, $index(T'^{[i+1]}) < index(T'^{[j']}) \Leftarrow: index(T^{[i]}) < index(T^{[j'-1]}) \Leftarrow: T'^{[i+1]} < T'^{[j']}$.

We have shown that the lemma is true for $j = i+1$, now let us prove it recursively for $j-1$.

By definition, $T_0'^{[j-1]} = T_{n+1}'^{[j]}$, let $r = rank_{T_{n+1}'^{[j]}}(L, index(T'^{[j]}))$. The index of $T'^{[j-1]}$ is computed using LF with the following formula:

$index(T'^{[j-1]}) = C[T_0'^{[j-1]}] + r - 1$. We distinguish two different cases:

- if the first letter of $T'^{[j-1]}$ is different from the first one of $T'^{[j']}$, then $C[T_0'^{[j-1]}] \neq C[T_0'^{[j']}]$. Without loss of generality, consider $T_0'^{[j-1]} < T_0'^{[j']}$. By definition, $r \leq C[T_0'^{[j']}] - C[T_0'^{[j-1]}]$. Thus $C[T_0'^{[j-1]}] + r - 1 \leq C[T_0'^{[j']}] - 1$. However, the *rank* computed for the index of $T'^{[j']}$ is strictly positive. Finally $T_0'^{[j-1]} < T_0'^{[j']}$:$index(T'^{[j-1]}) < index(T'^{[j']})$.
- otherwise, both letters are equal. Then, we can write $T'^{[j-1]} < T'^{[j']} \Leftarrow: T'^{[j-1]}[1\,..\,n+1] < T'^{[j']}[1\,..\,n+1] \Leftarrow: T'^{[j-1]}[1\,..\,n+1]T_0'^{[j-1]} < T'^{[j']}[1\,..\,n+1]T_0'^{[j']} \Leftarrow: T'^{[j]} < T'^{[j'+1]}$. We know that the lemma is true for j, thus we have $T'^{[j]} < T'^{[j'+1]} \Leftarrow: index(T'^{[j]}) < index(T'^{[j'+1]})$.
  Let $k = index(T'^{[j]})$, $k' = index(T'^{[j'+1]})$, $r' = rank_{T_{n-1}'^{[j'+1]}}(L, k')$ and $c = T_0'^{[j-1]} = T_0'^{[j']}$.
  $$\begin{aligned} index(T'^{[j-1]}) &= C[c] + rank_c(L,k) - 1 \\ index(T'^{[j']}) &= C[c] + rank_c(L,k') - 1 \end{aligned}$$
  We know that $T_{n+1}'^{[j]} = L_k = c$, $T_{n+1}'^{[j'+1]} = L_{k'} = c$ and $k' > k$. So $rank_c(L,k') > rank_c(L,k)$ and eventually $index(T'^{[j-1]}) < index(T'^{[j']})$.
  Finally, $T'^{[j-1]} < T'^{[j']}$:$index(T'^{[j-1]}) < index(T'^{[j']})$. We can prove $T'^{[j-1]} < T'^{[j']} \Longleftarrow index(T'^{[j-1]}) < index(T'^{[j']})$ in a similar way.

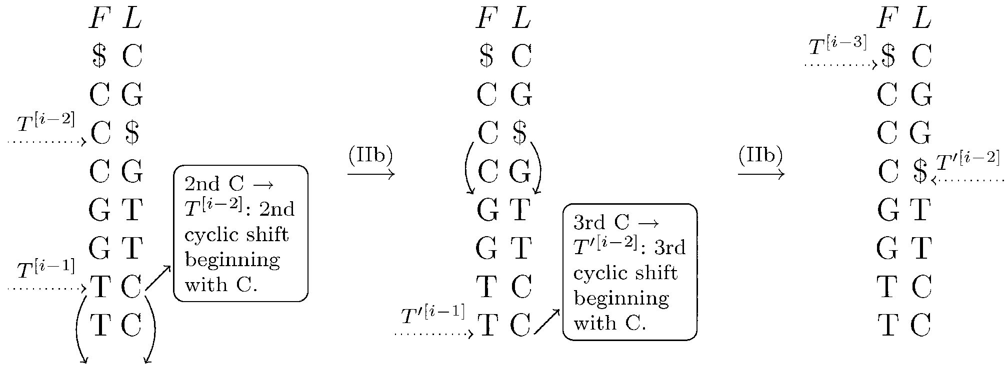
Thus, if the property is true for j, it is also true for $j-1$. Finally, when the algorithm finishes (with $j = 0$), we have $\forall j, j', \quad T'^{[j]} < T'^{[j']} \Leftarrow: index(T'^{[j]}) < index(T'^{[j']})$. In other words, at the end of the algorithm, the cyclic shifts are ordered.

We now have to prove that stopping the algorithm when the computed position and the initial one are identical is sufficient, all cyclic shifts being ordered.

Lemma 7. *$index(T^{[k]}) = index(T'^{[k]})$:$index(T^{[j]}) = index(T'^{[j]})$, for $j < k < i$.*

Proof. Given $index(T^{[k]})$,
$$\begin{aligned} index(T^{[k-1]}) &= C[T_n^{[k]}] + rank_{T_n^{[k]}}(L, index(T^{[k]})) \\ &= C[T_{n+1}'^{[k]}] + rank_{T_{n+1}'^{[k]}}(L, index(T'^{[k]})) = index(T'^{[k-1]}) \end{aligned}$$
Therefore, $index(T^{[k]}) = index(T'^{[k]})$:$index(T^{[k-1]}) = index(T'^{[k-1]})$.
By induction, we prove the property for each $j < k$.

Consider a cyclic shift $T^{[j]}$ and k the number of times $T_n^{[j]}$ appears in L from the beginning to the position of $T^{[j]}$. The LF-value for the cyclic shift $T^{[j]}$ is the position corresponding to $T^{[j-1]}$ in L which is the k-th cyclic shift beginning with a $T_n^{[j]}$.



At the position of $T'^{[i-2]}$, we have the first \$ in L, and at the position of $T^{[i-3]}$, we have the first \$ in F. Therefore, we do not need to move a cyclic shift anymore. In fact, we reach the leftmost position of the text, preventing us from considering further move.

Finally, $L = bwt(T')$.

3.3 Insertion of a Factor rather than a Single Letter

We can generalize our approach to handle the insertion of a factor S at position i in T. Consider $T' = T[0..i-1]S[0..m-1]T[i..n]$ with $m > 1$.

The four stages can be extended as follows:

(Ia) Cyclic shifts $T'^{[j]}$ with $j > i+m$: unchanged.
(Ib) Cyclic shift $T'^{[i+m]}$: modification $L{=}S_{m-1}$ instead of T_{i-1}.
(IIa) Cyclic shifts $T'^{[j]}$ from $j{=}i+m-1$ downto $i+1$:
  insertion $F{=}S_{j-i}$ and $L{=}S_{j-i-1}$.
  $T'^{[i]}$: **insertion** $F{=}S_0$ and $L{=}T_{i-1}$.
(IIb) Cyclic shifts $T'^{[j]}$ with $j < i$: as presented in algorithm on page 18.

However a problem arises: we delete T_{i-1} from L during stage (Ib), and reintroduce it after all the other insertions at the end of stage (IIa). During this stage, all $rank_{T_{i-1}}$ values that have been computed before the final insertion may be wrong. These values have to be computed only if a S_j, $j > 0$, is such that $S_j = T_{i-1}$.

A simple solution consists in not relying on $rank_{T_{i-1}}$ and, depending on the location we are considering and the location of the original T_{i-1}, adding 1 to the obtained value.

More precisely, if we are computing $LF(\ell)$ such that $L[\ell] = T_{i-1}$ and $\ell > \pi^{-1}(i)$, then we must add one to the result of $LF(i)$ (see Fig. 4).

3.4 Deletion of a Factor

Consider a deletion of m consecutive letters in T, starting at position i. The resulting text is $T' = T[0..i-1]T[i+m..n]$. The four stages can be modified as follows:

(Ia) Cyclic shifts $T'^{[j]}$ with $j > i+m$: unchanged.
(Ib) Cyclic shift $T^{[i+m]}$: modification $L{=}T_{i-1}$ instead of T_{i+m-1}.
(IIa) Cyclic shifts $T^{[j]}$ from $j{=}i+m-1$ downto i:
  deletion of the corresponding row.
  We still have to pay attention to $rank_{T_{i-1}}$: during the deletion of cyclic shifts, T_{i-1} appears twice in L. Therefore, we may have to subtract one from the value returned by $rank_{T_{i-1}}$.
(IIb) Cyclic shifts $T'^{[j]}$ with $j < i$: as presented in algorithm page 18.

π	F	L		F	L	i
6	\$	C		\$	C	0
5	C	G		C	G	1
0	C	\$		C	\$	2
2	C	T		C	T	3
4	G	T		G	T	4
1	T	**C**	$\xrightarrow{\text{(Ib)}}$	T	**G**	5
3	T	C		T	C	6

Assume we are having an insertion at position 1 which causes such a modification in L.
During step (Ib) a disequilibrium is introduced between L and F (two G in L, one G in F and two C in L, three C in F).

Computing LF at position 6 gives position 2 (ie. the position of the second C in F). However it should be position 3: $\pi(6) = 3$ and $\pi(3) = \pi(6) - 1 = 2$. To correct this, we have to remember, until we insert back the original C, that at position $p = 5$ we had a C.
Using the solution we proposed, since $L[6] = \text{C}$ and $6 > \pi^{-1}(1) = 5$, we must add one to the original LFvalue obtained and finally the value is correct (that is 3).

Figure 4. Example of the problem induced by the insertion of a factor.

3.5 Substitution of a Factor

Consider the substitution of $T[i\,..\,i+m-1]$ by $S[0\,..\,m-1]$: that is T'=$T[0\,..\,i-1]S[0\,..\,m-1]T[i+m\,..\,n]$.

(Ia) Cyclic shifts $T'^{[j]}$ with $j > i+m$: unchanged.
(Ib) Cyclic shift $T'^{[i+m]}$: modification L=S_{m-1} instead of T_{i+m-1}.
(IIa) Cyclic shifts $T'^{[j]}$ from j=$i+m-1$ downto $i+1$:
substitution F=S_{j-i} and L=S_{j-i-1}
move this row to the appropriate position.
$T'^{[i]}$: modification F=S_0.
(IIb) Cyclic shifts $T'^{[j]}$ with $j < i$: as presented in algorithm on page 18.

3.6 Complexity

After the three first stages, a modification and an insertion have modified the two columns. The fourth stage, that consists in finding the new ranking of all extended cyclic shifts of order less than i, is the greediest part of the algorithm. The worst-case scenario occurs when the new ranking is obtained after each cyclic shift has been considered (e.g. $\text{A}^m\$ \rightarrow \text{A}^m\text{C}\$$). It follows that the worst-time complexity depends on the $O(n)$ iterations presented in the algorithm on page 18.

Since we are dealing with insertions and deletions, we cannot use constant-time static structures in the functions MoveRow and UpdateLF. Very recent dynamic data structures can handle insertions and deletions while allowing to perform $rank_c$, insertions and deletions in logarithmic time [16,13], leading to an overall practical complexity bounded by $O(n \log n \log \sigma)$.

These structures [16,13] can store any text in $nH_0 + o(n \log \sigma)$ bits. However Mäkinen and Navarro proved [16] that storing the BWT with such structures needs only $nH_k + o(n \log \sigma)$ bits, where H_k corresponds to the k-th order entropy of the text. C is represented in little space using $O(\sigma \log n)$ bits. Our algorithm by itself needs only constant space consisting in few variables which store values that have been replaced.

4 Experiments and Results

In the previous section, we presented a four-stage algorithm for updating the Burrows-Wheeler Transform of a modified text. We conducted experiments on real-life texts as follows: we downloaded four texts from the Pizza&Chili corpus[1] on March, 15th 2008. We added two other type of texts: a random text drawn on an alphabet of size 100 and a Fibonacci word. These texts are of various types (length, content, entropy and alphabet size). For each category, we extracted randomly 10 texts of length 100, 250 and 500 KB, and 1 MB. For each text T, the letter at a random position i was replaced by another letter c drawn from T, resulting in T'. Because of the closeness between the Burrows-Wheeler Transform and the suffix array, we generated, for each sample, the two suffix arrays, one for T and one for T'. We measured the number of differences between these two suffix arrays and repeated this operation 100 times to compute an average value. We used substitution, instead of insertion, in these tests because the number of modifications is much easier to compute: with an insertion at position i, the suffix beginning at position $j > i$ in T begins at position $j + 1$ in T'. Thus, all values greater than i in the original suffix array are incremented by one in the modified suffix array. Note that the impact an insertion or a deletion has on the lexicographical order of suffixes (or cyclic shifts) is not different from the impact of a substitution.

The results are presented in Table 1.

	Entropy H_0	100 KB	250 KB	500 KB	1 MB	Ratio 1 MB:100 KB
DNA	1.982	10.12	9.52	10.26	10.91	1.08
English	4.53	7.75	7.94	9.03	10.31	1.33
Fibo	0.96	25,414.13	63,527.09	119,780.37	261,910.49	10.31
Random	6.60	3.89	4.03	4.21	4.36	1.12
Source	5.54	92.88	55.76	118.54	72.22	0.77
XML	5.23	26.43	28.84	34.8	44.08	1.67

Table 1. Number of modifications for a random substitution of a single letter.

These results are encouraging since multiplying the size of the text by 10 does not increase by the same factor the number of differences (apart from Fibonacci). Moreover, the number of modifications is closer to $\log(n)$ rather than n. We would like to conduct an in-depth study of these experiments to examine the impact of the size of the alphabet, the entropy and other possible factors that are impacting the update.

Using dynamic structures implemented by Gerlach [11], we compare the time needed for running our update algorithm to a total reconstruction of the Burrows-Wheeler Transform (with both static and dynamic structures). The computation of the Burrows-Wheeler Transform using static structures is due to Maniscalco and Puglisi [18] and is one of the most time-efficient.

Due to technical restrictions of the implementation of the dynamic structures, we run the tests on different kinds of texts: DNA, random text and Fibonacci word. We are considering two types of updates with our algorithm: factor insertion (of length 500) and 500 insertions of a single letter.

[1] http://pizzachili.dcc.uchile.cl/texts.html

The tests are conducted on a machine under Linux 2.6.24 and the programs were compiled using `gcc` 4.2. The results are presented in Fig. 5. Note that in the graphs, y-axis uses a logarithmic scale.

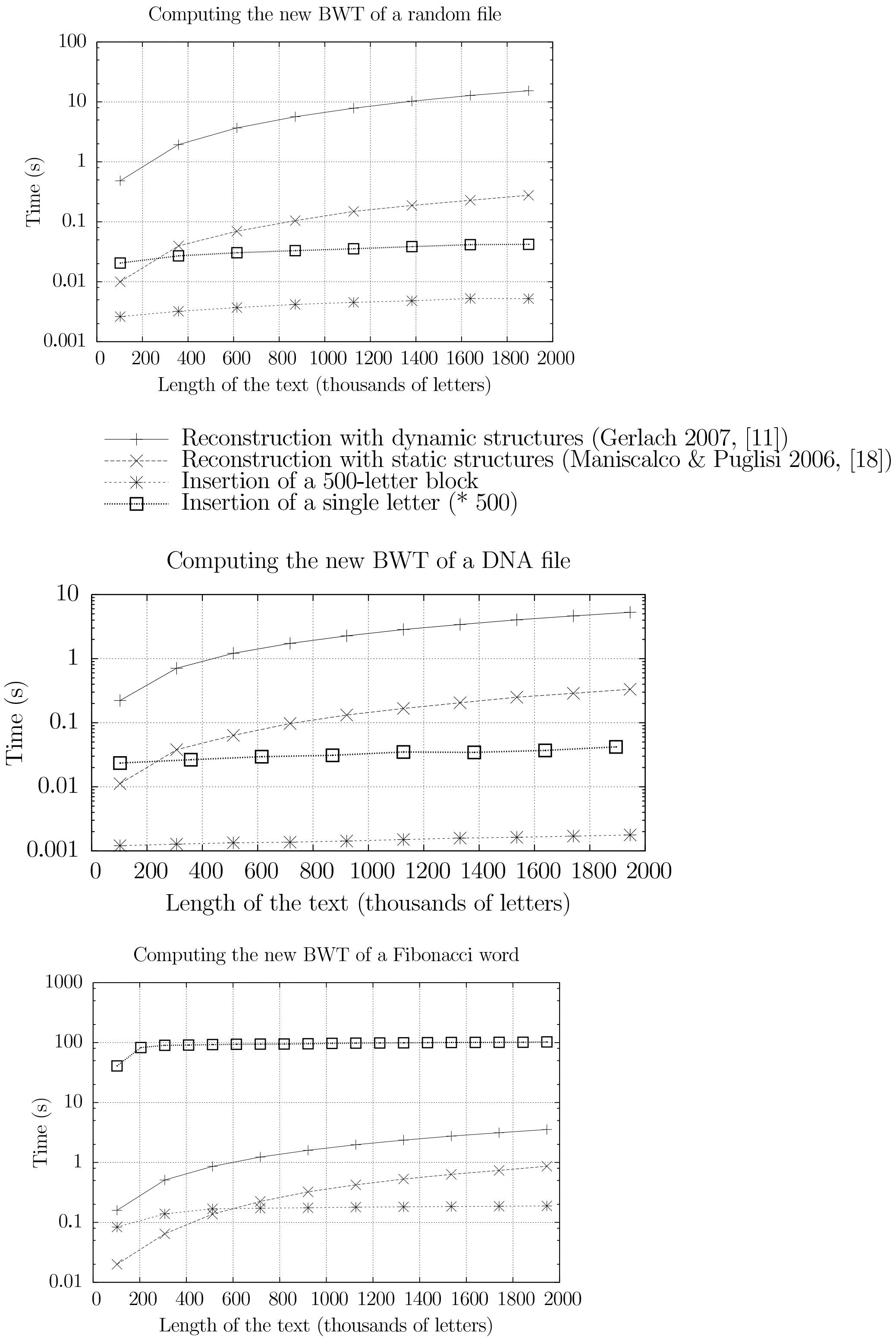


Figure 5. Time for updating and reconstructing the Burrows-Wheeler Transform.

We note that the insertion of a factor outperforms Maniscalco and Puglisi's very efficient algorithm. For the Fibonacci word, as soon as the text is long enough, our algorithm is still more efficient for the insertion of a factor although the number of iterations in step (IIb) is very high (see Table 1). However, due to the very particular structure of a Fibonacci word, one insertion of a single letter is as costly as the insertion of a 500-letter block, which explains the upper curve for Fibonacci. Note also that the reconstruction using dynamic structures is about 10 times slower than the static reconstruction, and thus our implementation may suffer from the slowdown induced by the dynamic structures.

5 Conclusions and Perspectives

We proposed an algorithm of theoretical worst-case time complexity $O(|T|)$ that modifies the Burrows-Wheeler Transform of a text T whenever standard edit operations are modifying T. The correctness of this algorithm has been proved and its efficiency in practice has been demonstrated: we selected various texts, edited randomly these texts and, with respect to the results, we confirmed that we are far from the worst-case bound. Yet, determining precisely the average-case bound of our algorithm still needs some extra work.

Moreover, this algorithm can be adapted for updating a suffix array. From a suffix array, we deduce the corresponding L, update it and retrieve the updated suffix array. Here is a pseudocode for retrieving the suffix array SA from L:

RetrieveSA(L)

$j \leftarrow index(L, T^{[n]})$
$i \leftarrow 0$
repeat $SA[j] \leftarrow i$
    $j \leftarrow LF[j]$
    $i \leftarrow (i-1) \mod (n+1)$
until $i = 0$

From the practical viewpoint, the dynamic structures that need to be maintained during the conversions are slowing down the process, losing the fight against "from scratch" SA constructions. Nevertheless, as far as we know, this is the first method for updating a suffix array rather than reconstructing it from scratch.

Our plan is now to adapt our strategy for updating directly a suffix array without using intermediate Burrows-Wheeler Transforms.

The algorithm we developed is also of interest for compressed indexes. Structures that are based on the Burrows-Wheeler Transform, such as FM-index, can be maintained in a way that is very similar to the one we developed for the transform, paving the way for the first fully-dynamic compressed full-text index.

References

1. M. Burrows and D. J. Wheeler: *A block-sorting lossless data compression algorithm.*, Tech. Rep. 124, DEC, Palo Alto, California, 1994.
2. C. C. Chen, C. Lee, and C. H. Ke: *Compression-based broadcast strategies in wireless information systems*, in Proc. of Advanced Information Networking and Applications (AINA), 2003, pp. 13–18.
3. J. G. Cleary, W. J. Teahan, and I. Witten: *Unbounded length contexts for PPM.* Comput. J., 40(2/3) 1997, pp. 67–76.

4. J. G. Cleary and I. Witten: *Data compression using adaptive coding and partial string matching.* IEEE Trans. Commun., 32(4) 1984, pp. 396–402.
5. M. Crochemore, J. Désarménien, and D. Perrin: *A note on the Burrows-Wheeler transformation.* Theor. Comput. Sci., 332(1-3) 2005, pp. 567–572.
6. P. Ferragina and R. Grossi: *Fast incremental text editing*, in Proc. of Symposium on Discrete Algorithms (SODA), 1995, pp. 531–540.
7. P. Ferragina and R. Grossi: *Optimal on-line search and sublinear time update in string matching*, in Proc. of Foundations of Computer Science (FOCS), 1995, pp. 604–612.
8. P. Ferragina and G. Manzini: *Opportunistic data structures with applications*, in Proc. of Foundations of Computer Science (FOCS), 2000, pp. 390–398.
9. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro: *Compressed representation of sequences and full-text indexes.* ACM Trans. Alg., 3 2007, p. article 20.
10. P. Ferragina, G. Manzini, and S. Muthukrishnan: *The Burrows-Wheeler Transform (special issue).* Theor. Comput. Sci., 387(3) 2007, pp. 197–360.
11. W. Gerlach: *Dynamic FM-Index for a collection of texts with application to space-efficient construction of the compressed suffix array*, Master's thesis, Universität Bielefeld, Germany, 2007.
12. G. H. Gonnet, R. A. Baeza-Yates, and T. Snider: *New indices for text: Pat trees and pat arrays.* Information Retrieval: Data Structures & Algorithms, 1992, pp. 66–82.
13. R. González and G. Navarro: *Improved dynamic rank-select entropy-bound structures*, in Proc. of the Latin American Theoretical Informatics (LATIN), vol. 4957 of Lecture Notes in Computer Science, 2008, pp. 374–386.
14. W. K. Hon, T. W. Lam, K. Sadakane, W. K. Sung, and S. M. Yiu: *Compressed index for dynamic text*, in Proc. of Data Compression Conference (DCC), 2004, pp. 102–111.
15. J. Kärkkäinen: *Fast BWT in small space by blockwise suffix sorting.* Theor. Comput. Sci., 387(3) 2007, pp. 249–257.
16. V. Mäkinen and G. Navarro: *Dynamic entropy-compressed sequences and full-text indexes.* ACM Trans. Alg., 2008, p. , To appear.
17. U. Manber and G. Myers: *Suffix arrays: a new method for on-line string searches*, in Proc. of Symposium on Discrete Algorithms (SODA), 1990, pp. 319–327.
18. M. A. Maniscalco and S. J. Puglisi: *Faster lightweight suffix array construction*, in Proc. of International Workshop On Combinatorial Algorithms (IWOCA), 2006, pp. 16–29.
19. S. J. Puglisi, W. F. Smyth, and A. Turpin: *A taxonomy of suffix array construction algorithms.* ACM Comp. Surv., 39(2) 2007, pp. 1–31.

Lossless Image Compression by Block Matching on Practical Massively Parallel Architectures

Luigi Cinque and Sergio De Agostino

Computer Science Department
Sapienza University
Via Salaria 113, 00198 Roma, Italy
{cinque, deagostino}@di.uniroma1.it

Abstract. Work-optimal $O(\log M \log n)$ time implementations of lossless image compression by block matching are shown on the PRAM EREW, where n is the size of the image and M is the maximum size of the match, which can be implemented on practical architectures such as meshes of trees, pyramids and multigrids. The work-optimal implementations on pyramids and multigrids are possible under some realistic assumptions. Decompression on these architectures is also possible with the same parallel computational complexity.

Keywords: lossless compression, sliding dictionary, bi-level image, parallel architecture.

1 Introduction

Storer suggested that fast encoders are possible for two-dimensional lossless compression by showing a square greedy matching heuristic for bi-level images, which can be implemented by a simple hashing scheme [8]. Rectangle matching improves the compression performance, but it is slower since it requires $O(M \log M)$ time for a single match, where M is the size of the match [9]. Therefore, the sequential time to compress an image of size n by rectangle matching is $\Omega(n \log M)$.

The technique is a two-dimensional extension of LZ1 compression [7]. Simple and practical heuristics exist to implement LZ1 compression by means of hashing techniques [1], [10], [11]. The hashing technique used for the two-dimensional extension is even simpler.

Among the different ways of reading an image, we assume that the rectangle matching compression heuristic is scanning an m x m' image row by row (*raster* scan). A 64K table with one position for each possible 4x4 subarray is the only data structure used. All-zero and all-one rectangles are handled differently. The encoding scheme is to precede each item with a flag field indicating whether there is a monochromatic rectangle, a match, or raw data. When there is a match, the 4x4 subarray in the current position is hashed to yield a pointer to a copy. This pointer is used for the current rectangle greedy match and then replaced in the hash table by a pointer to the current position. As mentioned above, the procedure for computing the largest rectangle match with left upper corners in positions (i, j) and (k, h) takes $O(M \log M)$ time, where M is the size of the match. Obviously, this procedure can be used for computing the largest monochromatic rectangle in a given position (i, j) as well. If the 4 x 4 subarray in position (i, j) is monochromatic, then we compute the largest monochromatic rectangle in that position. Otherwise, we compute the largest rectangle match in the position provided by the hash table and update the table with the current position. If the subarray is not hashed to a pointer, then it is left

Luigi Cinque, Sergio De Agostino: Lossless Image Compression by Block Matching on Practical Massively Parallel Architectures, pp. 26–34.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

uncompressed and added to the hash table with its current position. The positions covered by matches are skipped in the linear scan of the image and the sequential time to compress an image of size n by rectangle matching is $\Omega(n \log M)$. We want to point out that besides the proper matches we call a match every rectangle of the parsing of the image produced by the heuristic. We also call pointer the encoding of a match.

The analysis of the running time of these algorithms involve a so called *waste factor*, defined as the average number of matches covering the same pixel. In [9], it is conjectured that the waste factor is less than 2 on realistic image data. Therefore, the square greedy matching heuristic takes linear time while the rectangle greedy matching heuristic takes $\mathrm{O}(n \log M)$ time. On the other hand, the decoding algorithms are both linear.

Work-optimal parallel coding algorithms for lossless image compression by block matching were shown on the PRAM-EREW [3], [4] and the mesh of trees [5], which is a hybrid network architecture based on arrays and trees [6], requiring $\mathrm{O}(\log M \log n)$ time and $\mathrm{O}(n/\log n)$ processors. The design of a parallel decoder was left as an open problem as well as the implementation on even simpler architectures as pyramids and multigrids. By slightly modifying the encoder, new parallel coding and decoding algorithms were shown in [2] still requiring $\mathrm{O}(\log M \log n)$ time and $\mathrm{O}(n/\log n)$ processors on the PRAM-EREW. On the mesh of trees though the decoder required $\mathrm{O}(\log^2 n)$ time. In this paper, we show how to implement $\mathrm{O}(\log M \log n)$ time, $\mathrm{O}(n/\log n)$ processors coding and decoding algorithms on the PRAM EREW, mesh of trees, pyramidal, and multigrid architectures.

In section 2, we describe the PRAM EREW encoder and decoder. In section 3, we explain how the parallel encoder and decoder are implemented on the mesh of trees. In section 4, we explain how the parallel encoder and decoder are implemented on the pyramid with the same parallel complexity under some realistic assumptions. Conclusions and future work are given in section 4 where the implementations on the multigrid, which is the simplest of the architectures mentioned above, are discussed.

2 The PRAM EREW Encoder and Decoder

To achieve sublinear time we partition an m x m' image I in x x y rectangular areas, where x and y are $\Theta(\log^{1/2} n)$, and n is the size of the image. In parallel for each area, one processor applies the sequential parsing algorithm so that in $\mathrm{O}(\log M \log n)$ time each area will be parsed in rectangles, some of which are monochromatic. Before encoding we wish to compute larger monochromatic rectangles.

2.1 Computing the Monochromatic Rectangles

Differently from [3], we compute larger monochromatic rectangles by merging adjacent monochromatic areas without considering those monochromatic rectangles properly contained in some area. In practice, these areas are very small and such limitation has no relevant effect on the compression ratio.

We denote with $A_{i,j}$ for $1 \leq i \leq \lceil m/x \rceil$ and $1 \leq j \leq \lceil m'/y \rceil$ the areas into which the image is partitioned. In parallel for $1 \leq i \leq \lceil m/x \rceil$, if i is odd, a processor merges areas $A_{2i-1,j}$ and $A_{2i,j}$ provided they are monochromatic and have the same color. The same is done horizontally for $A_{i,2j-1}$ and $A_{i,2j}$. At the k-th step, if areas $A_{(i-1)2^{k-1}+1,j}$, $A_{(i-1)2^{k-1}+2,j}$, ..., $A_{i2^{k-1},j}$, with i odd, were merged, then they will merge with areas

$A_{i2^{k-1}+1,j}$, $A_{i2^{k-1}+2,j}$, ..., $A_{(i+1)2^{k-1},j}$, if they are monochromatic with the same color. The same is done horizontally for $A_{i,(j-1)2^{k-1}+1}$, $A_{i,(j-1)2^{k-1}+2}$, ..., $A_{i,j2^{k-1}}$, with j odd, and $A_{i,j2^{k-1}+1}$, $A_{i,j2^{k-1}+2}$, ..., $A_{i,(j+1)2^{k-1}}$. After $O(\log M)$ steps, the procedure is completed and each step takes $O(\log n)$ time and $O(n/\log n)$ processors since there is one processor for each area of logarithmic size. Therefore, the image parsing phase is realized with $O(\log M \log n)$ time and $O(n/\log n)$ processors on the PRAM EREW.

2.2 The Parallel Encoder

We derive the sequence of pointers from the image parsing computed above. In $O(\log n)$ time with $O(n/\log n)$ processors we can identify every upper left corner of a match (proper, monochromatic, or raw) by assigning a different segment of logarithmic length on a row to each processor. Each processor detects the upper left corners on its segment. Then, by parallel prefix we obtain a sequence of pointers decodable by the decompressor paired with the sequential heuristic. However, the decoding of such sequence seems hard to parallelize. In order to design a parallel decoder, it is more suitable to produce the sequence of pointers by a raster scan of each of the areas into which the image was originally partitioned, where the areas are ordered by a raster scan themselves. Then, again we can easily derive the sequence of pointers in $O(\log n)$ time with $O(n/\log n)$ processors by detecting in each of the areas the upper left corners of a match and producing the sequence of pointers by parallel prefix.

As mentioned in the introduction, the encoding scheme for the pointers uses a flag field indicating whether there is a monochromatic rectangle (0 for the white ones and 10 for the black ones), a proper match (110), or raw data (111). For the feasibility of the parallel decoder, we want to indicate the end of the encoding of the sequence of matches with the upper left corner in a specific logarithmic area. Therefore, we change the encoding scheme by associating the flag field 1110 to the raw match so that we can indicate with 1111 the end of the sequence of pointers corresponding to a given area. Moreover, since some areas could be entirely covered by a monochromatic match 1111 is followed by the index associated with the next area by the raster scan. The pointer of a monochromatic match has fields for the width and the length while the pointer of a proper match also has fields for the coordinates of the left upper corner of the copy in the window. In order to save bits, the value stored in any of these fields is the binary value of the field plus 1 (so, we employ the zero value). This coding technique is more redundant than others previously designed, but its compression effectiveness is still better than the one of the square greedy matching technique.

2.3 The Parallel Decoder

The parallel decoder has three phases. Observe that at each position of the binary sequence encoding the image, we read a subsequence of bits that is either 1111 (recall that the k bits following 1111 provide the area index, where k is the number of bits used to encode it) or can be interpreted as a flag field of a pointer. Then, in the first phase we reduce the binary sequence to a doubly-linked structure and apply the well-known Euler tour technique in order to identify for each area the corresponding pointers. The reduction works as follows: link each position p of the sequence to the position next to the end of the subsequence starting in position p that either can be interpreted as a pointer or is equal to 1111 followed by k bits. For those suffixes of the sequence which can be interpreted as pointers, their first positions are linked to a special node denoting the end of the coding. For those suffixes of the sequence which cannot be interpreted as pointers, their first positions are not linked to anything. The linked structure is a forest with one tree rooted in the special node denoting the end of the coding and the other trees rooted in the first position of a suffix of the encoding sequence not interpretable as a pointer. The first position of the binary sequence is a leaf of the tree rooted in the special node. The sequence of pointers encoding the image is given by the path from the first position to the root. In order to compute such path we need the children to be doubly-linked to the parent. Then, we need to reserve space for each node to store the links to the children. Each node has at most five children since there are only four different pointer sizes (white, black, raw, or proper match). So, for each position p of the binary sequence we set aside five locations $[p,1]\cdots[p,5]$, initially set to zero. When a link is added from position p' to p, depending on whether the subsequence starting in position p' is 1111 or can be interpreted as a pointer to a raw, white, black or proper match, the value p' is overwritten on location $[p,1]$, $[p,2]$, $[p,3]$ $[p,4]$ or $[p,5]$, respectively. Then, by means of the well-known Euler technique we can linearize the linked structure and apply list ranking to obtain the path from the first position of the sequence to the root of its tree. It is well-known that all this can be computed in $\mathrm{O}(\log n)$ time with $\mathrm{O}(n/\log n)$ processors on the PRAM EREW, since the row image size is greater than the size of the sequence. Then, still in $\mathrm{O}(\log n)$ time with $\mathrm{O}(n/\log n)$ processors we can identify the positions on the path corresponding to 1111.

In the second phase of the parallel decoder a different processor decodes the sequence of pointers corresponding to a different area. As far as the pointers to monochromatic matches are considered, each processor decompresses either a match contained in an area or the portion of the match corresponding to the left upper area. Therefore, after the second phase an area might not be decompressed. Obviously, the second phase requires $\mathrm{O}(\log n)$ time and $\mathrm{O}(n/\log n)$ processors on the PRAM EREW.

The third phase completes the decoding. In the previous subsection, we denoted with $A_{i,j}$ for $1 \le i \le \lceil m/x \rceil$ and $1 \le j \le \lceil m'/y \rceil$ the areas into which the image was partitioned by the encoder. At the first step of the third phase, one processor for each area $A_{2i-1,j}$ decodes $A_{2i,j}$, if $A_{2i-1,j}$ is the upper left portion of a monochromatic match and the length field of the corresponding pointer informs that $A_{2i,j}$ is part of the match. The same is done horizontally for $A_{i,2j-1}$ and $A_{i,2j}$ (using the width field of its pointer) if it is known already by the decoder that $A_{i,2j-1}$ is part of a monochromatic match. Similarly at the k-th step, one processor for each of the areas $A_{(i-1)2^{k-1}+1,j}$, $A_{(i-1)2^{k-1}+2,j}$, $\ldots$, $A_{i2^{k-1},j}$, with i odd, decodes the areas $A_{i2^{k-1}+1,j}$, $A_{i2^{k-1}+2,j}$, $\ldots$, $A_{(i+1)2^{k-1},j}$, respectively. The same is done horizontally for $A_{i,(j-1)2^{k-1}+1}$, $A_{i,(j-1)2^{k-1}+2}$, $\ldots$, $A_{i,j2^{k-1}}$, with j odd, and $A_{i,j2^{k-1}+1}$, $A_{i,j2^{k-1}+2}$, $\ldots$, $A_{i,(j+1)2^{k-1}}$. After $\mathrm{O}(\log M)$ steps the image is entirely decompressed. Each step takes $\mathrm{O}(\log n)$ time and $\mathrm{O}(n/\log n)$ processors since there is one processor for each area of logarithmic size. Therefore, the decoder is realized with $\mathrm{O}(\log M \log n)$ time and $\mathrm{O}(n/\log n)$ processors on the PRAM EREW.

3 The Mesh of Trees Implementations

A *mesh of trees* is a network of $3N^2 - 2N$ processors with N being a power of 2, consisting of an N x N grid where a complete binary tree of processors is built on

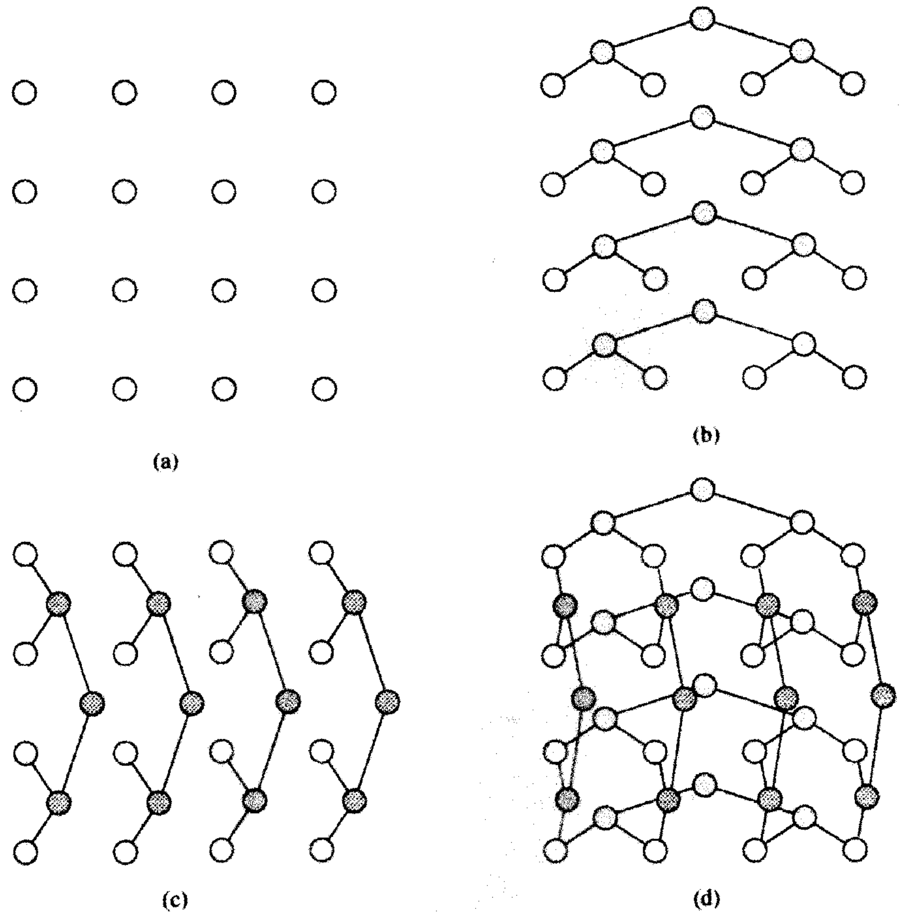


Figure 1. The construction of a mesh of trees with a 4x4 processor grid.

each row and each column as shown in Figure 1. First, we give a detailed description of the mesh of trees compression algorithm. Then, we provide the implementation of the parallel decoder.

Let max be equal to max $\{m, m'\}$. We assume m and m' have the same order of magnitude, as in practice with the height and the width of an image. Let N be the smallest power of two greater than $\lceil max/ \log^{1/2} n \rceil$, where n is the size of an m x m' image. Then, the number of processors of the mesh of trees is $\mathrm{O}(n/ \log n)$ and we can store the logarithmic rectangular areas into which the parallel algorithm partitions the image into the N x N grid. Starting from the upper left corner of the grid, a different processor stores a different rectangular area and applies the sequential compression heuristic to such area. The remaining processors are inactive. From now on, we will refer only to the active processors.

3.1 Computing the Monochromatic Rectangles

After the compression heuristic has been executed on each area, it is easy to see that the PRAM EREW procedure to compute larger monochromatic rectangles can be implemented on a mesh of trees with the same number of processors without slowing it down. In fact, if i is odd, the processors storing areas $A_{2i-1,j}$ and $A_{2i,j}$ merge them provided they are monochromatic and have the same color. The same is done horizontally for $A_{i,2j-1}$ and $A_{i,2j}$. At the k-th step, if areas $A_{(i-1)2^{k-1}+1,j}$, $A_{(i-1)2^{k-1}+2,j}$, ..., $A_{i2^{k-1},j}$, with i odd, were merged, the processor storing area $A_{i2^{k-1},j}$ will broadcast to the processors storing the areas $A_{i2^{k-1}+1,j}$, $A_{i2^{k-1}+2,j}$, ..., $A_{(i+1)2^{k-1},j}$ to merge with the above areas, if they are monochromatic with the same color. This is done in logarithmic time using the column trees. The same is done horizontally for $A_{i,(j-1)2^{k-1}+1}$, $A_{i,(j-1)2^{k-1}+2}$, ..., $A_{i,j2^{k-1}}$, with j odd, and $A_{i,j2^{k-1}+1}$, $A_{i,j2^{k-1}+2}$, ..., $A_{i,(j+1)2^{k-1}}$, using the row trees. After $O(\log M)$ steps, the procedure is completed and each step takes $O(\log n)$ time and $O(n/\log n)$ processors since there is one processor for each area of logarithmic size. Therefore, the image parsing phase is realized with $O(\log M \log n)$ time and $O(n/\log n)$ processors on the mesh of trees.

3.2 The Parallel Encoder

The sequence of pointers for each area can be trivially produced on the grid. If we assume the possibility of a parallel output, the sequences can be put together by parallel prefix. This can be realized in $O(\log n)$ time on a mesh of trees with $O(n/\log n)$ processors.

3.3 The Parallel Decoder

We know that the end of the encoding of an area is indicated by 1111 followed by the index of the area corresponding to the next encoding. Then, we can store the encodings in the positions of the grid corresponding to the locations of the areas in the image. In fact, the first phase of the PRAM EREW decoding algorithm corresponds to the input process of a distributed memory system (as the mesh of trees is) and is not part of our complexity analysis. At this point, each processor on the grid completes the second phase of the decoder described in subsection 2.3. Then, it is easy to see that the third and last phase of the PRAM decoder is implementable on a mesh of trees with the same number of processor and no slowdown. In conclusion, the decoder takes $O(\log M \log n)$ time on a mesh of trees with $O(n/\log n)$ processors.

4 The Pyramid Implementations

An N x N *pyramid* is a network consisting of $\log N + 1$ two-dimensional grids with N being a power of 2, each one of size $N/2^k$ x $N/2^k$ for $0 \leq k \leq \log N$. The grids are interconnected so that the (i, j) processor on the 2^k x 2^k grid is connected to processors $(2i-1, 2j-1)$, $(2i-1, 2j)$, $(2i, 2j-1)$ and $(2i, 2j)$ on the 2^{k+1} x 2^{k+1} grid, as shown in Figure 2 for the 4 x 4 pyramid network. As for the mesh of trees, let N be the smallest power of two greater than $\lceil max/\log^{1/2} n \rceil$, where n is the size of an m x m' image and max is equal to max $\{m, m'\}$. If we assume that m and m' have the same order of magnitude, the number of processors of the pyramid is $O(n/\log n)$ and we can store the logarithmic rectangular areas into which the parallel algorithm partitions the image into the N x N grid. Starting from the upper left corner of the grid, a different processor stores a different rectangular area and applies the sequential compression heuristic to such area while the other processors remain inactive. From now on, we will refer only to the active processors.

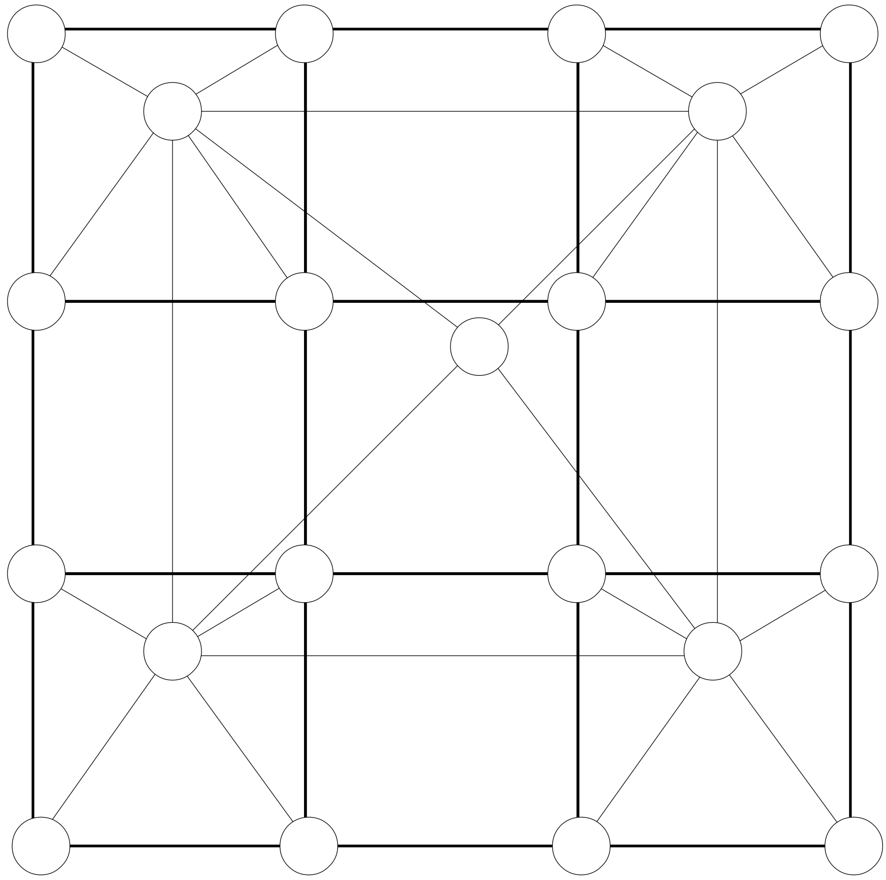


Figure 2. A 4 x 4 pyramid network.

4.1 Computing the Monochromatic Rectangles

After the compression heuristic has been executed on each area, we have to show how the PRAM EREW procedure to compute larger monochromatic rectangles can be implemented on a pyramid with the same number of processors without slowing it down. This is possible by making some realistic assumptions. Let ℓ_R and w_R be the length and the width of a monochromatic match R, respectively. We define $s_R = \max \{\ell_R, w_R\}$. We make a first assumption that the number of monochromatic matches R with $s_R \geq 2^k \lceil \log^{1/2} n \rceil$ is $O(N^2/2^{2k})$ for $1 \leq k \leq \log N - 1$. If i is odd, the processors storing areas $A_{2i-1,j}$ and $A_{2i,j}$ merge them provided they are monochromatic and have the same color. The same is done horizontally for $A_{i,2j-1}$ and $A_{i,2j}$. At the k-th step, if areas $A_{(i-1)2^{k-1}+1,j}$, $A_{(i-1)2^{k-1}+2,j}$, …, $A_{i2^{k-1},j}$, with i odd, were merged for $w_1 \leq j \leq w_2$, the processor storing area A_{i2^{k-1},w_2} will broadcast to the processors storing the areas $A_{i2^{k-1}+1,j}$, $A_{i2^{k-1}+2,j}$, …, $A_{(i+1)2^{k-1},j}$ to merge with the above areas for $w_1 \leq j \leq w_2$, if they are monochromatic with the same color. The same is done horizontally, that is, if $A_{i,(j-1)2^{k-1}+1}$, $A_{i,(j-1)2^{k-1}+2}$, …, $A_{i,j2^{k-1}}$, with j odd, were merged for $\ell_1 \leq i \leq \ell_2$, the processor storing area $A_{\ell_2,j2^{k-1}}$ will broadcast to the processors storing the areas $A_{i,j2^{k-1}+1}$, $A_{i,j2^{k-1}+2}$, …, $A_{i,(j+1)2^{k-1}}$ to merge with the above areas for $\ell_1 \leq i \leq \ell_2$, if they are monochromatic with the same color. After $O(\log M)$ steps, the procedure is completed. If the waste factor is less than 2, as conjectured in [9], we can make a second assumption that each pixel is covered by a constant small number of monochromatic matches. It follows from this second

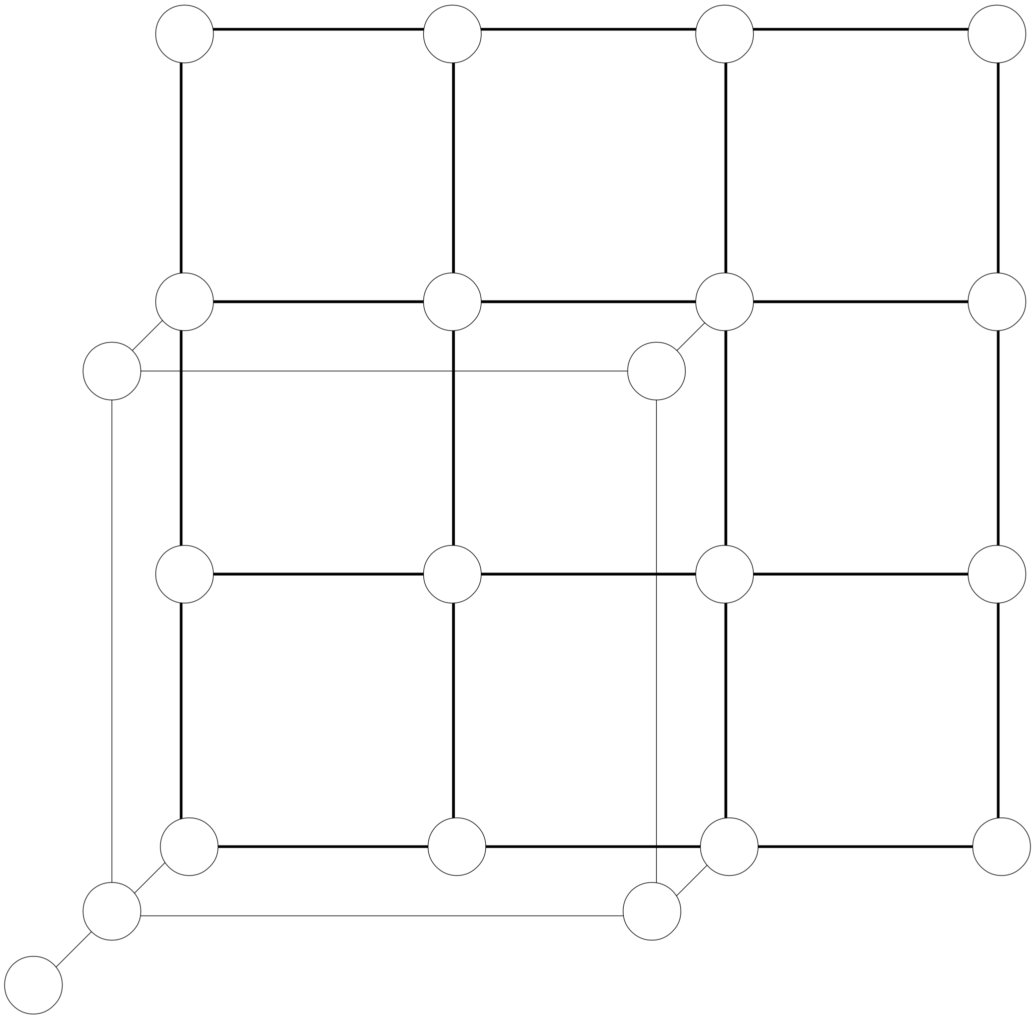


Figure 3. A 4 x 4 multigrid network.

assumption that the information about the monochromatic matches is distributed among the processors of a grid in a way very close to uniform. Then, it follows from the first assumption that the amount of information each processor of the grid at level k must broadcast is constant, for $1 \leq k \leq \log N - 1$. Therefore, each step takes $O(\log n)$ time and the image parsing phase is realized with $O(\log M \log n)$ time and $O(n/\log n)$ processors on the pyramid. Finally, we want to point out that the unique processor of the 1 x 1 grid at level $\log n$ is not involved in the computation of the image parsing and is used only for the ineherently sequential input/output operations which have, generally speaking, standard solutions for network algorithms.

4.2 The Parallel Encoder

The sequence of pointers for each area can be trivially produced on the grid at level 0. This is, obviously, realized in $O(\log n)$ time on a pyramid with $O(n/\log n)$ processors.

4.3 The Parallel Decoder

As for the mesh of trees, we can store the encodings of each area in the positions of the grid at level 0 corresponding to the locations of the areas in the image. At this point, each processor on the grid completes the second phase of the decoder described in subsection 2.3. Then, it is easy to see that the third and last phase of the PRAM decoder is implementable on a pyramid with the same number of processor and no slowdown, if the same realistic assumptions are made. In conclusion, the decoder takes $O(\log M \log n)$ time on a pyramid with $O(n/\log n)$ processors.

5 Conclusions

Parallel coding and decoding algorithms for lossless image compression by block matching were shown requiring $O(\log M \log n)$ time and $O(n/\log n)$ processors on the PRAM-EREW, the mesh of trees and the pyramid. The parallel coding algorithms are work-optimal since the sequential time required by the coding is $\Omega(n \log M)$. On the other hand, the parallel decoding algorithms are not work-optimal since the sequential decompression time is linear. The mesh of trees and pyramid implementations of the decoder have the same performance of the PRAM EREW implementation if we do not consider the input process. The pyramid is a simpler architecture than the mesh of trees [6] and needs some realistic assumptions to give the same performance. There exist real parallel machines whose architecture is a pyramid. One of them is at the University of Pavia in Italy (PAPIA). As future work, we wish to implement our algorithm on this machine. An even simpler architecture than the pyramid is the multigrid, which is computationally equivalent up to a factor of 2 in speed to the pyramid [6]. Since the multigrid is a subgraph of the pyramid (Figure 3), we wish to implement and experiment our algorithm on this architecture as well.

References

1. R. P. Brent: *A linear algorithm for data compression.* Australian Computer Journal, 19 1987, pp. 64–68.
2. L. Cinque and S. DeAgostino: *A parallel decoder for lossless image compression by block matching*, in Proceedings IEEE Data Compression Conference, 2007, pp. 183–192.
3. L. Cinque, S. DeAgostino, and F. Liberati: *A work-optimal parallel implementation of lossless image compression by block matching.* Nordic Journal of Computing, 10 2003, pp. 13–20.
4. S. DeAgostino: *A work-optimal parallel implementation of lossless image compression by block matching*, in Proceedings Prague Stringology Conference, 2002, pp. 1–8.
5. S. DeAgostino: *Lossless image compression by block matching on a mesh of trees*, in Proceedings IEEE Data Compression Conference, Poster Session, 2006, p. 443.
6. F. T. Leighton: *Introduction to Parallel Algorithms and Architectures*, Morgan-Kaufmann, 1992.
7. A. Lempel and J. Ziv: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23 1977, pp. 337–343.
8. J. A. Storer: *Lossless image compression using generalized lz1-type methods*, in Proceedings IEEE Data Compression Conference, 1996, pp. 290–299.
9. J. A. Storer and H. Helfgott: *Lossless image compression by block matching.* The Computer Journal, 40 1997, pp. 137–145.
10. J. R. Waterworth: *Data compression system.* US Patent 4 701 745, 1987.
11. D. A. Whiting, G. A. George, and G. E. Ivey: *Data compression apparatus and method.* US Patent 5016009, 1991.

Speeding up Lossless Image Compression: Experimental Results on a Parallel Machine

Luigi Cinque[1], Sergio De Agostino[1], and Luca Lombardi[2]

[1] Computer Science Department
Sapienza University
Via Salaria 113, 00198 Roma, Italy
{cinque, deagostino}@di.uniroma1.it
[2] Computer Science Department
University of Pavia
Via Ferrara 1, 27100 Pavia, Italy
luca.lombardi@unipv.it

Abstract. Arithmetic encoders enable the best compressors both for bi-level images (JBIG) and for grey scale and color images (CALIC), but they are often ruled out because too complex. The compression gap between simpler techniques and state of the art compressors can be significant. Storer extended dictionary text compression to bi-level images to avoid arithmetic encoders (BLOCK MATCHING), achieving 70 percent of the compression of JBIG1 on the CCITT bi-level image test set. We were able to partition an image into up to a hundred areas and to apply the BLOCK MATCHING heuristic independently to each area with no loss of compression effectiveness. On the other hand, we presented in [5] a simple lossless compression heuristic for gray scale and color images (PALIC), which provides a highly parallelizable compressor and decompressor. In fact, it can be applied independently to each block of 8x8 pixels, achieving 80 percent of the compression obtained with LOCO-I (JPEG-LS), the current lossless standard in low-complexity applications. We experimented the BLOCK MATCHING and PALIC heuristics with up to 32 processors of a 256 Intel Xeon 3.06 GHz processors machine in Italy (`avogadro.cilea.it`) on a test set of large topographic bi-level images and color images in RGB format. We obtained the expected speed-up of the compression and decompression times, achieving parallel running times about twenty-five times faster than the sequential ones.

Keywords: lossless image compression, sliding dictionary, differential coding, parallelization

1 Introduction

Lossless image compression is often realized by extending string compression methods to two-dimensional data. Standard lossless image compression methods extend model driven text compression [1], consisting of two distinct and independent phases: *modeling* [16] and *coding* [15]. In the coding phase, arithmetic encoders enable the best model driven compressors both for bi-level images (JBIG [10]) and for grey scale and color images (CALIC [20]), but they are often ruled out because too complex. The compression gap between simpler techniques and state of the art compressors can be significant.

Storer [18] extended dictionary text compression [17] to bi-level images to avoid arithmetic encoders by means of a square greedy matching technique (BLOCK MATCHING), achieving 70 percent of the compression of JBIG1 on the CCITT bi-level image test set. The technique is a two-dimensional extension of LZ1 compression [12] and is suitable for high speed applications by means of a simple hashing scheme.

Luigi Cinque, Sergio De Agostino, Luca Lombardi: Speeding up Lossless Image Compression: Experimental Results on a Parallel Machine, pp. 35–45.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

Rectangle matching improves the compression performance, but it is slower since it requires $O(M \log M)$ time for a single match, where M is the size of the match [19]. Therefore, the sequential time to compress an image of size n by rectangle matching is $\Omega(n \log M)$. However, rectangle matching is more suitable for polylogarithmic time work-optimal parallel implementations on the PRAM EREW [3], [6] and the mesh of trees [2], [7]. Polylogarithmic time parallel implementations were also presented for decompression on both the PRAM EREW and the mesh of trees in [2].

Parallel models have two types of complexity, the interprocessor communication and the input-output mechanism. While the input/output issue is inherent to any sublinear algorithm and has standard solutions, the communication cost of the computational phase after the distribution of the data among the processors and before the output of the final result is obviously algorithm-dependent. So, we need to limit the interprocessor communication and involve more local computation. The simplest model for this phase is, of course, a simple array of processors with no interconnections and, therefore, no communication cost. The parallel implementations mentioned above require more sophisticated architectures than a simple array of processors to be executed on a distributed memory system.

Dealing with square matches, we were able to partition an image into up to a hundred areas and to apply the BLOCK MATCHING heuristic independently to each area with no loss of compression effectiveness. With rectangles we cannot obtain the same performance since the width and the length are shortened while the corresponding pointers are more space consuming than with squares. So we would rather implement the square BLOCK MATCHING heuristic on an array of size up to a hundred processors.

The extension of Storer's method to grey scale and color images was left as an open problem, but it seems not feasible since the high cardinality of the alphabet causes an unpractical exponential blow-up of the hash table used in the implementation.

As far as the model driven method for grey scale and color image compression is concerned, the modeling phase consists of three components: the determination of the context of the next pixel, the prediction of the next pixel and a probabilistic model for the *prediction residual*, which is the value difference between the actual pixel and the predicted one. In the coding phase, the prediction residuals are encoded. A first step toward a good low complexity compression scheme was FELICS (Fast Efficient Lossless Image Compression System) [11], which involves Golomb-Rice codes [9], [14] rather than the arithmetic ones. With the same complexity level for compression (but with a 10 percent slower decompressor) LOCO-I (Low Complexity Lossless Compression for Images) [13] attains significantly better compression than FELICS, only a few percentage points of CALIC (Context-Based Adaptive Lossless Image Compression). As explained in [5], also polylogarithmic time parallel implementations of FELICS and LOCO-I would require more sophisticated architectures than a simple array of processors.

The use of prediction residuals for grey scale and color image compression relies on the fact that most of the times there are minimal variations of color in the neighborood of one pixel. Therefore, differently than for bi-level images we should be able to implement an extremely local procedure which is able to achieve a satisfying degree of compression by working independently on different very small blocks. In [5], we showed such procedure. We presented the heuristic for grey scale images, but it could also be applied to color images by working on the different components [4]. We call such procedure PALIC (Parallelizable Lossless Image Compression). In fact, the main advantage of PALIC is that it provides a highly parallelizable compressor and decompressor since it can be applied independently to each block of 8x8 pixels, achieving 80 percent of the compression obtained with LOCO-I (JPEG-LS), the current lossless standard in low-complexity applications.

255	255	255	254	254	110	110	110
255	255	255	254	254	110	110	110
255	255	255	254	254	110	110	110
255	255	255	254	254	110	110	110
255	255	254	128	127	128	129	130
255	253	253	128	128	129	130	131
254	253	252	129	129	130	131	132
253	252	251	130	130	130	254	255

Figure 1. An 8x8 pixel block of a grey scale image.

The compressed form of each block employs a header and a fixed length code. Two different techniques might be applied to compress the block. One is the simple idea of reducing the alphabet size by looking at the values occurring in the block. The other one is to encode the difference between the pixel value and the smallest one in the block. Observe that this second technique can be interpreted in terms of the model driven method, where the block is the context, the smallest value is the prediction and the fixed length code encodes the prediction residual. More precisely, since the code is fixed length the method can be seen as a two-dimensional extension of differential coding [8]. Differential coding, often applied to multimedia data compression, transmits the difference between a given signal sample and another sample.

In this paper, we experimented the square BLOCK MATCHING and PALIC heuristics with up to 32 processors of a 256 Intel Xeon 3.06 GHz processors machine in Italy (`avogadro.cilea.it`) on a test set of large topographic bi-level images and color images in RGB format. We obtained the expected speed-up of the compression and decompression times, achieving parallel running times about twenty-five times faster than the sequential ones.

In section 2, we explain the heuristics. In section 3 we provide the experimental results on the parallel machine. Conclusions are given in section 4.

2 BLOCK MATCHING and PALIC

Among the different ways of reading an image, we assume the square BLOCK MATCHING heuristic scans an m x m' image row by row (*raster* scan). A 64K table with one position for each possible 4x4 subarray is the only data structure used. All-zero and all-one rectangles are handled differently. The encoding scheme is to precede each item with a flag field indicating whether there is a monochromatic square, a match or raw data. When there is a match, the 4x4 subarray in the current position is hashed to yield a pointer to a copy. This pointer is used for the current square greedy match and then replaced in the hash table by a pointer to the current position. The procedure for computing the largest square match with left upper corners in positions (i, j) and (k, h) takes $O(M)$ time, where M is the size of the match. Obviously, this procedure can be used for computing the largest monochromatic square in a given position (i, j) as well. If the 4 x 4 subarray in position (i, j) is monochromatic, then we compute the largest monochromatic square in that position. Otherwise, we compute the largest square match in the position provided by the hash table and update the table with the current position. If the subarray is not hashed to a pointer, then it is left uncompressed and added to the hash table with its current position. The positions covered by matches are skipped in the linear scan of the image. Therefore, the sequential time to compress an image of size n by square matching is $O(n)$. We want to point out that besides the proper matches we use to call a match every rectangle of the parsing of the image produced by the heuristic. We also call a pointer the encoding of every match. As mentioned above, the encoding scheme for the pointers uses a flag field indicating whether there is a monochromatic rectangle (0 for the white ones and 10 for the black ones), a proper match (110) or raw data (111).

As mentioned in the introduction, we were able to partition an image into up to a hundred areas and to apply the BLOCK MATCHING heuristic independently to each area with no loss of compression effectiveness on both the CCITT bi-level image test set and the bi-level version of the set of five 4096 x 4096 pixels images in Figures 2–6.

Moreover, in order to implement decompression on an array of processors, we want to indicate the end of the encoding of a specific area. Therefore, we change the encoding scheme by associating the flag field 1110 to the raw match so that we can indicate with 1111 the end of the sequence of pointers corresponding to a given area.

We explain now how to apply the PALIC heuristic independently to blocks of 8x8 pixels of a grey scale image. We still assume to read the image with a raster scan on each block. The heuristic applies at most three different ways of compressing the block and chooses the best one. The first one is the following.

The smallest pixel value is computed on the block. The header consists of three fields of 1 bit, 3 bits and 8 bits, respectively. The first bit is set to 1 to indicate that we compress a block of 64 pixels. This is because one of the three ways will partition the block in four sub-blocks of 16 pixels and compress each of these smaller areas. The 3-bits field stores the minimum number of bits required to encode in binary the distance between the smallest pixel value and every other pixel value in the block. The 8-bits field stores the smallest pixel value. If the number of bits required to encode the distance, say k, is at most 5, then a code of fixed length k is used to encode the 64 pixels, by giving the difference between the pixel value and the smallest one in the block. To speed up the procedure, if k is less or equal to 2 the other ways are not tried because we reach a satisfying compression ratio on the block. The second and third ways are the following.

The second way is to detect all the different pixel values in the 8x8 block and to create a reduced alphabet. Then, to encode each pixel in the block using a fixed length

Figure 2. Image 1.

Figure 3. Image 2.

code for this alphabet. The employment of this technique is declared by setting the 1-bit field to 1 and the 3-bits field to 110. Then, an additional three bits field stores the reduced alphabet size d with an adjusted binary code in the range $2 \leq d \leq 9$.

Figure 4. Image 3.

Figure 5. Image 4.

The last componenent of the header is the alphabet itself, a concatenation of d bytes. Then, a code of fixed length $\lceil \log d \rceil$ bits is used to encode the 64 pixels.

The third way compresses the four 4x4 pixel sub-blocks. The 1-bit field is set to 0. Four fields follow the flag bit, one for each 4x4 block. The two previous techniques

Figure 6. Image 5.

are applied to the blocks and the best one is chosen. If the first technique is applied to a block, the corresponding field stores values from 0 to 7 rather than from 0 to 5 as for the 8x8 block. If such value is in between 0 and 6, the field stores three bits. Otherwise, the three bits (111) are followed by three more. This is because 111 is used to denote the application of the second way to the block as well, which is less frequent to happen. In this case, the reduced alphabet size stored in these three additional bits has range from 2 to 7, it is encoded with an adjusted binary code from 000 to 101 and the alphabet follows. 110 denotes the application of the first technique with distances expressed in seven bits and 111 denotes that the block is not compressed. After the four fields, the compressed forms of the blocks follow, which are similar to the ones described for the 8x8 block. When the 8x8 block is not compressed, 111 follows the flag bit set to 1.

We now show how PALIC works on the example of Figure 1.

Since the difference between 110, the smallest pixel value, and 255 requires a code with fixed length 8 and the number of different values in the 8x8 block is 12, the way employed to compress the block is to work separately on the 4x4 sub-blocks. Each block will be encoded with a raster scan (row by row). The upper left block has 254 as its smallest pixel value and 255 is the only other value. Therefore, after setting the 1-bit field to zero the corresponding field is set to 001. The compressed form after the header is 1110111011101110. The reduced alphabet technique is more expensive since the raw pixel values must be given. On the other hand, the upper right block needs the reduced alphabet technique. In fact, one byte is required to express the difference between 110 and 254. Therefore, the corresponding field is set to 111000, which indicates that the reduced alphabet size is 2, and the sequence of two bytes 0110111011111110 follows. The compressed form after the header is 1000100010001000. The lower left block has 8 different values so we do not use the reduced alphabet technique since the alphabet size should be between 2 and 7. The smallest pixel value in the block is 128 and the largest difference is 127 with the pixel value 255. Since a code of fixed length 7 is required, the corresponding field is 111110. The compressed form after the header is (we introduce a space between pixel encodings in the text to make it more readable): 1111111 1111111 1111110 0000000 1111111 1111101 1111101 0000000 1111110 1111101 1111100 0000001 1111101 1111100 1111011 0000010. Observe that the compression of the block would have been the same if we had allowed the reduced alphabet size to grow up to 8. However, experimentally we found more advantageous to exclude this case in favor of the other technique. Our heuristic does not compress the lower right block since it has 8 different values and the difference between pixel values 127 and 255 requires 8 bits. Therefore, the corresponding field is 111111 and the uncompressed block follows.

We experimented PALIC on the kodak image test set, which is an extension of the standard jpeg image test set and reached 70 to 85 percent of LOCO-I compression ratio (78 percent in average). We also experimented it on the set of five 4096 x 4096 pixels grey scale topographic images in Figure 2-6 and the compression effectiveness was about 80 percent of LOCO-I compression effectiveness as for the kodak image set. The heuristic can be trivially extended to RGB color images by working sequentially on each of the three components of the block and the same compression effectiviness results in comparison with LOCO-I were obtained for the RGB version of the five images in Figures 2–6.

3 Experimental Results on a Parallel Machine

We show in Figures 7–8 the compression and decompression times of PALIC on the RGB version of the five images in Figures 2–6 doubling up the number of processors of the `avogadro.cilea.it` machine from 1 to 32. We executed the compression and decompression on each image several times. The variances of both the compression and decompression times were small and we report the greatest running times, conservatively. As it can be seen from the values on the tables, also the variance over the test set is quite small. The decompression times are faster than the compression ones and in both cases we obtain the expected speed-up, achieving parallel running times about twenty-five times faster than the sequential ones.

Image	1 proc.	2 proc.	4 proc.	8 proc.	16 proc.	32 proc.
1	227	117	57	34	17	9
2	243	120	69	33	16	10
3	235	118	72	35	17	9
4	236	131	71	34	16	9
5	232	113	67	30	16	11
Avg.	234.6	119.8	67.2	33.2	16.4	9.6

Figure 7. PALIC compression times (cs.).

The images of Figures 2–4 have the greatest parallel decompression times with 32 processors. On the other hand, the image of Figure 3 has the greatest sequential compression and decompression times. The smallest compression time with 32 processors

Image	1 proc.	2 proc.	4 proc.	8 proc.	16 proc.	32 proc.
1	128	65	32	18	11	6
2	133	66	35	21	10	6
3	130	66	44	21	13	6
4	129	91	36	20	10	5
5	123	95	46	17	10	5
Avg.	128.6	76.6	38.6	19.4	10.8	5.6

Figure 8. PALIC decompression times (cs.).

is given by the image of Figure 4, together with the images of Figure 2 and Figure 5. Instead, the smallest decompression time with 32 processors is given by the images of Figures 5–6. The image of Figure 6 also has the smallest sequential decompression time and the greatest compression time with 32 processors.

Image	1 proc.	2 proc.	4 proc.	8 proc.	16 proc.	32 proc.
1	76	39	19	11	6	3
2	81	40	23	11	5	3
3	78	39	24	12	6	3
4	79	44	24	11	5	3
5	77	38	22	10	5	4
Avg.	78.2	40	22.4	11	5.4	3.2

Figure 9. BLOCK MATCHING compression times (cs.).

Image	1 proc.	2 proc.	4 proc.	8 proc.	16 proc.	32 proc.
1	43	22	11	6	4	2
2	44	22	12	7	3	2
3	43	22	15	7	4	2
4	43	30	12	7	3	2
5	41	32	15	6	3	2
Avg.	42.8	25.6	13	6.6	3.4	2

Figure 10. BLOCK MATCHING decompression times (cs.).

We obtained similar results for the BLOCK MATCHING heuristic. In Figures 9–10 we show the compression and decompression times of the square BLOCK MATCHING heuristic on the bi-level version of the five images in Figures 2–6, doubling up the number of processors of the `avogadro.cilea.it` machine from 1 to 32. This means that when 2^k processors are involved, for $1 \leq k \leq 5$, the image is partitioned into 2^k areas and the compression heuristic is applied in parallel to each area, independently.

As far as decompression is concerned, each of the 2^k processors decodes the pointers corresponding to a given area.

4 Conclusions

In this paper, we showed experimental results on the coding and decoding times of two lossless image compression methods on a real parallel machine. By doubling up the number of processors from 1 to 32, we obtained the expected speed-up on a test set of large topographic bi-level images and color images in RGB format, achieving parallel running times about twenty-five times faster than the sequential ones. The feasibility of a highly parallelizable compression method for grey scale and color images relied on the fact that most of the times there are minimal variations of color in the neighborood of one pixel. Therefore, we were able to implement an extremely local procedure which achieves a satisfying degree of compression by working independently on different very small blocks. On the other hand, we designed a non-massive approach to bi-level image compression which could be implemented on an array of processors of reasonable size, achieving a satisfying degree of compression. Such goal was realized by making each processor work on a single large block rather than on many very small blocks as when the non-massive way is applied to grey scale or color images.

References

1. T. C. Bell, J. G. Cleary, and I. H. Witten: *Text Compression*, Prentice Hall, 1980.
2. L. Cinque and S. DeAgostino: *A parallel decoder for lossless image compression by block matching*, in Proceedings IEEE Data Compression Conference, 2007, pp. 183–192.
3. L. Cinque, S. DeAgostino, and F. Liberati: *A work-optimal parallel implementation of lossless image compression by block matching*. Nordic Journal of Computing, 2003, pp. 183–192.
4. L. Cinque, S. DeAgostino, and F. Liberati: *A simple lossless compression heuristic for rgb images*, in Proceedings IEEE Data Compression Conference, Poster Session, 2004.
5. L. Cinque, S. DeAgostino, F. Liberati, and B. Westgeest: *A simple lossless compression heuristic for grey scale images*. International Journal of Foundations of Computer Science, 16 2005, pp. 1111–1119.
6. S. DeAgostino: *A work-optimal parallel implementation of lossless image compression by block matching*, in Proceedings Prague Stringology Conference, 2002, pp. 1–8.
7. S. DeAgostino: *Lossless image compression by block matching on a mesh of trees*, in Proceedings IEEE Data Compression Conference, Poster Session, 2006, p. 443.
8. J. D. Gibson: *Adaptive prediction in speech differential encoding system*. Proceedings of the IEEE, 68 1980, pp. 488–525.
9. S. W. Golomb: *Run-length encodings*. IEEE Transactions on Information Theory, 12 1966, pp. 399–401.
10. P. G. Howard, F. Kossentini, B. Martinis, S. Forchammer, W. J. Rucklidge, and F. Ono: *The emerging jbig2 standard*. IEEE Transactions on Circuits and Systems for Video Technology, 8 1998, pp. 838–848.
11. P. G. Howard and J. S. Vitter: *Fast and efficient lossles image compression*, in Proceedings IEEE Data Compression Conference, 1993, pp. 351–360.
12. A. Lempel and J. Ziv: *A universal algorithm for sequential data compression*. IEEE Transactions on Information Theory, 23 1977, pp. 337–343.
13. G. S. G. M. J. Weimberger and G. Sapiro: *Loco-i: A low complexity, context based, lossless image compression algorithm*, in Proceedings IEEE Data Compression Conference, 1996, pp. 140–149.
14. R. F. Rice: *Some practical universal noiseless coding technique - part i*, Tech. Rep. JPL-79-22, Jet Propulsion Laboratory, Pasadena, California, USA, 1979.

15. J. Rissanen: *Generalized kraft inequality and arithmetic coding.* IBM Journal on Research and Development, 20 1976, pp. 198–203.
16. J. Rissanen and G. G. Langdon: *Universal modeling and coding.* IEEE Transactions on Information Theory, 27 1981, pp. 12–23.
17. J. A. Storer: *Data Compression: Methods and Theory*, Computer Science Press, 1988.
18. J. A. Storer: *Lossless image compression using generalized lz1-type methods*, in Proceedings IEEE Data Compression Conference, 1996, pp. 290–299.
19. J. A. Storer and H. Helfgott: *Lossless image compression by block matching.* The Computer Journal, 40 1997, pp. 137–145.
20. X. Wu and N. D. Memon: *Context-based, adaptive, lossless image coding.* IEEE Transactions on Communications, 45 1997.

Huffman Coding with Non-Sorted Frequencies

Shmuel T. Klein and Dana Shapira

[1] Department of Computer Science
Bar Ilan University, Ramat Gan, Israel
tomi@cs.biu.ac.il

[2] Department of Computer Science
Ashkelon Academic College, Ashkelon, Israel
shapird@ash-college.ac.il

Abstract. A standard way of implementing Huffman's optimal code construction algorithm is by using a sorted sequence of frequencies. Several aspects of the algorithm are investigated as to the consequences of relaxing the requirement of keeping the frequencies in order. Using only partial order may speed up the code construction, which is important in some applications, at the cost of increasing the size of the encoded file.

1 Introduction

Huffman's algorithm [6] is one of the major milestones of data compression, and even though more than half a century has passed since its invention, the algorithm or its variants find their way into many compression applications to this very day. The algorithm repeatedly combines the two smallest frequencies, and thus stores the set of frequencies either in a heap or in sorted form, yielding an $\Omega(n \log n)$ algorithm for the construction of the Huffman code, where n is the size of the alphabet to be encoded.

Working with a sorted set of frequencies is indeed a sufficient condition to get an optimal code, but the condition is not necessary. In certain cases, one can get optimal results even if the frequencies are not fully sorted, in other cases the code might not be optimal, but very closely so. On the other hand, relaxing the requirement of keeping the frequencies in order may yield time savings, as the generation of the code, if the frequencies are already given in order, or if their order can be ignored, takes only $O(n)$ steps.

One might object that since the alphabet size n can often be considered as constant relative to the size of the text to be encoded, there is no much sense in trying to improve the code construction process, and any gained savings will only marginally affect the overall compression time. But there are other scenarios for which the above mentioned effort may be justifiable: the ratio between the sizes of the text and the code is not always very large; instead of using a single Huffman code, better results are obtained when several such codes are used. For example, when the text is considered as being generated by a first order Markov process, one might use a different code for the successors of the different characters. When dynamic coding is used, the code is rebuilt periodically, sometimes even after each character read.

The loss incurred by not using an optimal (Huffman) code is often tolerable, and other non-optimal variants with desirable features, such as faster processing and simplicity have been suggested, for example Tagged Huffman codes [4], End-Tagged Dense codes [2] and (s, c)-Dense codes [1]. Similarly, the loss of optimality caused by moving to not fully sorted frequencies can also be acceptable in certain applications,

Shmuel T. Klein, Dana Shapira: Huffman Coding with Non-Sorted Frequencies, pp. 46–53.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

for example when based on estimations rather than on actual counts. In a dynamic encoding of a sequence of text blocks $B_1, B_2, \ldots$, block B_t is often encoded on the basis of the character frequencies in $B_1, \ldots, B_{t-1}$. The encoder could use the frequencies from block B_t itself, but deliberately ignores them because they are yet unknown to the decoder. By using the frequencies gathered up to block B_{t-1} only, decoding is possible without transmitting the code itself. The accuracy, however, of these estimates is based on the assumption that block t is similar to the preceding ones as to the distribution of its characters. If this assumption does not hold, the code may be non-optimal anyway, so an additional effort of producing an optimal code for a set of underlying frequencies that are not reliable, may be an overkill.

In the next section, we investigate some properties of the Huffman process on non-sorted frequencies. Section 3 then deals with a particular application, designing an algorithm for the dynamic compression of a sequence of data packets, and report on some experiments. In Section 4 we investigate whether a similar approach may have applications to other compression schemes than Huffman's.

2 Using non-sorted frequencies

The following example shows that working with sorted frequencies is not a necessary condition for obtaining optimality. Consider the sequence of weights $\{7, 5, 3, 3, 2, 2\}$, yielding the Huffman tree in Figure 1a. If we start with a slightly perturbed sequence $\{7, 5, 3, 2, 3, 2\}$ and continue according to Huffman's algorithm, we get the tree in Figure 1b, which is still optimal since its leaves are on the same levels as before, but it is not a Huffman tree, in which we would not combine 2 with 3. The tree of Figure 1c corresponds to starting with the sorted sequence, but not keeping the order afterwards, working with the sequence $\{7, 5, 6, 4\}$ instead of $\{7, 6, 5, 4\}$ after two merges.

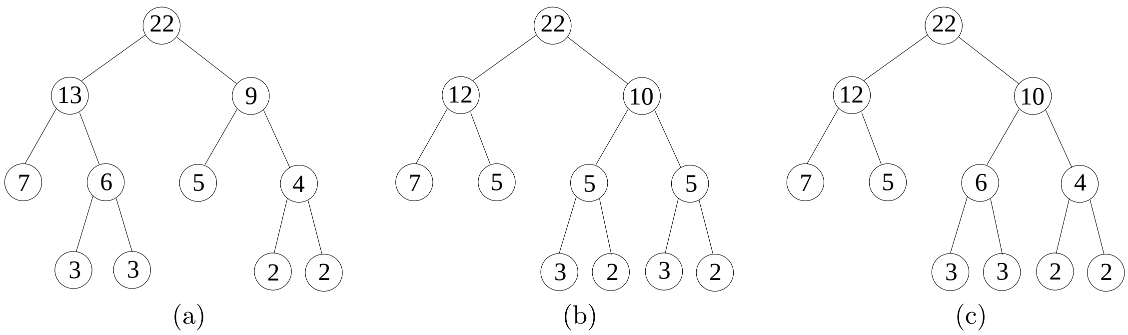


FIGURE 1: *Optimal trees*

Obviously, not paying at all attention to the order of the weights can yield very bad encodings. Consider a typical sequence of weights yielding a maximally skewed tree, that is, a tree with one leaf on each level (except the lowest level, on which there are two leaves). The Fibonacci sequence is known to be the one with the slowest increasing pace among the sequences giving such a biased tree [7], but for the ease of description we shall consider the sequence of powers of 2, more precisely, the weights 1, 1, 2, 4, $\ldots$, 2^n, for some n.

Applying regular Huffman coding to this sorted sequence, we get

$$S_{\text{Huf}} = (n+1) + \sum_{i=0}^{n}(n-i+1)2^i = 2^{n+2} - 2$$

as total size of the encoded file. If one uses the same skewed tree, but assigns the codewords in reverse order, which can happen if the initial sequence is not sorted and the tree is built without any comparisons between weights, the size of the encoded file will be

$$S_{\text{rev}} = 1 + \sum_{i=0}^{n}(i+2)2^i - 2^n = (n+1)2^{n+1} - 2^n + 1.$$

The ratio $S_{\text{rev}}/S_{\text{Huf}}$ may thus increase linearly with n, the size of the alphabet.

We therefore turn to a more realistic scenario, in which some partial ordering is allowed, but requiring an upper bound of $O(n)$ order operations, as opposed to $\theta(n \log n)$ for a full sort. Indeed, the simplest implementation of Huffman coding, after an initial sort of the weights, is keeping a sorted linked list, and repeatedly removing the two smallest elements and inserting their sum in its proper position, overall a $\theta(n^2)$ process. Using two queues Q_1 and Q_2, the first for the initial weights and the other for those created by adding two previous weights, the complexity can be reduced to $O(n)$ because the elements to be inserted into Q_2 appear in order [9]. If one starts with a sequence which is inversely sorted, the first element to be inserted into Q_2 will be the largest; hence if one continues as in the original algorithm by extracting either the two smallest elements of Q_1, or those of Q_2, or the smallest from Q_1 and that of Q_2, the first element of Q_2 will be used again only after the queue Q_1 has been emptied. The resulting tree is thus a full binary tree, with all its leaves on the same level if n is a power of 2, or on two adjacent levels if not. The depth of this tree, for the case $n = 2^k$, will be k. Returning to the above sequence of weights, the total size of the encoded file will thus be

$$S_{\text{fixed}} = \log n \left(1 + \sum_{i=0}^{n} 2^i\right) = 2^{n+1} \log n.$$

The ratio $S_{\text{fixed}}/S_{\text{Huf}}$ still tends to infinity, but increases only as $\log n$ as opposed to n above.

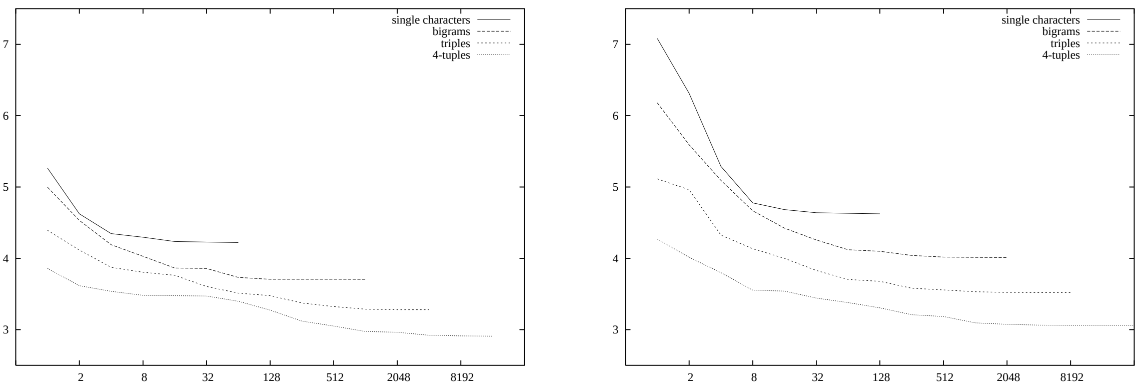
One of the ways to get some useful partial ordering in linear time is the one used in Yao's Minimum Spanning tree algorithm [12]: a parameter K is chosen, and the set of weights W is partitioned into K subsets of equal size $W_1, \ldots, W_K$, such that all the elements of W_i are smaller than any element in W_{i+1}, for $i = 1, \ldots, K-1$, but without imposing any order within each of the sets W_i. The total time for such a partition is only $O(n \log K)$, using repeatedly an $O(n)$ algorithm for finding the median first of the whole set W, then of its two halves (the $n/2$ lower and the $n/2$ upper values), then of the quarters, etc. Starting with such a partition and continuing with the help of two queues, one gets an overall linear algorithm, since K is fixed. On the other hand, K can be used as a parameter of how close the initial ordering should be to a full sort.

To empirically test this partition approach, we chose the following input files of different sizes and languages: the Bible (King James version) in English, and the

	1-grams	2-grams	3-grams	4-grams
English	52	808	6026	21886
French	131	2965	18864	56078

TABLE 1: *Alphabet sizes*

French version of the European Union's JOC corpus, a collection of pairs of questions and answers on various topics used in the ARCADE evaluation project [10]. To get also different alphabet sizes, the Bible text was stripped of all punctuation signs, whereas the French text has not been altered. We then also considered extended alphabets, consisting of bigrams, trigrams and 4-grams, that is, the text was split into a sequence of k-grams, $1 \leq k \leq 4$, and for fixed k, the set of the different non-overlapping k-grams was considered as an alphabet. Table 1 shows the sizes of the alphabets so obtained.



English text *French text*

FIGURE 2: *Average number of bits per char as function of number of blocks in partition*

Each sequence of weights was then partitioned as explained above into K equal parts, with $K = 1, 2, 4, 8, \ldots$, where in each part the original lexicographic order of the elements has been retained. Figure 2 plots the average number of bits needed to encode a single character as function of the number of partition parts K. All the plots exhibit a decreasing trend and obviously converge to the optimum when K reaches the alphabet size, but it should be noted that the convergence pace is quite fast. For example, for the 4-tuple alphabets, using $K = 1024$ corresponding to 10 partition phases, there is a loss of only 1.1% for the English and 2.2% for the French texts over the optimal Huffman code.

Another kind of partial ordering relates to a dynamic environment where the Huffman trees to be used are constantly updated. An application of this idea to a packet transmission system is discussed in the next section.

3 Dynamic compression of a sequence of data packets

Consider a stream of data packets $P_1, P_2, \ldots$ of varying sizes, which should be transmitted in compressed form over some channel. In practice, the sizes have great variability, ranging from small packets of several bytes up to large ones, spanning Megabytes. Compression of packet P_t will be based on $P_{t-k}, P_{t-k+1}, \ldots, P_{t-1}$, where

k could be chosen as $t-1$ if one wishes to use the full history, or as some constant if the compression of each packet should only depend on the distribution in some fixed number of preceding packets.

Normally, after having processed P_t, the distribution of the weights should be updated and a new Huffman tree should be built accordingly. The weights of elements which did not appear earlier are treated similarly to the appearance of new elements in dynamic Huffman coding. We suggest, however, to base the Huffman tree reconstruction not on a full sort of the updated frequencies, but on a partial one obtained from a single scan of a bubble-sort procedure. For the formal description, let s_i, $1 \le i \le n$, be the elements to be encoded. These elements can typically be characters, but could also be pairs or triplets of characters as in the example above, or even words, or more generally, any set of strings or more general elements, as long as there is some unambiguous way to partition the text into a sequence of such elements. Let $f(s_i)$ be the frequency of s_i and note that we do not require the sequence $f(s_1), f(s_2), \ldots$ to be non-decreasing. The update algorithm to be applied after each block is:

```
Update after having read P_t:
    for i ⟵ 1 to n
        add frequency of s_i within P_t to f(s_i)
        subtract frequency of s_i within P_{t-k} from f(s_i)
    for i ⟵ 1 to n − 1
        if f(s_i) > f(s_{i+1})   swap(s_i, s_{i+1})
    Build Huffman tree for sequence (f(s_1), f(s_2), ..., f(s_n)) using two queues
```

The gain of using only a single iteration of possible swaps is not only in processing time. It also allows a more moderate adaptation to changing character distributions in the case of the appearance of some very untypical data packets. Only if the changed frequencies persist also in several subsequent packets, will the Huffman tree gradually change its form to reflect the new distributions. On the other hand, if the packets are homogeneous, the procedure will zoom in on the optimal order after a small number of steps.

To simulate the above packet transmission algorithm, we took the English and French texts mentioned earlier, and partitioned them into sequences of blocks, each representing a packet. For simplicity, the block size has been kept fixed. The tests were run with single character and bigram alphabets. The following methods were compared:

1. Blocked – Block encoding: each block uses the Huffman tree built for the cumulative frequencies of all the preceding blocks to encode its characters.
2. Bubble – Using one bubble-sort iteration: each block uses the cumulative frequencies of all previous blocks as before, but after each block, only a single bubble-sort iteration is performed on the frequencies instead of sorting them completely. Huffman's algorithm is then applied on the non-sorted sequence of weights.
3. Bubble-For-k – Forgetful variant of Bubble: each block uses the cumulative frequencies not of all, but only the k previous blocks ($k \ge 0$). The frequencies of blocks that appear more than k blocks earlier are thus not counted for building the Huffman tree of the current block. This allows a better adaptation in case of heterogeneous blocks, at the price of slower convergence in the case of a more uniform behavior of the character distributions within the blocks.

For the last case we considered both Bub-For-1 and Bub-For-5, using the frequencies of the preceding block only and of the last five blocks, respectively. The first block was encoded with a fixed length code using the full single character or bigram alphabet. After each block read, the statistics were updated and a new code was generated according to the methods above. The recorded time is that of the average code construction time per block, not including the actual encoding of the block.

Single characters		Block size	Blocked	Bubble	Bubble For-1	Bubble For-5
English	Compression	200	4.112	5.532	5.697	5.607
		2000	4.114	5.532	5.553	5.541
		10000	4.123	5.533	5.536	5.533
	Time	200	0.13	0.06	0.06	0.06
		2000	0.63	0.44	0.27	0.27
		10000	2.56	1.32	1.13	1.26
French	Compression	200	4.699	6.020	5.901	5.875
		2000	4.700	6.020	5.877	5.825
		10000	4.705	6.022	5.834	5.865
	Time	200	0.27	0.09	0.09	0.11
		2000	0.49	0.30	0.30	0.31
		10000	1.47	1.26	1.28	1.28

TABLE 2: *Dynamic compression of data packets using single characters*

Single characters		Block size	Blocked	Bubble	Bubble For-1	Bubble For-5
English	Compression	2000	3.805	5.061	5.061	5.061
		10000	3.805	5.061	5.061	5.062
		20000	3.806	5.062	5.062	5.062
	Time	2000	30.1	7.3	9.0	11.6
		10000	34.9	9.2	10.8	13.4
		20000	37.4	11.1	12.9	15.2
French	Compression	2000	4.109	6.343	6.345	6.345
		10000	4.109	6.342	6.344	6.344
		20000	4.108	6.342	6.345	6.342
	Time	2000	286.2	9.9	11.3	14.0
		10000	286.6	11.1	12.9	16.1
		20000	290.4	13.4	15.1	17.6

TABLE 3: *Dynamic compression of data packets using bigrams*

Table 2 brings the results for the single character alphabets and Table 3 the corresponding values for the bigram alphabets. The block sizes used were 200, 2000 and 10000 for the single characters and 2000, 10000 and 20000 for the bigrams. The compression figures are given in bits per character and the time is measured in milliseconds.

As can be seen, there is a significant loss, on our data, in compression efficiency, when using non-sorted frequencies. The block size seems not to have an impact on the compression. For the bigrams, there is also no difference between the forgetful variants and that using all the preceding data blocks, but for the smaller single character alphabets, the compression using only the information of the few last blocks is marginally better on the French text, and worse on the English one. This can be explained by the different nature of the texts: The English Bible is one homogeneous entity, and its partition into blocks is purely artificial. We may thus expect that using more global statistics will yield better compression performance. The French text, on the other hand, consists of many independent queries and their answers, covering a very large variety of topics. Using the distribution of one block to compress a subsequent one may thus not always yield good results, so a variant which is able to "forget" a part of what it has seen, may be advantageous in this case.

The loss in compression is compensated by savings in sorting time. These savings are more pronounced for the larger bigram alphabets, but also noticeable for the character alphabets. The time is increasing with the size of the blocks, because a larger block gives more possibilities for a larger variability of the frequencies. The exception here is for the bigrams of the French text: the alphabet in this case is so large, that the block size has only a minor impact on the processing time. On the other hand, it is in this case that the savings using partial order are the most significant.

4 Relevance of partial sort to other compression schemes

We check in this section whether the idea of not fully sorting the frequencies could be applicable to other compression methods.

4.1 Arithmetic coding

In fact, for both encoding and decoding using an arithmetic coder [11], the weights need not be in any specific order, as long as encoder and decoder agree upon the same. This has the advantage for the dynamic variant, that the same order of the elements can be used at each step, for example that induced by the lexicographic order of the elements to be encoded. Partial ordering is thus not relevant here.

4.2 256-ary Huffman codes, (s, c)-dense codes, Fibonacci codes

All these codes can be partitioned into blocks of several codewords having all the same length. For 256-ary Huffman, the codeword lengths are multiples of bytes, so that even for very large alphabets, it is very rare to get codewords longer than 3 or 4 bytes; the same is true for (s, c)-dense codes. It follows that, almost always, all the codewords can be partitioned into 3 or 4 groups, so a full sort is not even necessary. It suffices to partition the weights into these classes, as suggested above, just that the sizes of the blocks of the partition are not equal, but rather derived from the specific code.

For Fibonacci codes [5,8], there are F_n codewords of length $n + 2$, where F_i are Fibonacci numbers, and this set is fixed, just as for (s, c)-codes. The number of blocks here is larger, but even for an alphabet of one million characters, there are no more than 29 blocks, and the partition can be done in 5 iterations.

4.3 Burrows-Wheeler Transform (BWT)

At first sight, partially sorting seems to be relevant to BWT [3], as the method works on a string of length n and applies all the n cyclic rotations on it, yielding an

$n \times n$ matrix which is then lexicographically sorted by rows. The first column of the sorted matrix is thus sorted, but BWT stores the *last* column of the matrix, which together with a pointer to the index of the original string in the matrix lets the file to be recovered. The last column is usually not sorted, but it often is very close to be sorted, which is why it is more compressible than the original string. The BWT uses a move-to-front strategy to exploit this nearly sorted nature of the string to be compressed.

One could think that since the last column is anyway only nearly sorted, then if the initial lexicographic sort of the matrix rows is only partially done, the whole damage would be that the last row will be even less sorted, so we would trade compression efficiency for time savings. However, the reversibility of BWT is based on the fact that the first column is sorted, so a partial sort would invalidate the whole method and not just reduce its performance.

5 Conclusion

We have dealt with the simple idea of not fully sorting the weights used by Huffman's algorithm, expecting some time savings in applications where the sort is a significant part of the encoding process. This may include large alphabets, or using several alphabets like in dynamic applications, or when encoding according to a first order Markov chain. The tests showed that by using partial sorts, the execution time can be reduced at the cost of some loss in compression efficiency.

References

1. Brisaboa, N. R., Fariña, A., Navarro, G., and Esteller, M. F.: *(S,C)-dense coding: an optimized compression code for natural language text databases.* Proc. Symposium on String Processing and Information Retrieval SPIRE'03, 2857 2003, pp. 122–136.
2. Brisaboa, N. R., Iglesias, E. L, Navarro, G., and Paramá, J. R: *An efficient compression code for text databases.* Proc. European Conference on Information Retrieval ECIR'03, 2633 2003, pp. 468–481.
3. Burrows, M. and Wheeler, D. J.: *A block-sorting lossless data compression algorithm.* Technical Report SRC 124, Digital Systems Research Center, 1994.
4. de Moura, E. S., Navarro, G., Ziviani, N., and Baeza-Yates, R.: *Fast and flexible word searching on compressed text.* ACM Trans. on Information Systems, 18 2000, pp. 113–139.
5. Fraenkel, A. S. and Klein, S. T.: *Robust universal complete codes for transmission and compression.* Discrete Applied Mathematics, 64 1996, pp. 31–55.
6. Huffman, D.: *A method for the construction of minimum redundancy codes.* Proc. of the IRE, 40 1952, pp. 1098–1101.
7. Katona, G. H. O. and Nemetz, T. O. H.: *Huffman codes and self-information.* IEEE Trans. on Information Theory, IT–11 1965, pp. 284–292.
8. Klein, S. T. and Kopel Ben-Nissan, M.: *Using Fibonacci compression codes as alternatives to dense codes.* Proc. Data Compression Conference DCC–2008, 2008, pp. 472–481.
9. Van Leeuwen, J.: *On the construction of Huffman trees.* Proc. 3^{rd} ICALP Conference, 1976, pp. 382–410.
10. Véronis, J. and Langlais, P.: *Evaluation of parallel text alignment systems: The ARCADE project.* Parallel Text Processing, J. Véronis, ed., 2000, pp. 369–388.
11. Witten, I. H, Neal, R. M., and Cleary, J. G.: *Arithmetic coding for data compression.* Comm. of the ACM, 30 1987, pp. 520–540.
12. Yao, A. C. C.: *An $O(|E| \log\log |V|)$ algorithm for finding minimum spanning trees.* Inf. Processing Letters, 4 1975, pp. 21–23.

In-place Update of Suffix Array while Recoding Words

Matthias Gallé, Pierre Peterlongo, and François Coste

IRISA / INRIA Rennes Bretagne Atlantique
Campus de Beaulieu, 35042 Rennes Cedex, France
{matthias.galle, pierre.peterlongo, francois.coste}@irisa.fr

Abstract. Motivated by grammatical inference and data compression applications, we propose an algorithm to update a suffix array after the substitution, in the indexed text, of some occurrences of a given word by a new character. Compared to other published index update methods, the problem addressed here may require the modification of a large number of distinct positions over the original text. The proposed algorithm uses the specific internal order of suffix arrays in order to update simultaneously groups of entries, and ensures that only entries to be modified are visited. Experiments confirm a significant execution time speed-up compared to the construction of suffix array from scratch at each step of the application.

Keywords: suffix array, in-place update, dynamic indexing, word-interval

1 Motivation

In this paper, we propose an algorithm to update efficiently a suffix array, after substituting a word by a new character in the indexed text. This work is motivated by grammatical inference or grammar-based compression, along the lines initiated by SEQUITUR [21] in the framework formalized by Kieffer and Yang [11,12]. The goal is to infer a grammar G which generates only a given (long) sequence s in order to discover the structure that underlies the sequence, or simply, to compress the sequence thanks to a code based on the grammar. Learning and compression are often subtly intertwined (as for instance in the Occam's razor principle): in both cases the grammar is expected to be as small as possible. Kieffer and Yang introduced the definition of irreducible grammars and proposed several reduction rules allowing to transform a reducible grammar into an irreducible one, giving rise to efficient universal compression algorithms [11]. The sketch of these algorithms is to begin with a unique $S \rightarrow s$ rule generating the whole given sequence and essentially, to reduce iteratively the size of the grammar at each step by: 1) choosing a repeated pattern, 2) replacing the occurrences of the repeat by a new (non-terminal) symbol and 3) adding a new rewriting rule from this new symbol into the repeated pattern. For instance, the grammar $S \rightarrow uRvRw$, where u, v, w and R are substrings, and the length of R is strictly bigger than one, can be reduced at the first step into the grammar with two rules $S \rightarrow uAvAw$ and $A \rightarrow R$, where A is a new non-terminal symbol. At the following step, another repeated pattern, including eventually the new inserted symbol A, is selected and factorized by the introduction of a third rule, and so forth for the next steps. As a result, the algorithm returns a compact grammar which can be used to get a hierarchical point of view on the structure of the sequence or which can be encoded in order to get a better compression than by encoding directly the sequence.

Matthias Gallé, Pierre Peterlongo, François Coste: In-place Update of Suffix Array while Recoding Words, pp. 54–67.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1

© Czech Technical University in Prague, Czech Republic

Algorithms of this kind are thus mainly based on the successive detection of repeats. They differ mostly in the order in which repeats are factorized. In SEQUITUR [21] and its variant [12], each repeat is replaced as soon as it is detected by a left to right scan of the sequence. More elaborate strategies for choosing the repeat to replace have been proposed. Kieffer and Yang proposed to replace longest matching substring [11]. Apostolico and Lonardi [3] proposed in their algorithm OFF-LINE to choose the substring yielding the best compression in a steepest-descent fashion. Efficient implementation of an elaborate choice of repeat often requires to use data structures from the suffix tree family. These index structures are well suited for efficient computations on repeats but they have to be built at initialization, and then updated at each step of the algorithm with respect to sequence modifications. Yet, as pointed out by Apostolico and Lonardi, most of the published work on updating a suffix tree, or – more general – on dynamic indexing problem – [20,7,17,8,6,23,5] focuses on localized modifications of the string and does not seem appropriate for replacing efficiently *more than one* occurrence of a given substring. Thus, index structures have usually to be built from scratch at each step of the algorithm. To our knowledge, only GTAC [16], an algorithm applied successfully on genomic sequences by Lanctot, Li and Yang, updates a suffix tree data structure after the deletion of all occurrences of a word. However, its updating scheme is specific to the longest matching substrings and seems difficult to adapt to other strategies.

In this paper, we propose a solution to the problem of updating efficiently an index structure while replacing some non-overlapping occurrences of a word of the indexed text by a new symbol. The first originality of our approach relies on the use of enhanced suffix arrays instead of suffix trees. Enhanced suffix arrays are known to be equivalent to suffix trees while being more space efficient [1]. They can be built in linear time [10,13,15] but non-linear algorithms [18,19] are usually more efficient for practical applications. A simple way of updating suffix array (instead of enhanced suffix array, thus without the same efficiency objective) by lazy bubble sort has been used in [22]. We propose here, to take advantage of the internal order offered by enhanced suffix arrays, to handle simultaneously groups of entries. This enables us to implement efficiently an update procedure for grammatical inference or grammar-based compression algorithm, choosing at each step a repeated substring, and replacing some or all of its occurrences by a new symbol.

2 Algorithm

2.1 Definitions and notations

A *sequence* is a concatenation of zero or more characters from an alphabet Σ. The number of characters in Σ is denoted by $|\Sigma|$. A sequence s of length n on Σ is represented by $s[0]s[1]\cdots s[n-1]$, where $s[i] \in \Sigma\ \forall 0 \leq i < n$. We denote by $s[i,j] (j \geq i)$ the sequence $s[i]s[i+1]\cdots s[j]$ of s (if $j < i$ then $s[i,j] = \epsilon$, the empty string). In this case, we say that the sequence $s[i,j]$ occurs at position i in s. Its length, denoted by $|s[i,j]|$, is equal to $j-i+1$. Furthermore, the sequence $s[0,j]$ $(0 \leq j < n)$, also denoted by $s[..j]$, is called a prefix of s, and symmetrically, $s[i,n-1]$ $(0 \leq i < n)$, also denoted by $s[i..]$, is called a suffix of s.

Definition 1 (Suffix Array). *Consider a sequence s of length n over an alphabet Σ with a lexicographic order $\prec$ extensible to Σ^*. Let $\tilde{s} = s\$$, with a special character \$ not contained in Σ, lexicographically smaller than every element of Σ.*

The suffix array, denoted by sa, is a permutation of $[0..n]$ *such that:*

$$\forall\ i,\ 0 < i \leq n\ :\ \tilde{s}[sa[i-1]..] \prec \tilde{s}[sa[i]..]$$

Usually, the suffix array is used conjointly with an array called lcp, that gives the longest common prefix length between two suffixes whose starting positions are adjacent in sa. Formally,

$$\begin{aligned} &lcp[0] = 0,\\ \textit{and}\ \forall\ i \in [1,n]\ :\ &lcp[i] = k\ \textit{such that}\\ &\tilde{s}[sa[i-1]..][0,k-1] = \tilde{s}[sa[i]..][0,k-1]\ \textit{and}\ \tilde{s}[sa[i-1]..][k] \neq \tilde{s}[sa[i]..][k]. \end{aligned}$$

Eventually, a third array called isa (for inverse suffix array) may be used conjointly with sa and lcp. This array gives, for a position p in s, the index i in sa such that $sa[i] = p$*. Thus* $sa[isa[p]] = p$*.*

The association of sa, lcp and isa arrays is called an *Enhanced Suffix Array* (*ESA*). An *ESA* enables $O(n)$ computation of occurrences of different kinds of repeats (repeats, maximal repeats [9,14] or super maximal repeats [1,9]).

In this paper, we propose to update an *ESA*, deleting and moving some of its indexes and keeping lcp consistent. In order to avoid shifting set of entries, we link consecutive entries using two additional arrays called *next* and *prev*. Thus, $next[i]$ (resp. $prev[i]$) gives the index of the next (resp. previous) valid entry in the *ESA*. Initially, $next[i] = i+1$ and $prev[i+1] = i$. We call the set *ESA* plus *next* and *prev* arrays the ESA_{DL} for *Double Linked Enhanced Suffix Array.*

It is worth noticing that an ESA_{DL} has not exactly the same properties as an *ESA*. Indeed, going from an entry i to entry $i+j$ may be done in constant time on an *ESA*, while this operation in an ESA_{DL} requires $O(j)$ time, as the *next* array has to be used j times.

Anyway, an ESA_{DL} still allows the detection of repeats (general repeats, maximal repeats or super maximal repeats) in linear time, because the algorithms used advance one by one over the arrays like most of the algorithm over *ESA* (a notable exception is the algorithm searching for a subsequence proposed in [25]).

We propose an *in-place* solution, where we always work with the same arrays and only update the values of their fields. Moreover, during the whole process, we modify only the *prev*, *next* and *lcp* arrays. Arrays *sa* and *isa* remain unchanged. This approach forces to extend the in-place behavior to the sequence: we also add two arrays to imitate a double linked list over the sequence.

The j^{th} position after position i, is denoted by $i \oplus j$. We compute $i \oplus j$ using links between sequence positions, indicating for each position its successor. Similarly $i \ominus j$ points to the j^{th} position before i. We define that, if $i \oplus j$ (respectively $i \ominus j$) is out of range, then $i \oplus j = n+1$ (respectively $i \ominus j = -1$).

The left context tree. One of the most useful characteristic of a suffix array is that all indexes corresponding to suffixes starting with the same word correspond to an adjacent block. We define here the corresponding concept of word interval. Based on this, we will define the *left context tree* of a word ω where the nodes correspond to a left context of ω.

An ω-interval is the set $\{k\ :\ \exists\ell,\ k = isa[\ell] \wedge \tilde{s}[\ell..\ell+|\omega|-1] = \omega\}$. This can also be denoted as an $[i..j]$-interval, where i and j are respectively the lowest and highest indices of an ω-interval. Let us note that different words can share the same interval. More precisely, any pair of words ω and $\omega\alpha$ share the same interval if each occurrence of ω is followed by α.

This definition is thus slightly more general than the definition of ω-interval given by Abouelhoda, Kurtz and Ohlebusch [1], since we also define ω-interval for words leading to implicit nodes of a compact suffix tree, and not only to internal nodes.

The *left context tree of* ω ($\omega \in \Sigma^*$) for a sequence $\tilde{s}$ is an implicit tree whose nodes are v-intervals ($v \in \Sigma^*$) such that:

– the root is the ω-interval
– for each v-interval node corresponding to a non-empty interval, its children are all the av-intervals, for all $a \in \Sigma$
– the leaves are empty intervals

Given the *isa* array, it is easy to obtain the parent of a node. Let $[i..j]$ be an av-interval node. Given $k \in [i..j]$, $isa[sa[k]+1]$ is an index belonging to the v-interval. Inversely, $isa[sa[k]-1]$ belongs to one of the child interval. The exact child depends on the character at $\tilde{s}[sa[k]-1]$. We introduce the *successor* and *predecessor* notations:

$$successor(i) = \begin{cases} isa[sa[i] \oplus 1] & \text{if } sa[i] \oplus 1 \neq n+1 \\ n+1 & \text{otherwise,} \end{cases}$$

and

$$predecessor(i) = \begin{cases} isa[sa[i] \ominus 1] & \text{if } sa[i] \neq 0 \\ -1 & \text{otherwise.} \end{cases}$$

One may remark that $predecessor(i)$ is the equivalent of the "*suffix link*" in a suffix tree [26].

The problem that an *ESA* update algorithm must face is that the changes over the occurrences of a word ω not only affect the ω-interval, but also some of the $v\omega$-intervals ($v \in \Sigma^*$). The core of our algorithm is based on moving $v\omega$-interval in constant time, using the two following properties implied by the internal order of suffix arrays:

Proposition 2. *Let $[i..j]$ be an v-interval ($v \in \Sigma^*$), and $k_1, k_2 \in [i..j]$ with $k_1 > k_2$ and such that $predecessor(k_1)$ and $predecessor(k_2)$ belong to the same αv-interval ($\alpha \in \Sigma$). Then $predecessor(k_1) > predecessor(k_2)$.*

Proposition 3. *With $i < j$, the longest common prefix between $\tilde{s}[sa[i]..]$ and $\tilde{s}[sa[j]..]$ is $min_{k\in[next[i],j]}(lcp[k])$.*

In this paper, we consider that the grammatical inference or grammar based compression algorithm proceeds by steps. At each step, the alphabet grows because of the introduction of a new character: Σ_k will denote the alphabet in step k. In each, of this steps, the algorithm **i)** finds a repeat $\mathcal{R}_k$ in a sequence $\tilde{s}^{(k)}$ defined on the alphabet Σ_k and returns a list $\mathcal{O}_k$ of non-overlapping occurrences of $\mathcal{R}_k$ **ii)** updates the sequence $\tilde{s}^{(k)}$ and its associated ESA_{DL} replacing the given occurrences of $\mathcal{R}_k$ by a single new character $\mathcal{C}_k$, thus defining a new alphabet $\Sigma_{k+1} = \Sigma_k \cup \{\mathcal{C}_k\}$. The modified sequence is then called $\tilde{s}^{(k+1)}$. The whole iterative process stops either if no more repeat is found in the sequence or after a fixed number of iterations.

Our contribution focuses on updating the ESA_{DL}, at each step k of this algorithm (part **ii**).

In the next sections, we describe how to perform the three tasks needed for updating an ESA_{DL} at each step k: **1)** delete entries of suffixes starting inside an $\mathcal{R}_k$ occurrence; **2)** move entries with respect to the new alphabetic order; and **3)** update *lcp* array with respect to recoding occurrences of $\mathcal{R}_k$ by one single character. Note that a few values of the *lcp* array are also modified during part 1 and 2, but only as a consequence of deletions and moves.

2.2 Delete entries of suffixes occurring inside $\mathcal{R}_k$ substituted occurrences

entry	*lcp*	suffix
$prev[j]$	4	$ATAC\ldots$
j	2	$ATGA\ldots$
$next[j]$	3 **2**	$ATGT\ldots$

Figure 1. Deletion of entry j.

By replacing the word $\mathcal{R}_k$ by a single letter, the sequence is compressed and so is its ESA_{DL}: consequently, any suffix of sequence $\tilde{s}^{(k)}$ appearing inside an $\mathcal{R}_k$ substituted occurrence must be deleted. Thus for i in $\mathcal{O}_k$ and for ℓ in $[1, |\mathcal{R}_k| - 1]$, suffix $\tilde{s}^{(k)}[i \oplus \ell..]$ and the associated index in the suffix array $j = isa[i \oplus \ell]$ have to be removed. We simulated this deletion by *jumping over it* by setting *next* and *prev* arrays to their previous and next index: $next[prev[j]] \leftarrow next[j]$ and $prev[next[j]] \leftarrow prev[j]$. Furthermore, the *lcp* value of the index following j ($lcp[next[j]]$) has to be modified according to the deletion of index j. As a consequence of proposition 3, after the deletion of index j, the longest common prefix of entry $next[j]$ is equal to the minimal longest common prefix value of entries j and $next[j]$.

An example is shown in Figure 1 where the deletion of entry j affects the $lcp[next[j]]$ that now should contain the length of longest common prefix between $ATGT$ and $ATAC$ which is 2, equal to the longest common prefix of $ATGT$, $ATGA$ and $ATAC$.

Algorithm 1 presents the procedure for deleting indexes. The notation END refers to the last index of the suffix array ($prev[n + 1]$).

Algorithm 1 Delete entries at step k, replacing $\mathcal{R}_k$ by $\mathcal{C}_k$

$delete\_entries\{ESA^{(k)}_{DL}, \mathcal{R}_k, \mathcal{O}_k\}$

for $i \in \mathcal{O}_k$ **do**
  for $\ell \in [1, |\mathcal{R}_k| - 1]$ **do**
    $j \leftarrow isa[i \oplus \ell]$
    if $next[j] \neq END$ **then**
      $lcp[next[j]] \leftarrow min(lcp[j], lcp[next[j]])$
    end if
    $next[prev[j]] = next[j]$
    $prev[next[j]] = prev[j]$
  end for
end for

2.3 Move entries, with respect to new alphabetic order

After replacing the word $\mathcal{R}_k$ by the new character $\mathcal{C}_k$, some ESA_{DL} lines may be misplaced with respect to the chosen order of $\mathcal{C}_k$ in Σ_{k+1}.

Entries in the $\mathcal{R}_k$-interval are potentially misplaced. Moreover, for $v \in \Sigma_k^*$, index entries inside an $v\mathcal{R}_k$-interval are misplaced if the substitution of $\mathcal{R}_k$ into $\mathcal{C}_k$ affects their lexicographical order with respect to the previous and next index over the suffix array. Thus, lines belonging to node-intervals of the left-context tree of $\mathcal{R}_k$ may have to be moved.

In our approach, we decided to give to $\mathcal{C}_k$ the largest rank in the lexicographic order of the alphabet Σ_k, *i.e.* $\forall\, \alpha \in \Sigma_k \,:\, \alpha \prec \mathcal{C}_k$.

With respect to this arbitrary choice, the $\mathcal{R}_k$-interval is moved after the last entry of the suffix array. Furthermore, for any $v \in \Sigma_k^*$, the $v\mathcal{R}_k$-interval is moved after the last entry of the v-interval.

If an $v\mathcal{R}_k$-interval is already at the end of the v-interval (it is naturally well ordered), for any $v' \in \Sigma_k^*$, the $v'v\mathcal{R}_k$-interval is also at the end of the $v'v\mathcal{R}_k$-interval and has not to be moved.

Algorithm 2 Restore consistency of suffix array order

$update\_order\{ESA_{DL}^{(k)}, \mathcal{R}_k, \mathcal{O}_k, i_{start}, depth, move\}$

if Couple $(i_{start}l, depth)$ already treated during another recursion call **then**
    End procedure
end if
$i \leftarrow i_{start}$
while $i \neq END \wedge lcp[next[i]] \geq depth + |\mathcal{R}_k|$ **do**
    $i \leftarrow next[i]$
end while
$i_{end} \leftarrow i$
$minLCP \leftarrow min_{j\in[i_{start}, i_{end}]} lcp[j]$
if $move$ **then**
    while $i \neq END \wedge lcp[next[i]] \geq depth$ **do**
        $i \leftarrow next[i]$
    end while
end if
$i_{dest} \leftarrow i$
if $i_{end} \neq i_{dest}$ **then**
    $lcp[next[i_{end}]] \leftarrow min(lcp[next[i_{end}]], minLCP)$
    $lcp[i_{start}] \leftarrow depth$
    if $i_{start} = i_{first} \wedge depth \neq 0$ **then**
        $i_{first} \leftarrow next[i_{end}]$
    end if
    move_group$(i_{start}, i_{end}, i_{dest})$
else
    $lcp[i_{start}] \leftarrow min(lcp[i_{start}, depth)$
    $move \leftarrow false$
end if
$i \leftarrow i_{start}$
while $i \neq next[i_{end}]$ **do**
    $newdepth \leftarrow depth+$ (**if** $predecessor(i) \in \mathcal{O}_k$ **then** len **else** 1)
    if $move \vee (sa[prev[predecessor(i)]] > newdepth \wedge sa[prev[predecessor(i)]] \oplus newdepth \in \mathcal{O}_k)$ **then**
        $update\_order(ESA_{DL}^{(k)}, \mathcal{R}_k, \mathcal{O}_k, predecessor(i), newdepth, i_{dest} \neq i_{end})$
    end if
    $i \leftarrow next[i]$
end while

Based on this property, our algorithm uses a recursive approach in order to move groups. The recursion starts on the initial $\mathcal{R}_k$-interval. During recursion, if the group of an $v\mathcal{R}_k$-interval is moved, the recursion continues on groups of $\alpha v\mathcal{R}_k - intervals$, with $\alpha \;\in\; \Sigma_k$.

From a theoretical point of view, the algorithm starts on the root of the left-context tree of $\mathcal{R}_k$ and if the group corresponding to the interval of the node is moved, it recursively treats its children in a breadth first traversal (a FIFO is used).

In practice, the recursion on an $v\mathcal{R}_k$-interval works as follow:

1. detects the end position of the $v\mathcal{R}_k$-interval,
2. detects the end position of the v-interval,
3. if necessary:
   3.a. moves the group to the end position of the v-interval,
   3.b. call the recursion on predecessors of entries of the group.

During a call on predecessor of an entry of the group, either this is the first time the matched group is called and by construction the call is done on its first element, or the group was already treated, and the recursion stops.

The algorithm for this step is shown in algorithm 2. This recursion function receives three parameters besides the data structures: the starting position of the group, the current depth over the left-context and a boolean flag (see below).

At first, the end of the $v\mathcal{R}_k$-interval is found (lines 2, 2 and 2).

l	sa	lcp	suffix
0	8	0	\$
~~1~~	~~1~~	~~0~~	~~$AAGAA\dots$~~
~~2~~	~~4~~	~~3~~	~~$AAGC\$$~~
3	2	0	$A\boxed{GA}AG\dots$
4	5	$\cancel{2}$ 0	$AGC\$$
5	7	0	$C\$$
6	0	0	$\boxed{GA}A\boxed{GA}\dots$
7	3	$\cancel{4}$ 0	$\boxed{GA}AGC\dots$
8	6	$\cancel{1}$ 0	$GC\$$

Figure 2. Moves induced by substituting GA by $\mathcal{C}_1$.

This is done from the first element of the interval, following the *next* array while the visited entry corresponds to a suffix starting with $v\mathcal{R}_k$ ($lcp \geq |v| + |\mathcal{R}_k|$). After finding the extremes of the group, the destination index of this group according to the chosen order for the new character is found (lines 2, 2 and 2). This is done by finding the end of the v-interval in the same way ($lcp \geq |v|$).

Moving now the group to its new position is simple and is done in constant time. Thanks to the well-ordered property of the suffix array, the whole interval is moved by changing only the delimiting positions. Let $i_{start}, i_{end}, i_{dest}$ be respectively the starting and ending positions of the $v\mathcal{R}_k$-interval, and the last position of the v-interval. Move the group $[i_{start}, i_{end}]$ to the position after i_{dest} is simply done by jumping over the group and *inserting* it into i_{dest} and $next[i_{dest}]$. See the algorithm 3 for implementation details.

Two longest common prefix values are modified as a consequence of the deletion of the group and its insertion:

1. $lcp[next[i_{end}]]$: contains the value of the length of the longest common prefix between $prev[i_{start}]$ and $next[i_{end}]$, which according to proposition 3, is the minimum of the lcp values of the group and itself
2. $lcp[i_{start}]$: we assign to it the value of *depth*, that is the correct value over $\tilde{s}_{k+1}$. This serves also to set a stop-point for future recursions calls (see below).

As i_{first} points to the first line over the suffix array that contains a selected repetition, we also update i_{first} (line 2) if this line is moved.

Figure 2 shows the ESA_{DL} of sequence $GAAGAAGC$, where $\mathcal{R}_1 = GA$ is substituted by $\mathcal{C}_1$. One remarks that the initial interval of suffixes starting with GA (indexes 6 and 7) is moved as well as suffix starting with AGA (index 3). Note also that suffix starting with $GAAGA$ has to be moved with respect to suffix $GAAGC$.

Algorithm 3 Move the group $[i_{start}, i_{end}]$ after the position i_{dest}

$move\_group\{ESA_{DL}^{(k)}, \mathcal{R}_k, \mathcal{O}_k, i_{start}, i_{end}, i_{dest}\}$

$next[prev[i_{start}]] = next[i_{end}]$
$prev[next[i_{end}]] = prev[i_{start}]$
$next[i_{end}] = next[i_{dest}]$
$prev[next[i_{dest}]] = i_{end}$
$next[i_{dest}] = start$
$prev[i_{start}] = i_{dest}$

A special case Once an interval is treated, the recursion continues either if the current group was moved, or in the special case described in what follows.

Consider for instance the following case, where the substituted repeat is TA.

i C**TA**TT**TA**C...
i+1 C**TA**TT**TA**G...
i+2 C**TA**T**TA**...,

and suppose that the TTA-interval containing the index $isa[sa[i+2] \oplus 3]$ (the underlined suffix in the figure) was already at its right position and therefore has not to be moved. So, its children in the left-context tree are not considered for future moves, and as a consequence, neither is index $i+2$. Supposing that we cut the recursion here, that means that when treating the $CTATT$-interval, $lcp[i+2] = 5$. This interval ends at the index $i+1$, but because we use the *lcp* array to detect it, we also consider index $i+2$ as part of the $CTATT$-interval.

To resolve this special case, the recursion continues even when the current interval was not moved. In this case, it will never be necessary to move an interval, but maybe update some *lcp* values to set *stop-points* for future recursion calls.

This is the reason for introducing the last parameter in algorithm 2 (the boolean flag *move*). It differentiates the normal case (when it is necessary to detect the destination index and move the interval) from the case in which the current interval is considered only to set a *stop-point* at the first index of the interval. The recursion continues in both cases.

Filtering non substituted $\boldsymbol{\mathcal{R}_k}$ occurrences Among each $v\mathcal{R}_k$-interval, suffixes starting with $v\mathcal{R}_k$ where $\mathcal{R}_k$ is not substituted (whose position does not belong to $\mathcal{O}_k$) may occur. The associated entries in the ESA_{DL} should not be moved with the $v\mathcal{R}_k$-interval. Thus, before to apply the recursive procedure previously exposed, a straightforward *filtering step* is applied. During the recursion, each line i of each group is first checked in order to detect if it corresponds to an entry of a selected occurrence ($sa[i] \oplus depth \in \mathcal{O}_k$). Once detected a non-selected occurrence, we move it to the beginning of the group (before i_{start}). As previously mentioned, this also involves modifications of the *lcp* array for maintaining its consistency.

2.4 Update lcp values after the substitution of $\boldsymbol{\mathcal{R}_k}$ occurrences to a single character

The substitution of any occurrence of $\mathcal{R}_k$ of length $|\mathcal{R}_k| \geq 2$ by $\mathcal{C}_k$ of length 1 involves the modification of the length of all common prefixes involving such an occurrence.

In the previous step, it was easy to update the *lcp* values of the limits of the intervals while they were moved. In this step, we update the *lcp* values of the internal position of the intervals.

For this, we traverse the left-context tree of $\mathcal{R}_k$. Contrary to the moving step, where it was possible to move one line several times, in this step we update each *lcp* index only once. To do this, we recalculate all the *lcp* values for the root ($\mathcal{R}_k$-interval) and use this information to update the *lcp* of the other intervals.

As a consequence of propositions 2 and 3, the *lcp* between two indexes of the same interval-node is simply one plus the *lcp* between their successor indexes belonging to the parent interval-node:

Let i, j belong to the same aw-interval and let us assume that $i > j$.

Then $lcp(\tilde{s}[sa[i]..], \tilde{s}[sa[j]..]) = min_{\ell \in [next[successor(i)], successor(j)]} lcp[\ell]$

With this inductive approach, it is sufficient to re-calculate the *lcp* of only the first interval (the root of the left-context tree). This is straightforward (see algorithm 4).

Algorithm 4 Calculate the value of the *lcp* for index i

recalculate_lcp$\{ESA_{DL}, i\}$

$lcp[i] \leftarrow 0$
if $prev[i] \geq 0$ **then**
  $i \leftarrow sa[i]$
  $j \leftarrow sa[prev[i]]$
  while $i < n \wedge j < n \wedge s[i] = s[j]$ **do**
    $i \leftarrow i \oplus 1$
    $j \leftarrow j \oplus 1$
    $lcp[i] \leftarrow lcp[i] + 1$
  end while
end if

During the iterative call, if an index already treated appears, it is skipped. Indeed, its *lcp* value is then up-to-date. The pseudo-code for this step is exposed in algorithm 5.

Algorithm 5 Update *lcp* of step k

update_lcp$\{ESA^{(k)}_{DL}, \mathcal{R}_k, \mathcal{O}_k\}$

$q \leftarrow queue()$
for $i \in \mathcal{O}_k$ **do**
  $recalculate\_lcp(ESA^{(k)}_{DL}, isa[i])$
  $q.push((predecessor(isa[i]), 1))$
end for
while not *q.empty()* **do**
  $(i, depth) \leftarrow q.top$
  q.pop
  if $i \geq 0 \wedge lcp[i]$ not already updated $\wedge\ lcp[i] \geq depth$ **then**
    $lcp[i] \leftarrow (min_{j \in [next[successor(prev[i])], successor(i)]} lcp[j]) + 1$
    $q.push((predecessor(i), depth + 1))$
  end if
end while

Because in each step we use the value of all the lines of the previous group, we traverse once again the left context tree in a breadth-first order.

3 Efficiency

The space complexity is in $O(n)$. The ESA_{DL} structure needs to complete the *ESA* with two arrays of length n. During the execution, a queue of length $O(n)$, plus an array of length n are used to check in constant time whether a couple $(i, depth)$ was already used.

The worst case time complexity of the update algorithm is bounded by $O(n^2)$. This case is reached while replacing for instance AA occurrences in an ESA_{DL} indexing the text A^nT. A better bound on time complexity could be obtained by considering amortized complexity, but it will still be unlikely to be better than the $O(n)$ complexity required for building the suffix array from scratch. Nevertheless, the algorithms building suffix arrays that currently perform best in practical cases, are not the linear ones (see [24] for a complete description of the different suffix array construction algorithms and their strengths). We propose in this section to evaluate the practical efficiency of our algorithm.

A prototype implementing the proposed algorithm has been developed using the C++ language. It is available at `http://www.irisa.fr/symbiose/people/galle/update_sarray/`. It has been tested on different types of text. For the sake of briefness, in this paper we only report the results on the following classical corpora from the literature:

– the standard and large Canterbury corpus (`http://corpus.canterbury.ac.nz/`, [4]),
– the Purdue corpus (`http://www.cs.ucr.edu/~stelo/Offline/`, [2])

Similar tests on other corpora can be found on our internet site.

We compared the execution times of our algorithm with the linear time suffix array construction algorithm proposed by Kärkkäinen and Sanders [10], and non-linear algorithm of Larsson and Sadakane [18] that is in practice faster. Both source codes were retrieved from the Internet sites specified in the associated articles. Note that Kärkkäinen and Sanders' code "strives for conciseness rather than for speed" [10]. The Manzini and Ferragina's algorithm [19], doesn't fulfill our requirement of variable alphabet size, it was then not used for our experiments. The tests were executed on 1 GHz AMD Opteron processors with 4 GB of memory.

First, to have an idea of the complexity of the algorithm, we studied how the length of the sequence influences the execution time of the algorithm. From the large Calgary corpus, we extracted sequences of different length by considering successively bigger (by steps of 100 characters) prefixes of the sequences. On each extracted sequence, we performed 250 iterations of selecting a random repeat, replacing it over the sequence by a new character and updating the associated suffix array. Time (user + system time) required for updating the suffix array was reported, averaged over 5 different runs corresponding to 5 different random seeds. The same experiments, replacing the update algorithm by the "from scratch" construction algorithms of the suffix array by Kärkkäinen and Sanders (*K & S*) and Larsson and Sadakane (*L & S*) have been performed. The plots, shown in figure 3, confirm that the execution time of our updating algorithm is not directly correlated to the length of the sequence, and is significantly smaller than the execution time required by reconstruction "from scratch" algorithms, especially when the length of the sequence increases.

We present a more exhaustive evaluation and comparison on all the corpora using different strategies for the selection of the repeated word. In each test we performed 500 iterations of selecting a repeat, replacing it over the sequence and updating (or building from scratch) the associated suffix array. The different strategies for the selection of the repeat were:

– take a random one (using the same seed for the random number generator),
– take the longest,

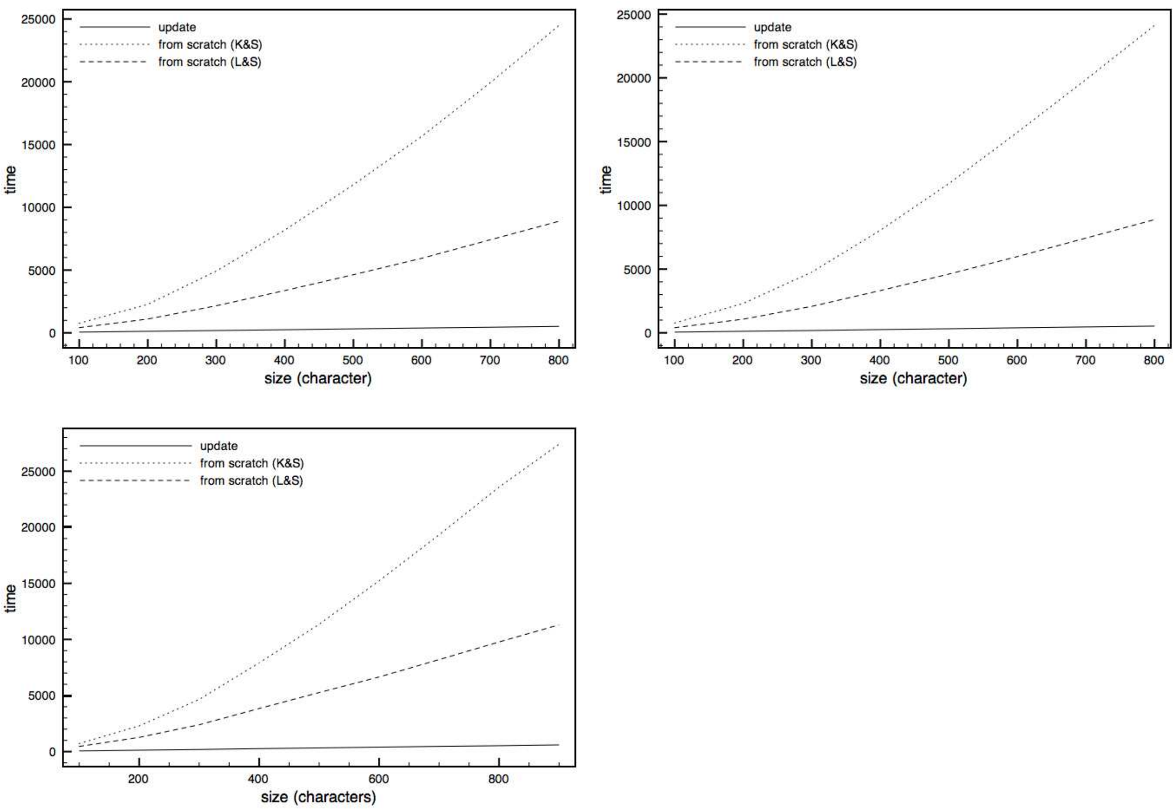


Figure 3. Large corpus: *bible.txt*, *world192.txt* and *E.coli*. Times are given in hundredth of seconds

– take the one that covers the maximal number of positions[1].

Results are given in figure 4 (page 67). For each selection strategy, we report time (user + system time) spent in updating ESA_{DL} with our algorithm (column *update*), and time spent in building *ESA* from scratch at each iteration with the linear algorithm from Kärkkäinen and Sanders (column *K & S*) and the algorithm from Larsson and Sadakane (column *L & S*). For easier comparison, the ratios of the time spent by each of the two "from scratch" algorithms over the update algorithm are also given.

Some of the files (*fields.c, grammar.lsp* and *xargs.1*) are too small to draw significant conclusions, but results are shown here for the sake of completeness. On the other files, results show that a significant speedup is usually achieved by using our algorithm. The main exceptions are the $Spor\_All\_2x.fasta$ files (an artificial file obtained by concatenating $Spor\_All.fasta$ with itself) from the Purdue corpus, and the *ptt*5 file from the Canterbury corpus (a fax image with very long zones of the same byte). One can also remark that the ratio is less favorable when the repeat to replace is chosen according to the maximal compression strategy. On the one hand, in each iteration the resulting sequence is smaller and the suffix array creation from scratch for this sequence faster. On the other hand, there are more positions affected by the substitution and this affects the update algorithm.

[1] Maximisation of $(|\mathcal{O}_k| - 1) * (|w| - 1) - 1$, corresponding to a maximal compression approach.

These cases allow us to illustrate an intrinsic limit of the update approach when the length of the sequence is highly reduced by recoding: when the number of positions to update is larger than the number of positions in the resulting sequence, it may be worth adopting the "from scratch" construction algorithm (let us remark that the best algorithm to use can vary along the iterations). A solution to handle these extreme cases, would be to design a criterion on the repeat and its coverage to automatically choose the best algorithm to use (eventually at each iteration).

4 Conclusion and future work

We introduced in this paper an approach allowing to keep up-to-date an enhanced suffix array with respect to the substitution of some of the occurrences of a word in the indexed text We didn't consider singular insertions or deletions, but simultaneuos substitution.This is of particular interest for grammatical inference or grammar based compression methods which are using these data structures and are performing iteratively a large number of such substitutions.

Our approach uses the specific internal order of suffix arrays to update simultaneously groups of adjacent entries and ensures that only entries to be modified are visited. This specific property of the suffix arrays allows to design an efficient update procedure which has been implemented and tested on classical corpora. The experimentation confirms that, in regard to the direct method reconstructing the suffix array, our approach enables significant speed-up of the execution time.

However, in some cases, the update method is less efficient than building the enhanced suffix array from scratch. Intuitively, when the number of lines to change is larger than the number of lines in the new suffix array, a reconstruction algorithm is likely to be more efficient than an update approach. In order to be even more efficient, a criterion allowing to decide automatically which algorithm to use could be designed. This would require a finer complexity analysis of the update algorithm, but also of the chosen building algorithm, in order to identify easy-to-compute key parameters involved in the execution time complexity.

Of course, the question of the existence of a practical efficient $O(n)$ algorithm remains open. But the results on the construction of suffix arrays suggest that a better way of improvement could be the design of other practical update algorithms. Finally, these results have been obtained by using a suffix array. It would be interesting to study how easily this approach can be adapted to suffix trees and how much it depends on the suffix array specific properties.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch: *Replacing suffix trees with enhanced suffix arrays*. J. Discrete Algorithms, 2(1) 2004, pp. 53–86.
2. A. Apostolico and S. Lonardi: *Compression of biological sequences by greedy off-line textual substitution*, in Proc. DCC, 28-30 March 2000, pp. 143–152.
3. A. Apostolico and S. Lonardi: *Off-line compression by greedy textual substitution*, in Proc. IEEE, vol. 88, Nov. 2000, pp. 1733–1744.
4. R. Arnold and T. Bell: *A corpus for the evaluation of lossless compression algorithms*, in Proc. Conference on Data Compression, Washington, DC, USA, 1997, IEEE Computer Society, p. 201.
5. H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane: *Compressed indexes for dynamic text collections*. ACM Transactions on Algorithms, 3(2) May 2007.

6. P. Ferragina, R. Grossi, and M. Montangero: *On updating suffix tree labels.* Theor. Comp. Science, 201(1-2) 1998, pp. 249–262.
7. E. Fiala and D. H. Greene: *Data compression with finite windows.* Comm. ACM, 32(4) 1989, pp. 490–505.
8. M. Gu, M. Farach, and R. Beigel: *An efficient algorithm for dynamic text indexing*, in Proc. the ACM-SODA, 1994, pp. 697–704.
9. D. Gusfield: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, Jan. 1997.
10. J. Kärkkäinen and P. Sanders: *Simple linear work suffix array construction*, in Proc. ICALP, Springer, 2003.
11. J. Kieffer and E.-H. Yang: *Grammar-based codes: A new class of universal lossless source codes.* IEEE TIT, 46 2000.
12. J. Kieffer and E.-H. Yang: *Grammar-based codes: a new class of universal lossless source codes.* IEEE TIT, 46 2000.
13. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*, in Proc. CPM, vol. 2676, 2003, pp. 200–210.
14. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in Proc. IEEE FOCS, New York, USA, 1999, IEEE Computer Society Press, pp. 596–604.
15. K. D. Kyue, S. J. Seop, P. Heejin, and P. Kunsoo: *Linear time construction of suffix arrays*, in Proc. Combinatorial Pattern Matching, vol. 2676, 2003, pp. 186–2003.
16. J. K. Lanctot, M. Li, and E.-H. Yang: *Estimating dna sequence entropy*, in Proc. ACM-SODA, 2000, pp. 409–418.
17. N. J. Larsson: *Extended application of suffix trees to data compression*, in Proc. DCC, 1996, pp. 190–199.
18. N. J. Larsson and K. Sadakane: *Faster suffix sorting*, Tech. Rep. LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden, May 1999.
19. G. Manzini and P. Ferragina: *Engineering a lightweight suffix array construction algorithm.* Algorithmica, 40(1) 2004, pp. 33–50.
20. E. M. McCreight: *A space-economical suffix tree construction algorithm.* J. ACM, 23(2) 1976, pp. 262–272.
21. C. Nevill-Manning and I. Witten: *Identifying hierarchical structure in sequences: A linear-time algorithm.* J. AI Research, 7 1997, pp. 67–82.
22. C. Nevill-Manning and I. Witten: *On-line and off-line heuristics for inferring hierarchies of repetitions in sequences.* Proc. IEEE, 88(11) Nov 2000, pp. 1745–1755.
23. S. C. Sahinalp and U. Vishkin: *Efficient approximate and dynamic matching of patterns using a labeling paradigm*, in FOCS, 1996.
24. K.-B. Schürmann and J. Stoye: *An incomplex algorithm for fast suffix array construction.* Software - Pract. and Exp., 37(3) 2007, pp. 309–329.
25. J. S. Sim: *Time and space efficient search for small alphabets with suffix arrays*, in Proc. FSKD, 2005.
26. E. Ukkonen: *On-line construction of suffix trees.* Algorithmica, 14 1995, pp. 249–260.

sequence	length (chars.)	random					maximal length					maximal compression				
		update	K & S	L & S	ratio K & S	ratio L & S	update	K & S	L & S	ratio K & S	ratio L & S	update	K & S	L & S	ratio K & S	ratio L & S
CANTERBURY CORPUS																
alice29.txt	152089	163	2812	1497	17.25	9.18	192	2357	1371	12.28	7.14	269	1091	510	4.06	1.90
asyoulik.txt	125179	131	2111	1109	16.11	8.47	127	1727	1059	13.60	8.34	182	866	405	4.76	2.23
cp.html	24603	15	132	95	8.80	6.33	15	96	64	6.40	4.27	18	55	40	3.06	2.22
fields.c	11150	6	38	31	6.33	5.17	8	19	21	2.38	2.62	3	18	6	6.00	2.00
grammar.lsp	3721	3	5	5	1.67	1.67	0	2	3	div 0	div 0	0	1	0	div 0	div 0
kennedy.xls	1029744	1323	34905	12829	26.38	9.70	1230	35962	13796	29.24	11.22	1541	4871	1671	3.16	1.08
lcet10.txt	426754	516	17151	6871	33.24	13.32	522	16447	6449	31.51	12.35	749	5815	2259	7.76	3.02
plrabn12.txt	481861	588	22657	8853	38.53	15.06	606	19295	9304	31.84	15.35	887	7841	2911	8.84	3.28
ptt5	513216	1248	7389	4617	5.92	3.70	696	5323	3705	7.65	5.32	1900	842	369	0.44	0.19
sum	38240	42	234	151	5.57	3.60	34	187	99	5.50	2.91	28	82	48	2.93	1.71
xargs.1	4227	6	25	9	4.17	1.50	2	6	2	3.00	1.00	2	4	2	2.00	1.00
LARGE CORPUS																
bible.txt	4047392	5055	337725	115481	66.81	22.84	5168	332777	116260	64.39	22.50	10285	158048	38038	15.37	3.70
E.coli	4638690	5534	382636	151405	69.14	27.36	6307	337196	151540	53.46	24.03	14808	140788	31189	9.51	2.11
world192.txt	2473400	3084	200643	67079	65.06	21.75	3089	187505	65213	60.70	21.11	5573	90738	25276	16.28	4.54
PURDUE CORPUS																
All_Up_1M.fasta	1001002	1238	61657	24597	49.80	19.87	1200	55389	23982	46.16	19.98	2350	14109	4167	6.00	1.77
All_Up_400k.fasta	399615	501	13959	6777	27.86	13.53	481	13294	6698	27.64	13.93	884	2758	1129	3.12	1.28
Helden_All.fasta	112507	119	1511	963	12.70	8.09	122	1363	933	11.17	7.65	165	382	191	2.32	1.16
Helden_CGN.fasta	32871	31	244	172	7.87	5.55	34	232	178	6.82	5.24	19	55	50	2.89	2.63
Spor_All_2x.fasta	444906	112	82	94	0.73	0.84	57	34	44	0.60	0.77	61	35	71	0.57	1.16
Spor_All.fasta	222453	246	3658	2107	14.87	8.57	250	3314	2140	13.26	8.56	413	775	401	1.88	0.97
Spor_EarlyI.fasta	31039	34	187	152	5.50	4.47	26	220	190	8.46	7.31	25	56	47	2.24	1.88
Spor_EarlyII.fasta	25008	20	145	151	7.25	7.55	15	166	121	11.07	8.07	33	60	39	1.82	1.18
Spor_Middle.fasta	54325	S 51	526	351	10.31	6.88	62	506	396	8.16	6.39	73	117	66	1.60	0.90

Figure 4. Comparison between update and reconstruction from scratch of the suffix array. Times are given in hundredth of seconds

The Virtual Suffix Tree: An Efficient Data Structure for Suffix Trees and Suffix Arrays*

Jie Lin, Yue Jiang, and Don Adjeroh

Lane Department of Computer Science and Electrical Engineering
West Virginia University, Morgantown, WV 26506
jlin@mix.wvu.edu, yue@csee.wvu.edu, don@csee.wvu.edu

Abstract. We introduce the VST (virtual suffix tree), an efficient data structure for suffix trees and suffix arrays. Starting from the suffix array, we construct the suffix tree, from which we derive the virtual suffix tree. The VST provides the same functionality as the suffix tree, including suffix links, but at a much smaller space requirement. It has the same linear time construction even for large alphabets, Σ, requires $O(n)$ space to store (n is the string length), and allows searching for a pattern of length m to be performed in $O(m \log |\Sigma|)$ time, the same time needed for a suffix tree. Given the VST, we show an algorithm that computes all the suffix links in linear time, independent of Σ. The VST requires less space than other recently proposed data structures for suffix trees and suffix arrays, such as the enhanced suffix array [1], and the linearized suffix tree [16]. On average, the space requirement (including that for suffix arrays and suffix links) is $13.8n$ bytes for the regular VST, and $12.05n$ bytes in its compact form.

1 Introduction

The suffix tree is an important data structure used to represent the set of all suffixes of a string. The suffix tree is efficient in both time and space, and has been used in a variety of applications, such as pattern matching, sequence alignment, the identification of repetitions in genome-scale biological sequences, and in data compression. Various algorithms have been developed for efficient construction of suffix trees [28,22,27,8]. However, one major problem with the suffix tree is its practical space requirement. The suffix array is a related data structure, which was originally introduced in [21] as a space-efficient alternative to the suffix tree. The suffix array simply provides a listing of all the suffixes of a given string in lexicographic order. The suffix array can be used in most (though, not all) situations where a suffix tree can be used.

Although the theoretical space complexity is linear for both data structures, typically, for a given string T of length n, the suffix array requires about three to five times less space than the suffix tree. The construction time for both algorithms is also $O(n)$ on average. For suffix arrays, construction algorithms that run in $O(n \log n)$ worst case[1] are relatively easy to develop, but $O(n)$ worst case algorithms are much harder to come by. Recent suffix sorting algorithms with worst-case linear time have been reported in [13,18,17,3]. Gusfield [11] provides a comprehensive treatment of suffix trees and its applications. Puglisi et al [26] provide a recent survey on suffix arrays. Adjeroh et al (2008) provide an extensive discussion on the connection between the Burrows-Wheeler transform [6] and suffix trees and suffix arrays.

For small alphabet sizes, the suffix tree and the suffix array have about the same complexity in pattern matching. For pattern matching, the suffix array requires time

* Partly supported by a DOE CAREER award.

[1] All logarithms are to base 2, unless otherwise stated.

Jie Lin, Yue Jiang, Don Adjeroh: The Virtual Suffix Tree: An Efficient Data Structure for Suffix Trees and Suffix Arrays, pp. 68–83.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

in $O(m \log n)$ to locate one occurrence of a pattern of length m in T. However, with additional data structures, such as the `lcp` array, this time can be reduced to $O(m + \log n)$. With the suffix tree, the same search can be performed in $O(m)$ time. The problem, however, is for sequences with large alphabets. Here, $|\Sigma|$, the alphabet size is no longer negligible. Using the array representation of nodes in the suffix tree will require $O(n|\Sigma|)$ space for the suffix tree, and $O(m)$ time for pattern matching. For linear space, the linked list or binary search tree can be used, but the search time becomes $O(m|\Sigma|)$ or $O(m \log |\Sigma|)$ respectively.

The challenge therefore is to develop space-efficient data structures that can support pattern matching using the same time complexity as suffix trees, but at a practical space requirement that approaches that of the suffix array. Such a data structure should also support the complete functionality of the suffix tree, such as support for suffix links, as may be required in certain applications. Two recent data structures that have tried to address this problem are the ESA – *enhanced suffix array* [1], and the LST – *linearized suffix tree* [15,16]. Both methods are based on the notion of `lcp`-intervals [14], constructed using the suffix array and the `lcp` array. Other related data structures that have been proposed include the suffix cactus [12], suffix vectors [23,25], compact suffix trees [20], the lazy suffix trees [9], level-compressed suffix trees [4], compressed suffix trees [24], and compressed suffix arrays [10]. See also [2].

1.1 Main results

We introduce another data structure, *the virtual suffix tree* (VST), an efficient data structure for suffix trees and suffix arrays. The VST does not use the `lcp`-intervals, but rather exploits the inherent nature of the suffix tree topology. We state our main results in the form of two theorems about the VST.

Theorem 1. *Given a string $T = T[1..n]$, with symbols from an alphabet Σ, and the virtual suffix tree for T, we can count the number of occurrences of a pattern $P = P[1..m]$ in T in $O(m \log |\Sigma|)$ time, and locate all the η_{occ} occurrences of P in T in $O(m \log |\Sigma| + \eta_{occ})$ time.*

Theorem 2. *Given a string $T = T[1..n]$, with symbols from an alphabet Σ, the virtual suffix tree, including the suffix link, can be constructed in $O(n)$ time, and $O(n)$ space, independent of Σ.*

Essentially, the VST provides the same functionality as the suffix tree, but at a much smaller space requirement. It has the same linear time construction for large $|\Sigma|$, requires $O(n)$ space to store, and allows searching for a pattern of length m to be performed in $O(m \log |\Sigma|)$ time, the same time needed for a suffix tree. To provide the complete functionality of the suffix tree, we describe a simple linear time algorithm that computes the suffix links based on the VST. Although the space needed for the VST is linear (as in suffix tree implementations using linked lists or binary trees), the practical space requirement is much smaller than that of a suffix tree. The VST requires less space than other recently proposed data structures for suffix trees and suffix arrays, such as the ESA [1], and the LST [16]. On average, the space requirement (including that for suffix arrays and suffix links) is $13.8n$ bytes for the regular VST, and $12.05n$ bytes in its compact form. This can be compared with the $20n$ bytes needed by the LST or the ESA.

1.2 Organization

The next section introduces the key notations and definitions used. In Section 3, we introduce the basic data structure and discuss the properties of the VST. Section 4 presents an improved data structure, along with algorithms for its construction. A complexity analysis on the construction and use of the VST is also presented in this section. Section 5 shows how the suffix link can be constructed on the VST. The paper is concluded in Section 6.

2 Basic notations and definitions

Let $T = T[1..n]$ be the input string of length n, over an alphabet Σ. Let $T = \alpha\beta\gamma$, for some strings α, β, and γ (α and γ could be empty). The string β is called a *substring* of T, α is called a *prefix* of T, while γ is called a *suffix* of T. The prefix α is called a proper prefix of T if $\alpha \neq T$. Similarly, the suffix γ is called a proper suffix of T if $\gamma \neq T$. We will also use $t_i = T[i]$ to denote the i-th symbol in T — both notations are used interchangeably. We use $T_i = T[i..n] = t_i t_{i+1} \cdots t_n$ to denote the i-th suffix of T. For simplicity in constructing suffix trees, we usually ensure that no suffix of the string is a proper prefix of another suffix by appending a special symbol, \$ to T, such that $\$ \notin \Sigma$, and $\$ < \sigma$, $\forall \sigma \in \Sigma$.

Given a string T, its suffix tree (ST) is a rooted tree with n leaves, where the i-th leaf node corresponds to the i-th suffix T_i of T. Except for the root node and the leaf nodes, every node must have at least two descendant child nodes. Each edge in the suffix tree represents a substring of T, and no two edges out of a node start with the same character. For a given edge, the *edge label* is simply the substring in T corresponding to the edge. We use l_i to denote the i-th leaf node. Then, l_i corresponds to T_i, the i-th suffix of T. When the edges from each node are sorted alphabetically, then l_i will correspond to $T_{SA[i]}$, the i-th suffix of T in lexicographic order.

For edge (u, v) between nodes u and v in ST, the edge label (denoted $label(u, v)$) is a non-empty substring of T. The edge length is simply the length of the edge label. The edge label is usually represented compactly using two pointers to the beginning and end of its corresponding substring in T. For a given node u in the suffix tree, its *path label*, $L(u)$ is defined as the label of the path from the root node to u. Since each edge represents a substring in T, $L(u)$ is essentially the string formed by the concatenation of the labels of the edges traversed in going from the root node to the given node, u. The *string depth* of node u, (also called its length) is simply $|L(u)|$, the number of characters in $L(u)$. The *node depth* (also called node level) of node u is the number of nodes encountered in following the path from the root to u. The root is assumed to be at node depth 0.

Certain suffix tree construction algorithms make use of *suffix links*. The notion of suffix links is based on a well-known fact about suffix trees [28,20], namely, if there is an internal node u in ST such that its path label $L(u) = a\alpha$ for some single character $a \in \Sigma$, and a (possibly empty) string $\alpha \in \Sigma^*$, then there is a node v in ST such that $L(v) = \alpha$. A pointer from node u to node v is called a *suffix link*. If α is an empty string, then the pointer goes from u to the root node. Suffix links are important in certain applications, such as in computing matching statistics needed in approximate pattern matching, regular expression matching, or in certain types of traversal of the suffix tree.

A predominant factor in the space cost for suffix trees is the number of interior nodes in the tree, which depends on the tree topology. Thus, a major consideration is how the outgoing edges from a node in the suffix tree are represented. The three major representations used for outgoing edges are arrays, linked lists, and binary search trees. While the array is simple to implement, it could require a large memory for large alphabets. However, independent of the specific method adopted, a simple implementation of the suffix tree can require as large as $33n$ bytes of storage with suffix links, or $25n$ bytes without suffix links [2].

The suffix array (SA) is another data structure, closely related to the suffix tree. The suffix array simply provides a lexicographically ordered list of all the suffixes of a string. If $SA[i] = j$, it means that the i-th smallest suffix of T is T_j, the suffix starting at position j in T. A related structure, the `lcp` array contains the length of the longest common prefixes between adjacent positions in the suffix array. Combining the suffix array with the `lcp` information provides a powerful data structure for pattern matching. With this combination, decisions on the occurrence (or otherwise) of a pattern P of length m in the string T of length n can be made in $O(m + \log n)$ time. Given the new worst-case linear-time direct SA construction algorithms, and the small memory footprint of suffix arrays, it is becoming more attractive to construct the suffix tree from the suffix array. A linear-time algorithm for constructing ST from SA is presented in [2].

3 Basic Data Structure

Starting from the suffix array, we construct an efficient data structure to simulate the suffix tree (ST). We call this structure a Virtual Suffix Tree (VST). The VST stores information about the basic topology of the suffix tree, the suffix array, and the suffix links. Thus, the VST is represented as a set of arrays that maintains information on the internal nodes of the suffix tree. The leaf nodes are not stored directly. However, whenever needed, information about any leaf node can be obtained via the suffix array. Unlike the ESA and LST, the VST neither uses the `lcp`-interval tree nor stores the `lcp` array. We call the data structure a virtual suffix tree in the sense that it provides all the functionalities of the suffix tree using the same space and time complexity as a suffix tree, but without storing the actual suffix tree. Later, we show that the VST leads to a more compact representation of suffix trees and suffix arrays. (We mention that [14] also used the term "virtual suffix tree", but for a limited form of the enhanced suffix array).

Below, we present the basic VST. This structure will require 14 bytes for each node in the VST and supports pattern matching in $O(m \log |\Sigma|)$ time, for an m-length pattern. In the next section, we present an improved data structure that reduces the space cost by eliminating the need to store edge lengths, while still maintaining $O(m \log |\Sigma|)$ time for pattern matching. We also describe a more compact structure for the VST that uses only 10 bytes for each internal node of the VST, and 5 bytes for each leaf node. Pattern matching on this compact representation will, however, be in $O(m|\Sigma|)$ time.

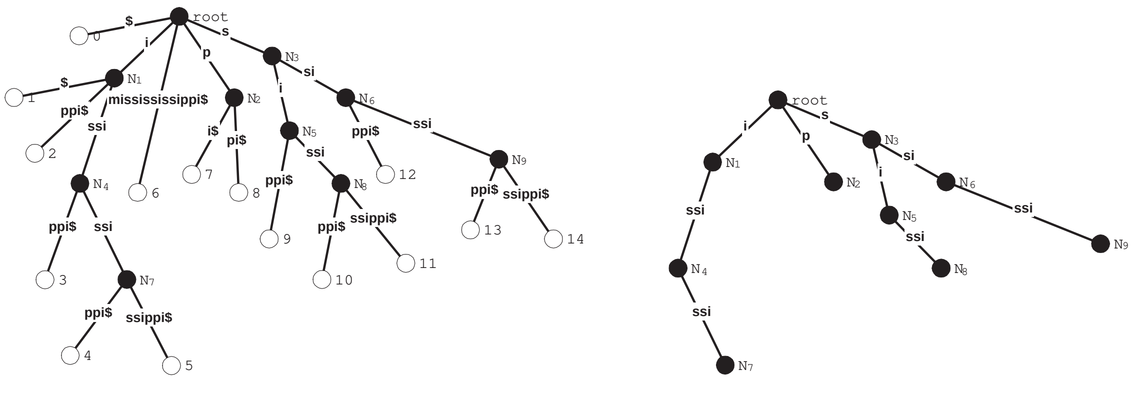
Each node in the VST corresponds to a distinct internal node in the suffix tree. In its basic form, each node in the VST is characterized by five attributes. For a given node in the VST (say node u), with a corresponding internal node in ST (say node u_{ST}), the five attributes are defined as follows.

- `sa_index`: index in the suffix array (SA index) of the leftmost leaf node under the internal node u_{ST} of the suffix tree.
- `fchild`: the node ID of the first child node of u_{ST} that is also an internal node. (Scanning is done left to right; edges at a node are also sorted left to right in ascending lexicographic order). If node u is a leaf node in the VST, the value will be negative. The absolute value will point to the first child node of the next internal node in the VST.
- `elength`: The edge length of the edge (v, u) in the VST, or equivalently (v_{ST}, u_{ST}) in the suffix tree, where v is the parent node of u and v_{ST} is the parent node of u_{ST}.
- `nfleaf`: the number of child leaf nodes *before* the first child of u_{ST} that is also an internal node.
- `nnleaf`: the number of sibling leaf nodes *after* u_{ST}, the current internal node of the suffix tree, but before the next sibling internal node.

In terms of storage, the `sa_index`, `fchild` and `elength` each requires one integer (4 bytes), while `nfleaf` and `nnleaf` each requires one byte of storage (assuming $|\Sigma| \leq 256$).

3.1 Example VST

We use an example sequence to explain the above definitions. The suffix tree and VST for the string `mississsissippi$` are shown in Figure 1. Note that the string `mississsissippi$` is made intentionally different from `mississippi$`, to capture some of the cases involved in a VST. Only the internal nodes (dark nodes) are explicitly stored in the VST. The leaf nodes (empty circles) are not stored. The order of storage is based on the node-depths, from top to bottom. Table 1 shows the corresponding values of the VST node attributes for each VST node in the example.



(a) (b)

Figure 1. Suffix tree and virtual suffix tree for the string $T =$ `mississsissippi$`. (a) suffix tree; (b) virtual suffix tree. The number at each leaf node indicates the position in SA. The number at each internal node indicates the node ID in the VST.

3.2 Properties of the Virtual Suffix Tree

We can trace the properties of the VST based on the standard properties of a suffix tree.

node	root	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9
`sa_index`	0	1	7	9	3	9	12	4	10	13
`fchild`	N_1	N_4	$-N_5$	N_5	N_7	N_8	N_9			
`elength`	0	1	1	1	3	1	2	3	3	3
`nfleaf`	1	2	2	0	1	1	1	2	2	2
`nnleaf`	0	1	0	0	0	0	0	0	0	0

Table 1. VST node attributes for the example sequence $T =$ `mississippi$` used in Figure 1.

1. The VST only stores the internal nodes of the suffix tree. No leaf nodes in the ST are represented in the VST. Information about the leaf nodes can be obtained from the SA when needed. Then the space requirement of the VST depends on the topology of the the suffix tree, or more specifically, on the number of internal nodes.
2. The number of leaf nodes in a suffix tree is n. The number of internal nodes in the suffix tree (and hence number of nodes in the VST) is at most n.
3. The VST stores only the SA index of the leftmost leaf nodes and information about the child nodes.
4. For a given node in the VST, the number of child nodes will be no larger than $|\Sigma|$. Thus, the time needed to match a symbol is at most $O(\log |\Sigma|)$.
5. The nodes in the VST are ordered based on the internal nodes of the suffix tree using the HSAM (hierarchy sequential access method). The child nodes from any given node will be stored sequentially. The child nodes of two nearby nodes will therefore be stored in nearby locations. This is an important property for addressing problems involving locality of reference.

We introduce further definitions needed in the description below. For a given node u in the VST, we use the term *prior* node to denote the node that appears before the current node u in the HSAM ordering. Similarly, *next* node denotes the node that appears after the current node u in this ordering. We use `lsa_index` (left `sa_index`) to denote the SA index of the leftmost leaf node that is a descendant of u. Similarly, `rsa_index` (right `sa_index`) denotes the rightmost leaf node that has u as its ancestor. Figure 2 shows an example.

It is simple to determine the `lsa_index` and the leftmost child node of any given node. The properties of the VST and the organization of the VST lead to the following lemma about the VST (we omit the proof for brevity):

Lemma 3. *For a given node in the VST, its rightmost child node, and the right* `sa_index` *can each be determined in constant time.*

3.3 Pattern matching on VST

Lemma 3 provides an indication of how pattern matching can be performed on the VST. For pattern matching using the suffix tree, an important issue is how to quickly locate all the child nodes for a given internal node. In the VST, each node points to its leftmost leaf node using the `sa_index`. During pattern matching, at any given node in the VST, we will need to determine four parameters, namely the leftmost child node (`lchild`), the rightmost child node (`rchild`), the left `sa_index` (`lsa_index`) and the right `sa_index` (`rsa_index`). These parameters define the boundaries of the search at the given node. To search in a leaf node of the VST, we will need only the left

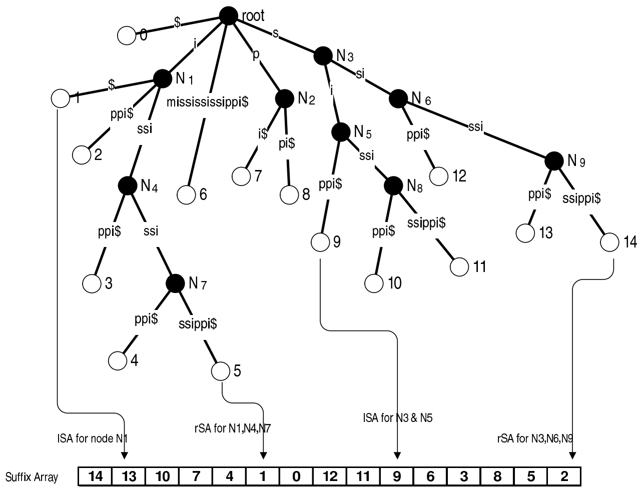


Figure 2. Example VST (solid nodes) showing left SA index (lSA) and right SA index (rSA) for sample nodes.

sa_index and right sa_index of the node. When we search in an internal node, we will need all the four parameters to match a pattern. Lemma 3 shows that for any given node, we can determine each of these parameters in constant time. The following two examples further illustrate the two cases involved in computing the rsa_index, and how pattern matching can be performed on the VST.

Example 4. Determining the right boundary from a next sibling node. Consider node N_5 in Figure 2. The left sa_index of N_5 is 9 and the right sa_index is 11, since N_5.sa_index=9 and N_{5+1}.sa_index=12, and hence the right sa_index of N_5=12-1=11. The leftmost child node is the fchild of the current node, thus the leftmost child of N_5 is N_8. The next node of the rightmost child node is N_{5+1}.fchild=N_9. Then the rightmost child node is N_{9-1}=N$_8$, since the child node will be stored side by side between sibling nodes.

Example 5. Determining the right boundary from the right boundary of the parent node. Consider node N_1 in Figure 2. The left sa_index of N_1 is N_1.sa_index=1. The right sa_index of N_1 is N_2.sa_index - (N_1.nnleaf -1)=7-1-1=5. The leftmost child node of N_1 is N_1.fchild=N_4. The next node of N_1 is N_2. Since N_2.fchild=-N_5 is negative, N_2 must be a leaf node in the VST. The right node of N_1 will thus point to N_5. We therefore know that the next node of the rightmost child node of N_1 will be N_5. Finally, the rightmost child node of N_1 can be determined as $N_{5-N_1.\texttt{nnleaf}} = N_{5-1} = N_4$.

We summarize the foregoing discussion as the first main result of this paper:

Theorem 6. *Given a string $T = T[1..n]$ of length n, with symbols from an alphabet Σ, and the virtual suffix tree for T, we can count the number of occurrences of a pattern $P = P[1..m]$ in T in $O(m \log |\Sigma|)$ time, and locate all the η_{occ} occurrences of P in T in $O(m \log |\Sigma| + \eta_{occ})$ time.*

Proof. The theorem is a consequence of Lemma 3. First consider the cost of one single symbol-by-symbol comparison at a node in the VST. The number of child nodes at any internal node can be no larger than $|\Sigma|$, and we can find the boundaries of the search in constant time. Since the edges are ordered lexically at each internal node, and given the HSAM ordering, matching a single symbol can be done in $O(\log|\Sigma|)$ time steps using binary search. To find the first match, we need to consider the m symbols in the pattern. We perform the above symbol-by-symbol comparisons at most m times to decide whether there is a match or not. After a match is found, we can again use binary search (using `lsa_index` and `rsa_index` as bounds) to determine all the η_{occ} occurrences of the pattern. Reporting each occurrence can be done in constant time, or an additional η_{occ} time for all the occurrences. □

4 Improved Virtual Suffix Tree

The basic data structure introduced above stores the length of each edge in the VST. We can improve the structure to reduce the space requirement by avoiding the need to store information about the edge lengths directly. The improved data structure has only four attributes rather than five. The attributes `sa_index` and `elength` in the basic structure are now combined into one attribute called the adjusted SA index (`asa_index`). This requires a key modification to the suffix tree, leading to an important distinction between the suffix tree and the virtual suffix tree.

4.1 Adjusting edge lengths

A well-known property of the suffix tree is that no two edges out of a node in the tree can start with the same symbol. For efficient representation of the VST, this characteristic of the ST is modified such that, for a given node, every edge that leads to an internal node in the VST has an equal length. This modification is done as follows: Start from the root node and progress towards the leaf nodes in the VST. For a given internal node, say u, adjust the edge label from u to each of its children such that all edges that lead to an internal node will have the same edge length. The major criteria is that, for two sibling internal nodes, their edge labels differ only in the last symbol. If for some edge, say (u, w), the original edge length (or edge label) is longer than the new length, prepend the extraneous part of old $label(u, w)$ to each outgoing edge from w. The edge length for edges that lead to leaf nodes are left unchanged. Then repeat the adjustment at each child node of u. Figure 3 shows an example of this procedure. We can observe that this adjustment only affects the edge lengths, and does not change the general topology of the suffix tree.

The above adjustment procedure leads to an important property of the VST:

Property: *In the improved VST, all internal sibling nodes occur at the same node-depth, and same string-depth, and the edge labels for the edges from the parent to each sibling differ only in the last symbol. This means that, in the VST, two branches from the same node can start with the same symbol, but their edge labels will differ.*

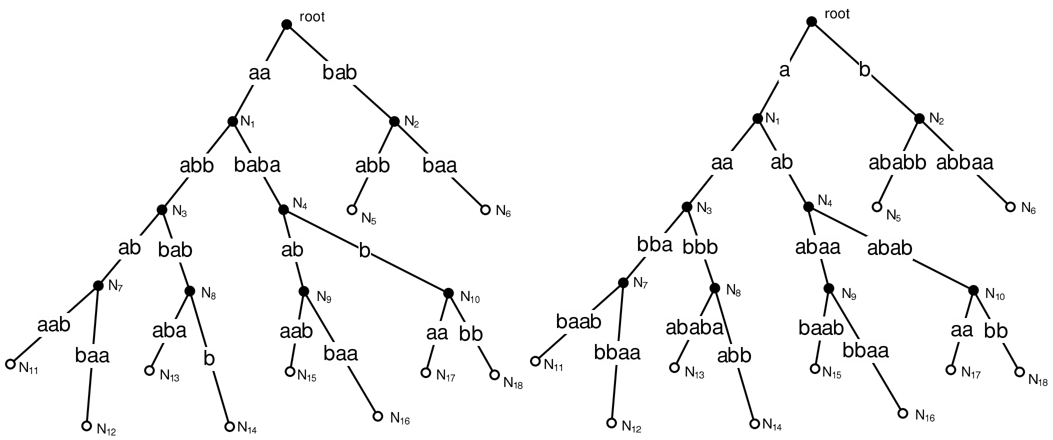
This property provides an important difference between the suffix tree and the VST. The suffix tree mandates that no two edges from the same node have the same starting symbol. Further, the suffix tree only guarantees that the node-depth of two sibling nodes are the same, but not their string depth. This property of equal-length sibling edge labels is the key to more efficient representation of the VST, without explicit edge labels. Figure 4 shows an example of the modified suffix tree with equal-length edges for sibling nodes that are also internal nodes, and the corresponding improved virtual suffix tree. Table 2 shows the corresponding values of the attributes for each node in the improved VST. What remains is how we compute `asa_index`, the adjusted SA index. This is done by combining the original `sa_index` with `elength`. We state the following lemma without proof:

Lemma 7. *Given a node in the VST say u, and its parent node (say v), we can compute the adjusted SA index in constant time. Further, when required, the edge length can be determined in constant time.*

While we store only the `asa_index`, our calculations will still use the original `sa_index`. However, this can be derived from `asa_index` in constant time. In fact, we can observe that in practice, we need to compute the `asa_index` for only the leftmost child node at each node-level, while keeping the original `sa_index` for all other nodes. To determine the `new_elength` for these other nodes, we simply make a constant time access to their leftmost (sibling) node (at the same node-level), and then use this to compute the length. For searching with the VST, we will calculate the length of the common string at each level. If the length is greater than 0, then we know there is a common string in the child nodes and only the last character is different. Thus, we do not need to store the edge lengths explicitly, leading to a reduction of one integer per node over the basic VST.

NodeName	root	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9
`sa_index`	0	1	7	9	3	9	12	4	10	13
`fchild`	N_1	N_4	-N_5	N_5	N_7	N_8	N_9			
`new_elength`	0	1	1	1	1	1	1	3	1	2
`nfleaf`	1	2	2	0	1	1	1	2	2	2
`nnleaf`	0	1	0	0	0	0	0	0	0	0
`asa_index`	0	1	7	9	3	9	12	4+3=7	10	13+2=15

Table 2. Node attributes in the improved VST for the example sequence, T = `mississississippi$`. We have included `new_elength`, so one can compare with `elength` in Table 1. However, in practice this will not be stored in the VST.



(a) (b)

Figure 3. VST edge-length adjustment procedure. (a) original tree; (b) improved tree after adjusting the edge lengths.

4.2 Construction algorithm

Construction of the VST makes use of an array Q which records the internal nodes of the suffix tree. This array maps the internal nodes of the suffix tree to nodes in the VST. Thus, elements in the array are in the same ordering as the corresponding nodes in the VST.

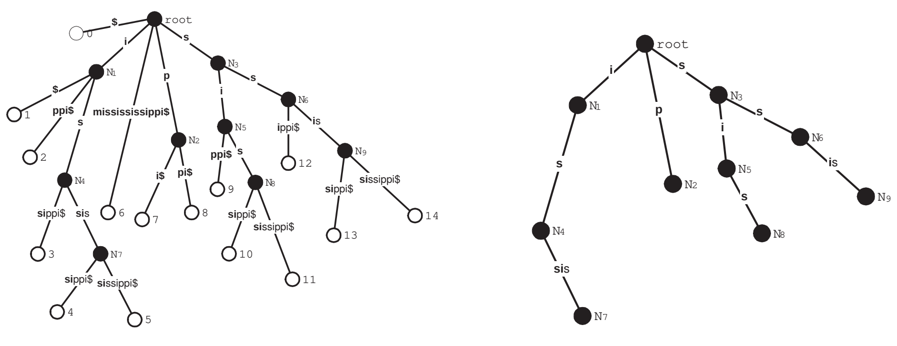
Given an input string T, the first step is to construct the suffix array for T. This can be done in worst case linear time and linear space using any of the existing algorithms [13,18,17,3]. Using the SA, we construct the suffix tree as described in [2]. While the suffix tree can be constructed directly in linear time, working from the SA to the ST will require less space for the construction. The suffix tree is then preprocessed in linear time to adjust the edges from a given parent node that lead to internal child nodes to equal-length edges. Using the adjusted suffix tree, the algorithm will process the internal nodes in the suffix tree in a top-down manner to determine the attributes (`fchild`, `nfleaf` and `nnleaf`) for the corresponding nodes in the VST. Next, we process the VST from the VST leaf nodes to the root, using the Q array to update the `asa_index` at each node. The adjusted `asa_index` field includes information on the `sa_index` and edge length.

The steps for constructing the VST for a given input string are summarized in Algorithm 1.

4.3 Further space reduction

We can further reduce the space needed by the VST, at the cost of an increased time for pattern matching. In the pattern matching phase, if the algorithm is to compare symbols one-by-one, rather than using binary search on the branches from a given node in the VST, we will only need to compute the `lsa_index` and `rsa_index` of the node.

Consider an arbitrary node (say node u) in the VST. The number of children from u or the number of u's leaf nodes cannot be larger than $|\Sigma|$. Thus, the `sa_index` of any child node of u will lie between node u's `lsa_index` and `rsa_index`. Then comparing one symbol from the pattern against the first symbol on each edge from u to its children will require at most $O(|\Sigma|)$ time steps. The left child node and the right child node will not need to be used again. Thus, the attributes `fchild` and



(a) (b)

Figure 4. Improved VST for the string $T =$ `mississippi$`: (a) modified suffix tree; (b) improved virtual suffix tree

`nfleaf` in the leaf nodes of the VST are no longer required. We make the `asa_index` to be negative for the leaf nodes. Thus, during pattern matching, this serves as a flag for the VST leaf nodes. This compact structure will reduce the space requirement at each leaf node of the VST by 5 bytes. Pattern matching time, however, will increase to $O(|\Sigma|)$ for each symbol in P, or $O(m|\Sigma|)$ overall.

4.4 Complexity Analysis

Time and space complexity The time cost for lines 1-3 in the construction algorithm (Algorithm 1) is $O(n)+O(n)+O(n)=O(n)$. Lines 5-17 in the algorithm perform a one time traversal of the nodes in the suffix tree. The respective values of $pTop$ and $pBottom$ range from 1 to $2n$. Thus the cost for the traversals is $O(n)$. Lines 18-27 in the algorithm run at most $pBottom$ times. The time for lines 18-27 in the algorithm is thus $O(n)$, since each iteration of the loop requires constant time. Therefore, for the regular VST, the overall construction time is $O(n)$. The time for pattern matching is in $O(m \log |\Sigma|)$. For the compact structure, the construction time is the same as the regular structure, but the VST is no longer stored linearly. Here we use an array to store the relation between the Q array and the compact VST. The searching time is now $O(m|\Sigma|)$.

The space requirement clearly depends on the number of nodes in the VST, which is at most n for a sequence of length n. Each node requires a fixed amount of memory to store, leading to an $O(n)$ space requirement.

Number of nodes and practical space requirement The actual space needed for the VST depends on the topology of the suffix tree. This topology can be captured by the number of internal nodes in the suffix tree, or alternatively, by the quantity R_{IL}, the ratio between the number of internal nodes and the number of leaf nodes. We call R_{IL} the *density* or *branching factor* for the suffix tree. We conducted an experiment to evaluate the effect of this branching factor on the storage requirement of the VST. The suffix tree was constructed and the branching factors computed for a set of files taken from [26]. For each file, we used the first 2^{24} symbols as the text, and computed the branching factor. Table 3 shows the results. The maximum ratio of 0.76 was observed for the file `Jdk13c`. On average, however, the maximum ratio was around 0.63. The worst case occurs for a sequence with $|\Sigma| = 1$, (that is, $T = a^n$), leading to a branching factor of 1. The table shows that, for a given sequence, the branching factor depends on a complex relationship between n, $|\Sigma|$, and the mean LCP.

The space requirement for the VST, for both the compact and regular structures depends directly on the branching factor. The last two columns in Table 3 show the maximum space requirement for each file.

The foregoing discussion leads to the following lemma on the construction of the VST:

Lemma 8. *Given a string $T = T[1..n]$, with symbols from an alphabet Σ, the virtual suffix tree (without the suffix link) can be constructed in $O(n)$ time, and $O(n)$ space, independent of Σ.*

File	$\|\Sigma\|$	Max Ratio	Compact	Regular	Description
Bible	63	0.61	8.60n	10.13n	King James bible
Chr22	5	0.73	9.50n	11.33n	Human chromosome 22
E.coli	4	0.65	8.89n	10.52n	*Escherichia coli* genome
Etext	146	0.54	8.02n	9.36n	Texts from Gutenberg project
Howto	197	0.55	8.13n	9.51n	Linux Howto files
Jdk13c	113	0.76	9.69n	11.59n	JDK 1.3 documentation
Rctail	93	0.66	8.95n	10.60n	Reuters news in XML format
Rfc	120	0.64	8.77n	10.36n	Concatenated IETF RFC files
Sprot	94	0.61	8.54n	10.05n	
World	94	0.54	8.06n	9.41n	CIA world fact book
Average		0.63	8.71n	10.29n	

Table 3. Branching factor and maximum space requirement for various sample files.

5 Computing Suffix Links

Constructing the suffix tree from the suffix array as described in [2] does not include the suffix link. There are also a number of other suffix tree construction algorithms that build the suffix tree without the suffix link. See Farach et al [8], and Cole and Hariharan [7]. The suffix link, however, is a significant component of the suffix tree, and is important in certain applications, such as approximate pattern matching using matching statistics, and other forms of traversal on the suffix tree. Thus, a data structure to support the complete functionality of the suffix tree requires an inclusion of the suffix link. Recent efficient data structures for suffix trees have thus provided mechanisms for constructing the suffix link. The ESA [1] provided suffix links using complicated RMQ preprocessing [5]. The LST [16] also supported suffix links using the `lcp`-interval tree and intervals defined on the inverse suffix array. A recent work by Maaβ [19] focused exclusively on suffix link construction from suffix arrays, or from suffix trees that do not have such links.

The virtual suffix tree provides a natural mechanism for constructing suffix links. The key idea is that suffix links in the VST can be computed bottom-up, from the nodes with the highest node-depth (leaf nodes) in the VST to those with the least (the root). This is based on the following two observations about suffix links.

1. Consider a leaf node u_{ST} in the suffix tree corresponding to suffix T_i in the original sequence. The suffix link from u_{ST} will point to the leaf node corresponding to the suffix T_{i+1} (that is, the suffix that starts at the next position in the sequence).
2. The suffix link from a node u in the VST will point to some node w with a smaller string-depth in the VST, such that $|L(u)| = |L(w)| + 1$ (or equivalently $|L(u_{ST})| = |L(w_{ST})| + 1$).

The following lemma establishes how we can build suffix links on the VST.

Lemma 9. *Given the VST for a string $T = T[1..n]$ of length n, the suffix links can be constructed in $O(n)$ time using additional $O(n)$ space.*

Proof. Let u and w be two arbitrary nodes in the VST. Let v be the parent node of u. Let u.`slink` be the node to which the suffix link from node u points to. We consider two cases:

Case A: *u is a leaf node in the VST.* Then, using the above observations, the suffix link from node u will point to node w in the VST (that is, u.`slink` $= w$) such that

$SA[w.\texttt{sa\_index}] = SA[u.\texttt{sa\_index}] + 1$. Clearly, $|L(w)| = |L(u)| - 1$, where $L(x)$ is the path label of node x. Note that this path label is not explicitly stored in the VST, but for each node, the length can be computed in constant time. This computation can be performed in constant time by maintaining two arrays and observing that $n - |L(w)| = n - |L(u)| + 1$. One array is the inverse suffix array (ISA) for the given string, defined as follows: $ISA[i] = j$ `if` $SA[j] = i$, $(i, j = 1, 2, ..., n)$. The second is an array M that maps the SA values to the corresponding parent nodes in the VST, defined as follows: $M[i] = u$, if u_{ST} in ST is the parent node of the leaf node corresponding to the suffix $T_{SA[i]}$. Clearly, both arrays can be computed in linear time, and require linear space.

Case B: *u is not a leaf node in the VST.* This is a simpler case. When u is an internal node in the VST, the suffix link of u will point to some node w, such that w is an ancestor of node $u.\texttt{fchild.slink}$, such that $|label(u, u.\texttt{fchild})| = |label(w, u.\texttt{fchild.slink})|$. The $O(n)$ time result then follows by using the skip/count trick [11], by observing that a VST has at most n nodes, a node depth of at most n, and that each upward traversal on the suffix link decreases the node depth by at least 1. □

Algorithm 1: VST Construction Algorithm

```
Construct-VST(T, n)
SA ← Compute-SuffixArray(T, n)
ST ← SuffixTree-From-SuffixArray(SA)
ST ← Adjust-EdgeLengths(ST)
Initialize VST[],Q[], pTop=0, pBottom=0, curNode=root, Q[pTop]=root
while (pBottom >= pTop)
   for ( each childnode in curNode) do
      if (childnode is internal node in ST ) then
         pBottom ← pBottom + 1; Q[pBottom] ← childNode
        if childnode is first internal node then
           VST[pTop].fchild ← pBottom
         end if
      else
         Update VST[pTop].nfleaf and VST[pBottom].nnleaf
      end if
   end for
   pTop ← pTop + 1; curNode ← Q[pTop]
end while
for (pb ← pBottom down to 0) do
   if (VST[pb] is leaf node) then
      VST[pb].asa_index ← Q[pb].fchild
      else if (Q[pb].elength=1) then
         VST[pb].asa_index←VST[pb].fchild.asa_index
          + VST[pb].nfleaf - Q[pb].elength
      else
         VST[pb].asa_index←VST[pb].fchild.asa_index
          + VST[pb].nfleaf - Q[pb].elength+ Q[pb].elength
      end if
   end if
end for
```

Although the above description is from the viewpoint of a VST already constructed, the suffix links can be constructed as the VST is being constructed, by some modification of the VST construction algorithm. Algorithm 2 shows a modification of Algorithm 1 (the VST construction algorithm) to incorporate sections to compute the suffix link. The suffix link construction algorithm is based on the Q array used during the VST construction.

Algorithm 2: VST construction with suffix links

Initialize VST[],Q[],ISA[],M[], pTop←0, pBottom←0, curNode←root, Q[pTop]←root

⋮

for (pb ← pBottom **down to** 0) **do**
  if (VST[pb] is leaf node) **then**
    Update array M to map SA index and node VST [pb]

⋮

  end if
end for
for (pb ← pBottom **down to** 0) **do**
  if (VST[pb] is leaf node) **then**
    VST[pb].slink ← M[ISA[VST[pb].`sa_index`+1]]
  else
    Find ancestor w of VST[pb].`fchild.slink` s.t.
    $|label(w,$ VST[pb].`fchild.slink`$)|=|label($VST[pb], VST[pb].`fchild`$)|$
    Set VST[pb].`slink` ← w
  end if
end for

Figure 5 shows the result of the suffix link algorithm when applied to the VST of our example string $T =$ `mississississippi$`. Essentially, given the VST, the suffix link is constructed right to left, node-depth by node-depth, starting with the rightmost node at the deepest node-depth, and moving up the VST until we reach the root. Thus, the order of suffix link construction in the example will be $SL_1, SL_2, \ldots, SL_9$.

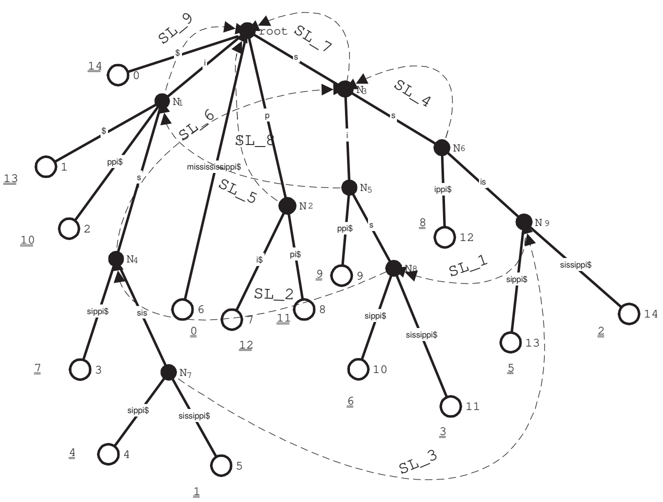


Figure 5. Suffix link on the VST for the sample string $T =$ `mississississippi$`.

Algorithm 2 shows that the additional work required to compute all the suffix links is linear in the length of the string. After construction, the suffix link on the

VST will require one additional integer per internal node in the VST. This can be compared with the 2 integers per node required to store the suffix link using the ESA, or LST. In a typical VST, where the maximum leaf node to internal node ratio is usually less than 0.7, the suffix link will require a maximum total extra space of $0.7n * 4 = 2.8n$ bytes. Table 4 shows the space required for the VST (including the suffix array and suffix links) for both the compact structure and the regular VST, at varying values of the branching factor.

Table 4. Storage requirement for the VST, including suffix links

	Ratio	Compact	Regular
Worst Case	1	15.50n	18.00n
Average Case	0.75	12.63n	14.50n
	0.7	12.05n	13.80n
	0.65	11.48n	13.10n
	0.6	10.90n	12.40n

We summarize the above discussion in the following theorem which captures the second main result of the paper:

Theorem 10. *Given a string $T = T[1..n]$, with symbols from an alphabet Σ, the virtual suffix tree, including the suffix link, can be constructed in $O(n)$ time and $O(n)$ space, independent of Σ.*

Proof. The theorem follows directly from Lemma 8 and Lemma 9. □

6 Conclusion

In this paper, we have presented the virtual suffix tree (VST), an efficient data structure for suffix trees and suffix arrays. The searching performance is the same as the suffix tree, that is, $O(m \log |\Sigma|)$ for a pattern of length m, with symbol alphabet Σ. We also showed how suffix links can be constructed on the VST in linear time, independent of the alphabet size. The VST does not store the edge lengths explicitly. This is achieved by modifying a key property of the suffix tree - the requirement that no two edges from a given node in the suffix tree can start with the same symbol. This key modification leads to a major distinction between the VST and the suffix tree, and results in extra space saving. However, whenever needed, the length for any arbitrary edge in the VST can be obtained in constant time using a simple computation. A further space reduction leads to a more compact representation of the VST, but at the expense of an increased search time, from $O(m \log |\Sigma|)$ to $O(m|\Sigma|)$.

The space requirement depends on the topology of the suffix tree, in particular on the branching factor. For the compact structure, the worst case space requirement (including the suffix array) is $11.5n$ bytes without suffix links, and $15.5n$ bytes with suffix links, where n is the length of the string. However, in practice, the branching factor is typically less than 0.7. For the compact structure, this gives less than $9.25n$ bytes on average without the suffix links, or $12.05n$ bytes with suffix links.

In this work, we have focused on efficient storage of the suffix tree and suffix array after they have been constructed. Thus, we constructed the VST from the suffix tree, which in turn was constructed from the suffix array. An interesting question is whether the virtual suffix tree can be constructed directly, without the intermediate suffix tree stage. This could lead to a significant reduction in space requirement at the time of VST construction.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch: *Replacing suffix trees with enhanced suffix arrays.* J. Discrete Algorithms, 2(1) 2004, pp. 53–86.
2. D. Adjeroh, T. Bell, and A. Mukherjee: *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching*, Springer, to appear, 2008.
3. D. Adjeroh and F. Nan: *Suffix sorting via Shannon-Fano-Elias codes*, in DCC, IEEE Computer Society, 2008, p. to appear.
4. A. Andersson and S. Nilsson: *Efficient implementation of suffix trees.* Softw., Pract. Exper., 25(2) 1995, pp. 129–141.
5. M. A. Bender and M. Farach-Colton: *The LCA problem revisited.*, in LATIN, G. H. Gonnet, D. Panario, and A. Viola, eds., vol. 1776 of Lecture Notes in Computer Science, Springer, 2000, pp. 88–94.
6. M. Burrows and D. J. Wheeler: *A block-sorting lossless data compression algorithm*, Tech. Rep. 124, Digital Equipment Corporation, Palo Alto, California, May 1994.
7. R. Cole and R. Hariharan: *Faster suffix tree construction with missing suffix links*, in STOC, 2000, pp. 407–415.
8. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan: *On the sorting-complexity of suffix tree construction.* Journal of the ACM, 47(6) 2000, pp. 987–1011.
9. R. Giegerich, S. Kurtz, and J. Stoye: *Efficient implementation of lazy suffix trees.* Software — Practice and Experience, 33(11) 2003.
10. R. Grossi and J. S. Vitter: *Compressed suffix arrays and suffix trees with applications to text indexing and string matching.* SIAM Journal on Computing, 35(2) 2005, pp. 378–407.
11. D. Gusfield: *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
12. J. Kärkkäinen: *Suffix cactus: A cross between suffix tree and suffix array*, in CPM: 6th Symposium on Combinatorial Pattern Matching, 1995.
13. J. Kärkkäinen, P. Sanders, and S. Burkhardt: *Linear work suffix array construction.* Journal of the ACM, 53(6) 2006, pp. 918–936.
14. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *An efficient index data structure with the capabilities of suffix trees and suffix arrays for alphabets of non-negligible size*, in 12th Annual Symposium on Combinatorial Pattern Matching, 2001.
15. D. K. Kim, J. E. Jeon, and H. Park: *An efficient index data structure with the capabilities of suffix trees and suffix arrays for alphabets of non-negligible size*, in SPIRE 2004, 2004.
16. D. K. Kim, M. Kim, and H. Park: *Linearized suffix tree: an efficient index data structure with the capabilities of suffix trees and suffix arrays.* Algorithmica, 2007.
17. D. K. Kim, J. S. Sim, H. Park, and K. Park: *Constructing suffix arrays in linear time.* J. Discrete Algorithms, 3(2-4) 2005, pp. 126–142.
18. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays.* J. Discrete Algorithms, 3(2-4) 2005, pp. 143–156.
19. M. G. Maaβ: *Computing suffix links for suffix trees and arrays.* Information Processing Letters, 101(6) 2007.
20. V. Mäkinen: *Compact suffix array – a space-efficient full-text index.* Fundam. Inform., 56(1-2) 2003, pp. 191–210.
21. U. Manber and E. W. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM Journal on Computing, 22(5) 1993, pp. 935–948.
22. E. M. McCreight: *A space-economical suffix tree construction algorithm.* Journal of the ACM, 23(2) 1976, pp. 262–272.
23. K. Monostori, A. Zaslavsky, and H. Schmidt: *Suffix vector: Space- and time-efficient alternative to suffix trees*, in Twenty-Fifth Australasian Computer Science Conference (ACSC2002), M. J. Oudshoorn, ed., Melbourne, Australia, 2002, ACS.
24. J. I. Munro, V. Raman, and S. S. Rao: *Space efficient suffix trees.* J. Algorithms, 39(2) 2001, pp. 205–222.
25. E. Prieur and T. Lecroq: *From suffix trees to suffix vectors*, in Prague Stringology Conference(PCS2005), Prague, 2005.
26. S. J. Puglisi, W. F. Smyth, and A. Turpin: *A taxonomy of suffix array construction algorithms.* ACM Computing Surveys, 39(2) 2007.
27. E. Ukkonen: *On-line construction of suffix trees.* Algorithmica, 14(3) 1995, pp. 249–260.
28. P. Weiner: *Linear pattern matching algorithm.* Proceedings, 14th IEEE Symposium on Switching and Automata Theory, 21 1973, pp. 1–11.

Parameterized Suffix Arrays for Binary Strings

Satoshi Deguchi[1], Fumihito Higashijima[1],
Hideo Bannai[1], Shunsuke Inenaga[2], and Masayuki Takeda[1]

[1]Department of Informatics, Kyushu University
[2]Graduate School of Information Science and Electrical Engineering, Kyushu University
744 Motooka, Nishiku, Fukuoka 819-0395, Japan
{satoshi.deguchi,bannai,takeda}@i.kyushu-u.ac.jp
inenaga@c.csce.kyushu-u.ac.jp

Abstract. We consider the suffix array for parameterized binary strings that consist of only two types of parameter symbols. We show that the parameterized suffix array, as well as its longest common prefix (LCP) array of such strings can be constructed in linear time. The construction is direct, in that it does not require the construction of a parameterized suffix tree. Although parameterized pattern matching of binary strings can be done by either searching for a pattern and its inverse on a standard suffix array, or constructing two independent standard suffix arrays for the text and its inverse, our approach only needs a single p-suffix array and a single search.

1 Introduction

1.1 Parameterized Pattern Matching

Consider strings over $\Pi \cup \Sigma$, where Π is the set of *parameter symbols* and Σ is the set of *constant symbols*. These strings are called *parameterized strings* (*p-strings*). Baker [7] introduced the notion of *parameterized pattern matching*, where two p-strings of the same length are said to parameterized match (p-match) if one string can be transformed into the other by using a bijection on $\Sigma \cup \Pi$. The bijection should be the identity on the constant symbols of Σ, namely, it maps any $a \in \Sigma$ to a itself, while symbols of Π can be interchanged. Examples of applications of parameterized pattern matching are software maintenance [7,8], plagiarism detection [12], and RNA structural matching [25].

Similar to standard string matching, preprocessing for the text strings is efficient for p-string matching. In [8], Baker proposed the *parameterized suffix tree* (*p-suffix tree*) structure to locate all positions of the text string where a given pattern string p-matches. She presented an $O(n(\pi + \log(\pi + \sigma)))$ time algorithm to construct the p-suffix tree for a given text string of length n. The algorithm uses $O(n)$ space, where $\pi = |\Pi|$ and $\sigma = |\Sigma|$. Kosaraju [22] proposed an improved algorithm for constructing p-suffix trees in $O(n(\log \pi + \log \sigma))$ time. Both algorithms are based on McCreight's algorithm that builds standard suffix trees [24]. Shibuya [25] developed an on-line construction algorithm working in $O(n(\log \pi + \log \sigma))$ time, which is based on Ukkonen's construction algorithm for standard suffix trees [26]. Given a pattern p of length m, we can compute the set $Pocc$ of all positions of t where the corresponding substring of t p-matches pattern p in $O(m \log(\pi + \sigma) + |Pocc|)$ time, by using the p-suffix tree of a text string t.

In this paper, we consider *parameterized suffix arrays* (*p-suffix arrays*), whose relation to p-suffix trees is analogous to the relation between standard suffix arrays [23] and standard suffix trees [27]. As is the case with suffix trees and suffix arrays, the

Satoshi Deguchi, Fumihito Higashijima, Hideo Bannai, Shunsuke Inenaga, Masayuki Takeda : Parameterized Suffix Arrays for Binary Strings, pp. 84–94.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

array representation is superior in terms of memory usage and memory access locality. Also, most operations on a p-suffix tree can be efficiently simulated with the p-suffix array and an array containing the lengths of longest common prefixes of the p-suffixes, which we shall call the PLCP array. For instance, using p-suffix and PLCP arrays, the parameterized pattern matching problem can be solved in $O(m \log n + |Pocc|)$ time with a simple binary search, or $O(m + \log n + |Pocc|))$ with a binary search utilizing PLCP information, or $O(m \log(\pi + \sigma) + |Pocc|)$ time if we consider an *enhanced* p-suffix array [1,18].

P-suffix arrays and PLCP arrays can be obtained from a simple linear time traversal of the corresponding p-suffix trees. However, unlike the case of standard suffix arrays [16,19,21], linear time algorithms for *direct* construction of parameterized suffix arrays are not known so far.

In this paper, we take a first step in this problem, and show that for any text p-string t over binary parameter alphabet Π, p-suffix arrays and PLCP arrays can be constructed directly in $O(n)$ time. Our construction algorithm does not need the p-suffix tree of t as an intermediate structure.

There is a naïve solution to the p-matching problem for a binary alphabet using two standard suffix arrays. Given a text t and pattern p over $\Pi = \{a, b\}$, compute another text t' by exchanging a and b in t. Then, compute two suffix arrays for t and t', and search the arrays for pattern p. Or alternatively, compute another pattern p' by exchanging a and b in p, and search for the array of t for p and p'. On the other hand, our approach only needs a *single* p-suffix array and a *single* search for p-matching, and thus is both space- and time-efficient. We also performed some experiments to show the efficiency of our p-suffix arrays.

Parameterized strings on binary parameter alphabet were investigated in literature. Apostolico and Giancarlo [6] pointed out that parameterized strings over a binary parameter alphabet behave in a similar way to standard strings with respect to periodicity and repetitions, but the case with larger parameter alphabet remains open. Our result on the direct linear-time construction of p-suffix arrays for a binary alphabet is yet another one showing the similarity of parameterized strings on a binary alphabet to standard strings.

1.2 Related Work

Another approach for solving the parameterized pattern matching is to preprocess patterns. Idury and Schäffer [15] proposed a variation of the Aho-Corasick automaton [2], which can be constructed in $O(m \log(\pi + \sigma))$ time for a single pattern, and scanning the text takes $O(n \log(\pi + \sigma))$ time. Amir et al. [4] presented a KMP algorithm [20] based approach with $O(m \log(\mathbf{min}\{m, \pi\}))$ preprocessing time and $O(n \log(\mathbf{min}\{m, \pi\}))$ scanning time. They also stated that it suffices to consider strings over Π rather than $\Pi \cup \Sigma$ for the p-matching problem, showing that the problem with $\Pi \cup \Sigma$ can be reduced in linear time to that with Π.

Parameterized pattern matching has been extended to two dimensional parameterized pattern matching [3,14] and approximate parameterized pattern matching [13,5]. Parameterized edit distance was considered in [9].

2 Preliminaries

Let Σ and Π be two disjoint finite sets of *constant symbols* and *parameter symbols*, respectively. An element of $(\Sigma \cup \Pi)^*$ is called a *p-string*. The length of any p-string

s is the total number of constant and parameter symbols in s and is denoted by $|s|$. For any p-string s of length n, the i-th symbol is denoted by $s[i]$ for each $1 \leq i \leq n$, and the *substring* starting at position i and ending at position j is denoted by $s[i:j]$ for $1 \leq i \leq j \leq n$. In particular, $s[1:j]$ and $s[i:n]$ denote the *prefix* of length j and the *suffix* of length $n-i+1$ of s, respectively. For any two p-strings s and t, $lcp(s,t)$ denotes the length of the *longest common prefix* of s and t.

Definition 1 (Parameterized Matching). *Any two p-strings s and t of the same length m are said to* parameterized match *(*p-match*) iff one of the following conditions hold for every $1 \leq i \leq m$:*

1. $s[i] = t[i] \in \Sigma$,
2. $s[i], t[i] \in \Pi$, $s[i] \neq s[j]$ *and* $t[i] \neq t[j]$ *for any* $1 \leq j < i$,
3. $s[i], t[i] \in \Pi$, $s[i] = s[i-k]$ *for any* $1 \leq k < i$ *iff* $t[i] = t[i-k]$.

We write $s \simeq t$ when s and t p-match.

For example, let $\Pi = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $\Sigma = \{\mathtt{X}, \mathtt{Y}\}$, $s = \mathtt{abaXabY}$ and $t = \mathtt{bcbXbcY}$. Observe that $s \simeq t$.

Let $\mathcal{N}$ be the set of non-negative integers. For any non-negative integers $i \leq j \in \mathcal{N}$, let $[i,j] = \{i, i+1, \ldots, j\} \subset \mathcal{N}$.

Definition 2. *We define $pv : (\Sigma \cup \Pi)^* \to (\Sigma \cup \mathcal{N})^*$ to be the function such that for any p-string s of length n, $pv(s) = u$ where, for $1 \leq i \leq n$,*

$$u[i] = \begin{cases} s[i] & \text{if } s[i] \in \Sigma, \\ 0 & \text{if } s[i] \in \Pi \text{ and } s[i] \neq s[j] \text{ for any } 1 \leq j < i, \\ i-k & \text{if } s[i] \in \Pi \text{ and } k = \boldsymbol{max}\{j \mid s[i] = s[j], 1 \leq j < i\}. \end{cases}$$

In the running example, $pv(s) = 002\mathtt{X}24\mathtt{Y}$ with $s = \mathtt{abaXabY}$.

The following proposition is clear from Definition 2.

Proposition 3. *For any p-string s of length n, it holds for any $1 \leq i \leq j \leq n$ that*

$$pv(s[i:j]) = v[1:j-i+1],$$

where $v = pv(s[i:n])$.

The pv function is useful for p-matching, because:

Proposition 4 ([8]). *For any two p-strings s and t of the same length, $s \simeq t$ iff $pv(s) = pv(t)$.*

In the running example, we then have $s \simeq t$ and $pv(s) = pv(t) = 002\mathtt{X}24\mathtt{Y}$.

We also define the dual of the pv function, as follows:

Definition 5. *We define $fw : (\Sigma \cup \Pi)^* \to (\Sigma \cup \mathcal{N} \cup \{\infty\})^*$ to be the function such that for any p-string s of length n, $fw(s) = w$ where, for $1 \leq i \leq n$,*

$$w[i] = \begin{cases} s[i] & \text{if } s[i] \in \Sigma, \\ \infty & \text{if } s[i] \in \Pi \text{ and } s[i] \neq s[j] \text{ for any } i < j \leq n, \\ k-i & \text{if } s[i] \in \Pi \text{ and } k = \boldsymbol{min}\{j \mid s[i] = s[j], i < j \leq n\}. \end{cases}$$

Here, ∞ denotes a value for which $i < \infty$ for any $i \in \mathcal{N}$. [1]

[1] In practice, n can be used in place of ∞ as long as we are considering a single p-string of length n, and its substrings.

In the running example, $fw(s) = 242\mathtt{X}\infty\infty\mathtt{Y}$ with $s = \mathtt{abaXabY}$.

The following proposition is clear from Definition 5.

Proposition 6. *For any p-string s of length n, it holds for any $1 \leq i \leq n$ that*

$$fw(s[i:n]) = w[i:n],$$

where $w = fw(s)$.

For any p-string s of length n, $pv(s)$ and $fw(s)$ can be computed in $O(n)$ time with extra $O(\pi)$ space, using a table of size π recording the last position of each parameter symbol in the left-to-right (resp. right-to-left) scanning of s [8].

3 P-matching Problem and P-suffix Arrays

In this section we introduce a new data structure *p-suffix arrays* that are useful to solve the p-matching problem given below.

3.1 Problem

The problem considered in this paper is the following:

Problem 7 (P-matching problem). Given any two p-strings t and p of length n and m respectively, $n \geq m$, compute $Pocc(t,p) = \{i \mid t[i:i+m-1] \simeq p\}$.

It directly follows from Proposition 4 that

$$Pocc(t,p) = \{i \mid pv(p) = pv(t[i:i+m-1])\}.$$

Therefore, from Proposition 3, we are able to compute $Pocc(t,p)$ efficiently, by indexing all elements of the set $\{pv(t[i:n]) \mid 1 \leq i \leq n\}$. The corresponding indexing structure is introduced in the next subsection.

Amir et al. [4] showed that actually we have only to consider p-strings from Π to solve Problem 7, as follows.

Lemma 8 ([4]). *Problem 7 on alphabet $\Sigma \cup \Pi$ is reducible in linear time to Problem 7 on alphabet Π.*

Hence, in the remainder of the paper, we consider only p-strings in Π^*. Then, note that for any p-string s of length n, $pv(s) \in \{[0, n-1]\}^n$ and $fw(s) \in \{[1, n-1] \cup \{\infty\}\}^n$. We also see that if $pv(s)[i] > 0$ then $fw(s)[i - pv(s)[i]] = pv(s)[i]$. Similarly, if $fw(s)[i] < n$ then $pv(s)[i + fw(s)[i]] = fw(s)[i]$.

3.2 P-suffix Arrays

In this section we introduce our data structure p-suffix arrays. Let us begin with normal suffix arrays [23] that have widely been used for standard pattern matching.

Let $\preceq_+$ and $\preceq_-$ denote the total order and its reverse on integers, i.e., for integers $x, y \in \mathcal{N} \cup \{\infty\}$, $x \preceq_+ y \iff: x \leq y$ and $x \preceq_- y \iff: x \geq y$. The lexicographic ordering on strings of an integer alphabet $[i,j] \cup \{\infty\}$ with respect to a total order $\preceq$ on integers can be defined as follows. For $x, y \in ([i,j] \cup \{\infty\})^*$,

$$x \preceq y \iff: \begin{cases} x \text{ is a prefix of } y, \text{ or} \\ \exists \alpha, u, v \in ([i,j] \cup \{\infty\})^*, a, b \in [i,j] \cup \{\infty\}, \\ \qquad \text{such that } a \prec b,\ x = \alpha a u,\ y = \alpha b v. \end{cases}$$

We define a variation of suffix arrays on the integer alphabet $[1, n-1] \cup \{\infty\}$ and the lexicographical ordering of suffixes w.r.t. $\preceq_-$.

Definition 9 (Suffix Arrays). *For any string $w \in ([1, n-1] \cup \{\infty\})^n$ of length n, its* suffix array *SA_w is an array of length n such that $SA_w[i] = j$, where $w[j : n]$ is the lexicographically i-th suffix of w w.r.t. $\preceq_-$.*

We abbreviate SA_w as SA when clear from the context.

Definition 10 (LCP Arrays). *For any string $w \in ([1, n-1] \cup \{\infty\})^n$ of length n, its* LCP array *LCP_w is an array of length n such that*

$$LCP_w[i] = \begin{cases} -1 & \text{if } i = 1, \\ lcp(w[SA[i-1] : n], w[SA[i] : n]) & \text{if } 2 \le i \le n. \end{cases}$$

We abbreviate LCP_w as LCP when clear from the context.

Remark 11. We emphasize that suffix array $SA_{fw(t)}$ of p-string text $t \in \Pi^*$ does *not* solve Problem 7 directly. Let $p \in \Pi^*$ be a p-string pattern. A careful consideration reveals that an ordinary binary search on $SA_{fw(t)}$ for $fw(p)$ does not work, in that we may miss some occurrences of $fw(p)$ in $fw(t)$. The suffix tree for $fw(t)$ is not useful either, since a node of the tree can have $O(n)$ children and searching the tree for $fw(p)$ requires $O(n^m)$ time, where $n = |t|$ and $m = |p|$. Thus the combination of $SA_{fw(t)}$ and $LCP_{fw(t)}$ does not efficiently solve Problem 7, either. Interestingly, however, $SA_{fw(t)}$ and $LCP_{fw(t)}$ are very helpful to construct the following data structures which provide us with efficient solutions to the problem, to be shown in Section 4.

In order to solve Problem 7 efficiently, we use the two following data structures corresponding to $pv(s)$.

Definition 12 (P-suffix Arrays). *For any p-string $s \in \Pi^n$ of length n, its* p-suffix array *PSA_s is an array of length n such that $PSA_s[i] = j$, where $pv(s[j : n])$ is the lexicographically i-th element of $\{pv(s[i : n]) \mid 1 \le i \le n\}$ w.r.t. $\preceq_+$.*

We abbreviate PSA_s as PSA when clear from the context. Note that PSA_s is not necessarily equal to SA_s, since $pv(s[i : n])$ may not always be a suffix of $pv(s)$.

The following function is useful for p-matching with PSA.

Definition 13. *We define $short : \mathcal{N}^* \to \mathcal{N}^*$ to be the function such that for any string $x \in \mathcal{N}^n$ of length n, $short(x) = y$ where, for $1 \le i \le n$,*

$$y[i] = \begin{cases} x[i] & \text{if } x[i] < i, \\ 0 & \text{if } x[i] \ge i. \end{cases}$$

Lemma 14 ([15]). *For any p-string $s \in \Pi^n$ of length n, let $v = pv(s)[i : n]$ for any $1 \le i \le n$. Then, $short(v) = pv(s[i : n])$.*

Lemma 14 implies that, when using PSA_s, we do not have to store $pv(s[i : n])$ for all $1 \le i \le n$; only $pv(s)$ is sufficient.

Theorem 15. *Problem 7 can be solved in $O(m \log n + |Pocc(t, p)|)$ time by using PSA_t and $pv(t)$.*

Proof. By Proposition 3 and Lemma 14, we can compute $Pocc(t,p)$ by a binary search on PSA_t and $pv(t)$, which takes $O(m \log n + |Pocc(t,p)|)$ time in total. $\square$

The following auxiliary array enables us to solve Problem 7 more efficiently.

Definition 16 (PLCP Arrays). *For any p-string $s \in \Pi^n$ of length n, its* PLCP array *$PLCP_s$ is an array of length n such that*

$$PLCP_s[i] = \begin{cases} -1 & \textit{if } i = 1, \\ lcp(pv(s[PSA[i-1]:n]), pv([s[PSA[i]:n])) & \textit{if } 2 \le i \le n. \end{cases}$$

We abbreviate $PLCP_s$ as $PLCP$ when clear from the context.

Using $PLCP$, we can achieve an improved solution, as follows:

Theorem 17. *Problem 7 can be solved in $O(m + \log n + |Pocc(t,p)|)$ time by using PSA_t, $PLCP_t$ and $pv(t)$.*

Proof. The time complexity can be improved to $O(m+\log n+|Pocc(t,p)|)$ by a similar manner to [23] for standard suffix and LCP arrays. $\square$

Considering the *enhanced* [1] p-suffix array, we obtain the following bound:

Theorem 18. *Problem 7 can be solved in $O(m \log \pi + |Pocc(t,p)|)$ time.*

Proof. For any p-string $t \in \Pi^*$, the number of children of any internal node of the p-suffix tree for $pv(t)$ is at most π [8]. Hence the enhanced p-suffix array enables us to solve Problem 7 in $O(m + \log \pi + |Pocc(t,p)|)$ time (see [18] for more details). $\square$

In the next section, we present our algorithm to construct PSA and $PLCP$ arrays for binary strings, that is, for the case where $\pi = 2$. Our algorithm runs in linear time, and uses SA and LCP arrays.

4 P-Suffix and PLCP Arrays of Binary P-strings

In this section, we will show that for any binary p-string s, its p-suffix array PSA_s and PLCP array $PLCP_s$ can be computed in linear time, directly from s without the use of p-suffix trees.

We first show that the p-suffix array PSA_s of a binary p-string s is equivalent to the suffix array $SA_{fw(s)}$ of $fw(s)$. Then, we show the relationship between $PLCP_s$ and $LCP_{fw(s)}$, so that $PLCP_s$ can be calculated from $fw(s)$ and $LCP_{fw(s)}$.

Figure 1 shows PSA_s, $PLCP_s$ for $s =$ `abaabaaaabba` and corresponding suffixes of $fw(s)$ and $LCP_{fw(s)}$.

Lemma 19. *For any p-string s and $1 \le i \le n$, $|\{j \mid i = j - pv(s)[j]\}| \le 1$ and $|\{j \mid i = j + fw(s)[j]\}| \le 1$.*

Proof. Let $i = x - pv(s)[x] = y - pv(s)[y]$ for some $i < x < y$. Then, by definition, $s[i] = s[x] = s[y]$ and $y - i = pv(s)[y] = y - \mathbf{max}\{j \mid s[y] = s[j], 1 \le j \le y\} \le y - x$, which is a contradiction. Similar arguments hold for $fw(s)$.

Lemma 20. *For any p-strings $s, t \in \Pi^*$ with $\pi = 2$, $pv(s) \preceq_+ pv(t)$ if and only if $fw(s) \preceq_- fw(t)$.*

i	$PSA[i]$	$s[PSA[i]:n]$	$pv(s[PSA[i]:n])$	$PLCP_s[i]$	$fw(s[PSA[i]:n])$	$LCP_{fw(s)}[i]$
1	12	a	0	-1	∞	-1
2	11	ba	0 0	1	$\infty\infty$	1
3	5	baaaabba	0 0 1 1 1 $\underline{5}$ 1 3	2	$\underline{5}$ 1 1 1 3 1 $\infty\infty$	0
4	9	abba	0 0 1 $\underline{3}$	3	$\underline{3}$ 1 $\infty\infty$	0
5	2	baabaaaabba	0 0 1 $\underline{3}$ 2 1 1 1 5 1 3	4	$\underline{3}$ 1 2 5 1 1 1 3 1 $\infty\infty$	2
6	4	abaaaabba	0 0 $\underline{2}$ 1 1 1 5 1 3	2	$\underline{2}$ 5 1 1 1 3 1 $\infty\infty$	0
7	1	abaabaaaabba	0 0 $\underline{2}$ 1 3 2 1 1 1 5 1 3	4	$\underline{2}$ 3 1 2 5 1 1 1 3 1 $\infty\infty$	1
8	10	bba	0 $\underline{1}$ 0	1	$\underline{1}$ $\infty\infty$	0
9	8	aabba	0 $\underline{1}$ 0 1 3	3	$\underline{1}$ 3 1 $\infty\infty$	1
10	3	aabaaaabba	0 $\underline{1}$ 0 2 1 1 1 5 1 3	3	$\underline{1}$ 2 5 1 1 1 3 1 $\infty\infty$	1
11	7	aaabba	0 $\underline{1}$ 1 0 1 3	2	$\underline{1}$ 1 3 1 $\infty\infty$	1
12	6	aaaabba	0 $\underline{1}$ 1 1 0 1 3	3	$\underline{1}$ 1 1 3 1 $\infty\infty$	2

Figure 1. The p-suffix array PSA of $s = \mathtt{abaabaaaabba}$ and corresponding suffixes of $fw(s)$, as well as the PLCP and LCP values. Here, $n = 12$.

Proof. It is clear that $pv(s) = pv(t)$ iff $fw(s) = fw(t)$. Let us now consider the other case.
($\Longrightarrow$) Assume $pv(s) \prec_+ pv(t)$. If $pv(s)$ is a prefix of $pv(t)$, then $fw(s) \prec_- fw(t)$ since

$$fw(s)[i] = fw(t[1:|s|])[i] = \begin{cases} fw(t)[i] & i + fw(t)[i] \leq |s| \\ \infty & \text{otherwise,} \end{cases}$$

for all $1 \leq i \leq |s|$. Next, assume that $pv(s)$ is not a prefix of $pv(t)$, and let $i = \mathbf{min}\{j \mid pv(s)[j] \prec_+ pv(t)[j]\}$, $\ell = pv(t)[i]$ and $r = pv(s)[i]$. Then, we have $t[i-\ell] = t[i] \neq t[k]$ for any $i - \ell < k < i$. Therefore, we get $fw(t)[i-\ell] = \ell$. On the other hand, we have $s[i-\ell] \neq s[i] = s[k]$ for any $i - \ell < k < i$, since $\pi = 2$. Hence $fw(s)[i-\ell] > \ell$, which implies that $fw(s)[i-\ell] > fw(t)[i-\ell]$. Thus if $i - \ell = 1$, then clearly $fw(s) \prec_- fw(t)$. Now we consider the case where $i - \ell > 1$. For any $1 \leq h < i - \ell$, we have

$$fw(s)[h] \leq \begin{cases} i - \ell - h & \text{if } fw(s)[h] = fw(s)[i-\ell] \text{ or } r = 0, \\ i - \ell + 1 - h & \text{otherwise,} \end{cases}$$

where the second case comes from the fact that $\pi = 2$. Note that the same inequality stands for t. This implies that there exists $1 \leq p, q \leq i-1$ such that $h = p - pv(s)[p] = q - pv(t)[q]$. From the assumption, $pv(s)[1:i-1] = pv(t)[1:i-1]$ and Lemma 19, we have $p = q$ and hence, $fw(s)[h] = fw(t)[h]$. Therefore, $fw(s)[1:i-\ell-1] = fw(t)[1:i-\ell-1]$, and consequently $fw(s) \prec_- fw(t)$.
($\Longleftarrow$) Assume $fw(s) \prec_- fw(t)$. If $fw(s)$ is a prefix of $fw(t)$, $fw(s) = fw(t)[1:|s|] = fw(t[1:|s|)$. Then $pv(s) = pv(t[1:|s|]) = pv(t)[1:|s|]$, and therefore $pv(s) \prec_+ pv(t)$. Next, assume $fw(s)$ is not a prefix of $fw(t)$, and let $l = lcp(fw(s), fw(t)) < \mathbf{min}\{|s|, |t|\}$. Then, since $fw(s)[1:l] = fw(t)[1:l]$, we have $fw(s[1:l]) = fw(t[1:l])$, and $pv(s[1:l]) = pv(t[1:l])$, and finally $pv(s)[1:l] = pv(t)[1:l]$. Furthermore, $pv(s)[l+1] = pv(t)[l+1]$ holds, since either there exists $1 \leq j \leq l$ such that $j + fw(s)[j] = j + fw(t)[j] = l + 1$ in which case $pv(s)[l+1] = pv(t)[l+1]$, or there doesn't, in which case $pv(s)[l+1] = pv(t)[l+1] = 0$. Therefore, assume $|s| \geq l + 2$ since otherwise the proof is finished.

Let $p = fw(s)[l+1]$ and $q = fw(t)[l+1]$. From the assumption, $\infty \geq p > q \geq 1$. Since $\pi = 2$, $t[l+1] = t[l+1+q] \neq t[k]$ for any $l+1 < k < l+1+q$, and $s[l+1] \neq s[k]$ for any $l+1 < k \leq \mathbf{min}\{|s|, l+1+q\}$. If $q = 1$, then $pv(s)[l+2] = 0$ since by Lemma 19, there cannot exist $1 \leq j \leq l$ such that $j + fw(s)[j] = j + fw(t)[j] = l+1$. If $q \geq 2$, this gives us $pv(s)[l+2] = pv(t)[l+2]$, $pv(t)[k] = 1$ for any $l+2 < k < l+1+q$, and $pv(s)[k] = 1$ for any $l+2 < k \leq \mathbf{min}\{|s|, l+1+q\}$, while $pv(t)[l+1+q] = fw(t)[l+1] = q \geq 2$. Either way, we have $pv(s) \prec_+ pv(t)$. ◻

The next lemma is a direct consequence of Lemma 20.

Lemma 21. *For any p-string $s \in \Pi^*$ with $\pi = 2$, $PSA_s = SA_{fw(s)}$.*

It is well known that the suffix array can be constructed directly from the string in linear time.

Theorem 22 ([16,19,21]). *For any string $w \in ([1,n])^n$ of length n, SA_w can be directly constructed in $O(n)$ time.*

The next theorem follows from Lemma 21 and Theorem 22:

Theorem 23. *For any p-string $s \in \Pi^n$ of length n with $\pi = 2$, PSA_s can be constructed directly in $O(n)$ time by constructing $SA_{fw(s)}$.*

From now on let us consider construction of $PLCP_s$.

Lemma 24. *For any p-strings $s, t \in \Pi^*$ with $\pi = 2$,*

$$lcp(pv(s), pv(t)) = \begin{cases} l & l = k, \\ \boldsymbol{min}\{k, l + \boldsymbol{min}\{fw(s)[l+1], fw(t)[l+1]\}\} & \text{otherwise,} \end{cases}$$

where $l = lcp(fw(s), fw(t))$, and $k = \boldsymbol{min}\{|s|, |t|\}$.

Proof. By similar arguments as in ($\Longleftarrow$) of Lemma 20. ◻

It is well known that the LCP array for strings can be constructed efficiently from its corresponding suffix array.

Theorem 25 ([17]). *For any string s of length n, LCP_s can be constructed in $O(n)$ time, given s and its suffix array SA_s.*

Due to Lemma 24 and Theorem 25, we have:

Theorem 26. *For any p-string $s \in \Pi^n$ of length n with $\pi = 2$, $PLCP_s$ can be constructed in $O(n)$ time from $PSA_s = SA_{fw(s)}$ and $fw(s)$.*

5 Computational Experiments

Here, we consider parameterized pattern matching on binary strings. We compare the method using parameterized suffix arrays and two naïve methods that either uses two patterns or two suffix arrays. We run our algorithm for various pattern and text lengths for random binary strings.

Figure 2 shows the time for p-matching not including the p-suffix array construction for random texts of length 100 and 1000. The average of 1000 p-matchings for the same text and pattern strings is further averaged by 100 runs for different random strings. We can see that our approach is the fastest for short patterns. However, the overhead for creating $pv(p)$ for pattern p seems to take over, when p becomes longer.

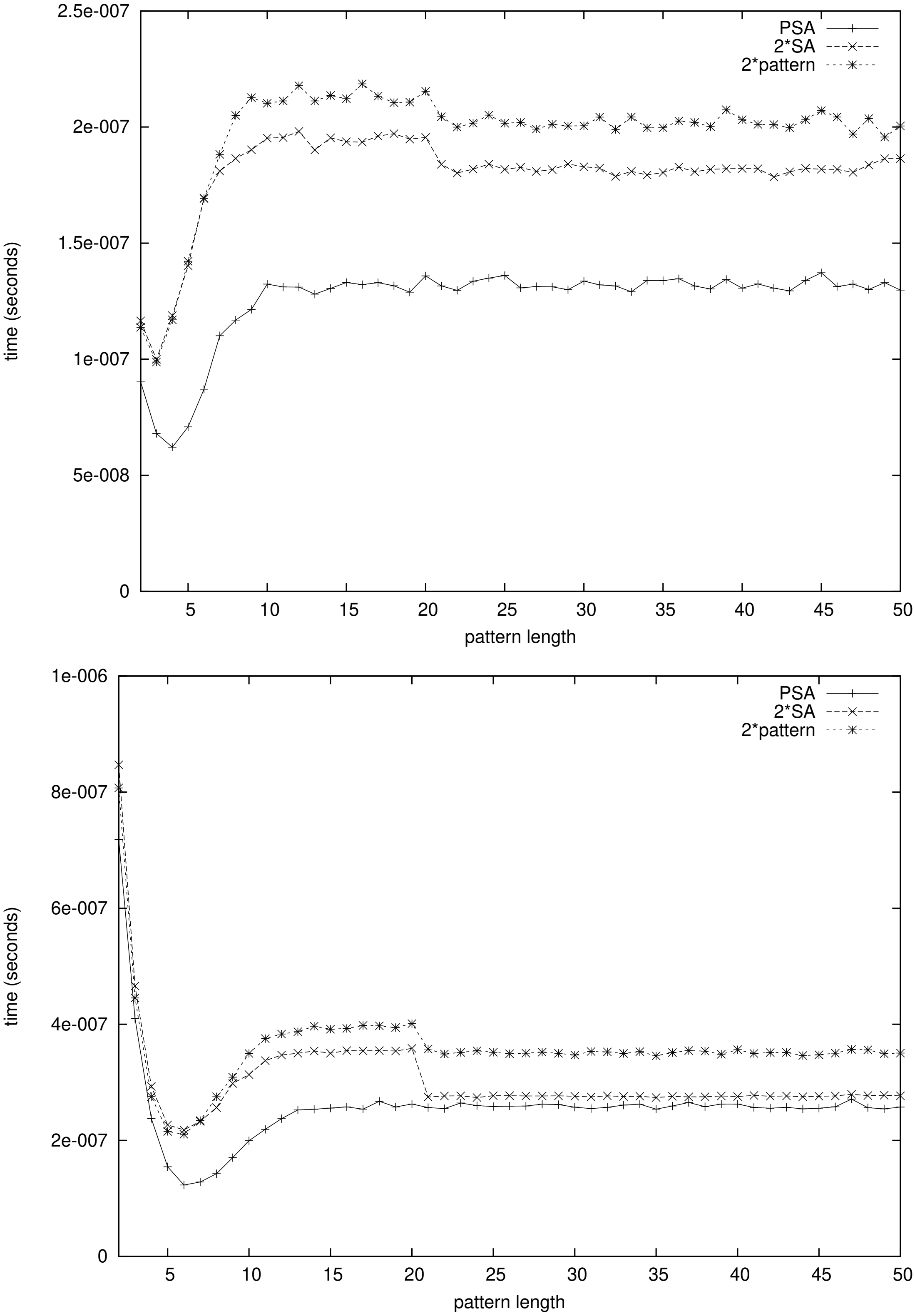


Figure 2. Comparison of running times for p-matching on random binary strings. The length of the text is 100 (upper) and 1000 (lower). The increase in time for short patterns is due to the increase of $|Pocc|$.

6 Conclusion and Future Work

We showed that p-suffix arrays and PLCP arrays for binary strings can be constructed in linear time. It is an open problem whether or not the parameterized suffix array and PLCP array for larger alphabets can be constructed directly in linear time. It is difficult to apply standard suffix array algorithms or LCP calculation algorithms, since an important property does not hold for p-strings. Namely, a suffix $pv(s)[i:n]$ of $pv(s)$ is not necessarily equal to the $pv(s[i:n])$ of the suffix $s[i:n]$. As an important consequence, for any p-strings s,t with $lcp(pv(s),pv(t)) > 0$, $pv(s) \preceq_+ pv(t)$ does not necessarily imply $pv(s[2:|s|]) \preceq_+ pv(t[2:|t|])$ which is essential in the standard case.

For similar reasons, the reverse problem of finding a p-string whose p-suffix array is equal to a given array of integers also does not seem to be as simple as in the case for standard suffix arrays [11,10], and is another open problem.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch: *Replacing suffix trees with enhanced suffix arrays.* Journal of Discrete Algorithms, 2(1) 2004, pp. 53–86.
2. A. V. Aho and M. J. Corasick: *Efficient string matching: An aid to bibliographic search.* Communications of the ACM, 18(6) 1975, pp. 333–340.
3. A. Amir, Y. Aumann, R. Cole, M. Lewenstein, and E. Porat: *Function matching: Algorithms, applications, and a lower bound*, in Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP'03), vol. 2719 of Lecture Notes in Computer Science, 2003, pp. 929–942.
4. A. Amir, M. Farach, and S. Muthukrishnan: *Alphabet dependence in parameterized matching.* Information Processing Letters, 49(3) 1994, pp. 111–115.
5. A. Apostolico, P. L. Erdös, and M. Lewenstein: *Parameterized matching with mismatches.* Journal of Discrete Algorithms, 5(1) 2007, pp. 135–140.
6. A. Apostolico and R. Giancarlo: *Periodicity and repetitions in parameterized strings.* Discrete Applied Mathematics, 156(9) 2008, pp. 1389–1398.
7. B. S. Baker: *A program for identifying duplicated code.* Computing Science and Statistics, 24 1992, pp. 49–57.
8. B. S. Baker: *Parameterized pattern matching: Algorithms and applications.* Journal of Computer and System Sciences, 52(1) 1996, pp. 28–42.
9. B. S. Baker: *Parameterized diff*, in Proc. 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99), 1999, pp. 854–855.
10. H. Bannai, S. Inenaga, A. Shinohara, and M. Takeda: *Inferring strings from graphs and arrays*, in Proc. 28th International Symposium on Mathematical Foundations of Computer Science (MFCS 2003), vol. 2747 of Lecture Notes in Computer Science, 2003, pp. 208–217.
11. J.-P. Duval and A. Lefebvre: *Words over an ordered alphabet and suffix permutations.* Theoretical Informatics and Applications, 36 2002, pp. 249–259.
12. K. Fredriksson and M. Mozgovoy: *Efficient parameterized string matching.* Information Processing Letters, 100(3) 2006, pp. 91–96.
13. C. Hazay, M. Lewenstein, and D. Sokol: *Approximate parameterized matching.* ACM Transactions on Algorithms, 3(3) 2007, Article No. 29.
14. C. Hazay, M. Lewenstein, and D. Tsur: *Two dimensional parameterized matching*, in Proc. 16th Annual Symposium on Combinatorial Pattern Matching (CPM'05), vol. 3537 of Lecture Notes in Computer Science, 2005, pp. 266–279.
15. R. M. Idury and A. A. Schäffer: *Multiple matching of parameterized patterns.* Theoretical Computer Science, 154(2) 1996, pp. 203–224.
16. J. Kärkkäinen and P. Sanders: *Simple linear work suffix array construction*, in Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP'03), vol. 2719 of Lecture Notes in Computer Science, 2003, pp. 943–955.

17. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *Linear-time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications*, in Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01), vol. 2089 of Lecture Notes in Computer Science, 2001, pp. 181–192.
18. D. K. Kim, J. E. Jeon, and H. Park: *An efficient index data structure with the capabilities of suffix trees and suffix arrays for alphabets of non-negligible size*, in Proc. 11th International Symposium on String Processing and Information Retrieval (SPIRE'04), vol. 3246 of Lecture Notes in Computer Science, 2004, pp. 138–149.
19. D. K. Kim, J. S. Sim, H. Park, and K. Park: *Linear-time construction of suffix arrays*, in Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM'03), vol. 2676 of Lecture Notes in Computer Science, 2003, pp. 186–199.
20. D. E. Knuth, J. H. Morris, and V. R. Pratt: *Fast pattern matching in strings*. SIAM J. Comput., 6(2) 1977, pp. 323–350.
21. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*, in Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM'03), vol. 2676 of Lecture Notes in Computer Science, 2003, pp. 200–210.
22. S. Kosaraju: *Faster algorithms for the construction of parameterized suffix trees*, in Proc. 36th Annual Symposium on Foundations of Computer Science (FOCS'95), 1995, pp. 631–637.
23. U. Manber and G. Myers: *Suffix arrays: a new method for on-line string searches*. SIAM J. Computing, 22(5) 1993, pp. 935–948.
24. E. M. McCreight: *A space-economical suffix tree construction algorithm*. Journal of the ACM, 23(2) 1976, pp. 262–272.
25. T. Shibuya: *Generalization of a suffix tree for RNA structural pattern matching*. Algorithmica, 39(1) 2004, pp. 1–19.
26. E. Ukkonen: *On-line construction of suffix trees*. Algorithmica, 14(3) 1995, pp. 249–260.
27. P. Weiner: *Linear pattern-matching algorithms*, in Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory, 1973, pp. 1–11.

An Adaptive Hybrid Pattern-Matching Algorithm on Indeterminate Strings*

William F. Smyth, Shu Wang, and Mao Yu

Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton ON L8S 4K1, Canada
{smyth,shuw,yum5}@mcmaster.ca

Abstract. We describe a hybrid pattern-matching algorithm that works on both regular and indeterminate strings. This algorithm is inspired by the recently proposed hybrid algorithm FJS [11] and its indeterminate successor [15]. However, as discussed in this paper, because of the special properties of indeterminate strings, it is not straightforward to directly migrate FJS to an indeterminate version. Our new algorithm combines two fast pattern-matching algorithms, ShiftAnd and BMS (the Sunday variant of the Boyer-Moore algorithm), and is highly adaptive to the nature of the text being processed. It avoids using the border array, therefore avoids some of the cases that are awkward for indeterminate strings. Although not always the fastest in individual test cases, our new algorithm is superior in overall performance to its two component algorithms — perhaps a general advantage of hybrid algorithms.

1 Introduction

String pattern-matching has been studied extensively for many years because of the fundamental role it plays in many areas: the operation of a text editor or compiler, bioinformatics, data compression, firewall interception, and so on. Two main approaches have been proposed for computing all the occurrences of a given nonempty pattern $p = p[1..m]$ in a given nonempty text $x = x[1..n]$. One is the use of window-shifting techniques to skip over sections of text [17,8], the other the use of the bit-parallel processing capability of computers to achieve fast processing [10,23,7,18]. For more complete descriptions of various string matching algorithms, see [19,9,20].

Driven by applications in DNA sequence analysis and search engine techniques, indeterminate pattern-matching (IPM) is becoming more and more widely used. But for this modifications have to be made. An intuitive approach to IPM is to make use of exact pattern-matching algorithms and make necessary changes. Some pattern-matching algorithms that use bit-array methods such as ShiftAnd[23] and BNDM [18] can be adapted to IPM. On the other hand, efforts have also been made to develop indeterminate pattern-matching algorithms that are based on fast window-shifting algorithms such as BMS (the Sunday variant of the Boyer-Moore algorithm) [14] and FJS [15]. In this paper, we present a new fast algorithm that not only works on regular strings but also on indeterminate strings — it inherits from the BMS and ShiftAnd algorithms, while exceeding both of them in overall performance.

We believe that this paradigm will lead to the design of other very efficient IPM algorithms with the ability to flip-flop seamlessly between two or more methods, in response to the changing nature of local segments of the text.

* Supported in part by grants from the Natural Sciences & Engineering Research Council of Canada. The authors express their gratitude to three anonymous referees, whose comments have materially improved the quality of this paper.

William F. Smyth, Shu Wang, Mao Yu: An Adaptive Hybrid Pattern-Matching Algorithm on Indeterminate Strings , pp. 95–107.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

2 Preliminaries

A ***string*** x is a finite sequence of ***letters*** drawn from a set Σ called an ***alphabet***. Let λ_i, $|\lambda_i| \geq 2$, $1 \leq i \leq m$, be pairwise distinct subsets of the alphabet Σ. We form a new alphabet $\Sigma' = \Sigma \cup \{\lambda_1, \lambda_2, .., \lambda_m\}$ and define a new relation ***match*** ($\approx$) on Σ' as follows:

- for every $\mu_1, \mu_2 \in \Sigma$, $\mu_1 \approx \mu_2$ if and only if $\mu_1 = \mu_2$;
- for every $\mu \in \Sigma$ and every $\lambda \in \Sigma' - \Sigma$, $\mu \approx \lambda$ and $\lambda \approx \mu$ if and only if $\mu \in \lambda$;
- for every $\lambda_i, \lambda_j \in \Sigma' - \Sigma$, $\lambda_i \approx \lambda_j$ if and only if $\lambda_i \cap \lambda_j \neq \emptyset$.

In a string x on an alphabet Σ', a position i is said to be ***indeterminate*** iff $x[i] \in \Sigma' - \Sigma$, and $x[i]$ itself is said to be an ***indeterminate letter***. A string that may contain indeterminate letters is said to be ***indeterminate*** (or ***generalized*** [5]). Two indeterminate strings x and y are said to ***match*** iff they are of the same length and the letters in corresponding positions match.

Indeterminate strings can arise in DNA and amino acid sequences as well as in cryptoanalysis applications and the analysis of musical texts. A simple example of an indeterminate letter is the don't-care letter $*$ which matches any other letter in the alphabet.

We identify three models of IPM in increasing order of sophistication:

(M1) The only indeterminate letter is the don't-care $*$, whose occurrences may be in either patterns or strings, or both.
(M2) Arbitrary indeterminate letters can occur, but only in patterns (or only in texts).
(M3) Indeterminate letters can occur in both patterns and strings.

In addition, two different constraints can be imposed on IPM:

- Quantum (**q**). Allow an indeterminate letter to match two or more distinct letters during a single matching process.
- Determinate (**d**). Restrict each indeterminate letter to be assigned to only one regular letter during a single matching process.

For example, given two strings $u = 551, v = 121$ including one indeterminate letter $5 = \{1, 2\}$, does $u \approx v$? The answer is yes in quantum pattern-matching and no in determinate pattern-matching, because we require that 5 first match 1 and then match 2 in a single match between 551 and 121.

Combining the three models and the two constraints q and d, we identify six interesting versions of IPM:

$$\text{M1q, M1d, M2q, M2d, M3q, M3d.} \tag{1}$$

3 Nontransitivity of Indeterminate Matching

In this section we briefly discuss a central problem that arises in IPM due to the possible nontransitivity of the match relation: in the example considered above, $1 \approx 5$ and $5 \approx 2$ does not imply $1 \approx 2$.

To describe the consequences of nontransitivity, recall that a *border* of x is any proper prefix of x that equals a suffix of x. For a string $x[1..n]$, an array $\beta[1..n]$ is called the *border array* of x iff for $i = 1, 2, ..., n$, $\beta[i]$ gives the length of the longest border of

$x[1..i]$. The classic border array algorithm is given in [6], variants for indeterminate strings can be found in [13].

A great many of the exact pattern-matching algorithms (for example, Knuth-Morris-Pratt [20], Boyer-Moore [8], and their numerous variants) make use of the border array of the pattern or some version of it. The trouble is that for indeterminate strings, the nontransitivity of matching causes essential properties of the border array to fail [22], as we now demonstrate by example.

Index		1	2	3	4	5	6	7	
x	$\cdots$	a	a	b	b	a	b	b	$\cdots$
p		a	$*$	$*$	b	a	$*$	a	
$1st$ Shift				a	$*$	$*$	b	a	$\cdots$
$2nd$ Shift						a	$*$	$*$	$\cdots$
$3rd$ Shift							a	$*$	$\cdots$

Table 1. First example of the nontransitivity effect

Table 1 shows KMP pattern-matching of p against x. The first six positions of p match x, but there is a mismatch in position 7. According to the traditional definition of border, the longest border of $p[1..6]$ is $a{*}{*}b$, the second-longest border is $a*$ and the third is a. KMP performs shifts according to the borders of p in decreasing order of length, as shown by the shifts in the table. Observe however that if we perform a shift according to the longest border, aligning $p[1..4]$ with $x[3..6]$, we will then have letter a aligned with b in position 3. So indeterminate strings have the following property as opposed to traditional strings:

Proposition 1. *Shifting the indeterminate pattern p to the right in x according to the longest border does not guarantee a prefix match.*

Moreover, we see that between the first and second shifts lies a border $a{*}{*} = {*}{*}b$ of length 3 that is the longest border of substring $a{*}{*}b$. This reveals another property of indeterminate strings:

Proposition 2. *A border of a border of indeterminate string x is not necessarily a border of x.*

Index		1	2	3	4	5	6	7	
x	$\cdots$	a	b	a	$*$	a	$*$	a	$\cdots$
p		a	b	a	a	a	b	b	
Wrong Shift						a	b	a	$\cdots$
Correct Shift				a	b	a	a	a	$\cdots$

Table 2. Second example of the nontransitivity effect

In Table 2 we see that the length of the longest border of substring $p[1..6]$ is 2. But if we shift the pattern p to the right according to its longest border by $6 - 2 = 4$, we miss a prefix match in position 3, again due to nontransitivity. Thus:

Proposition 3. *Shifting the indeterminate pattern p to the right in x according to the longest border can miss occurrences of p.*

4 The New Hybrid Algorithm

The results of Section 3 warn us that a variant of any exact pattern-matching algorithm adapted for IPM is problematic if it depends on any form of border array calculation. In fact, one such variant has been proposed: Algorithm iFJS [15] describes an IPM adaptation of the FJS exact pattern-matching algorithm [11], that combines the border-independent Sunday version BMS [21] of the Boyer-Moore algorithm with the border-dependent KMP algorithm. This variant uses the border array only up to the longest prefix of p that does not contain any indeterminate letters. The problem is that if an indeterminate letter appears close to the left end of the pattern, then only a very small shift can occur each time, slowing the algorithm's speed significantly.

As a result, we propose replacing the KMP algorithm in iFJS by the ShiftAnd algorithm [10,7,23] that not only makes no use of the border array, but that furthermore has already been suggested [23] as a paradignm for IPM. We note that this strategy could be extended in a straightforward manner to use more sophisticated versions of ShiftAnd, such as the BNDM algorithm described in [18]. Our experiments suggest that the judicious combination of algorithms flipflopping from one to another based on the nature of local segments of text is more efficient than a single algorithm on its own.

Our algorithm adopts the following simple strategy:

(1) Perform a Sunday shift along the text.
(2) When a match is found at the end of the pattern, switch to ShiftAnd matching.
(3) Continue ShiftAnd matching until no match is found at the current position, then skip to the next possible position and switch back to Sunday shift.

Figure 1 shows the pseudocode for finding all the matches of pattern $p = p[1..m]$ in text $x = x[1..n]$:

```
i' ← m; m' ← m − 1;
while i' ≤ n do
  Sunday-Shift;
  — After Sunday-Shift exits, perform ShiftAnd-Match
  i ← i' − m';
  ShiftAnd-Match;
  — After ShiftAnd-Match exits, shift pattern right
  i' ← i + m';
```

Figure 1. Algorithm ShiftAnd-Sunday

For completeness we provide sketches of the Sunday and ShiftAnd algorithms:

The Sunday (BMS) Algorithm [21]

BMS has a $O(mn)$ worst-case running time but in practice is one of the fastest exact pattern-matching algorithms. To control shifts, it computes a Δ array in a preprocessing phase as follows:

For every $\lambda \in \Sigma$, $\Delta[\lambda] = m-l+1$, where l is the rightmost position in p where λ occurs; if λ does not occur in p, then $\Delta[\lambda] = m+1$.

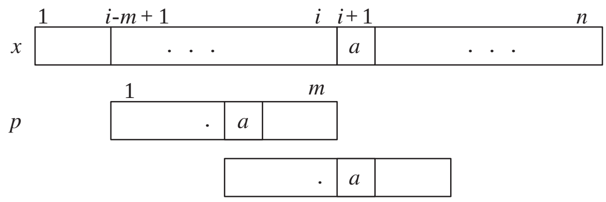


Figure 2. Sunday Shift of BMS

Figure 2 demonstrates the basic shift strategy of BMS: positions in pattern and text are compared until a mismatch occurs, say at position i in x, at which point the pattern is shifted to the next position at which an occurrence is possible, using Δ to align $x[i+1] = a$ with the rightmost occurrence of a in p. Since there can be no occurrence in between (otherwise Δ does not record the rightmost occurrence of a, a contradiction), we are safe to do so.

The ShiftAnd Algorithm [10,7,23]

ShiftAnd makes use of the bit-parallel capability inherent in a computer word. It has time complexity $O(mn/w)$, where w is the computer word length in bits, and is widely used in pattern-matching programs such as Unix agrep [1]. In a preprocessing phase, for each $\lambda \in \Sigma$ and every $i \in 1..m$, the algorithm computes a bit-array $S = S[1..m, 1..\alpha]$ such that $S[i, \lambda] = 1$ iff $p[i] = \lambda$, otherwise 0. This table controls the state of the calculation at each of w preceding positions in x as the pattern is shifted from position i to $i+1$. For example, for a DNA alphabet $\Sigma = \{A, C, G, T\}$ and a pattern $p = AATCG$, ShiftAnd preprocesses S as shown in Table 3.

$m \backslash \Sigma$	A	C	G	T
A	1	0	0	0
A	1	0	0	0
T	0	0	0	1
C	0	1	0	0
G	0	0	1	0

Table 3. Bit-array S after Preprocessing

The New Algorithm: ShiftAnd-Sunday

Pseudocode for the Sunday and ShiftAnd preprocessing is shown in Figures 3–4.

```
for i = 1 to |Δ|
   Δ[i] = m + 1
for i = 1 to m
      for j = 1 to |Σ|
         if MATCH(p[i], Σ[j]) then Δ[p[i]] = i
```

Figure 3. `Sunday-Preprocessing`

It is formally identical to the pseudocode used for exact pattern matching when indeterminate letters are not involved — the difference resides in the implementation

```
for i = 1 to m
  for j = 1 to |Σ|
    if MATCH(p[i], Σ[j]) then S[i, j] = 1
    else S[i, j] = 0
```

Figure 4. `ShiftAnd-Preprocessing`

of the `MATCH` function that determines whether or not two letters of the possibly extended alphabet Σ' match. The various implementations of MATCH corresponding to each of the six indeterminate processing models (1) are discussed in detail in [15].

The procedures `Sunday-Shift` and `ShiftAnd-Match` are also formally identical to their exact matching equivalents, again depending only on an implementation of `MATCH`. They are shown in Figures 5–6. Note that in practice the ShiftAnd algorithm needs to be implemented in a more sophisticated way in order to allow pattern length longer than the system word size. An example of pattern matching using this new algorithm is shown in Figures 11–17 in the Appendix.

```
while not MATCH(p[m], x[i']) do
  i' ← i'+Δ[x[i'+1]]
  if i' > n then return
```

Figure 5. `Sunday-Shift`

```
D ← 0
repeat
   — Here and throughout this paper operator ≪ means shifting D one position
   — towards the most significant bit and bring a 1 to the least significant bit
  D ← (D ≪ 1)&S_x[i]
  if D&10^m ≠ 0 then output i
  i ← i + 1
 — If D = 0, exit: no position in p has a current match.
until D = 0 or i > n
```

Figure 6. `ShiftAnd-Match`

Since the subroutine `Sunday-Shift` increases the variable i' monotonically and subroutine `ShiftAnd-Match` increases the variable i monotonically, these two subroutines can be executed at most n times altogether. Each loop in `Sunday-Shift` runs in constant time and each loop in `ShiftAnd-Match` runs in $O(m/w)$ time. Therefore the worst case running time is $O(\frac{mn}{w})$, where w is the system word size. This asymptotic time complexity is the same as ShiftAnd and better than BMS. Moreover, the new algorithm adapts well to the input, as shown in the test results.

5 Experiments

5.1 Experimental Details

Since the new algorithm is a hybrid of Sunday and ShiftAnd, we compare its running time with its components.

Factors that affect the performance of string pattern-matching are text length, pattern occurrence frequency, pattern length, alphabet size and frequency of indeterminate letters. We try to show the behaviour of the algorithms by changing only one factor at a time. However, there could be interactions between them. For example, changing the alphabet size might cause the pattern occurrence frequency to change. We have tried to design our test cases to be both meaningful and realistic.

The main platform for our tests is a SUN X4600 server with four 2.6 GHz dual core Opteron CPUs (total 8), 16 GB RAM, four SAS disks, running GNU Linux 2.6.18-53.1.4.e15. We also ran tests, with consistent results, on other platforms such as a PC running Windows XP SP2.

To time the CPU time consumed by different algorithms, we use the standard C library function clock() [2]. Since the running time can be affected by factors such as CPU and memory usage of the system, temperature etc, each test was repeated 20 times. From our past experience we take the minimum time as the most accurate result. All preprocessing time is included. Functions are declared inline to eliminate the effect of function call overhead. The results are very stable across different runs.

The main test file corpus was taken from [3], itself collected from sources such as [12] for English text, [4] for DNA and protein files.

5.2 Experimental Results

Since all three algorithms are capable of handling both regular and indeterminate strings, we first test their performance on regular pattern-matching without specifying any indeterminate letters.

Execution Time against Text Length in English Files Here we run the algorithms on ten English files from [12] of sizes ranging from 240KB to 5158KB (Table 4). We use a pattern set from [16] consisting of several words that occur with moderate frequency in regular English text:

better enough govern public someth system though

File Name	Length(bytes)	Description
English0.txt	237599	HAMLET, PRINCE OF DENMARK
English1.txt	389204	The Mysterious Affair at Styles
English2.txt	491905	Secret Adversary
English3.txt	699594	Pride and Prejudice (partial)
English4.txt	754019	Pride and Prejudice
English5.txt	1186876	The Adventures of Harry Richmond(partial)
English6.txt	2672650	The Adventures of Harry Richmond(partial)
English7.txt	3251887	War and Peace(partial)
English8.txt	4387156	War and Peace
English9.txt	5872902	The Adventures of Harry Richmond

Table 4. English text files

From Figure 7 we see that the new algorithm has performance close to BMS. This is because it adapts to the nature of the text and chooses to use the BMS engine most of the time. Table 5 gives the average speed of the three algorithms in microseconds per million letters (Minimum execution time divided by the length of text then take the average result of 10 files, the same for all following tables).

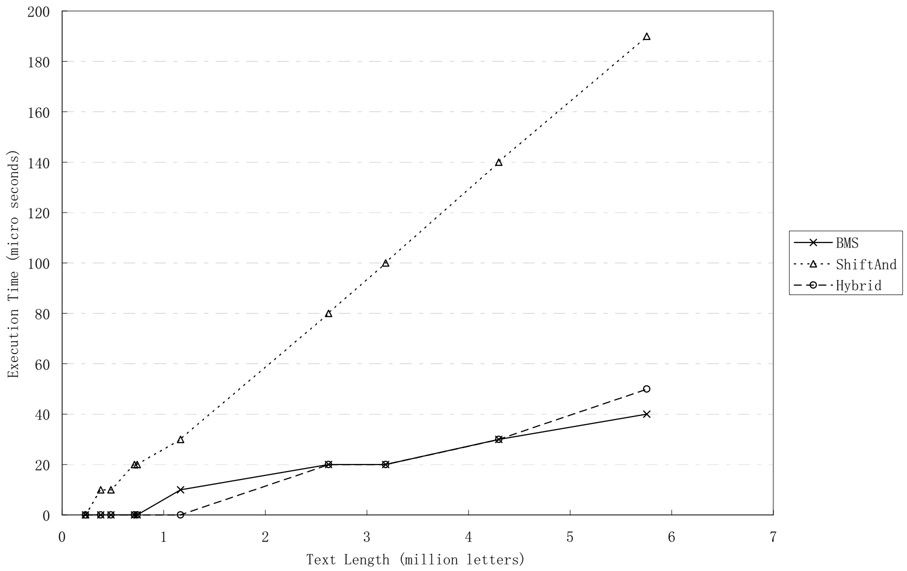


Figure 7. Execution time against text length in English files

BMS	ShiftAnd	Hybrid
990	4550	1060

Table 5. Average microseconds/million letters in Figure 7

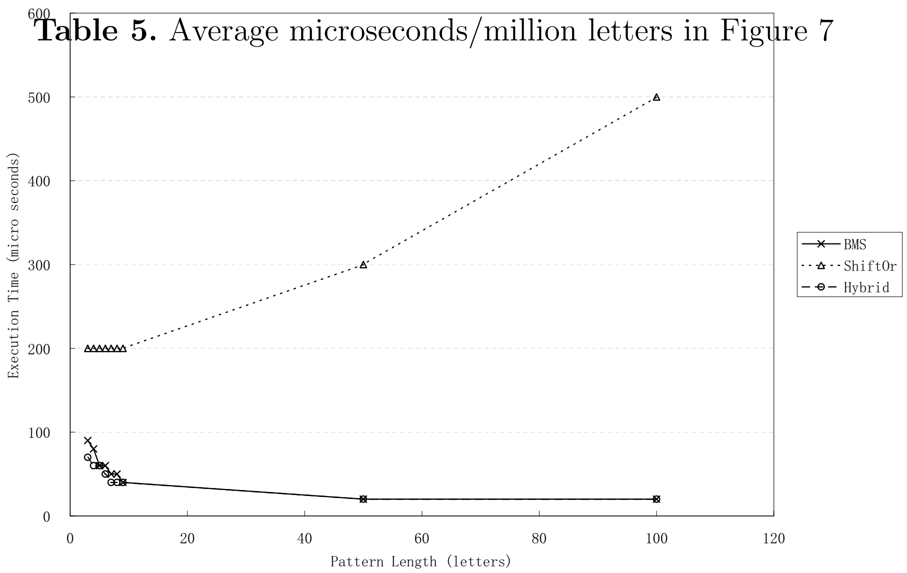


Figure 8. Execution time against pattern length in English files

Execution Time against Pattern Length in English Files Next we test the performance of the algorithms on varying pattern lengths. We use the file English8.txt, gradually increasing pattern length from 3 to 100 (see Table 6). Since longer patterns will as a rule occur less frequently, we insert the patterns randomly into the text with a frequency that decreases as pattern length increases.

From Figure 8 we see that the running times of both BMS and Hybrid decrease as pattern length increases. This is expected since the longer the pattern, the longer the skip that can be achieved by both BMS and Hybrid. As indicated by the increasing slope of the line from pattern lengths 9 to 50, when the pattern length passes the

File Name	Pattern length	Example	Total occurrences
p3.txt	3	air, age, ago	5563
p4.txt	4	body, half, held	4160
p5.txt	5	death,field, money	2665
p6.txt	6	became, behind, cannot	2426
p7.txt	7	already,brought, college	1038
p8.txt	8	anything, evidence	1685
p9.txt	9	available, community	612
p50.txt	50	Welcome To The World of ...	286
p100.txt	100	"If you have nothing better to do, ..."	275

Table 6. Details of the pattern sets used

BMS	ShiftAnd	Hybrid
1600	14390	1430

Table 7. Average microseconds/million letters in Figure 8

system word size (32), the running time of ShiftAnd begins to increase. By mainly using its BMS engine, Hybrid avoids this kind of performance slowdown.

Table 7 gives the average speed of the three algorithms over all the pattern sizes. Note that in this case the hybrid algorithm is slightly faster overall.

Execution Time against Number of Indeterminate Letters in the Alphabet Next we test the ability of our algorithm to handle indeterminate strings. In this test we again use English8.txt and the same pattern set as in our first test, but gradually increase the number of indeterminate letters in the alphabet, thus increasing their number in both text and pattern. We use the `MATCH` function corresponding to the M3q version of the hybrid algorithm, the most general (and therefore slowest) of the three quantum versions identified in Section 2. Run times are shown in Figure 9. We can see that BMS_3q runs fastest when indeterminate letters are few, but is overtaken by both ShiftAnd and the new algorithm as the number of indeterminate letters grows. Table 8 gives the average speed of the three algorithms.

BMS	ShiftAnd	Hybrid
5220	4651	4841

Table 8. Average microseconds/million letters in Figure 9

Execution Time against Text Length in DNA Files with Indeterminate Letters Finally we test the execution time against text length in DNA files with a 4-letter alphabet. We use FASTA files of increasing length as described in Table 9, with the following patterns:

CTGTAA, CAGACC, TATCCA, GGAGCC, TCCAGG, GCGGAT, AGAGAC

Letters A and C are defined as indeterminate letters. From Figure 10 we see that the three algorithms have very similar performance.

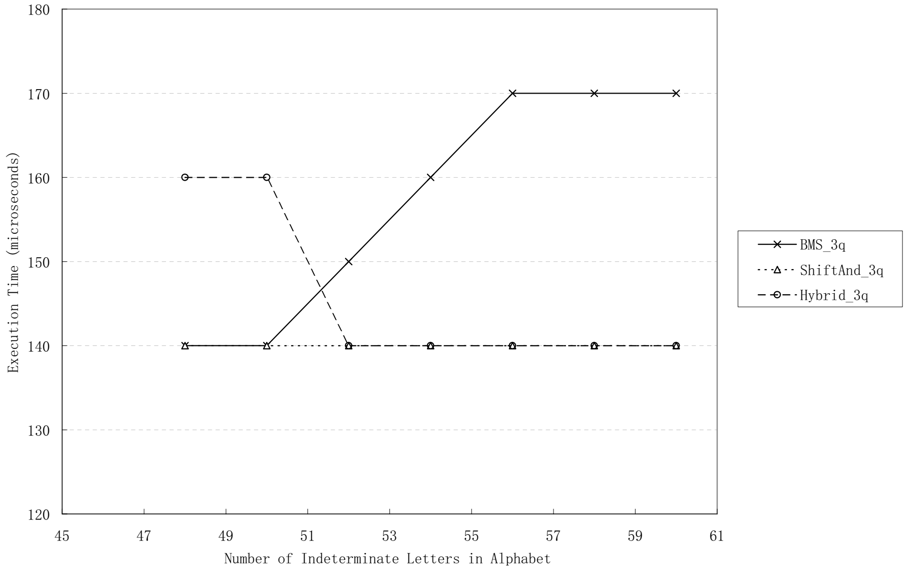


Figure 9. Execution time against number of indeterminate letters in the alphabet

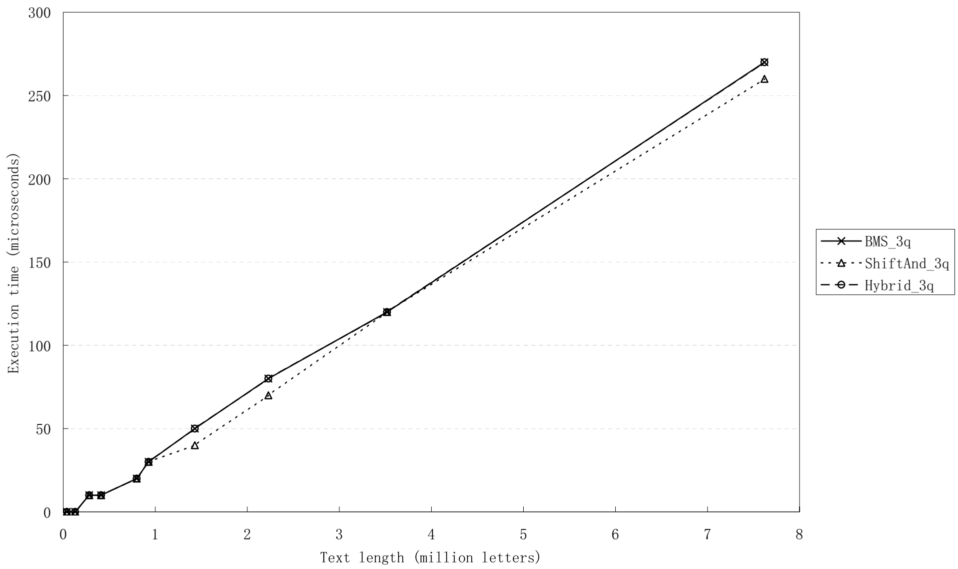


Figure 10. Execution time against text length in DNA files with indeterminate letters

File Name	Length(bytes)
DNA0.fasta	40302
DNA1.fasta	129145
DNA2.fasta	282348
DNA3.fasta	411493
DNA4.fasta	798564
DNA5.fasta	927709
DNA6.fasta	1430159
DNA7.fasta	2228723
DNA8.fasta	3518496
DNA9.fasta	7618319

Table 9. Lengths of DNA text files

BMS	ShiftAnd	Hybrid
4531	4297	4531

Table 10. Average microseconds/million letters in Figure 10

Tests\ Algorithms	BMS	ShiftAnd	Hybrid
Text Length	990	4550	1060
Pattern Length	1600	14390	1430
Number of Indeterminate Letters	5220	4651	4841
DNA file	4531	4297	4531
TOTAL	12341	27888	11862

Table 11. Summary of all test results in microseconds/million letters

Table 10 gives the average speed of the three algorithms.

We see from Table 11 that in all of these tests, the hybrid algorithm's behaviour is very close to that of the better of BMS and ShiftAnd. Moreover, due to its adaptiveness, its overall running time is actually the least over all of these rather diverse test cases. This dynamic adaptivity is useful when we do not know in advance the nature of the text or pattern: we don't need to make a decision ahead of time which algorithm to use.

6 Conclusion

We designed a new algorithm that performs fast pattern-matching on both regular and indeterminate strings. We showed in the experiments that although this new algorithm is not always the fastest, it has a strong ability to adapt to the nature of text/pattern and to achieve very good performance in all cases. In future we would like to see more competitive IPM algorithms, perhaps adapted from other exact pattern-matching algorithms such as BNDM or the convolution method.

References

1. AGREP V3.37, Homepage V1.12, T. Gries, `http://www.tgries.de/agrep/`:
2. GNU C Library, `http://www.gnu.org/software/libc/manual`:
3. Simon's Collection of Test Strings, `http://www.cas.mcmaster.ca/~bill/strings`:
4. National Center for Biotechnology Information, `http://www.ncbi.nlm.nih.gov/`:
5. K. Abrahamson: *Generalized string matching.* SIAM Journal on Computing, 16(6) 1987, pp. 1039–1051.
6. A. V. Aho, J. E. Hopcroft, and J. D. Ullman: *The Design & Analysis of computer Algorithms*, Addison-Wesley, 1974.
7. R. Baeza-Yates and G. Gonnet: *A new approach to text searching.* Communications of the ACM, 35(10) 1992, pp. 74–82.
8. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Communications of the ACM, 20(10) 1977, pp. 762–772.
9. C. Charras and T. Lecroq: *Handbook of Exact String Matching Algorithms*, King's College Publications, 2004.
10. B. Dömölki: *A universal computer system based on production rules.* BIT, 8 1968, pp. 262–275.
11. F. Franek, C. G. Jennings, and W. F. Smyth: *A simple fast hybrid pattern-matching algorithm (preliminary version)*, in Proc. 16th Annual Symposium on Combinatorial Pattern Matching, LNCS 3537, Springer-Verlag, 2005, pp. 288–297.

12. M. Hart: *Project gutenberg, project gutenberg literary archive foundation (2004):*.
13. J. Holub and W. F. Smyth: *Algorithms on indeterminate strings*, in Proc. 14th Australasian Workshop on Combinatorial Algorithms, 2003, pp. 36–45.
14. J. Holub, W. F. Smyth, and S. Wang: *Fast pattern-matching on indeterminate strings.* Journal of Discrete Algorithms, 6 2006, pp. 37–50.
15. J. Holub, W. F. Smyth, and S. Wang: *Hybrid pattern-matching algorithms on indeterminate strings.* London Algorithmics and Stringology 2006, J. Daykin, M. Mohamed and K. Steinhoefel (eds.), King's College London Series Texts in Algorithmics, 2006, pp. 115–133.
16. C. G. Jennings: *A linear-time algorithm for fast exact pattern matching in strings*, Master's thesis, McMaster University, 2002.
17. D. E. Knuth, J. H. Morris, and V. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
18. G. Navarro and M. Raffinot: *A bit-parallel approach to suffix automata: Fast extended string matching*, in Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching, M. Farach-Colton, ed., no. 1448, Piscataway, NJ, 1998, Springer-Verlag, Berlin, pp. 14–33.
19. G. Navarro and M. Raffinot: *Flexible Pattern Matching In Strings : Practical on-line search algorithms for texts and biological sequences*, Cambridge University Press, 2002.
20. B. Smyth: *Computing Patterns in Strings*, Addison Wesley, 2003.
21. D. M. Sunday: *A very fast substring search algorithm.* Communications of the ACM, 8 1990, pp. 132–142.
22. S. Wang: *Pattern-matching algorithms on indeterminate strings*, Master's thesis, McMaster University, Hamilton, Canada, 2006.
23. S. Wu and U. Manber: *Fast text searching with errors.* Communications of the ACM, 35(10) 1992, pp. 83–91.

A An Example of ShiftAnd-Sunday Algorithm

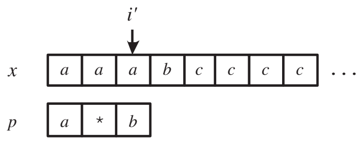


Figure 11. Starting position

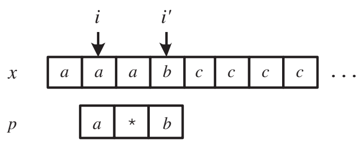


Figure 12. After one step in Sunday-Shift

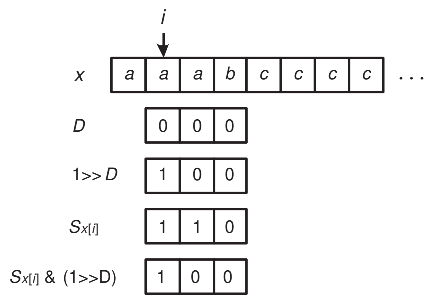


Figure 13. Switch to ShiftAnd-Matching

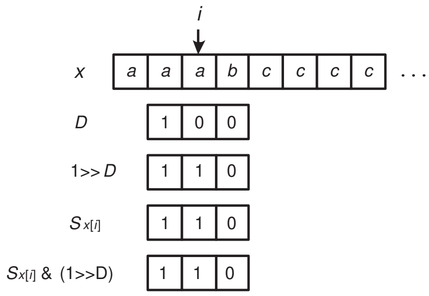


Figure 14. ShiftAnd-Matching Continues

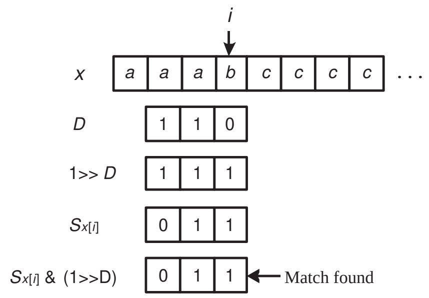


Figure 15. A match is found

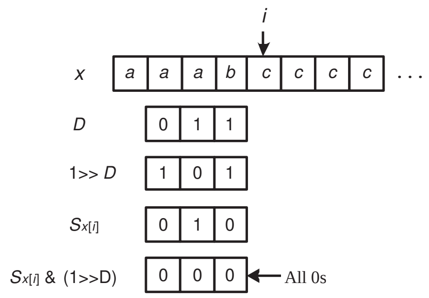


Figure 16. D' contains all zeros

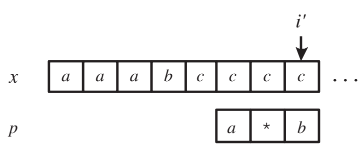


Figure 17. Switch back to Sunday-Shift

Conservative String Covering of Indeterminate Strings

Pavlos Antoniou[1], Maxime Crochemore[1], Costas S. Iliopoulos[1], Inuka Jayasekera[1], and Gad M. Landau[2]

[1] Dept. of Computer Science, King's College London, London WC2R 2LS, England, UK
[2] Department of Computer Science, University of Haifa, Mount Carmel, Haifa 31905, Israel

Abstract. We study the problem of finding local and global covers as well as seeds in conservative indeterminate strings. An indeterminate string is a sequence $T = T[1]T[2]\dots T[n]$, where $T[i] \subseteq \Sigma$ for each i, and Σ is a given alphabet of fixed size. A conservative indeterminate string, is an indeterminate string where the number of indeterminate symbols in the positions of the string, i.e the non-solid symbols, is bounded by a constant κ. We present an algorithm for finding a conservative indeterminate pattern p in an indeterminate string t. Furthermore, we present algorithms for computing conservative covers and seeds of the string t.

1 Introduction

Covers are considered as common regularities in a string along with repetitions and periods. They are periodically repetitive. A substring w of a string x is called a cover of x if and only if x can be constructed by concatenations and superpositions of w. A seed is an extended cover in the sense of a cover of a superstring of x.

Finding the regularities present in strings is not only interesting in string algorithms but it is also useful in many applications. These applications include molecular biology, data compression and computational music analysis. Regularities in strings have been studied widely the last 20 years. There are several $O(n \log n)$- time algorithms for finding repetitions ([4],[7]), in a string x, where n is the length of x. Apostolico and Breslauer [2] gave an optimal $O(\log \log n)$-time parallel algorithm for finding all the repetitions. The preprocessing of the Knuth-Morris-Pratt algorithm [11] finds all periods of every prefix of x in linear time.

In many cases, it is desirable to relax the meaning of repetition. For instance, if we allow overlapping and concatenations of periods in a string we get the notion of *covers*. The notion of covers was introduced by Apostolico, Farach and Iliopoulos in [3], where a linear-time algorithm to test superprimitivity, was given. Moore and Smyth in [12] gave linear-time algorithms for finding all covers of a string x.

An extension of the notion of covers, is that of seeds; that is, covers of a superstring of x. The notion of seeds was introduced by Iliopoulos, Moore and Park [10] and an $O(nlogn)$-time algorithm was given for computing all seeds of x. A parallel algorithm for finding all seeds was presented by Berkman, liopoulos and Park [6], that requires $O(\log n)$ time and $O(n \log n)$ work.

In this work, we study and design algorithms for these string regularities in indeterminate strings. An indeterminate string is a sequence $T = T[1]T[2]\dots T[n]$, where $T[i] \subseteq \Sigma$ for each i, and Σ is a given alphabet of potentially large size. The simplest form of indeterminate string is one in which indeterminate positions can contain only

Pavlos Antoniou, Maxime Crochemore, Costas S. Iliopoulos, Inuka Jayasekera, Gad M. Landau: Conservative String Covering of Indeterminate Strings, pp. 108–115.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

a don't care letter, that is, a letter $*$ that matches any letter of the alphabet Σ on which X is defined.

In biology, usually, the number of indeterminate positions in a sequence is naturally bounded by a constant value. Otherwise, we would have a cover of length 1 with just a don't care symbol that corresponds to all the letters of the alphabet Σ. Therefore, we impose a constraint on the strings, which requires that the number of indeterminate positions in a cover c is less than the constant, that is a "conservative" cover. An example of a sequence containing indeterminate positions is shown in Figure 1 which depicts a sequence logo of an indeterminate sequence.The bottom logo is the consensus sequence derived by the 12 sequences on top of it. If we look at the logo we can see that position 1 is indeterminate as we can have $[TCAG]$ occurring, position 2 is indeterminate also having possible occurrence of $[TCA]$,position 3 is solid, non indeterminate, as in that position only A occurs.

An algorithm was described [8] for computing all occurrences of a pattern p in a text string x, but although efficient in theory, the algorithm was not useful in practice. Indeterminate string pattern matching has mainly been handled by bit mapping techniques (ShiftOr method) [5],[15]. These techniques have been used to find matches for an indeterminate pattern p in a string x [9] and the agrep utility [14] has been one of the few practical algorithms available for indeterminate pattern-matching.

In [9], the authors extended the notion of indeterminate strings by distinguishing two distinct forms of indeterminate match: "quantum" and "deterministic". Roughly speaking, a "quantum" match allows an indeterminate letter to match two or more distinct letters during a single matching process; a "determinate" match restricts each indeterminate letter to a single match[9].

In this paper, we describe algorithms for finding string regularities in constrained indeterminate strings. The next section introduces the basic definition, Section 3 describes the algorithm for conservative pattern matching. Additionally, Section 4 and Section 5 describe the algorithms for computing the covers and seeds of a string respectively.

2 Basic definitions

A *string* is a sequence of zero or more symbols from an alphabet Σ. The set of all strings over Σ is denoted by Σ^* . The length of a string x is denoted by $|x|$. The *empty* string, the string of length zero, is denoted by ϵ. The i-th symbol of a string x is denoted by $x[i]$.

A string w is a substring of x if $x = uwv$, where $u, v \,\epsilon\, \Sigma^*$. We denote by $x[i \dots j]$ the substring of x that starts at position i and ends at position j. Conversely, x is called a *superstring* of w. A string w is a *prefix* of x if $x = wy$, for $y \,\epsilon\, \Sigma^*$. Similarly, w is a *suffix* of x if $x = yw$, for $y \,\epsilon\, \Sigma^*$.

We call a string w a *subsequence* of x (or x is a supersequence of w) if w is obtained by deleting zero or more symbols at any positions from x. For example, ace is a subsequence of $aabcdef$. For a given set S of strings, a string w is called a common supersequence of S if s is a supersequence of every string in S.

The string xy is the *concatenation* of the strings x and y. The concatenation of k copies of x is denoted by x^k. For two strings $x = x[1 \dots n]$ and $y = y[1 \dots m]$ such that $x[n-i+1 \dots n] = y[1 \dots i]$ for some $i \geq 1$ (that is, such that x has a suffix equal to a prefix of y), the string $x[1 \dots n]y[i+1 \dots m]$ is said to be a *superposition* of x

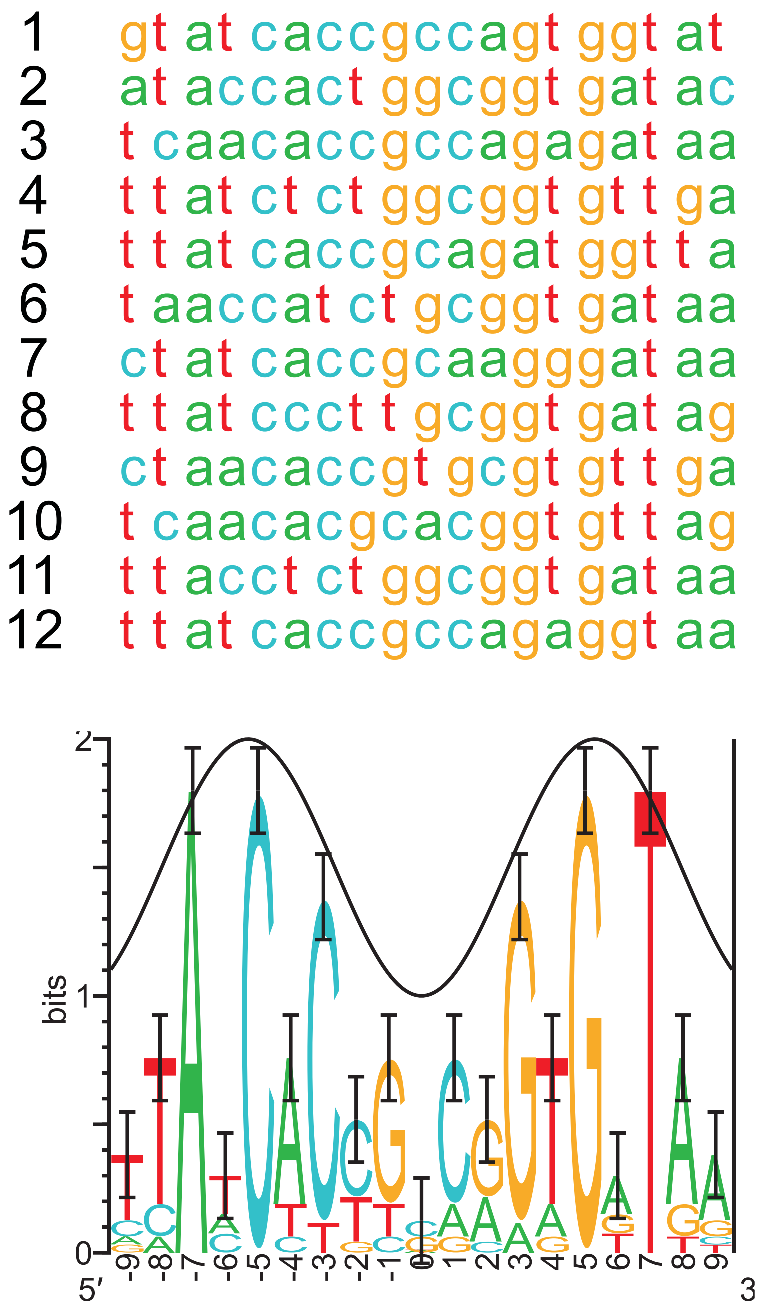


Figure 1. A sequence logo of a biological indeterminate sequence. Picture taken from [13]

and y. We also say that x overlaps with y. A substring y of x is called a *repetition* in x, if $x = uy^kv$, where u, y, v are substrings of x and $k \geq 2$, $|y| \neq 0$. For example, if $x = aababab$, then a (appearing in positions 1 and 2) and ab (appearing in positions 2, 4 and 6) are repetitions in x; in particular $a^2 = aa$ is called a *square* and $(ab)^3 = ababab$ is called a *cube*.

A non-empty substring w is called a *period* of a string x, if x can be written as $x = w^k w^r$ where $k \leq 1$ and w' is a prefix of w. The shortest period of x is called the period of x. For example, if $x = abcabcab$, then abc, $abcabc$ and the string x itself are periods of x, while abc is the period of x.

A substring w of x is called a *cover* of x, if x can be constructed by concatenating or overlapping copies of w. We also say that w *covers* x. For example, if $x = ababaaba$, then aba and x are covers of x. If x has a cover $w \neq x$, x is said to be *quasiperiodic*; otherwise, x is *superprimitive*.

A substring w of x is called a seed of x, if w covers one superstring of x (this can be any superstring of x, including x itself). For example, aba and $ababa$ are some seeds of $x = ababaab$.

An *indeterminate string* is a sequence $T = T[1]T[2]\dots T[n]$, where $T[i] \subseteq \Sigma$ for each i, and Σ is a given alphabet of potentially large size. When a position of the string is indeterminate, and it can match more than one element from the alphabet Σ, we say that this position has *non-solid* symbol. If in a position we have only one element of the alphabet Σ present, then we refer to this symbol as *solid*. A *conservative* indeterminate string, is an indeterminate string where its number of indeterminate symbols is bounded by a constant k.

Building the Aho-Corasick Automaton [1]. The Aho-Corasick Automaton for a given finite set P of patterns is a Deterministic Finite Automaton G accepting the sets of all words containing a word of P as a suffix.

$G = (Q, \Sigma, g, f, q_0, F)$, where function Q is the set of states, Σ is the alphabet, g is the forward transition, f is the failure link i.e. $f(q_i) = q_j$, if and only if S_j is the longest suffix of S_i that is also a prefix of any pattern, q_0 is the initial state and F is the set of final (terminal) states [1]. The construction of the AC automaton can be done in $O(d)$-time and space complexity, where d is the size of the dictionary, i.e. the sum of the lengths of the patterns which the AC automata will match.

3 Finding constrained pattern p in indeterminate string T

As a building step, here, we study the constrained pattern matching problem on indeterminate strings. The problem of constrained indeterminate pattern matching is defined as follows:

INPUT: We are given a pattern p of length m with at most κ non solid symbols, where κ is a constant. We are given an indeterminate string T, the text, of length n.

QUERY: Find all the occurrences of the pattern p in the text T, i.e find the positions in T where the intersection of the pattern and the text is non-empty.

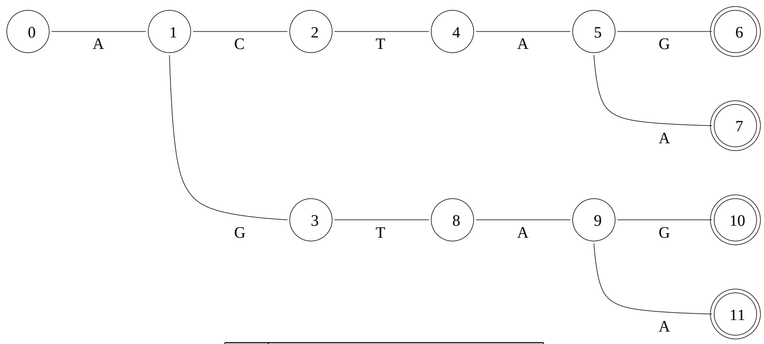
Example 1. We consider a pattern, $p = A[CG]TA[AG]$ and text, $T = GA[CG][CT]A$ $G[AT]A[AG][CT][AT]AG$. Figure 2 shows the result of searching for p in t. It can be seen from the figure that p occurs in t starting at positions 2, 5, 8 and 9.

i	0	1	2	3	4	5	6	7	8	9	10	11	12
t	G	A	[CG]	[CT]	A	G	[AT]	A	[AG]	[CT]	[AT]	A	G
Matches		A	[CG]	T	A	[AG]							
					A	[CG]	T	A	[AG]				
								A	[CG]	T	A	[AG]	
									A	[CG]	T	A	[AG]

Figure 2. Pattern matching with p and t

The algorithm works in two steps:

Step 1
Let the pattern p be $p = P_1P_2 \ldots P_m$. We built the Aho-Corasick automaton for the dictionary of the prefixes of the pattern $D = \{\pi_1\pi_2 \ldots \pi_m, \forall \pi_i \in P_i, 1 \le i \le m\}$. Note that $|D| = \prod_{i=1}^{m} |P_i| < 2^\kappa$ as there are at most κ non-solid symbols.



i	0	1	2	3	4	5	6	7	8	9	10	11
f(i)	0	0	0	0	0	1	3	1	0	1	3	1

Figure 3. Aho-Corasick automata and its failure function for p

Step 2
Assume that we have processed $T[1, i]$. At this point we have a set, P, of prefixes of the strings in the dictionary in the Aho-Corasick automaton. We will now perform iteration $i + 1$. For each symbol τ occurring at $T[i + 1]$, we try to extend each prefix in P by that symbol τ, or we follow its failure link provided by the Aho-Corasick automaton. Figures 3 and 4 present a part of the matching process for the previous example.

Note that $|P|$ is bounded by the maximum number of possible prefixes, which in turn is bounded by the size of the automaton, therefore this is constant. Thus, this method is linear.

i	0	1	2	3	4	5	6	
t	G	A	[CG]	[CT]	A	G	[AT]	...
P	0	{1}	{2,3}	{4,8}	{5,9}	{6, 10}	{8}	...

Figure 4. Matches of prefixes of P in text t

4 Computing λ-conservative covers of indeterminate strings

Here, we study another string regularity, conservative covering of an indeterminate string with a fixed length cover. The λ-conservative cover problem is defined as follows:

Input: We are given a conservative indeterminate string t, of length n, a constant κ, which is the maximum number of non-solid symbols allowed in a cover and an integer λ, which is the length of the cover.
Query: Is there a conservative cover, c, of t of length λ?

Step 1
We consider the prefix, $\hat{T}$, of t of length λ,

$$\hat{T} = T_1 \dots T_\lambda$$

and the suffix, $\tilde{T}$ of t of length λ,

$$\tilde{T} = T_{n-\lambda+1}, \dots T_n$$

We build the Aho-Corasick automaton for the dictionary

$$D = \{t_1 \dots t_\lambda \mid \forall\ t_i \in T_i \cap T_{i+n-\lambda},\ 1 \leq i \leq \lambda\}$$

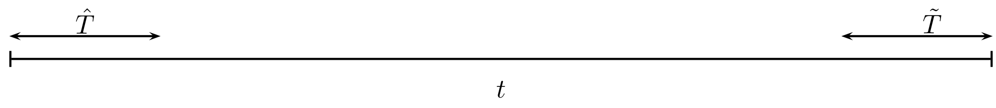


Figure 5. The cover, c, covers the beginning and the end of T. Thus $\hat{T}$ and $\tilde{T}$ provide the set of potential candidates.

Step 2
For each $d \in D$ we find all of its occurrences in T, parsing the text T through the Aho-Corasick Automaton built in Step 1. If a word d occurs at position i then we set a flag $L(i) = true$. If the distance $|i - j|$ of any two consecutive flags is less than λ, then we have a cover

$$C_1 C_2 \dots C_\lambda, \text{where}$$

$$C_i = \{d_i,\ \text{is the } i - th \text{ letter of every word in } D,\ 1 \leq i \leq \lambda\}$$

The overall complexity of the above two steps is linear.

5 Computing λ-conservative seeds of indeterminate strings

Here we study yet another regularity, covering an indeterminate string with seed of a given length. The λ-constrained seed problem is defined as follows:

Input: We are given an indeterminate string t, of length n, a constant κ, which is the maximum number of non-solid symbols allowed in a seed and an integer λ, which is the length of the seed.
Query: Is there a conservative seed, s, of t of length λ?
Step 1
The first occurrence of the seed can be in any of the positions $\{1 \dots \lambda\}$. Thus we consider the following strings of length λ :

$$L_1 = \{T[1..\lambda], T[2..\lambda+1], \dots T[\lambda..2\lambda - 1]\}$$

and all the suffixes of string t of length λ :

$$L_2 = \{T[n - \lambda..n], T[n - \lambda - 1..n - 1] \dots T[n - 2\lambda - 1]\}$$

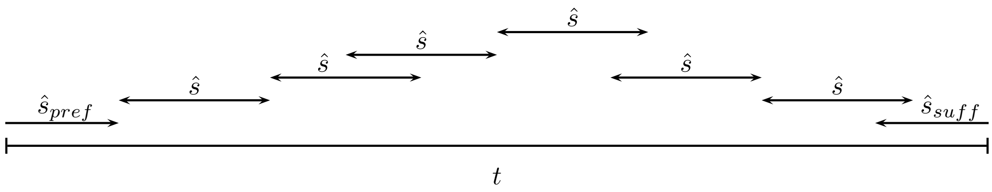


Figure 6. Above, $\hat{s}$ is a seed of the string t, where each $\hat{s}$ contains at most κ non-solid symbols and is of length λ. Also, $\hat{s}_{pref}$ and $\hat{s}_{suff}$ are a prefix and suffix of $\hat{s}$ respectively.

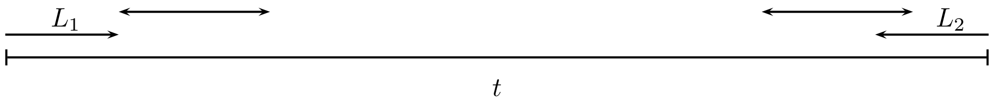


Figure 7. The positions of candidate seeds from lists L_1 and L_2 are shown above.

We build the Aho-Corasick automaton for the dictionary

$$D = \{t_{i_1} \dots t_{i_\lambda} \mid \forall t_{i_j}, \mathrm{where\ t_{i_j}\ is\ the\ j-th\ symbol\ of\ T \in L_1 \cup L_2}\}.$$

Step 2
For each $d \in D$ we find all of its occurrences in T, parsing the text T through the Aho-Corasick Automaton built in Step 1. If a word d occurs at position i then we set a flag $L_d(i) = true$. If the distance $|i-j|$ of any two consecutive flags in L_d is less than λ, then we d is a candidate for a seed. Let i_1 and i_2 be the first and last occurrences of d in T. We check if $T[1,i_1]$ is a suffix of d and if $T[i_2,n]$ is a prefix of d, if that is the case then d is a suffix. The overall complexity is $O(\lambda n)$.

6 Conclusion

In conclusion, we have shown $O(n)$ algorithms for finding the smallest conservative cover, λ-conservative local covers. We have also presented a $O(\lambda n)$ algorithm for finding the λ-conservative seeds of a string. All the algorithms which we have used are easily adaptable to allow the bit-matching technique to be used, in order to allow efficient implementations.

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search.* Commun. ACM, 18(6) 1975, pp. 333–340.
2. A. Apostolico and D. Breslauer: *An optimal o(loglog n)-time parallel algorithm for detecting all squares in a string.* SIAM J. Comput., 25(6) 1996, pp. 1318–1331.
3. A. Apostolico, M. Farach, and C. S. Iliopoulos: *Optimal superprimitivity testing for strings.* Information Processing Letters, 39 1991, pp. 17–20.
4. A. Apostolico and F. P. Preparata: *Optimal off-line detection of repetitions in a string.* Theor. Comput. Sci., 22 1983, pp. 297–315.
5. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Commun. ACM, 35(10) 1992, pp. 74–82.

6. O. Berkman, C. S. Iliopoulos, and K. Park: *The subtree max gap problem with application to parallel string covering*. Information and Computation, 123(1) 1995, pp. 127–137.
7. M. Crochemore: *An optimal algorithm for computing the repetitions in a word*. Inf. Process. Lett., 12(5) 1981, pp. 244–250.
8. M. J. Fischer and M. S. Paterson: *String-matching and other products*, tech. rep., Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.
9. J. Holub, W. F. Smyth, and S. Wang: *Fast pattern-matching on indeterminate strings*. J. of Discrete Algorithms, 6(1) 2008, pp. 37–50.
10. C. S. Iliopoulos, D. W. G. Moore, and K. Park: *Covering a string*, in Proceedings of the 4-th Symposium on Combinatorial Pattern Matching, vol. 684 of Lecture Notes in Computer Science, Berlin, 1993, Springer-Verlag, pp. 54–62.
11. D. E. Knuth, J. Morris, and V. R. Pratt: *Fast pattern matching in strings*. SIAM Journal of Computing, 6(2) 1977, pp. 323–350.
12. D. Moore and W. F. Smyth: *An optimal algorithm to compute all the covers of a string*. Inf. Process. Lett., 50(5) 1994, pp. 239–246.
13. M. C. Shaner, I. M. Blair, and T. D. Schneider: *Sequence logos: A powerful, yet simple, tool*, in Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences, Volume 1: Architecture and Biotechnology Computing, T. N. Mudge, V. Milutinovic, and L. Hunter, eds., IEEE Computer Society Press, 1993, pp. 813–821.
14. S. Wu and U. Manber: *Agrep – a fast approximate pattern-matching tool*, in Proceedings USENIX Winter 1992 Technical Conference, San Francisco, CA, 1992, pp. 153–162.
15. S. Wu and U. Manber: *Fast text searching: allowing errors*. Commun. ACM, 35(10) 1992, pp. 83–91.

On the Uniform Distribution of Strings

Sébastien Rebecchi* and Jean-Michel Jolion

Universit de Lyon, F-69361 Lyon
INSA Lyon, F-69621 Villeurbanne
CNRS, LIRIS, UMR 5205
{sebastien.rebecchi, jean-michel.jolion}@liris.cnrs.fr

Abstract. In this paper, we propose the definition of a measure for sets of strings of length not greater than a given number. This measure leads to an instanciation of the uniform distribution definition in sets of such limited-size strings, for which we provide a linear time complexity generative algorithm.
Some ideas could rather easily be extended to other ordered structure types.

Keywords: string, uniform distribution, measure

1 Introduction

For many years, several research teams have tried to blend the two main approaches of pattern recognition, namely the statistical and structural ones [3,2]. This choice is justified by the desire for being able at the same time to benefit from the undeniable advantages of the two approaches, while being detached of their respective drawbacks.

The statistical pattern recognition is based on a coding of the data in the form of numerical vectors, often unable to accurately reproduce the complexity of the data. However this choice is justified by the broad pallet of statistical algorithms published in the literature and recognized as powerful for the classification of numerical data[1].

In the structural pattern recognition paradigm, the coding part is rich because of being based on data structures of great expressivity (graphs, strings, trees…), allowing in particular to represent in an adequate way any kind of intra/inter patterns relations (reflexivity, sequentiality, hierarchy…). However, the tools related to the classification of structures are too restrictive (isomorphism, edit distance[4,6]...) and not robust enough for some applications specific to pattern recognition. Another limitation comes from the lack of a structural formalism for the processing of sets of data, in the sense that the tools classically used are generally based on only unary or binary operators. Finally, association between, on the one hand, the size of the data structures used, and, on the other hand, the complexity of the relative algorithms, tends to reject this approach for the processing of large volumes of data.

To be able to reconcile these two approaches, a necessary condition is to define a statistical characterization of spaces of structures. We propose to contribute to this vision, by translating the concept of distribution. We especially concentrate our attention to strings for which we propose the definition of a uniform distribution.

The uniform process is specified in a set S, with respect to a measure function μ, by the distribution for which the probability of a subset E of S, measurable by μ, is proportional to its measure: $\mathrm{P}(E) = \alpha \times \mu(E)$. More precisely, α is just the

* A part of this work was done when the author visited the *Gruppo di Ricerca sulle Macchine Intelligenti per il riconoscimento di Video, Immagini e Audio*, *Università degli Studi di Salerno*, Italy. The visit was supported by a grant from the *Région Rhône-Alpes*, France.

Sébastien Rebecchi, Jean-Michel Jolion: On the Uniform Distribution of Strings, pp. 116–125.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

inverse of total measure $\mu(S)$ of S, that makes the uniform process be considered as a mere normalization one, passing from a measure to a distribution while respecting the *relative* measure of E in S.

Two well-known examples of such a specification are given in the discrete and continous one-dimension numerical cases, where the measure functions are, respectively, the cardinality ($\mu(E) = |E|$), and the Lebesgue measure (roughly speaking, $\mu(E) = \sum_{I \in \mathrm{MaxLen}(E)} l(I)$, with $\mathrm{MaxLen}(E)$ the set of maximal-length intervals that are subset of E, and $l(I)$ the length of the interval I). With respect to these measures, the uniform property demands, in the discrete case, that all elements have the same probability, and, in the continous one, that all intervals of the same length have the same probability.

As for the string case, we have to define a consistent measure for sets of such structures, *i.e.* a one that would take care of the inherent structural and combinatorial nature of this type of elements. Before going into details in section 3, we introduce some necessary notations and definitions.

2 Preliminary notations and definitions

Definition 1 (Alphabet). *An alphabet is a non-empty finite set whose elements are called letters.*

In the rest of this paper, A denotes an alphabet, and λ a special object, called *empty letter*, that does not belong to A . Moreover, we denote by $|A|$ the size of A, *i.e.* its cardinal.

Definition 2 (String). *A string over A is a finite-length sequence of letters of A.*

Let X be a string over A, and $n \in \mathbb{N}$. We use the following notation:

- $|X|$ the size of X, *i.e.* its length
- X_i the letter at position i in X, $i \in \{1, \dots, |X|\}$
- Λ the empty string over A, *i.e.* of length 0
- A^n the set of strings over A of length n
- $A^{\leqslant n}$ the set of strings over A of length not greater than n
- A^* the set of strings over A

Notice that:

- $A^0 = \{\Lambda\}$
- $A^{\leqslant n} = \bigcup_{i=0}^{n} A^i$
- $A^* = \bigcup_{i \in \mathbb{N}} A^i$

Definition 3 (Concatenation). *The concatenation over A is the binary operation* $. : A^* \times (A^* \cup A \cup \{\lambda\}) \longrightarrow A^*$*, such that:* $\forall (X, Y \in A^*, a \in A)$*:*

- $X.\lambda = X$
- $[|X.a| = |X| + 1] \wedge [\forall i \in \{1, \dots, |X|\}, (X.a)_i = X_i] \wedge [(X.a)_{|X|+1} = a]$
- $[|X.Y| = |X| + |Y|] \wedge [\forall i \in \{1, \dots, |X|\}, (X.Y)_i = X_i] \wedge [\forall i \in \{1, \dots, |Y|\}, (X.Y)_{|X|+i} = Y_i]$

Remark. $\Lambda.X = X.\Lambda = X$ follows from the last point of this definition.

Thanks to definition 3, we can "promote" a letter of $A \cup \{\lambda\}$ as a string of $A^{\leqslant 1}$, simply by concatenating it to Λ.

Definition 4 (Promotion). *The promotion over A is the bijection:*

$$\begin{aligned} A \cup \{\lambda\} &\to A^{\leqslant 1} \\ a &\to \Lambda.a \end{aligned}$$

Moreover, we say that a is the *promoted* of $\Lambda.a$. Notice that the empty letter λ is promoted to the empty string Λ, and that a non-empty letter is promoted to a non-empty string of length 1.

Definition 5 (σ-algebra). *Let S be a set. A σ-algebra σ over S is a set of subsets of S, such that:*

- *σ contains the empty set: $\emptyset \in \sigma$*
- *σ is closed under complementation: $E \in \sigma =:(S \setminus E) \in \sigma$*
- *σ is closed under countably infinite union:*

$$[\forall n \in \mathbb{N}, E_n \in \sigma] =: \left(\bigcup_{n\in\mathbb{N}} E_n\right) \in \sigma$$

Remark. If S is countable, then the power set (set of all subsets) of S is the only σ-algebra over S containing all singletons $\{x\}$, $x \in S$.

Definition 6 (Measure). *Let S be a set, and σ a σ-algebra over S. A measure μ over σ is a function $\sigma \to \mathbb{R}^+ \cup \{\infty\}$, such that:*

- *The empty set has a null measure: $\mu(\emptyset) = 0$*
- *μ is additive under disjoint countably infinite union:*

$$[\forall (i, j \in \mathbb{N} | i < j), E_i, E_j \in \sigma, E_i \cap E_j = \emptyset]$$

$$=:$$

$$\mu\left(\bigcup_{n\in\mathbb{N}} E_n\right) = \sum_{n\in\mathbb{N}} \mu(E_n)$$

In the rest of this paper, we simplify the notation $\mu(\{x\})$ by $\mu(x)$, for all singletons $\{x\}$, $x \in S$.

Definition 7 (Uniform distribution). *Let S be a set, σ a σ-algebra over S, and μ a measure over σ. A distribution is uniform w.r.t. μ iff: $\forall E \in \sigma$:*

$$\mathrm{P}(E) = \mu(E) \times \mu(S)^{-1}$$

In the rest of this paper, we simplify the notation $\mathrm{P}(\{x\})$ by $\mathrm{P}(x)$, for all singletons $\{x\}$, $x \in S$.

3 String uniform distribution

3.1 The measure

Let μ_A be a measure over the power set of $A \cup \{\lambda\}$, $n \in \mathbb{N}$, and σ^n the power set of $A^{\leqslant n}$. Our wish is to define a measure μ^n over σ^n.

We wish μ^n to respect two necessary properties relatives to the connection between, on the one hand, the combinatorial and structural nature of the string type, and, on the other hand, the set $A^{\leqslant n}$ for which is defined this measure:

1. μ^n has an additive effect under combination
2. μ^n has an multiplicative effect under concatenation

The first property would be related with the combinatorial nature of a string, considering it within the set $A^{\leqslant n}$: one can attempt to describe a string X of $A^{\leqslant n}$ with a n-tuple of letters of $A \cup \{\lambda\}$, simply by preserving the respective order of the letters of X, and padding with the appropriate number of λ. But in this case, one must face the problem that the number of possible λ-paddings is equal to the number of combinations of size $|X|$ from a set of cardinal n. This caracteristic must hold in the definition of μ^n: if we see a string X of $A^{\leqslant n}$ as being the canonical representation of a set of n-tuples, then, according to the additive property of a measure (*cf.* definition 6), the measure $\mu^n(X)$ should be obtained by the sum of the measures of all tuples associated to X, with respect to a measure defined over the power set of the set of n-tuples of letters of $A \cup \{\lambda\}$. If we denote by μ_t^n this intermediate n-tuple measure, then we have:

$$\mu^n(X) = \sum_{T \in \text{tuples}_n(X)} \mu_t^n(T)$$

where $\text{tuples}^n(X)$ stands for the set of n-tuples of $A \cup \{\lambda\}$ associated to X.

As for the second property, it would be related with the structural sequential nature of a string. Keeping in mind the viewpoint introduced above, according to which a string is the canonical representation of a set of tuples, we wish to consider a letter of X as its expression in a certain dimension of each of these tuples. We wish not to regard the particular position of this letter in a tuple, because it could introduce a specific relative importance of any particular position in a string, which is not our purpose in this general study, as we wish to preserve the one and only this one induced by μ_A. Then, each tuple composed of the same letters should have the same measure, and, according to this dimensional point of view, we have:

$$\mu_t^n(T) = \prod_{i=1}^{|T|} \mu_A(T_i)$$

The reasonning above drives us to the following definition:

Definition 8 (String measure). $\forall X \in A^{\leqslant n}$*:*

$$\mu^n(X) = \mathrm{C}_n^{|X|} \times \left(\prod_{i=1}^{|X|} \mu_A(X_i) \right) \times \mu_A(\lambda)^{n-|X|}$$

where $\mathrm{C}_n^{|X|}$ stands for the number of combinations of size $|X|$ from a set of cardinal n.

The measure of non-singleton sets simply follows from the property of a measure (*cf.* definition 6): sum of the measures of all the singletons that are subset of it (hence 0 for the empty set). Therefore, it would be straightforward to prove that μ^n *is* a measure over the power set of $A^{\leqslant n}$.

Finally, notice that the above formula follows the recursive rule below, that is going to be useful in 3.2:

Proposition 9 (String measure recursion). $n > 0 =: \forall(X \in A^{\leqslant n-1}, a \in A)$*:*

$$\mu^n(X.a) = \mu^n(X) \times \mathrm{C}_n^{|X|+1} \,/\, \mathrm{C}_n^{|X|} \times \mu_A(a)/\mu_A(\lambda)$$

Proof.

$$\begin{aligned}
\mu^n(X.a) = & \ \mathrm{C}_n^{|X.a|} \times \left(\prod_{i=1}^{|X.a|} \mu_A((X.a)_i)\right) \times \mu_A(\lambda)^{n-|X.a|} \\
=^* & \ \mathrm{C}_n^{|X|+1} \times \left(\prod_{i=1}^{|X|} \mu_A(X_i)\right) \times \mu_A(a) \times \mu_A(\lambda)^{n-|X|-1} \\
= & \ \left[\mathrm{C}_n^{|X|} \times \left(\prod_{i=1}^{|X|} \mu_A(X_i)\right) \times \mu_A(\lambda)^{n-|X|}\right] \times \mathrm{C}_n^{|X|+1} \,/\, \mathrm{C}_n^{|X|} \times \mu_A(a) \times \mu_A(\lambda)^{-1} \\
= & \ \mu^n(X) \times \mathrm{C}_n^{|X|+1} \,/\, \mathrm{C}_n^{|X|} \times \mu_A(a)/\mu_A(\lambda)
\end{aligned}$$

$*$ Definition 3 □

We impose $n > 0$ because the set A^{-1} is undefined ($-1 \notin \mathbb{N}$).

3.2 The distribution

Following the general requirement of the uniform specification (*cf.* definition 7) with respect to the measure μ^n, the probability of a string is given by the following formula:

Definition 10 (Uniform string distribution). $\forall X \in A^{\leqslant n}$*:*

$$\mathrm{P}^n(X) = \mu^n(X) \times \mu^n(A^{\leqslant n})^{-1}$$

It would be much complex to compute the total measure $\mu^n(A^{\leqslant n})$ of $A^{\leqslant n}$ by a naive recursive exponential time demanding enumeration of all strings of this set (remind that $\mu^n(A^{\leqslant n}) = \sum_{Y \in A^{\leqslant n}} \mu^n(Y)$). Fortunately, we can simplify it analytically:

Proposition 11 (Total string measure).

$$\mu^n(A^{\leqslant n}) = \mu_A(A \cup \{\lambda\})^n$$

The following formula is going to be helpful to prove the proposition:

Lemma 12. $\forall i \in \{0, \ldots, n-1\}$*:*

$$\mu^n(A^{i+1}) = \mu^n(A^i) \times \mathrm{C}_n^{i+1} \,/\, \mathrm{C}_n^{i} \times \mu_A(A)/\mu_A(\lambda)$$

Then, the proof of the proposition follows in a simple way:

Proof (Proposition 11).

$$\text{Lemma 12} \wedge \left[\mu^n(A^0) = \mu^n(\Lambda) =^* \mu_A(\lambda)^n\right] =: \forall i \in \{0, \ldots, n\}:$$

$$\begin{aligned}
\mu^n(A^i) &= \mu_A(\lambda)^n \times \textstyle\prod_{j=1}^{i} \left(\mathrm{C}_n^j \,/\, \mathrm{C}_n^{j-1} \times \mu_A(A)/\mu_A(\lambda)\right) \\
&= \mu_A(\lambda)^n \times \mu_A(A)^i/\mu_A(\lambda)^i \times \textstyle\prod_{j=1}^{i} \mathrm{C}_n^j \,/\, \mathrm{C}_n^{j-1} \\
&= \mu_A(\lambda)^{n-i} \times \mu_A(A)^i \times \mathrm{C}_n^i \,/\, \mathrm{C}_n^0 \\
&= \mu_A(\lambda)^{n-i} \times \mu_A(A)^i \times \mathrm{C}_n^i
\end{aligned}$$

Then, we have:

$$\begin{aligned}
\mu^n(A^{\leqslant n}) &= \textstyle\sum_{i=0}^{n} \mu^n(A^i) \\
&= \textstyle\sum_{i=0}^{n} \left(\mu_A(\lambda)^{n-i} \times \mu_A(A)^i \times \mathrm{C}_n^i\right) \\
&=^{**} \left(\mu_A(\lambda) + \mu_A(A)\right)^n \\
&= \mu_A(A \cup \{\lambda\})^n
\end{aligned}$$

$*$ Definition 8
$**$ Binomial theorem □

It now remains to prove the lemma:

Proof (Lemma 12). If $n = 0$, then $\{0, \ldots, n-1\} = \emptyset$, and thereby the lemma is true by vacuity. Else ($n > 0$), we have:

$$\begin{aligned}
\mu^n(A^{i+1}) &= \textstyle\sum_{Y \in A^{i+1}} \mu^n(Y) \\
&=^* \textstyle\sum_{X \in A^i} \sum_{a \in A} \mu^n(X.a) \\
&=^{**} \textstyle\sum_{X \in A^i} \sum_{a \in A} \left(\mu^n(X) \times \mathrm{C}_n^{|X|+1} \,/\, \mathrm{C}_n^{|X|} \times \mu_A(a)/\mu_A(\lambda)\right) \\
&= \textstyle\sum_{X \in A^i} \sum_{a \in A} \left(\mu^n(X) \times \mathrm{C}_n^{i+1} \,/\, \mathrm{C}_n^{i} \times \mu_A(a)/\mu_A(\lambda)\right) \\
&= \textstyle\sum_{X \in A^i} \left(\mu^n(X) \times \mathrm{C}_n^{i+1} \,/\, \mathrm{C}_n^{i} \times \sum_{a \in A} \mu_A(a)/\mu_A(\lambda)\right) \\
&= \textstyle\sum_{X \in A^i} \left(\mu^n(X) \times \mathrm{C}_n^{i+1} \,/\, \mathrm{C}_n^{i} \times \mu_A(A)/\mu_A(\lambda)\right) \\
&= \left(\textstyle\sum_{X \in A^i} \mu^n(X)\right) \times \mathrm{C}_n^{i+1} \,/\, \mathrm{C}_n^{i} \times \mu_A(A)/\mu_A(\lambda) \\
&= \mu^n(A^i) \times \mathrm{C}_n^{i+1} \,/\, \mathrm{C}_n^{i} \times \mu_A(A)/\mu_A(\lambda)
\end{aligned}$$

$*$ Definition 3
$**$ Proposition 9 □

According to definition 8 and proposition 11, we can compute the probability of a string in $O(n)$:

Definition 13 (Uniform string distribution). $\forall X \in A^{\leqslant n}$*:*

$$\mathrm{P}^n(X) = \mathrm{C}_n^{|X|} \times \left(\prod_{i=1}^{|X|} \mu_A(X_i)\right) \times \mu_A(\lambda)^{n-|X|} \times \mu_A(A \cup \{\lambda\})^{-n}$$

3.3 Preservation

An interesting property of our string uniform distribution is the preservation under concatenation: the concatenation of two uniform strings remains a uniform string:

Proposition 14 (Uniform string preservation). $\forall(i \in \{0, \ldots, n\}, Y \in A^{\leqslant i}, Z \in A^{\leqslant n-i})$*, if* Y *is uniformly distributed w.r.t.* μ^i*, and* Z *uniformly distributed w.r.t.* μ^{n-i}*, then* $X = Y.Z$ *is uniformly distributed w.r.t.* μ^n*.*

Proof. First, remind that we have: $\forall i, k \in \{0, \dots, n\}$:

$$\mathrm{C}_n^k = \sum_{j=0}^{k} \left(\mathrm{C}_i^j \times \mathrm{C}_{n-i}^{k-j} \right)$$

Then, we have: $\forall X \in A^{\leqslant n}$:

$$\begin{aligned} \mathrm{C}_n^{|X|} =& \sum_{j=0}^{|X|} \left(\mathrm{C}_i^j \times \mathrm{C}_{n-i}^{|X|-j} \right) \\ =^*& \sum_{Y \in A^{\leqslant i}, Z \in A^{\leqslant n-i} | Y.Z = X} \left(\mathrm{C}_i^{|Y|} \times \mathrm{C}_{n-i}^{|Z|} \right) \end{aligned}$$

Moreover, we have: $\forall (Y \in A^{\leqslant i}, Z \in A^{\leqslant n-i} | Y.Z = X)$:

$$\begin{aligned} \mathrm{P}^i(Y) \times \mathrm{P}^{n-i}(Z) =^{**}& \left[\mathrm{C}_i^{|Y|} \times \left(\prod_{j=1}^{|Y|} \mu_A(Y_j) \right) \times \mu_A(\lambda)^{i-|Y|} \times \mu_A(A \cup \{\lambda\})^{-i} \right] \times \\ & \left[\mathrm{C}_{n-i}^{|Z|} \times \left(\prod_{j=1}^{|Z|} \mu_A(Z_j) \right) \times \mu_A(\lambda)^{(n-i)-|Z|} \times \mu_A(A \cup \{\lambda\})^{-(n-i)} \right] \\ =^*& \ \mathrm{C}_i^{|Y|} \times \mathrm{C}_{n-i}^{|Z|} \times \left(\prod_{j=1}^{|Y.Z|} \mu_A((Y.Z)_j) \right) \times \mu_A(\lambda)^{n-|Y.Z|} \times \\ & \mu_A(A \cup \{\lambda\})^{-n} \\ =& \ \mathrm{C}_i^{|Y|} \times \mathrm{C}_{n-i}^{|Z|} \times \left(\prod_{j=1}^{|X|} \mu_A(X_j) \right) \times \mu_A(\lambda)^{n-|X|} \times \\ & \mu_A(A \cup \{\lambda\})^{-n} \end{aligned}$$

Finally, we have:

$$\begin{aligned} \mathrm{P}^n(X) =^{**}& \ \mathrm{C}_n^{|X|} \times \left(\prod_{j=1}^{|X|} \mu_A(X_j) \right) \times \mu_A(\lambda)^{n-|X|} \times \mu_A(A \cup \{\lambda\})^{-n} \\ =& \sum_{Y \in A^{\leqslant i}, Z \in A^{\leqslant n-i} | Y.Z = X} \left(\mathrm{C}_i^{|Y|} \times \mathrm{C}_{n-i}^{|Z|} \right) \times \left(\prod_{j=1}^{|X|} \mu_A(X_j) \right) \times \mu_A(\lambda)^{n-|X|} \times \\ & \mu_A(A \cup \{\lambda\})^{-n} \\ =& \sum_{Y \in A^{\leqslant i}, Z \in A^{\leqslant n-i} | Y.Z = X} \left[\mathrm{C}_i^{|Y|} \times \mathrm{C}_{n-i}^{|Z|} \times \left(\prod_{j=1}^{|X|} \mu_A(X_j) \right) \times \mu_A(\lambda)^{n-|X|} \times \right. \\ & \left. \mu_A(A \cup \{\lambda\})^{-n} \right] \\ =& \sum_{Y \in A^{\leqslant i}, Z \in A^{\leqslant n-i} | Y.Z = X} \left[\mathrm{P}^i(Y) \times \mathrm{P}^{n-i}(Z) \right] \end{aligned}$$

We deduce that, for all $i \in \{0, \dots, n\}$, the probability of a uniform string X w.r.t. μ^n is equal to the sum of the probabilities of all the possible concatenations of a uniform string Y w.r.t. μ^i, with a uniform string Z w.r.t. μ^{n-i}, such that $X = Y.Z$. This is exactly the meaning of the proposition.

$*$ Definition 3
$**$ Definition 13 □

This general reasoning leads to the following specific corollary, that is going to be useful in 3.4:

Corollary 15 (Uniform string preservation). $\forall (X \in A^{\leqslant n}, a \in A \cup \{\lambda\})$, *if X is uniformly distributed w.r.t. μ^n, and a uniformly distributed w.r.t. μ_A, then $X.a$ is uniformly distributed w.r.t. μ^{n+1}.*

The proof of this corollary follows from the following lemma:

Lemma 16. $\forall a \in A \cup \{\lambda\}$, *$a$ is uniformly distributed w.r.t. μ_A iff $\Lambda.a$ is uniformly distributed w.r.t. μ^1.*

Proof (Corollary 15). According to lemma 16, $\Lambda.a$ is uniformly distributed w.r.t. μ^1. Then, according to proposition 14, $X.(\Lambda.a) =^* X.a$ is uniformly distributed w.r.t. μ^{n+1}.

$*$ Definition 3 □

Proof (Lemma 16). Let us denote by $\mathrm{P}_A(a)$ the probability of a acording to a uniform distribution w.r.t. μ_A. Then, we have:

$$\mathrm{P}_A(a) =^* \mu_A(a) \times \mu_A(A \cup \{\lambda\})^{-1}$$

If $a = \lambda$, then we have: $\mathrm{P}_A(a) = \mathrm{C}_1^0 \times 1 \times \mu_A(\lambda)^{1-0} \times \mu_A(A \cup \{\lambda\})^{-1}$.
Else ($a \in A$), we have: $\mathrm{P}_A(a) = \mathrm{C}_1^1 \times \mu_A(a) \times \mu_A(\lambda)^{1-1} \times \mu_A(A \cup \{\lambda\})^{-1}$.
So in all cases, we have:

$$\begin{aligned}\mathrm{P}_A(a) &=^{**} \mathrm{C}_1^{|\Lambda.a|} \times \left(\prod_{i=1}^{|\Lambda.a|} \mu_A((\Lambda.a)_i)\right) \times \mu_A(\lambda)^{1-|\Lambda.a|} \times \mu_A(A \cup \{\lambda\})^{-1} \\ &=^{***} \mathrm{P}^1(\Lambda.a)\end{aligned}$$

This proves the lemma, as the promotion is a bijection (*cf.* definition 4).

$*$ Definition 7
$**$ Definition 3
$***$ Definition 13 □

3.4 Generation

We wish to generate strings according to our uniform distribution. First, notice that, according to definition 13, we have: $\forall X \in A^{\leqslant n}$:

$$\begin{aligned}\mathrm{P}^n(X) &= \mathrm{C}_n^{|X|} \times \left(\prod_{i=1}^{|X|} \mu_A(X_i)\right) \times \mu_A(\lambda)^{n-|X|} \times \mu_A(A \cup \{\lambda\})^{-n} \\ &= \mathrm{C}_n^{|X|} \times \prod_{i=1}^{|X|} \left(\mu_A(X_i) \times \mu_A(A \cup \{\lambda\})^{-1}\right) \times \prod_{i=1}^{n-|X|} \left(\mu_A(\lambda) \times \mu_A(A \cup \{\lambda\})^{-1}\right)\end{aligned}$$

This equation tells us that it is sufficient to, first, generate a n-tuple by the concatenation of n elements of $A \cup \{\lambda\}$ generated according to a uniform distribution w.r.t. μ_A, and, then, remove all the λ in T, to finaly obtain a string X generated according to a uniform distribution w.r.t. μ^n: according to this procedure, T would have a probability $\prod_{i=1}^{|X|} (\mu_A(X_i) \times \mu_A(A \cup \{\lambda\})^{-1}) \times \prod_{i=1}^{n-|X|} (\mu_A(\lambda) \times \mu_A(A \cup \{\lambda\})^{-1})$, and therefore X a probability $\mathrm{P}^n(X)$, as we have seen in 3.1 that X is the canonical representation of $\mathrm{C}_n^{|X|}$ such tuples T.

But this sufficiency can obviously also be retrieved in a slightly different form from a recursive use of the uniformity conversation of a string when concatenated with a uniform string w.r.t. μ^1, or its promoted uniform letter w.r.t. μ_A, as exhibit by corollary 15.

A simple $O(n)$-complex pseudo-code implementation of such a procedure is given by algorithm 1. Instead of passing by an intermediate tuple initialisation, filling, to a final canonization to string, the algorithm works directly under the string representation, thanks to a sequence of concatenations of a uniform letter to a string initialised to the empty one, for the same result, as mentioned above.

Input: A positive integer n
Output: A string X generated according to a uniform distribution w.r.t. μ^n
begin
  $D \longleftarrow$ uniform distribution w.r.t. μ_A: $\forall a \in A \cup \{\lambda\}$, $\mathrm{P}(a) = \mu_A(a)/\mu_A(A \cup \{\lambda\})$;
  $X \longleftarrow \Lambda$;
  for $i \longleftarrow 1$ **to** n **do**
    $l \longleftarrow$ random choice according to D;
    $X \longleftarrow X.l$;
  end
  return X;
end

Algorithm 1: Uniform string generation

4 Relation with a previous work

The present work subsumes and generalizes the part concerning the uniform distribution of the one exposed in [5] (in french): one can retrieve the special definition proposed in the later paper by setting $\mu_A(\lambda) = 0$, and $\mu_A(a) = \mu_A(b), \forall a, b \in A$ (this last equality can reasonably be supposed in numerous applications). One can also consider the restriction of μ^n over the restriction of σ^n over A^n, by ignoring the null λ influence, that leads to $\mathrm{P}^n(A^{\leqslant n-1}) = 0$ if $n > 0$. This would also define a uniform distribution w.r.t. the considered restriction of μ^n.

5 Discussion and further work

In fact, this work can be summarized as an analytical justification of this proposition: one can generate a uniform string over the alphabet A by concatenating uniform letters of $A \cup \{\lambda\}$. The simplicity of this property is obsiously due to the particular measure defined for sets of strings, but we have seen that this definition does not only please our wish for simplicity, but also fulfill some relevant arguments concerning the nature of the type of elements taken into consideration.

This simplicity could be rather easily extended to more complex structure types, for which we can define an order of the primitives. Let us consider, for instance, the (ordered) bounded arity and depth tree over A: if a is the maximum number of children that can have a node (arity), and d the maximum depth that can a tree, we could recursively define such a tree by a couple composed of the root and a string of children of length not greater than a, the children alphabet being the set of non-empty trees of arity a and maximum depth $(d-1)$, with the empty tree as associated empty letter. Then, we could take some ideas to the work above to instanciate a uniform distribution of such trees. A part of the associated tree measure would be defined thanks to the definition of the string one, with a necessary supplementary condition (and potential difficulty) induced by the hierarchical structural nature of trees.

To conclude, we can say that this work is a prelude and it would be interesting to study the possible properties that could arise when combining, in a manner to be specified yet, independent executions of the same uniform string distribution. This would tend to a string distributed according to a gaussian string distribution, as told by the central limit theorem. A necessary condition is to define the concept of random variable in a measurable vector space for strings. It could be sufficient to order the strings to keep the real line as the state set of such random variables, and thus take advantage of the classical (numerical) probabilistic statistical theory, but we could also develop a specific different point of view.

References

1. A. K. Jain, R. P. W. Duin, and J. Mao: *Statistical pattern recognition: a review.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 22(1) 2000, pp. 4–37.
2. J.-M. Jolion: *The deviation of a set of strings.* Pattern Analysis and Applications, 6(3) 2003, pp. 224–231.
3. T. Kohonen: *Median strings.* Pattern Recognition Letters, 3(5) 1985, pp. 309–313.
4. V. I. Levenshtein: *Binary codes capable of correcting deletions, insertions, and reversals.* Soviet Physics Doklady, 10(8) 1966, pp. 707–710.
5. S. Rebecchi and J.-M. Jolion: *Lois uniformes et normales de chanes discrtes*, in RFIA, Amiens, France, January 2008, pp. 471–480.
6. R. A. Wagner and M. J. Fischer: *The string-to-string correction problem.* Journal of the ACM, 21(1) 1974, pp. 168–173.

Infinite Smooth Lyndon Words

Geneviève Paquin

Laboratoire de Combinatoire et d'Informatique Mathématique,
Université du Québec à Montréal,
Montréal (QC) CANADA H3C 3P8,
genevieve.paquin@gmail.com

Abstract. In a recent paper, Brlek et al. showed that some extremal infinite smooth words are also infinite Lyndon words. This result raises a natural question: what are the infinite smooth words that are also infinite Lyndon words? In this paper, we give the answer: the only infinite smooth Lyndon words are $m_{\{a<b\}}$, with a, b even, and $m_{\{1<b\}}$, with b odd, where $m_{\mathcal{A}}$ is the minimal infinite smooth word with respect to lexicographic order over the numerical alphabet $\mathcal{A}$.

Keywords: Lyndon words, smooth words, Kolakoski sequence

1 Introduction

Lyndon words were introduced by Lyndon in [9] for constructing bases of the lower central series for free groups. The authors proved that any finite word can be expressed as a unique non-increasing product of Lyndon words. Later, Lyndon words were studied by Duval [11,12]. He gave an algorithm that generates Lyndon words of bounded length for a finite alphabet and another one that computes the Lyndon factorization in linear time. Siromoney et al [26] defined infinite Lyndon words in order to introduce Lyndon factorization of infinite words. Lyndon words also appeared in [18,20,22]. This factorization gives nice properties about the structure of words. Since a few years, a wide literature is devoted to Lyndon words: [2,13,23,24,25]. For instance, Melançon [19] studied Lyndon factorization of Sturmian infinite words.

Smooth infinite words over $\mathcal{A} = \{1, 2\}$ form an infinite class $\mathcal{K}$ of infinite words containing the well-known Kolakoski word K [17] defined as one of the two fixed points of the run-length encoding function Δ, that is

$$\Delta(K) = K = 2211212212211211221211212211211212212211212212\cdots .$$

They are characterized by the property that the orbit obtained by iterating Δ is contained in $\{1, 2\}^*$. In the early work of Dekking [10], there are some challenging conjectures on the structure of K that still remain unsolved despite the efforts devoted to the study of patterns in K. For instance, we know from Carpi [8] that K and more generally, any word in the infinite class $\mathcal{K}$ of smooth words over $\mathcal{A} = \{1, 2\}$, contain only a finite number of squares, implying by direct inspection that K and any $w \in \mathcal{K}$ are cube-free. Weakley [17] showed that the number of factors of length n of $\mathcal{K}$ is polynomially bounded. In [6], a connection was established between the palindromic complexity and the recurrence of K. Then, Berthé et al. [3] studied smooth words over arbitrary alphabets and obtained a new characterization of the infinite Fibonacci word F. Relevant work may also be found in [1] and in [3,16], where generalized Kolakoski words are studied for arbitrary alphabets. The authors investigated in [7] the extremal infinite smooth words, that is the minimal and the

Geneviève Paquin: Infinite Smooth Lyndon Words, pp. 126–139.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

maximal ones w.r.t. the lexicographic order, over $\{1,2\}$ and $\{1,3\}$: a surprising link is established between F and the minimal infinite smooth word over $\{1,3\}$.

More recently, Brlek et al. [5] studied the extremal smooth words for any 2-letter alphabet and they showed the existence of infinite smooth words that are also Lyndon words: the minimal smooth word over an even alphabet and the one over the alphabet $\{1,b\}$, with b odd, are Lyndon words. Then a natural question arises: are there other infinite smooth words that are infinite Lyndon words?

In this paper, we show that the minimal smooth words that are also Lyndon words given in [5] are the only smooth Lyndon words. In order to prove it, we study the words over a 2-letter alphabet depending on the parity of the letters. The paper is organized as follow. In Section 2, we recall the basic definitions in combinatorics on words, we state the notation we will use next and we recall useful known results. Section 3 is devoted to the characterization of infinite smooth Lyndon words. It is divided in 4 subsections. In Section 3.1, we study the case of an alphabet $\mathcal{A}=\{a<b\}$, with a even and b odd. We show that there is no infinite Lyndon words that is also smooth. In Section 3.2, we are interested in even alphabets. We show that only the minimal smooth word is a Lyndon word. Section 3.3 is devoted to odd alphabet. We prove that only $m_{\{1,b\}}$ is a Lyndon word. Finally, Section 3.4 studies the words over an alphabet $\{a<b\}$ with a odd and b even. In this last case, we show that there is no infinite Lyndon words that are also smooth.

Notice that some proofs are omitted for lack of space and will appear in a full paper.

2 Preliminaries

Throughout this paper, $\mathcal{A}$ is a finite *alphabet* of *letters* equipped with a total order $<$. A *finite word* w is a finite sequence of letters $w=w[0]w[1]\cdots w[n-1]$, where $w[i]\in\mathcal{A}$ denotes its $(i+1)$-th letter. Its length is n and we write $|w|=n$. The set of n-length words over $\mathcal{A}$ is denoted by $\mathcal{A}^n$. By convention the *empty* word is denoted by ε and its length is 0. The free monoid generated by $\mathcal{A}$ is defined by $\mathcal{A}^*=\bigcup_{n\geq 0}\mathcal{A}^n$ and $\mathcal{A}^*\setminus\varepsilon$ is denoted $\mathcal{A}^+$. The set of right infinite words, also called infinite words for short, is denoted by $\mathcal{A}^\omega$ and $\mathcal{A}^\infty=\mathcal{A}^*\cup\mathcal{A}^\omega$. Adopting a consistent notation for finite words over the infinite alphabet $\mathbb{N}$, $\mathbb{N}^*=\bigcup_{n\geq 0}\mathbb{N}^n$ is the set of finite sequences and $\mathbb{N}^\omega$ is that of infinite ones. Given a word $w\in\mathcal{A}^*$, a *factor* f of w is a word $f\in\mathcal{A}^*$ satisfying

$$\exists x,y\in\mathcal{A}^*, w=xfy.$$

If $x=\varepsilon$ (resp. $y=\varepsilon$) then f is called a *prefix* (resp. *suffix*). Note that by convention, the empty word is suffix and prefix of any word. A *block* of length k is a maximal factor of the particular form $f=\alpha^k$, with $\alpha\in\mathcal{A}$. The set of all factors of w, also called the *language* of w, is denoted by $F(w)$, and those of length n is $F_n(w)=F(w)\cap\mathcal{A}^n$. We denote by Pref$(w)$ (resp. Suff(w)) the set of all prefixes (resp. suffixes) of w.

Over an arbitrary 2-letter alphabet $\mathcal{A}=\{a,b\}$, there is a usual length preserving morphism, the *complementation*, defined by $\overline{a}=b$, $\overline{b}=a$, which extends to words as follows. The complement of $u=u[0]u[1]\cdots u[n-1]\in\mathcal{A}^n$ is the word $\overline{u}=\overline{u[0]}\,\overline{u[1]}\cdots\overline{u[n-1]}$. The *reversal* of u is the word $\widetilde{u}=u[n-1]\cdots u[1]u[0]$.

For $u,v\in\mathcal{A}^*$, we write $u<v$ if and only if u is a proper prefix of v or if there exists an integer k such that $u[i]=v[i]$ for $0\leq i\leq k-1$ and $u[k]<v[k]$. The relation $\leq$ defined by $u\leq v$ if and only if $u=v$ or $u<v$, is called the *lexicographic order*.

That definition holds for $\mathcal{A}^\infty$. Note that in general the complementation does not preserve the lexicographic order. Indeed, when u is not a proper prefix of v then

$$u > v \iff: \overline{u} < \overline{v}. \tag{1}$$

A word $u \in \mathcal{A}^*$ is a *Lyndon word* if $u < v$ for all proper non-empty suffixes v of u. For instance, the word 11212 is a Lyndon word while 12112 is not since $112 < 12112$. A word of length 1 is clearly a Lyndon word. The set of Lyndon words is denoted by $\mathcal{L}$.

From Lothaire [18], we have the following theorem.

Theorem 1. [9] *Any non empty finite word w is uniquely expressed as a non increasing product of Lyndon words*

$$w = \ell_0\ell_1\cdots\ell_n = \bigodot_{i=0}^{n} \ell_i, \text{ with } \quad \ell_i \in \mathcal{L}, \text{ and } \quad \ell_0 \geq \ell_1 \geq \cdots \geq \ell_n. \tag{2}$$

Siromoney et al. [26] extended Theorem 1 to infinite words. The set $\mathcal{L}_\infty$ of *infinite Lyndon words* consists of infinite words smaller than any of their suffixes.

Theorem 2. [26] *Any infinite word w is uniquely expressed as a non increasing product of Lyndon words, finite or infinite, in one of the two following forms:*

i) *either there exists an infinite sequence $(\ell_k)_{k\geq 0}$ of elements in $\mathcal{L}$ such that*
$$w = \ell_0\ell_1\ell_2\cdots \text{ and for all } k,\ \ell_k \geq \ell_{k+1}.$$
ii) *there exist a finite sequence $\ell_0, \ldots, \ell_m$ $(m \geq 0)$ of elements in $\mathcal{L}$ and $\ell_{m+1} \in \mathcal{L}_\infty$ such that*
$$w = \ell_0\ell_1\cdots\ell_m\ell_{m+1} \text{ and } \ell_0 \geq \cdots \geq \ell_m > \ell_{m+1}.$$

Let us recall from ([18] Chapter 5.1) a useful property concerning Lyndon words.

Lemma 3. *Let $u, v \in \mathcal{L}$. We have $uv \in \mathcal{L}$ if and only if $u < v$.*

A direct corollary of this lemma is:

Corollary 4. *Let $u, v \in \mathcal{L}$, with $u < v$. Then uv^n, $u^nv \in \mathcal{L}$, for all $n \geq 0$.*

The widely known *run-length encoding* is used in many applications as a method for compressing data. For instance, the first step in the algorithm used for compressing the data transmitted by Fax machines consists of a run-length encoding of each line of pixels. Let $\mathcal{A} = \{a < b\}$ be an ordered alphabet. Then every word $w \in \mathcal{A}^*$ can be uniquely written as a product of factors as follows:

$$w = a^{i_0}b^{i_1}a^{i_2}\cdots \qquad \text{or} \qquad w = b^{i_0}a^{i_1}b^{i_2}\cdots$$

with $i_k \geq 1$ for $k \geq 0$. The operator giving the size of the blocks appearing in the coding is a function $\Delta : \mathcal{A}^* \longrightarrow \mathbb{N}^*$, defined by $\Delta(w) = i_0, i_1, i_2, \cdots$ which is easily extended to infinite words as $\Delta : \mathcal{A}^\omega \longrightarrow \mathbb{N}^\omega$.

For instance, let $\mathcal{A} = \{1, 3\}$ and $w = 13333133111$. Then

$$w = 1^1 3^4 1^1 3^2 1^3 \quad \text{and} \quad \Delta(w) = [1, 4, 1, 2, 3]\,.$$

When $\Delta(w) \subseteq \{1, 2, \cdots, 9\}^*$, the punctuation and the parentheses are often omitted in order to manipulate the more compact notation $\Delta(w) = 14123$. This example is a special case where the coding integers do not coincide with the alphabet on which is encoded w, so that Δ can be viewed as a partial function $\Delta : \{1,3\}^* \longrightarrow \{1,2,3,4\}^*$.

From now on, we only consider 2-letter alphabets $\mathcal{A} = \{a < b\}$, with $a, b \in \mathbb{N}\backslash\{0\}$.

Recall from [6] that Δ is not bijective since $\Delta(w) = \Delta(\overline{w})$, but commutes with the reversal ($\widetilde{\ }$), is stable under complementation ($\overline{\ }$) and preserves palindromicity. Since Δ is not bijective, pseudo-inverse functions

$$\Delta_a^{-1}, \Delta_b^{-1} : \mathcal{A}^* \longrightarrow \mathcal{A}^*$$

are defined for 2-letter alphabets by

$$\Delta_\alpha^{-1}(u) = \alpha^{u[1]}\overline{\alpha}^{u[2]}\alpha^{u[3]}\overline{\alpha}^{u[4]}\cdots, \quad \text{for } \alpha \in \{a, b\}.$$

Note that the pseudo-inverse function Δ^{-1} also commutes with the mirror image, that is,

$$\widetilde{\Delta_\alpha^{-1}(w)} = \Delta_\beta^{-1}(\widetilde{w}), \tag{3}$$

where $\beta = \alpha$ if $|w|$ odd and $\beta = \overline{\alpha}$ if $|w|$ is even.

The operator Δ may be iterated, provided the process is stopped when the coding alphabet changes or when the resulting word has length 1.

Example 5. Let $w = 13331113331333111333131333111333133311113331$. The successive application of Δ gives :

$\Delta^0(w) = \mathbf{1}3331113331333111333131333111333133311113331$;
$\Delta^1(w) = \mathbf{1}333133311133313331$;
$\Delta^2(w) = \mathbf{1}31333131$;
$\Delta^3(w) = \mathbf{1}113111$;
$\Delta^4(w) = \mathbf{3}13$;
$\Delta^5(w) = \mathbf{1}11$;
$\Delta^6(w) = \mathbf{3}$.

The set of *finite smooth words* over the alphabet $\mathcal{A}$ is defined by

$$\Delta_{\mathcal{A}}^{+} = \{w \in \mathcal{A}^* \mid \exists n \in \mathbb{N}, \Delta^n(w) = \alpha^i, \alpha \in \mathcal{A}, i \leq \beta \text{ and } \forall k \leq n, \Delta^k(w) \in \mathcal{A}^*\},$$

with β the greatest letter of the alphabet.

The operator Δ extends to infinite words (see [6]). Define the set of *infinite smooth words* over $\mathcal{A} = \{a, b\}$ by

$$\mathcal{K}_{\mathcal{A}} = \{w \in \mathcal{A}^\omega \mid \forall k \in \mathbb{N}, \Delta^k(w) \in \mathcal{A}^\omega\}.$$

The well-known [17] *Kolakoski word* denoted K is defined as the fix-point starting with the letter 2 of the operator Δ over the alphabet $\{1, 2\}$:

$$K = 2211212212211211221211212211211212212211\cdots$$

More generally, the operator Δ has two fix-points in $\mathcal{K}_{\mathcal{A}}$, namely

$$\Delta(K_{(a,b)}) = K_{(a,b)} \quad \text{and} \quad \Delta(K_{(b,a)}) = K_{(b,a)},$$

where $K_{(a,b)}$ is the generalized Kolakoski word [16] over the alphabet $\{a, b\}$ starting with the letter a.

Example 6. The Kolakoski word over $\mathcal{A}=\{1,2\}$ starting with the letter 2 is $K=K_{(2,1)}$. We also have $K_{(2,3)} = 22332223332233223332\cdots$ and $K_{(3,1)} = 333111333131333111133\cdots$.

A bijection $\Phi : \mathcal{K}_{\mathcal{A}} \longrightarrow \mathcal{A}^{\omega}$ is defined by

$$\Phi(w) = \Delta^0(w)[0]\Delta^1(w)[0]\Delta^2(w)[0]\cdots = \prod_{i\geq 0} \Delta^i(w)[0]$$

and its inverse is defined as follows. Let $u \in \mathcal{A}^k$, then $\Phi^{-1}(u) = w_k$, where

$$w_n = \begin{cases} u[k-1], & \text{if } n = 1; \\ \Delta^{-1}_{u[k-n]}(w_{n-1}), & \text{if } 2 \leq n \leq k. \end{cases}$$

Then for $k = \infty$, $\Phi^{-1}(u) = \lim_{k\to\infty} w_k = \lim_{k\to\infty} \Phi^{-1}(u[0..k-1])$.

Remark 7. With respect to the usual topology defined by

$$d((u_n)_{n\geq 0}, (v_n)_{n\geq 0}) := 2^{-\mathbf{min}\{j\in\mathbb{N}, u_j\neq v_j\}},$$

the limit exists because each iteration is a prefix of the next one.

Example 8. For the word w = 1333111333133311133313133311133313331113331 of Example 5, $\Phi(w) = \mathbf{1111313}$.

Note that since Φ is a bijection, the set of infinite smooth words is infinite. Moreover, given a prefix of $\Phi(w)$, for w a smooth word, we can construct a prefix of w as in the following example.

Example 9. Let $p = 1221$ be a prefix of $\Phi(w)$, with $w \in \{1,2\}^{\omega}$ an infinite smooth word. Then we compute from bottom to top, using the operator Δ^{-1}:
$\Delta^0(w) = 11221221\cdots$
$\Delta^1(w) = 2212\cdots$
$\Delta^2(w) = 21\cdots$
$\Delta^3(w) = 1\cdots$
Note that in $\Delta^2(w)$, the letter 1 is obtained by deduction, since $\Delta^3(w)$ indicates that the first block of letters of $\Delta^2(w)$ has length 1. The last written letter of every line is deduced by the same argument.

We recall from [7] the useful *right derivative* $D_r : \mathcal{A}^* \to \mathbb{N}^*$ defined by

$$D_r(w) = \begin{cases} \varepsilon & \text{if } \Delta(w) = \alpha,\ \alpha < b \text{ or } w = \varepsilon, \\ \Delta(w) & \text{if } \Delta(w) = xb, \\ x & \text{if } \Delta(w) = x\alpha,\ \alpha < b, \end{cases}$$

where $\alpha \in \mathbb{N}$ and $x \in \mathcal{A}^*$. A word w is *r-smooth* (also said *smooth prefix*) if $\forall k \geq 0,\ D_r^k(w) \in \mathcal{A}^*$. In other words, if a word w is *r-smooth*, then it is a prefix of at least one infinite smooth word (see [4] for more details).

Example 10. Let $w = 112112212$. Then $\Delta(w) = 212211$, $\Delta^2(w) = 1122$, $\Delta^3(w) = 22$ and $D_r(w) = 21221$, $D_r^2(w) = 112$, $D_r^3(w) = 2$.

Similarly, the operator D is defined over the alphabet $\{a < b\}$ by

$$D(w) = \begin{cases} \varepsilon & \text{if } \Delta(w) < b \text{ or } w = \varepsilon, \\ \Delta(w) & \text{if } \Delta(w) = bxb \text{ or } \Delta(w) = b, \\ bx & \text{if } \Delta(w) = bxu, \\ xb & \text{if } \Delta(w) = uxb, \\ x & \text{if } \Delta(w) = uxv, \end{cases}$$

where u and v are blocks of length $< b$. A finite word is called a *smooth factor* (also called a C^∞-word in [8,14,15,27]) if there exists $k \in \mathbb{N}$ such that $D^k(w) = \varepsilon$ and $\forall j < k,\ D^j(w) \in \mathcal{A}^*$.

The *minimal* (resp. the *maximal*) *infinite smooth word* over the alphabet $\mathcal{A}$ is the smallest (resp. biggest) infinite smooth word, with respect to the lexicographic order. It is denoted by $m_\mathcal{A}$ (resp. $M_\mathcal{A}$).

An alphabet $\mathcal{A} = \{a < b\}$ is called an *odd alphabet* (resp. *even alphabet*) if both a and b are odd (resp. even). The extremal smooth words satisfy the following properties established in a previous paper.

Proposition 11. [5] *Let $\mathcal{A} = \{a, b\}$ be a 2-letter alphabet with $a < b$. Then the following properties hold:*

i) *If a and b are both even we have :*
   $\Phi(M_{\{a,b\}}) = b^\omega$; $\Phi(m_{\{a,b\}}) = ab^\omega$; *and* $m_{\{a,b\}} \in \mathcal{L}_\infty$.
ii) *If a and b are both odd we have :*
   $\Phi(M_{\{a,b\}}) = (ba)^\omega$; $\Phi(m_{\{a,b\}}) = (ab)^\omega$; *and* $m_{\{a,b\}} \in \mathcal{L}_\infty \iff a = 1$.

Let us recall some known results about smooth words.

Lemma 12. [3] *Let u, v be finite smooth words. If there exists an index m such that, for all i, $0 \le i \le m$, the last letter of $\Delta^i(u)$ differs from the first letter of $\Delta^i(v)$, and $\Delta^i(u) \neq 1$, $\Delta^i(v) \neq 1$, then*

i) $\Phi(uv) = \Phi(u)[0..m] \cdot \Phi \circ \Delta^{m+1}(uv)$;
ii) $\Delta^i(uv) = \Delta^i(u)\Delta^i(v)$.

The following properties follow immediately from the definitions. For more details, the reader is referred to [21].

Recall from [4] that in the case of the alphabet $\mathcal{A} = \{1, 2\}$, every finite word $w \in \Delta^+_\mathcal{A}$ can be easily extended to the right in a smooth word by means of the function Φ as follows:

$$\forall u \in \mathcal{A}^\infty,\ w \in \mathrm{Pref}(\Phi^{-1}(\Phi(w) \cdot u)).$$

Its generalization to arbitrary alphabets is immediate (see [21]).

Proposition 13. *Let $\mathcal{A} = \{a < b\}$. Then the following properties hold.*

i) *Any smooth prefix can be arbitrarily right extended to an infinite smooth word.*
ii) *Let $u = \Phi(w)$, with $w \in \mathcal{A}^\omega$ an infinite smooth word. If $u = u'u''$, then $\Phi^{-1}(u')$ is prefix of w.*

3 Characterization of infinite smooth Lyndon words

In this section, we prove our main result: the only infinite smooth words that are also infinite Lyndon words are $m_{\{2a<2b\}}$ and $m_{\{1<2b+1\}}$, for $a, b \in \mathbb{N}$. In order to prove it, we study the four possible combinations of the parity of the letters.

Let us consider all the possible words p of a fixed length $\leq n$ such that $\Phi^{-1}(p)$ is prefix of an infinite smooth word w. We suppose that w is also an infinite Lyndon word. In the following, for each word p, either we show that $\Phi^{-1}(p)$ can not be a prefix of a Lyndon word by showing the existence of a smaller suffix, or we describe an infinite smooth Lyndon word having $\Phi^{-1}(p)$ as prefix.

Lemma 14. $p[0] = a$.

Proof. Follows from the equality $p[0] = w[0]$ and since a Lyndon word w must start by the smallest letter. □

Lemma 14 will be used in this section to exclude the cases numbered (0) in the proofs.

3.1 Over $\mathcal{A}$ with a even and b odd

In this section, we prove the following result.

Theorem 15. *Over the alphabet $\{a < b\}$, with a even and b odd, there is no infinite smooth word that is also a Lyndon word.*

Proof. Figure 1 illustrates the 5 possible cases to consider, using a tree. The leaves correspond to the first letter of $\Phi(w)$ that leads to a contradiction: the prefix $\Phi^{-1}(p)$ obtained can not be the prefix of an infinite Lyndon word. We will prove it by showing that there exists a factor f of w not prefix of $\Phi^{-1}(p)$ such that $f < w$. For clarity issues, the first letter of f is underlined.

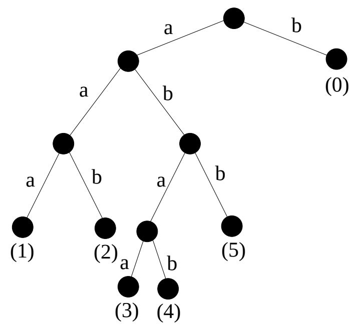


Figure 1. Possible cases for an even-odd alphabet

Case (1) If $p = aaa$, then
$\Delta^0(w) = (a^a b^a)^{\frac{a}{2}} (\underline{a}^b b^b)^{\frac{a}{2}} \cdots$
$\Delta^1(w) = a^a b^a \cdots$
$\Delta^2(w) = aa \cdots$
Since w has the prefix $a^a b^a$ and the factor $f = a^b$, it can not be a Lyndon word.

Remark 16. Since the smallest letter a of the alphabet is even, $p[3] \geq a \geq 2$. That allows us to assume that $\Delta^2(w)$ starts with a block of length at least 2. This argument holds for $\Delta^i(w)$, $i \geq 0$, and will be used for almost all cases considered in this paper.

Remark 17. In the previous case, we construct $\Delta^0(w)$ from $\Delta^2(w)$, applying Δ^{-1} twice. We will always proceed this way.

Case (2) If $p = aab$, then
$\Delta^0(w) = (a^a b^a)^{\frac{b-1}{2}} a^a (b^b \underline{a}^b)^{\frac{b-1}{2}} b^b \cdots$
$\Delta^1(w) = a^b b^b \cdots$
$\Delta^2(w) = bb \cdots$
w has the factor $f = a^b$ smaller than its prefix $a^a b^a$.

Case (3) If $p = abaa$, then
$\Delta^0(w) = ((a^b b^b)^{\frac{a}{2}} (a^a b^a)^{\frac{a}{2}})^{\frac{a}{2}} ((\underline{a}^b b^b)^{\frac{b-1}{2}} a^b (b^a a^a)^{\frac{b-1}{2}} b^a)^{\frac{a}{2}} \cdots$
$\Delta^1(w) = (b^a a^a)^{\frac{a}{2}} (b^b a^b)^{\frac{a}{2}} \cdots$
$\Delta^2(w) = a^a b^a \cdots$
$\Delta^3(w) = aa \cdots$
w has the factor $f = (a^b b^b)^{\frac{b-1}{2}} a^b$.

Case (4) If $p = abab$, then
$\Delta^0(w) = ((a^b b^b)^{\frac{a}{2}})(a^a b^a)^{\frac{a}{2}})^{\frac{b-1}{2}} (a^b b^b)^{\frac{a}{2}} ((a^a b^a)^{\frac{b-1}{2}} a^a (b^b \underline{a}^b)^{\frac{b-1}{2}} b^b)^{\frac{b-1}{2}} (a^a b^a)^{\frac{b-1}{2}} a^a \cdots$
$\Delta^1(w) = (b^a a^a)^{\frac{b-1}{2}} b^a (a^b b^b)^{\frac{b-1}{2}} a^b \cdots$
$\Delta^2(w) = a^b b^b \cdots$
$\Delta^3(w) = bb \cdots$
w has the prefix $(a^b b^b)^{\frac{a}{2}} a^a b^a$ and the smaller factor $f = (a^b b^b)^{\frac{b-1}{2}}$ contained in $(b^b a^b)^{\frac{b-1}{2}} b^b$.

Remark 18. Since $b > a$ and a is even, $b > a \geq 2$. Thus, $b \geq 3$ and $\frac{b-1}{2} \geq 1$. This insures that the factor $(b^b a^b)^{\frac{b-1}{2}} b^b$ occurs at least once.

Case (5) If $p = abb$, then
$\Delta^0(w) = (a^b b^b)^{\frac{b-1}{2}} \underline{a}^b (b^a a^a)^{\frac{b-1}{2}} b^a \cdots$
$\Delta^1(w) = b^b a^b \cdots$
$\Delta^2(w) = bb \cdots$
w has the factor $f = a^b b^a a^a$.

Using Proposition 13, we conclude. □

3.2 Over an even alphabet

Let us now consider the case of an alphabet $\mathcal{A}$ with even letters.

Theorem 19. *Over the alphabet $\{a < b\}$, with a and b even, the only smooth word that is also an infinite Lyndon word is $m_{\{a<b\}}$.*

Proof. We proceed similarly as in the previous section. The 4 possibilities are illustrated in Figure 2.

Case (1) If $p = aax$, with $x \in \mathcal{A}$, then
$\Delta^0(w) = (a^a b^a)^{\frac{x}{2}} (\underline{a}^b b^b)^{\frac{x}{2}} \cdots$
$\Delta^1(w) = a^x b^x \cdots$
$\Delta^2(w) = xx \cdots$
w has the factor $f = a^b$.

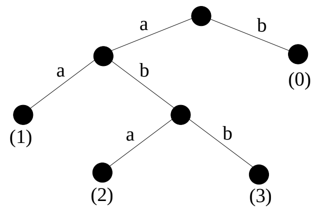


Figure 2. Possible cases for an even alphabet

Case (2) If $p = abax$, with $x \in \mathcal{A}$, then
$\Delta^0(w) = ((a^bb^b)^{\frac{a}{2}}(a^ab^a)^{\frac{a}{2}})^{\frac{x}{2}}((\underline{a}^bb^b)^{\frac{b}{2}}(a^ab^a)^{\frac{b}{2}})^{\frac{x}{2}}\cdots$
$\Delta^1(w) = (b^aa^a)^{\frac{x}{2}}(b^ba^b)^{\frac{x}{2}}\cdots$
$\Delta^2(w) = a^xb^x\cdots$
$\Delta^3(w) = xx\cdots$
w has the factor $f = (a^bb^b)^{\frac{b}{2}}$.

Case (3) Recall that the minimal smooth word $\Phi^{-1}(ab^\omega)$ is a Lyndon word. Let us show that this is the only smooth word that is also a Lyndon word. In order to prove it, let us suppose that we can write $p = ab^kay$, with $k \geq 2$ maximal (since Case (2) has already excluded the possibility $k = 1$) and $y \in \mathcal{A}^*$. Let us compute $u = \Phi^{-1}(bbax)$, with $x \in \mathcal{A}$. We get
$\Delta^0(u) = ((b^ba^b)^{\frac{a}{2}}(b^aa^a)^{\frac{a}{2}})^{\frac{x}{2}}((b^ba^b)^{\frac{b}{2}}(b^aa^a)^{\frac{b}{2}})^{\frac{x}{2}}$
$\Delta^1(u) = (b^aa^a)^{\frac{x}{2}}(b^ba^b)^{\frac{x}{2}}$
$\Delta^2(u) = a^xb^x$
$\Delta^3(u) = xx$

Since a and b are even and using Lemma 12, $\Phi^{-1}(b^kay)$ can be written as

$$(w_1^{\frac{a}{2}}w_2^{\frac{a}{2}})^{\frac{y}{2}}(w_1^{\frac{b}{2}}w_2^{\frac{b}{2}})^{\frac{y}{2}}s,$$

with $w_1 = \Delta_b^{-(k-2)}(b^ba^b)$, $w_2 = \Delta_b^{-(k-2)}(b^aa^a)$ and $s \in \mathcal{A}^*$.
Moreover, since $\Phi^{-1}(b^\omega)$ is the maximal smooth word, $\Phi^{-1}(b^k)$ (resp. $\Phi^{-1}(b^{k-1}a)$) is prefix of w_1 (resp. w_2), we have that $w_1 > w_2$ and w_1 is not prefix of w_2. Furthermore for $k \geq 2$, using Equation (1) we get

$$\Delta_b^{-1}(w_1) > \Delta_b^{-1}(w_2) \;\Leftarrow:\; \Delta_a^{-1}(w_1) < \Delta_a^{-1}(w_2),$$

that implies

$$\Delta_a^{-1}(w_1^{\frac{b}{2}}) < \Delta_a^{-1}(w_1^{\frac{a}{2}}w_2^{\frac{a}{2}}).$$

Thus $\Phi^{-1}(ab^kay)$ is not a Lyndon word.

The only Lyndon smooth word over an even 2-letter alphabet is the minimal smooth word $m_{\mathcal{A}}$ with $\Phi(m_{\mathcal{A}}) = ab^\omega$. □

3.3 Over an odd alphabet

In this section, we prove the following result.

Theorem 20. *Over the alphabet $\{a < b\}$, with a and b odd, there exists an infinite smooth Lyndon word if and only if $a = 1$. More precisely, the smooth Lyndon word is the minimal smooth word $m_{\{1<b\}}$, with $b \in 2\mathbb{N}+1$.*

Before proving Theorem 20, some results are required.

Lemma 21. *Let $\mathcal{A} = \{a < b\}$ be an odd alphabet. Let w, w' be two factors of a smooth word such that $w < w'$ and $w = xay$, $w' = xby'$, with $x, y, y' \in \mathcal{A}^*$. Then, if $|x|$ is even,*

$$\Delta_\alpha^{-1}(w) < \Delta_\alpha^{-1}(w') \iff: \overline{\alpha} < \alpha,$$

with $\alpha \in \mathcal{A}$ and $\overline{\alpha}$ its complement. If $|x|$ is odd, then

$$\Delta_\alpha^{-1}(w) < \Delta_\alpha^{-1}(w') \iff: \alpha < \overline{\alpha}.$$

Proof. Assume $|x|$ even. By direct computation, we have the following equations:

$$\Delta_\alpha^{-1}(w) = \Delta_\alpha^{-1}(xay) = \Delta_\alpha^{-1}(x)\Delta_\alpha^{-1}(a)\Delta_{\overline{\alpha}}^{-1}(y) = \Delta_\alpha^{-1}(x)\alpha^a\Delta_{\overline{\alpha}}^{-1}(y)$$
and
$$\Delta_\alpha^{-1}(w') = \Delta_\alpha^{-1}(xby') = \Delta_\alpha^{-1}(x)\Delta_\alpha^{-1}(b)\Delta_{\overline{\alpha}}^{-1}(y') = \Delta_\alpha^{-1}(x)\alpha^b\Delta_{\overline{\alpha}}^{-1}(y').$$

Then $\Delta_\alpha^{-1}(w) < \Delta_\alpha^{-1}(w')$ if and only if $\Delta_{\overline{\alpha}}^{-1}(y)[0] < \alpha$. We conclude using $\Delta_{\overline{\alpha}}^{-1}(y)[0] = \overline{\alpha}$. A similar argument holds for $|x|$ odd. □

Let us now prove 2 sub-cases of Theorem 20: $a \neq 1$ (Theorem 22) and $a = 1$ (Theorem 25).

Theorem 22. *Over the alphabet $\{a < b\}$, with a, b odd and $a \neq 1$, there is no infinite smooth word that is also a Lyndon word.*

Proof. As in Sections 3.1 and 3.2, we proceed by inspection of the different possible prefixes of $\Phi(w)$ (see Figure 3) for an infinite smooth word w.

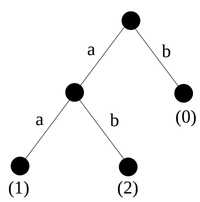


Figure 3. Possible cases for an odd alphabet, with $a \neq 1$

Case (1) If $p = aax$, then
$\Delta^0(w) = (a^ab^a)^{\frac{x-1}{2}}a^a(b^b\underline{a}^b)^{\frac{x-1}{2}}b^b(a^ab^a)^{\frac{x-1}{2}}a^a\cdots$
$\Delta^1(w) = a^xb^xa^x\cdots$
$\Delta^2(w) = xxx\cdots$
Since $x \geq a > 1$ and x is odd, $\frac{x-1}{2} \geq 1$. Thus, w has the factor $f = a^b$.

Remark 23. In the same way as in Remark 16, we can suppose that $\Delta^i(w)$ starts by a block of length at least 3.

Case (2) If $p = abx$, then
$\Delta^0(w) = (a^bb^b)^{\frac{x-1}{2}}\underline{a}^b(b^aa^a)^{\frac{x-1}{2}}b^a(a^bb^b)^{\frac{x-1}{2}}a^b\cdots$
$\Delta^1(w) = b^xa^xb^x\cdots$
$\Delta^2(w) = xxx\cdots$
w has the factor $f = a^bb^aa^a$.

Since in the 3 cases, it is possible to find a factor smaller than the prefix, we conclude that there is no smooth Lyndon word over an odd alphabet $\mathcal{A}$, with $a \neq 1$. $\square$

Proposition 24. *Let $w \in \{1 < b\}^\omega$ be an infinite smooth word that is also a Lyndon word. Then $1b \in \mathrm{Pref}(\Phi(w))$.*

We know from [5] that the minimal smooth word over an odd 2-letter alphabet $\{1 < b\}$ is a Lyndon word. The next theorem shows that this is the only infinite Lyndon word over the alphabet $\{1 < b\}$.

Theorem 25. *Over the alphabet $\{1 < b\}$, with b odd, the only infinite smooth word that is also a Lyndon word is $m_{\{1<b\}}$.*

Proof. Recall that $\Phi(m_{\{1,b\}}) = (1b)^\omega$. Let us show that for an infinite smooth word w, any prefix p of $\Phi(w)$ that is not prefix of $(1b)^\omega$ can not be such that $\Phi^{-1}(p)$ is prefix of an infinite Lyndon word. By Proposition 24, p starts by $1b$. We proceed by inspection of the different possibilities (see Figure 4).

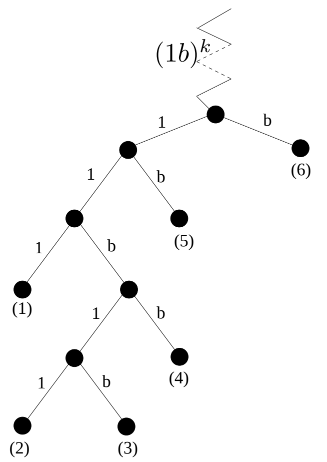


Figure 4. Possible prefixes of p starting by $(1b)^k$, $k \geq 1$

Case (1): $p = (1b)^k 111$. Let us consider the prefix $\Phi^{-1}(1b111)$ of u:

$\Delta^0(u) = 1^b((b1)^{\frac{b-1}{2}} b(1^b b^b)^{\frac{b-1}{2}} \underline{1}^b)^{\frac{b-1}{2}} (b1)^{\frac{b-1}{2}} b1^b b1 \cdots$

$\Delta^1(u) = b(1^b b^b)^{\frac{b-1}{2}} 1^b b1 \cdots$

$\Delta^2(u) = 1b^b 1 \cdots$

$\Delta^3(u) = 1b \cdots$

$\Delta^4(u) = 1 \cdots$

u has the factor $f = 1^b(b1)^{\frac{b-1}{2}} b1^b b1$. Using Lemma 21 $2(k-1)$ times, we conclude that the prefix p does not describe a smooth Lyndon word. In Cases (2), (3), (4) and (6), we get the same conclusion with a similar argument. Thus the only prefix leading to a smooth Lyndon word is the one considered in Case (5): $(1b)^\omega$. $\square$

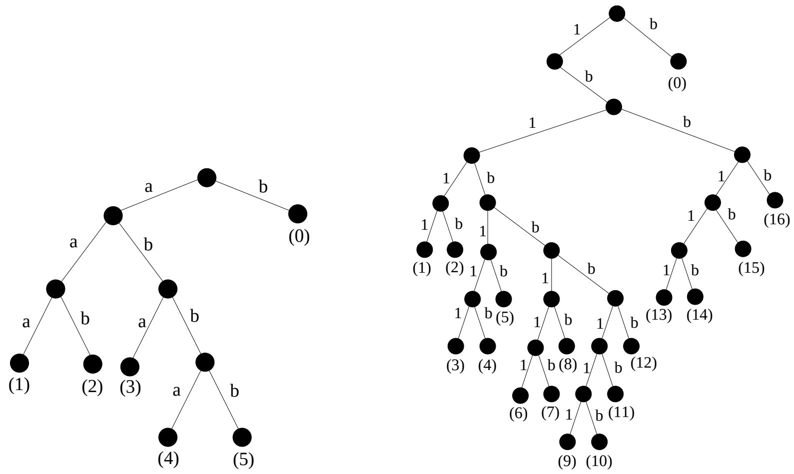
Proof. (of Theorem 20) Follows from Theorems 22 and 25. $\square$

3.4 Over $\mathcal{A}$ with a odd and b even

In this section, we consider infinite smooth words over an alphabet $\{a < b\}$, with a odd and b even. We prove that over this alphabet, there is no infinite smooth word that is also a Lyndon word. In order to prove it, we consider 2 cases, $a \neq 1$ and $a = 1$, that have to be analysed separately.

Theorem 26. *Over the alphabet $\{a < b\}$, with $a \neq 1$ odd and b even, there is no smooth infinite word that is a Lyndon word.*

Proof. There are 5 possibilities to consider, illustrated in Figure 5 (a).



(a) for an odd-even alphabet $\{a < b\}$, with $a \neq 1$.

(b) for the alphabet $\{a < b\}$, with $a = 1$ and $b = 4n$.

Figure 5. Possible cases...

In each case, it is possible to find a factor f smaller than the smooth word. Thus, there is no smooth Lyndon word. □

Theorem 27. *Over the alphabet $\{a < b\}$, with $a = 1$ and $b = 4n$, there is no infinite smooth word that is a Lyndon word.*

Proof. Figure 5 (b) shows the different cases to consider. For each of the 16 cases, it is again possible to find a factor of $\Phi^{-1}(p)$ in order to prove that it is not a prefix of an infinite Lyndon word. □

Theorem 28. *Over the alphabet $\{a < b\}$, with $a = 1$ and $b = 2(2n + 1)$, there is no infinite smooth word that is a Lyndon word.*

Proposition 24 can be generalized to an alphabet $\{1, b\}$, with b even. This result will be used in the following proof.

Proof. Figure 6 shows the different cases to consider. Cases numbered less or equal to 16 are the same as in Theorem 27. For the other cases, it is possible to find a factor in $\Phi^{-1}(p)$ smaller than its prefix, following that the word is not a Lyndon word. □

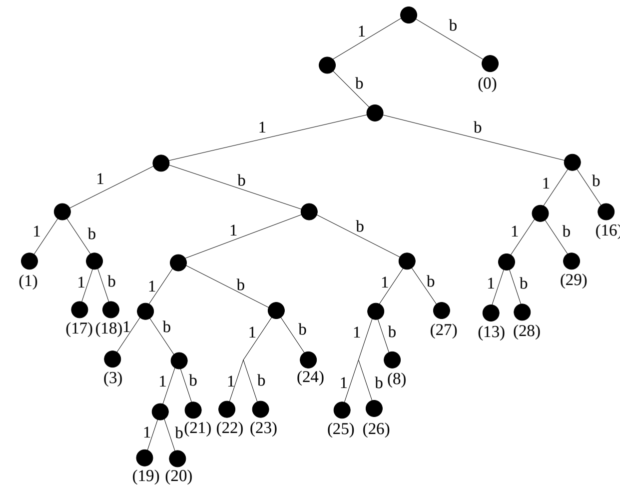


Figure 6. Different cases for the alphabet $\{a < b\}$, with $a = 1$ and $b = 2(2n + 1)$

4 Summary and concluding remarks

The next theorem summarizes the results of Section 3.

Theorem 29. *Over any* 2*-letter alphabet, the only infinite smooth words that are also infinite Lyndon words are* $m_{\{2a<2b\}}$ *and* $m_{\{1<2b+1\}}$*, for* $a, b \in \mathbb{N} \setminus \{0\}$*.*

Recall that for the alphabet $\{1, 2\}$, it is conjectured [4] that in any infinite smooth word, any smooth factor appears. From this conjecture follows that no infinite smooth word is a Lyndon word. This is exactly what we have proved for the alphabet $\{1, 2\}$. Moreover, the existence of infinite smooth Lyndon words over the alphabets $\{2a < 2b\}$ and $\{1 < 2b + 1\}$ leads to the following corollary.

Corollary 30. *Let* $\mathcal{A}$ *be a* 2*-letter alphabet such that* $\mathcal{A} = \{2a < 2b\}$ *or* $\mathcal{A} = \{1 < 2b + 1\}$*. Then, any infinite smooth words* $w \in \mathcal{A}^\omega$ *does not contain every smooth factors.*

Otherwise, no infinite Lyndon word would exist: a factor smaller than the prefix necessarily occurs. It is also interesting to notice that our main result completely characterized the trivial finite Lyndon factorization of infinite smooth words: the only infinite smooth words that have a finite Lyndon factorization composed of only one factor are $m_{\{2a<2b\}}$ and $m_{\{1<2b+1\}}$. It is still an open problem to characterized infinite smooth words that have a non trivial finite Lyndon factorization. Giving an explicit computation of the Lyndon factorization, finite or infinite, of any infinite smooth words, as Melançon did for standard Sturmian words [19] is still a challenging problem.

Acknowledgements. The author would like to thank Pierre Lalonde for his interesting question during the LaCIM seminar that leads to this paper and Srecko Brlek for his comments.

References

1. A. Bergeron-Brlek, S. Brlek, A. Lacasse, and X. Provençal: *Patterns in smooth tilings*, in Proceedings of WORDS'03, vol. 27 of TUCS Gen. Publ., Turku Cent. Comput. Sci., Turku, 2003, pp. 370–381.

2. J. Berstel and A. de Luca: *Sturmian words, Lyndon words and trees.* Theoret. Comput. Sci., 178(1-2) 1997, pp. 171–203.
3. V. Berthé, S. Brlek, and P. Choquette: *Smooth words over arbitrary alphabets.* Theoret. Comput. Sci., 341 2005, pp. 293–310.
4. S. Brlek, S. Dulucq, A. Ladouceur, and L. Vuillon: *Combinatorial properties of smooth infinite words.* Theoret. Comput. Sci., 352 2006, pp. 306–317.
5. S. Brlek, D. Jamet, and G. Paquin: *Smooth words on 2-letter alphabets having same parity.* Theoret. Comput. Sci., 393 2008, pp. 166–181.
6. S. Brlek and A. Ladouceur: *A note on differentiable palindromes.* Theoret. Comput. Sci., 302 2003, pp. 167–178.
7. S. Brlek, G. Melançon, and G. Paquin: *Properties of the extremal infinite smooth words.* Discrete Math. Theoret. Comput. Sci., 9(2) 2007, pp. 33–49 (electronic).
8. A. Carpi: *On repeated factors in C^∞-words.* Information Processing Letters, 52 1994, pp. 289–294.
9. K. T. Chen, R. H. Fox, and R. C. Lyndon: *Free differential calculus, IV. The quotient groups of the lower central series.* Ann. of Math., 68 1958, pp. 81–95.
10. F. M. Dekking: *On the structure of self generating sequences.* Séminaire de théorie des nombres de Bordeaux, exposé 31, 1980–1981.
11. J.-P. Duval: *Factorizing words over an ordered aphabet.* Journal of Algorithms, 4 1983, pp. 363–381.
12. J.-P. Duval: *Génération d'une section des classes de conjugaison et arbre de mots de Lyndon de longueur bornée.* Theoret. Comput. Sci., 60 1988, pp. 255–283.
13. C. Hohlweg and C. Reutenauer: *Lyndon words, permutations and trees.* Theoret. Comput. Sci., 307 2003, pp. 173–178.
14. Y. B. Huang: *On the condition of powers of C^∞-word to be a C^∞-word.* Applied Mathematics - A Journal of Chinese Universities (Ser. A), 2 1997, pp. 243–246.
15. Y. B. Huang: *On the number of C^∞-words of form $\tilde{w}xw$.* Theoret. Comput. Sci., 393 2008, pp. 280–286.
16. D. Jamet and G. Paquin: *Discrete surfaces and infinite smooth words*, in FPSAC'05 - 17th annual International conference on Formal Power Series and Algebraic Combinatorics (Taormina, Italie) June 20–25, 2005.
17. W. Kolakoski: *Self generating runs, Problem 5304.* Amer. Math. Monthly, 72 1965, p. 674.
18. M. Lothaire: *Combinatorics on words*, Addison Wesley, Reading MA, 1983.
19. G. Melançon: *Lyndon factorization of Sturmian words.* Discrete Appl. Math., 210 2000, pp. 137–149.
20. G. Melançon and C. Reutenauer: *Lyndon words, free algebras and shuffles.* Canad. J. Math., 4 1989, pp. 577–591.
21. G. Paquin: *Mots équilibrés et mots lisses*, PhD thesis, Université du Québec à Montréal, 2008.
22. C. Reutenauer: *Free Lie algebras*, vol. 7, London Math. Soc. Monographs New Ser., 1993.
23. C. Reutenauer: *Mots de Lyndon généralisés.* Sém. Lothar. Combin., 54 2005/07, pp. Art. B54h, 16 pp. (electronic).
24. G. Richomme: *Conjugacy of morphisms and Lyndon decomposition of standard Sturmian words.* Theoret. Comput. Sci., 380(3) 2007, pp. 393–400.
25. P. Séébold: *Lyndon factorization of the Prouhet words.* Theoret. Comput. Sci., 307 2003, pp. 179–197.
26. R. Siromoney, L. Matthew, V. R. Dare, and K. G. Subramanian: *Infinite lyndon words.* Information Processing Letters, 50 1994, pp. 101–104.
27. W. D. Weakley: *On the number of C^∞-words of each length.* J. Combin. Theory Ser. A, 51(1) 1989, pp. 55–62.

New Lower Bounds for the Maximum Number of Runs in a String

Wataru Matsubara[1], Kazuhiko Kusano[1], Akira Ishino[1], Hideo Bannai[2], and Ayumi Shinohara[1]

[1] Graduate School of Information Science, Tohoku University,
Aramaki aza Aoba 6-6-05, Aoba-ku, Sendai 980-8579, Japan
{matsubara@shino., kusano@shino., ishino@, ayumi@ }ecei.tohoku.ac.jp
[2] Department of Informatics, Kyushu University,
744 Motooka, Nishiku, Fukuoka 819-0395 Japan.
bannai@i.kyushu-u.ac.jp

Abstract. We show a new lower bound for the maximum number of runs in a string. We prove that for any $\varepsilon > 0$, $(\alpha - \varepsilon)n$ is an asymptotic lower bound, where $\alpha = 174719/184973 \approx 0.944565$. It is superior to the previous bound $3/(1+\sqrt{5}) \approx 0.927$ given by Franěk *et al.* [6,7]. Moreover, our construction of the strings and the proof is much simpler than theirs.

1 Introduction

Repetitions in strings is an important element in the analysis and processing of strings. It was shown in [9] that when considering *maximal repetitions*, or *runs*, the maximum number of runs $\rho(n)$ in any string of length n is $O(n)$, leading to a linear time algorithm for computing all the runs in a string. Although they were not able to give bounds for the constant factor, there have been several works to this end [12,13,11,2,1,8]. The currently known best upper bound[3] is $\rho(n) \leq 1.048n$ [3], obtained by calculations based on the proof technique of [2]. The technique bounds the number of runs for each string by considering runs in two parts: runs with long periods, and runs with short periods. The former is more sparse and easier to bound while the latter is bounded by an exhaustive calculation concerning how runs of different periods can overlap in an interval of some length. On the other hand, an asymptotic lower bound on $\rho(n)$ is presented in [7], where it is shown that for any $\varepsilon > 0$, there exists an integer $N > 0$ such that for any $n > N$, $\rho(n) \geq (\alpha - \varepsilon)n$, where $\alpha = \frac{3}{1+\sqrt{5}} \approx 0.927$. It was conjectured in [6] that this bound is optimal.

In this paper, we prove that the conjecture was false, by showing a new lower bound $\alpha = 174719/184973 \approx 0.944565$. First we show a concrete string τ of length 184973, which contains 174697 runs in it. It immediately disproves the conjecture, since $174697/184973 \approx 0.944445$ is already higher than the previous bound 0.927. Then we prove that the string τ^k, which is the string obtained by concatenating k copies of τ, contains $174719k - 21$ runs for any $k \geq 2$. Since $|\tau^k| = 184973k$, it yields the new lower bound 174719/184973 as $k \to \infty$.

2 Preliminaries

Let Σ be a finite set of symbols, called an *alphabet*. Strings x, y and z are said to be a *prefix*, *substring*, and *suffix* of the string $w = xyz$, respectively. The length of

[3] Presented on the website http://www.csd.uwo.ca/faculty/ilie/runs.html

Wataru Matsubara, Kazuhiko Kusano, Akira Ishino, Hideo Bannai, Ayumi Shinohara : New Lower Bounds for the Maximum Number of Runs in a String, pp. 140–145.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

a string w is denoted by $|w|$. The i-th symbol of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of w that begins at position i and ends at position j is denoted by $w[i:j]$ for $1 \leq i \leq j \leq |w|$. A string w has period p if $w[i] = w[i+p]$ for $1 \leq i \leq |w| - p$. A string w is called *primitive* if w cannot be written as u^k, where k is a positive integer, $k \geq 2$.

A string u is a *run* if it is periodic with (minimum) period $p \leq |u|/2$. A substring $u = w[i:j]$ of w is a *run in* w if it is a run of period p and neither $w[i-1:j]$ nor $w[i:j+1]$ is a run of period p, that means the run is maximal. We denote the run $u = w[i:j]$ in w by the triple $\langle i, j-i+1, p\rangle$ consisting of the begin position i, the length $|u|$, and the minimum period p of u. A run of w which is a prefix (resp. suffix) of w is called a prefix (resp. suffix) run of w, For a string w, we denote by *run*(w) the number of runs in w.

For example, the string `aabaabaaaacaacac` contains the following 7 runs: $\langle 1,2,1\rangle = \mathtt{a}^2$, $\langle 4,2,1\rangle = \mathtt{a}^2$, $\langle 7,4,1\rangle = \mathtt{a}^4$, $\langle 12,2,1\rangle = \mathtt{a}^2$, $\langle 13,4,2\rangle = (\mathtt{ac})^2$, $\langle 1,8,3\rangle = (\mathtt{aab})^{\frac{8}{3}}$, and $\langle 9,7,3\rangle = (\mathtt{aac})^{\frac{7}{3}}$. Thus *run*(`aabaabaaaacaacac`) = 7.

We are interested in the behavior of the *maxrun function* defined by

$$\rho(n) = \mathbf{max}\{run(w) \mid w \text{ is a string of length } n\}.$$

Franěk, Simpson and Smyth [6] showed a beautiful construction of a series of strings which contains many runs, and later Franěk and Qian Yang [7] formally proved a family of true asymptotic lower bounds arbitrarily close to $\frac{3}{1+\sqrt{5}}n$ as follows.

Theorem 1 ([7]). *For any $\varepsilon > 0$ there exists a positive integer N so that $\rho(n) \geq \left(\frac{3}{1+\sqrt{5}} - \varepsilon\right) n$ for any $n \geq N$.*

3 Basic Properties

In this section, we summarize some basic properties concerning periods and repetitions in strings, which will be utilized in the sequel.

The next Lemma given by Fine and Wilf [5] provides an important property on periods of a string.

Lemma 2 (Periodicity Lemma (see [10,4])). *Let p and q be two periods of a string w. If $p + q - \gcd(p,q) \leq |w|$, then $\gcd(p,q)$ is also a period of w.*

For a string w, let us consider a series of strings w, w^2, w^3, $w^4 \ldots$, and observe all runs contained in these strings. There are many cases, which confuse the task of counting the number of runs in these strings.

1. A run in w^k which is neither a suffix nor prefix run of w^k is also a run in w^{k+1}.
2. A suffix run in w^k and a prefix run in w may be merged into one run in w^{k+1}.
3. A suffix run in w^k may be extended to a run in w^{k+1}.
4. A new run may be newly created at the border between w^{k+1} and w.

Concerning case 4, note that a new run that did not appear in w or w^2 may be created in w^3. For example, consider strings $w =$ `abcacabc`, and $r = (\mathtt{cabca})^2$. We can verify that r is a run $\langle 8,10,5\rangle$ of $w^3 = \mathtt{abcacab}\underline{\mathtt{cabcacabca}}\mathtt{bcacabc}$, while r does not appear in $w^2 =$ `abcacabcabcacabc`. Moreover, the same argument holds also for binary alphabet `0,1`; Replace `a`, `b`, `c` into `01`, `10`, `00`, respectively in the above example.

However, the following lemma shows that the length of such new runs can be bounded.

Lemma 3. *Let w be a string of length n. For any $k \geq 3$, let $r = \langle i, l, p \rangle$ be a run in w^k. If $l \geq 2n$, then $i = 1$ and $l = kn$, that is, $r = w^k$.*

Proof. We assume that $n > 1$, since it is trivial for the case $n = 1$. Since p is the minimum period of the run r, we know $|r| = l \geq 2p$ and $l \geq 2n$. Let u be a primitive string of length m where $w = u^t$ for some integer $t \geq 1$. Then, $|u| = m \leq n$ is also a period of run r. Since $p + m \leq l$, Lemma 2 claims that $\gcd(p, m)$ is also a period of run r. If $p > m$, then $\gcd(p, m) < p$, which contradicts the assumption that p is the minimum period of r. If $p < m$, then it contradicts the assumption that u is primitive. Therefore we have $p = m$. Since m is a period of w^k, we have $r = \langle 1, kn, m \rangle = w^k$.

This lets us prove the following lemma which gives a formula for $run(w^k)$.

Lemma 4. *Let w be a string of length n. For any $k \geq 2$, $run(w^k) = Ak - B$, where $A = run(w^3) - run(w^2)$ and $B = 2run(w^3) - 3run(w^2)$.*

Proof. We think about the increase in the number of runs, when concatenating w^k and w. Let $r = \langle i, l, p \rangle$ be a run of w^{k+1} such that $i + l > nk + 1$, that is, r ends somewhere in the last w of w^{k+1}. By Lemma 3, if $i \leq (k-2)n$ then $r = w^{k+1}$. In such a case, r does not increase the number of runs since the run will have already been considered in w^2. Therefore, the increase in runs can be considered by restricting our attention to runs with $i > (k-2)n$, that is, the increase in runs for the last 3 w's of w^{k+1} when concatenating w to the last 2 w's of w^k. This gives us $run(w^{k+1}) - run(w^k) = run(w^3) - run(w^2)$.

$$\begin{aligned} run(w^k) &= run(w^{k-1}) + run(w^3) - run(w^2) \\ &= run(w^{k-2}) + 2(run(w^3) - run(w^2)) \\ &= run(w^2) + (k-2)(run(w^3) - run(w^2)) \\ &= k(run(w^3) - run(w^2)) - (2run(w^3) - 3run(w^2)) \end{aligned}$$

for $k \geq 3$. It is easy to see that the equation also holds for $k = 2$.

Theorem 5. *For any string w and any $\varepsilon > 0$, there exists a positive integer N such that for any $n \geq N$,*

$$\frac{\rho(n)}{n} > \frac{run(w^3) - run(w^2)}{|w|} - \varepsilon.$$

Proof. By Lemma 4, $run(w^k) = Ak - B$, where $A = run(w^3) - run(w^2)$ and $B = 2run(w^3) - 3run(w^2)$.

For any given $\varepsilon > 0$, we choose $N > \frac{A-B}{\varepsilon}$. For any $n \geq N$, let k be the integer satisfying $|w|(k-1) \leq n < |w|k$. Notice that $k > \frac{n}{|w|} \geq \frac{N}{|w|} \geq \frac{A-B}{|w|\varepsilon}$. Since $\rho(i+1) \geq \rho(i)$ for any i, and $|w^{k-1}| = |w|(k-1)$,

$$\begin{aligned} \frac{\rho(n)}{n} &\geq \frac{\rho(|w|(k-1))}{|w|k} \geq \frac{run(w^{k-1})}{|w|k} = \frac{A(k-1) - B}{|w|k} = \frac{Ak - A - B}{|w|k} \\ &= \frac{A}{|w|} - \frac{A - B}{|w|k} > \frac{A}{|w|} - \varepsilon. \end{aligned}$$

□

4 New Lower Bounds

We found some strings which contain many runs, by running a computer program which utilizes a simple heuristic search for run-rich binary strings. Given a buffer size, the search first starts with the single string `0` in the buffer. At each round, two new strings are created from each string in the buffer by appending `0` or `1` to the string. The new strings are then sorted in order of $run(w^3) - run(w^2)$, and only those that fit in the buffer are retained for the next round. Strings that give a high ratio of runs are recorded.

We tried several variations of the algorithm, and found many run-rich strings. Among these strings found so far, the string τ, lets us prove the currently best lower bound on the maximum number of runs in a string. Since τ is too long to include in the paper, we will make τ available on our web site [4]. Once we have τ, it is straightforward to confirm that the following lemma holds. Any naïve program to count runs in a string would be sufficient.

Lemma 6. *There exists a string τ such that $|\tau| = 184973$, $run(\tau) = 174697$, $run(\tau^2) = 349417$, and $run(\tau^3) = 524136$.*

It immediately disproves the conjecture, since $174697/184973 \approx 0.944445$ is already higher than the previous bound $\frac{3}{1+\sqrt{5}} \approx 0.927$. We now show the main result of this paper.

Theorem 7. *For any $\varepsilon > 0$ there exists a positive integer N so that $\rho(n) > (\alpha - \varepsilon)\, n$ for any $n \geq N$, where $\alpha = \frac{174719}{184973} \approx 0.944565$.*

Proof. From Theorem 5 and Lemma 6, we have

$$\frac{\rho(n)}{n} > \frac{524136 - 349417}{184973} - \varepsilon = \frac{174719}{184973} - \varepsilon.$$

$\square$

For proof of concept, we present in the Appendix, a shorter string τ_{1558} with $|\tau_{1558}| = 1558, run(\tau_{1558}) = 1455, run(\tau_{1558}^2) = 2915, run(\tau_{1558}^3) = 4374$ that gives a smaller bound $(4374 - 2915)/1558 \approx 0.93645$ compared to τ, but is still better than previously known.

5 Conclusion and Further Research

We presented a new lower bound $174719/184973 \approx 0.944565$ for the maximum number of runs in a string. The proof was very simple, once after we verified that the runs in the string τ is 174697, and noticed some trivial properties of the string. We do not think that the bound is optimal. We believe that our work would revive the interests to push the lower bound higher up, since the previous bound $3/(1+\sqrt{5}) \approx 0.927$ was conjectured to be the optimal since 2003.

Further research will include trying to find properties of run-rich strings by analyzing strings obtaining from heuristic search. We believe that compression gives a clue to understanding the property of run-rich strings, since while τ has length 184973, it can be represent by mere 24 terms of LZ factors (see Appendix).

[4] `http://www.shino.ecei.tohoku.ac.jp/runs/`

Acknowledgements: We thank Professor Rytter for useful comments about the compressibility of run-rich strings.

References

1. P. Baturo, M. Piątkowski, and W. Rytter: *The number of runs in Sturmian words*, in Proc. CIAA 2008, 2008, To appear.
2. M. Crochemore and L. Ilie: *Maximal repetitions in strings*. J. Comput. Syst. Sci., 74 2008, pp. 796–807.
3. M. Crochemore, L. Ilie, and L. Tinta: *Towards a solution to the "runs" conjecture*, in Proc. CPM 2008, vol. 5029 of LNCS, 2008, pp. 290–302.
4. M. Crochemore and W. Rytter: *Jewels of Stringology*, World Scientific, 2002.
5. N. Fine and H. Wilf: *Uniqueness Theorems for Periodic Functions*. Proceedings of the American Mathematical Society, 16(1) 1965, pp. 109–114.
6. F. Franěk, R. Simpson, and W. Smyth: *The maximum number of runs in a string*, in Proc. 14th Australasian Workshop on Combinatorial Algorithms (AWOCA2003), 2003, pp. 26–35.
7. F. Franěk and Q. Yang: *An asymptotic lower bound for the maximal-number-of-runs function*, in Proc. Prague Stringology Conference (PSC'06), 2006, pp. 3–8.
8. M. Giraud: *Not so many runs in strings*, in Proc. LATA 2008, 2008, pp. 245–252.
9. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in Proc. 40th Annual Symposium on Foundations of Computer Science (FOCS'99), 1999, pp. 596–604.
10. M. Lothaire: *Algebraic combinatorics on words*, Cambridge University Press New York, 2002.
11. S. J. Puglisi, J. simpson, and W. F. Smyth: *How many runs can a string contain?* Theoretical Computer Science, 401(1–3) 2008, pp. 165–171.
12. W. Rytter: *The number of runs in a string: Improved analysis of the linear upper bound*, in Proc. 23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS 2006), vol. 3884 of LNCS, 2006, pp. 184–195.
13. W. Rytter: *The number of runs in a string*. Inf. Comput., 205(9) 2007, pp. 1459–1469.

Appendix

The binary string τ_{1558} with $|\tau_{1558}| = 1558$, $run(\tau_{1558}) = 1455$, $run(\tau^2_{1558}) = 2915$, $run(\tau^3_{1558}) = 4374$, giving lower bound $(4374 - 2915)/1558 \approx 0.93645 > 0.927$.

```
110101101001011010110100101101011001101011010010110101101001011010
```

```
110101101001011010110100101101011001101011010010110101101001011010
110010110101101001011010110100101101011001101011010010110101101001
011010110010110101101001011010110010110100101101011010010110101100
101101011010010110101101001011010110010110100101101011010010110101
100101101011010010110101100101101001011010110100101101011001011010
110100101101011010010110101100101101011010010110101100101101001011
010110100101101011001011010110100101101011010010110101100101101001
011010110100101101011001011010110100101101011001011010010110101101
001011010110010110101101001011010110100101101011001011010110100101
101011001011010010110101101001011010110010110101101001011010110010
110100101101011010010110101100101101011010010110101101001011010110
010110100101101011010010110101100101101011010010110101100101101001
011010110100101101011001011010110100101101011010010110101100101101
011010010110101100101101001011010110100101101011001011010110100101
101011010010110101100101101001011010110100101101011001011010110100
101101011001011010010110101101001011010110010110101101001011010110
100101101011001011010110100101101011001011010010110101101001011010
110010110101101001011010110010110100101101011010010110101100101101
011010010110101101001011010110010110100101101011010010110101100101
101011010010110101100101101001011010110100101101011001011010110100
101101011010010110101100101101011010010110101100101101001011010110
100101101011001011010110100101101011010010110101100101101001011010
110100101101011001011010110100101101011001011010010110101101001011
010110010110101101001011010110100101101011010
```

By interpreting τ_{1558} as a binary representation of an integer, it can be expressed in hexadecimal representation by:

```
0x35A5AD2D66B4B5A5ACB5A5AD2D66B4B5A5ACB5A5ACB4B5A5ACB5A5AD2D65A5AD
2D65AD2D65A5AD2D65AD2D696B2D696B2D2D696B2D696B4B59696B4B596B4B5969
6B4B596B4B5A5ACB5A5ACB4B5A5ACB5A5ACB4B5A5ACB5A5AD2D65A5AD2D65AD2D6
5A5AD2D65AD2D696B2D696B2D2D696B2D696B4B59696B4B596B4B59696B4B596B4
B5A5ACB5A5ACB4B5A5ACB5A5ACB4B5A5ACB5A5AD2D65A5AD2D65AD2D65A5AD2D65
AD2D696B2D696B2D2D696B2D696B4B59696B4B596B4B59696B4B596B4B5A5A
```

The string τ of Lemma 6 can be represented by 24 terms of LZ factors.

```
τ = a / (0,1) / b / (1,3) / (1,4) / (2,8) / (5,13) / (12,19) /
(26,31) / (49,38) / (50,63) / (89,93) / (113,162) / (57,317) /
(249,693) / (275,984) / (879,2120) / (942,3041) / (2811,6521) /
(2999,9374) / (8764,20072) / (9332,28878) / (27096,45341) /
(38210,67195)
```

Efficient Variants of the Backward-Oracle-Matching Algorithm

Simone Faro[1] and Thierry Lecroq[2]

[1]Dipartimento di Matematica e Informatica, Università di Catania, Viale A.Doria n.6, 95125 Catania, Italy
[2]Université de Rouen, LITIS EA 4108, 76821 Mont-Saint-Aignan Cedex, France
faro@dmi.unict.it, thierry.lecroq@univ-rouen.fr

Abstract. In this article we present two efficient variants of the BOM string matching algorithm which are more efficient and flexible than the original algorithm. We also present bit-parallel versions of them obtaining an efficient variant of the BNDM algorithm. Then we compare the newly presented algorithms with some of the most recent and effective string matching algorithms. It turns out that the new proposed variants are very flexible and achieve very good results, especially in the case of large alphabets.

Keywords: string matching, experimental algorithms, text processing, automaton

1 Introduction

Given a text t of length n and a pattern p of length m over some alphabet Σ of size σ, the *string matching problem* consists in finding *all* occurrences of the pattern p in the text t. It is an extensively studied problem in computer science, mainly due to its direct applications to such diverse areas as text, image and signal processing, speech analysis and recognition, information retrieval, computational biology and chemistry, etc.

Many string matching algorithms have been proposed over the years (see [8]). The Boyer-Moore algorithm [5] deserves a special mention, since it has been particularly successful and has inspired much work.

Automata play a very important role in the design of efficient pattern matching algorithms. For instance the well known Knuth-Morris-Pratt algorithm [14] uses a deterministic automaton that searches a pattern in a text by performing its transitions on the text characters. The main result relative to the Knuth-Morris-Pratt algorithm is that its automaton can be constructed in $\mathcal{O}(m)$-time and -space, whereas pattern search takes $\mathcal{O}(n)$-time.

Automata based solutions have been also developed to design algorithms which have optimal sublinear performance on average. This is done by using factor automata [4,9,3,1], data structures which identify all factors of a word. Among the algorithms which make use of a factor automaton the BOM (Backward Oracle Matching) algorithm [1] is the most efficient, especially for long patterns. Another algorithm based on the bit-parallel simulation [2] of the nondeterministic factor automaton, and called BNDM (Backward Nondeterministic Dawg Match) algorithm [16], is very efficient for short patterns.

In this article we present two efficient variations of the BOM string matching algorithm which turn out to be more efficient and flexible than the original BOM algorithm. We also present a bit-parallel version of the previous solution which efficiently extends the BNDM algorithm.

Simone Faro, Thierry Lecroq: Efficient Variants of the Backward-Oracle-Matching Algorithm, pp. 146–160.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

The article is organized as follows. In Section 2 we introduce basic definitions and the terminology used along the paper. In Section 3 we survey some of the most effective string matching algorithms. Next, in Section 4, we introduce two new variations of the BOM algorithm. Experimental data obtained by running under various conditions all the algorithms reviewed are presented and compared in Section 5. Finally, we draw our conclusions in Section 6.

2 Basic Definitions and Terminology

A string p of length m is represented as a finite array $p[0 \mathinner{..} m-1]$, with $m \geq 0$. In particular, for $m = 0$ we obtain the empty string, also denoted by ε. By $p[i]$ we denote the $(i+1)$-st character of p, for $0 \leq i < m$. Likewise, by $p[i \mathinner{..} j]$ we denote the substring of p contained between the $(i+1)$-st and the $(j+1)$-st characters of p, for $0 \leq i \leq j < m$. Moreover, for any $i, j \in \mathbb{Z}$, we put $p[i \mathinner{..} j] = \varepsilon$ if $i > j$ and $p[i \mathinner{..} j] = p[\mathbf{max}(i, 0), \mathbf{min}(j, m-1)]$ if $i \leq j$. A substring of the form $p[0 \mathinner{..} i]$ is called a *prefix* of p and a substring of the form $p[i \mathinner{..} m-1]$ is called a *suffix* of p for $0 \leq i \leq m-1$. For any two strings u and w, we write $w \sqsupset u$ to indicate that w is a suffix of u. Similarly, we write $w \sqsubset u$ to indicate that w is a prefix of u. The *reverse* of a string $p[0 \mathinner{..} m-1]$ is the string built by the concatenation of its letters from the last to the first: $p[m-1]p[m-2] \cdots p[1]p[0]$.

A Finite State Automaton is a tuple $A = \{Q, q_0, F, \Sigma, \delta\}$, where Q is the set of states of the automaton, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states, Σ is the alphabet of characters labeling transitions and $\delta() : (Q \times \Sigma) \rightarrow Q$ is the transition function. If $\delta(q, c)$ is not defined for a state $q \in Q$ and a character $c \in \Sigma$ we say that $\delta(q, c)$ is an undefined transition and write $\delta(q, c) = \perp$.

Let t be a text of length n and let p be a pattern of length m. When the character $p[0]$ is aligned with the character $t[s]$ of the text, so that the character $p[i]$ is aligned with the character $t[s+i]$, for $i = 0, \ldots, m-1$, we say that the pattern p has *shift* s in t. In this case the substring $t[s \mathinner{..} s+m-1]$ is called the *current window* of the text. If $t[s \mathinner{..} s+m-1] = p$, we say that the shift s is *valid*.

Most string matching algorithms have the following general structure. First, during a *preprocessing phase*, they calculate useful mappings, in the form of tables, which later are accessed to determine nontrivial shift advancements. Next, starting with shift $s = 0$, they look for all valid shifts, by executing a *matching phase*, which determines whether the shift s is valid and computes a *positive* shift increment.

For instance, in the case of the naive string matching algorithm, there is no preprocessing phase and the matching phase always returns a unitary shift increment, i.e. all possible shifts are actually processed.

In contrast the Boyer-Moore algorithm [5] checks whether s is a valid shift, by scanning the pattern p from right to left and, at the end of the matching phase, it computes the shift increment as the maximum value suggested by two heuristics: the *good-suffix heuristic* and the *bad-character heuristic*, provided that both of them are applicable (see [8]).

3 Very Fast String Matching Algorithms

In this section we briefly review the BOM algorithm and other efficient algorithms for exact string matching that have been recently proposed. In particular, we present algorithms in the Fast-Search family [7], algorithms in the q-Hash family [15] and some among the most efficient algorithms based on factor automata.

3.1 Fast-Search and Forward-Fast-Search Algorithms

The Fast-Search algorithm [6] is a very simple, yet efficient, variant of the Boyer-Moore algorithm. Let p be a pattern of length m and let t be a text of length n over a finite alphabet Σ. The Fast-Search algorithm computes its shift increments by applying the bad-character rule if and only if a mismatch occurs during the first character comparison, namely, while comparing characters $p[m-1]$ and $t[s+m-1]$, where s is the current shift. Otherwise it uses the good-suffix rule.

The Forward-Fast-Search algorithm [7] maintains the same structure of the Fast-Search algorithm, but it is based upon a modified version of the good-suffix rule, called *forward good-suffix* rule, which uses a look-ahead character to determine larger shift advancements. Thus, if the first mismatch occurs at position $i < m-1$ of the pattern p, the forward good-suffix rule suggests to align the substring $t[s+i+1\,..\,s+m]$ with its rightmost occurrence in p preceded by a character different from $p[i]$. If such an occurrence does not exist, the forward good-suffix rule proposes a shift increment which allows to match the longest suffix of $t[s+i+1\,..\,s+m]$ with a prefix of p. This corresponds to advance the shift s by $\overrightarrow{gs}_P(i+1, t[s+m])$ positions, where

$$\begin{aligned}\overrightarrow{gs}_P(j,c) =_{\text{Def}} \mathbf{min}(\{0 < k \le m \mid\ & p[j-k\,..\,m-k-1] \sqsupset p \\ & \text{and } (k \le j-1 \rightarrow p[j-1] \neq p[j-1-k]) \\ & \text{and } p[m-k] = c\} \ \cup\ \{m+1\})\ ,\end{aligned}$$

for $j = 0, 1, \ldots, m$ and $c \in \Sigma$.

The good-suffix rule and the forward good-suffix rule require tables of size m and $m \cdot |\Sigma|$, respectively. These can be constructed in time $\mathcal{O}(m)$ and $\mathcal{O}(m \cdot \mathbf{max}(m, |\Sigma|))$, respectively.

More effective implementations of the Fast-Search and Forward-Fast-Search algorithm are obtained along the same lines of the Tuned-Boyer-Moore algorithm [13] by making use of a fast-loop, using a technique described in Section 4.1 and shown in Figure 3(A). Then subsequent matching phase can start with the $(m-2)$-*nd* character of the pattern. At the end of the matching phase the algorithms uses the good-suffix rule for shifting.

3.2 The q-Hash Algorithms

Algorithms in the q-Hash family have been introduced in [15] where the author presented an adaptation of the Wu and Manber multiple string matching algorithm [18] to single string matching problem.

The idea of the q-Hash algorithm is to consider factors of the pattern of length q. Each substring w of such a length q is hashed using a function h into integer values within 0 and 255. Then the algorithm computes in a preprocessing phase a function $\mathit{shift}() : \{0, 1, \ldots, 255\} \rightarrow \{0, 1, \ldots, m-q\}$. Formally for each $0 \le c \le 255$ the value $\mathit{shift}(c)$ is defined by

$$\mathit{shift}(c) = \mathbf{min}\Big(\{0 \le k < m-q \mid h(p[m-k-q\,..\,m-k-1]) = c\} \cup \{m-q\}\Big)\ .$$

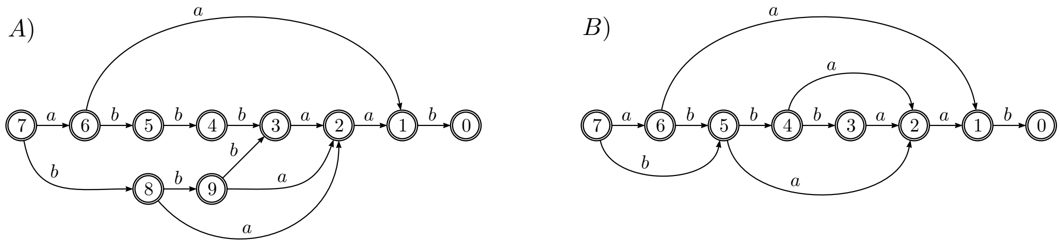


Figure 1. The factor automaton (A) and the factor oracle (B) of the reverse of pattern $p = baabbba$. The factor automaton recognizes all, and only, the factors of the reverse pattern. On the other hand note that the word aba is recognized by the factor oracle whereas it is not a factor.

The searching phase of the algorithm consists of reading, for each shift s of the pattern in the text, the substring $w = t[s+m-q \mathinner{..} s+m-1]$ of length q. If $shift[h(w)] > 0$ then a shift of length $shift[h(w)]$ is applied. Otherwise, when $shift[h(w)] = 0$ the pattern x is naively checked in the text. In this case a shift of length sh is applied where $sh = m - 1 - i$ with

$$i = max\{0 \leq j \leq m - q | h(x[j \mathinner{..} j + q - 1]) = h(x[m - q + 1 \mathinner{..} m - 1]\}.$$

3.3 The Backward-Automaton-Matching Algorithms

Algorithms based on the Boyer-Moore strategy generally try to match suffixes of the pattern but it is possible to match some prefixes or some factors of the pattern by scanning the current window of the text from right to left in order to improve the length of the shifts. This can be done by the use of factor automata and factor oracles.

The factor automaton [4,9,3] of a pattern p, $Aut(p)$, is also called the factor DAWG of p (for Directed Acyclic Word Graph). Such an automaton recognizes all the factors of p. Formally the language recognized by $Aut(p)$ is defined as follows

$$\mathcal{L}(Aut(p)) = \{u \in \Sigma^* : \text{ exists } v, w \in \Sigma^* \text{ such that } p = vuw\}.$$

The factor oracle of a pattern p, $Oracle(p)$, is a very compact automaton which recognizes at least all the factors of p and slightly more other words. Formally $Oracle(p)$ is an automaton $\{Q, m, Q, \Sigma, \delta\}$ such that

1. Q contains exactly $m + 1$ states, say $Q = \{0, 1, 2, 3, \ldots, m\}$
2. m is the initial state
3. all states are final
4. the language accepted by $Oracle(p)$ is such that $\mathcal{L}(Aut(p)) \subseteq \mathcal{L}(Oracle(p))$.

Despite the fact that the factor oracle is able to recognize words that are not factors of the pattern, it can be used to search for a pattern in a text since the only factor of p of length greater or equal to m which is recognized by the oracle is the pattern itself. The computation of the oracle is linear in time and space in the length of the pattern.

In Figure 1 are shown the factor automaton and the factor oracle of the reverse of pattern $p = baabbba$.

The data structures factor automaton and factor oracle are used respectively in [10,11] and in [1] to get optimal pattern matching algorithms on the average. The algorithm which makes use of the factor automaton of the reverse pattern is called

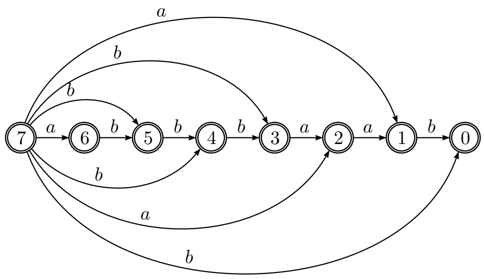


Figure 2. The nondeterministic factor automaton of the string *abbbaab*.

BDM (for Backward Dawg Matching) while the algorithm using the factor oracle is called BOM (for Backward Oracle Matching). Such algorithms move a window of size m on the text. For each new position of the window, the automaton of the reverse of p is used to search for a factor of p from the right to the left of the window. The basic idea of the BDM and BOM algorithms is that if its backward search failed on a letter c after the reading of a word u then cu is not a factor of p and moving the beginning of the window just after c is secure. If a factor of length m is recognized then we have found an occurrence of the pattern.

The BDM and BOM algorithms have a quadratic worst case time complexity but are optimal in average since they perform $\mathcal{O}(n(log_\sigma m)/m)$ inspections of text characters reaching the best bound shown by Yao [19] in 1979.

3.4 The BNDM Algorithm

The BNDM algorithm [16] (for Backward Nondeterministic Dawg Match) is a bit-parallel simulation [2] of the BDM algorithm. It uses a nondeterministic automaton instead of the deterministic one in the BDM algorithm.

Figure 2 shows the nondeterministic version of the factor automaton for the reverse of pattern $p = baabbba$. For each character $c \in \Sigma$, a bit vector $B[c]$ is initialized in the preprocessing phase. The i-th bit is 1 in this vector if c appears in the reversed pattern in position i. Otherwise the i-th bit is set to 0. The state vector D is initialized to 1^m. The same kind of right to left scan in a window of size m is performed as in the BOM algorithm while the state vector is updated in a similar fashion as in the Shift-And algorithm [2]. If the m-th bit is 1 after this update operation, we have found a prefix starting at position j where j is the number of updates done in this window. Thus if j is the first position in the window, a match has been found.

A simplified version of the BNDM, called SBNDM, as been presented in [12]. This algorithm differs from the original one in the main loop which starts each iteration with a test of two consecutive text characters. Moreover it implements a fast-loop to obtain better results on average. Experimental results show that this simplified variant is always more efficient than the original one.

4 New Variations of the BOM Algorithm

In this section we present two variations of the BOM algorithm which perform better in most cases. The first idea consists in extending the BOM algorithm with a fast-loop over oracle transitions, along the same lines of the Tuned-Boyer-Moore algorithm [13]. Thus we are able to perform factor searching if and only if a portion of the pattern has already matched against a portion of the current window of the text. We present this idea in Section 4.1.

An other efficient variation of the BOM algorithm can be obtained by applying the idea suggested by Sunday in the Quick-Search algorithm and then implemented in the Forward-Fast-Search algorithm. This consists in taking into account, while shifting, the character which follows the current window of the text, since it is always involved in the next alignment. Such a variation is presented in Section 4.2.

4.1 Extending the BOM Algorithm with a Fast-Loop

In this section we present an extension of the BOM algorithm by introducing a fast-loop with the aim of obtaining better results on the average. We discuss the application of different variations of the fast-loop, listed in Figure 3, and present experimental results in order to identify the best choice.

The idea of a fast loop has been proposed in [5]. The fast-loop we are using here has first introduced in the Tuned-Boyer-Moore algorithm [13] and later largely used in almost all variations of the Boyer-Moore algorithm. Generally a fast-loop is implemented by iterating the bad character heuristic in a checkless cycle, in order to quickly locate an occurrence of the rightmost character of the pattern. Suppose $bc() : \Sigma \rightarrow \{0, 1, \ldots, m\}$ is the function which implements the bad-character heuristic defined, for all $c \in \Sigma$, by

$$bc(c) = \mathbf{min}(\{0 \leq k < m \mid p[m-1-k] = c\} \cup \{m\}) \ .$$

If we suppose that $t[j]$ is the rightmost character of the current window of the text for a shift s, i.e. $j = s + m - 1$, then the original fast-loop can be implemented in a form similar to that presented in Figure 3(A).

In order to avoid testing the end of the text we could append the pattern at the end of the text, i.e. set $t[n \,..\, n+m-1]$ to p. Thus we exit the algorithm only when an occurrence of p is found. If this is not possible (because memory space is occupied) it is always possible to store $t[n-m \,..\, n-1]$ in z then set $t[n-m \,..\, n-1]$ to p and check z at the end of the algorithm without slowing it.

However algorithms based on the bad character heuristic obtain good results only in the case of large alphabets and short patterns. It turns out moreover from experimental results [15] that the strategy of using an automaton to match prefixes or factors is much better when the length of the pattern increases.

This behavior is due to the fact that for large patterns an occurrence of the rightmost character of the window, i.e. $t[j]$, can be found in the pattern and the probability that the rightmost occurrence is near the rightmost position increases for longer patterns and smaller alphabets. In this latter case an iteration of the fast loop leads to a short shift. In contrast, when using an oracle for matching, it is common that after a small number of characters we are not able to perform other transitions. So generally this strategy looks for a number of characters greater than 1, for each iteration, but leads to shift of length m.

As a first step we can translate the idea of the fast-loop over to automaton transitions. This consists in shifting the pattern along the text with no more check until a non-undefined transition is found with the rightmost character of the current window of the text. This can be translated in the fast-loop presented in Figure 3(B).

It turns out from experimental results presented in Figure 4 that the variation of the BOM algorithm which uses the fast-loop on transitions (col.B) performs better

(A)

$k = bc(t_j)$
while $(k \neq 0)$ **do**
  $j = j + k$
  $k = bc(t_j)$

(B)

$q = \delta(m, t_j)$
while $(q == \perp)$ **do**
  $j = j + m$
  $q = \delta(m, t_j)$

(C)

$q = \delta(m, t_j)$
if $q \neq \perp$ **then**
  $p = \delta(q, t_{j-1})$
while $(p == \perp)$ **do**
  $j = j + m - 1$
  $q = \delta(m, t_j)$
  if $q \neq \perp$ **then**
    $p = \delta(q, t_{j-1})$

(D)

$q = \lambda(t_j, t_{j-1})$
while $(q == \perp)$ **do**
  $j = j + m - 1$
  $q = \lambda(t_j, t_{j-1})$

Figure 3. Different variations of the fast-loop where $t_j = t_{s+m-1}$ is the rightmost character of the current window of the text. (A) The original fast-loop based on the bad character rule. (B) A modified version of the fast-loop based on automaton transitions. (C) A fast-loop based on two subsequent automaton transitions. (D) An efficient implementation of the previous fast loop which encapsulate two subsequent transitions in a single λ table.

Experimental results with $\sigma = 8$

m	BOM	(A)	(B)	(C)	(D)
4	157.62	95.95	135.95	109.03	55.35
8	85.48	58.66	78.70	58.63	34.16
16	43.04	43.36	43.00	37.15	26.82
32	26.63	35.00	28.29	25.93	21.25
64	17.39	28.05	17.13	17.00	14.42
128	15.28	23.68	15.75	15.87	12.87
256	10.79	19.86	9.60	9.76	8.53
512	6.18	14.29	6.11	5.76	4.76
1024	3.29	8.20	3.45	3.35	2.64

Experimental results with $\sigma = 16$

m	BOM	(A)	(B)	(C)	(D)
4	103.28	66.81	86.28	93.53	40.63
8	71.59	38.72	60.27	44.02	21.73
16	39.61	26.57	35.70	23.68	14.91
32	18.68	21.80	18.82	15.71	12.73
64	12.67	20.09	12.55	12.73	12.49
128	14.22	19.38	14.14	12.35	10.38
256	8.81	19.05	8.12	7.83	6.88
512	4.62	17.73	4.62	4.53	3.60
1024	2.35	11.49	2.66	2.89	2.67

Experimental results with $\sigma = 32$

m	BOM	(A)	(B)	(C)	(D)
4	78.76	55.23	57.75	88.57	37.44
8	51.68	30.37	42.03	39.84	18.59
16	35.40	19.92	30.18	20.34	12.29
32	20.62	16.12	19.34	12.20	11.58
64	12.11	14.84	11.55	10.63	11.10
128	12.60	15.63	11.26	10.01	7.46
256	7.58	16.73	6.32	5.90	3.79
512	4.29	17.90	3.73	3.83	3.20
1024	2.87	14.19	2.67	2.79	2.01

Experimental results with $\sigma = 64$

m	BOM	(A)	(B)	(C)	(D)
4	64.84	50.93	42.34	88.52	37.04
8	39.35	27.44	29.29	38.84	17.99
16	26.09	17.12	22.03	20.07	11.57
32	19.45	14.09	17.11	11.81	10.76
64	13.15	13.58	12.28	10.37	10.70
128	13.11	17.67	10.86	9.76	6.35
256	6.25	18.04	5.79	5.55	3.60
512	2.91	18.00	3.12	5.32	1.98
1024	2.71	16.89	2.58	2.42	1.57

Figure 4. Experimental results obtained by comparing the original BOM algorithm (in the first column) against variations implemented using the four fast-loop presented in Figure 3. The results have been obtained by searching 200 random patterns in a 40Mb text buffer with a uniform distribution over an alphabet of dimension σ. Running times are expressed in hundredths of seconds.

than the original algorithm (first column), especially for large alphabets. However it is not flexible since its performances decrease when the length of the pattern increases or when the dimension of the alphabet is small. This is the fast-loop finds only a small number of undefined transitions for small alphabets or long patterns.

The variation of the algorithm we propose tries two subsequent transitions for each iteration of the fast-loop with the aim to find with higher probability an undefined transition. This can be translated in the fast-loop presented in Figure 3(C). From experimental results it turns out that such a variation (Figure 4, col.C) obtains better results than the previous one only for long pattern and large alphabets. This is for each iteration of the fast-loop the algorithm performs two subsequent transitions affecting the overall performance.

To avoid this problem we could encapsulate the two first transitions of the oracle in a function $\lambda() : (\Sigma \times \Sigma) \rightarrow Q$ defined, for each $a, b \in \Sigma$, by

$$\lambda(a, b) = \begin{cases} \bot & \text{if } \delta(m, a) = \bot \\ \delta(\delta(m, a), b) & \text{otherwise.} \end{cases}$$

Thus the fast loop can be implemented as presented in Figure 3(D). At the end of the fast-loop the algorithm could start standard transitions with the Oracle from state $q = \lambda(t[j], t[j-1])$ and character $t[j-2]$. The function λ can be implemented with a two dimensional table in $\mathcal{O}(\sigma^2)$ time and space.

The resulting algorithm, here named Extended-BOM algorithm, is very fast and flexible. Its pseudocode is presented in Figure 6(A). From experimental results in Figure 4 it turns out that the Extended-BOM algorithm (col.D) is the best choice in most cases and, differently from the original algorithm, it has very good performance also for short patterns.

4.2 Looking for the Forward Character

The idea of looking for the forward character for shifting has been originally introduced by Sunday in the Quick-Search algorithm [17] and then efficiently implemented in the Forward-Fast-Search algorithm [7]. Specifically, it is based on the following observation: when a mismatch character is encountered while comparing the pattern with the current window of the text $t[s \mathinner{..} s+m-1]$, the pattern is always shifted to the right by at least one character, but never by more than m characters. Thus, the character $t[s+m]$ is always involved in testing for the next alignment.

In order to take into account the forward character of the current window of the text without skip safe alignment we construct the *forward factor oracle* of the reverse pattern. The forward factor oracle of a word p, $FOracle(p)$, is an automaton which recognizes at least all the factors of p, eventually preceded by a word $x \in \Sigma \cup \{\varepsilon\}$. More formally the language recognized by $FOracle(p)$ is defined by

$$\mathcal{L}(FOracle(p)) = \{xw \mid x \in \Sigma \cup \{\varepsilon\} \text{ and } w \in \mathcal{L}(Oracle(p))\}$$

Observe that in the previous definition the prefix x could be the empty string. Thus if w is a word recognized by the factor oracle of p then the word cw is recognized by the forward factor oracle, for all $c \in \Sigma \cup \{\varepsilon\}$.

The forward factor oracle of a word p can be constructed, in time $\mathcal{O}(m + \Sigma)$, by simply extending the factor oracle of p with a new initial state which allows to perform transitions starting at the text character of position $s+m$ of the text, avoiding to skip valid shift alignments.

Suppose $Oracle(p) = \{Q, m, Q, \Sigma, \delta\}$, for a pattern p of length m. We construct $FOracle(p)$ by adding a new initial state $(m+1)$ and introducing transitions from state $(m+1)$. More formally, given a pattern p of length m, $FOracle(p)$ is an automaton $\{Q', (m+1), Q, \Sigma, \delta'\}$, where

1. $Q' = Q \cup \{(m+1)\}$

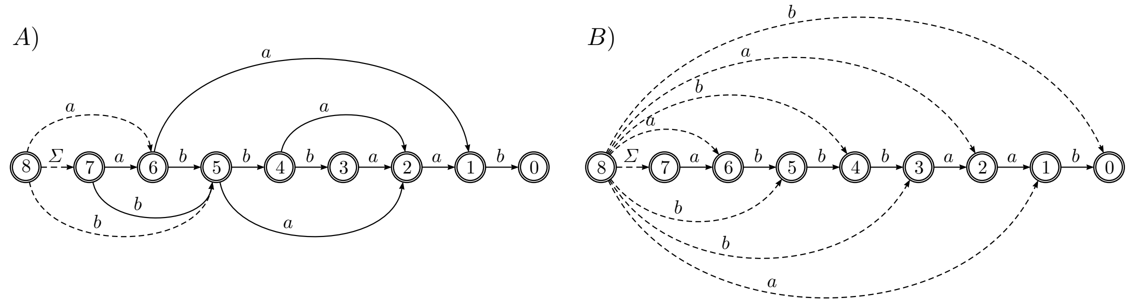


Figure 5. (A) The forward factor oracle of the reverse pattern $p = baabbba$ (B) The nondeterministic version of the forward factor automaton of the reverse pattern $p = baabbba$

2. $(m+1)$ is the initial state
3. all states are final
4. $\delta'(q,c) = \delta(q,c)$ for all $c \in \Sigma$, if $q \neq (m+1)$
5. $\delta'(m+1,c) = \{m, \delta(m,c)\}$ for all $c \in \Sigma$

Figure 5(A) shows the forward factor oracle of the reverse pattern $p = baabbba$. The dashed transitions are those outgoing from the new initial state. A transition labeled with all characters of the alphabet has been introduced from state $(m+1)$ to state m. Note that, according to rule n.5, the forward factor oracle of the reverse pattern p is a non-deterministic automaton. For example, starting from the initial state 8 in Figure 5(A), after reading the couple of characters aa, both states 6 and 1 are active.

Observe moreover that we have to read at least two consecutive characters to find an undefined transition. This is state m is always active after reading any character of the alphabet.

Suppose we start transitions from the initial state of $FOracle(p)$. Then after reading a word $w = au$, with $a \in \Sigma$ and $u \in \Sigma^+$, at most two different states could be active, i.e., state $x = \delta^*(w)$ and state $y = \delta^*(u)$. Where we recall that δ is the transition function of $Oracle(p)$ and where $\delta^*() : \Sigma^* \leftarrow Q$ is the *final state* function induced by δ and defined recursively by

$$\delta^*(w) = \delta(\delta^*(w'), c), \quad \text{for each } w = w'c, \text{ with } w' \in \Sigma^*, c \in \Sigma.$$

The idea consists in simulating the behavior of the nondeterministic forward factor oracle by following transition for only one of the two active states. More precisely we are interested only in transitions from state q where

$$q = \begin{cases} y = \delta^*(u) & \text{if } u[0] = p[m-1] \\ x = \delta^*(w) & \text{otherwise} \end{cases}$$

To prove the correctness of our strategy, suppose first we have read a word $w = au$, as defined above, and $u[0] \neq p[m-1]$. If $Oracle(p)$ recognizes a word u (i.e. $\delta^*(u) \neq \perp$) then by definition $FOracle(p)$ recognize the word au, since $a \in \Sigma \cup \{\varepsilon\}$.

Suppose now that $u[0] = p[m-1]$. If $Oracle(p)$ recognizes a word w then it recognizes also word u which is a suffix of w. Thus by definition $FOracle(p)$ recognizes the word xu, with $x = \varepsilon$.

The simulation of the forward factor oracle can be done by simply changing the computation of the λ table in the following way

(A)

```
Extended-BOM(p, m, t, n)
δ ← precompute-factor-oracle(p)
for a ∈ Σ do
    q ← δ(m, a)
    for b ∈ Σ do
        if q = ⊥ then λ(a, b) ← ⊥
        else λ(a, b) ← δ(q, b)
t[n .. n + m − 1] ← p
j ← m − 1
while j < n do
    q ← λ(t[j], t[j − 1])
    while q = ⊥ do
        j ← j + m − 1
        q ← λ(t[j], t[j − 1])
    i ← j − 2
    while q ≠ ⊥ do
        q ← δ(q, t[i])
        i ← i − 1
    if i < j − m + 1 then
        output(j)
        i ← i + 1
    j ← j + i + m
```

(B)

```
Forward-Bom(p, m, t, n)
1.  δ ← precompute-factor-oracle(p)
2.  for a ∈ Σ do
3.      q ← δ(m, a)
4.      for b ∈ Σ do
5.          if q = ⊥ then λ(a, b) ← ⊥
6.          else λ(a, b) ← δ(q, b)
7.  q ← δ(m, p[m − 1])
8.  for a ∈ Σ do λ(a, p[m − 1]) ← q
9.  t[n .. n + m − 1] ← p
10. j ← m − 1
11. while j < n do
12.     q ← λ(t[j + 1], t[j])
13.     while q = ⊥ do
14.         j ← j + m
15.         q ← λ(t[j + 1], t[j])
16.     i ← j − 1
17.     while q ≠ ⊥ do
18.         q ← δ(q, t[i])
19.         i ← i − 1
20.     if i < j − m + 1 then
21.         output(j)
22.         i ← i + 1
23.     j ← j + i + m
```

Figure 6. (A) The Extended-BOM algorithm which extend the original BOM algorithm by using an efficient fast-loop. (B) The Forward-BOM algorithm which performs a look ahead for character of position $t[j+1]$ in text to obtain larger shift advancements.

```
Forward-SBNDM(p, m, t, n)
for all c ∈ Σ do B[i] ← 1
for i = 0 to m − 1 do B[p[i]] ← B[p[i]] | (1 << (m − i))
j ← m − 1
while j < n do
    D ← (B[t[j + 1]] << 1) & B[t[j]]
    if D ≠ 0 then pos ← j
    while D ← (D + D) & B[t[j − 1]] do j ← j − 1
    j ← j + m − 1
    if j = pos then
        output(j)
        j ← j + 1
    else j ← j + m
```

Figure 7. The Forward SBNDM algorithm which simulates using bit-parallelism the non deterministic forward automaton of the reverse pattern.

$$\lambda(a,b) = \begin{cases} \delta(m,b) & \text{if } \delta(m,a) = \bot \ \vee \ b = p[m-1] \\ \delta(\delta(m,a),b) & \text{otherwise} \end{cases}$$

Figure 6(B) shows the code of the Forward-Bom algorithm. Here the fast loop has been modified to take into account also the forward character of position $t[s+m]$. However if there is no transition for the first two characters, $t[s+m]$ and $t[s+m-1]$, the algorithm can shift the pattern of m position to the right. Line 1 of the preprocessing phase can be performed in $\mathcal{O}(m)$-time, lines 2 to 6 in $\mathcal{O}(\sigma^2)$ and line 8 in $\mathcal{O}(\sigma)$. Thus the preprocessing phase can be performed in $\mathcal{O}(m+\sigma^2)$ time and space.

This idea can be applied also to the SBNDM algorithm based on bit-parallelism. In this latter case we have to add a new first state and change the preprocessing in order to perform correct transitions from the first state. Moreover we need $m + 1$ bits for representing the NFA, thus we are able to search only for patterns with $1 \le m < w$, if w is the dimension of a machine word. Figure 7 shows the code of the Forward-SBNDM algorithm.

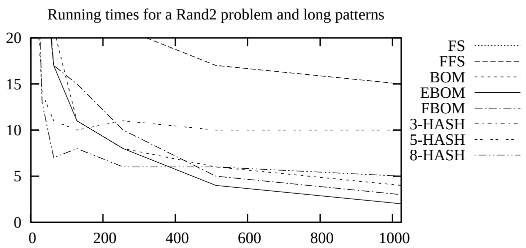
5 Experimental Results

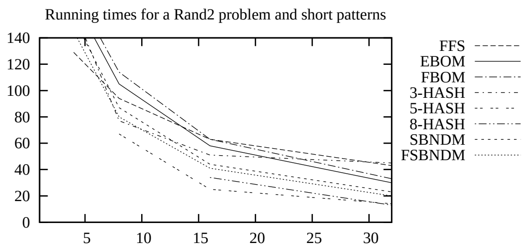
We present next experimental data which allow to compare in terms of running time the following string matching algorithms under various conditions: Fast-Search (FS), Forward-Fast-Search (FFS), BOM (BOM), SBNDM (SBNDM), q-Hash (q-HASH with $q = 3, 5, 8$), Extended-BOM (EBOM), Forward-BOM (FBOM) and Forward-SBNDM (FSBNDM).

All algorithms have been implemented in the **C** programming language and were used to search for the same strings in large fixed text buffers on a PC with Intel Core2 processor of 1.66 GHz. In particular, the algorithms have been tested on seven Randσ problems, for $\sigma = 2, 4, 8, 16, 32, 64, 128$, on a genome, on a protein sequence and on a natural language text buffer. Searching have been performed for patterns of length $m = 2, 4, 8, 16, 32, 64, 128, 256, 512$, and 1024. In the following tables, running times are expressed in hundredths of seconds.

5.1 Running Times for Random Problems

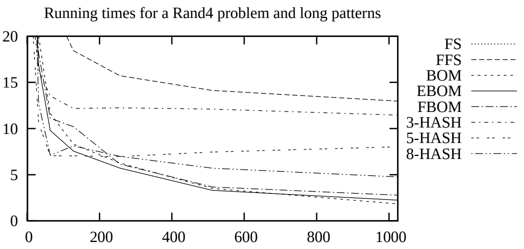
For the case of random texts the algorithms have been tested on seven Randσ problems. Each Randσ problem consists of searching a set of 400 random patterns of a given length in a 20Mb random text over a common alphabet of size σ, whit a uniform distribution of characters.

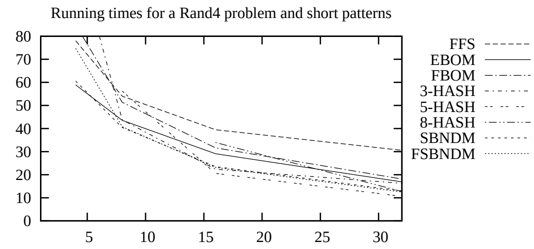




m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	153.52	**129.07**	209.07	169.22	177.31	162.98	-	-	155.38	145.47
8	115.44	94.42	133.73	105.08	114.17	77.03	**67.91**	-	87.42	80.72
16	83.60	63.05	71.75	58.91	63.23	51.65	**25.27**	34.33	44.87	41.31
32	61.96	43.40	38.55	30.58	33.24	45.38	14.85	**13.50**	23.88	20.77
64	48.16	32.69	21.24	17.43	17.91	44.65	11.53	**7.42**	-	-
128	39.55	24.90	11.91	11.73	15.63	44.02	10.09	**8.34**	-	-
256	32.80	21.14	8.45	8.43	10.00	44.92	11.02	**6.86**	-	-
512	28.07	17.27	6.36	**4.87**	5.87	45.65	10.04	6.21	-	-
1024	23.39	15.47	4.00	**2.79**	3.95	44.72	10.59	5.14	-	-

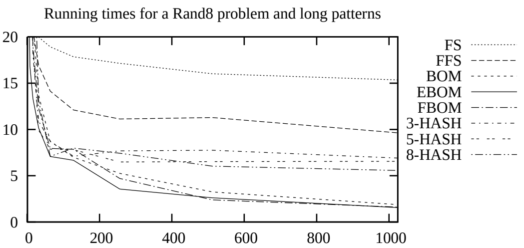
Running times for a Rand2 problem

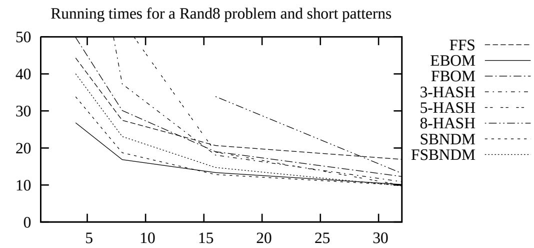




m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	82.12	78.03	111.55	**58.93**	84.93	117.82	-	-	60.57	74.73
8	60.00	54.02	61.31	43.57	51.38	43.77	56.23	-	**40.46**	40.79
16	49.05	39.49	35.58	29.11	31.66	22.40	**20.54**	33.98	23.49	23.15
32	41.72	30.56	19.98	16.88	18.13	16.27	**10.60**	12.70	12.97	12.48
64	37.11	23.71	11.63	9.79	11.11	13.53	**7.05**	7.11	-	-
128	32.02	18.43	8.30	7.56	10.20	12.17	**7.05**	8.12	-	-
256	28.54	15.72	6.27	**5.72**	6.16	12.25	6.97	6.99	-	-
512	26.07	14.13	3.52	**3.31**	3.67	12.10	7.46	5.71	-	-
1024	22.14	12.97	**1.83**	2.25	2.78	11.46	8.02	4.76	-	-

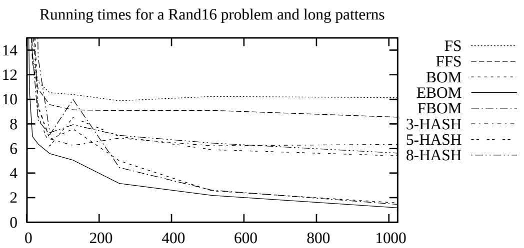
Running times for a Rand4 problem

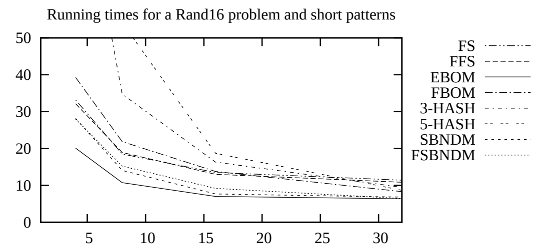




m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	46.34	44.32	78.94	**26.81**	49.87	105.12	-	-	33.82	40.05
8	29.61	27.46	43.23	**16.85**	30.11	37.30	54.47	-	18.70	23.09
16	22.46	20.67	21.72	13.34	19.01	18.07	18.90	33.90	**12.81**	14.71
32	19.97	16.91	13.70	10.15	12.29	10.89	9.92	13.14	9.92	**9.73**
64	18.93	14.14	8.70	**7.07**	7.95	8.71	7.87	7.09	-	-
128	17.85	12.10	6.99	**6.66**	7.85	7.11	7.81	7.98	-	-
256	17.15	11.13	5.26	**3.56**	4.70	7.68	6.48	7.43	-	-
512	16.02	11.29	3.25	2.61	**2.38**	7.75	6.53	6.03	-	-
1024	15.35	9.63	1.88	**1.55**	1.61	6.91	6.56	5.57	-	-

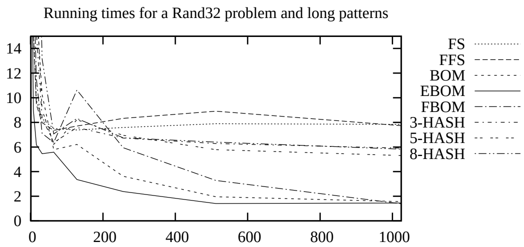
Running times for a Rand8 problem

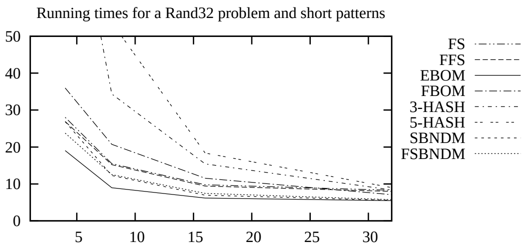




m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	33.17	32.13	52.02	**20.09**	39.26	102.31	-	-	28.16	27.98
8	18.52	18.91	35.48	**10.73**	21.87	34.74	54.09	-	14.04	15.22
16	13.48	13.01	19.61	**6.98**	13.76	16.33	18.71	33.78	7.66	9.18
32	11.41	10.83	9.33	**6.36**	8.29	9.46	8.64	13.35	6.80	6.43
64	10.54	9.57	6.74	**5.58**	7.12	6.79	6.21	7.29	-	-
128	10.39	9.14	7.58	**5.05**	9.99	6.25	8.52	7.93	-	-
256	9.88	9.08	5.00	**3.16**	4.45	6.84	6.98	7.07	-	-
512	10.23	9.10	2.55	**2.18**	2.61	6.22	5.90	6.44	-	-
1024	10.14	8.55	1.57	**1.18**	1.45	6.33	5.40	5.62	-	-

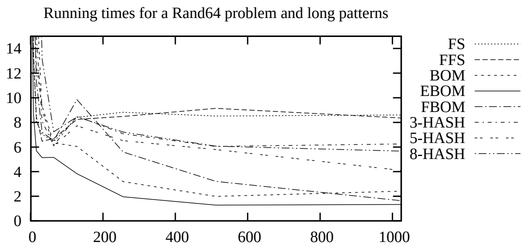
Running times for a Rand16 problem

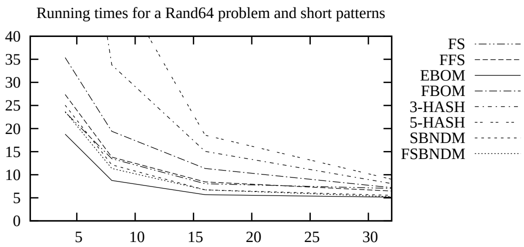




m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	28.04	26.91	35.98	**19.03**	35.98	100.51	-	-	26.88	23.75
8	15.51	15.23	24.54	**8.98**	20.74	34.34	53.71	-	12.28	12.54
16	9.78	9.44	17.46	**6.18**	11.56	15.44	18.36	34.14	6.95	7.46
32	8.29	7.98	10.26	**5.46**	7.11	8.36	9.02	13.16	5.59	5.75
64	7.50	7.35	5.78	**5.58**	6.37	6.37	6.22	7.07	-	-
128	7.38	7.70	6.21	**3.36**	10.62	7.58	8.21	8.32	-	-
256	7.59	8.33	3.62	**2.38**	5.94	6.73	6.95	6.75	-	-
512	7.89	8.91	1.96	**1.41**	3.28	6.28	5.78	6.40	-	-
1024	7.84	7.73	1.57	1.45	**1.39**	5.91	5.31	5.83	-	-

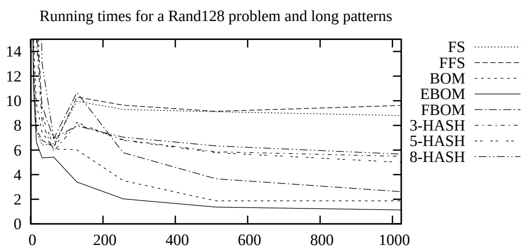
Running times for a Rand32 problem

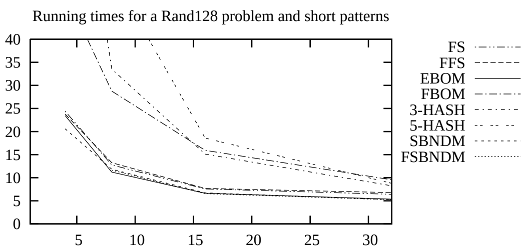




m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	23.55	27.38	29.10	**18.79**	35.38	97.23	-	-	25.05	23.67
8	13.48	13.82	18.51	**8.76**	19.41	33.79	53.80	-	12.15	11.37
16	8.06	8.44	12.64	**5.69**	11.35	15.07	18.56	33.32	6.72	6.72
32	7.04	6.47	9.33	**5.14**	7.20	8.09	9.00	13.15	5.55	5.25
64	6.44	6.68	6.34	**5.16**	6.52	6.13	6.09	7.23	-	-
128	8.41	8.24	6.05	**3.84**	9.85	8.51	7.72	8.45	-	-
256	8.82	8.49	3.19	**1.96**	5.59	7.08	6.52	7.21	-	-
512	8.52	9.14	1.99	**1.28**	3.21	6.05	5.79	6.07	-	-
1024	8.60	8.36	2.41	**1.33**	1.64	6.25	4.10	5.67	-	-

Running times for a Rand64 problem





m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	24.42	23.79	26.16	23.45	50.15	99.57	-		27.21	**20.60**
8	12.77	13.24	15.06	**11.21**	28.77	33.54	53.99	-	11.83	11.65
16	7.58	7.69	9.81	6.64	15.92	15.10	18.59	33.84	6.72	**6.55**
32	6.40	6.79	7.43	5.36	9.56	8.24	8.79	13.01	5.25	**5.24**
64	6.52	6.25	6.09	**5.42**	6.92	5.90	6.32	6.96	-	-
128	9.96	10.31	6.02	**3.40**	10.67	8.08	8.24	7.96	-	-
256	9.29	9.64	3.52	**2.03**	5.78	6.83	6.79	7.04	-	-
512	9.11	9.14	1.87	**1.37**	3.67	5.86	5.78	6.33	-	-
1024	8.80	9.62	1.88	**1.13**	2.62	5.49	5.00	5.67	-	-

Running times for a Rand128 problem

Experimental results show that the Extended-BOM and the Forward-BOM algorithms obtain the best run-time performance in most cases. In particular for small alphabets and short patterns the presented variations are second to algorithms in the q-Hash family. Moreover for large alphabets and short patterns algorithms based on bit-parallelism are the best choice. Note however that for alphabets of medium dimension, when the pattern is short, the performance of the Extended-BOM algorithm outperform those of bit-parallel algorithms that until now have been considered the best choice for short patterns.

5.2 Running Times for Real World Problems

The tests on real world problems have been performed on a genome sequence and on a natural language text buffer. A genome is a DNA sequence composed of the four nucleotides, also called base pairs or bases: Adenine, Cytosine, Guanine and Thymine. The genome we used for these tests is a sequence of 4, 638, 690 base pairs of *Escherichia coli*. We used the file E.coli file of the Large Canterbury Corpus (http://www.data-compression.info/Corpora/CanterburyCorpus/).

The tests on the protein sequence has been performed using a 2.4Mb file containing a protein sequence from the human genome with 22 different characters.

For the experiments on the natural language text buffer we used the file world192.txt (The CIA World Fact Book) of the Large Canterbury Corpus. The alphabet is composed of 94 different characters. The text is composed of 2, 473, 400 characters.

From experimental results it turns out that the Extended-BOM algorithm obtains in most cases the best results and sporadically is second to algorithms of the q-Hash family. Again better results are obtained for medium dimensions of the alphabet.

m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	18.64	16.91	23.25	**12.65**	19.09	25.48	-	-	12.96	17.30
8	13.85	11.63	13.04	10.27	11.40	9.90	12.34	-	**8.73**	9.01
16	11.48	8.47	7.73	6.77	6.47	4.76	**4.39**	7.74	5.28	5.50
32	9.58	6.44	4.53	3.52	4.07	3.20	2.77	2.85	3.04	**2.62**
64	8.56	4.92	2.50	1.95	2.42	2.65	**1.60**	1.84	-	-
128	7.05	4.01	1.74	**1.73**	1.91	2.42	1.84	2.08	-	-
256	6.41	3.35	1.33	**1.32**	1.33	2.90	1.60	1.41	-	-
512	5.66	3.20	0.94	0.82	**0.78**	2.39	1.60	1.61	-	-
1024	5.97	2.19	0.98	0.66	**0.51**	2.50	1.21	1.21	-	-

Running times for a genome sequence ($\sigma = 4$)

m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	4.33	2.93	8.30	**2.14**	5.51	14.49	-	-	5.19	3.59
8	**1.68**	2.64	4.21	2.27	3.58	4.38	8.09	-	2.31	1.85
16	1.71	1.57	2.66	**1.05**	1.92	2.50	2.58	4.54	1.25	1.05
32	1.41	1.47	1.62	**0.87**	1.27	1.30	1.37	1.64	0.89	0.89
64	1.21	1.02	1.10	**0.63**	1.18	0.85	0.82	1.25	-	-
128	1.09	1.33	1.13	**0.67**	1.51	0.98	1.14	1.22	-	-
256	1.37	1.44	0.59	0.51	**0.47**	0.90	0.90	0.82	-	-
512	1.20	1.56	0.50	**0.27**	0.30	0.77	0.90	0.88	-	-
1024	1.25	1.64	0.39	0.35	**0.27**	0.87	0.70	0.74	-	-

Running times for a protein sequence ($\sigma = 22$)

m	FS	FFS	BOM	EBOM	FBOM	3-HASH	5-HASH	8-HASH	SBNDM	FSBNDM
4	3.66	3.73	5.70	**2.70**	4.79	11.70	-	-	3.65	3.25
8	2.12	2.01	3.95	**1.52**	2.44	3.83	6.50	-	1.96	1.82
16	1.54	1.29	2.73	**0.82**	1.80	2.10	1.96	3.66	1.14	1.05
32	1.14	1.09	1.35	1.06	1.29	0.95	1.60	1.05	**0.55**	0.86
64	0.91	0.82	1.14	0.82	1.45	0.70	0.66	**0.63**	-	-
128	1.10	1.17	0.86	**0.79**	1.32	0.86	0.90	0.94	-	-
256	0.93	1.28	**0.48**	0.59	0.67	0.83	0.75	0.70	-	-
512	0.78	1.21	0.59	**0.27**	0.71	0.66	0.66	0.40	-	-
1024	0.65	1.55	0.69	0.52	0.80	0.63	**0.28**	0.47	-	-

Running times for a natural language text buffer ($\sigma = 93$)

6 Conclusion

We presented two efficient variants of the Backward Oracle Matching algorithm which is considered one of the most effective algorithm for exact string matching. The first variation, called Extended-BOM, introduces an efficient fast-loop over transitions of the oracle by reading two consecutive characters for each iteration. The second variation, called Forward-BOM, extends the previous one by using a look-ahead character at the beginning of transitions in order to obtain larger shift advancements.

It turns out from experimental results that the new proposed variations are very fast in practice and obtain the best results in most cases, especially for long patterns and alphabets of medium dimension.

References

1. C. Allauzen, M. Crochemore, and M. Raffinot: *Factor oracle: a new structure for pattern matching*, in SOFSEM'99, J. Pavelka, G. Tel, and M. Bartosek, eds., LNCS 1725, Milovy, Czech Republic, 1999, Springer-Verlag, Berlin, pp. 291–306.
2. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Commun. ACM, 35(10) 1992, pp. 74–82.
3. A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, M. T. Chen, and J. Seiferas: *The smallest automaton recognizing the subwords of a text.* Theor. Comput. Sci., 40(1) 1985, pp. 31–55.
4. A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnel: *Linear size finite automata for the set of all subwords of a word: an outline of results.* Bull. Eur. Assoc. Theor. Comput. Sci., 21 1983, pp. 12–20.
5. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) 1977, pp. 762–772.
6. D. Cantone and S. Faro: Fast-Search*: a new efficient variant of the Boyer-Moore string matching algorithm.* WEA 2003, LNCS 2647(4/5) 2003, pp. 247–267.
7. D. Cantone and S. Faro: *Fast-Search Algorithms: New Efficient Variants of the Boyer-Moore Pattern-Matching Algorithm.* J. Autom. Lang. Comb., 10(5/6) 2005, pp. 589–608.
8. C. Charras and T. Lecroq: *Handbook of exact string matching algorithms*, King's College Publications, 2004.
9. M. Crochemore: *Optimal factor tranducers*, in Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, eds., vol. 12 of NATO Advanced Science Institutes, Series F, Springer-Verlag, Berlin, 1985, pp. 31–44.
10. M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter: *Speeding up two string matching algorithms.* Algorithmica, 12(4/5) 1994, pp. 247–267.
11. M. Crochemore and W. Rytter: *Text algorithms*, Oxford University Press, 1994.
12. J. Holub and B. Durian: *Fast variants of bit parallel approach to suffix automata.* Talk given in: The Second Haifa Annual International Stringology Research Workshop of the Israeli Science Foundation, http://www.cri.haifa.ac.il/events/2005/string/presentations/Holub.pdf, 2005.
13. A. Hume and D. M. Sunday: *Fast string searching.* Softw. Pract. Exp., 21(11) 1991, pp. 1221–1248.
14. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt: *Fast pattern matching in strings.* SIAM J. Comput., 6(1) 1977, pp. 323–350.
15. T. Lecroq: *Fast exact string matching algorithms.* Inf. Process. Lett., 102(6) 2007, pp. 229–235.
16. G. Navarro and M. Raffinot: *A bit-parallel approach to suffix automata: Fast extended string matching*, in Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching, M. Farach-Colton, ed., LNCS 1448, Piscataway, NJ, 1998, Springer-Verlag, Berlin, pp. 14–33.
17. D. M. Sunday: *A very fast substring search algorithm.* Commun. ACM, 33(8) 1990, pp. 132–142.
18. S. Wu and U. Manber: *A fast algorithm for multi-pattern searching*, Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.
19. A. C. Yao: *The complexity of pattern matching for a random string.* SIAM J. Comput., 8(3) 1979, pp. 368–387.

Fast Optimal Algorithms for Computing All the Repeats in a String*

Simon J. Puglisi[1], William F. Smyth[2,3], and Munina Yusufu[2]

[1] School of Computer Science & Information Technology,
RMIT University, GPO Box 2476V, Melbourne, Victoria 3001, Australia
sjp@cs.rmit.edu.au

[2] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton, Ontario, Canada L8S 4K1
{smyth,yusufum}@mcmaster.ca
http://www.cas.mcmaster.ca/cas/research/algorithms.htm

[3] Digital Ecosystems & Business Intelligence Institute
Curtin University, GPO Box U1987, Perth WA 6845, Australia
W.Smyth@curtin.edu.au

Abstract. Given a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an alphabet of size α, and a threshold $p_{min} \geq 1$, we first describe a new algorithm PSY1 that, based on suffix array construction, computes all the complete ***nonextendible*** repeats in $\boldsymbol{x}$ of length $p \geq p_{min}$. PSY1 executes in $\Theta(n)$ time independent of alphabet size and is an order of magnitude faster than the two other algorithms previously proposed for this problem. Second, we describe a new fast algorithm PSY2 for computing all complete ***supernonextendible*** repeats in $\boldsymbol{x}$ that also executes in $\Theta(n)$ time independent of alphabet size, thus asymptotically faster than methods previously proposed. Both algorithms require $9n$ bytes of storage, including preprocessing (with a minor *caveat* for PSY1). We conclude with a brief discussion of applications to bioinformatics and data compression.

1 Introduction

A ***repeating substring*** $\boldsymbol{u}$ in a string $\boldsymbol{x}$ is a substring of $\boldsymbol{x}$ that occurs more than once. A ***repeat*** in $\boldsymbol{x}$ is a set of repeating substrings $\boldsymbol{u}$ of $\boldsymbol{x}$; it can be specified by the length $p \geq 1$ of $\boldsymbol{u}$ (what we call its ***period***) and the locations at which $\boldsymbol{u}$ occurs. Thus in $\boldsymbol{x} = abaababa$, the tuple $(3;\ 1, 4, 6)$ describes the repeat of $\boldsymbol{u} = aba$ $(p = 3)$ at positions 1, 4, 6.

Following [20] we say that a repeat $(p;\ i_1, i_2, \ldots, i_k)$, $k \geq 2$, is ***complete*** iff it includes all occurrences of $\boldsymbol{u}$ in $\boldsymbol{x}$; ***left-extendible*** (LE) iff

$$\boldsymbol{x}[i_1{-}1] = \boldsymbol{x}[i_2{-}1] = \cdots = \boldsymbol{x}[i_k{-}1];$$

and ***right-extendible*** (RE) iff

$$\boldsymbol{x}[i_1{+}p] = \boldsymbol{x}[i_2{+}p] = \cdots = \boldsymbol{x}[i_k{+}p].$$

A repeat is NLE iff it is not LE; NRE iff it is not RE; ***nonextendible*** (NE) iff it is both NLE and NRE. A repeat is ***supernonextendible*** (SNE) iff it is NE and its repeating substring $\boldsymbol{u}$ is not a proper substring of any other repeating substring of $\boldsymbol{x}$.

* The work of the first author was supported by the Australian Research Council, that of the second and third authors by the Natural Sciences & Engineering Research Council of Canada.

Simon J. Puglisi, William F. Smyth, Munina Yusufu : Fast Optimal Algorithms for Computing All the Repeats in a String, pp. 161–169.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

In [8, p. 147] an algorithm is described that, given the suffix tree $\mathrm{ST}_{\boldsymbol{x}}$ of $\boldsymbol{x}$, computes all the NE (called "maximal") *pairs of repeats* in $\boldsymbol{x}$ in time $O(\alpha n+q)$, where q is the number of pairs output. [4] uses similar methods to compute all NE pairs $(p; i_1, i_2)$ such that $i_2-i_1 \geq g_{min}$ (or $\leq g_{max}$) for user-defined ***gaps*** g_{min}, g_{max}. [1] shows how to use the suffix array $\mathrm{SA}_{\boldsymbol{x}}$ of $\boldsymbol{x}$ to compute the NE pairs in time $O(\alpha n+q)$. Since it may be that $\alpha \in O(n)$, all of these algorithms require $O(n^2)$ time in the worst case, though in applications usually $\alpha = 4$ (DNA alphabet). [7] uses the suffix arrays of both $\boldsymbol{x}$ and its reversed string $\overline{\boldsymbol{x}} = \boldsymbol{x}[n]\boldsymbol{x}[n-1]\cdots\boldsymbol{x}[1]$ to compute all the complete NE repeats in $\boldsymbol{x}$ in $\Theta(n)$ time. More recently, [17] describes suffix array-based $\Theta(n)$-time algorithms to compute all ***substring equivalence classes*** — essentially the complete NE repeats — in $\boldsymbol{x}$.

In this paper we first describe an algorithm PSY1 that computes all the complete NE repeats in a given string $\boldsymbol{x}$ whose length (period) $p \geq p_{min}$, where $p_{min} \geq 1$ is a user-specified minimum. PSY1 executes in $\Theta(n)$ time independent of alphabet size and requires $5n$ bytes of storage, plus a stack, but its preprocessing includes suffix array construction that raises the storage requirement to $9n$ bytes. PSY1 is an order of magnitude faster than the complete repeats algorithms described in [7,17].

We also describe a new fast algorithm PSY2 that computes all the complete SNE repeats in $\boldsymbol{x}$ in time $\Theta(n+\alpha)$. This improves on the algorithm described in [8, p. 146] that does the same calculation (of "supermaximal" repeats) in time $O(n \log \alpha)$ using a suffix tree, as well as on the algorithm described in [1, p. 59] that uses a suffix array and requires $O(n+\alpha^2)$ time. For $\alpha \in O(n)$ these times become $O(n \log n)$ and $O(n^2)$, respectively, whereas PSY2 remains $\Theta(n)$.

In Section 2 we describe our algorithms. Section 3 summarizes the results of experiments that compare the algorithms with each other and with existing algorithms. Section 4 discusses these results, including the strategy of computing complete (NE and SNE) repeats in the context of applications to bioinformatics and data compression.

2 Description of the Algorithms

We suppose that a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is given, defined on an ordered alphabet A of size α (where if there is no explicit bound on alphabet size, we suppose $\alpha \leq n$). We refer to the suffix $\boldsymbol{x}[i..n]$, $i \in 1..n$, simply as ***suffix*** i. Then the ***suffix array*** $\mathrm{SA}_{\boldsymbol{x}}$ is an array $[1..n]$ in which $\mathrm{SA}_{\boldsymbol{x}}[j] = i$ iff suffix i is the j^{th} in lexicographical order among all the suffixes of $\boldsymbol{x}$. Let $\mathrm{lcp}_{\boldsymbol{x}}(i_1, i_2)$ denote the ***longest common prefix*** of suffixes i_1 and i_2 of $\boldsymbol{x}$. Then $\mathrm{LCP}_{\boldsymbol{x}}$ is an array $[1..n{+}1]$ in which $\mathrm{LCP}_{\boldsymbol{x}}[1] = \mathrm{LCP}_{\boldsymbol{x}}[n{+}1] = -1$, while for $j \in 2..n$,

$$\mathrm{LCP}_{\boldsymbol{x}}[j] = \Big|\mathrm{lcp}_{\boldsymbol{x}}\big(\mathrm{SA}_{\boldsymbol{x}}[j-1], \mathrm{SA}_{\boldsymbol{x}}[j]\big)\Big|.$$

$\mathrm{SA}_{\boldsymbol{x}}$ can be computed in $\Theta(n)$ worst-case time [9,12], though various supralinear methods [16,14] are certainly much faster, as well as more space-efficient, in practice [18], in some cases requiring space only for $\boldsymbol{x}$ and $\mathrm{SA}_{\boldsymbol{x}}$ itself. Given $\boldsymbol{x}$ and $\mathrm{SA}_{\boldsymbol{x}}$, $\mathrm{LCP}_{\boldsymbol{x}}$ can also be computed in $\Theta(n)$ time [11,15]: the first algorithm described in [15] requires $9n$ bytes of storage and is almost as fast in practice as that of [11], which requires $13n$ bytes. (For space calculations, we make throughout the usual assumption that an integer occupies four bytes, a letter one.) When the context is clear, we write SA for $\mathrm{SA}_{\boldsymbol{x}}$, LCP for $\mathrm{LCP}_{\boldsymbol{x}}$.

We also define the Burrows-Wheeler Transform $\text{BWT}_{\boldsymbol{x}}$ or BWT [5]: for $\text{SA}[j] > 1$, $\text{BWT}[j] = \boldsymbol{x}\big[\text{SA}[j]{-}1\big]$, while for j such that $\text{SA}[j] = 1$, $\text{BWT}[j] = \$$, a sentinel letter not equal to any other in $\boldsymbol{x}$. We set $\text{BWT}[n{+}1] = \$$. BWT can clearly be computed in linear time from SA; since it occupies only n rather than $4n$ bytes, we use BWT rather than SA if there is a choice. Examples of these standard data structures follow:

$$
\begin{array}{rl}
 & \;\;\scriptstyle 1\;\; 2\;\; 3\;\; 4\;\; 5\;\; 6\;\; 7\;\; 8\;\; 9 \\
\boldsymbol{x} = & \;\; a\;\; b\; a\; a\; b\; a\; b\; a\;\; \$ \\
\text{SA}_{\boldsymbol{x}} = & \;\; 8\;\; 3\; 6\; 1\; 4\; 7\; 2\; 5 \\
\text{LCP}_{\boldsymbol{x}} = & \text{-1}\; 1\; 1\; 3\; 3\; 0\; 2\; 2\; \text{-1} \\
\text{BWT}_{\boldsymbol{x}} = & \;\; b\;\; b\; b\; \$\; a\; a\; a\; a\;\; \$
\end{array}
$$

Here as in the Introduction the repeating substring $\boldsymbol{u} = aba$ of length 3 occurs in positions $6, 1, 4$ of $\boldsymbol{x}$; our algorithms report this fact as a complete repeat (it is both NE and SNE) in the form $(3; 3, 5)$ with period $p = 3$, where $3, 5$ is a range identifying $\text{SA}[3] = 6, \text{SA}[4] = 1, \text{SA}[5] = 4$. Note that $p = \text{LCP}[4] = \text{LCP}[5]$.

All of the algorithms described in this paper make direct use of LCP and BWT (or equivalent), but not of SA, and therefore require only $5n$ bytes of storage (plus relatively small stack space in the case of PSY1). However, the calculation [15] of LCP requires SA, a further $4n$ bytes, and so, as noted above, the total space requirement is $9n$. The output of both algorithms is a range $i..j$ of positions in SA that specifies a complete repeat (NE for PSY1, SNE for PSY2).

PSY1

Given a threshold $p_{min} \geq 1$, PSY1 outputs all the complete NE repeats in a given string $\boldsymbol{x}$, each one a triple $(p; i, j)$ specifying a period $p \geq p_{min}$ and a range $i..j$ in SA such that the suffixes $\text{SA}[i], \text{SA}[i{+}1], \ldots, \text{SA}[j]$ form a maximal set with the same longest common prefix of length

$$p\ (lcp) = \text{LCP}[i{+}1] = \text{LCP}[i{+}2] = \cdots = \text{LCP}[j].$$

As shown in Figure 1, PSY1 performs a single left-to-right scan of LCP, inspecting each position j from 1 to n. During the scan, whenever a position lb (initially $lb = j$) is found for which the LCP value increases, an entry is pushed onto a stack LB. LB specifies the Left Boundary lb and period p of a repeat that must be NRE, but that may or may not be NLE: lb marks the leftmost occurrence in SA of a repeating substring of length $p = \text{LCP}[lb{+}1] > \text{LCP}[lb]$, thus the left boundary of a repeat. In fact, a triple (p, lb, bwt) is pushed onto the stack, where bwt is a letter that determines the left-extendibility of the repeat: initially bwt equals the sentinel letter \$ if $\text{BWT}[lb] \neq \text{BWT}[lb{+}1]$, and otherwise equals $\text{BWT}[lb]$. This is the calculation performed repeatedly by the function `LEletter`. Thus $bwt = \$$ if the repeat is NLE (and so eventually should be printed), but assumes a regular letter value if the repeat (so far at least) is LE.

Since the pushes to LB occur in increasing order of position lb, the pops occur in decreasing order of lb: the most recently pushed triple is popped when a position j is reached for which $\text{LCP}[j{+}1] < \texttt{top}(LB).lcp$. Then j is the right boundary for the popped triple $(p, i, prevbwt)$ and a repeat $(p; i, j)$ is identified. Observe that this repeat is NRE: if the same letter followed each occurrence of the repeating substring of length p, then p could not be maximum, contradicting the definition of LCP.

```
    — Preprocessing: compute SA, BWT & LCP
    — in Θ(n) time and 9n bytes of space.
lcp ← LCP[1]; lb ← 1; bwt1 ← BWT[1]
push(LB; lcp, lb, bwt1)
for j ← 1 to n do
    lb ← j; lcp ← LCP[j+1]
      — Compute LEletter of BWT[j] and BWT[j+1].
    bwt2 ← BWT[j+1]; bwt ← LEletter(bwt1, bwt2); bwt1 ← bwt2
    while top(LB).lcp > lcp do
        pop(LB; p, i, prevbwt)
        if prevbwt = $ and p ≥ p_min then
            output(p; i, j)
        lb ← i
        top(LB).bwt ← LEletter(prevbwt, top(LB).bwt)
        bwt ← LEletter(prevbwt, bwt)
    if top(LB).lcp = lcp then
        top(LB).bwt ← LEletter(top(LB).bwt, bwt)
    else
        push(LB; lcp, lb, bwt)

function LEletter(ℓ1, ℓ2)
if ℓ1 = $ or ℓ1 ≠ ℓ2 then return $
else return ℓ1
```

Figure 1. Algorithm PSY1: compute all NE repeats of period $p \geq p_{min}$ as ranges in SA

It remains to determine whether or not the popped triple is NLE. For this the popped value *prevbwt* needs to be inspected to determine whether it is \$ — that is, whether the repeat is NLE, whether it should be output. To ensure that `top`(LB).*bwt* is maintained correctly, we use a simple property of ranges of repeats: two ranges are either disjoint (empty common prefix) or else one range contains the other (common prefix over the longer range). It follows that if `top`(LB).*bwt* = \$ for a contained range, then for every range that encloses it, we must also have `top`(LB).*bwt* = \$. Moreover, if for some letter $\lambda \in A$, a contained range is LE with $bwt = \lambda$, then the enclosing range will be LE only if every other contained range also has $bwt = \lambda$. In PSY1 the correct *bwt* value for the enclosing range is maintained by invoking `LEletter` to update `top`(LB).*bwt* whenever LCP[j+1] $\leq$ `top`(LB).*lcp*. For LCP[j+1] $<$ `top`(LB).*lcp*, `LEletter` is used again to update the current *bwt* based on the *prevbwt* just popped.

In view of this discussion, we claim the correctness of PSY1. Execution time is $\Theta(n)$, since the number of executions of the **while** loop is at most the number of triples pushed onto LB, thus $O(n)$. Space required is $5n$ bytes plus maximum stack size at 9 bytes per entry (four bytes each for *lb* and *lcp*, plus a byte for *bwt*). The largest number of entries in LB is exactly the maximum depth of the suffix tree — in fact n for $\boldsymbol{x} = a^n$ — but expected depth on an alphabet of size $\alpha > 1$ is $2\log_\alpha n$ [10]. Thus even for $\alpha = 2$, expected space for LB is $18\log_\alpha n$ bytes — if $n = 2^{20}$, 360 bytes. On strings arising in practice, LB requires negligible space (Section 4).

PSY2

The SNE ("supermaximal") repeats algorithm described in [1] does not deal explicitly with the problem of determining whether or not a complete super NRE (SNRE) repeat is also SNLE. This determination requires that the left extensions (BWT values) of

```
— Preprocessing: compute SA, LAST & LCP.
j ← 0; p ← −1; q ← 0
while j < n do
    high ← 0
    repeat
        j ← j+1; p ← q; q ← LCP[j+1]
        if q > p then high ← q; i ← j
    until p > q
    if high > 0 and SNLE(i, j, LAST) then
        output(p; i, j)

function SNLE(start, end, LAST)
k ← end−start+1
if k > α then return FALSE
else
    for h ← start+1 to end do
        if h−LAST[h] > start then return FALSE
    return TRUE
```

Figure 2. Algorithm PSY2 with a simplified SNLE function using LAST

the k positions in the repeat be pairwise distinct. The approach apparently proposed by the authors requires at most $\binom{k}{2}$ letter comparisons, where k can be order n, thus leading to $O(n^2)$ time in the worst case. A perhaps more efficient approach would be to use a bit map B$[1..\alpha]$ to determine if any letter in the alphabet has occurred more than once as a left extension over the range of the repeat. However, this would require initializing the α positions in B for each of $O(n)$ candidate repeats, and since possibly $\alpha \in O(n)$, the time required could again be $O(n^2)$. Our proposed algorithm PSY2 (Figure 2) incorporates two improvements, one to decrease execution time in practice, the other to reduce asymptotic complexity to $O(n+\alpha)$.

We observe first that the cardinality k of an SNE repeat cannot exceed the alphabet size α. Thus as shown in function SNLE of Figure 2, a single test suffices to eliminate candidate SNRE repeats of cardinality greater than α, thus substantially reducing processing time in many cases. We now describe a more sophisticated approach that reduces worst-case complexity to $\Theta(n+\alpha)$ with a negligible effect on actual processing time.

Instead of BWT$_{\boldsymbol{x}}$, we compute an array LAST = LAST$[1..n]$ in which for every $j \in 1..n$, LAST$[j]$ is the offset between the BWT letter corresponding to the current position j in SA and the position *jprev* of the rightmost previous occurrence in SA of the same BWT letter — if *jprev* does not exist or if $j-jprev \geq \alpha$, then LAST$[j] \leftarrow \alpha$–1. However, if *jprev* exists and satisfies j–$jprev < \alpha$, we set LAST$[j] \leftarrow j$–$jprev$–1, so that LAST$[j]$ takes values in the range $0..\alpha-2$. See Figure 3. Then when function SNLE processes a possibly supernonextendible repeat consisting of $end-start+1$ substrings of $\boldsymbol{x}$, for every position $h \in start+1..end$, the value of BWT$[h]$ will be unique within the range if and only if $h-$LAST$[h] > start$. See Figure 2.

In general it is possible that the offsets stored in LAST could be integers of size $O(n)$. But offsets of magnitude greater than $\alpha-1$ need not be stored, since if the interval *start..end* actually is an SNE repeat, it can contain no more than α positions. Thus LAST requires the same amount of storage as BWT, which stores letters that are also restricted to be at most $\alpha-1$ in magnitude. The method can be implemented for any finite α, but with the usual convention that each letter in the

```
— Initialize an array storing rightmost positions of each letter.
for ℓ ← 1 to α do
    lastpos[ℓ] ← 0
— Compute LAST in a single left-to-right scan of SA.
α′ ← α−1
for j ← 1 to n do
    i ← SA[j]−1
    if i ← 0 then
        LAST[j] ← α′
    else
        letter ← x[i]; jprev ← lastpos[letter]
        if jprev = 0 or j−jprev ≥ α then
            LAST[j] ← α′
        else
            LAST[j] ← j−jprev−1
        lastpos[letter] ← j
```

Figure 3. Preprocessing for Algorithm PSY2 — computing LAST

alphabet is confined to a single byte ($\alpha \leq 256$), the array LAST becomes an array of bytes, just like BWT. (In fact, in order to take advantage of the CPU cache, our implementation of this algorithm actually computes BWT first, then makes a pass over BWT to convert it into LAST — an approach that turns out to be 2–3 times faster than a straightforward implementation of the preprocessing algorithm.)

3 Experimental Results

Experiments were conducted on a diverse selection of files (see Table 1) chosen from `http://www.cas.mcmaster.ca/~bill/strings/`. Tests were conducted using a 2.6 GHz Opteron 885 processor with 2 GB main memory available, under Red Hat Linux 4.1.2–14. The compiler was `gcc` with the `-O3` option. The run times used were the minima over four runs, not including input/output.

File Type	Name	No. Bytes	Description
highly periodic	fibo35	9,227,465	Fibonacci
	fibo36	14,930,352	Fibonacci
	fss9	2,851,443	run-rich [6]
	fss10	12,078,908	run-rich [6]
random	rand2	8,388,608	$\alpha = 2$
	rand21	8,388,608	$\alpha = 21$
DNA	ecoli	4,638,690	*escherichia coli* genome
	chr22	34,553,758	human chromosome 22
	chr19	63,811,651	human chromosome 19
Genbank protein database	prot-a	16,777,216	sample
	prot-b	33,554,432	sample
English	bible	4,047,392	King James bible
	howto	39,422,105	Linux howto files
	mozilla	51,220,480	Mozilla source code

Table 1. Files used for testing.

Test results are shown in Table 2, where the vertical line separates preprocessing from processing. For SA construction the KS algorithm was used [9] — the fastest such algorithm is perhaps MP2 [14] that, based on experiments documented in [14,18], would perform 5–10 times faster on average, using about $5.2n$ bytes of storage. For LCP construction the algorithm of Kasai *et al.* [11] was used, the fastest one known — according to experiments documented in [15], the first Manzini variant runs almost as fast. Table 2 compares PSY1 with the algorithm of [17]. The algorithm of [7] was not tested because it computes SA twice, and so could not be competitive. Not shown in the table are tests against three variants of PSY1, two of them using heuristics designed to speed up processing, another using a different approach that also achieves $\Theta(n)$ worst case time: on each of the test files listed in Table 1, PSY1 is at least as fast as any of the three. Note that for each program tested, the number of microseconds per letter is generally stable within each file type and not highly variable overall. Averages are not weighted by file size. Tests shown for PSY1 used $p_{min} = 1$; as expected, for larger p_{min} run time was unchanged.

File	SA	LCP	BWT	LAST	PSY1	[17]	PSY2
fibo35	0.898	0.169	0.025	0.031	0.012	0.448	0.009
fibo36	0.886	0.170	0.027	0.033	0.012	0.475	0.007
fss9	0.826	0.154	0.026	0.031	0.014	0.330	0.007
fss10	0.958	0.177	0.025	0.032	0.013	0.469	0.008
periodic AVG	0.892	0.168	0.026	0.032	0.013	0.430	0.008
rand2	0.947	0.188	0.026	0.031	0.017	0.215	0.012
rand21	1.135	0.199	0.025	0.031	0.012	0.122	0.012
random AVG	1.041	0.193	0.025	0.031	0.015	0.169	0.012
ecoli	1.413	0.175	0.025	0.031	0.015	0.155	0.011
chr22	1.635	0.285	0.035	0.040	0.016	0.278	0.012
chr19	1.873	0.333	0.044	0.053	0.016	0.242	0.012
DNA AVG	1.754	0.309	0.035	0.041	0.016	0.225	0.012
prot-a	1.778	0.222	0.027	0.032	0.013	0.211	0.012
prot-b	1.971	0.277	0.034	0.039	0.013	0.247	0.012
protein AVG	1.874	0.249	0.030	0.036	0.013	0.229	0.012
bible	1.417	0.151	0.024	0.030	0.015	0.168	0.012
howto	1.912	0.214	0.035	0.039	0.016	0.219	0.012
mozilla	1.815	0.187	0.032	0.036	0.013	0.139	0.011
English AVG	1.417	0.151	0.024	0.035	0.014	0.175	0.012
AVERAGE	1.390	0.207	0.029	0.035	0.014	0.266	0.011

Table 2. Microseconds per letter used by each run.

4 Discussion

We make the following observations:

* Both new algorithms are very fast, especially on strings that arise in practice: even if SA were to execute 10 times faster, still each algorithm would require less than 5 % of total SA/LCP time.
* Computing LAST for PSY2 requires about 20 % more time than computing BWT for PSY1. Both requirements are small compared to SA/LCP computation time.
* For PSY1 we have computed maximum stack size for each of the test files: for `prot-a` (the worst case) the maximum storage for LB was less than 0.1 % of the $5n$ bytes required for LCP and BWT.

∗ The algorithm of [17] appears to execute 10–15 times slower than PSY1 on real-world files, while requiring $12n$ bytes of storage (SA, inverse of SA, and LCP). (The timing facilities for this algorithm were included in the code kindly provided by the authors.)

∗ Assuming the use of a fast space-efficient SA construction algorithm, LCP construction turns out to be the main obstacle to further improvement, due to both its time and its space requirements.

The output of PSY1 and PSY2 can be used in various ways and for various purposes. For offline data compression the output can be used for phrase selection [2,13,21]. It is also useful for duplicate text/document detection [3]. If the user requires positions in $\boldsymbol{x}$ to be output, this can trivially be achieved, since SA is available, by postprocessing that replaces $i..j$ by $\mathrm{SA}[i], \mathrm{SA}[i{+}1], \ldots, \mathrm{SA}[j]$. In applications to protein sequences, such as the detection of low-complexity regions, the use of either PSY1 or PSY2 will provide significant algorithmic speed-up over currently-proposed methods [19] that are effective but slow. In the context of genome analysis the postprocessing of interest may be to compute NE pairs as in [8,4,1]. Assuming an integer alphabet $1..\alpha$, this can be accomplished as follows for each range $i..j$. Introduce a new array $\mathrm{BWT}' = \mathrm{BWT}'[1..n]$, where for $\mathrm{SA}[h] < n$, $\mathrm{BWT}'[h] = \boldsymbol{x}[\mathrm{SA}[h]{+}1]$, otherwise $\mathrm{BWT}'[h] = \$$.

(1) Perform a radix sort on the pairs

$$(\mathrm{BWT}[i], \mathrm{BWT}'[i]),\ (\mathrm{BWT}[i{+}1], \mathrm{BWT}'[i{+}1]),\ \ldots,\ (\mathrm{BWT}[j], \mathrm{BWT}'[j])$$

into bins that are accessed from an array $\mathrm{B} = \mathrm{B}[1..\alpha, 1..\alpha]$. As a byproduct of the sort, positions in a Boolean array $\mathrm{E} = \mathrm{E}[1..\alpha]$ are set: $\mathrm{E}[b] =$ `TRUE` if and only if row b of B is empty.

(2) For every nonempty row b_1 of B, and for every $b_2 \in 1..\alpha$, perform the following simple processing:

for $h_1 \leftarrow b_1{+}1$ **to** α **do**
  if not $\mathrm{E}[h_1]$ **then**
    for $h_2 \leftarrow (1$ **to** $b_2{-}1)$ **and** $(b_2{+}1$ **to** $\alpha)$ **do**
      output all pairs $\mathrm{B}(b_1, b_2)$ with $\mathrm{B}(h_1, h_2)$

This approach requires checking at most $\alpha^2(\alpha-1)^2/2$ positions in B for each range processed; in the DNA case with $\alpha = 4$, this amounts to at most 72 (that is, $\alpha^3{+}2\alpha$) positions, but will for most ranges be much less. Otherwise the time required is proportional to the number of pairs output. Due to cache effects, we believe this will be an efficient algorithm for computing NE pairs: it depends only on $i, j, \mathrm{BWT}, \mathrm{BWT}'$.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch: *Replacing suffix trees with enhanced suffix arrays.* Journal of Discrete Algorithms, 2(1) 2004, pp. 53–86.
2. A. Apostolico and S. Lonardi: *Off-line compression by greedy textual substitution.* Proceedings of the IEEE, 88(11) 2000, pp. 1733–1744.
3. Y. Berstein and J. Zobel: *Accurate discovery of co-derivative documents via duplicate text detection.* Information Systems, 31 2006, pp. 595–609.
4. G. S. Brodal, R. B. Lyngso, C. N. S. Pederesen, and J. Stoye: *Finding maximal pairs with bounded gap.* Journal of Discrete Algorithms, 1 2000, pp. 77–103.

5. M. Burrows and D. J. Wheeler: *A block sorting lossless data compression algorithm*, Tech. Rep. 124, Digital Equipment Corporation, Palo Alto, California, 1994.
6. F. Franek, J. Simpson, and W. F. Smyth: *The maximum number of runs in a string*, in Proceedings of the 14th Australasian Workshop on Combinatorial Algorithms, M. Miller and K. Park, eds., Seoul, Korea, 2003, pp. 36–45.
7. F. Franek, W. F. Smyth, and Y. Tang: *Computing all repeats using suffix arrays*. Journal of Automata, Languages and Combinatorics, 8(4) 2003, pp. 579–591.
8. D. Gusfield: *Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology*, Cambridge University Press, Cambridge, United Kingdom, 1997.
9. J. Kärkkäinen and P. Sanders: *Simple linear work suffix array construction*, in Proceedings of the 30th International Colloquium Automata, Languages and Programming, vol. 2971 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2003, pp. 943–955.
10. S. Karlin, G. Ghandour, F. Ost, S. Tavare, and L. J. Korn: *New approaches for computer analysis of nucleic acid sequences*. Proceedings of the National Academy of Science, 80(18) September 1983, pp. 5660–5664.
11. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, A. Amir and G. M. Landau, eds., vol. 2089 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2001, pp. 181–192.
12. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*, in Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, R. Baeza-Yates, E. Chávez, and M. Crochemore, eds., vol. 2676 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2003, pp. 200–210.
13. J. Larsson and A. Moffat: *Off-line dictionary-based compression*. Proceedings of the IEEE, 88(11) 2000, pp. 1722–1732.
14. M. A. Maniscalco and S. J. Puglisi: *Faster lightweight suffix array construction*, in Proceedings of 17th Australasian Workshop on Combinatorial Algorithms, J. Ryan and Dafik, eds., 2006, pp. 16–29.
15. G. Manzini: *Two space saving tricks for linear time LCP computation*, in Proceedings of 9th Scandinavian Workshop on Algorithm Theory (SWAT '04), T. Hagerup and J. Katajainen, eds., vol. 3111 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2004, pp. 372–383.
16. G. Manzini and P. Ferragina: *Engineering a lightweight suffix array construction algorithm*. Algorithmica, 40 2004, pp. 33–50.
17. K. Narisawa, S. Inenaga, H. Bannai, and M. Takeda: *Efficient computation of substring equivalence classes with suffix arrays*, in Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching, B. Ma and K. Zhang, eds., vol. 4580 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2007, pp. 340–351.
18. S. J. Puglisi, W. F. Smyth, and A. Turpin: *A taxonomy of suffix array construction algorithms*. ACM Computing Surveys, 39(2) 2007, pp. 1–31.
19. S. W. Shin and S. M. Kim: *A new algorithm for detecting low-complexity regions in protein sequences*. Bioinformatics, 21(2) 2005, pp. 160–170.
20. B. Smyth: *Computing Patterns in Strings*, Pearson Addison-Wesley, Essex, England, 2003.
21. A. Turpin and W. F. Smyth: *An approach to phrase selection for offline data compression*, in Proceedings of the 25th Australasian Computer Science Conference, M. Oudshoorn, ed., 2000, pp. 267–273.

New Efficient Bit-Parallel Algorithms for the δ-Matching Problem with α-Bounded Gaps in Musical Sequences

Domenico Cantone, Salvatore Cristofaro, and Simone Faro

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

Abstract. We present new efficient variants of the (δ, α)-Sequential-Sampling algorithm, recently introduced by the authors, for the δ-approximate string matching problem with α-bounded gaps. These algorithms, which have practical applications in music information retrieval and analysis, make use of the well-known technique of bit-parallelism. An extensive comparison with the most efficient algorithms present in the literature for the same search problem shows that our newly proposed solutions achieve very good results in practice, in terms of both space and time complexity, and, in most cases, they outperform existing algorithms.

Keywords: approximate string matching with gaps, bit-parallel algorithms, music information retrieval

1 Introduction

The δ-approximate string matching problem with α-bounded gaps [5,4,2] is a generalization of the δ-approximate string matching problem [1] and arise in many questions in music information retrieval and music analysis. This is particularly true in the context of monophonic music, where one wants to retrieve occurrences of a given melody from a complex musical score.

We recall that two (monophonic) musical sequences have a δ-approximate matching if they have the same length (i.e., they contain the same number of notes) and notes at the same positions differ by at most δ semitones. Then, we say that a melody (or pattern) P has a δ-approximate occurrence with α-bounded gaps within a musical score (or text) T (or, more shortly, a (δ, α)-occurrence), if the melody has a δ-approximate matching with a subsequence of the musical score in which it is allowed to skip up to a fixed number α of notes (the gap) between any two consecutive positions. Thus, δ-approximate matching with α-bounded gaps turns out to be very effective for finding closely related but not necessarily identical occurrences of melodies (δ-approximation), when small values of δ are allowed. In addition, the gaps allow to skip over various kinds of musical ornamentations (e.g., arpeggios) which are of common use, especially in classical music. See Figure 1 for a pictorial illustration.

We mention also that many variants and generalizations of the δ-approximate string matching problem with α-bounded gaps have been considered for applications in other fields other than music, such as, for instance, molecular biology [9,10].

The paper is organized as follows. In the next section we introduce some basic notations and give a formal definition of the δ-approximate string matching problem with α-bounded gaps. In Section 3 we review some of the most efficient algorithms for this problem. Then, in Section 4, we describe our newly proposed algorithms.

Domenico Cantone, Salvatore Cristofaro, Simone Faro: New Efficient Bit-Parallel Algorithms for the δ-Matching Problem with α-Bounded Gaps in Musical Sequences, pp. 170–184.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

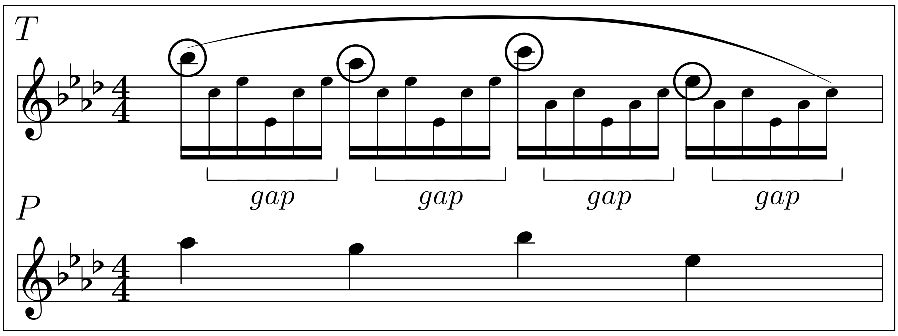


Figure 1. An excerpt from study *Op. 25 Nr. 1 for Piano Solo* by F. Chopin (first score). Melody P has a δ-approximate occurrence with α-bounded gaps in T, for $\delta \geq 2$ and $\alpha \geq 5$, indicated by the circled notes. Tiny notes represent arpeggios and form the gaps. Notice that in this case the gaps are all of the same size 5. Observe also that the first and the third note of P differ from the corresponding matchings in T (circled notes) by 2 semitones; the second note differ by 1 semitone, while the last note equals its matching. In any case, the difference between a note and its matching does not exceed 2 semitones, so that we have a (δ, α)-occurrence of P in T, for any $\delta \geq 2$ and $\alpha \geq 5$.

In Section 5, we report the experimental results of an extensive comparison of our algorithms with some of the most efficient ones present in the literature. Finally, in Section 6 we draw our conclusions.

2 Basic Definitions and Properties

Before entering into details, we review a bit of notations and terminology. We represent a string P as a finite array $P[0\,..\,m-1]$, with $m \geq 0$. In such a case we say that P has length m and write $|P| = m$. In particular, for $m = 0$ we obtain the EMPTY STRING. By $P[i]$ we denote the $(i+1)$-st symbol of P, with $0 \leq i < |P|$, provided that $|P| > 0$. Likewise, by $P[i\,..\,j]$ we denote the substring of P contained between the $(i+1)$-st and the $(j+1)$-st symbols of P (both inclusive), where $0 \leq i \leq j < |P|$. The substrings of the form $P[0\,..\,j]$, also denoted by P_j, with $0 \leq j < |P|$, are the nonempty PREFIXES of P.

Let Σ be a finite alphabet of integer numbers and let δ and α be nonnegative integers. Two symbols a and b of Σ are said to be δ-APPROXIMATE, in which case we write $a =_\delta b$, if $|a - b| \leq \delta$. Given a pattern P of length m and a text T of length n over the alphabet Σ, by a δ-APPROXIMATE OCCURRENCE WITH α BOUNDED GAPS OF P IN T, or simply a (δ, α)-OCCURRENCE OF P IN T, we mean a sequence $(i_0, i_1, \ldots, i_{m-1})$ of indices such that

(1) $0 \leq i_0 < i_1 < \cdots < i_{m-1} < n$,
(2) $T[i_j] =_\delta P[j]$, for $0 \leq j < m$, and
(3) $i_h - i_{h-1} \leq \alpha + 1$, for $0 < h < m$, provided that $m > 1$.

Given an index i, with $0 \leq i < n$, a (δ, α)-OCCURRENCE OF P AT POSITION i IN T is a (δ, α)-occurrence $(i_0, i_1, \ldots, i_{m-1})$ of P in T such that $i_{m-1} = i$. We write $P \trianglelefteq^i_{\delta,\alpha} T$ to mean that there is a (δ, α)-occurrence of P at position i in T (in fact, when the bounds δ and α are well understood from the context, one can simply write $P \trianglelefteq^i T$).

The δ-APPROXIMATE STRING MATCHING PROBLEM WITH α-BOUNDED GAPS, or (δ, α)-MATCHING PROBLEM, is the problem of finding the (δ, α)-occurrences of a given pattern P in a given text T. More precisely, the following variants may be considered [2]: (a) find all (δ, α)-occurrences of P in T; (b) find all positions i in T such that $P \trianglelefteq^i_{\delta,\alpha} T$; (c) for each position i in T, find the number of all distinct (δ, α)-occurrences of P at position i in T. In this paper we will concentrate only on variant (b).

The following property is an immediate consequence of the above definitions:

Lemma 1. *Let P and T be respectively a pattern of length m and a text of length n over an alphabet Σ of integer numbers. Moreover, let δ and α be nonnegative integers. Then,*

(a) $P_0 \trianglelefteq^i_{\delta,\alpha} T \Leftrightarrow T[i] =_\delta P[0]$;
(b) $P_j \trianglelefteq^i_{\delta,\alpha} T \Leftrightarrow T[i] =_\delta P[j]$ AND $(\exists k \in \{1, \ldots, \alpha+1\} : i - k \geq 0$ AND $P_{j-1} \trianglelefteq^{i-k}_{\delta,\alpha} T)$, *for* $0 \leq i < n$ *and* $0 < j < m$.

The following notations and terminology will be used in connection with the bit-parallelism technique. A BIT MASK (or BINARY STRING) is a string whose symbols are the two bits $\mathsf{0}$ and $\mathsf{1}$. In writing bit masks, we will use exponentiation to denote the concatenation of multiple copies of single bits or of bit masks as well. Thus, for instance, $\mathsf{101^30}$ denotes the bit mask $\mathsf{101110}$ and $\mathsf{1(01)^30}$ denotes $\mathsf{10101010}$.

We will employ the following standard operations on bit masks: the bit-wise and and bit-wise or operations, denoted respectively by & and $|$, and the right-shift and left-shift operations, denoted respectively by $\gg$ and $\ll$. We will also use the arithmetic operations of addition "+" and subtraction "−" between bit masks to calculate, respectively, the binary representations of the sum and of the difference between the nonnegative integers represented by the bit masks. It turns out that in all expressions of the form $X - Y$ which we will encounter in the rest of the paper, the nonnegative integer represented by the bit mask X is always no less than the integer represented by Y. Likewise, in all expressions of the form $X \,\&\, Y$, $X \,|\, Y$, $X + Y$, and $X - Y$, the two bit masks X and Y will have the same length, so that we will not need to deal with special cases. Notice that if X and Y are bit masks of the same length ℓ, then the length of the bit masks $X \,\&\, Y$, $X \,|\, Y$, and $X - Y$ is ℓ, whereas the length of $X + Y$ might be $\ell + 1$, due to the carry bit.

Concerning the unitary left-shift operation, we will assume that the string $X \ll 1$ has the same length as X, if the leading bit of X is $\mathsf{0}$ (which corresponds to dropping from X its leading bit $\mathsf{0}$), otherwise its length is one more than that of X. The k-ary left-shift operation is then defined as k iterations of the unitary left-shift. Instead, the right-shift is defined in such a way that $X \gg k$ has always the same length of X. Thus, for instance, if $X = \mathsf{00110}$ we have: $X \ll 1 = \mathsf{01100}$, $X \ll 2 = \mathsf{11000}$, $X \ll 3 = \mathsf{110000}$, $X \gg 1 = \mathsf{00011}$, $X \gg 2 = \mathsf{00001}$, $X \gg 3 = X \gg 4 = \cdots = \mathsf{00000}$.

As far as concerns complexity issues, we will assume the computational model in which each of the above operations can be executed in $\mathcal{O}(\lceil L/w \rceil)$-space and time, where L is the length of the result and w is the computer word length. In fact, a bit mask B whose length exceeds the computer word length w can be readily represented by $\lceil |B|/w \rceil$ computer words.

The following additional notations will also be used. Given a matrix $\mathcal{M}$ of dimensions $h \times k$, we denote by $(\mathcal{M})_{i,j}$ the element of $\mathcal{M}$ located in the intersection of the $(i+1)$-st row and $(j+1)$-st column of $\mathcal{M}$, for $0 \leq i < h$ and $0 \leq j < k$. A bit-matrix is a matrix whose entries belong to $\{\mathsf{0}, \mathsf{1}\}$. Given two integers h and k, with $h \leq k$, we denote by $[h \,..\, k]$ the set (interval) of all integers x such that $h \leq x \leq k$.

In the sequel, we will assume that all patterns and texts in the paper are strings over an alphabet Σ of size $\sigma > 0$, having the form $\{0, 1, \ldots, \sigma - 1\}$.

3 Efficient algorithms for (δ, α)-matching

The δ-approximate string matching problem with α-bounded gaps has been first formally defined in [5], where the δ-Bounded-Gaps algorithm has been proposed (see also [4,2]). The δ-Bounded-Gaps algorithm, whose time and space complexity is $\mathcal{O}(nm)$, with n and m the lengths of the text T and of the pattern P respectively, is presented as an incremental procedure, based on the dynamic programming approach. Scanning the pattern P from left to right, the δ-Bounded-Gaps algorithm looks for the (δ, α)-occurrences of each prefix P_j of the pattern P in the whole text T, for $0 \leq j < m$. Specifically, the δ-Bounded-Gaps algorithm proceeds by filling in a table D of dimensions $m \times n$ such that $D[j, i] = \mathbf{max}(\{k \geq 0 : i - \alpha \leq k \leq i \text{ and } P_j \trianglelefteq^k T\} \cup \{-1\})$, for $0 \leq j < m$ and $0 \leq i < n$. Notice that $P_j \trianglelefteq^i T$ if and only if $D[j, i] = i$, for $0 \leq j < m$ and $0 \leq i < n$.

An algorithm, slightly more efficient than the δ-Bounded-Gaps, has been presented by the authors in [2], under the name (δ, α)-Sequential-Sampling. As in the case of the δ-Bounded-Gaps algorithm, also the (δ, α)-Sequential-Sampling is based on dynamic programming, but it follows a different computation ordering than the δ-Bounded-Gaps algorithm does; more precisely, it scans the text T from left to right and for each position i of T it looks for the (δ, α)-occurrences at position i of all prefixes of the pattern P. The (δ, α)-Sequential-Sampling algorithm has an $\mathcal{O}(nm)$ running time and requires $\mathcal{O}(m\alpha)$-space. A much more efficient variant of it is the (δ, α)-Tuned-Sequential-Sampling algorithm, which has an average case running time of $\mathcal{O}(n)$, in the case in which α is assumed constant (cf. [3]).

Another algorithm, named (δ, α)-Shift-And, has also been described in [3]. The (δ, α)-Shift-And algorithm is a very simple variant of a forward search algorithm presented in [9] for a pattern matching problem with gaps and character classes, particularly suited for applications to protein searching. It uses bit-parallelism to simulate the behavior of a nondeterministic finite automaton with ε-transitions. The automaton has $\ell = (\alpha + 1)(m - 1) + 2$ states, and the simulation is carried out by representing it as a bit mask B of length $\ell - 1$ (the initial state of the automaton need not be represented in the bit mask since it is always active during the computation). When $\ell < w$ (the computer word length), the entire bit mask B fits in a single computer word. In this case the (δ, α)-Shift-And algorithm becomes extremely fast in practice.

Other efficient algorithms for the (δ, α)-matching problem have been presented more recently in [6] and [7]. In particular, [6] presents two algorithms, called DA-bpdb and DA-mloga-bits. The first one inherits the basic idea from the dynamic programming algorithm δ-Bounded-Gaps presented in [4]. It uses bit-parallelism to compute an $m \times n$ bit-matrix $\mathcal{D}$ such that $(\mathcal{D})_{j,i} = 1$ if and only if $P_j \trianglelefteq^i T$, for $0 \leq j < m$ and $0 \leq i < n$. Basically, the algorithm DA-bpdb partitions each row of the matrix $\mathcal{D}$ as a sequence of $\lceil n/w \rceil$ consecutive bit masks, each of which represents a group of w bits on that row. Then, the computation of the j-th bit mask in row i is performed bit-parallely by using the $(j - 1)$-st and the j-th bit masks of the $(i - 1)$-st row. It turns out that DA-bpdb has an $\mathcal{O}(n\delta + \lceil n/w \rceil m)$ worst-case execution time, which becomes $\mathcal{O}(\lceil n/w \rceil \lceil \alpha\delta/\sigma \rceil + n)$ on the average. The second algorithm, namely DA-mloga-bits, is based on a compact representation, in the form of a systolic array, of the nondeterministic automaton used in the algorithm (δ, α)-Shift-And. The systolic array is composed of m building blocks, called *counters* in [6], one for each symbol of the pattern, and is represented as a bit mask of length $(m-1)(\lceil \log_2(\alpha+1)\rceil+1)+1$. Notice that this improves the representations used in [9,3] in which $(\alpha+1)(m-1)+1$ bits are needed to represent the automaton. It turns out that the DA-mloga-bits algorithm has an $\mathcal{O}(n\lceil (m\log_2\alpha)/w\rceil)$ worst-case searching time.

The algorithms presented in [7], called SDP-rows, SDP-columns, SDP-simple, and SDP-simple-compute-L_0, use different computation orderings, in combination with sparse dynamic programming techniques, to implement the calculation of the table D above. Specifically, in the case of the SDP-rows algorithm, the computation is performed row-wise, whereas a column-wise computation is used by SDP-columns. The algorithm SDP-simple, which can be considered as a brute force variant of SDP-rows, performs very well in practice, especially for small values of δ and α; SDP-simple-compute-L_0 improves the average case running time of SDP-simple by using a Boyer-Moore-Horspool-like shifting strategy [8], suitably adapted to handle gaps. In particular, the latter two algorithms turn out to be among the most efficient ones, in terms of running time, in many practical cases, especially for small values of α, as shown in [7]. However, although these algorithms are very fast in practice, they require additional $\mathcal{O}(n)$-space, plus $\mathcal{O}(\sigma)$-space in the case of SDP-simple-compute-L_0.

4 New efficient variants of the (δ,α)-Sequential-Sampling algorithm

In this section we present four efficient variants of the algorithm (δ,α)-Sequential-Sampling, all based on bit-parallelism. In particular, one of these variants, the (δ,α)-Tuned-Sequential-Sampling-HBP algorithm, is extremely efficient in most practical cases and outperforms both algorithms SDP-simple and SDP-simple-compute-L_0. Also, the variant (δ,α)-Sequential-Sampling-BP$^+$ turns out to be faster than existing algorithms (e.g., (δ,α)-Shift-And) in the case of short patterns and very small values of the gap α.

We begin by describing the general approach.

Given a text T of length n and a pattern P of length m, let $\mathcal{M}_i$ be the bit-matrix of dimensions $(\alpha+1)\times m$ such that

$$(\mathcal{M}_i)_{k,j} = \begin{cases} 1, & \text{if } i-\alpha+k \geq 0 \text{ AND } P_j \trianglelefteq^{i-\alpha+k} T \\ 0, & \text{otherwise}, \end{cases}$$

for $-1 \leq i < n$, $0 \leq j < m$ and $0 \leq k \leq \alpha$. Notice that, for $0 \leq i < n$ and $0 \leq j < m$, we have $P_j \trianglelefteq^i T$ if and only if $(\mathcal{M}_i)_{\alpha,j} = 1$. Thus, the problem of determining the positions i of T at which $P \trianglelefteq^i T$ holds, translates into the problem of determining all values i such that $(\mathcal{M}_i)_{\alpha,m-1} = 1$, which in turn reduces to the problem of effectively computing the matrices $\mathcal{M}_{-1}, \mathcal{M}_0, \ldots, \mathcal{M}_{n-1}$. This can be done as follows. To begin with, notice that, by the very definition of the matrices $\mathcal{M}_{-1}, \mathcal{M}_0, \ldots, \mathcal{M}_{n-1}$, we have

$$(\mathcal{M}_i)_{k,j} = (\mathcal{M}_{i-1})_{k+1,j}\,, \tag{1}$$

for $0 \leq i < n$, $0 \leq j < m$ and $0 \leq k < \alpha$; i.e., the first α rows of $\mathcal{M}_i$ coincide with the last α rows of $\mathcal{M}_{i-1}$. In addition, by Lemma 1, we have also that

$$(\mathcal{M}_i)_{\alpha,j} = \begin{cases} 1, & \text{if } T[i] =_\delta P[j] \text{ AND} \\ & \quad (j=0 \text{ OR } (\exists k \in \{0,\ldots,\alpha\} : (\mathcal{M}_{i-1})_{k,j-1} = 1)) \\ 0, & \quad \text{otherwise}, \end{cases} \tag{2}$$

for $0 \le i < n$ and $0 \le j < m$, which expresses the $(j+1)$-st item in the last row of matrix $\mathcal{M}_i$ in terms of the j-th column of matrix $\mathcal{M}_{i-1}$. These recursive relations, coupled with the initial condition $\mathcal{M}_{-1} = \mathsf{0}^{(\alpha+1)\times m}$, allow one to compute the matrices $\mathcal{M}_0, \mathcal{M}_1, \dots, \mathcal{M}_{n-1}$ in an iterative fashion, starting from the initial matrix $\mathcal{M}_{-1}$.

For instance, in the case of the (δ,α)-Sequential-Sampling algorithm, the computation is carried out by calculating in sequence the matrices $\mathcal{M}_{-1}, \mathcal{M}_0, \dots, \mathcal{M}_{n-1}$, which are maintained in a circular fashion in a bit table M of dimensions $(\alpha+1)\times m$. More specifically, initially the table M is filled in with all $\mathsf{0}$'s (which corresponds to the initial matrix $\mathcal{M}_{-1}$). Then, n iterations are performed, for $i = 0, 1, \dots, n-1$. At iteration i, the last row of $\mathcal{M}_i$ is computed, by calculating in turn the elements $(\mathcal{M}_i)_{\alpha,m-1}, (\mathcal{M}_i)_{\alpha,m-2}, \dots, (\mathcal{M}_i)_{\alpha,0}$ according to recurrence (2), and stored at the row of index $i \bmod (\alpha+1)$ of table M; thus, just after step i, we have that $\mathsf{M}[(i+k+1) \bmod (\alpha+1), j] = (\mathcal{M}_i)_{k,j}$, for $0 \le k \le \alpha$ and $0 \le j < m$. In performing such step, the (δ,α)-Sequential-Sampling algorithm makes use of an additional array C, of length m, whose $(j+1)$-st entry $C[j]$ is used to count the number of $\mathsf{1}$'s in the $(j+1)$-st column of M, for $0 \le j < m$. This allows to perform each step of the computation in $\mathcal{O}(m)$-time, yielding an overall running time of $\mathcal{O}(nm)$.[1]

The computation of the matrices $\mathcal{M}_0, \mathcal{M}_1, \dots, \mathcal{M}_{n-1}$ can be carried out in various ways using the bit-parallelism technique, as we show next.

The basic idea is to represent each column of the matrices $\mathcal{M}_i$ as a bit mask of length $\alpha+1$ (which is very natural, since the columns of $\mathcal{M}_i$ are nothing but vectors of bits). Consequently, the whole matrix $\mathcal{M}_i$ can be represented as an array of m bit masks, each of which corresponds to a column of $\mathcal{M}_i$, and each of which fits in a single computer word in the case that $\alpha < w$, where w is the computer word length (see below for a brief discussion on the condition $\alpha < w$).[2]

To be more precise, let us denote with $\mathcal{C}_i^{(j)}$ the bit mask of length $\alpha+1$ such that $\mathcal{C}_i^{(j)}[k] = (\mathcal{M}_i)_{k,j}$, for $-1 \le i < n$, $0 \le j < m$, and $0 \le k \le \alpha$.[3] Then, by (1), we have that $\mathcal{C}_i^{(j)}[0\,..\,\alpha-1] = \mathcal{C}_{i-1}^{(j)}[1\,..\,\alpha]$, i.e., the first α bits of $\mathcal{C}_i^{(j)}$ coincide with the last α bits of $\mathcal{C}_{i-1}^{(j)}$. Moreover, by (2), we have that the last bit of $\mathcal{C}_i^{(j)}$ is $\mathsf{1}$, if $T[i] =_\delta P[j]$ and $\mathcal{C}_{i-1}^{(j-1)} \neq \mathsf{0}^{\alpha+1}$; otherwise it is $\mathsf{0}$, provided that $j > 0$. If $j = 0$, the last bit of $\mathcal{C}_i^{(0)}$ is $\mathsf{1}$ if and only if $T[i] =_\delta P[0]$ holds. Therefore, if we put $I = \mathsf{01}^\alpha$, we obtain

$$\mathcal{C}_i^{(j)} = \begin{cases} ((\mathcal{C}_{i-1}^{(j)} \,\&\, I) \ll 1) \,|\, \mathsf{0}^\alpha\mathsf{1}, & \text{if } T[i] =_\delta P[j] \text{ AND } (j = 0 \text{ OR } \mathcal{C}_{i-1}^{(j-1)} \neq \mathsf{0}^{\alpha+1}) \\ (\mathcal{C}_{i-1}^{(j)} \,\&\, I) \ll 1, & \text{otherwise}, \end{cases} \quad (3)$$

for $0 \le i < n$ and $0 \le j < m$. Such relations suggest the simple algorithm reported in Figure 2, named (δ,α)-Sequential-Sampling-HBP, which uses an array C of length m to maintain the bit masks $\mathcal{C}_i^{(0)}, \mathcal{C}_i^{(1)}, \dots, \mathcal{C}_i^{(m-1)}$.[4] This algorithm is very close in spirit to

[1] We mention here that the (δ,α)-Sequential-Sampling algorithm in its original form presented in [2] allows one to count the number of all distinct (δ,α)-approximate occurrences of each prefix P_j of the pattern P at any position i of the text T, and not only to check whether $P_j \trianglelefteq^i T$.

[2] Notice that a similar idea of packing the columns of a bit-matrix into computer words has been already introduced by the authors in [3], in connection with the algorithm (δ,α)-Tuned-Sequential-Sampling. Here, we have further refined it.

[3] Notice that $P \trianglelefteq^i T$ holds if and only if the the last bit of $\mathcal{C}_i^{(m-1)}$ (i.e., $\mathcal{C}_i^{(m-1)}[\alpha]$) is a $\mathsf{1}$, which corresponds to the condition that $\mathcal{C}_i^{(m-1)} \,\&\, \mathsf{0}^\alpha\mathsf{1} \neq \mathsf{0}^{\alpha+1}$.

[4] In the pseudo-code of Figure 2 it is plainly assumed that the bit masks $\mathsf{0}^{\alpha+1}$ and $\mathsf{0}^\alpha\mathsf{1}$ are supplied as constants. Concerning, instead, the bit mask I, notice that it can be computed, e.g., as $I =$

```
(δ, α)-Sequential-Sampling-HBP(P, m, T, n, δ, α)
 1.  for i := 0 to m − 1 do
 2.     C[i] := 0^{α+1}
 3.  I := 01^α
 4.  for i := 0 to n − 1 do
 5.     for j := m − 1 downto 1 do
 6.        C[j] := (C[j] & I) ≪ 1
 7.        if T[i] =_δ P[j] AND C[j − 1] ≠ 0^{α+1}
 8.        then C[j] := C[j] | 0^α1
 9.     C[0] := (C[0] & I) ≪ 1
10.     if T[i] =_δ P[0] then
11.        C[0] := C[0] | 0^α1
12.     if (C[m − 1] & 0^α1) ≠ 0^{α+1} then
13.        print(i)
```

```
(δ, α)-Tuned-Sequential-Sampling-HBP(P, m, T, n, δ, α)
for i := 0 to m − 1 do C[i] := 0^{α+1}
next[0] := next[m] := m
I := 01^α
for i := 0 to n − 1 do
   p := m
   j := next[p]
   while j < m do
      if j < m − 1 AND T[i] =_δ P[j + 1] then
         C[j + 1] := C[j + 1] | 0^α1
         if p > j + 1 then
            next[p] := j + 1
            next[j + 1] := j
            p := j + 1
      C[j] := (C[j] & I) ≪ 1
      if C[j] = 0^{α+1} then next[p] := next[j]
      else p := j
      j := next[p]
   if T[i] =_δ P[0] then
      C[0] := C[0] | 0^α1
      if p > 0 then next[p] := 0
   if (C[m − 1] & 0^α1) ≠ 0^{α+1} then print(i)
```

Figure 2. The (δ,α)-Sequential-Sampling-HBP algorithm (on the left) and the (δ,α)-Tuned-Sequential-Sampling-HBP algorithm (on the right) for the δ-approximate string matching problem with α-bounded gaps.

the (δ,α)-Sequential-Sampling, improving the space complexity of the latter algorithm to $\mathcal{O}(m\lceil\alpha/w\rceil)$, though its running time, which is $\mathcal{O}(nm\lceil\alpha/w\rceil)$, is worse than that of the (δ,α)-Sequential-Sampling algorithm. The reason is that, in general, we need $\lceil(\alpha+1)/w\rceil$ computer words to represent a bit mask of length $\alpha+1$. Consequently, any update of the entry $C[j]$ costs $\mathcal{O}(\lceil\alpha/w\rceil)$-time, for $j=0,1,\ldots,m-1$. However, we notice that in almost all practical applications in music information retrieval the value of the gap bound α is at most 10 (or less), therefore smaller than the size w of a computer word (which is 32 or 64). Thus, in practice, a bit mask of length $\alpha+1$ can be maintained in a single computer word and in this case it turns out that the (δ,α)-Sequential-Sampling-HBP algorithm is faster than the (δ,α)-Sequential-Sampling.

Now, by using a trick similar to the one employed in the (δ,α)-Tuned-Sequential-Sampling algorithm, we obtain a variant of the (δ,α)-Sequential-Sampling-HBP which performs extremely well in practice, as will be shown by extensive experimentation in the next section.

As in the case of the (δ,α)-Tuned-Sequential-Sampling algorithm, we observe that, during each step of the computation of the (δ,α)-Sequential-Sampling-HBP algorithm, an iteration of the **for-loop** at line 5 relative to a value of $j>0$ has no effect if the items $C[j]$ and $C[j-1]$ are both null, i.e., if $C[j]=C[j-1]=0^{\alpha+1}$. In fact, the only items of the array C which need to be updated are the $C[j]$'s such that $C[j]\neq 0^{\alpha+1}$ or (if $j>0$) $C[j-1]\neq 0^{\alpha+1}$. Therefore, it is enough to scan only those positions j of the array C such that $C[j]\neq 0^{\alpha+1}$. Thus, for each such j, we first check whether $T[i]=_\delta P[j+1]$, provided that $j<m-1$, and, if this is the case, we update the entry $C[j+1]$ by assigning to it the bit mask $C[j+1]\,|\,0^\alpha 1$. After that, $C[j]$ is updated as in line 6 of the (δ,α)-Sequential-Sampling-HBP algorithm. To perform such process, the positions j of the nonnull items of C (i.e., the j's such that $C[j]\neq 0^{\alpha+1}$) are

$(0^\alpha 1 \ll \alpha) - 0^\alpha 1$. Similar considerations will hold for the remaining algorithms to be presented in this section.

maintained into an ordered, linked list $\mathcal{L}$, which is scanned from the highest value of j up to the lowest one. The resulting algorithm, named (δ,α)-Tuned-Sequential-Sampling-HBP, is reported in Figure 2. Notice that the list $\mathcal{L}$ is implemented as a circular array, $next$, of length $m+1$, whose last entry, $next[m]$, is used as a pointer to the location which contains the first (i.e., highest) element of $\mathcal{L}$ (or $next[m] = m$, in the case the list $\mathcal{L}$ is empty).

By a simple inspection, it is immediate to verify that the (δ,α)-Tuned-Sequential-Sampling-HBP algorithm has an $\mathcal{O}(nm\lceil\alpha/w\rceil)$ worst-case running time and requires $\mathcal{O}(m\lceil\alpha/w\rceil)$-space. Moreover, by arguing as in [3], it can be shown that the running time of the (δ,α)-Tuned-Sequential-Sampling-HBP algorithm is $\mathcal{O}(n)$ on the average (for a fixed α).

Notice that a slightly simpler variant of the (δ,α)-Tuned-Sequential-Sampling-HBP algorithm could be obtained if we maintained into the array C the reverses of the bit masks $\mathcal{C}_i^{(0)}, \mathcal{C}_i^{(1)}, \ldots, \mathcal{C}_i^{(m-1)}$, rather than the bit masks themselves. In essence, this would involve replacing each left-shift by a right-shift. More precisely, we would have to replace the instruction of line 14 by the assignment $C[j] := C[j] \gg 1$ (thus avoiding to perform any operation prior to the shift) and the instructions of lines 9 and 19 by the assignments $C[j+1] := C[j+1] \,|\, 10^{\alpha}$ and $C[0] := C[0] \,|\, 10^{\alpha}$, respectively. Also, the condition in the **if**-statement of line 21 would need to be replaced by the condition "$(C[m-1] \,\&\, 10^{\alpha}) \neq 0^{\alpha+1}$". The above modifications would have the effect to slightly reduce the number of operations performed during each step of the computation.

Observe also that the last entry $C[m-1]$ of the array C is used by the (δ,α)-Tuned-Sequential-Sampling-HBP algorithm only in the conditional test of line 21. Therefore we do not need to maintain it, since such a test could be implicitly performed during the execution of the **while-loop** of lines 7-17 as follows. If during the execution of the **while-loop** the variable j assumes the value $m-2$ (which means that position $m-2$ is in the list $\mathcal{L}$, i.e., $C[m-2]$ is nonnull), then we check whether $T[i] =_{\delta} P[m-1]$ and, if this is the case, the value i can be directly reported as the position of a (δ,α)-occurrence of the pattern P in the text T. Otherwise, if the variable j does not ever take the value $m-2$ during the execution of the **while-loop**, then the pattern P can have no (δ,α)-occurrence at position i in the text, and therefore, even in this case, the test at line 21 does not need to be checked. It turns out that the variation just outlined slightly improves the overall running time of the algorithm.

In the last variant of the (δ,α)-Sequential-Sampling algorithm, which we are going to describe (actually a variant of the (δ,α)-Sequential-Sampling-HBP algorithm), each matrix $\mathcal{M}_i$ is represented as a single bit mask of length $L = (\alpha+1)m$, obtained by concatenating the bit masks corresponding to the columns of $\mathcal{M}_i$ (i.e., the bit masks $\mathcal{C}_i^{(j)}$). More precisely, the following bit mask is used as a representation of the matrix $\mathcal{M}_i$, for $-1 \leq i < n$:[5]

$$\mathcal{B}_i = \mathcal{C}_i^{(m-1)} \mathcal{C}_i^{(m-2)} \cdots \mathcal{C}_i^{(0)} .$$

Assuming such representation for the matrices $\mathcal{M}_i$ as single bit masks, the task is to find an efficient way to compute bit-parallely the bit mask $\mathcal{B}_i$ from the bit mask $\mathcal{B}_{i-1}$. (Notice that the initial bit mask $\mathcal{B}_{-1}$ is the null bit mask, i.e., $\mathcal{B}_{-1} = 0^L$.)

[5] Notice plainly that, once the bit mask $\mathcal{B}_i$ has been computed, we can check in constant time whether $P \trianglelefteq^i T$ holds by simply checking whether the $(\alpha+1)$-st bit of $\mathcal{B}_i$ is 1, i.e., if $\mathcal{B}_i[\alpha] = 1$, which corresponds to the condition that $\mathcal{B}_i \,\&\, U \neq 0^L$ where $U = 0^{\alpha}10^{L-\alpha-1}$.

To begin with, let $\mathcal{X}_i^{(j)}$ be the bit mask of length $\alpha+1$ defined by

$$\mathcal{X}_i^{(j)} = \begin{cases} 0^\alpha 1, & \text{if } T[i] =_\delta P[j] \text{ AND } (j=0 \text{ OR } \mathcal{C}_{i-1}^{(j-1)} \neq 0^{\alpha+1}) \\ 0^{\alpha+1}, & \text{otherwise}, \end{cases}$$

for $0 \le j < m$, and let $\mathcal{X}_i = \mathcal{X}_i^{(m-1)}\mathcal{X}_i^{(m-2)}\cdots\mathcal{X}_i^{(0)}$. Then, by (3) we have that $\mathcal{C}_i^{(j)} = ((\mathcal{C}_{i-1}^{(j)} \,\&\, 01^\alpha) \ll 1) \,|\, \mathcal{X}_i^{(j)}$, for $0 \le i < n$ and $0 \le j < m$, and therefore

$$\mathcal{B}_i = ((\mathcal{B}_{i-1} \,\&\, I) \ll 1) \,|\, \mathcal{X}_i\,, \tag{4}$$

for $0 \le i < n$, where $I = (01^\alpha)^m$. Thus, we need only to be able to compute effectively the bit mask $\mathcal{X}_i$ from the bit mask $\mathcal{B}_{i-1}$, which we do as follows.

For each symbol s of the alphabet Σ and each $0 \le j < m$, let $\mathsf{b}_s^{(j)}$ be the bit value 1, if $s =_\delta P[j]$ holds, otherwise let $\mathsf{b}_s^{(j)}$ be the bit value 0. Also, let

$$\mathcal{H}(s) = 0^\alpha(\mathsf{b}_s^{(m-1)}0^\alpha)(\mathsf{b}_s^{(m-2)}0^\alpha)\cdots(\mathsf{b}_s^{(1)}0^\alpha)\mathsf{b}_s^{(0)}\,.$$

Furthermore, let $\mathsf{x}_i^{(j)}$ be the last bit of the bit mask $\mathcal{X}_i^{(j)}$ (i.e., $\mathsf{x}_i^{(j)} = \mathcal{X}_i^{(j)}[\alpha]$), for $0 \le j < m$, so that we have

$$\mathcal{X}_i = 0^\alpha(\mathsf{x}_i^{(m-1)}0^\alpha)(\mathsf{x}_i^{(m-2)}0^\alpha)\cdots(\mathsf{x}_i^{(1)}0^\alpha)\mathsf{x}_i^{(0)}\,. \tag{5}$$

Then, we claim that

$$\mathsf{x}_i^{(0)} = \mathsf{b}_i^{(0)}\,, \tag{6}$$

and

$$\mathsf{x}_i^{(j)}0^\alpha = (\mathsf{b}_i^{(j)}0^\alpha) \,\&((((\mathcal{C}_{i-1}^{(j-1)} \,\&\, 01^\alpha) + 01^\alpha) \,|\, \mathcal{C}_{i-1}^{(j-1)})\,, \tag{7}$$

for $0 < j < m$, where we have written $\mathsf{b}_i^{(j)}$ in place of $\mathsf{b}_{T[i]}^{(j)}$ (just to simplify the notation). We need only to verify (7), since (6) is an immediate consequence of the definitions of $\mathsf{b}_i^{(0)}$ and $\mathcal{X}_i^{(0}$. To do this, we begin by noting that the operation $\mathcal{C}_{i-1}^{(j-1)} \,\&\, 01^\alpha$ sets the first bit of $\mathcal{C}_{i-1}^{(j-1)}$ to 0, leaving unchanged the remaining bits. Thus, by performing the arithmetic addition of $\mathcal{C}_{i-1}^{(j-1)} \,\&\, 01^\alpha$ with 01^α we obtain a bit mask whose first bit is 0 if and only if the last α bits of $\mathcal{C}_{i-1}^{(j-1)}$ are all 0's. Therefore, the bit mask $(((\mathcal{C}_{i-1}^{(j-1)} \,\&\, 01^\alpha) + 01^\alpha) \,|\, \mathcal{C}_{i-1}^{(j-1)})$ has its first bit equal to 0 if and only if $\mathcal{C}_{i-1}^{(j-1)}$ is null (i.e., if and only if $\mathcal{C}_{i-1}^{(j-1)} = 0^{\alpha+1}$). At this point (7) is an immediate consequence of the definitions of $\mathsf{b}_i^{(j)}$ and $\mathcal{X}_i^{(j)}$, and thus our claim is correct.

By (5), (6), (7), and the definition of the function $\mathcal{H}$, we get

$$\mathcal{X}_i = (((\mathcal{W}_{i-1} \,\&\, F) \ll 1) \,|\, 0^{L-1}1) \,\&\, \mathcal{H}(T[i])\,, \tag{8}$$

where we have put $F = 01^{L-1}$ and $\mathcal{W}_{i-1} = ((\mathcal{B}_{i-1} \,\&\, I) + I) \,|\, \mathcal{B}_{i-1}$. Relations (8) and (4) provide the required recursive formulae for computing the bit mask $\mathcal{B}_i$ from the bit mask $\mathcal{B}_{i-1}$. The resulting algorithm, named (δ,α)-Sequential-Sampling-BP, is reported in Figure 3. It uses an array H, indexed by the symbols of the alphabet Σ, which is computed in such a way that $H[s] = \mathcal{H}(s)$, for each $s \in \Sigma$. Notice also that at the end of the execution of the **for-loop** of line 5, we have $I = (01^\alpha)^m$ and $U = 0^\alpha 10^{L-\alpha-1}$, as required (cf. footnote 5).

It can easily be verified that time and space complexities of the (δ,α)-Sequential-Sampling-BP algorithm are $\mathcal{O}((\sigma + n + m\delta)\lceil(m\alpha)/w\rceil)$ and $\mathcal{O}(\sigma\lceil(m\alpha)/w\rceil)$, respectively.

```
(δ,α)-Sequential-Sampling-BP(P, m, T, n, δ, α)
 1.  L := (α + 1)m
 2.  for s ∈ Σ do H[s] := 0^L
 3.  I := 0^(L−α) 1^α
 4.  U := 0^(L−1) 1
 5.  for j := 0 to m − 1 do
 6.      for s ∈ Σ ∩ [P[j] − δ .. P[j] + δ] do
 7.          H[s] := H[s] | U
 8.      if j < m − 1 then
 9.          I := (I ≪ (α + 1)) | 0^(L−α) 1^α
10.          U := U ≪ (α + 1)
11.  F := 0 1^(L−1)
12.  B := 0^L
13.  for i := 0 to n − 1 do
14.      W := ((B & I) + I) | B
15.      X := (((W & F) ≪ 1) | 0^(L−1) 1) & H[T[i]]
16.      B := ((B & I) ≪ 1) | X
17.      if (B & U) ≠ 0^L then print(i)
```

```
(δ,α)-Sequential-Sampling-BP+(P, m, T, n, δ, α)
 1.  ℓ := (α + 1)(m − 1) + 1
 2.  for s ∈ Σ do H[s] := 0^ℓ
 3.  A := 0^ℓ
 4.  U := 0^(ℓ−1) 1
 5.  for j := 0 to m − 1 do
 6.      for s ∈ Σ ∩ [P[j] − δ .. P[j] + δ] do
 7.          H[s] := H[s] | U
 8.      if j < m − 1 then
 9.          A := A | U
10.          U := U ≪ (α + 1)
11.  J := U − A
12.  F := 0 1^(ℓ−1)
13.  B := 0^ℓ
14.  for i := 0 to n − 1 do
15.      B := (B & F) ≪ 1
16.      C := B & J
17.      B := (((C + J) | B) & H[T[i]]) | C
18.      if (B & U) ≠ 0^ℓ then print(i)
```

Figure 3. The (δ,α)-Sequential-Sampling-BP algorithm (on the left) and the (δ,α)-Sequential-Sampling-BP$^+$ algorithm (on the right) for the δ-approximate string matching problem with α-bounded gaps.

Let us make some remarks on the latter algorithm. To begin with, notice that if we replace the instructions of lines 3 and 11 of the (δ,α)-Sequential-Sampling-BP algorithm by the assignments $I := 0^L$ and $F := 0^\alpha 1 0^{L-\alpha-1}$, respectively, then the resulting algorithm still does the same work of the original one, except that the first α bits of the bit mask B (and of all the other bit masks) are always left unset (i.e., they remain 0's) during the course of the computation;[6] but, since the conditional test of line 17 (i.e., the test whether $P \trianglelefteq^i T$) involves only the $(\alpha+1)$-st bit of B, the modified algorithm solves the (δ,α)-matching problem as well. Thus, the first α bits of the bit masks used by the (δ,α)-Sequential-Sampling-BP algorithm can be dropped, and therefore the number of bits of these bit masks which need to be actually stored during the computation is $\ell = L - \alpha = (\alpha+1)(m-1)+1$ (and hence, in particular, if $\ell \leq w$ all of these bit masks fit each in a single computer word). Observe also that for $F = 0^\alpha 1 0^{L-\alpha-1}$ (as above) and $I = 0^{\alpha+1}(01^\alpha)^{m-1}$ (cf. footnote 6), the part of code of the (δ,α)-Sequential-Sampling-BP algorithm from line 12 up to line 17 turns out to be equivalent to the following one:

```
B := 0^L
for i := 0 to n − 1 do
    B := (B & F) ≪ 1
    C := B & J
    B := (((C + J) | B) & H[T[i]]) | C
    if (B & U) ≠ 0^L then print(i)
```

where $J = 0^\alpha(01^\alpha)^{m-1}1$, as can be easily verified by very simple algebraic manipulations, thus reducing the overall number of operations which need to be performed.

Such considerations translate into the variant of the (δ,α)-Sequential-Sampling-BP algorithm reported in Figure 3, named (δ,α)-Sequential-Sampling-BP$^+$, which

[6] In fact, with such modifications, at the end of the **for-loop** of line 5, we have that $I = 0^{\alpha+1}(01^\alpha)^{m-1}$, as can be easily verified.

although characterized by the same asymptotic space and time complexity of the original algorithm, turns out to be slightly more efficient in practice.[7]

Notice that at the end of the execution of the **for-loop** of line 5 of the (δ, α)-Sequential-Sampling-BP$^+$ algorithm, we have that $A = 0(0^\alpha 1)^{m-1}$ and $U = 10^{\ell-1}$, so that $U - A = (01^\alpha)^{m-1}1$ (as the J above, except that the first α 0's are dropped).[8]

Finally, we observe that it is easy to adapt our algorithms to handle also classes of characters and patterns with variable sized gaps, which arise in several search problems in molecular biology, but due to lack of space we will not give any details.

5 Experimental Results

In this section we report experimental data relative to an extensive comparison of our newly presented algorithms (δ, α)-Tuned-Sequential-Sampling-HBP and (δ, α)-Sequential-Sampling-BP$^+$, described in Section 4, and the algorithms SDP-simple, DA-mloga-bits, and (δ, α)-Shift-And, reviewed in Section 3, which are among the most efficient algorithms for the (δ, α)-matching problem.[9]

In particular, we have performed two main sets of experimental tests: the first one, the experimental set Es1, concerns the comparison of the algorithms (δ, α)-Tuned-Sequential-Sampling-HBP and SDP-simple, whereas the second one, the experimental set Es2, involves the algorithms (δ, α)-Tuned-Sequential-Sampling-HBP, (δ, α)-Sequential-Sampling-BP$^+$, DA-mloga-bits, and (δ, α)-Shift-And.

All algorithms have been implemented in the **C** programming language using the Borland C++ compiler, version 5.5, and were used to search for the same patterns in large fixed text sequences on a PC with a Pentium IV processor at 2.66 GHz, with 512 MB of RAM, running Windows XP. In particular, they have been tested on three Randσ problems, for $\sigma = 50, 90, 130$, and on a real music text buffer. Each Randσ problem consisted in searching for a set of 150 random patterns of length $m = 6, 8, 10, 20, 30, 40, 50, 60, 70, 85, 100$ in a random text sequence of length $n =$ 5,242,880, over a common alphabet of size σ. For each Randσ problem, the values of the approximation bound δ and of the gap bound α have been set to 1, 3, 5 and to 2, 5, 8, respectively. The running times of the algorithms have been averaged over all patterns. Concerning the tests on the real music text buffer, these have been performed on a fixed text sequence T of length $n = 2,982,507$ obtained by combining a set of various classical pieces in MIDI format, with an overall alphabet of 76 distinct symbols, i.e., the MIDI values of the notes of the pieces. For each m as above, we have randomly selected a set of 150 substrings of T of length m which subsequently have been searched for in T.

[7] Observe, however, that for $0 \leq j < m$, when iteration j of the **for-loop** of line 5 starts, we have $U = 0^\ell 1 \ll (j(\alpha+1))$. Therefore, the assignments of lines 7 and 9 could be implemented so as to take constant time, assuming the model in which a bit mask X is represented as a sequence of $\lceil |X|/w \rceil$ computer words, thus yielding an overall running time of $\mathcal{O}((n+\sigma)\lceil (m\alpha)/w \rceil + m\delta)$ rather than $\mathcal{O}((\sigma + n + m\delta)\lceil (m\alpha)/w \rceil)$.

[8] Notice that, in practice, the bit mask $(01^\alpha)^{m-1}1$ could also be computed as $(\sim(A \,|\, U)) \,|\, 0^{\ell-1}1$, where $\sim$ denotes the operation of bit complementation, which replaces each 0 in the bit mask by 1 and each 1 by 0.

[9] We have also considered in our experimental tests the algorithm SDP-simple-compute-L_0, but, due to lack of space, we omitted to report its timings, since it turned out to be always slower than the SDP-simple algorithm.

In the case of the experimental set Es2, the tests have been performed just as described above except that, this time, the algorithms involved in the comparison, i.e., (δ,α)-Tuned-Sequential-Sampling-HBP, (δ,α)-Sequential-Sampling-BP$^+$, DA-mloga-bits, and (δ,α)-Shift-And, have been tested using only short patterns and very small values of α. More precisely, the following pairs (α, m) have been used, where $(\alpha, m) \in \{1\} \times \{6, 8, 10, 12, 14, 16\} \cup \{2\} \times \{6, 8, 10\}$. The main reason behind this choice is that, for such pairs, each of the bit masks used by the last three algorithms fit in a single computer word, a condition which allows these algorithms to reach their best performances in practice.[10] The algorithm (δ,α)-Tuned-Sequential-Sampling-HBP has been included in this set of experimental tests mainly for comparing it with the algorithm DA-mloga-bits.

All running times in the tables are expressed in hundredths of second and, for each length of the pattern, the best result has been boldfaced. Moreover, the following abbreviations have been used to denote the algorithms: TSS-HBP for (δ,α)-Tuned-Sequential-Sampling-HBP; SS-BP for (δ,α)-Sequential-Sampling-BP$^+$; DA-NFA for (δ,α)-Shift-And; DA-CNFA for DA-mloga-bits; SDP-S for SDP-simple.

Experimental results on a Real Music Problem (Es1)

ALGS	(δ,α)	$m=6$	$m=8$	$m=10$	$m=20$	$m=30$	$m=40$	$m=50$	$m=60$	$m=70$	$m=85$	$m=100$
TSS-HBP	$(1,2)$	**2.36**	**2.40**	**2.44**	**2.50**	**2.54**	**2.30**	**2.27**	**2.39**	**2.42**	**2.44**	**2.48**
SDP-S	$(1,2)$	3.50	3.38	3.79	3.75	3.87	3.42	3.76	3.70	3.66	3.81	3.73
TSS-HBP	$(1,5)$	**4.14**	**4.33**	**4.95**	**4.93**	**5.17**	**4.35**	**4.53**	**4.85**	**4.72**	**4.85**	**4.63**
SDP-S	$(1,5)$	4.93	5.15	5.95	6.03	6.16	5.39	5.55	5.58	5.75	5.83	5.71
TSS-HBP	$(1,8)$	**5.65**	**6.36**	**7.69**	**7.93**	**8.13**	**6.67**	**7.05**	**7.45**	**7.31**	**7.61**	**7.33**
SDP-S	$(1,8)$	6.15	6.79	8.06	8.76	8.65	7.58	7.83	8.17	8.05	8.52	8.07
TSS-HBP	$(3,2)$	**5.11**	**4.78**	**5.27**	**5.16**	**5.90**	**4.87**	**5.05**	**5.53**	**5.41**	**4.97**	**5.57**
SDP-S	$(3,2)$	6.60	6.27	7.00	6.81	7.38	6.40	6.77	7.01	7.08	6.77	7.06
TSS-HBP	$(3,5)$	**9.57**	**10.00**	**12.40**	**12.96**	**14.61**	**12.68**	**12.50**	**13.42**	**13.17**	**12.59**	**13.49**
SDP-S	$(3,5)$	10.59	10.73	12.92	14.52	16.32	14.35	14.11	15.27	14.75	14.12	15.23
TSS-HBP	$(3,8)$	**11.13**	**12.63**	**16.59**	**20.43**	**24.57**	**22.56**	**21.45**	**24.19**	**23.53**	**21.83**	**23.60**
SDP-S	$(3,8)$	12.73	14.20	18.11	23.41	28.91	27.10	25.09	28.86	28.97	26.04	28.49
TSS-HBP	$(5,2)$	**9.03**	**9.05**	**10.54**	**10.58**	**15.46**	**18.78**	**19.95**	**18.98**	**19.32**	**18.88**	**21.63**
SDP-S	$(5,2)$	10.39	10.49	11.68	12.44	18.66	22.22	23.59	22.60	22.92	22.83	25.07
TSS-HBP	$(5,5)$	**13.14**	**15.02**	**19.46**	**23.73**	**24.83**	**25.06**	**27.69**	**51.78**	**33.51**	**28.93**	**37.44**
SDP-S	$(5,5)$	15.85	17.91	22.64	28.41	30.73	31.15	35.34	65.30	41.79	36.76	47.81
TSS-HBP	$(5,8)$	**12.94**	**15.92**	**21.29**	**30.10**	**36.26**	**36.67**	**47.64**	**48.03**	**52.02**	**55.14**	**52.40**
SDP-S	$(5,8)$	17.59	20.36	26.91	38.40	46.99	48.06	63.97	64.66	70.58	76.90	75.33

[10] Notice however, as already remarked, that by allowing only small values of the gap bound α (e.g., $\alpha \leq 2$) is not a real limitation in many practical applications in music. In fact, searching with small gaps is enough to take into account various kinds of musical ornamentations, such as mordent, acciaccatura and appoggiatura, as well as many other common musical technicalities such as pedal notes.

EXPERIMENTAL RESULTS ON A Rand50 PROBLEM (Es1)

ALGS	(δ, α)	$m=6$	$m=8$	$m=10$	$m=20$	$m=30$	$m=40$	$m=50$	$m=60$	$m=70$	$m=85$	$m=100$
TSS-HBP	$(1,2)$	**3.02**	**2.92**	**2.98**	**2.84**	**2.94**	**2.94**	**3.03**	**2.90**	**2.92**	**2.98**	**2.96**
SDP-S	$(1,2)$	4.60	4.78	4.58	4.69	4.75	4.73	4.80	4.77	4.63	4.65	4.77
TSS-HBP	$(1,5)$	**4.33**	**4.17**	**4.35**	**4.29**	**4.35**	**4.27**	**4.39**	**4.32**	**4.23**	**4.25**	**4.35**
SDP-S	$(1,5)$	6.19	6.11	6.25	6.21	6.17	6.19	6.15	6.01	6.19	6.09	6.11
TSS-HBP	$(1,8)$	**5.65**	**5.59**	**5.79**	**5.89**	**5.89**	**5.69**	**5.73**	**5.73**	**5.77**	**5.68**	**5.79**
SDP-S	$(1,8)$	7.43	7.65	7.79	7.61	7.71	7.67	7.81	7.59	7.63	7.67	7.67
TSS-HBP	$(3,2)$	**5.89**	**5.81**	**5.79**	**5.95**	**5.94**	**5.91**	**5.77**	**5.84**	**6.03**	**5.84**	**5.71**
SDP-S	$(3,2)$	8.85	8.73	8.85	8.83	8.83	8.62	8.87	8.79	8.88	8.85	8.79
TSS-HBP	$(3,5)$	**12.24**	**12.96**	**13.31**	**13.84**	**13.75**	**13.47**	**13.73**	**13.71**	**14.09**	**13.95**	**13.58**
SDP-S	$(3,5)$	13.10	14.63	15.56	16.27	16.09	15.80	16.13	16.28	16.57	16.28	16.11
TSS-HBP	$(3,8)$	17.38	20.48	22.83	**26.10**	**26.29**	**25.95**	**26.79**	**26.46**	**26.99**	**51.53**	**50.98**
SDP-S	$(3,8)$	**17.22**	**19.63**	**22.61**	29.15	29.75	29.28	30.56	30.32	30.64	59.02	58.44
TSS-HBP	$(5,2)$	**11.49**	**11.79**	**11.68**	**11.79**	**11.99**	**11.94**	**17.55**	**22.87**	**21.73**	**22.27**	**22.07**
SDP-S	$(5,2)$	14.11	14.82	15.28	15.12	15.28	15.51	22.95	29.34	28.47	29.10	28.77
TSS-HBP	$(5,5)$	**22.91**	**27.01**	**29.66**	**35.88**	**37.35**	**37.61**	**68.71**	**39.97**	**36.32**	**64.85**	**36.72**
SDP-S	$(5,5)$	24.04	27.65	30.35	40.83	44.26	45.15	82.58	47.06	43.83	77.20	43.95
TSS-HBP	$(5,8)$	**26.53**	**34.68**	**43.38**	**121.11**	**175.83**	**212.14**	**231.21**	**257.04**	**285.96**	**329.08**	**320.51**
SDP-S	$(5,8)$	29.82	37.62	46.58	135.99	205.92	254.88	281.20	318.26	357.93	429.03	422.84

EXPERIMENTAL RESULTS ON A Rand90 PROBLEM (Es1)

ALGS	(δ, α)	$m=6$	$m=8$	$m=10$	$m=20$	$m=30$	$m=40$	$m=50$	$m=60$	$m=70$	$m=85$	$m=100$
TSS-HBP	$(1,2)$	**2.27**	**2.27**	**2.40**	**2.42**	**2.40**	**2.38**	**2.38**	**2.26**	**2.32**	**2.30**	**2.36**
SDP-S	$(1,2)$	3.70	3.78	3.64	3.71	3.71	3.71	3.63	3.77	3.69	3.69	3.67
TSS-HBP	$(1,5)$	**2.94**	**3.03**	**3.11**	**3.00**	**3.05**	**2.93**	**2.96**	**2.96**	**2.86**	**2.94**	**2.97**
SDP-S	$(1,5)$	4.42	4.31	4.27	4.43	4.25	4.37	4.32	4.39	4.49	4.43	4.36
TSS-HBP	$(1,8)$	**3.57**	**3.21**	**3.61**	**3.36**	**3.35**	**3.43**	**3.36**	**3.41**	**3.27**	**3.57**	**3.45**
SDP-S	$(1,8)$	4.97	4.81	4.87	5.00	4.97	4.87	4.91	4.93	4.95	4.97	4.87
TSS-HBP	$(3,2)$	**3.41**	**3.51**	**3.55**	**3.39**	**3.47**	**3.49**	**3.50**	**4.93**	**6.59**	**6.53**	**6.66**
SDP-S	$(3,2)$	5.40	5.37	5.40	5.39	5.42	5.40	5.36	7.47	10.37	10.15	10.17
TSS-HBP	$(3,5)$	**5.59**	**5.55**	**5.71**	**5.57**	**5.81**	**5.71**	**5.59**	**5.63**	**5.65**	**5.61**	**5.61**
SDP-S	$(3,5)$	7.66	7.63	7.92	7.62	7.74	7.64	7.76	7.68	7.60	7.56	7.66
TSS-HBP	$(3,8)$	**8.06**	**8.25**	**8.33**	**8.32**	**8.51**	**8.45**	**8.24**	**8.28**	**8.39**	**8.29**	**8.15**
SDP-S	$(3,8)$	9.59	10.17	10.56	10.28	10.30	10.19	10.38	10.24	10.51	10.24	10.31
TSS-HBP	$(5,2)$	**7.11**	**7.01**	**9.86**	**9.66**	**9.28**	**9.68**	**9.76**	**9.63**	**9.63**	**9.72**	**9.75**
SDP-S	$(5,2)$	9.55	9.57	14.81	14.63	14.36	14.65	14.46	14.60	14.53	14.77	14.68
TSS-HBP	$(5,5)$	**10.04**	**10.54**	**10.81**	**10.79**	**10.45**	**10.85**	**10.75**	**10.77**	**10.79**	**10.93**	**10.82**
SDP-S	$(5,5)$	11.43	12.43	13.28	13.17	12.77	13.15	13.09	12.99	12.93	13.29	13.39
TSS-HBP	$(5,8)$	**14.48**	16.77	**17.86**	**19.45**	**19.39**	**19.80**	**19.46**	**19.57**	**19.59**	**19.85**	**19.86**
SDP-S	$(5,8)$	14.53	**16.72**	18.82	21.70	21.55	21.81	21.69	21.71	21.62	21.99	22.06

EXPERIMENTAL RESULTS ON A Rand130 PROBLEM (Es1)

ALGS	(δ, α)	$m=6$	$m=8$	$m=10$	$m=20$	$m=30$	$m=40$	$m=50$	$m=60$	$m=70$	$m=85$	$m=100$
TSS-HBP	$(1,2)$	**2.16**	**2.12**	**2.14**	**2.12**	**2.14**	**2.10**	**2.12**	**2.14**	**2.24**	**2.14**	**2.15**
SDP-S	$(1,2)$	3.37	3.30	3.44	3.53	3.41	3.38	3.34	3.47	3.32	3.33	3.34
TSS-HBP	$(1,5)$	**2.61**	**2.56**	**2.52**	**2.60**	**2.62**	**2.44**	**2.50**	**2.52**	**2.53**	**2.50**	**2.54**
SDP-S	$(1,5)$	3.72	3.83	3.74	3.66	3.70	3.78	3.70	3.81	3.80	3.83	3.75
TSS-HBP	$(1,8)$	**2.92**	**2.78**	**2.96**	**2.78**	**2.88**	**2.80**	**2.76**	**2.89**	**2.85**	**2.82**	**2.80**
SDP-S	$(1,8)$	4.15	4.33	4.09	4.12	4.06	4.08	4.15	4.07	4.06	4.11	4.09
TSS-HBP	$(3,2)$	**2.83**	**2.95**	**2.83**	**2.84**	**2.83**	**2.84**	**2.74**	**2.81**	**2.89**	**2.88**	**2.80**
SDP-S	$(3,2)$	4.42	4.57	4.59	4.52	4.62	4.59	4.59	4.52	4.60	4.39	4.58
TSS-HBP	$(3,5)$	**4.07**	**4.04**	**4.15**	**4.04**	**3.94**	**4.05**	**4.03**	**4.01**	**4.10**	**4.07**	**7.65**
SDP-S	$(3,5)$	5.66	5.66	5.66	5.68	5.79	5.77	5.68	5.72	5.70	5.78	10.76
TSS-HBP	$(3,8)$	**5.08**	**4.96**	**5.23**	**5.17**	**5.13**	**5.11**	**5.18**	**5.22**	**5.20**	**5.04**	**5.19**
SDP-S	$(3,8)$	6.87	6.95	7.04	7.01	6.99	6.94	6.99	6.96	6.97	6.89	6.99
TSS-HBP	$(5,2)$	**3.78**	**3.78**	**3.84**	**3.68**	**3.72**	**6.14**	**7.14**	**7.06**	**7.09**	**7.05**	**6.91**
SDP-S	$(5,2)$	5.79	5.74	5.93	5.71	5.66	9.69	10.89	10.91	10.95	10.96	10.77
TSS-HBP	$(5,5)$	**9.14**	**9.39**	**9.78**	**9.53**	**9.77**	**9.56**	**9.62**	**9.82**	**9.86**	**9.71**	**9.46**
SDP-S	$(5,5)$	10.39	11.27	11.52	11.48	11.52	11.39	11.53	11.40	11.67	11.50	11.31
TSS-HBP	$(5,8)$	**10.23**	**10.15**	**12.34**	**11.96**	**11.82**	**12.00**	**11.92**	**11.97**	**12.14**	**11.94**	**11.69**
SDP-S	$(5,8)$	12.20	12.29	16.42	15.68	15.88	15.78	15.88	15.96	15.89	15.94	15.72

Experimental results on a Real Music Problem (Es2)

ALGS	(δ,α)	$m=6$	$m=8$	$m=10$	$m=12$	$m=14$	$m=16$	ALGS	(δ,α)	$m=6$	$m=8$	$m=10$
TSS-HBP	$(1,1)$	2.00	1.88	2.04	2.06	2.00	1.98	TSS-HBP	$(1,2)$	2.36	2.27	2.42
SS-BP	$(1,1)$	**1.00**	**0.84**	**0.96**	**0.94**	**0.94**	**0.88**	SS-BP	$(1,2)$	**0.92**	**0.90**	**0.82**
DA-NFA	$(1,1)$	1.02	1.00	1.00	1.00	1.04	1.00	DA-NFA	$(1,2)$	1.02	1.00	1.05
DA-CNFA	$(1,1)$	9.15	9.14	9.03	9.12	9.13	9.17	DA-CNFA	$(1,2)$	9.22	9.19	9.09
TSS-HBP	$(3,1)$	3.26	3.28	3.29	3.41	3.39	3.48	TSS-HBP	$(3,2)$	5.14	4.58	4.99
SS-BP	$(3,1)$	**0.94**	**0.94**	**0.96**	**0.88**	**0.94**	**0.98**	SS-BP	$(3,2)$	**0.92**	**0.94**	**0.94**
DA-NFA	$(3,1)$	1.06	1.04	1.05	1.02	1.04	1.02	DA-NFA	$(3,2)$	1.16	1.14	1.06
DA-CNFA	$(3,1)$	9.34	9.35	9.23	9.49	9.24	9.20	DA-CNFA	$(3,2)$	9.40	9.25	9.30
TSS-HBP	$(5,1)$	5.13	5.35	4.93	5.69	6.02	5.28	TSS-HBP	$(5,2)$	8.70	8.87	9.91
SS-BP	$(5,1)$	1.08	**0.92**	**0.90**	**0.88**	**0.96**	**0.84**	SS-BP	$(5,2)$	**1.18**	**1.06**	**1.02**
DA-NFA	$(5,1)$	**1.07**	1.06	1.04	1.06	1.04	1.06	DA-NFA	$(5,2)$	1.20	1.08	1.06
DA-CNFA	$(5,1)$	9.38	9.25	9.26	9.25	9.33	9.29	DA-CNFA	$(5,2)$	9.54	9.44	9.28

Experimental results on a Rand50 problem (Es2)

ALGS	(δ,α)	$m=6$	$m=8$	$m=10$	$m=12$	$m=14$	$m=16$	ALGS	(δ,α)	$m=6$	$m=8$	$m=10$
TSS-HBP	$(1,1)$	2.68	2.76	2.66	2.82	2.68	2.78	TSS-HBP	$(1,2)$	3.05	2.98	2.92
SS-BP	$(1,1)$	**1.68**	**1.68**	**1.58**	**1.52**	**1.64**	**1.58**	SS-BP	$(1,2)$	**1.72**	**1.68**	1.78
DA-NFA	$(1,1)$	1.94	1.70	1.84	1.92	1.86	1.76	DA-NFA	$(1,2)$	1.90	1.81	**1.70**
DA-CNFA	$(1,1)$	16.07	15.88	15.96	15.95	16.04	15.92	DA-CNFA	$(1,2)$	16.07	16.06	15.95
TSS-HBP	$(3,1)$	4.52	4.46	4.46	4.60	4.58	4.56	TSS-HBP	$(3,2)$	5.95	5.81	5.73
SS-BP	$(3,1)$	**1.74**	**1.64**	**1.74**	**1.67**	**1.55**	1.73	SS-BP	$(3,2)$	**1.79**	**1.78**	**1.78**
DA-NFA	$(3,1)$	1.92	1.89	1.84	1.74	1.92	**1.72**	DA-NFA	$(3,2)$	1.84	1.80	1.92
DA-CNFA	$(3,1)$	16.68	16.28	16.30	16.36	16.34	16.34	DA-CNFA	$(3,2)$	16.53	16.33	16.24
TSS-HBP	$(5,1)$	10.37	13.13	13.49	13.55	13.49	13.91	TSS-HBP	$(5,2)$	11.35	11.57	11.57
SS-BP	$(5,1)$	**2.46**	**3.44**	**3.30**	**3.36**	**3.43**	**3.22**	SS-BP	$(5,2)$	**1.82**	**1.74**	**1.68**
DA-NFA	$(5,1)$	2.70	3.56	3.51	3.46	3.54	3.50	DA-NFA	$(5,2)$	1.94	1.86	1.84
DA-CNFA	$(5,1)$	23.32	31.23	31.16	31.28	31.05	31.15	DA-CNFA	$(5,2)$	16.58	16.35	16.31

Experimental results on a Rand90 problem (Es2)

ALGS	(δ,α)	$m=6$	$m=8$	$m=10$	$m=12$	$m=14$	$m=16$	ALGS	(δ,α)	$m=6$	$m=8$	$m=10$
TSS-HBP	$(1,1)$	2.24	2.24	2.21	2.28	2.25	2.30	TSS-HBP	$(1,2)$	2.38	2.30	2.36
SS-BP	$(1,1)$	**1.68**	**1.71**	**1.64**	**1.68**	**1.70**	**1.68**	SS-BP	$(1,2)$	**1.68**	**1.70**	**1.68**
DA-NFA	$(1,1)$	1.84	1.78	1.86	1.76	1.82	1.80	DA-NFA	$(1,2)$	2.02	1.78	1.84
DA-CNFA	$(1,1)$	16.12	15.92	15.98	16.02	15.94	15.96	DA-CNFA	$(1,2)$	16.14	15.94	15.88
TSS-HBP	$(3,1)$	3.16	3.03	3.09	3.03	3.05	2.97	TSS-HBP	$(3,2)$	3.59	3.29	3.48
SS-BP	$(3,1)$	1.79	**1.78**	**1.70**	**1.74**	**1.74**	1.84	SS-BP	$(3,2)$	**1.76**	**1.83**	**1.72**
DA-NFA	$(3,1)$	**1.76**	1.84	1.80	1.82	1.82	**1.74**	DA-NFA	$(3,2)$	1.89	1.88	1.84
DA-CNFA	$(3,1)$	16.57	16.22	16.34	16.39	16.28	16.30	DA-CNFA	$(3,2)$	16.56	16.33	16.38
TSS-HBP	$(5,1)$	3.99	4.09	4.07	4.00	4.05	3.97	TSS-HBP	$(5,2)$	5.26	4.97	4.96
SS-BP	$(5,1)$	**1.86**	**1.68**	**1.74**	1.77	**1.60**	**1.68**	SS-BP	$(5,2)$	**1.72**	**1.76**	**1.76**
DA-NFA	$(5,1)$	**1.86**	1.88	1.78	**1.76**	1.94	1.88	DA-NFA	$(5,2)$	1.84	1.78	1.92
DA-CNFA	$(5,1)$	16.51	16.31	16.39	16.30	16.35	16.34	DA-CNFA	$(5,2)$	16.47	16.27	16.31

Experimental results on a Rand130 problem (Es2)

ALGS	(δ,α)	$m=6$	$m=8$	$m=10$	$m=12$	$m=14$	$m=16$	ALGS	(δ,α)	$m=6$	$m=8$	$m=10$
TSS-HBP	$(1,1)$	2.30	2.07	2.00	2.20	2.10	2.02	TSS-HBP	$(1,2)$	2.26	2.16	2.14
SS-BP	$(1,1)$	**1.58**	**1.72**	**1.72**	**1.62**	**1.68**	**1.62**	SS-BP	$(1,2)$	**1.52**	**1.70**	**1.66**
DA-NFA	$(1,1)$	1.82	**1.72**	1.82	1.88	1.84	1.88	DA-NFA	$(1,2)$	1.98	**1.70**	1.82
DA-CNFA	$(1,1)$	16.12	15.98	15.95	16.00	15.92	15.96	DA-CNFA	$(1,2)$	16.12	16.02	16.00
TSS-HBP	$(3,1)$	2.73	2.85	2.62	2.61	2.60	2.62	TSS-HBP	$(3,2)$	2.97	2.89	2.85
SS-BP	$(3,1)$	**1.71**	**1.42**	1.86	**1.57**	**1.76**	**1.74**	SS-BP	$(3,2)$	**1.65**	**1.73**	**1.70**
DA-NFA	$(3,1)$	1.88	1.92	**1.80**	1.98	1.80	1.94	DA-NFA	$(3,2)$	1.98	1.84	1.84
DA-CNFA	$(3,1)$	16.44	16.32	16.16	16.32	16.32	16.49	DA-CNFA	$(3,2)$	16.45	16.30	16.34
TSS-HBP	$(5,1)$	3.15	3.20	3.11	5.51	6.20	6.14	TSS-HBP	$(5,2)$	3.76	3.72	3.80
SS-BP	$(5,1)$	**1.75**	**1.75**	**1.78**	**2.97**	**3.38**	**3.30**	SS-BP	$(5,2)$	**1.79**	**1.69**	**1.49**
DA-NFA	$(5,1)$	1.86	1.82	1.86	3.09	3.48	3.62	DA-NFA	$(5,2)$	1.84	1.78	1.82
DA-CNFA	$(5,1)$	16.48	16.29	16.37	27.50	31.51	31.15	DA-CNFA	$(5,2)$	16.52	16.30	16.34

From the experimental results it turns out that our algorithms (δ,α)-Tuned-Sequential-Sampling-HBP and (δ,α)-Sequential-Sampling-BP$^+$ are very efficient in practice. In the case of very short patterns and very small values of α (cf. the experimental set Es2), the algorithm (δ,α)-Sequential-Sampling-BP$^+$ is in general the fastest one, and beats also the automaton based algorithm (δ,α)-Shift-And. Moreover, it is about 8-9 times faster than DA-mloga-bits. Notice also that the algorithm (δ,α)-Tuned-Sequential-Sampling-HBP is always faster than DA-mloga-bits.

In the more general case of patterns of very varied lengths (cf. the experimental set Es1), the algorithm (δ, α)-Tuned-Sequential-Sampling-HBP outperforms almost always the very efficient SDP-simple; very rarely SDP-simple wins against (δ, α)-Tuned-Sequential-Sampling-HBP (just in the 0.9 per cent of the cases, with very short patterns). However, we recall that, in the worst case, the (δ, α)-Tuned-Sequential-Sampling-HBP algorithm requires only $\mathcal{O}(m\lceil\alpha/w\rceil)$ extra space, whereas the SDP-simple algorithm uses $\mathcal{O}(n)$ extra space.

6 Conclusions

We have presented some efficient practical algorithms for the δ-approximate string matching problem with α-bounded gaps, which have important applications in music information retrieval. Despite their non-optimal asymptotic behavior, our algorithms perform very well in practice and, in particular, one of them wins against the fastest existing algorithms in most practical cases.

Acknowledgments

We thank K. Fredriksson and S. Grabowski for having provided us with the **C** source code of their algorithms SDP-simple, DA-mloga-bits, and SDP-simple-compute-L_0, which we have used in our tests.

We also thank the anonymous referees for helpful comments.

References

1. E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon: *Algorithms for computing approximate repetitions in musical sequences*. International Journal of Computer Mathematics, 79(11) 2002, pp. 1135–1148.
2. D. Cantone, S. Cristofaro, and S. Faro: *An efficient algorithm for δ-approximate matching with α-bounded gaps in musical sequences*, in Proceedings of 4-th International Workshop on Experimental and Efficient Algorithms (WEA'05), S. E. Nikoletseas, ed., vol. 3503 of Lecture Notes in Computer Science, Springer-Verlag, 2005, pp. 428–439.
3. 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 6-th International Conference on Music Information Retrieval (ISMIR'05), S. D. Reiss and G. A. Wiggins, eds., 2005, pp. 454–459.
4. M. Crochemore, C. Iliopoulos, C. Makris, W. Rytter, A. Tsakalidis, and K. Tsichlas: *Approximate string matching with gaps*. Nordic J. of Computing, 9(1) 2002, pp. 54–65.
5. M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and W. Rytter: *Finding motifs with gaps*, in Proceedings of the International Symposium on Music Information Retrieval (ISMIR'00), Plymouth, USA, 2000, pp. 306–317, poster paper.
6. K. Fredriksson and S. Grabowski: *Efficient bit-parallel algorithms for (δ, α)-matching*, in Proceedings of 5-th Workshop on Efficient and Experimental Algorithms (WEA'06), LNCS 4007, Springer–Verlag, 2006, pp. 170–181.
7. K. Fredriksson and S. Grabowski: *Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance*. Information Retrieval, March 2008, to appear (currently available only online).
8. R. N. Horspool: *Practical fast searching in strings*. Software, Practice & Experience, 10(6) 1980, pp. 501–506.
9. G. Navarro and M. Raffinot: *Fast and simple character classes and bounded gaps pattern matching, with application to protein searching*, in RECOMB'01: Proceedings of the fifth annual international conference on Computational biology, New York, NY, USA, 2001, ACM, pp. 231–240.
10. Y. J. Pinzon and S. Wang: *Simple algorithm for pattern-matching with bounded gaps in genomic sequences*, in Proceedings of the International Conference on Numerical Analysis and Applied Mathematics (ICNAAM'05), 2005, pp. 827–831.

Average Value of Sum of Exponents of Runs in Strings

Kazuhiko Kusano, Wataru Matsubara, Akira Ishino, and Ayumi Shinohara

Graduate School of Information Science, Tohoku University,
Aramaki aza Aoba 6-6-05, Aoba-ku, Sendai 980-8579, Japan
{kusano@shino., matsubara@shino., ishino@, ayumi@}ecei.tohoku.ac.jp

Abstract. A substring $w[i..j]$ in w is called a repetition of period p if $s[k] = s[k+p]$ for any $i \leq k \leq j-p$. Especially, a maximal repetition, which cannot be extended neither to left nor to right, is called a run. The ratio of the length of the run to its period, i.e. $\frac{j-i+1}{p}$, is called an exponent. The sum of exponents of runs in a string is of interest. The maximal value of the sum is still unknown, and the current upper bound is $2.9n$ given by Crochemore and Ilie, where n is the length of a string. In this paper we show a closed formula which exactly expresses the average value of it for any n and any alphabet size, and the limit of this value per unit length as n approaches infinity. For binary strings, the limit value is approximately 1.13103.

1 Introduction

Repetitions in strings are an important element in the analysis and processing of strings. We especially focus on the runs, which are non-extendable repetitions. Kolpakov and Kucherov showed that the maximal number of runs $\rho(n)$ in any strings of length n is at most cn for some constant c [4]. Although they gave no value for c, recently there have been several results lowering the value [1,3,9,11]. The currently known best upper bound is $c = 1.048$ [2,3]. It is conjectured that $c < 1$.

A repetition count of run is called an exponent, and the maximal sum of exponents is also well studied [1,5,10]. It is proved that the maximal sum of exponents is linear and the current best upper bound is $2.9n$ [1]. It is conjectured that the sum of exponents is less than $2n$.

Although the exact estimation of the maximal number $\rho(n)$ of runs is still unknown, Puglisi and Simpson [8] presented a formula that gives the number of runs in a string of length n on average as follows:

$$r(n) = \sum_{p=1}^{\frac{n}{2}} \sigma^{n-2p-1} \left((n-2p+1)\sigma - (n-2p)\right) \sum_{d|p} \mu(d)\sigma^{\frac{p}{d}},$$

where σ is the alphabet size and $\mu(n)$ is the Möbius function.

In this paper, we consider the average value $e(n)$ of sum of exponents of runs in strings of length n and prove that

$$e(n) = \sum_{p=1}^{\frac{n}{2}} L(p) \left(2p(n-2p+1)\sigma^{-2p} - (2p-1)(n-2p)\sigma^{-2p-1}\right).$$

Moreover, we show that

$$\lim_{n\to\infty} \frac{e(n)}{n} = \sum_{d=1}^{\infty} \mu(d) \left(\frac{2(\sigma-1)}{\sigma^{2d}-\sigma} + \frac{1}{d\sigma} \ln\left(\frac{\sigma^{2d}}{\sigma^{2d}-\sigma}\right)\right),$$

where $L(n)$ is the number of Lyndon words of length n.

Kazuhiko Kusano, Wataru Matsubara, Akira Ishino, Ayumi Shinohara : Average Value of Sum of Exponents of Runs in Strings, pp. 185–192.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

2 Definitions

Let $\Sigma = \{0, 1, 2, \ldots, \sigma - 1\}$ be an *alphabet* of size σ, that is, $|\Sigma| = \sigma$. The set of all the strings on Σ is denoted by Σ^*, and the set of all the strings of length n by Σ^n. For a string w, we denote its length by $|w|$. We index w from 0 to $|w|-1$ and denote its ith letter by w, i.e. $w = w[0]w[1]\ldots w[|w|-1]$. We denote by $w[i..j]$ the *substring* $w[i]w[i+1]\ldots w[j]$ of w. Let ε be the empty string. We say that p is a *period* of w if $w[i] = w[i+p]$ holds for any $i \geq 0$. For a string $w = xyz$, x, y and z are called a *prefix*, *substring* and *suffix* of w, respectively.

A substring $w[i..j]$ of w is called a *run* if it is *periodic*, i.e., it has the shortest period p satisfying $p \leq \frac{j-i+1}{2}$ and it is *non-extendable*, i.e., it satisfies the following two conditions:

$$\begin{array}{ll} i = 0 & \text{or } \; w[i-1] \neq w[i+p-1], \\ j = n-1 & \text{or } \; w[j+1] \neq w[j-p+1]. \end{array}$$

We denote the run $w[i..j]$ by a triple (i, j, p). The *exponent* is the ratio $\frac{j-i+1}{p}$, and the *root* is the prefix $w[i..i+p-1]$ of length p. We denote by $Runs(w)$ the number of runs contained in string w, and by $Exp(w)$ the sum of exponents of all runs in string w.

We say that a string w is *primitive* if w cannot be written as $w = u^k$ by any string u and any integer $k \geq 2$. A string w is called a *Lyndon word* if w is minimal in the lexicographical ordering of all its non-empty suffixes [6]. We denote by $L(n)$ the number of Lyndon words of length n over the given alphabet. By these definitions, Lyndon words must be primitive and the number of primitive strings of length n is equal to $nL(n)$. The Möbius function $\mu(n)$ is defined by

$$\mu(n) = \begin{cases} 0 & \text{if } n \text{ has one or more repeated prime factors,} \\ 1 & \text{if } n \text{ has an even number of distinct prime factors,} \\ -1 & \text{if } n \text{ has an odd number of distinct prime factors.} \end{cases}$$

It is known that $L(n)$ can be expressed as follows [7]:

$$L(n) = \frac{1}{n} \sum_{d|n} \mu\left(\frac{n}{d}\right) \sigma^d.$$

The notation $d|p$ means that d is a divisor of p.

3 Main Result

We are interested in the average number $r(n)$ of runs and the average value $e(n)$ of sum of exponents of runs in strings of length n, defined as follows:

$$\begin{aligned} r(n) &= average\{Runs(w) : w \in \Sigma^n\}, \\ e(n) &= average\{Exp(w) : w \in \Sigma^n\}. \end{aligned}$$

Puglisi and Simpson showed that the following equation holds.

Theorem 1 (Puglisi and Simpson [8])**.**

$$r(n) = \sum_{p=1}^{\frac{n}{2}} \sigma^{n-2p-1} \left((n-2p+1)\sigma - (n-2p)\right) \sum_{d|p} \mu(d)\sigma^{\frac{p}{d}}.$$

We prove the following equation in the sequel.

Theorem 2.

$$e(n) = \sum_{p=1}^{\frac{n}{2}} L(p) \left(2p(n-2p+1)\sigma^{-2p} - (2p-1)(n-2p)\sigma^{-2p-1}\right).$$

For a string w of length n and a positive integer p, we define a string $d(w,p)$ of length $n-p$ as follows:

$$d(w,p)[i] = w[i+p] - w[i] \pmod{\sigma} \qquad \text{for } 0 \le i < n-p,$$

where the operators $-$ and mod are applied to symbols as if these symbols are numbers. For example, for a string $w = \texttt{21010}$ on $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{2}\}$, we have $d(w,1) = \texttt{2212}$ and $d(w,2) = \texttt{100}$.

A substring $w[i..j]$ of $w \in \Sigma^n$ is called a 0-*segment* if $w[i..j]$ is a maximal block of 0's, that is, $w[t] = \texttt{0}$ for every t $(i \le t \le j)$ and it satisfies the following two conditions:

$$\begin{array}{ll} i = 0 & \text{or } \ w[i-1] \neq \texttt{0}, \\ j = n-1 & \text{or } \ w[j+1] \neq \texttt{0}. \end{array}$$

We denote the 0-segment by a pair (i,j).

Example 3. For string 0012000102, 0-segments are $(0,1)$, $(4,6)$, $(8,8)$.

Lemma 4. *For any string w, a substring $w[i..j+p]$ is a run with period p if and only if $d(w,p)[i..j]$ is a* 0*-segment of length $\ge p$.*

Proof. When there exists 0-segment (i,j) in $d(w,p)$, it holds that $w[t] = w[t+p]$ $(i \le t \le j)$, i.e., $w[i..j+p]$ has the period p. $|w[i..j+p]| = j+p-i+1 \ge 2p$ if and only if $|d(w,p)[i..j]| = j-i+1 \ge p$. Moreover, $w[i..j+p]$ satisfies the non-extendable condition. Therefore, $w[i..j+p]$ is a run. The "only if" part is clear. □

We denote by $c(n,p)$ the number of 0-segments of length p in all strings from Σ^n, and by $C(n,p)$ the number of 0-segments of length $\ge p$ in all strings from Σ^n. By definition, $C(n,p) = \sum_{i=p}^{n} c(n,i)$. For 0-segments of length $\ge p$ in Σ^n, we denote the sum of $\frac{l}{p}$ by $C_e(n,p)$, where l is the length of each 0-segments, i.e., $C_e(n,p) = \sum_{i=p}^{n} c(n,i)\frac{i}{p}$.

Example 5. For $\sigma = 2$, $c(5,2)$ is 12 because among all strings of length 5, all 0-segments of length 2 are underlined as follows:

00000	$\underline{00}1\underline{00}$	01000	$011\underline{00}$	10000	$101\underline{00}$	11000	$111\underline{00}$
00001	$\underline{00}101$	$01\underline{00}1$	01101	10001	10101	$11\underline{00}1$	11101
00010	$\underline{00}110$	01010	01110	$1\underline{00}10$	10110	11010	11110
00011	$\underline{00}111$	01011	01111	$1\underline{00}11$	10111	11011	11111

Similarly, $c(5,3) = 5$, $c(5,4) = 2$ Macro 3 $c(5,5) = 1$. Then $C(5,2) = 12+5+2+1 = 20$. $C_e(5,2) = 12 \cdot \frac{2}{2} + 5 \cdot \frac{3}{2} + 2 \cdot \frac{4}{2} + \frac{5}{2} = 26$.

Lemma 6. *For any positive integer n and $p \le n$, it holds that*

$$C(n,p) = (n-p+1)\sigma^{n-p} - (n-p)\sigma^{n-p-1}, \text{ and}$$
$$C_e(n,p) = \frac{1}{p}\left(p(n-p+1)\sigma^{n-p} - (p-1)(n-p)\sigma^{n-p-1}\right).$$

Proof. Let $Q_{n,p}$ be the set of pairs of strings separated by $\mathtt{0}$-segments of length p giving in concatenation strings of length n:

$$Q_{n,p} = \{(\alpha,\beta) : \alpha\mathtt{0}^p\beta \in \Sigma^n \wedge \alpha,\beta \in \Sigma^* \wedge \alpha[|\alpha|-1] \neq \mathtt{0} \wedge \beta[0] \neq \mathtt{0}\}.$$

Then $c(n,p) = |Q_{n,p}|$ because a one-to-one correspondence exists between $\mathtt{0}$-segments of length p and $Q_{n,p}$.

Example 7. For $\sigma = 2, n = 3$ and $p = 1$, there are $\mathtt{0}$-segments of length p in Σ^n as follows:

$$\Sigma^3 = \{\mathtt{000}, \mathtt{001}, \underline{\mathtt{0}}\mathtt{1}\underline{\mathtt{0}}, \underline{\mathtt{0}}\mathtt{11}, \mathtt{100}, \mathtt{1}\underline{\mathtt{0}}\mathtt{1}, \mathtt{11}\underline{\mathtt{0}}, \mathtt{111}\}.$$

The number of $\mathtt{0}$-segments $c(3,1)$ is 5. Then $Q_{3,1}$ becomes as follows:

$$Q_{3,1} = \{(\varepsilon, \mathtt{10}), (\mathtt{01}, \varepsilon), (\varepsilon, \mathtt{11}), (\mathtt{1}, \mathtt{1}), (\mathtt{11}, \varepsilon)\}.$$

We get expression for $c(n,p)$ by considering the number of elements of $Q_{n,p}$ in two cases.

(1) For $p \le n-1$,
When $\alpha = \varepsilon$, since $|\beta| = n-p$ and $\beta[0] \neq \mathtt{0}$, there are $(\sigma-1)$ choices for $\beta[0]$ and σ^{n-p-1} choices for $\beta[1..n-p-1]$. $|Q_{n,p}| = (\sigma-1)\sigma^{n-p-1}$. Similarly, when $\beta = \varepsilon$, $|Q_{n,p}| = (\sigma-1)\sigma^{n-p-1}$. In the case of $\alpha \neq \varepsilon$ and $\beta \neq \varepsilon$, there are $(n-p-1)$ choices for the position of $\mathtt{0}^p$, $(\sigma-1)$ choices for $\alpha[|\alpha|-1]$ and $\beta[0]$, and σ^{n-p-2} choices for the other characters since $|\alpha| + |\beta| = n-p$, $\alpha[|\alpha|-1] \neq \mathtt{0}$, and $\beta[0] \neq \mathtt{0}$. $|Q_{n,p}| = (n-p-1)(\sigma-1)^2\sigma^{n-p-2}$. For $p = n-1$, α or β is ε, and $(n-p-1)(\sigma-1)^2\sigma^{n-p-2}$ equal to 0. Therefore,

$$\begin{aligned} c(n,p) &= |Q_{n,p}| \\ &= 2(\sigma-1)\sigma^{n-p-1} + (n-p-1)(\sigma-1)^2\sigma^{n-p-2} \\ &= (n-p+1)\sigma^{n-p} - 2(n-p)\sigma^{n-p-1} + (n-p-1)\sigma^{n-p-2}. \end{aligned}$$

(2) For $p = n$,
Since $\alpha = \beta = \varepsilon$,

$$c(n,p) = |Q_{n,p}| = 1.$$

For $p \le n-1$,

$$\begin{aligned} C(n,p) &= \sum_{i=p}^{n} c(n,i) \\ &= \sum_{i=p}^{n-2} \left((n-i+1)\sigma^{n-i} - 2(n-i)\sigma^{n-i-1} + (n-i-1)\sigma^{n-i-2}\right) + 2(\sigma-1) + 1 \\ &= (n-p+1)\sigma^{n-p} - (n-p)\sigma^{n-p-1}. \end{aligned}$$

This equation holds for $C(n,n) = 1$.

For $p \le n-1$,

$$\begin{aligned} C_e(n,p) &= \sum_{i=p}^{n} c(n,i)\frac{i}{p} \\ &= \sum_{i=p}^{n-2} \left((n-i+1)\sigma^{n-i} - 2(n-i)\sigma^{n-i-1} + (n-i-1)\sigma^{n-i-2}\right)\frac{i}{p} \\ &\quad +2(\sigma-1)\frac{n-1}{p} + \frac{n}{p} \\ &= \frac{1}{p}\left(p(n-p+1)\sigma^{n-p} - (p-1)(n-p)\sigma^{n-p-1}\right). \end{aligned}$$

This equation holds for $C_e(n,n) = 1$. □

Lemma 8. *For any integer p and strings w and v of length n such that $d(w,p) = d(v,p)$, $w[i..i+p-1] = v[i..i+p-1]$ for some i if and only if $w = v$.*

Proof. $(=:)$ We prove this by induction. Let $i \le j < i+p$ and k are integers. For $k = 0$, $w[j+kp] = v[j+kp]$ is hold. For $k \ge 1$, if $w[j+kp] = v[j+kp]$ is hold, $w[j+(k+1)p] = w[j+kp] + d(w,p)[j+kp] \pmod{\sigma} = v[j+kp] + d(v,p)[j+kp] \pmod{\sigma} = v[j+(k+1)p]$. Then, $w[i..n-1] = v[i..n-1]$. Similarly, $w[0..i+p-1] = v[0..i+p-1]$. Therefore, $w = v$.
$(\Longleftarrow)$ It is clear. □

When $w \in \Sigma^n$, the length of $d(w,p)$ is $n-p$ and the number of `0`-segments in Σ^{n-p} is $C(n-p,p)$. By Lemma 4 and 8, there are $\sigma^p C(n-p,p)$ runs which have period p in Σ^n. However, runs may have different periods. For example, `0101010101` has periods both 2 and 4. To prevent counting these runs more than once, we should consider counting runs with the minimum period.

Lemma 9. *The ratio of the number of runs whose shortest period is p to the number of runs which have period p is $\frac{pL(p)}{\sigma^p}$.*

Proof. The number of runs which have period p is $\sigma^p C(n-p,p)$. On the other hand, if a run of period p has different period $q < p$, the run also has period $\gcd(p,q)$ by Periodicity Lemma [6]. So $w[i..j]$ has no period $q < p$ if its root is primitive. The number of primitive strings of length p is $pL(p)$. Therefore, the number of runs whose shortest period is p is $pL(p)C(n-p,p)$. □

By Lemma 4, `0`-segments of length l $(l \ge p)$ in $d(w,p)$ correspond to runs of length $l+p$ in w. The exponents of these runs are $\frac{l}{p}+1$. This and Lemma 6, 8, and 9 derive $\sigma^n e(n)$, the sum of exponents of all runs in Σ^n, as follows:

$$\begin{aligned} \sigma^n e(n) &= \sum_{p=1}^{\frac{n}{2}} pL(p)\left(C_e(n-p,p) + C(n-p,p)\right) \\ &= \sum_{p=1}^{\frac{n}{2}} L(p)\left(2p(n-2p+1)\sigma^{n-2p} - (2p-1)(n-2p)\sigma^{n-2p-1}\right). \end{aligned}$$

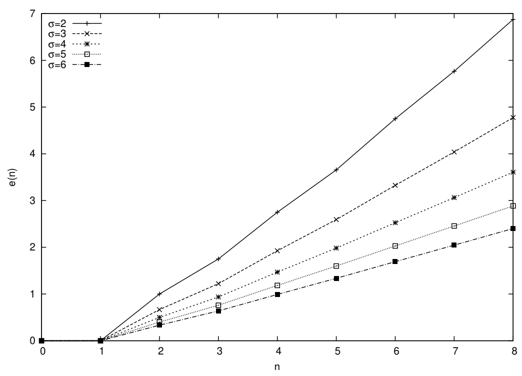


Figure 1. The average values $e(n)$ of sum of exponents of runs in strings on various sized alphabets.

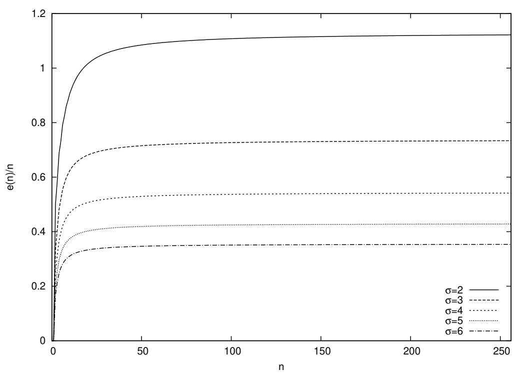


Figure 2. The average values $\frac{e(n)}{n}$ per unit length of sum of exponents of runs in strings on various sized alphabets.

We now get the Theorem 2. In Figure 1 it is shown that the average value $e(n)$ grows almost linearly, as n increases. The convergence of $\frac{e(n)}{n}$ is illustrated in Figure 2. For the limit of $e(n)$, we get the following theorem.

Theorem 10. *The limit of* $\frac{e(n)}{n}$, *as* $n \to \infty$, *is*

$$\sum_{d=1}^{\infty} \mu(d) \left(\frac{2(\sigma-1)}{\sigma^{2d}-\sigma} + \frac{1}{d\sigma} \ln \left(\frac{\sigma^{2d}}{\sigma^{2d}-\sigma} \right) \right).$$

To prove the theorem we deform $\frac{e(n)}{n}$.

Proposition 11.

$$\frac{e(n)}{n} = \sum_{d=1}^{\frac{n}{2}} \mu(d) \sum_{p=1}^{\frac{n}{2d}} \sigma^{-2pd+p-1} \left(2(\sigma-1) - \frac{4pd}{n}(\sigma-1) + \frac{1}{pd} + \frac{2}{n}(\sigma-1) \right)$$

Proof. Let $2p(n-2p+1)\sigma^{-2p} - (2p-1)(n-2p)\sigma^{-2p-1}$ be $f(p)$.

$$\begin{aligned} e(n) &= \sum_{p=1}^{\frac{n}{2}} L(p) f(p) \\ &= \sum_{p=1}^{\frac{n}{2}} \sum_{d|p} \mu\left(\frac{p}{d}\right) \sigma^d \frac{f(p)}{p} \\ &= \begin{array}{llllll} \mu(1)\sigma^1\frac{f(1)}{1} & & & & & \\ +\mu(2)\sigma^1\frac{f(2)}{2} & +\mu(1)\sigma^2\frac{f(2)}{2} & & & & \\ +\mu(3)\sigma^1\frac{f(3)}{3} & & +\mu(1)\sigma^3\frac{f(3)}{3} & & & \\ +\mu(4)\sigma^1\frac{f(4)}{4} & +\mu(2)\sigma^2\frac{f(4)}{4} & & +\mu(1)\sigma^4\frac{f(4)}{4} & & \\ +\mu(5)\sigma^1\frac{f(5)}{5} & & & & +\mu(1)\sigma^5\frac{f(5)}{5} & \\ +\mu(6)\sigma^1\frac{f(6)}{6} & +\mu(3)\sigma^2\frac{f(6)}{6} & +\mu(2)\sigma^3\frac{f(6)}{6} & & & +\mu(1)\sigma^6\frac{f(6)}{6} \\ & & \vdots & & & \end{array} \end{aligned}$$

$$
\begin{aligned}
&= \sum_{d=1}^{\frac{n}{2}} \mu(d) \sum_{p=1}^{\frac{n}{2d}} \sigma^p \frac{f(pd)}{pd} \qquad \text{(Factor } \mu(d) \text{ out)} \\
&= \sum_{d=1}^{\frac{n}{2}} \mu(d) \sum_{p=1}^{\frac{n}{2d}} \frac{1}{pd} \left(2pd(n-2pd+1)\sigma^{-2pd+p} - (2pd-1)(n-2pd)\sigma^{-2pd+p-1}\right) \\
\frac{e(n)}{n} &= \sum_{d=1}^{\frac{n}{2}} \mu(d) \sum_{p=1}^{\frac{n}{2d}} \frac{1}{npd} \left(2pd(n-2pd+1)\sigma^{-2pd+p} - (2pd-1)(n-2pd)\sigma^{-2pd+p-1}\right) \\
&= \sum_{d=1}^{\frac{n}{2}} \mu(d) \sum_{p=1}^{\frac{n}{2d}} \sigma^{-2pd+p-1} \left(2(\sigma-1) - \frac{4pd}{n}(\sigma-1) + \frac{1}{pd} + \frac{2}{n}(\sigma-1)\right)
\end{aligned}
$$

□

Now we prove Theorem 10.

Proof. When $n \to \infty$, $\frac{1}{n} \to 0$ and $\sigma^{-2pd+p-1}\frac{4pd}{n}$ is also negligible because $\sigma^{-2pd+p-1}$ is small enough when $\frac{pd}{n}$ is considerable.

$$
\begin{aligned}
\lim_{n\to\infty} \frac{e(n)}{n} &= \lim_{n\to\infty} \sum_{d=1}^{\frac{n}{2}} \mu(d) \sum_{p=1}^{\frac{n}{2d}} \sigma^{-2pd+p-1} \left(2(\sigma-1) + \frac{1}{pd}\right) \\
&= \lim_{n\to\infty} \sum_{d=1}^{\frac{n}{2}} \mu(d) \left(\frac{2\sigma^{-2d}(\sigma-1)}{1-\sigma^{1-2d}} - \frac{1}{d\sigma} \ln(1-\sigma^{1-2d})\right) \\
&= \sum_{d=1}^{\infty} \mu(d) \left(\frac{2(\sigma-1)}{\sigma^{2d}-\sigma} + \frac{1}{d\sigma} \ln\left(\frac{\sigma^{2d}}{\sigma^{2d}-\sigma}\right)\right)
\end{aligned}
$$

□

Table 1 shows the limit values of $\frac{e(n)}{n}$ and $\frac{r(n)}{n}$.

σ	2	3	4	5	6
$\lim_{n\to\infty} \frac{e(n)}{n}$	1.13103	0.73822	0.54459	0.43039	0.35536
$\lim_{n\to\infty} \frac{r(n)}{n}$	0.41165	0.30491	0.23736	0.19329	0.16268

Table 1. The limit values of $\frac{e(n)}{n}$ and $\frac{r(n)}{n}$ for various sized alphabets.

4 Conclusion

We showed a formula which expresses the average value of sum of exponents of runs in strings exactly, although the upper bound of it is still open. The situation is similar to the numbers of runs, as Puglisi and Simpson showed in [8]. Moreover we gave the limit of the value per unit length as the length of strings approaches infinity. For the alphabet size $\sigma = 2$, the value is approximately 1.13103.

References

1. M. Crochemore and L. Ilie: *Analysis of Maximal Repetitions in Strings*, in Proc. 32nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2007), vol. 4708 of LNCS, 2007, pp. 465–476.
2. M. Crochemore, L. Ilie, and L. Tinta: *The "runs" conjecture*. http://www.csd.uwo.ca/~ilie/runs.html.
3. M. Crochemore, L. Ilie, and L. Tinta: *Towards a solution to the "runs" conjecture*, in Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching (CPM 2008), vol. 5029 of LNCS, 2008, pp. 290–302.
4. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in Proc. 40th Annual Symposium on Foundations of Computer Science (FOCS'99), 1999, pp. 596–604.
5. R. Kolpakov and G. Kucherov: *On the sum of exponents of maximal repetitions in a word*, Tech. Rep. 99-R-034, LORIA, France, 1999.
6. M. Lothaire: *Algebraic combinatorics on words*, Cambridge University Press New York, 2002.
7. M. Lothaire: *Applied Combinatorics on Words*, Cambridge University Press, 2005.
8. S. J. Puglisi and J. Simpson: *The expected number of runs in a string.* Australasian Journal of Combinatorics, 2008, in press.
9. S. J. Puglisi, J. Simpson, and W. F. Smyth: *How many runs can a string contain?* Theoretical Computer Science, 2007, in press.
10. W. Rytter: *The number of runs in a string: Improved analysis of the linear upper bound*, in Proc. 23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS 2006), vol. 3884 of LNCS, 2006, pp. 184–195.
11. W. Rytter: *The number of runs in a string.* Information and Computation, 205(9) 2007, pp. 1459–1469.

Usefulness of Directed Acyclic Subword Graphs in Problems Related to Standard Sturmian Words

Paweł Baturo[1], Marcin Piątkowski[1], and Wojciech Rytter[2,1]⋆

[1] Faculty of Mathematics and Computer Science, Nicolaus Copernicus University
[2] Institute of Informatics, Warsaw University, Warsaw, Poland

Abstract. The class of finite Sturmian words consists of words having particularly simple compressed representation, which is a generalization of the Fibonacci recurrence for Fibonacci words. The subword graphs of these words (especially their compacted versions) have a very special regular structure. The regularity of their structure has been discovered in the context of the counting property of graphs. In this paper we investigate the structure of these subword graphs in more detail than in the previous papers. As an application we show how several syntactical properties of Sturmian words follow their graph properties. Alternative graph-based proofs of several known facts are presented. Also the neat structure of subword graphs of Sturmian words leads to algorithms computing several parameters (e.g. number of subwords, critical factorization point, short description of lexicographically maximal suffix, the structure of occurrences of subwords of a fixed length, right special factors) of standard Sturmian words in linear time with respect to the length n of the compressed representation: the directive sequence (though the words themselves can be of exponential size with respect to n). Some of the computed parameters can be of exponential size, however they have linear size grammar-based representation. This gives more examples of fast computations for highly compressed words.

1 Introduction

The *standard Sturmian words* (*standard words*, in short) are generalization of Fibonacci words and have a very simple *grammar-based* representation which has some algorithmic consequences.

Let $\mathcal{S}$ denote the set of all standard Sturmian words. These words are described by recurrences (or grammar-based representation) corresponding to so called *directive sequences*: integer sequences

$$\gamma = (\gamma_0, \gamma_1, \ldots, \gamma_n),$$

where $\gamma_0 \geq 0$, $\gamma_i > 0$ for $0 < i \leq n$. The word x_{n+1} corresponding to γ, denoted by Word(γ), is defined by recurrences:

$$x_{-1} = b, \quad x_0 = a, \quad \forall_{0 \leq i < n}\; x_{i+1} = x_i^{\gamma_i} x_{i-1} \tag{1}$$

Fibonacci words are standard Sturmian words given by directive sequences of the form

$$\gamma = (1, 1, \ldots, 1).$$

We consider here standard words starting with the letter a, hence assume $\gamma_0 > 0$. The case $\gamma_0 = 0$ can be considered similarly.

⋆ Supported by grant N206 004 32/0806 of the Polish Ministry of Science and Higher Education

Pawe l Baturo, Marcin Piątkowski, Wojciech Rytter: Usefulness of Directed Acyclic Subword Graphs in Problems Related to Standard Sturmian Words, pp. 193–207.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

For even $n > 0$ a standard word x_n has suffix ba, and for odd $n > 0$ it has suffix ab. The number $N = |x_{n+1}|$ is the (real) size, while $n + 1$ can be thought as the compressed size.

Example 1.
Consider directive sequence $\gamma = (1, 2, 1, 3, 1)$. We have:

$$\text{Word}(1,2,1,3,1) \;=\; ababaabababaabababaabababaababaab$$

$$x_{-1} = b, \quad x_0 = a, \quad x_1 = x_0^1 x_{-1} = ab, \quad x_2 = x_1^2 x_0 = ababa,$$
$$x_3 = x_2^1 x_1 = ababaab, \quad x_4 = x_3^3 x_2 = ababaababaabababaababaababa,$$
$$x_5 = x_4^1 x_3 = ababaabababaabababaabababaababaab$$

Some of the outputs of our algorithms will be given in the grammar-compressed form which consists in giving a context-free grammar G generating a single word x. The size of G is the total length of all productions of G.

In particular each directive sequence of a standard Sturmian word corresponds to such a compression – the sequence of recurrences corresponding to the directive sequence. In this case the size of the grammar is proportional to the length of the directive sequence.

For some lexicographic properties and structure of repetitions of standard Sturmian words see [3] and [2].

2 The structure of subword graphs of standard Sturmian words

Let $Subwords(x)$ be the set of all nonempty subwords of x. We distinguish some subwords as special ones:

- A **special prefix** of x is a prefix z of x such that $za, zb \in Subwords(x)$.
- A **basic prefix** of x is a proper nonempty prefix of the type $x_k^j x_{k-1}$, where $0 \le k \le n$ and $0 \le j \le \gamma_k$.
- A **basic subword** of x is a reverse of x_k, for some k. Denote $y_k = Reverse(x_k)$.

Denote by $BP(x)$ the set of basic prefixes of x and by $SP(x)$ the set of special prefixes of x. Denote by $\hat{x}$ the prefix of x of size 2, assuming $|x| \ge 2$. Assume that $\hat{y}_0 = ab$.

Lemma 1.
Assume $x_{-1}, x_0, \ldots, x_{n+1}$ is the sequence of standard Sturmian words given by $(\gamma_0, \gamma_1, \ldots, \gamma_n)$.
(a) *For $i \ge 1$ we can represent standard word x_i as*

$$x_i \;=\; y_0^{\gamma_0} y_1^{\gamma_1} \cdots y_{i-2}^{\gamma_{i-2}} y_{i-1}^{\gamma_{i-1}-1} \hat{y}_{i-1},$$

(b) *Each special prefix of x_n has the form*

$$y_0^{\gamma_0} y_1^{\gamma_1} \cdots y_i^{j},$$

where $0 \le j \le \gamma_i$ for $i < n-1$ and $0 \le j \le \gamma_i - 1$ for $i = n-1$,
(c) *Each special prefix results by cutting off two last symbols from a basic prefix.*

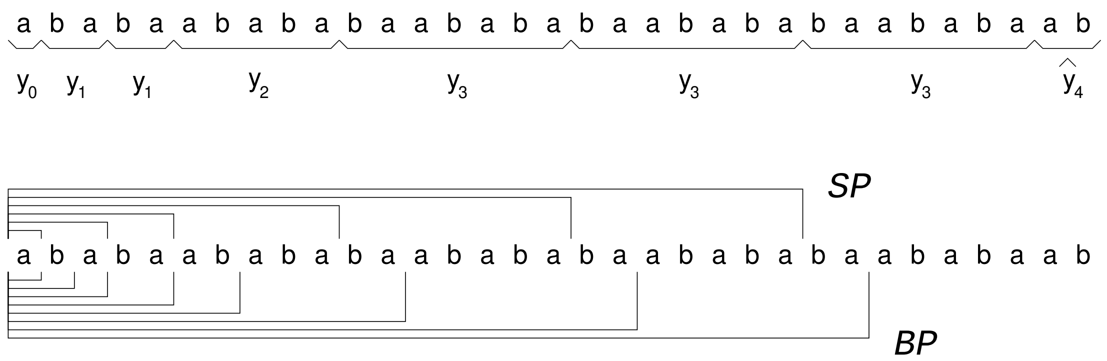


Figure 1. The structure of basic prefixes (BP), special prefixes (SP) and basic subwords of $Word(1,2,1,3,1)$.

Proof.

Point (a)

Notice that $\hat{y}_i = \hat{y}_{i+2}$ and $y_{i+1} = y_{i-1}y_i^{\gamma_i}$ for $i \geq 0$.

First we show by induction that

$$y_i = \hat{y}_i y_0^{\gamma_0} y_1^{\gamma_1} \cdots y_{i-1}^{\gamma_{i-1}-1}\,. \tag{2}$$

For $i = 1$ we have

$$y_1 = ba^{\gamma_0} = \hat{y}_1 y_0^{\gamma_0 - 1}$$

Assume that for $i \leq n$ the equation (2) is true. We have

$$\begin{aligned} y_{n+1} &= y_{n-1} \cdot y_n^{\gamma_n} \\ &= \left(\hat{y}_{n-1} y_0^{\gamma_0} y_1^{\gamma_1} \cdots y_{n-2}^{\gamma_{n-2}-1}\right) \cdot \left(y_{n-2} y_{n-1}^{\gamma_{n-1}} y_n^{\gamma_n - 1}\right) \\ &= \hat{y}_{n+1} y_0^{\gamma_0} y_1^{\gamma_1} \cdots y_n^{\gamma_n - 1} \end{aligned}$$

Now we can prove equation from the point (a) using induction. For $i = 1$ we have:

$$x_1 = x_0^{\gamma_0} x_{-1} = y_0^{\gamma_0 - 1} \hat{y}_0$$

Assume that for $i \leq n$ equation from the point (a) is true. We have

$$\begin{aligned} x_{n+1} &= x_n^{\gamma_n} x_{n-1} \\ &= \left(y_0^{\gamma_0} \cdots y_{n-2}^{\gamma_{n-2}} y_{n-1}^{\gamma_{n-1}-1} \hat{y}_{n-1}\right)^{\gamma_n} \cdot y_0^{\gamma_0} \cdots y_{n-2}^{\gamma_{n-2}-1} \hat{y}_{n-2} \\ &\overset{\text{due to (2)}}{=} y_0^{\gamma_0} \cdots y_{n-1}^{\gamma_{n-1}} y_n^{\gamma_n - 1} \hat{y}_n \end{aligned}$$

Point (b).

Let $\overline{w}$ denotes here a word w with removed last two letters and assume that w contains at least two letters.

From point (a) we know that

$$z = y_0^{\gamma_0} y_1^{\gamma_1} \cdots y_i^{j}$$

is a prefix of standard word x_n generated by directive sequence $(\gamma_0, \gamma_1, \ldots, \gamma_n)$, where $0 \leq j \leq \gamma_i$ for $i < n-1$ and $0 \leq j \leq \gamma_i - 1$ for $i = n-1$. We can also deduce, that prefix $\overline{x_n}$ is a palindrome (see [4] for proof that every standard word x a word $\overline{x}$ is a palindrome). Hence, if z is special prefix of standard word x, then z is also suffix of $\overline{x}$.

First assume that $i < n-1$ and i is odd, the case for even i is similar.

If $0 \le j < \gamma_i$, then z is prefix of x_{i+2} and zb is also prefix of x_{i+2} (first letter of y_i is b). Suffix of x_{i+2} is ab, hence za, as a suffix of $\overline{x_{i+2}}$, is also subword of x_{i+2}.

If $j = \gamma_i$, then z is prefix of x_{i+3} and za is also prefix of x_{i+3} (first letter of y_{i+1} is a). Suffix of x_{i+3} is is ba, hence zb, as a suffix of $\overline{x_{i+3}}$, is also subword of x_{i+3}.

Now assume that $i = n-1$. For $0 \le j < \gamma_{n-1}$ proof is similar to the case $i < n-1$. It is obvious, due to the deduction above, that for $i = n-1$, j must be less than γ_{n-1}.

Point (c)
Notice that $\hat{y}_i = \hat{y}_{i+2}$ and $y_{i+1} = y_{i-1}y_i^{\gamma_i}$ for $i \ge 0$.

From (a) for basic prefix $x_k^j x_{k-1}$ we have:

$$\begin{aligned} x_k^j x_{k-1} &= \left(y_0^{\gamma_0} \cdots y_{k-2}^{\gamma_{k-2}} y_{k-1}^{\gamma_{k-1}-1} \hat{y}_{k-1}\right)^j \cdot y_0^{\gamma_0} \cdots y_{k-3}^{\gamma_{k-3}} y_{k-2}^{\gamma_{k-2}-1} \hat{y}_{k-2} \\ &\overset{\text{due to (2)}}{=} y_0^{\gamma_0} \cdots y_{k-1}^{\gamma_{k-1}} y_k^{j-1} \hat{y}_k \end{aligned}$$

From (b) we have that basic prefix $x_k^j x_{k-1}$ with last two letters removed ($\hat{y}_k$) is special prefix.

Example 2.
For Word$(1,2,1,3,1) = ababaababababaababababaababababaababaab$ we have:

$$\begin{aligned} &BP = \{x_0,\ x_1,\ x_1x_0,\ x_2,\ x_3,\ x_3x_2,\ x_3^2x_2,\ x_4\} \\ &SP = \{y_0,\ y_0y_1,\ y_0y_1^2,\ y_0y_1^2y_2,\ y_0y_1^2y_2y_3,\ y_0y_1^2y_2y_3^2\} \\ &y_0 = a \qquad y_1 = ba \qquad y_2 = ababa \qquad y_3 = baababa \\ &\text{Word}(1,2,1,3,1) = a\ \ ba\ \ ba\ \ ababa\ \ baababa\ \ baababa\ \ baababa\ \ ab \\ &\qquad\qquad\qquad\quad = y_0\ y_1^2\ y_2\ y_3^3\ \hat{y}_4 \end{aligned}$$

The subword graph is a classical data structure representing all subwords of a given word in a succinct manner. More precisely: the *Directed Acyclic Word Graph* (dawg in short) of the word w is the minimal deterministic automaton (not necessarily complete) that accepts all suffixes of w. We refer the reader to [6] for the complete definition and more information of subword graphs.

The compacted subword graph (cdawg, in short) results from the subword graph by removing all nodes of out-degree one (except the source node and the terminal nodes) and replacing each chain by a single edge with the label representing the path label of this chain. Internal nodes of dawg of out-degree greater than one, which are copied to cdawg, are called fork nodes. In case of standard words the subword graph can be considerably compressed.

The regularity of the structure of compacted subword graphs has been discovered in [8]. The following theorem follows from the results of [8], Lemma 1 and our terminology.

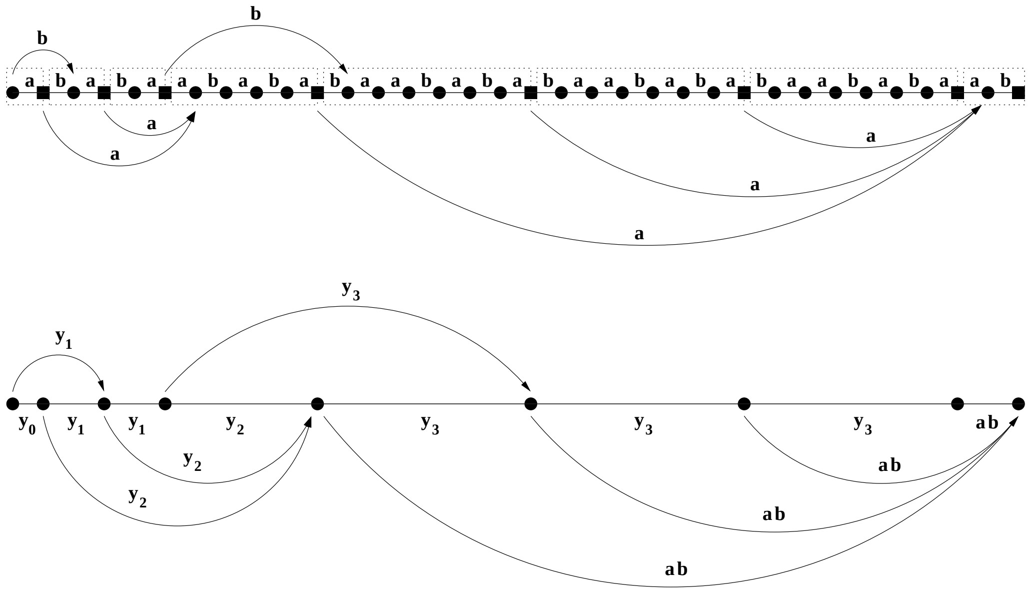


Figure 2. The structure of the subword graph (dawg) of $Word(1, 2, 1, 3, 1)$ and its compacted version (cdawg)

Theorem 2.
Let $w = \mathrm{Word}(\gamma_0, \gamma_1, \ldots, \gamma_n)$ *be a standard Sturmian word.*
(1) *The labels of edges in compacted subword graph of* w *are basic subwords of* w.
(2) *The compacted subword graph of* w *has the structure illustrated on Figure 3.*

3 The number of subwords

It is known that the number of distinct subwords in the n-th Fibonacci word is

$$Subwords(Fib_{n+1}) \;=\; |Fib_n| \cdot |Fib_{n-1}| + 2 \cdot |Fib_n| - 1$$

Surprisingly essentially the same formula works generally for Sturmian words.

Theorem 3. *Let* $\gamma_n = 1$, *and* $x_{n+1} \;=\; \mathrm{Word}(\gamma_0, \gamma_1, ..., \gamma_n)$, *then*

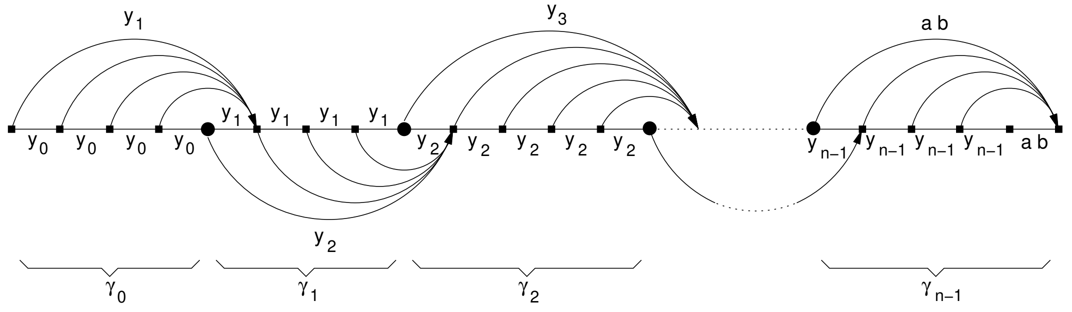
$$\left|Subwords(x_{n+1})\right| \;=\; |x_n| \cdot |x_{n-1}| + 2 \cdot |x_n| - 1$$

Proof.
Denote by v_0 the source node of the compacted subword graph for x_{n+1}. Let $t_k = |x_k|$. Define the multiplicity $mult(v)$ of a vertex v as the number of paths $v_0 \xrightarrow{*} v$, and the weights of edges as lengths of corresponding label-strings of these edges in the compacted subword graphs. Let $edges(v)$ be the sum of all weight edges outgoing from v.

Claim. Let $w \;=\; \mathrm{Word}(\gamma_0, \gamma_1, ..., \gamma_n)$. Then

$$\left|Subwords(w)\right| \;=\; \sum_{v \in G} mult(v) \cdot edges(v) \tag{3}$$

Case 1: $\gamma_n = 1$



Case 2: $\gamma_n > 1$

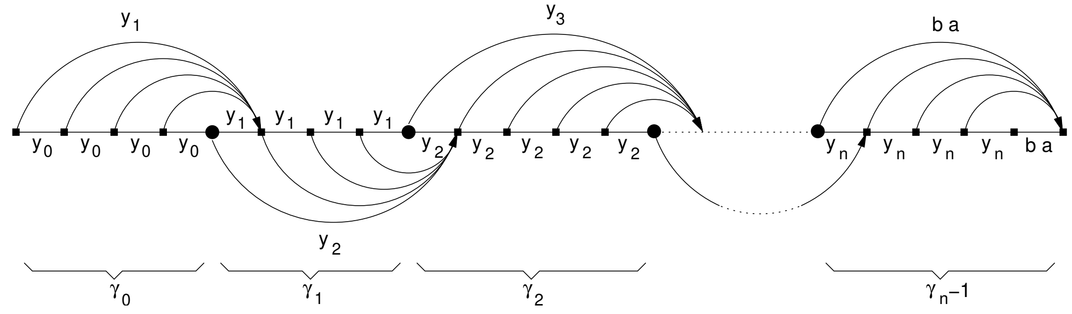


Figure 3. Compacted subwords graphs for words: Word$(\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_n)$ and Word$(\gamma_0, \gamma_1, \gamma_2, \ldots, \gamma_n - 1, 1)$ are isomorphic (in the sense of graph structure).

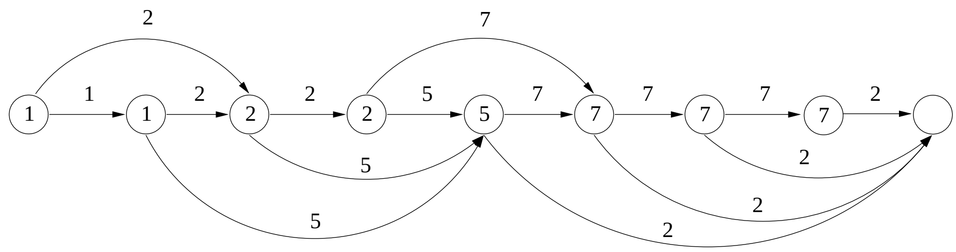


Figure 4. The structure of edge-lengths and multiplicities of nodes in the compacted subword graph of $Word(1, 2, 1, 3, 1)$. According to the Theorem 3 (and to the graph above) there are $|x_4| \cdot |x_3| + 2 \cdot |x_4| - 1 \ = \ 26 \cdot 7 + 2 \cdot 26 - 1 \ = \ 233$ subwords in our example word.

See Figure 4 for edge-lengths and node-multiplicities structure in the cdawg of example word.
We partition the set of edges into chunks, the first chunk consists of the first γ_0 consecutive vertices starting from the v_0, the second chunk contains the next γ_1 vertices, etc. The last chunk slightly differs.

The contribution of k-th internal chunk in the sum in equation (3) is

$$\big(t_{k-1} + (\gamma_k - 1)t_k\big) \cdot (t_k + t_{k+1}) \ = \ t_{k+1}^2 - t_k^2,$$

where $t_{-1} = 1$ (see Figure 5 for details).

The contribution of the last chunk is (see Figure 6)

$$(t_{n-1}+2)(t_n-t_{n-1})+2t_{n-1}.$$

Altogether we have

$$\sum_{k=0}^{n-2}\left(t_{k+1}^2-t_k^2\right)+(t_{n-1}+2)(t_n-t_{n-1})+2t_{n-1} \;=\; t_n\cdot t_{n-1}+2\cdot t_n-1$$

This completes the proof, since by definition $|x_k| = t_k$.

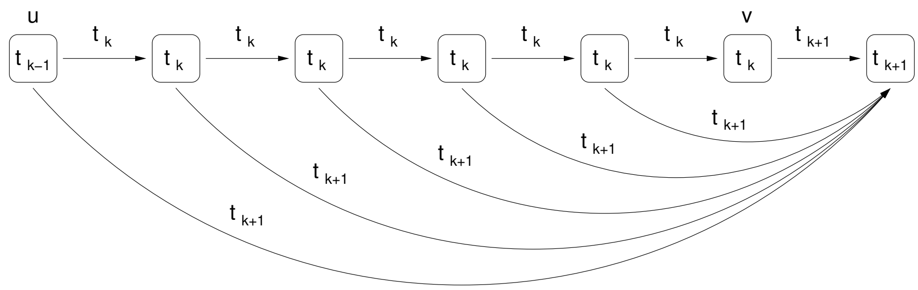


Figure 5. The k-th internal chunk G_k of the subword graph, consists of γ_k nodes from u to v (excluding u), and their outgoing edges. The multiplicity (number of path leading from v_0) of each node is written within the box corresponding to the node. The weight of the edges are the lengths of corresponding words in the cdawg.

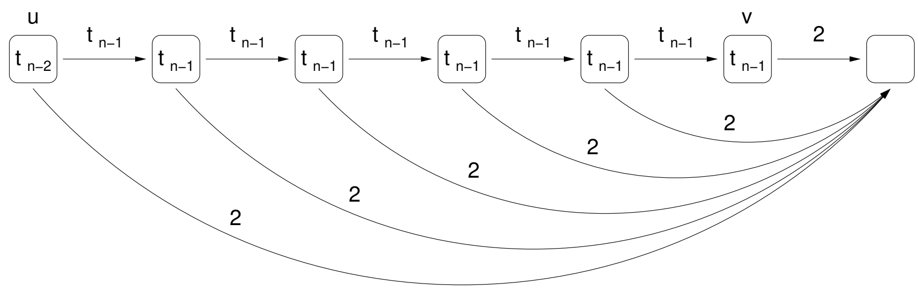


Figure 6. The final chunk G_{n-1} of the subword graph, consists of γ_{n-1} nodes from u to v, and their outgoing edges.

The case $\gamma_n > 1$ reduces to the previous case.

Theorem 4. *Let $\gamma_n > 1$. Then:*

$$\left|Subwords\big(\mathrm{Word}(\gamma_0,\gamma_1,...,\gamma_n)\big)\right| \;=\; \left|Subwords\big(\mathrm{Word}(\gamma_0,\gamma_1,...,\gamma_n-1,1)\big)\right|.$$

Proof.
Compacted subword graphs of $\mathrm{Word}(\gamma_0,\gamma_1,...,\gamma_n)$ and $\mathrm{Word}(\gamma_0,\gamma_1,...,\gamma_n-1,1)$ are isomorphic in the sense of graph structure (see Figure 3 for details). Hence we can use the result of Theorem 3 to compute $\left|Subwords\big(\mathrm{Word}(\gamma_0,\gamma_1,...,\gamma_n)\big)\right|$.

4 The structure of occurrences of subwords

In this section we are interested in the structure of first occurrences of the subwords of a given length. One type of these subwords is particularly interesting – a right special factors.

A **right special factor** of the word x is any word w such that both wa, wb are subwords of x. For each $k > 0$ there is at most one right special factor of length k of a given standard word. For standard Sturmian word x every right special factor is either special prefix or suffix of some special prefix.

Theorem 5. *Let $w = \mathrm{Word}(\gamma)$ be a standard Sturmian word. Then:*
(1) *For a given $k > 0$ the right special factor of w of length k has grammar-representation of size $O\big(|\gamma|\big)$.*
(2) *The compressed representation of the right special factor of w of length k can be computed in $O\big(|\gamma|\big)$ time.*

Proof.
Define length of the path in cdawg of w as number of edges in it and value of the path as word created by concatenation of the labels of edges in it.

Let v be an internal node in compacted subwords graph of w and z_π be a value of path π in this graph leading from root to v. It is clear that z_π is a subword of w.

Every internal node in compacted subword graph is a fork node, hence v has two outgoing edges: one with label starting with letter a and the second with label starting with letter b. This follows that $z_\pi \cdot a$ and $z_\pi \cdot b$ are also subwords of w and therefore z_π is a right special factor of w.

Observe that value of every path from root to v in cdawg of w is suffix of the value of the longest path from root to v. Moreover value of the longest path from root to v is a prefix of w, hence it is a special prefix of w. This implies that every right special factor of w is suffix of some special prefix of w.

Every right special factor of w is concatenation of some basic subwords of w. It follows easily from Lemma 1 that every right special factor of w has grammar-representation of size $O\big(|\gamma|\big)$ which can be computed in time linear to the length of directive sequence γ.

Example 3.
Let $w = \mathrm{Word}(1, 2, 1, 3, 1) = ababaababababaabababaababaababaab$.
Recall that:

$$y_0 = a \qquad y_1 = ba \qquad y_2 = ababa \qquad y_3 = baababa$$

Right special factors of w with their lengths are (special prefixes are bold):

								11	$y_1^2y_3$	18	$y_1^2y_3^2$
								12	y_2y_3	19	$y_2y_3^2$
						6	y_0y_2	13	$y_0y_2y_3$	20	$y_0y_2y_3^2$
						7	y_1y_2	14	$y_1y_2y_3$	21	$y_1y_2y_3^2$
						8	$y_0y_1y_2$	15	$y_0y_1y_2y_3$	22	$y_0y_1y_2y_3^2$
		2	y_1	4	y_1^2	9	$y_1^2y_2$	16	$y_1^2y_2y_3$	23	$y_1^2y_2y_3^2$
1	$\mathbf{y_0}$	3	$\mathbf{y_0y_1}$	5	$\mathbf{y_0y_1^2}$	10	$\mathbf{y_0y_1^2y_2}$	17	$\mathbf{y_0y_1^2y_2y_3}$	24	$\mathbf{y_0y_1^2y_2y_3^2}$

See Figure 2 for the structure of cdawg of the word w.

For a set X of integers and an integer k define

$$X \oplus k \;=\; \{\; x + k \;:\; x \in X \;\}$$

Let $occ(u, w)$ be the set of first positions of occurrences of u in w, we define also the set of final positions of occurrences of a word u :

$$fin(u, w) = occ(u, w) \oplus |u| \quad \text{and} \quad \textit{first-fin}(u, w) = \mathbf{min}\big(fin(u, w)\big).$$

For $k \geq$ we investigate also the structure of the set

$$FIN(k, w) \;=\; \big\{\textit{first-fin}(u, w) \;:\; u \;\text{ is a subword of }\; w \;\text{ of size }\; k\;\big\}.$$

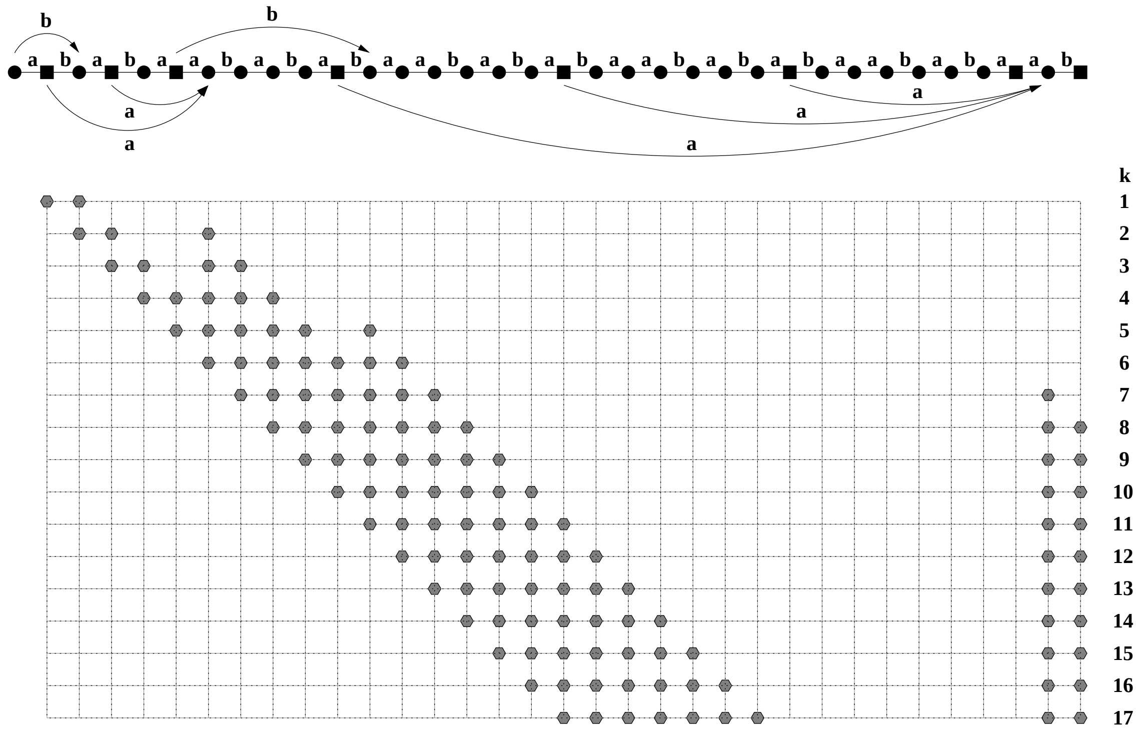


Figure 7. The subword graph of w and the structure of the sets $FIN(k, w)$ for $w = \text{Word}(1, 2, 1, 3, 1)$.

Theorem 6. *Let $w = \text{Word}(\gamma_0, \gamma_1, \ldots, \gamma_n)$ be a standard Sturmian word. Then:*
(1) *The set $FIN(k, w)$ consists of a single interval or of two disjoint intervals.*
(2) *For a given k we can compute the intervals representing $FIN(k, w)$ in linear time with respect to the size of the directive sequence.*

Proof.
The structure of the set $FIN(k, w)$ easily follows from the way how paths of length $k-1$ in dawg of w are extended into path of length k. Only fork nodes $i \in FIN(k-1, w)$ generate two elements of $FIN(k, w)$, each other node $i \in FIN(k-1, w)$ generates single element $i + 1$ in $FIN(k, w)$ (see Figure 7).

It is clear that the set $FIN(k+1, w)$ results from $FIN(k, w)$ by shifting each position by one to the right and adding an extra position for the fork node. Hence thesis follows from the structure of subword graphs of standard Sturmian words.

5 Relation of subword graphs to the dual Ostrovski numeration system

The dual Fibonacci numeration system has been introduced in [10], where its relation to the subword structure of Fibonacci words has been investigated. We extend these results to Sturmian words. In this case we have Ostrovski numeration system which is a generalization of Fibonacci system.

In (only) this section we consider infinite directive sequences.

For an infinite directive sequence $\gamma = (\gamma_0, \gamma_1, \ldots)$ we introduce $[*]_\gamma$-numeration system: a version of Ostrowski's numeration system from [1] which is a generalization of the Fibonacci number system. Let us define the *base* sequence q as a sequence:

$$q \ = \ (q_0, q_1, \ldots) \ = \ \big(|x_0|, |x_1|, ...\big),$$

where x_i's are as in equation (1).

The base sequence can be defined without reference to words x_i as follows:

$$q_{-1} = q_0 = 1, \ \ q_{i+1} = q_i \cdot \gamma_i + q_{i-1} \text{ for } i \geq 0.$$

Example 4.
If $\gamma = (1, 2, 1, 2, \ldots)$, then the base sequence is:

$$q = (1, 2, 5, 7, 19, \ldots)$$

If $\gamma = (1, 2, 1, 1, 1, \ldots)$, then the base sequence is:

$$q = (1, 2, 5, 7, 12, 19, \ldots)$$

Define:

$$\text{val}_\gamma(\alpha_0, \alpha_1, \ldots, \alpha_n) \ = \ \alpha_0 \cdot q_0 + \alpha_1 \cdot q_1 + \ldots + \alpha_n \cdot q_n$$

For $0 \leq i < |x_n|$ the representation of i in Ostrovski numeration system is defined as follows:

$$[i]_\gamma \ = \ (\alpha_0, \alpha_1, \ldots, \alpha_n),$$

where we require:

(1) $\text{val}_\gamma(\alpha_0, \alpha_1, \ldots, \alpha_n) = i$
(2) $\forall_{0 \leq j < n} \ \alpha_j \leq \gamma_j$
(3) $\alpha_{j+1} = \gamma_{j+1}{:}\alpha_j = 0$

In other words in the representation of a number i we take at most γ_k numbers $|x_k|$, for each k, and if we take exactly γ_k numbers $|x_k|$ then we take zero numbers $|x_{k-1}|$.

Example 5.
Let $\gamma = (1, 2, 1, 3, 1, \ldots)$. Then

$$q = \big(|x_0|, |x_1|, ...\big) = (1, 2, 5, 7, 26, 33, \ldots)$$

We have $[29]_\gamma = (1, 1, 0, 0, 1)$, because

$$29 = 1 \cdot 1 + 1 \cdot 2 + 0 \cdot 5 + 0 \cdot 7 + 1 \cdot 26$$

We have $[58]_\gamma = (0, 2, 0, 3, 0, 1)$, because

$$58 \;=\; 0 \cdot 1 + 2 \cdot 2 + 0 \cdot 5 + 3 \cdot 7 + 0 \cdot 26 + 1 \cdot 33$$

For $0 \leq i < |x_n|$ we define representation of i in the dual Ostrovski numeration system as:

$$\hat{[i]}_\gamma = (\alpha_0, \alpha_1, \ldots, \alpha_n),$$

where:

(1) $\mathrm{val}_\gamma(\alpha_0, \alpha_1, \ldots, \alpha_n) = i$
(2) $\forall_{0 \leq j < n}\ \alpha_j \leq \gamma_j$
(3) $\big(\ \alpha_j < \gamma_j$ and $\exists\ (i > j)\ \alpha_i > 0\ \big)$:$\alpha_{j+1} > 0$

In other words in the representation of a number i in numeration system defined above we take at most γ_k numbers $|x_k|$, and if we take $\alpha_k < \gamma_k$ numbers $|x_k|$ and α_k is not the last one component of this representation then we must take at least one number $|x_{k+1}|$.

Example 6.
Let $\gamma = (1, 2, 1, 3, 1, \ldots)$. Then

$$q = \big(|x_0|, |x_1|, ...\big) = (1, 2, 5, 7, 26, 33, \ldots)$$

We have $\hat{[29]}_\gamma = (1, 1, 1, 3)$, because

$$29 = 1 \cdot 1 + 1 \cdot 2 + 1 \cdot 5 + 3 \cdot 7$$

We have $\hat{[58]}_\gamma = (0, 2, 0, 3, 0, 1)$, because

$$58 \;=\; 0 \cdot 1 + 2 \cdot 2 + 0 \cdot 5 + 3 \cdot 7 + 0 \cdot 26 + 1 \cdot 33$$

Uniqueness of representation in Ostrovski numeration system was proved in [1]. Uniqueness of representation in dual Ostrovski numeration system was proved in [8].

Let $\mathcal{G}_\infty$ be the infinite compacted subword graph corresponding to a given directive sequence $\gamma = (\gamma_0, \gamma_1, \ldots)$.

The following fact is an interpretation of the corresponding result in [8] in terms of the dual Ostrovski numeration system.

Theorem 7.

(1) *Let π be a path from the root to another node of $\mathcal{G}_\infty$. Let* $\mathrm{rep}(\pi) = (h_0, h_1, \ldots)$, *where h_i is the number of edges of weight q_i on the path π. Then* $\mathrm{rep}(\pi)$ *is the representation of the length $|\pi|$ of this path in the dual Ostrovski numeration system corresponding to the directive sequence of $\mathcal{G}_\infty$.*
(2) *For each $k > 1$ there is exactly one fork-path of length k in $\mathcal{G}_\infty$.*

Proof.
Point (1)
Let π be a path from root to some node v in $\mathcal{G}_\infty$ – infinite compacted subwords graph corresponding to directive sequence $(\gamma_0, \gamma_1, \gamma_2, \ldots)$, and let $\mathrm{rep}(\pi) = (h_0, h_1, \ldots)$ be

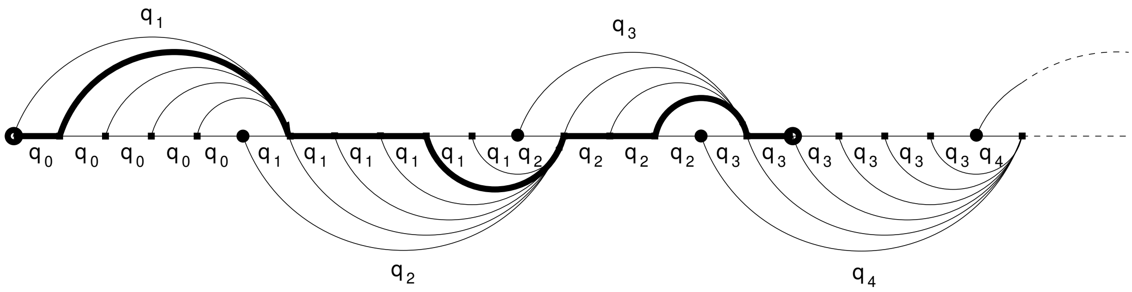


Figure 8. The illustration of the point (1) of Theorem 7. In this case representation of the length of the path π in dual Ostrovski numeration system is given by: $\mathrm{rep}(\pi) = (1, 4, 3, 2)$ and $|\pi| = 1 \cdot |q_0| + 4 \cdot |q_1| + 3 \cdot |q_2| + 2 \cdot |q_3|$.

defined as above. It is sufficient to check if requirements of definition of dual Ostrovski numeration system are satisfied.

Construction of π implies that

$$|\pi| = h_0 \cdot q_0 + h_1 \cdot q_1 + h_2 \cdot q_2 + \cdots$$

and $\forall_i \ 0 \leq h_i \leq \gamma_i$. Moreover from $\mathcal{G}_\infty$ structure (see Figure 8) it is obvious that if $h_i < \gamma_i$ (we have taken q_i less than γ_i times) and h_i is not the last non zero element in $\mathrm{rep}(\pi)$ then $h_{i+1} > 0$ (we must take at least one q_{i+1} to continue construction of π). This concludes the proof of point (1).

Point (2) follows directly from point (1) and uniqueness of representation in dual Ostrovski numeration system.

Ostrovski automata

For a directive sequence $\gamma = (\gamma_0, \gamma_1, \ldots, \gamma_n)$ we define $SD(\gamma)$ as the set of representations $(i_0, i_1, \ldots, i_n)$ in the dual Ostrovski numeration system of all numbers not exceeding the number written as γ in this representation.

Remark.
Observe that for any symbol a the value of a^0 is an empty word.

Denote

$$L(\gamma) = \{a_0^{i_0} a_1^{i_1} ... a_n^{i_n} \ : \ (i_0, i_1, \ldots, i_n) \in SD(\gamma)\}$$

for alphabet $\Sigma = \{a_0, a_1, \ldots, a_n\}$.

The minimal deterministic finite automaton accepting language $L(\gamma)$ is called the Ostrovski automaton and denoted by $OA(\gamma)$.

Theorem 8. *The minimal Ostrovski automaton for γ, without the dead state, is isomorphic as a graph to the compact directed acyclic subword graph of* $\mathrm{Word}(\gamma)$.

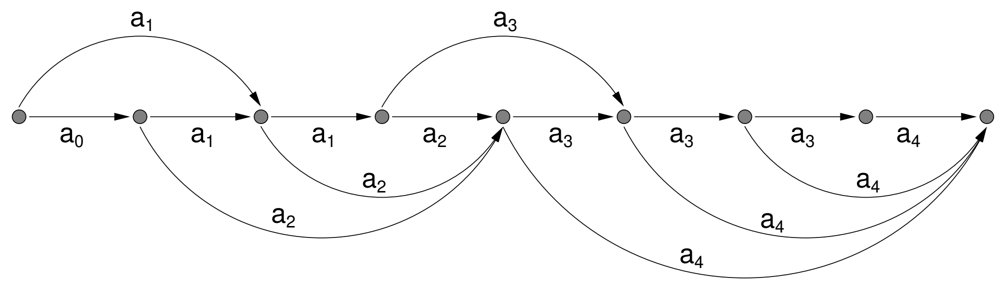


Figure 9. Minimal deterministic automaton (without dead state) $OA(1,2,1,3,1)$ accepting the set of strings $a_0^{i_0}a_1^{i_1}a_2^{i_2}a_3^{i_3}a_4^{i_4}$, where $(i_0, i_1, \ldots, i_4)$ is a representation in the dual Ostrovski numeration system of a natural number.

6 Critical factorization and maximal suffixes

The **minimal local period** in a word w at position k is a positive integer p such that $w[i-p] = w[i]$ for every $k \le i < k+p$, where $w[i]$ and $w[i-p]$ are defined.

The **critical factorization point** in a word w is position k in w for which minimal local period at k equals the (global) minimal period of w. We refer the reader to [6] for the more detailed definition of the critical factorization point.

The following nontrivial fact was shown by Crochemore and Perrin [5].

Fact 1
The critical factorization point of w is given as the starting position of a lexicographically maximal suffix, maximized over all possible orders of the alphabet.

Example 7
Let $w = \text{Word}(1,2,1,3,1) = ababaababababaababababaabababaababaab$.
Minimal local periods of w are as follows:

i		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	......
		a	b	a	b	a	a	b	a	b	a	b	a	a	b	a	b	a	......
$p(i)$	1	2	2	2	5	1	7	2	2	2	2	7	1	7	2	2	2	2	......

i	......		18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
	......		b	a	a	b	a	b	a	b	a	a	b	a	b	a	a	b
$p(i)$	......	2	7	1	7	2	2	2	4	**33**	1	5	2	2	5	1	3	1

where i denotes position in w and $p(i)$ – minimal local period at position i in w.
Hence critical factorization point is at position $i = 25$.

For a word w define $\pi_a(w)$ as a path in the dawg of w which starts in the root, ends in the sink, and in which we use the letter a whenever we have a choice (in every fork node). Similarly define $\pi_b(w)$. Path $\pi_a(w)$ ($\pi_b(w)$ respectively) can be also defined for cdawg of w: in every fork node we choose the edge with label starting with letter a (letter b respectively). Length of the path, denoted $|\pi|$, is is defined as length of the word given by π.

It is easily seen that lexicographically maximal suffix of w with respect to the letter ordering "$a < b$" is given by $\pi_b(w)$ and the lexicographically maximal suffix of w with respect to the letter ordering "$a > b$" is given by $\pi_a(w)$.

Lemma 9.
Let $w = \mathrm{Word}(\gamma_0, \gamma_1, \ldots, \gamma_n)$ be a standard Sturmian word and $\pi_a(w)$, $\pi_b(w)$ be defined as above. Then:

$$\pi_a(w) = y_0^{\gamma_0} y_2^{\gamma_2} \cdots y_{2k}^{\gamma_{2k}} \cdot \hat{y}_{n-1}$$
$$\pi_b(w) = y_1^{\gamma_1} y_3^{\gamma_3} \cdots y_{2l+1}^{\gamma_{2l+1}} \cdot \hat{y}_{n-1}$$

where $k = \lfloor \frac{n-1}{2} \rfloor$ and $l = \lfloor \frac{n-2}{2} \rfloor$.

Proof.
Recall that definition of basic subwords follows that y_i starts with letter a for even i and y_i starts with letter b for odd i.

We are constructing path $\pi_a(w)$ in cdawg of w by choosing edge with label starting with letter a whenever it is possible. From structure of cdawgs of standard Sturmian words (see Figure 3) we have that every fork node has two outgoing edges: one with label y_{2i} (starting with letter a) and second with label y_{2i+1} (starting with letter b).

To construct $\pi_a(w)$ we have to choose γ_0 times edge with label y_0, then γ_2 times edge with label y_2, and so on up to y_{2k}, where $k = \lfloor \frac{n-1}{2} \rfloor$. Finally, by Lemma 1, it suffices to add $\hat{y}_{n-1}$, the last two letters of w.

The same proof works for path $\pi_b(w)$.

Construction of paths $\pi_a(w)$ and $\pi_b(w)$ implies the following fact.

Theorem 10.
Let $w = \mathrm{Word}(\gamma_0, \gamma_1, \ldots, \gamma_n)$ be a standard Sturmian word. Then:
(1) *The critical factorization point of w is at position*

$$k = |w| - \boldsymbol{min}\{\ |\pi_a(w)|,\ |\pi_b(w)|\ \}$$

(2) *The critical factorization point of w can be computed in linear time with respect to the size of the directive sequence.*

Proof.
The proof is immediate by Fact 1 and recalling that $\pi_a(w)$ and $\pi_b(w)$ corresponds to lexicographically maximal suffixes of w with respect to letter orderings "$a > b$" and "$a < b$" respectively.

Example 8.
Let $w = \mathrm{Word}(1, 2, 1, 3, 1) = ababaababababaabababaababababaababaab$.
See Figure 2 for its subword graph structure.
We have

$$\pi_a(w) = y_0 y_2 ab = a\ ababa\ ab$$
$$\pi_b(w) = y_1^2 y_3^3 ab = ba\ ba\ baababa\ baababa\ baababa\ ab$$

Hence the position

$$i = |w| - |y_0 y_2 ab| = 33 - 8 = 25$$

is the critical factorization point of w.

Similar computations were given in [7,9] for Fibonacci words. The paths $\pi_a(w)$ and $\pi_b(w)$ have regular structure, consequently the words represented by them are well compressible. This implies the following fact.

Theorem 11. *Let $w = \mathrm{Word}(\gamma)$ be a standard Sturmian word. Then:*
(1) *The lexicographically maximal suffix of w has grammar-based representation of size $O\big(|\gamma|\big)$.*
(2) *The compressed representation of the lexicographically maximal suffix of w can be computed in $O\big(|\gamma|\big)$ time.*

Proof.
The lexicographically maximal suffix of a standard Sturmian word w is given either by path $\pi_a(w)$ or by path $\pi_b(w)$ (depending on which letter ordering was chosen). The thesis follows directly from the structure of $\pi_a(w)$, $\pi_b(w)$ and the subword graph of w (see Lemma 9).

References

1. J. Allouche and J. Shallit: *Automatic Sequences: Theory, Applications, Generalizations*, Cambridge University Press, 2003.
2. P. Baturo, M. Piątkowski, and W. Rytter: *The number of runs in Sturmian words*, CIAA, 2008.
3. P. Baturo and W. Rytter: *Occurrence and lexicographic properties of standard Sturmian words*, LATA, 2007.
4. J. Berstel and P. Seebold: *Sturmian words, in: M. Lothaire, Algebraic combinatorics on words, (Chapter 2), vol. 90 of Encyclopedia of Mathematics and its Applications*, Cambridge University Press, 2002.
5. M. Crochemore and D. Perrin: *Two-Way String Matching*, J. ACM 38(3): 651-675, 1991.
6. M. Crochemore and W. Rytter: *Jewels of stringology: text algorithms*, World Scientific, 2003.
7. T. Harju and D. Nowotka: *On the density of critical factorizations*, ITA 36(3): 315-327, 2002.
8. F. Mignosi, J. Shallit, and I. Venturini: *Sturmian Graphs and a Conjecture of Moser*, Lecture Notes in Computer Science 3340, 175-187, 2004.
9. W. Rytter: *The structure of subword graphs and suffix trees of Fibonacci words*, Theoretical Computer Science Volume 363, Issue 2, 211 - 223, 2006.
10. W. Rytter: *The number of runs in a string*, Information and Computation Volume 205, Issue 9, 1459-1469, 2007.

Edit Distance with Single-Symbol Combinations and Splits

Manolis Christodoulakis[1] and Gerhard Brey[2]

[1] School of Computing & Technology, University of East London
Docklands Campus, 4–6 University Way, London E16 2RD, UK
m.christodoulakis@uel.ac.uk

[2] Centre for Computing in the Humanities, King's College London
26–29 Drury Lane, London WC2B 5RL, UK
gerhard.brey@kcl.ac.uk

Abstract. In this article we introduce new variants of the *edit distance* string similarity measure, where apart from the traditional insertion, deletion and substitution operations, two new operations are supported. The first one is called a *combination* and it allows two or more symbols from one string, to be "matched" against one symbol of the other. The dual of a combination, is the operation of a *split*, where one symbol from the first string is broken down into a sequence of two or more other symbols, that can then be matched against an equal number of symbols from the second string. The notions of combining and splitting symbols can be defined in a variety of ways, depending on how the application in hand defines similarity. Here we introduce three different possible definitions, and we provide an algorithm that deals with one of them. Our algorithm requires $O(L)$ time for preprocessing, and $O(mnk)$ time for computing the edit distance, where L is the total length of all the valid combinations/splits, and k is an upper bound on the number of valid splits of any single symbol.

Keywords: edit distance, combination, split, OCR

1 Introduction

One of the fundamental problems in string algorithmics has been, for more than 40 years now, the pattern matching problem, where exact copies of a given string, the *pattern*, need to be identified within a normally much larger string, the *text*. However, the need to relax the "exactness" of the pattern matching process, very soon became obvious. An endless list of applications benefit from *approximate*, rather than exact, pattern matching algorithms, including text processors, spell checking applications, information retrieval and bioinformatics.

Numerous approaches have been used to incorporate "inexactness" in pattern-matching (see for example [4], [2, Ch.12] or [7, Ch.10]), one of the most commonly used being the *edit distance* metric (also known as *Levenshtein* distance, as a credit to Vladimir Levenshtein who first mentioned it [3]). The edit distance between two strings, is simply defined as the minimum number of *edit operations* (substitutions, insertions, deletions) that are required to transform one string to the other.

In this paper, we introduce a new edit operation, called a *combination*, and its dual, called a *split*. These operations, in contrast to the traditional ones, apply to *sequences* of input symbols, rather than single symbols. The *combination* operation is that of combining two or more symbols from one string, and considering this combination to be equal to a single symbol from a second string. Equivalently, the single symbol of the second string can be *split* into the combination of symbols from the

Manolis Christodoulakis, Gerhard Brey: Edit Distance with Single-Symbol Combinations and Splits, pp. 208–217.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

first. Defining which combinations match to what symbols, and vice versa, depends on the application. In all examples in this paper, we are referring to combinations that *look similar* to some symbol, but of course one can define any list of combinations that are meaningful for their application. For example, the sequence of symbols "rn" can be combined into the symbol "m", or "m" can be split to "rn", since the two look similar to each other.

The motivation for this new variant of the edit distance metric, comes from the approximate pattern matching problem on texts which originate from old documents that have been scanned and processed with Optical Character Recognition (OCR) Algorithms. In particular, we have been working on the Nineteenth-Century Serials Edition (NCSE) [5], which is a digital edition of six nineteenth-century newspaper and periodical titles. The corpus consists of about 100,000 pages that were micro-filmed, scanned in and processed using OCR software.

The quality of some of the text resulting from the OCR process varies from barely readable to illegible. This reflects the poor print quality of the original paper copies of the publications. An exact search for a pattern in the scanned and processed text would retrieve only a small number of matches, but ignore incorrectly spelled or distorted variations; on the other hand, an approximate search using general edit distance would yield too many false matches, since it cannot distinguish between "random" errors and errors that come from misinterpreted combinations of symbols which are common in OCR texts. For example, a general edit distance search for the name "Billington" in the corpus, would fail to distinguish between the approximate matches "Wellington" and "Billmgton", both of which have edit distance 2 from the pattern, but where in reality only the latter is a true match misinterpreted by the OCR software.

The paper is organised as follows. In Section 2 we describe the notation used throughout the paper, and formally define the notions of combinations and splits. In Section 3 we describe the preprocessing part of our algorithm and in Section 4 the main algorithm for computing the edit distance with combinations and splits. In Section 5 two variants of the problem we tackle here are presented. Finally, Section 6 contains our concluding remarks.

2 Preliminaries

Consider strings $x = x[1] \cdots x[n]$ and $y = y[1] \cdots y[m]$ over an alphabet Σ; the edit distance between x and y is defined as the minimum number of *edit operations* (insertions, deletions or substitutions) to transform x to y, or vice versa [6,3]. Implicitly, the simple edit distance assigns to each operation a unit cost, and computes the minimum cost of transforming x to y. A generalised version of the edit distance, is one that allows the different operations to have different costs; let d_{sub} be the cost for one substitution operation, and d_{indel} that of one insertion or deletion operation (one is a dual of the other, hence the identical cost). The generalised edit distance is defined then as the minimum cost of transforming x to y.

Notice how traditional variants of the edit distance always compare a single symbol from one string with either a single symbol from the other (e.g. $x[i]$ against $y[j]$) or a single symbol from one string with the empty string (e.g. $x[i]$ deletion or $y[j]$ insertion). In the variant of the edit distance that we introduce in this paper, we allow more than one symbol to be "matched" either against a single symbol (that is, many symbols are *combined* into one) or against a different combination of symbols (called

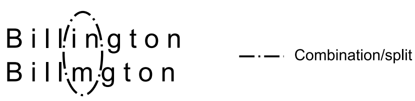


Figure 1. Example of a single-symbol combination

a *recombination*). For example, the symbol “m” is a combination of the symbols contained in the string “in” or “rri”, and “b” is a combination of “lo”. As seen in this example, there may very well exist, and they normally do, more than one combinations for the same symbol (symbol “m” in this example). Formally, we define combinations in the following way:

Definition 1. *Given a string* $x = x[1] \cdots x[n]$ *and a symbol* α*,* α *is called a* single-symbol combination*, or simply a* combination*, of* x *(equivalently,* x *is called a* split *of* α*), if and only if* x *is a valid match for* α*; we write* $\alpha \mapsto x$*.*

Obviously, any algorithm that makes use of combinations of symbols must be able to differentiate between meaningful (valid) and random combinations. In our case, meaningful ones are those combinations of symbols that *optically* resemble one or more other symbols. It is worth noting however that any kind of valid combinations may as well be used. For instance, one may consider combinations of symbols which *sound* similar to other (combinations of) symbols.

We assume that the list of valid combinations is given in the following way: for every symbol, α, for which valid combinations exist, a *combination list* $\mathcal{C}_\alpha$ is provided such that

$$\mathcal{C}_\alpha = \{x \in \Sigma^* | \alpha \mapsto x\}$$

We further introduce the following notation

$$k_\alpha = |\mathcal{C}_\alpha| \tag{1}$$

$$l_\alpha = \sum_{x \in \mathcal{C}_\alpha} |x| \tag{2}$$

$$L = \sum_{\alpha \in \Sigma} l_\alpha \tag{3}$$

Simply, k_α denotes the number of keywords (valid combinations) in $\mathcal{C}_\alpha$, l_α is sum of the lengths of all the strings in $\mathcal{C}_\alpha$, and L is the overall sum of lengths of all the strings over all combination lists.

With these definitions in place, the problem we are going to tackle in this paper is defined as follows:

Definition 2 (Edit Distance with Single-Symbol Combinations (EDSSC)). *Given strings* $x = x[1] \cdots x[n]$ *and* $y = y[1] \cdots y[m]$*, values* d_{sub}*,* d_{indel} *and* d_{comb}*, and lists* $\mathcal{C}_\alpha$ *for* $\alpha \in \Sigma$*, the* edit distance with single-symbol combinations *problem is that of finding the minimum cost of transforming* x *to* y *(equivalently,* y *to* x*) allowing substitutions, insertions or deletions, and single-symbol combinations or splits.*

An example of this problem is illustrated in Figure 1. The substring “in” of the first string is combined and matched against the symbol “m” of the second string. The EDSSC is therefore d_{comb}, since one combination operation is required to transform one string to the other. Normally, the cost for a combination/split will be smaller

Algorithm 6 Function NEXT for moving inside a tree T_α

function NEXT(v, a)
  while $g(v, a) =$"fail" **do**
    $v \leftarrow f(v)$
  return $g(v, a)$

(possibly zero) than that of a substitution or insertion/deletion; thus, the cost d_{comb} is going to be much smaller —reflecting the fact that the two strings look similar to each other— than the $d_{indel} + d_{sub}$ of the simple edit distance for the same pair of strings.

3 Preprocessing

In this stage we preprocess the lists of valid combinations and splits, which are given as input; the purpose of preprocessing is to allow the main algorithm to run faster.

Recall that for every symbol α we are given a list, $\mathcal{C}_\alpha$, of combinations that match α. The preprocessing starts by building an Aho-Corasick automaton [1], T_α, from the strings contained in $\mathcal{C}_\alpha$, for every $\alpha \in \Sigma$. We will then slightly modify these T_α's to better suit our edit distance algorithm. First, let us briefly describe the Aho-Corasick automaton.

An Aho-Corasick automaton, T, is constructed from a set of keywords, $\mathcal{C}$, essentially by generating a trie of all the strings contained in $\mathcal{C}$, and computing functions g, f and *out* as described below. Let v be one node in T, a be a symbol, and $\mathcal{L}_v$ be the string spelled out on the path from the root to node v. Then:

- The "forward" (or goto) function $g(v, a)$ returns a node u in T if there exists an outgoing edge from v to u labeled with a, or returns "fail" if no such node u exists; exceptionally for the root node, $g(root, a) = root$ if in the trie there is no outgoing edge from *root* labeled with a.
- The "failure" function $f(v)$ returns a node u whose label, $\mathcal{L}_u$, corresponds to the longest proper suffix of $\mathcal{L}_v$ that occurs in T; if no such suffix exists, $f(v) = root$.
- The "output" function $out(v)$ returns all the keywords of C that are suffixes of $\mathcal{L}_v$.

The algorithm that searches a text, s, for any of the strings in $\mathcal{C}$, operates by repeatedly calling the function $g(v, a)$, where v is the current node at any stage and a is the symbol of s currently being processed; initially, v is the root of T and $a = s[1]$. If at any stage $g(v, a)$ returns "fail", then the failure function is used and the search continues from the failure link node, $g(f(v), a)$. To make the notation easier, we define function NEXT, shown in Algorithm 6, which for a node v and input symbol a, follows as many failure links as necessary (possibly zero) and then calls g once.

For the purposes of our algorithm we need to slightly modify function *out*. Instead of storing the actual keywords that match at a specific node, we only need to know the *lengths* of all these matching keywords. This information will be later used to compute the cost of performing combination operations. The modification is easily implemented during the construction of the automaton, without modifying the running time of the algorithm.

Figure 2 demonstrates an example of preprocessing the combination list of the symbol m, $\mathcal{C}_m = \{$iii,iin,in,ni,nn,rn,rri$\}$. Nodes are represented by circles, solid edges

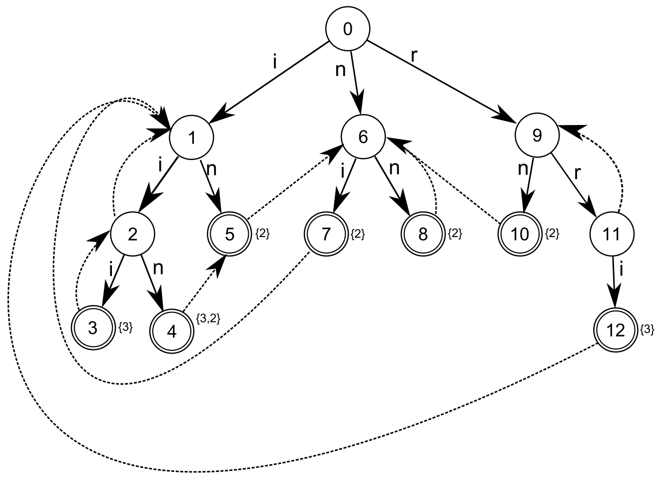


Figure 2. Preprocessing $\mathcal{C}_m = \{$iii,iin,in,ni,nn,rn,rri$\}$

represent the forward links (function g), and dashed edges represent the failure links (function f); for those nodes for which a failure link is not shown in the diagram, it is implied to be the root. Next to those nodes for which one or more keywords match, we show the lengths of the matching keywords. See for example, node 4 at which both keywords "iin" and "in" match, we store the lengths of these two keywords, 3 and 2 respectively.

4 EDSSC Algorithm

Let $x = x[1] \cdots x[n]$ and $y = y[1] \cdots y[m]$ be the two strings whose distance we want to compute; assume that the combination lists have already been provided and pre-processed, yielding Aho-Corasick automata T_α, for all $\alpha \in \Sigma$ for which valid combinations exist.

The algorithm works by processing gradually increasing prefixes of x and y. While processing prefixes $x[1..i]$ and $y[1..j]$, where $1 \leq i \leq n$ and $1 \leq j \leq m$, the permitted edit operations are as follows:

- substitution of a symbol $x[i]$ with the symbol $y[j]$, with cost d_{sub}
- insertion of the symbol $y[j]$ into x, with cost d_{indel}
- deletion of the symbol $x[i]$ from x, with cost d_{indel}
- combination of the symbols $x[\ell..i]$ (for some $1 \leq \ell < i$) to match $y[j]$, with cost d_{comb}
- split of the symbol $x[i]$ to match $y[h..j]$ (for some $1 \leq h < j$), with cost d_{comb}

The algorithm maintains a dynamic programming table $D(0..n, 0..m)$, where $D(i, j)$ represents the minimum cost of transforming $x[1..i]$ to $y[1..j]$, allowing the operations mentioned above. In contrast to the traditional edit distance dynamic programming algorithm, the whole D table must be maintained throughout the operation of the algorithm, rather than the last row (column) which were the only needed in the row-wise (column-wise) operation of traditional dynamic programming.

The base conditions for D are $D(i, 0) = i \cdot d_{indel}$ and $D(0, j) = j \cdot d_{indel}$, similarly to the simple edit distance algorithm. The correctness of these base conditions comes from the fact that the empty string ϵ is not a valid split for any $\alpha \in \Sigma$, and thus the only way to transform $x[1..i]$ to ϵ is by deleting $x[1..i]$ from x and similarly the only way of transforming ϵ to $y[1..j]$ is by inserting $y[1..j]$ into x.

The recurrence relation for $D(i,j)$, for $i,j > 0$, is

$$D(i,j) = \mathbf{min} \begin{cases} D(i-1,j-1) + \textsc{sub}(x[i],y[j]) & \{\text{substitution}\} \\ D(i,j-1) + d_{indel} & \{\text{insertion}\} \\ D(i-1,j) + d_{indel} & \{\text{deletion}\} \\ \textsc{comb}(x[1..i],y[j]) & \{\text{combination}\} \\ \textsc{split}(x[i],y[1..j]) & \{\text{split}\} \end{cases}$$

where the functions SUB, COMB and SPLIT are defined as follows:

- SUB: this function simply compares $x[i]$ with $y[j]$ and returns 0 if they are equal and d_{sub} otherwise;
- COMB: finds the suffix of $x[1..i]$ that can be combined into symbol $y[j]$ with the minimum cost and returns this cost, or returns $+\infty$ if there is no such suffix;
- SPLIT: finds the suffix of $y[1..j]$ that can be combined into symbol $x[i]$ with the minimum cost and returns this cost, or returns $+\infty$ if there is no such suffix.

Next, we will demonstrate how functions COMB and SPLIT operate. We will assume a row-wise scan of the dynamic programming table D (a column-wise scanning is equivalent).

4.1 Combinations

Recall that, while computing $D(i,j)$, a combination operation translates to finding one suffix of $x[1..i]$ that can be combined to $y[j]$; here we are interested in the suffix with the minimum cost. This suffix can be found by searching the combination list of $y[j]$, $\mathcal{C}_{y[j]}$, and more specifically by making use of the Aho-Corasick automaton, $T_{y[j]}$, which was created during preprocessing. $T_{y[j]}$ is traversed, starting from the root, and descending down the tree, using failure links where necessary (functions NEXT, see Section 3), until the whole prefix $x[1..i]$ has been spelled out. Then the number of valid combinations is the number of elements in the set returned by function *out*, in the last node we visited in $T_{y[j]}$. If this set is empty, then there is no suffix of $x[1..i]$ that is a valid split of $y[j]$ and thus the algorithm returns $+\infty$ cost. On the other hand, if it has one or more elements, the algorithm must check the cost of each of those combinations and return the minimum.

Let v be the last node visited in T when parsing $x[1..i]$, and $out(v) = \{v_1, \ldots, v_r\}$ be the output of node v, where $v_1, \ldots, v_r$ are integers representing the lengths of the keywords that match suffixes of $x[1..i]$. Thus, $x[i-v_1+1..i], \ldots, x[i-v_r+1..i]$ are all the valid splits of $y[j]$. What remains to be done is compute the cost of each of those.

If $x[i-v_1+1..i]$ is combined and matched to $y[j]$, then the prefix $x[1..i-v_1]$ of x and the prefix $y[1..j-1]$ of y must be aligned with each other. That is, the cost of this combination operation is the minimum cost of transforming $x[1..i-v_1]$ to $y[1..j-1]$, plus the cost of combining $x[i-v_1+1..i]$ into $y[j]$, i.e. $D(i-v_1, j-1)+d_{comb}$. We repeat the same process for all the values in $out(v)$ and return the minimum, and since the cost for every combination is constant, d_{comb}, the minimum cost for a combination at cell $D(i,j)$ is

$$\mathbf{min}\{D(i-v_1, j-1), \ldots, D(i-v_r, j-1)\} + d_{comb}$$

The algorithm for COMB, as it has been described so far, is not efficient: for every prefix $x[1..i]$ we spend $O(i)$ time to traverse $T_{y[j]}$ from the root until all i symbols of $x[1..i]$ have been processed. Let v be the last node visited in this traverse. We notice that, during the computation of $D(i+1, j)$, when the prefix $x[1..i+1]$ must be spelled out on $T_{y[j]}$, one can avoid parsing this whole prefix simply by remembering that $x[1..i]$ ended in node v. Then, the traversal for $x[1..i+1]$ requires only one call to function NEXT, $\textsc{next}(v, x[i+1])$.

To take advantage of this observation, and given the assumption that D is processed in a row-wise manner, the algorithm must store the node where the suffix $x[1..i]$ ends in each $T_{y[j]}$ (for all $1 \leq j \leq m$). To do that we create a vector $t[1..m]$, which while working on the i-th row of D will contain values $t[j] = \textsc{next}(root(T_{y[j]}), x[1..i])$, $1 \leq j \leq m$. Then, during processing the $(i+1)$-th row of D, $t[j]$ is updated with the value $t[j] = \textsc{next}(t[j], x[i+1])$. The initial values of vector t —that correspond to the empty prefix of x— are of course $t[j] = root(T_{y[j]})$.

4.2 Splits

A split operation is the dual of a combination; a split on x is a combination on y and vice versa. During the computation of $D(i, j)$, a split means finding the suffix of $y[1..j]$ that is a valid split of $x[i]$ and is of minimal cost. Similarly to the combination operation, this can be found by spelling out $y[1..j]$ on the Aho-Corasick automaton of $x[i]$, $T_{x[i]}$.

Storing and recalling the last node visited by the prefix $y[1..j]$ on $T_{x[i]}$, works here too, only somewhat in a simpler way. The row-wise processing of D ensures that all prefixes of y are processed one after the other on $T_{x[i]}$, before moving to $T_{x[i+1]}$. Thus in this case, only a single variable u is required such that, during the computation of $D(i, j)$, $u = \textsc{next}(root(T_{x[i]}), y[1..j])$, and then while computing $D(i, j+1)$, u is updated as $u = \textsc{next}(u, y[j+1])$. The initial value of u, which corresponds to the empty prefix of y, is $u = root(T_{x[i]})$.

4.3 Analysis of the Algorithm

The complete algorithm for the "edit distance with single-symbol combinations" problem is shown in Algorithm 7. Notice that the first argument of COMB (SPLIT) is only a node in an Aho-Corasick automaton, the function does not need to know the whole prefix of x (y) that has already been processed.

Theorem 3. *Algorithm 7 requires $O(L)$ time for preprocessing, and $O(mnk)$ time for computing the edit distance, where L is the total length of all the valid combinations (see Eq. 3), and k is an upper bound on the number of valid splits of any single symbol.*

Proof. The preprocessing consists of building an Aho-Corasick automaton, T_α, for each list of valid combinations, $\mathcal{C}_\alpha$. This process requires time linear in the length of the input [1], that is $O(l_\alpha)$ (see Eq. 2). The updating of function *out* to return the lengths of the matching keywords, rather than the keywords themselves, does not increase the running time since during construction we can, in constant time per entry in $out(v)$, replace every keyword $x \in out(v)$ with its length, $|x|$. Collectively, the preprocessing of all the combination lists requires $O(\sum_{\alpha \in \Sigma} l_\alpha) = O(L)$ time.

In the edit distance algorithm, initialization (lines 10–14 of Algorithm 7) takes $O(m+n)$ time for assigning values to the first column of D, of size $O(n)$, the first row

Algorithm 7 EDSSC algorithm

```
function EDSSC(x, y)
    n ← |x|, m ← |y|
    for all α ∈ Σ do            {Preprocess combination lists}
        T_α ← AHO-CORASICK(C_α)
        for all v ∈ T_α do
            out′ ← {}
            for all x ∈ out(v) do
                out′ ← out′ ∪ {|x|}
            out ← out′
    for i ← 1 to n do            {Initializations}
        D(i, 0) ← i · d_indel
    for j ← 1 to m do
        D(0, j) ← j · d_indel
        t[j] ← root(T_y[j])
    for i ← 1 to n do            {Main algorithm}
        u ← root(T_x[i])
        for j ← 1 to m do
            D(i, j) ← min(D(i − 1, j − 1)+SUB(x[i], y[j]),
                          D(i, j − 1) + d_indel, D(i − 1, j) + d_indel,
                          COMB(t[j], x[i]), SPLIT(u, y[j]))
    return D(n, m)

function SUB(α, β)
    if α = β then
        return 0
    else
        return d_sub

function COMB(v, α)
    v ← NEXT(v, α)            {Update v}
    min ← +∞
    for all r ∈ out(v) do
        if D(i − r, j − 1) + d_comb < min then
            min ← D(i − r, j − 1) + d_comb
    return min

function SUB(v, α)
    v ← NEXT(v, α)            {Update v}
    min ← +∞
    for all r ∈ out(v) do
        if D(i − 1, j − r) + d_comb < min then
            min ← D(i − 1, j − r) + d_comb
    return min
```

of D, of size $O(m)$, and the vector, t, of size $O(m)$, which is initialized with pointers to the roots of $T_{y[j]}$. The main body of the algorithm (lines 15–18) computes the values of $D(i,j)$ for $1 \leq i \leq n$ and $1 \leq j \leq m$, by performing one of the operations substitution, insertion, deletion, combination, or split. The first three clearly require constant time, to examine cells $D(i-1, j-1)$, $D(i, j-1)$ and $D(i-1, j)$ respectively, while combinations and splits need deeper analysis.

A combination on cell $D(i,j)$, first makes a call to NEXT (line 26), which in turn makes zero or more calls to function f and a single call to function g (Algorithm 6). Although the number of calls to f is not constant, from [1] we know that collectively for a string of size n, there are $O(n)$ calls to both f and g. At the end of our algorithm, we have processed m columns, each of which can be seen collectively as a single parsing of string x on one Aho-Corasick automaton, i.e. $O(mn)$ time; similarly each of the n rows can be seen collectively as a single parsing of string y on one Aho-Corasick automaton, i.e. $O(mn)$ time. Therefore, overall, the total time spent on NEXT is $O(mn)$.

After calling NEXT, the function COMB computes the cost of each combination suggested by $out(v)$, in time constant per entry or $|out(v)|$ in total, as can be seen in lines 28–30 of Algorithm 7. Unfortunately, in the worst case the number of elements in $out(v)$ can be as large as the number of keywords, k_α, in the Aho-Corasick automaton. Let k be an upper bound of k_α, that is $k_\alpha \leq k$ for all $\alpha \in \Sigma$. Then, for every call to COMB we spend $O(k)$ time to examine all the valid combinations. Thus, the total time for computing the whole table D is $O(mnk)$.

It is worth noting that, despite the worst-case running time being $O(mnk)$, in practice the algorithm is expected to run in time closer to $O(mn)$, since the number of matching strings, $out(v)$, for every node v, is rarely larger than two or three, and most often it is either zero or one (see for example Figure 2).

5 Variants of the EDSSC Problem

In this section we present two possible variants of the EDSSC problem, which further extend the notions of combining and splitting symbols. Both can be seen as useful generalisations of EDSSC, but unfortunately the algorithm we presented in this paper cannot (at least not trivially) extend to solve them, and we leave these as open problems for further investigation.

Similar to the way we defined the combination lists, $\mathcal{C}_\alpha$, for symbols $\alpha \in \Sigma$, we could also define *recombination* lists, $\mathcal{C}_x$, $x \in \Sigma^*$; that is, lists of combinations of symbols that validly represent a different combination of symbols. For example, $\mathcal{C}_{bl} = \{lol, ld\}$. In this way, when computing the edit distance between strings x and y, we allow whole substrings of x (rather than single symbols) to be matched against different substrings of y, and vice versa. Therefore, a more general version of the EDSSC problem is that of allowing the operation of symbol recombinations, in parallel to all operations allowed in the EDSSC problem.

Definition 4 (Edit Distance with Re-Combinations (EDRC)). *Given strings $x = x[1] \cdots x[n]$ and $y = y[1] \cdots y[m]$, values d_{sub}, d_{indel} and d_{comb}, and lists $\mathcal{C}_x$ for $x \in \Sigma^*$, the* edit distance with recombinations *problem is that of finding the minimum cost of transforming x to y (equivalently, y to x) allowing substitutions, insertions or deletions, single-symbol combinations or splits, and string recombination operations.*

An even more general variant of the edit distance with (re-)combinations problem, is that of allowing transitive (re-)combinations of symbols. We illustrate this problem with an example. Consider strings $x =$"m" and $y =$"rri", and combination lists $\mathcal{C}_m = \{rn, in\}$ and $\mathcal{C}_n = \{ri\}$; although the two strings look similar there is no explicit valid combination "rri" in $\mathcal{C}_m$. However, notice that the suffix "ri" of y is a valid combination in $\mathcal{C}_n$, and thus y could be matched to y'="rn"; now $x \mapsto y'$ since "rn" $\in \mathcal{C}_m$, and thus we can infer that $x \mapsto y$.

Definition 5 (Edit Distance with Transitive Combinations (EDTC)). *Given strings $x = x[1] \cdots x[n]$ and $y = y[1] \cdots y[m]$, values d_{sub}, d_{indel} and d_{comb}, and lists $\mathcal{C}_x$ for $x \in \Sigma^*$, the* edit distance with transitive combinations *problem is that of finding the minimum cost of transforming x to y (equivalently, y to x) allowing substitutions, insertions or deletions, single-symbol combinations or splits, and string recombination operations, where any of the (re-)combination operations can be transitive:*

$$x \mapsto y \text{ and } y \mapsto z \quad : \quad x \mapsto z \qquad \text{where } x, y, z \in \Sigma^*$$

6 Conclusions

In this paper we have introduced the problem of edit distance with single-symbol combinations and splits, where in addition to the traditional edit distance operations, consecutive symbols in one string may be combined and matched against one symbol from the other. Our algorithm runs in $O(mnk)$ time with a prior $O(L)$ time for preprocessing, where L is the sum of lengths of all the valid combinations and k is the maximum number of valid splits for any symbol.

We also defined two variants of the above problem which allow a) two or more consecutive symbols of one string to match a different sequence of two or more consecutive symbols in the other string (recombinations), and b) combinations of symbols to be constructed by combining known smaller valid combinations (transitive combinations). These two variants are equally, if not more, interesting problems, both from a practical/application point of view, as well as from an algorithmic point of view, and have been left as open problems. As it stands, the algorithm we presented here does not appear to extend towards solving either of these problems, and further research is required.

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: An aid to bibliographic search.* Communications of the ACM, 18(6) 1975, pp. 333–340.
2. M. Crochemore and W. Rytter: *Jewels of Stringology: Text Algorithms*, World Scientific, 2002.
3. V. I. Levenshtein: *Binary codes capable of correcting deletions, insertions and reversals.* Soviet Physics Doklady, 10 1966, pp. 707–710.
4. G. Navarro: *A guided tour to approximate string matching.* ACM Computing Surveys, 33(1) 2001, pp. 31–88.
5. *Nineteenth-Century Serials Edition (NCSE)*: http://www.ncse.ac.uk.
6. D. Sankoff and J. Kruskal, eds., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, 1983.
7. B. Smyth: *Computing Patterns in Strings*, Pearson Education, 2003.

A Concurrent Specification of an Incremental DFA Minimisation Algorithm

Tinus Strauss, Derrick G. Kourie, and Bruce W. Watson

FASTAR, University of Pretoria
Pretoria, South Africa
{tstrauss,dkourie,bwatson}@cs.up.ac.za
http://www.fastar.org

Abstract. In this paper a concurrent version of a published sequential incremental deterministic finite automaton minimisation algorithm is developed. Hoare's Communicating Sequential Processes (CSP) is used as the vehicle to specify the concurrent processes.
The specification that is proposed is in terms of the composition of a number of concurrent processes, each corresponding to a pair of nodes for which equivalence needs to be determined. Each of these processes are again composed of several other possibly concurrent processes.
To facilitate the specification, a new CSP concurrency operator is defined which, in contrast to the conventional CSP concurrency operator, does not require all processes to synchronise on common events. Instead, it is sufficient for any two (or more) processes to synchronise on such events.

Keywords: DFA minimisation, concurrency, CSP, automata

1 Introduction

As pointed out in [10], two contemporary trends drive the need for the development of concurrent implementations of automata algorithms. 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 [7], 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 [6,11,12].

The problem of minimising a finite state automaton has been studied quite extensively over the years and many algorithms have been proposed to address this problem. See [14, Chap. 7] for a comprehensive coverage of the area.

Previous parallel algorithms that have been proposed include [5,8,13]. These algorithms typically depend on a specific parallel computational models. In the present case, we abstract away from these considerations and model the algorithm as a process in the Communicating Sequential Processes (CSP) process algebra. Our purpose is to expose maximally the possibilities for concurrency inherent in the problem itself—at least to the extent that these possibilities may be latent in the sequential algorithm. As a consequence, we are not concerned with issues such as allocation of processes to processors, determination of which processes could be coalesced into a single process to limit context switching, etc. These are regarded as implementation issues for later consideration.

Tinus Strauss, Derrick G. Kourie, Bruce W. Watson: A Concurrent Specification of an Incremental DFA Minimisation Algorithm, pp. 218–226.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

The article is structured as follows. Section 2 gives the preliminaries of the problem domain under consideration, and the sequential algorithmic solution to the problem is given in section 3. A very brief account is given in section 4 of CSP. We also introduce a so-called optional-parallel operator that will be used. Section 5 then provides a concurrent specification to the problem, before a brief conclusion in the final section.

2 Preliminaries

Throughout this paper, we will consider a specific *DFA* $(Q, \Sigma, \delta, q_0, F)$ where Q is the finite set of states, Σ is the input alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, $q_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of *final* states. We further assume that all states in the automaton are reachable from q_0. The size of a *DFA*, $|(Q, \Sigma, \delta, q_0, F)|$, is defined as the number of states, $|Q|$.

To make some definitions simpler, we will use the shorthand Σ_q to refer to the set of all alphabet symbols which appear as out-transition labels from state q. Sometimes when it is the case that $\Sigma_p = \Sigma_q$ we will write Σ_{pq} instead of Σ_p or Σ_q to emphasise their equality.

We take $\delta^* : Q \times \Sigma^* \rightarrow Q$ to be the transitive closure of δ defined inductively (for state q) as $\delta(q, \varepsilon) = q$ and (for $a \in \Sigma_q$, $w \in \Sigma^*$) $\delta^*(q, aw) = \delta^*(\delta(q, a), w)$.

The right language of a state q, written $\overrightarrow{\mathcal{L}}(q)$, is the set of all words spelled out on paths from q to a final state. Formally, $\overrightarrow{\mathcal{L}}(q) = \{\, w \mid \delta^*(q, w) \in F \,\}$. With the inductive definition of δ^*, we can give an inductive definition of $\overrightarrow{\mathcal{L}}$:

$$\overrightarrow{\mathcal{L}}(q) = \left[\bigcup_{a \in \Sigma_q} \{a\} \overrightarrow{\mathcal{L}}(\delta(q, a)) \right] \cup \begin{cases} \{\varepsilon\} & \text{if } q \in F \\ \emptyset & \text{if } q \notin F \end{cases}$$

We define predicate *Equiv* to be ‘equivalence’ of states:

$$\mathit{Equiv}(p, q) \equiv \overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q)$$

With the inductive definition of $\overrightarrow{\mathcal{L}}$, we can rewrite *Equiv* as follows:

$$\begin{aligned} \mathit{Equiv}(p, q) \equiv {} & (p \in F \equiv q \in F) \wedge \Sigma_p = \Sigma_q \wedge \\ & \langle \forall\; a \in \Sigma_p \cap \Sigma_q : \mathit{Equiv}(\delta(p, a), \delta(q, a)) \rangle \end{aligned} \tag{1}$$

The primary definition of minimality of a *DFA* M is:

$$\langle \forall\; M' : M' \text{ is equivalent to } M : |M| \leq |M'| \rangle$$

where equivalence of *DFA*s means that they accept the same language. Using right languages, minimality can also be written as the following predicate:

$$\langle \forall\; p, q \in Q : p \neq q : \neg \mathit{Equiv}(p, q) \rangle$$

Equiv indicates whether two states are interchangeable. If they are, then one can be eliminated in favour of the other. (Of course, in-transitions to the eliminated state are redirected to the equivalent remaining one.) This reduction step is not addressed here.

3 The Sequential Algorithm

This section briefly details the sequential algorithm [15] for which we intend to provide a CSP model in the forthcoming sections. The algorithm is different from other minimisation algorithms in the sense that it is incremental, i.e. it may be halted at any time, yielding a partially minimised automaton.

From the problem of deciding the structural equivalence of two types, it is known that equivalence of two states can be computed recursively by turning the mutually recursive set of equivalences *Equiv* into a functional program. If the definition were to be used directly as a functional program, there is the possibility of non-termination in cyclic automata. In order for the functional program to work, it takes a third parameter along with the two states. An invocation $equiv(p, q, \emptyset)$ returns via the local variable *eq* the truth value of $Equiv(p, q)$. During the recursion, it assumes that two states are equivalent (by placing the pair of states in S, the third parameter) until shown otherwise.

Since it is known that the depth of recursion can be bounded by the larger of $|Q|-2$ and 0 (expressed as $(|Q|-2)\,\mathbf{max}\,0$) without affecting the result [14, §7.3.3], we add a parameter k to function *equiv* such that an invocation $equiv(p, q, \emptyset, (|Q|-2)\,\mathbf{max}\,0)$ returns $Equiv(p, q)$.

Purely for efficiency, the third parameter S is made a global variable. We assume that S is initialized to $\emptyset$. When $S = \emptyset$, an invocation $equiv(p, q, (|Q| - 2)\,\mathbf{max}\,0)$ returns $Equiv(p, q)$; after such an invocation $S = \emptyset$.

Algorithm 3.1 (Pointwise computation of $Equiv(p, q)$):

---

func $equiv(p, q, k) \rightarrow$
  if $k = 0 \rightarrow eq := (p \in F \equiv q \in F)$
  $[\!]$ $k \neq 0 \wedge \{p, q\} \in S \rightarrow eq := true$
  $[\!]$ $k \neq 0 \wedge \{p, q\} \notin S \rightarrow$
    $eq := (p \in F \equiv q \in F) \wedge (\Sigma_p = \Sigma_q)$;
    $S := S \cup \{\{p, q\}\}$;
    for $a : a \in \Sigma_p \cap \Sigma_q \rightarrow$
      $eq := eq \wedge equiv(\delta(p, a), \delta(q, a), k - 1)$
    rof;
    $S := S \setminus \{\{p, q\}\}$
  fi;
  return *eq*
cnuf

The function *equiv* can be used to compute the relation (i.e. set of pairs) *Equiv*. In variable G, we maintain a set consisting of the pairs of states known to be inequivalent (*distinguished*), while in H, we accumulate pairs of states belonging to the set *Equiv*. To initialize G and H, we note that final states are never equivalent to nonfinal ones, and that a state is always equivalent to itself. Since *Equiv* is an equivalence relation, we ensure that H is transitive at each step. Finally, we have global variable S used in Algorithm 3.1:

Algorithm 3.2 (Computing *Equiv*):

$S, G, H := \emptyset, ((Q \setminus F) \times F) \cup (F \times (Q \setminus F)), \{ (q,q) \mid q \in Q \};$
$\{$ invariant: $G \subseteq \neg \mathit{Equiv} \wedge H \subseteq \mathit{Equiv}$ $\}$
do $(G \cup H) \neq Q \times Q \rightarrow$
  let $p, q : (p,q) \in ((Q \times Q) \setminus (G \cup H));$
  if $\mathit{equiv}(p, q, (|Q| - 2) \mathbf{max}\, 0) \rightarrow$
    $H := H \cup \{(p,q),(q,p)\};$
    $H := H^+$
  $[\!]$ $\neg \mathit{equiv}(p, q, (|Q| - 2) \mathbf{max}\, 0) \rightarrow$
    $G := G \cup \{(p,q),(q,p)\}$
  fi
od; $\{$ $H = \mathit{Equiv}$ $\}$
merge states according to H
$\{$ $(Q, \Sigma, \delta, q_0, F)$ is minimal $\}$

The repetition in this algorithm can be interrupted and the partially computed H can be safely used to merge states.

4 CSP

Of the many process algebras that have been developed to concisely and accurately model concurrent systems, we have selected CSP [4,3,9] 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 first briefly introduce the conventional CSP operators that are used in the article. Thereafter we also introduce the so-called optional parallel operator—a new proposed CSP operator [2].

4.1 Introductory Remarks

CSP is concerned with specifying a system of concurrent sequential processes (hence the CSP acronym) in terms of sequences of atomic 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 operators used in this article.

Full details of the operator semantics and laws for their manipulation are available in [4,3,9]. 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 $(x?A \rightarrow P(x)) \mid\mid (y?B \rightarrow Q(y))$ engages in some nondeterministically chosen event $z \in A \cap B$ and then behaves as the process $P(z) \mid\mid Q(z)$. However, if $A \cap B = \emptyset$ then deadlock results. A special case of such parallel synchronization is the process $(b!e \rightarrow P) \underset{\alpha(b)}{\mid\mid} (b?x \rightarrow Q(x))$, where $\alpha(b)$ denotes the alphabet on channel b. This should be viewed as a process that engages in the event $b.e$ and thereafter behaves as the process $P \underset{\alpha(b)}{\mid\mid} Q(e)$. Note that parallel composition that involves more than two processes, requires that all processes always synchronise on common events. If they do not, then deadlock occurs. However, in the present context, it was considered desirable to introduce an alternative operator, called the optional parallel operator, that relaxes this requirement.

$a \rightarrow P$	event a then process P
$a \rightarrow P \mid b \rightarrow Q$	a then P choice b then Q
$x?A \rightarrow P(x)$	choice of x from set A then $P(x)$
$P \underset{X}{\parallel} Q$	P in parallel with Q
	Synchronize on events in set X
$b!e$	on channel b output event e
$b?x$	from channel b input to variable x
$P; Q$	process P followed by process Q
$P \,\vert\vert\vert\, Q$	process P interleave process Q
$P \triangle Q$	process P interrupted by process Q
$P \Box Q$	external choice between processes P and Q
$P \sqcap Q$	internal choice between processes P and Q

Table 1. Selected CSP Notation

4.2 Optional parallel operator

To position the new optional parallel operator, consider first the definition of the *generalised parallel* operator. The definition is expressed in terms of a so-called step law. The step law describes the initial actions in which the process may engage and then, for each possible initial action, it defines the behaviour of the process following that action. Suppose $R_1 = ?x : A_1 \rightarrow R_1'$ and $R_2 = ?x : A_2 \rightarrow R_2'$. Referring to Figure 1 the $\underset{X}{\parallel}$ -step law, provided by Roscoe [9, §2.4] can be expressed as the external choice between four different processes:

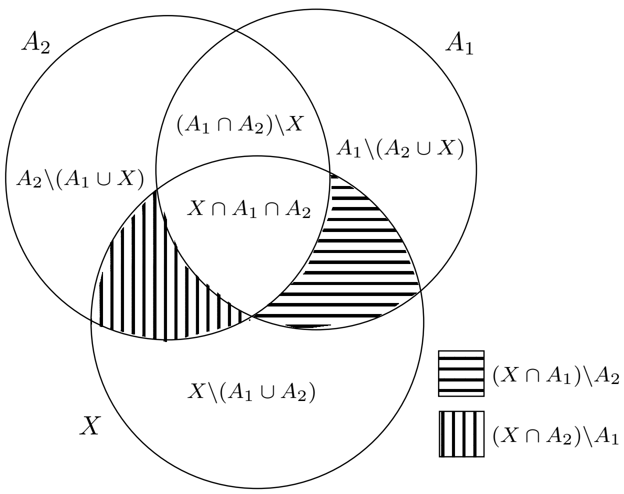


Figure 1. Venn diagram for sets of first actions.

$$
\begin{aligned}
R_1 \underset{X}{\parallel} R_2 = & \,(?x : X \cap A_1 \cap A_2 \rightarrow (R_1' \underset{X}{\parallel} R_2')) \\
& \Box\, (?x : (A_1 \cap A_2)\backslash X \rightarrow (R_1' \underset{X}{\parallel} R_2) \sqcap (R_1 \underset{X}{\parallel} R_2')) \\
& \Box\, (?x : A_1\backslash(X \cup A_2) \rightarrow (R_1' \underset{X}{\parallel} R_2)) \\
& \Box\, (?x : A_2\backslash(X \cup A_1) \rightarrow (R_1 \underset{X}{\parallel} R_2'))
\end{aligned}
$$

Note that the $\underset{X}{\parallel}$ -step law does *not* allow for any progress when the environment offers only some x in the shaded areas of Figure 1, i.e. when $x \in ((X \cap A_1)\backslash A_2) \cup ((X \cap A_2)\backslash A_1)$.

Suppose, however, that those restrictions were to be lifted. The resulting operator is then our optional (or partial) parallel operator, denoted by $\underset{X}{\Uparrow}$. The optional parallel operator's step law needs to indicate what is to happen when the environment offers $x \in ((X \cap A_1)\backslash A_2)$ or $x \in ((X \cap A_2)\backslash A_1)$. Evidently, in the first case an interaction between the environment and R_1 should occur, and in the second case, an interaction between the environment and R_2. The $\underset{X}{\Uparrow}$ -step law is therefore:

$$
\begin{aligned}
R_1 \underset{X}{\Uparrow} R_2 = {} & (?x : X \cap A_1 \cap A_2 \to (R_1' \underset{X}{\Uparrow} R_2')) \\
& \Box\, (?x : (A_1 \cap A_2)\backslash X \to (R_1' \underset{X}{\Uparrow} R_2) \sqcap (R_1 \underset{X}{\Uparrow} R_2')) \\
& \Box\, (?x : A_1\backslash(X \cup A_2) \to (R_1' \underset{X}{\Uparrow} R_2)) \\
& \Box\, (?x : A_2\backslash(X \cup A_1) \to (R_1 \underset{X}{\Uparrow} R_2')) \\
& \Box\, (?x : (X \cap A_1)\backslash A_2 \to (R_1' \underset{X}{\Uparrow} R_2)) \\
& \Box\, (?x : (X \cap A_2)\backslash A_1 \to (R_1 \underset{X}{\Uparrow} R_2'))
\end{aligned}
$$

This can be simplified to:

$$
\begin{aligned}
R_1 \underset{X}{\Uparrow} R_2 = {} & (?x : X \cap A_1 \cap A_2 \to (R_1' \underset{X}{\Uparrow} R_2')) \\
& \Box\, (?x : (A_1 \cap A_2)\backslash X \to (R_1' \underset{X}{\Uparrow} R_2) \sqcap (R_1 \underset{X}{\Uparrow} R_2')) \\
& \Box\, (?x : A_1\backslash A_2 \to (R_1' \underset{X}{\Uparrow} R_2)) \\
& \Box\, (?x : A_2\backslash A_1 \to (R_1 \underset{X}{\Uparrow} R_2'))
\end{aligned}
$$

Now, for $P = (x \to P')$, the process $P \underset{X}{\parallel} (R_1 \underset{X}{\Uparrow} R_2)$ will lead to the desired behaviour: the process evolves into $P' \underset{X}{\parallel} (R_1' \underset{X}{\Uparrow} R_2)$ or $P' \underset{X}{\parallel} (R_1 \underset{X}{\Uparrow} R_2')$, depending on whether $x \in (X \cap A_1)\backslash A_2$ or $x \in (X \cap A_2)\backslash A_1$ respectively, as depicted in the Venn diagram of Figure 1. Of course, if $x \in (X \cap A_1 \cap A_2)$ then the process evolves into $P' \underset{X}{\parallel} (R_1' \underset{X}{\Uparrow} R_2')$.

Of course, of itself the step law does not fully define the operator's semantics. This has been provided elsewhere [2], giving the denotational, trace, divergence and failures sematics, as well as the firing rules that specify the operational semantics.

It should be noted that the inspiration for this operator derives specifically from our earlier attempts to specify the present problem. The existing CSP operators were found to be deficient for our purposes. Once the semantics of the operator had been worked out, it became apparent that it can be usefully employed to specify a range of announcer/listener or reader/write-type problems. It will be seen that the operator neatly expresses the interaction between the fine-grained abstract processes that we have defined to solve the DFA minimisation problem.

5 Concurrent Specification

The principle concern in translating Algorithm 3.2 into a concurrent specification, is to translate its outer do-loop into a set of equivalent concurrent processes. To simplify the discussion, assume that P is the set of state pairs for which that loop would execute, i.e. $P = (Q \times Q)\backslash(G \cup H)$, where G and H are as initialised in Algorithm 3.2.

The overall system of interacting processes, called *Sys*, is conceived of as two processes that run in parallel with each other, and communicate via an alphabet, α. The two processes are called *Global* and *PairEquiv* respectively:

$$Sys = Global \underset{\alpha}{\|} PairEquiv \tag{2}$$

Global is a process that, for each state pair to be investigated, (p, q), receives information about the equivalence status of these states on a $from_{pq}$ channel (whose alphabet is therefore $\{from_{pq}.true,\ from_{pq}.false\}$) and announces the equivalence status of state-pairs on a to_{pq} channel (whose alphabet is therefore $\{to_{pq}.true,\ to_{pq}.false\}$). It is defined as follows:

$$\begin{aligned}
Global &= |||_{(p,q)\in P}\ In_{pq} \\
In_{pq} &= from_{pq}?e \rightarrow Announce_{pq}(e) \\
Announce_{pq}(e) &= (to_{pq}!e \rightarrow Announce_{pq}(e) \\
&\quad\ \mid from_{pq}?e \rightarrow Announce_{pq}(e))
\end{aligned} \tag{3}$$

The *Global* process is thus the interleaving of In_{pq} processes—one for each (p, q) pair in the system. Each In_{pq} process receives the equivalence status of its associated (p, q) pair on the $from_{pq}$ channel and then repeatedly announces this status on the to_{pq} channel. Each In_{pq} engages in events from the alphabet: $\alpha(pq) = \{from_{pq}.true,\ from_{pq}.false,\ to_{pq}.true,\ to_{pq}.false\}$. The *Global* process communicates with the *PairEquiv* process via the alphabet given by

$$\alpha = \bigcup_{(p,q)\in P} \alpha(pq)$$

The *PairEquiv* process not only passes on the equivalence status of each state pair to *Global*; it also acts as the “audience” to whom *Global* announces the equivalence status of pairs. It is the optional parallel composition, synchronising on events in α, of a set of processes called $Equiv_{pq}$, there being one such process for each (p, q) pair. *PairEquiv* is thus defined as:

$$PairEquiv = \underset{\alpha}{\Uparrow}{}_{(p,q)\in P}\ \ Equiv_{pq}(\emptyset, (|Q| - 2)\,\mathbf{max}\,0)$$

Note that, in general, each $Equiv_{pq}$ process has a parameter indicating the set S of pairs inspected by it to date, as well as the “recursion level”, k, apparent in the sequential algorithm. In the initial $Equiv_{pq}$ processes as encountered in *PairEquiv*, $S = \emptyset$ and $k = ((|Q| - 2)\,\mathbf{max}\,0))$.

Also note that the fact that these subprocesses of *PairEquiv*, namely $Equiv_{pq}$, mutually interact under optional parallelism with one another through α, while *PairEquiv* interacts with *Global* under generalised parallelism, also through α, means that whenever one *or more* of these subprocesses are ready to interact with *Global* on some arbitrary *to* or *from* channel, that interaction will take place as soon as the corresponding *Global* subprocess is ready to engage in it.

The generalised definition of $Equiv_{pq}$ is given by:

$$
\begin{aligned}
&Equiv_{pq}(S,k) = \\
&\qquad\qquad \text{if } (p = q) \text{ then } from_{pq}!true \to SKIP \\
&\qquad\qquad \text{else if } (k = 0) \text{ then } from_{pq}!(p \in F \equiv q \in F) \to SKIP \\
&\qquad\qquad\quad \text{else } (EqSet := \emptyset \\
&\qquad\qquad\qquad ;\ FanOut_{pq}(S,k) \\
&\qquad\qquad\qquad ;\ (eq := \wedge_{e \in EqSet}\ e) \\
&\qquad\qquad\qquad ;\ (from_{pq}!eq \to SKIP))
\end{aligned}
$$

The mapping from the above process to its sequential counterpart is direct, except that the test of circularity in paths visited to date is shifted to just before the recursive invocation of *Equiv*, as shown below, in the expansion of the *FanOut* process:

$$
\begin{aligned}
&FanOut_{pq}(S,k) = |||_{a \in \Sigma_{pq}} \\
&\qquad\qquad (\text{if } (\{\delta(p,a), \delta(q,a)\} \notin S) \text{ then} \\
&\qquad\qquad\quad (\ Equiv_{\delta(p,a),\delta(q,a)}(S \cup \{(p,q)\}, k-1) \triangle && (4) \\
&\qquad\qquad\quad (to_{\delta(p,a),\delta(q,a)}?eq_a \to (EqSet := EqSet \cup \{eq_a\})\ ) \\
&\qquad\qquad \text{else } (EqSet := EqSet \cup \{true\})\ ) && (5)
\end{aligned}
$$

The interrupt operator $\triangle$ in (4) requires justification. In the first place, it has been used there to ensure that the equivalence status of a pair is not needlessly sought by virtue of recursive calls to *Equiv* subprocesses, when that status had already been established and announced on the *from* channel to *Global* by some prior instance of an *Equiv* subprocess[1].

In the second place, the interrupt operartor's use here is justified, even though the CSP definition of the operator requires that if it is used in say $(P \triangle (a \to Q))$, then a may not be in the alphabet of P. This is in order to ensure that non-determinism cannot arise, such as in a situation where, say, $P = a \to R$. In such an case, the determination of the evolved process description after the occurrence of a would have to be non-deterministically chosen between Q and P. In the case of (4) above, such a non-deterministic choice will never be offered.

To realise that this is indeed the case, note the general form of the line, namely:

$$Equiv_{pq}(S,k) \triangle (to_{pq}?eq \to \cdots)$$

Now the only point at which an event on the channel to_{pq} can occur in $Equiv_{pq}(S,k)$ is when the chain of subprocesses spawned by $Equiv_{pq}(S,k)$ has cycled back to a new instance of itself. However, this is explicitly prevented by the condition of the if-statement preceding the process $Equiv_{pq}(S,k) \triangle (to_{pq}?eq \to \cdots)$. In such an case the process defined in (5) is executed. Thus, in the present context, the relaxation of the rule governing the use of the interrupt operator is justified—non-deterministic confusion cannot arise.

Note also that when the if-statement's condition is false—i.e. when a cycle has been detected on the fan-out branch from (p,q) for symbol a, then (as in the sequential

[1] Note that the semantics of the interrupt operator is such that when its first event occurs, all further activity of the initial process ceases, and the overall process behaves henceforth as the interrupting process. Furthermore, if the main process runs to completion, then the interrupting process plays no further role.

algorithm) this branch plays no role in the determination of (p, q)'s equivalence status. This is expressed by adding *true* into the *EqSet* set. It could equally well have been expressed by executing the *SKIP* process instead.

6 Conclusions

Just as in the case of the sequential algorithm, the foregoing specification has many optimisation possibilities. For example, the transitivity of the equivalence relation could be used to bring some of the *Equiv* processes more rapidly to an end. Also, as already mentioned, symmetry allows us to remove $Equiv_{qp}$ if $Equiv_{pq}$ is to be run.

This is the second well-known FA algorithm for which we have provided a concurrent CSP specification, the first one having been in [10]. The next phase of this ongoing project will be to experiment with implementations of these specifications. This will require reflection on ways in which the fine-grained processes that have been defined here can be coalesced with one another, and/or allocated to limited numbers of processors.

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. S. Gruner, D. G. Kourie, M. Roggenbach, T. Strauss, and B. W. Watson: *A new CSP operator for optional parallelism*, 2008, Submitted.
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. J. F. Jájá and K. W. Ryu: *An efficient parallel algorithm for the single function coarsest partition problem*, in SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures, New York, NY, USA, 1993, ACM Press, pp. 230–239.
6. R. McDougall: *Extreme software scaling.* ACM Queue, 3(7) September 2005, pp. 36–46.
7. K. Olukoton and L. Hammond: *The future of microprocessors.* ACM Queue, 3(7) September 2005, pp. 26–29.
8. B. Ravikumar and X. Xiong: *A parallel algorithm for minimization of finite automata*, in IPPS: 10^{th} International Parallel Processing Symposium, IEEE Computer Society Press, 1996.
9. A. W. Roscoe: *The theory and practice of concurrency*, Prentice Hall, 1997.
10. T. Strauss, D. G. Kourie, and B. W. Watson: *A concurrent specification of Brzozowski's DFA construction algorithm*, in Proceedings of Prague Stringology Conference '06, 2006, pp. 90–99.
11. H. Sutter: *A fundamental turn toward concurrency in software.* Dr. Dobb's Journal, 30(3) March 2005, pp. 16–20,22.
12. H. Sutter and J. Larus: *Software and the concurrency revolution.* ACM Queue, 3(7) September 2005, pp. 54–62.
13. A. Tewari, U. Srivastava, and P. Gupta: *A parallel DFA minimization algorithm*, in Proceedings of HiPC2002, Lecture Notes in Computer Science 2552, Springer, 2002, pp. 34–40.
14. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Eindhoven University of Technology, September 1995.
15. B. W. Watson and J. Daciuk: *An efficient incremental DFA minimization algorithm.* Journal of Natural Language Engineering, 9(1) March 2003, pp. 49–64.

On Regular Expression Hashing to Reduce FA Size

Wikus Coetser, Derrick G. Kourie, and Bruce W. Watson

Fastar Research Group, Department of Computer Science, University of Pretoria
spring.haas.meester@gmail.com, dkourie@cs.up.ac.za, bruce@bruce-watson.com

Abstract. In [7], a new version of Brzozowski's algorithm was put forward which relies on regular expression hashing to possibly decrease the number of states in the generated finite state automata. This method utilizes a hash function to decide which states are merged, but does not, in general, construct *-equivalence classes on automaton states, as is done in minimization algorithms. The consequences of this approach depends on the hash function used, and include the construction of a super-automaton and potential non-determinism. A revised version of the hashing algorithm in [7] is presented that constructs a deterministic automaton. A method for rewriting the hash function input is presented that allows the construction of a hash function that is an injection, mapping a unique integer to each regular language. A method for measuring the difference between the exact- and super-automaton is presented.

Keywords: finite state automaton, DFA, NFA, state merging, equivalence classes, regular languages, super-automaton, approximate automaton, hash function, minimization, exact automaton, sub-automaton

1 Introduction

Brzozowski has a well-known algorithm for deriving a finite automaton from a regular expression. In [7], a modified version of Brzozowski's Algorithm was presented for constructing an *approximate automaton*, hereafter referred to as a *super–automaton*. By merging the states in the event of hash function clashes, the resulting super-automaton may have fewer states than the finite state automaton[1] that would have been generated by the original algorithm.

The results of this merging process are explored in this article: in section 2, the original and modified versions of Brzozowski's Algorithm are presented. In section 3, a proof is given that the approach in section 2 always produces a super–automaton. In section 4, it is shown why the approach in section 2 may lead to non–determinism, and a new algorithm is put forward for constructing a deterministic automaton. In section 5.1 a method is put forward for judging the relative quality of super- and exact automata. In section 5.3, a method is given for modifying the input of the hash function in order to allow the entire language of a state to be taken into account when hashing. In section 5.4 the effect of the modulo function used in most hash functions is considered.

2 Brzozowski's Algorithm with state merging

In order to understand how state merging is implemented with a hash function, it is necessary to first look at Brozozwski's original algorithm, given in Algorithm 1, and taken from [7].

[1] Finite state automaton is abbreviated FA, a non-deterministic FA is abbreviated NFA and a deterministic FA is abbreviated DFA.

Wikus Coetser, Derrick G. Kourie, Bruce W. Watson: On Regular Expression Hashing to Reduce FA Size, pp. 227–241.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

Brzozowski's Algorithm takes a regular expression, and constructs a finite state automaton from that expression, using left derivatives[1], first symbol sets and a test for whether a regular language given by a regular expression contains the empty string:[2]

– The left derivative of a regular expression RE with respect to an input symbol s is written $s^{-1}RE$, and represents all the strings of the regular language defined by RE, with their respective first symbols removed.
– The first symbol set of a regular expression RE, written as $first(RE)$, is the set containing the first symbol of each string represented by the regular expression RE.
– Given that ε represents the empty string and that $L(RE)$ represents the set of strings of the language described by the regular expression RE, the test $\varepsilon \in L(RE)$ is written $nullable(RE)$.

Note that in Algorithms 1, 2 and 3, an FA is represented by a 4-tuple $\langle Q, \delta, E, F\rangle$, where Q is the set of states of the automaton, δ is the state transition function, E is the initial state and F is the set of final states. The alphabet Σ is not referenced in any of the respective algorithms and should therefore be regarded as implicit in the FA's representation.

The method used by Brzozowski's Algorithm is illustrated in Figure 1: regular expressions are associated with states in the algorithm.

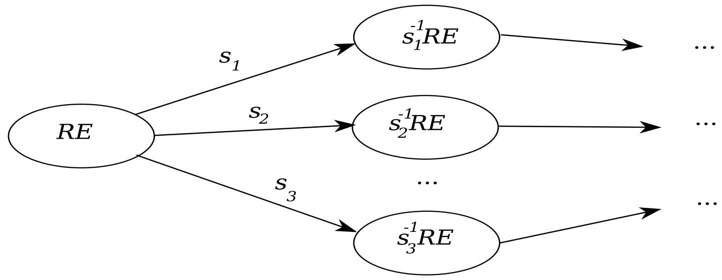


Figure 1. Brzozowski's Algorithm without state merging

Given the input regular expression RE, the first symbol set of RE is calculated. In Figure 1 the first symbol set $\{s_1, s_2, s_3\}$ corresponds to the out-transition symbols of the state marked as RE. For each first symbol s, the left derivative $s^{-1}RE$ is calculated. This left derivative represents the next state for s. The remap function in Algorithm 1 (Brzozowski's original algorithm) maps each regular expression to the next unassigned integer. If a regular expression re-appears (taking idempotence, associativity and commutativity of regular expression operators into account) as a result of computing the derivatives, a cycle forms in the automaton, representing a plus or star closure. When a regular expression represents a regular language that contains an empty string, a final state has been reached.

In the remainder of this text, $state(RE)$ is used to designate a state associated in some unspecified (i.e. abstract) way with the regular expression RE. If we wish to emphasise that in some concrete implementation, the association of the state with the

[2] The notation for first symbols, left derivatives and nullable regular expressions is taken from [3].

Algorithm 1 (Brzozowski's Algorithm with Remapping)

---

func $Brz(RE_{init})$
  $next, \delta, F, remap := 0, \emptyset, \emptyset, \emptyset;$
  $remap[RE_{init}], next := next, next + 1;$
  $done, todo := \emptyset, \{RE_{init}\};$
  do $todo \neq \emptyset \rightarrow$
    let RE_j be some regular expression such that $RE_j \in todo$;
    $done, todo := done \cup \{RE_j\}, todo \backslash \{RE_j\};$
    { Only expand out-transitions for symbols in the first symbol set ofRE_j }
    for $s : first(RE_j) \rightarrow$
      { Use the left derivatives to calculate the next state }
      $destination := s^{-1}RE_j$
      if $destination \notin done \cup todo \rightarrow$
        { Update the todo set in order to expand the automaton for *destination* }
        $todo := todo \cup \{destination\};$
        $remap[destination], next := next, next + 1$
      $[\!]$ $destination \in done \cup todo \rightarrow$ **skip**
      fi
      ; $\delta(remap[RE_j], s) := remap[destination]$
    rof
    ;
    if $nullable(RE_j) \rightarrow$
      { The final states all have a right language containing the empty string }
      $F := F \cup \{remap[RE_j]\}$
    $[\!]$ $\neg nullable(RE_j) \rightarrow$ **skip**
    fi
  od;
  return $\langle\{0, \ldots, next - 1\}, \delta, 0, F\rangle$
cnuf

---

regular expression is via a function, for example *hash*, then the notation $hash(RE)$ is used.

Consider two regular expressions RE_1 and RE_2. Jointly, these two regular expressions might form $state(RE_1)$ and $state(RE_2)$ in some FA, denoted by F. Suppose $L(RE_1) = \{ab\}$, $L(RE_2) = \{cd\}$ and $L(F) = \{ab, cd\}$ respectively. If F had been built by Algorithm 1, then states $remap(RE_1)$ and $remap(RE_2)$ respectively would have been constructed, and the languages associated with these states, as well as with F would be preserved.

In the Algorithm 2, hashing is used to assign an integer to each state, instead of the remap function. If the two regular expressions RE_1 and RE_2 hash to the same integer, then the collision is not resolved. Instead, the states are merged. This is illustrated in Figure 2. Note that the language of the automaton with the states merged is $\{ab, ad, cd, cb\}$, which is different from $L(F)$, given above as $\{ab, cd\}$.

This merging behaviour has two consequences: the construction of a super-automaton, and the automaton becoming potentially non-deterministic. These consequences are discussed in the next two sections.

3 Super-automata and exact automata

Previously in [7], approximate automata were described informally as being the output of Algorithm 2. Here the preferred nomenclature of *super-automata* will be used

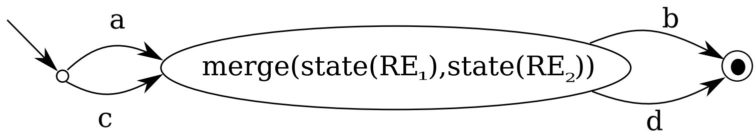


Figure 2. Brzozowski's Algorithm with state merging: $state(RE_1)$ and $state(RE_2)$ have been merged

Algorithm 2 (Brzozowski's Algorithm with Hashing — NFA version)

```
func Brz_hash_NFA(RE_init)
    Q, δ, F := ∅, ∅, ∅;
    done, todo := ∅, {RE_init};
    do todo ≠ ∅ →
        let  RE_j be some regular expression such that  RE_j ∈ todo;
        done, todo, h := done ∪ RE_j, todo\RE_j, hash(RE_j);
        Q := Q ∪ {h};
        { Only expand out-transitions for symbols in the first symbol set ofRE_j }
        for s : first(RE_j) →
            { Use the left derivatives to calculate the next state }
            destination := s^{-1}RE_j;
            if destination ∉ done ∪ todo →
                { Update the todo set in order to expand the automaton for destination }
                todo := todo ∪ destination
            [] destination ∈ done ∪ todo → skip
            fi
            ; δ(h, s) := δ(h, s) ∪ {hash(destination)};
        rof
        ;
        if nullable(RE_j) →
            { The final states all have a right language containing the empty string }
            F := F ∪ {h}
        [] ¬nullable(RE_j) → skip
        fi
    od;
    return ⟨Q, δ, hash(RE_init), F⟩
cnuf
```

instead of approximate automata. The notion of a super-automaton of a regular language is formally defined, and it is then formally shown that the output of Algorithm 2 is a super-automaton of the regular language associated with its input regular expression.

Let RL denote an *intended* regular language, for example the regular language described by the regular expression RE_{init} which is to form the input for Brzozowski's Algorithm. The definition of an exact automaton is:

Definition 1. *If RL is a regular language and FA is an automaton for which $L(FA) = RL$, then FA is an exact automaton of RL.*

The definition for a super-automaton is:

Definition 2. *If RL is a regular language and FA is an automaton for which $L(FA) \supseteq RL$, then FA is a super-automaton of RL.*

For the sake of completeness, the definition of a sub-automaton is also given, even though it does not play a direct role in this article.

Definition 3. *If RL is a regular language and FA is an automaton with $L(FA) \subseteq RL$, then FA is a sub-automaton of RL.*

As can be seen from the definitions above, a super-automaton accepts the same language as an exact automaton, and (possibly) also additional strings. The proof that Algorithm 2 produces a super-automaton of its input regular expression is presented next. Note that $\overrightarrow{L}(s)$ represents the right language of a state s, and $\overleftarrow{L}(s)$ represents the left language.

Theorem 4 (The construction of a super-automaton). *Algorithm 2 produces a super-automaton, FA^s, of $L(RE_{init})$, i.e. $L(FA^s) \supseteq L(RE_{init})$.*
Proof *Consider any two states s_1 and s_2 of $L(RE_{init})$. The language of s_1 is $L(s_1) = \overleftarrow{L}(s_1) \cdot \overrightarrow{L}(s_1)$ and similarly, the language of s_2 is $L(s_2) = \overleftarrow{L}(s_2) \cdot \overrightarrow{L}(s_2)$. If Algorithm 2 merges these states into one called $merge(s_1, s_2)$, then its language is:*

$$L(merge(s_1, s_2)) = (\overleftarrow{L}(s_1) \cup \overleftarrow{L}(s_2)) \cdot (\overrightarrow{L}(s_1) \cup \overrightarrow{L}(s_2))$$

Distributing $\cdot$ over $\cup$ gives

$$\begin{array}{lc} & \overleftarrow{L}(s_1) \cdot \overrightarrow{L}(s_1) \cup \overleftarrow{L}(s_1) \cdot \overrightarrow{L}(s_2) \cup \overleftarrow{L}(s_2) \cdot \overrightarrow{L}(s_1) \cup \overleftarrow{L}(s_2) \cdot \overrightarrow{L}(s_2) \\ \supseteq & \overleftarrow{L}(s_1) \cdot \overrightarrow{L}(s_1) \cup \overleftarrow{L}(s_2) \cdot \overrightarrow{L}(s_2) \\ = & L(s_1) \cup L(s_2) \end{array}$$

Since $L(merge(s_1, s_2)) \supseteq L(s_1) \cup L(s_2)$ for any two states s_1 and s_2 that are merged by Algorithm 2, it follows that $L(FA^s) \supseteq L(RE_{init})$.

Note that the notion of proper set containment does not play a role in theorem 4. Therefore it does not exclude the possibility that Algorithm 2 may produce an automaton FA^s that has the same language as the initial regular expression RE_{init}. What is particularly noteworthy is that this equality may hold even if two or more states are merged. An example of this is when $\overrightarrow{L}(s_1) = \overrightarrow{L}(s_2)$ and $\overleftarrow{L}(s_1) = \overleftarrow{L}(s_2)$, i.e. when the states being merged have the same language. In that case, Algorithm 2 partially fulfills the role of minimizing the output of Algorithm 1.

4 Non-determinism arising from the hashing algorithm

One of the consequences of merging states is that the resulting automaton may, under rather special circumstances, be non-deterministic. To see where non-determinism arises, consider two regular expressions RE_i and RE_j that represent *different* regular languages. Suppose that

$\exists\, s : (first(RE_j) \cap first(RE_i)) \cdot L(s^{-1}RE_i) \neq L(s^{-1}RE_j)$

Suppose also that the hash function used in Algorithm 2 hashed RE_i and RE_j to the same value, but hashed $s^{-1}RE_i$ and $s^{-1}RE_j$ to different values.

Example 5. Let $RE_i = sb, RE_j = sc, s^{-1}RE_i = b$ and $s^{-1}RE_j = c$ and let $hash(RE_i) = hash(RE_j)$ but let $hash(s^{-1}RE_i) \neq hash(s^{-1}RE_j)$. Assume that s is the only first symbol in RE_i and RE_j. In this case Algorithm 2 will construct the automaton shown in Figure 3. Note that, because of the duplicate out-transition for $hash(RE_i)$ (which has been merged with $hash(RE_j)$), the automaton is non-deterministic.

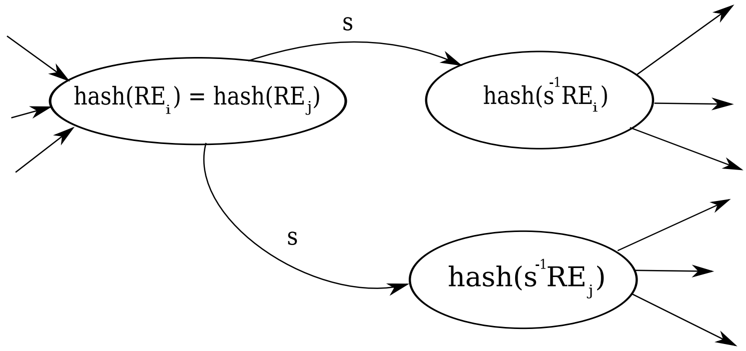


Figure 3. Non-determinism resulting from Algorithm 2

The fact that Algorithm 2 has been designed to generate a non-deterministic automaton is evident from its transition function, δ, that maps from a set of hashed states and transition symbol to a *set* of hashed states.

As is well-known, it is generally preferable to work with a DFA instead of an NFA. The reason for this is that the NFA cannot be represented as a two dimensional state transition table and this has implications for the size of the resulting automaton, as well as for the complexity of the associated FA-related algorithms. Of course, the NFA resulting from Algorithm 2 could be transformed to an equivalent DFA in the normal way. However, in Algorithm 3 a revised version of Algorithm 2 is presented that *directly* constructs a DFA. As in Algorithm 2, the revised algorithm relies on a hash function.

In the remainder of this text we will take the liberty of overloading the union operator, $\cup$. Thus, when used as a regular expression operator, as in $(RE_i \cup RE_j)$, the result is a regular expression such that $L(RE_i \cup RE_j) = L(RE_i) \cup L(RE_j)$. Nevertheless, the semantics of $\cup$ will be clear from the context in which it is used.

Algorithm 3 is premised on the observation that if the non-determinism in the automaton in Figure 3 is to be avoided, then an out-transition on s from a state $hash(RE_j)$ should not be inserted if:

- the state $hash(RE_j)$ corresponds to an existing state $hash(RE_i)$; and
- it is discovered that a transition on s out of state $hash(RE_i)$ already exists.

Indeed, in such an event, the existing transition from state $hash(RE_i)$ (which is equal to $hash(RE_j)$) should be removed. Additionally, a new transition on s from state $hash(RE_i)$ should then be provided to a *new* destination state. The new destination state should now be represented by the hashed value of of the regular expression $(s^{-1}RE_i \cup s^{-1}RE_j)$. The resulting automaton is shown in Figure 4. Note that the resulting automaton is a super-automaton of the regular language of the NDF that would have been constructed by Algorithm 2. It is therefore also a super-automaton of the regular language of the input regular expression to the algorithm.

The pseudocode of Algorithm 3 that appears below relies on the following:

- In the algorithm, the inverse mapping of *hash* is used, and is called *regex* Thus if RE is a regular expression such that $hash(RE) = p$, then $regex(p) = RE$. While one cannot in general rely on a hash function having an inverse, if the $hash(RE)$

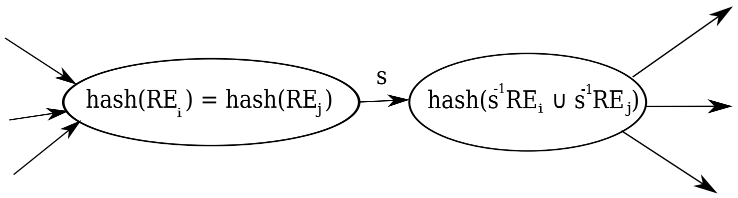


Figure 4. Determinism resulting from Algorithm 3

has previously been computed as p, it is a simple matter to store a backward reference to RE from the stored p.

- In the algorithm, the transition function δ is treated as a set of pairs of the form $\langle\langle h, s\rangle, d\rangle$, i.e. the first element of the pair is itself a pair. In this case, h is the hashed value of a regular expression corresponding to a source state, and d is the hashed value to the regular expression corresponding to the destination state when the symbol s is encountered.
- In order to update δ to account for a newly discovered transition from h to d upon input symbol s, a set union operation is used to augment the set that currently represents δ by an additional element. The form of the operation is thus: $\delta \cup \{\langle\langle h, s\rangle, d\rangle\}$
- In order to remove from δ a mapping that represents the transition from h to d upon input symbol s, the set difference operation is used. The form of the operation is thus: $\delta \setminus \{\langle\langle h, s\rangle, d\rangle\}$

The foregoing discussion about Algorithm 3 referred to the removal of an existing transition, the creation of a new transition and the creation of a new destination state. Examination of the details of the algorithm's pseudocode will indicate where and how these operations take place. However, both the discussion and the algorithm are silent about what should happen to the existing state labelled $hash(s^{-1}RE_i)$ in Figure 3. Should this state as well as its in- and out-transitions be removed? The answer to this question requires special consideration, which is given now, with reference to the contents of Figure 3.

Firstly, note that in dealing with the hash function clash, the algorithm inserts the regular expression $(s^{-1}RE_j \cup s^{-1}RE_i)$ into the *todo* set. (More specifically, the expression in the code $destination \cup regex(d)$ builds the regular expression $(s^{-1}RE_j \cup s^{-1}RE_i)$, which is then assigned to *destination*, and subsequently inserted into *todo* by computing $todo \cup \{destination\}$.) Thus, in some future iteration of the algorithm, it will be selected and all out-transitions that might have been generated previously on state $hash(s^{-1}RE_i)$ will be generated once more on state $hash(s^{-1}RE_i \cup s^{-1}RE_j)$. This is because $first(s^{-1}RE_i) \subseteq first(s^{-1}RE_i \cup s^{-1}RE_j)$. This means that if the only in-transition into state $hash(s^{-1}RE_i)$ was on s, nothing would be lost by discarding state $hash(s^{-1}RE_i)$ from Q, as well as all transitions out of it, as stored in the δ function.

However, if there were *more* transitions into state $hash(s^{-1}RE_i)$ than simply on s, then the algorithm might no longer generate a super-automaton of the input regular expression, if this state and associated transitions were to be discarded. For this reason, the state $hash(s^{-1}RE_i)$ should not be summarily discarded.

On the other hand, if s was indeed the only in-transition to state $hash(s^{-1}RE_i)$, then not removing this state means that it may become an unreachable state in

Algorithm 3 (Brzozowski's Algorithm with Hashing — DFA version)

func $Brz\_hash\_DFA(RE_{init})$
$\quad Q, \delta, F := \emptyset, \emptyset, \emptyset;$
$\quad done, todo, := \emptyset, \{RE_{init}\};$
$\quad$ **do** $todo \neq \emptyset \rightarrow$
$\quad\quad$ **let** RE_j be some regular expression such that $RE_j \in todo$;
$\quad\quad done, todo := done \cup \{RE_j\}, todo \backslash \{RE_j\};$
$\quad\quad h := hash(RE_j);$
$\quad\quad Q := Q \cup \{h\};$
$\quad\quad$ { expand out-transitions for symbols in the first symbol set of RE_j }
$\quad\quad$ **for** $s : first(RE_j) \rightarrow$
$\quad\quad\quad$ { compute the left derivative of RE_j with respect to$s$ }
$\quad\quad\quad destination := s^{-1}RE_j;$
$\quad\quad\quad$ **if** $(\exists d : \langle\langle h, s\rangle, d\rangle \in \delta) \rightarrow$
$\quad\quad\quad\quad \delta := \delta \setminus \{\langle\langle h, s\rangle, d\rangle\};$
$\quad\quad\quad\quad destination := destination \cup regex(d)$
$\quad\quad\quad [\,]\ (\nexists d : \langle\langle h, s\rangle, d\rangle \in \delta) \rightarrow$ **skip**
$\quad\quad\quad$ **fi**
$\quad\quad\quad$;
$\quad\quad\quad$ **if** $destination \notin done \cup todo \rightarrow$
$\quad\quad\quad\quad todo := todo \cup \{destination\}$
$\quad\quad\quad [\,]\ destination \in done \cup todo \rightarrow$ **skip**
$\quad\quad\quad$ **fi**
$\quad\quad\quad ;\delta := \delta \cup \{\langle\langle h, s\rangle, hash(destination)\rangle\}$
$\quad\quad$ **rof**
$\quad\quad$;
$\quad\quad$ **if** $nullable(RE_j) \rightarrow$
$\quad\quad\quad$ { The final states all have a right language containing the empty string }
$\quad\quad\quad F := F \cup \{h\}$
$\quad\quad [\,]\ \neg nullable(RE_j) \rightarrow$ **skip**
$\quad\quad$ **fi**
$\quad$ **od**;
$\quad$ **return** $\langle Q, \delta, hash(RE_{init}), F\rangle$
cnuf

---

the resulting DFA, since the algorithm removes its only in-transition. At the implementation level, this would mean that an amount of total storage used for the transition function would be wastefully occupied. In our implementation of Algorithm 3, the removal of such dead states and associated transitions is quite simple, but implementation-dependent. For this reason, and for the sake of brevity, these details have been omitted from Algorithm 3.

5 Characterising hash functions

5.1 A basis for measuring hash function quality

It has been established above that the automaton constructed through state merging is a super-automaton of the language associated with the algorithm's input regular expression. Informally, one might say that the "difference" between the languages recognised by these respective automata reflects the quality of the hash function used to generate the super-automaton. In this section, we propose a more precise and formal notion of the quality of the hash function. In the penultimate section, preliminary empirical investigations to assess these ideas are described.

The notion of hash function quality is based on the perception that the language associated with the algorithm's input regular expression has a unique minimum DFA, which may or may not be produced by Algorithm 3. The approach to finding this unique minimum DFA described in [6] is based on the notion of k-equivalence between states. The k-equivalence relation between states is inductively defined as follows [2]:

Definition 6. *Two states t_1 and t_2 are:*

- *0-equivalent iff they are both either accepting or rejecting states.*
- *k-equivalent iff for all input symbols s on the states t_1 and t_2, the next states $\delta(t_1, s)$ and $\delta(t_2, s)$ are $(k-1)$-equivalent.*
- **-equivalent iff they are k-equivalent for all values of k larger than some constant value.*

In [5] it is shown that if, in an automaton that contains $|Q|$ states, two states are k-equivalent for $k = |Q| - 2$, then these two states are *-equivalent. If two states are *-equivalent, it means that they represent the same regular language.

In the above-mentioned minimization algorithm, pairs of states of a given DFA are examined to determine their k-equivalence status. The algorithm therefore implicitly determines membership of k-equivalence classes in general, and of *-equivalence classes in particular. States in each *-equivalent class are eventually merged into a single state, resulting in the required unique minimal FA.

The foregoing suggests an approach to measuring the quality of a hash function, namely to associate quality with the extent to which state merging (caused by hash-clashes) approximates the merging of *-equivalent states.

However, before formalising this insight, note that hash functions in Algorithm 3 take regular expressions as input. For this reason, the notion k-equivalent regular expressions needs to be defined. The definition is analogous to the definition of k-equivalent states, namely:

Definition 7. *Two regular expressions RE_i and RE_j are:*

- *0-equivalent iff $nullable(RE_i) = nullable(RE_j)$*
- *k-equivalent iff $first(RE_i) = first(RE_j)$ and for all $s \in first(RE_i)$, $s^{-1}RE_i$ and $s^{-1}RE_j$ are $(k-1)$-equivalent.*

When a given hash function maps two regular expressions to the same value, we may enquire about the maximum k-equivalence. They may not have any equivalence relationship at all, or they be maximally k-equivalent for some $k < |Q|-2$, or they may be *-equivalent. Various weighting schemes could be proposed to reflect the overall quality of all clashes during a run of Algorithm 3: the higher the maximal k-equivalent status of two clashing regular expressions, the more favourably the clash should weigh in the overall measure. In the preliminary empirical experiments discussed below, the percentage of hash clashes that result from *-equivalent regular expressions relative to the total number of hash clashes is used as a measure of the quality of a number of different hash functions. Other measures of quality are not considered at this stage.

5.2 Ideal hash functions

The foregoing raises the question: what are the characteristics of an ideal hash function? To give such a characterisation, note that the signature of hash functions in Algorithm 3 are of the form $h : R^e \to \mathbb{N}$, where R^e is the set of regular expressions. Suppose that R^ℓ is the set of regular languages, and that $L : R^e \to R^\ell$, so that $L(R)$ is the regular language associated with regular expression R. Finally, suppose that f denotes a function $f : R^\ell \to \mathbb{N}$. An ideal hash function may now be defined as the composition of the latter two functions as follows:

Definition 8. *h is an ideal hash function for R^e iff $h = f \cdot L$ and f is an injection.*

This means an ideal hash function maps all regular expressions that have the same language (and only those expressions) to the same natural number. Put differently, an ideal hash function maps regular expressions to the same value if and only if they are *-equivalent. Thus, if it were possible to find an ideal hash function for use in Algorithm 3, then the algorithm would be guaranteed to produce the *minimum exact* DFA for the input regular expression.

Note that our notion of an *ideal* hash function for regular expressions is slightly different from the conventional notion of a *perfect* hash function for a set. The latter is an injection from that set to the natural numbers. In fact, the above definition could be reformulated to indicate that h is an ideal hash function on R^e if f is a perfect hash function on R^ℓ.

In fact, the remapping in Algorithm 1 may be viewed abstractly as the application of a perfect hash function on R^e, not to regular expressions representing the right language of a state, but to regular expressions representing the *language of a state*. This is the concept next defined.

5.3 A regular expression for the language of a state

Consider a regular expression, R, that is to be hashed or remapped in one of the algorithms previously discussed. Its language, $L(R)$, corresponds with the so-called *right* language of its associated state in the constructed FA, denote by $\overrightarrow{L}(state(R))$. That same state also has a *left* language, $\overleftarrow{L}(state(R))$, which is the set of all prefixes of all strings of the FA that pass through $state(R)$. Indeed the set of all strings passing through $state(R)$, is is called the language of $state(R)$. It is denoted by $L(state(R))$, and is given by $\overleftarrow{L}(state(R)) \cdot \overrightarrow{L}(state(R))$.

However, when constructing $state(R)$ during the execution of any of the algorithms, a regular expression whose language is $\overleftarrow{L}(state(R))$ is not available. All that is available during any iteration is the initial regular expression, RE_{init}, and the regular expression currently under consideration, R. Fortunately, this information is sufficient to find an explicit form for a regular expression, R', whose language corresponds $L(state(R))$, namely:

$$R' = (\Sigma^* \cdot R) \cap RE_{init} \tag{1}$$

Note that $(\Sigma^* \cdot R)$ designates all possible strings that end in a string that is in R's right language. Intersecting these strings with RE_{init} ensures that only strings in $L(state(R))$ remain.

Thus, abstractly, the remapping in Algorithm 1 can be viewed as a perfect hash function that is applied, not to the right language of a regular expression R, but to the corresponding regular expression R' as defined abovince the language of each state in a DFA is unique, this perfect hash function never merges any states. Of course, this claim has to be qualified in terms of the precise way in which a given application implements the remapping in the algorithm. A given implementation might have a more liberal notion of regular expression equality than strict lexicographic equality, taking into account operator properties such as idempotence or commutativity. (For example a may be treated as the same regular expression as $a \cup a$, and/or $a \cup b$ as the same regular expression as $b \cup a$, etc.)

Below we report briefly on a preliminary experiment in which Algorithm 2 is run with a variety of hash functions that are applied to R, while Algorithm 3 is run with these same hash functions applied to R'.

5.4 The effects of the modulo function

Most hash functions are of the form ($h(r)$ **mod** n)—i.e. they they apply modulo n to some integer value that they compute, where n reflects the address space being hashed to. This application of modulo n will clearly tend to undermine the quality of hash function h. If h happened to be an ideal hash function, then there is no guarantee that ($h(r)$ **mod** n) will deliver ideal behaviour.

6 Preliminary empirical investigations

In preliminary experiments to test the above ideas, Gödel numbers [4] were used to generate over 190 000 short different regular expressions of length 7, based on 4-character alphabet. Algorithm 1 was applied to each of these regular expressions. The largest FA generated by Algorithm 1 had 7 states.

In order to apply Algorithms 2 and 3, various hash functions were selected, based on the recommendations in [7]. In particular, various morphisms were selected, each structurally mapping the regular expressions to a different hash function. By this we mean that wherever an operator occurs in a regular expression, a corresponding integer operator is selected in the hash function which is structurally similar to the regular expression operator. For example, since the regular expression $\cup$-operator is idempotent and commutative, a hash function should be used that relies on an integer operator with these properties wherever the $\cup$-operator occurs in the regular expression. Such an integer operator might be, for example, addition, bitwise-and, bitwise-or, etc. Eight such mappings are shown in table 1.

Algorithm 2 was repeatedly invoked to construct DFAs for each of the more than 190000 regular expressions. Each of the eight hash functions in table 1 was used, as well as modulo i ($i = 2, \ldots, 5$) variants of the each hash function. Thus $8 \times 5 = 40$ DFAs were constructed for each regular expression. This entire experiment was then repeated using a modified version of Algorithm 3 in which the regular expression in equation (1) was used to determine the hashed value of each state, instead of the derivative representing the state's right language. (The results of Algorithms 2 and 3 based on hashing the right language of a state were similar, and consequently, the results of Algorithm 3 run in this mode are not further discussed here.)

In each application of Algorithms 2 and 3, whenever two states were to be merged, their *-equivalent status was assessed. This was done by verifying the k-equivalent status of the two states up to $k = |Q| - 2$, the point at which *-equivalence is attained [5]. Since the exact value of $|Q|$ was not known *a priori*, the upper bound on $|Q|$ established by Algorithm 1 was used, namely 7.

Mapping 1	Mapping 2
$\emptyset \mapsto 000\ldots000_{16}$	$\emptyset \mapsto 000\ldots000_{16}$
$\varepsilon \mapsto 000\ldots000_{16}$	$\varepsilon \mapsto 000\ldots000_{16}$
$arg^? \mapsto 000\ldots000_{16} \wedge arg$	$arg^? \mapsto 000\ldots000_{16} \wedge arg$
$arg^+ \mapsto \neg arg \vee (arg \vee (1 << (n-1)))$	$arg^+ = \neg arg \vee (arg \vee (1 << (n-1)))$
$arg^* \mapsto arg \vee (1 << (n-1))$	$arg^* = arg \vee (1 << (n-1))$
$arg_1 \cap arg_2 \mapsto arg_1 \wedge arg_2$	$arg_1 \cap arg_2 \mapsto arg_1 \vee arg_2$
$arg_1 \cup arg_2 \mapsto arg_1 \wedge arg_2$	$arg_1 \cup arg_2 \mapsto arg_1 \wedge arg_2$
$arg_1 \cdot arg_2 \mapsto \neg arg_1 \vee arg_2$	$arg_1 \cdot arg_2 \mapsto \neg arg_1 \vee arg_2$
Mapping 3	Mapping 4
$\emptyset \mapsto 000\ldots000_{16}$	$\emptyset \mapsto 000\ldots000_{16}$
$\varepsilon \mapsto 000\ldots000_{16}$	$\varepsilon \mapsto 000\ldots000_{16}$
$arg^? \mapsto 000\ldots000_{16} \vee arg$	$arg^? \mapsto 000\ldots000_{16} \vee arg$
$arg^+ \mapsto \neg arg \vee (arg \vee (1 << (n-1)))$	$arg^+ \mapsto \neg arg \vee (arg \vee (1 << (n-1)))$
$arg^* \mapsto arg \vee (1 << (n-1))$	$arg^* \mapsto arg \vee (1 << (n-1))$
$arg_1 \cap arg_2 \mapsto arg_1 \wedge arg_2$	$arg_1 \cap arg_2 \mapsto arg_1 \vee arg_2$
$arg_1 \cup arg_2 \mapsto arg_1 \vee arg_2$	$arg_1 \cup arg_2 \mapsto arg_1 \vee arg_2$
$arg_1 \cdot arg_2 \mapsto \neg arg_1 \vee arg_2$	$arg_1 \cdot arg_2 \mapsto \neg arg_1 \vee arg_2$
Mapping 5	Mapping 6
$\emptyset \mapsto FFF\ldots FFF_{16}$	$\emptyset \mapsto FFF\ldots FFF_{16}$
$\varepsilon \mapsto 000\ldots000_{16}$	$\varepsilon \mapsto 000\ldots000_{16}$
$arg^? \mapsto 000\ldots000_{16} \wedge arg$	$arg^? \mapsto 000\ldots000_{16} \wedge arg$
$arg^+ \neg arg \vee (arg \vee (1 << (n-1)))$	$arg^+ \neg arg \vee (arg \vee (1 << (n-1)))$
$arg^* \mapsto arg \vee (1 << (n-1))$	$arg^* \mapsto arg \vee (1 << (n-1))$
$arg_1 \cap arg_2 \mapsto arg_1 \wedge arg_2$	$arg_1 \cap arg_2 \mapsto arg_1 \vee arg_2$
$arg_1 \cup arg_2 \mapsto arg_1 \wedge arg_2$	$arg_1 \cup arg_2 \mapsto arg_1 \wedge arg_2$
$arg_1 \cdot arg_2 \mapsto \neg arg_1 \vee arg_2$	$arg_1 \cdot arg_2 \mapsto \neg arg_1 \vee arg_2$
Mapping 7	Mapping 8
$\emptyset \mapsto FFF\ldots FFF_{16}$	$\emptyset \mapsto FFF\ldots FFF_{16}$
$\varepsilon \mapsto 000\ldots000_{16}$	$\varepsilon \mapsto 000\ldots000_{16}$
$arg^? \mapsto 000\ldots000_{16} \vee arg$	$arg^? \mapsto 000\ldots000_{16} \vee arg$
$arg^+ \neg arg \vee (arg \vee (1 << (n-1)))$	$arg^+ \neg arg \vee (arg \vee (1 << (n-1)))$
$arg^* \mapsto arg \vee (1 << (n-1))$	$arg^* \mapsto arg \vee (1 << (n-1))$
$arg_1 \cap arg_2 \mapsto arg_1 \wedge arg_2$	$arg_1 \cap arg_2 \mapsto arg_1 \vee arg_2$
$arg_1 \cup arg_2 \mapsto arg_1 \vee arg_2$	$arg_1 \cup arg_2 \mapsto arg_1 \vee arg_2$
$arg_1 \cdot arg_2 \mapsto \neg arg_1 \vee arg_2$	$arg_1 \cdot arg_2 \mapsto \neg arg_1 \vee arg_2$

Table 1. Mappings of regular expression operators to hash function operators

Tables 2 and 3 summarise the results of these experiments. Columns headed *-eq % indicate the percentage of hash clashes (and thus merged states) that turned out to be *-equivalent for the specific hash function version. Columns headed "States" indicate the size of the largest DFAs (in terms of number of states) generated by the respective hash function. The tables reveal the following patterns:

– The relative hash function performance (as indicated in the *-eq% columns) appears very similar for the two algorithms: good hash functions seem to perform consistently well, and bad functions consistently badly. In fact, Spearman's rank correlation test was applied to the *-equivalent rankings in the two tables of the

	No MOD		MOD 2		MOD 3		MOD 4		MOD 5	
Mapping	*-eq %	States	*-eq %	States	*-eq %	States	*-eq %	States	*-eq %	States
1	77	4	73	2	73	3	75	4	75	4
2	79	4	73	2	74	3	76	4	76	4
3	83	4	75	2	76	3	78	4	79	4
4	82	4	74	2	75	3	77	4	77	4
5	77	4	73	2	73	3	75	4	75	4
6	79	4	73	2	74	3	76	4	75	4
7	83	4	75	2	76	3	78	4	79	4
8	81	4	74	2	74	3	76	4	76	4

Table 2. Algorithm 2 results: hashed regular expressions represent right language of states.

	No MOD		MOD 2		MOD 3		MOD 4		MOD 5	
Mapping	*-eq %	States	*-eq %	States	*-eq %	States	*-eq %	States	*-eq %	States
1	69	1	69	1	69	1	69	1	69	1
2	71	2	69	1	71	2	69	1	71	2
3	80	4	69	1	71	2	69	1	71	2
4	71	3	74	2	71	2	69	1	71	2
5	69	1	69	1	69	1	69	1	69	1
6	69	1	69	1	69	1	69	1	69	1
7	80	4	74	2	74	3	76	4	77	4
8	71	3	69	2	70	3	76	4	77	4

Table 3. Algorithm 3 results: hashed regular expressions represent full language of states.

hash functions without the modulo operation. A correlation value of 0.92 was obtained, which is well above the 95% confidence level of 0.72 (for sample size of 8) for rejecting the hypothesis that the hash function performance differed in the two algorithms.

- Data in table 2 indicating the largest DFA generated under each hash function is very much in line with expectations. All hash functions of a given modulo produce the same size largest DFAs, and these largest sizes rise as the modulo value rises. They attain a maximum size of 4, when modulo 4 is reached. This suggests (but does not prove) that 4 is the maximum size of the minimized DFA generated from the more than 190000 regular expressions.
- By contrast, data in table 3 in relation to the largest DFA generated under various hash functions, does not seem to be influenced significantly by the modulo operation. In fact, the overall quality measures in this table are lower than in table 2. If the inference above is correct that the maximum size of the minimized DFA is 4, then maximum DFA sizes increasingly less than 4 lead to super-automata increasingly different from the associated exact automata. This suggests that hashing on the language of a state is not a good idea. The reasons for this relatively poor performance will be further researched in future work.
- Entries in the *-eq % columns seem surprisingly high. It is interesting to note in table 3 that all hash functions that reduce the maximum sized DFA (and therefore all DFAs) down to 1 state, have a *-equivalent rating of 69%. This means that the percentage *-equivalent mergers attributable to *different* regular expressions being

hashed to the same value, ranges across the two tables from $69\% - 69\% = 0\%$ to a maximum of $83\% - 69\% = 14\%$.
- Hash functions 3 and 7 appear to perform consistently well, whereas hash functions 1 and 5 (and possibly 6) perform consistently badly. It would seem that the last three mappings in each block of table 1 play a critical role in determining the hash function quality. Worst case behaviour arises when the regular expression operator $\cup$ is associated with the bitwise operator $\wedge$ (see mappings 1, 2, 5 and 6), which significantly improves when the association is switched to the bitwise operator $\vee$ (see mappings 3, 4, 7 and 8). Optimal performance is reached (in mapping 3 and 7) when, in addition, $\cap$ is mapped to $\wedge$, but this mapping also leads to worst case performance is also reached (in mappings 1 and 5) if $\cup$ is wrongly mapped.

Even though the automata constructed are small in size, a large range of regular expressions have been tested, representing a diverse range of regular expression structures. Nevertheless, a shortcoming of this approach is it becomes impractical for larger regular expressions: the Gödel numbers involved become extremely large, making it impossible to iterate over them.

In the future, we intend generating a sample of random large regular expressions, to assess the impact of different hash functions under such circumstances.

7 Conclusions and Further Work

In this article, the consequences of regular expression hashing as a means of finite state automaton reduction was explored based on variations of Brzozowski's Algorithm. It was shown that a super-automaton is always constructed, no matter what the hash function may be. It was also demonstrated that a non-deterministic automaton can be constructed, and a new algorithm was put forward for constructing a deterministic FA, using the same approach as the original two algorithms.

An approach was proposed to measuring the quality of a hash function that derives a super-automaton of an exact automaton, based on k-equivalence classes on regular expressions. A derivation was also presented to represent the language of a state with regular expressions. These ideas were empirically tested on a large sample of relatively small regular expressions and their associated automata.

Further work will focus on searching for hash functions that are closer to the ideal, and on gaining a more precise understanding of why some hash functions are better than others, given the k-equivalence criteria and the definition of an ideal hash function. This will include exploring substitution variations on integer functions for regular expression operators.

References

1. J. Brzozowski: *Derivatives of regular expressions*. Journal of the Association of Computing Machinery, 11 October 1964, pp. pages 481–494.
2. S. Epp: *Discrete Mathematics with Applications*, International Tomson Publishing, Inc., 1995.
3. M. Frishert: *FIRE Works & FIRE Station: A finite automata & regular expression playground*, tech. rep., Technical University Eindhoven, 2004.
4. R. McNaughton: *Elementary Computability, Formal Languages and Automata*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.
5. B. W. Watson: *A taxonomy of finite automata minimization algorithms*, tech. rep., Technical University Eindhoven, 1994.

6. B. W. Watson and J. Daciuk: *An efficient incremental DFA minimization algorithm*. Journal of Natural Language Engineering, 9(1) March 2003, pp. 49–64.
7. B. W. Watson, D. G. Kourie, E. Ketcha Ngassam, T. Strauss, and L. Cleophas: *Efficient automata constructions and approximate automata*. International Journal of Foundations of Computer Science, Vol. 19(1) 2008, pp. 185–193.

Author Index