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 Ždárek

External Referees

Pavlos Antoniou
Miroslav Balík
Jérémy 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

The Road Coloring and Černý Conjecture by Avraham N. Trahtman ............ 1

Contributed Talks

Dynamic Burrows-Wheeler Transform by Mikaël Salson, Thierry Lecroq, Martine Léonard, and Laurent Mouchard ..................................................... 13

Lossless Image Compression by Block Matching on Practical Massively Parallel Architectures by Luigi Cinque and Sergio De Agostino ...................... 26

Speeding up Lossless Image Compression: Experimental Results on a Parallel Machine by Luigi Cinque, Sergio De Agostino, and Luca Lombardi .... 35

Huffman Coding with Non-Sorted Frequencies by Shmuel T. Klein and Dana Shapira .............................................................................................................. 46

In-place Update of Suffix Array while Recoding Words by Matthias Gallé, Pierre Peterlongo, and François Coste ................................................................. 54

The Virtual Suffix Tree: An Efficient Data Structure for Suffix Trees and Suffix Arrays by Jie Lin, Yue Jiang, and Don Adjeroh ................................. 68

Parameterized Suffix Arrays for Binary Strings by Satoshi Deguchi, Fumihito Higashijima, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda .... 84

An Adaptive Hybrid Pattern-Matching Algorithm on Indeterminate Strings by William F. Smyth, Shu Wang, and Mao Yu ............................................. 95

Conservative String Covering of Indeterminate Strings by Pavlos Antoniou, Maxime Crochemore, Costas S. Iliopoulos, Inuka Jayasekera, and Gad M. Landau .................................................................................................................. 108

On the Uniform Distribution of Strings by Sébastien Rebecchi and Jean-Michel Jolion ........................................................................................................ 116

Infinite Smooth Lyndon Words by Geneviève Paquin .................................. 126

New Lower Bounds for the Maximum Number of Runs in a String by Wataru Matsubara, Kazuhiko Kusano, Akira Ishino, Hideo Bannai, and Ayumi Shinohara .................................................................................................................. 140

Efficient Variants of the Backward-Oracle-Matching Algorithm by Simone Faro and Thierry Lecroq ................................................................. 146

Fast Optimal Algorithms for Computing All the Repeats in a String by Simon J. Puglisi, William F. Smyth, and Munina Yusufu ......................... 161
New Efficient Bit-Parallel Algorithms for the $\delta$-Matching Problem with $\alpha$-Bounded Gaps in Musical Sequences by Domenico Cantone, Salvatore Cristofaro, and Simone Faro .......................................................... 170

Average Value of Sum of Exponents of Runs in Strings by Kazuhiko Kusano, Wataru Matsubara, Akira Ishino, and Ayumi Shinohara ............................... 185

Usefulness of Directed Acyclic Subword Graphs in Problems Related to Standard Sturmian Words by Paweł Baturo, Marcin Piatkowski, and Wojciech Rytter ........................................... 193

Edit Distance with Single-Symbol Combinations and Splits by Manolis Christodoulakis and Gerhard Brey .......................................................... 208

A Concurrent Specification of an Incremental DFA Minimisation Algorithm by Tinus Strauss, Derrick G. Kourie, and Bruce W. Watson .............................. 218

On Regular Expression Hashing to Reduce FA Size by Wikus Coetser, Derrick G. Kourie, and Bruce W. Watson ......................................................... 227

Author Index ......................................................................................... 242
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].
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 Černý’s conjecture, was raised independently by distinct authors. Jan Černý 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 Černý and example of Černý, 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 \( O(n^3 + n^2q) \) time. The actual running time of our algorithm (\( O(n^2q) \)) on a lot of examples proved to be less than in the case of \( O(n^3q) \) 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 Černý 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 Černý 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 \( p \) to the state \( q \) and the edges of the path are consecutively labelled by \( \sigma_1, \ldots, \sigma_k \), then for \( s = \sigma_1 \ldots \sigma_k \in \Sigma^+ \) let us write \( q = ps \) and \( p \succeq r \).
Let $P_s$ be the map of the subset $P$ of states of an automaton by help of $s \in \Sigma^+$ and let $P_s^{-1}$ be the maximal set of states $Q$ such that $Qs \subseteq P$. For the transition graph $\Gamma$ of an automaton let $\Gamma_s$ 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 $\Gamma$ 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 $p, q$ of an automaton (of vertices of the transition graph) will be called synchronizing if $p_s = q_s$ for some $s \in \Sigma^+$. In the opposite case, if for any $s$ $ps \neq qs$, we call the pair deadlock.

A synchronizing pair of states $p, q$ of an automaton is called stable if for any word $u$ the pair $pu, qu$ is also synchronizing [6], [16].

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

A state [a vertex] $r$ is called sink of an automaton [of a graph] if $p \triangleright r$ for all states $p$.

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

## 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 $\rho$ on the set of the states of an automaton is called congruence if $\rho$ is equivalence and for any word $u$ from $p \rho q$ follows $pu \rho qu$. Let us formulate an important result from [16] in the following form:

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

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

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

For any $r$ from a strongly connected graph $\Gamma$, there exists a word $u$ such that $r = pu$ for $p$ from the $F$-clique $F$, whence $r$ 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 $\Gamma$. 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 \(p \in A \setminus B\) and \(q \in B \setminus A\) is not stable. Therefore for some word \(s\) the pair \((ps, qs)\) 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 \((A \cup B)\) 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 \(\Gamma\) have two incoming bunches. Then any coloring of \(\Gamma\) has a stable couple.

Proof. If a vertex \(p\) has two incoming bunches from vertices \(q\) and \(r\), then the couple \(q, r\) is stable for any coloring because \(q\alpha = r\alpha = p\) 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 \(\Gamma\) a spanning subgraph of \(\Gamma\) if to \(S\) belong all vertices of \(\Gamma\) 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 \(p\) through the edges of the tree of the spanning set \(S\) to the root of the tree is called the level of \(p\) 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 \(\Gamma\) and let all edges of \(S\) have a color \(\alpha\) by some coloring of \(\Gamma\). Then for any F-clique \(F\) of the coloring holds \(|F \cap L| \leq 1|\).

Proof. Some power of \(\alpha\) 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 \(\Gamma\) have a spanning subgraph \(R\) of only disjoint cycles (without trees). Then \(\Gamma\) 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 \(\Gamma\), not all edges belong to a bunch. Therefore there exist two edges \(u = p \to q \notin R\) and \(v = p \to s \in R\) with common first vertex \(p\) but such that \(q \neq s\). Let us replace the edge \(v = p \to s\) from \(R\) by \(u\). Then only the vertex \(s\) has maximal level \(L > 0\) in the new spanning subgraph.

**Lemma 8.** Let any vertex of an AGW graph \(\Gamma\) have no two incoming bunches. Then \(\Gamma\) 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 $p$ of maximal level $L$ and edge $\bar{a}$ from vertex $b$ to the tree root $r \in T$ on the path of length $L$ from $p$. Let the root $r$ belong to the cycle $H$ of $R$ with the edge $\bar{c} = c \rightarrow r \in H$. There exists also an edge $\bar{a} = a \rightarrow p$ that does not belong to $R$ because $\Gamma$ is strongly connected and $p$ has no incoming edge from $R$.

Let us consider the path from $p$ to $r$ 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 $a$ by the edge $\bar{a} = a \rightarrow p$,
2) replace the edge $\bar{b}$ from $R$ by some other outgoing edge of the vertex $b$,
3) replace the edge $\bar{c}$ from $R$ by some other outgoing edge of the vertex $c$.

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 $a \notin H$. If $a$ belongs to a path in $T$ from $p$ to $r$ then a new cycle with part of the path and edge $a \rightarrow p$ 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 $a$ in the new spanning subgraph is $L + 1$ and the vertex $r$ is a root of the new tree containing all vertices of maximal level (in particular, the vertex $a$ or its ancestors in $R$).

So let us assume $a \in H$ and suppose $\bar{w} = a \rightarrow d \in H$. In this case the vertices $p$, $r$ and $a$ 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 $r$ to $a$ in $H$ is $r_1$ then $H_1$ has length $L + r_1 + 1$. A path to $r$ from the vertex $d$ 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 $d$ to $r$ in a tree of $R_1$ (and the length of $d$) is greater than $L$ and the level of $d$ (or of some other ancestor of $r$ in a tree from $R_1$) is the desired unique maximal level.

So assume for further consideration $L = r_2$ and $a \in H$. Analogously, for any vertex of maximal level $L$ with root in the cycle $H$ and incoming edge from a vertex $a_1$ the proof can be reduced to the case $a_1 \in H$ and $L = r_2$ for the corresponding new value of $r_2$.

2) Suppose the set of outgoing edges of the vertex $b$ is not a bunch. So one can replace in $R$ the edge $\bar{b}$ from the vertex $b$ by an edge $\bar{v}$ from $b$ to a vertex $v \neq r$. 
The vertex \( v \) 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 \( v \) 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 \( p \) to the new root is extended. So only the tree \( T_1 \) has states of new maximal level.

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

Now there remains only the case when \( v \) belongs to the cycle \( H \). The vertex \( p \) also has level \( L \) in new tree \( T_1 \) with root \( v \). 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 \( d \) to the new root \( v \in H \).

For the spanning subgraph \( R_1 \), one can obtain \( L = r_3 \) just as it was done on the step 1) for \( R \). From \( v \neq r \) 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 \( b \) is a bunch to \( r \).  

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

Let us replace in \( R \) the edge \( \bar{c} \) by an edge \( \bar{c} = c \rightarrow u \) such that \( u \neq r \). The vertex \( u \) 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 \( u \not\in T \). Then the tree \( T \) is a part of a new tree with a new root and the path from \( p \) to the new root is extended at least by a part of \( H \) from the former root \( r \). The new level of \( p \) 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 \( \Gamma \) has a coloring with stable couple.

Proof. By Lemma 4, in the case of vertex with two incoming bunches \( \Gamma \) has a coloring with stable couples. In opposite case, by Lemma 8, \( \Gamma \) 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 \( \alpha \) and denote by \( C \) the set of all vertices from the cycles of \( R \). Then let us color the remaining edges of \( \Gamma \) by other colors arbitrarily.

By Lemma 2, in a strongly connected graph \( \Gamma \) for every word \( s \) and \( F \)-clique \( F \) of size \( |F| \geq 1 \), the set \( F \) also is an \( F \)-clique of the same size and for any state \( p \) there exists an \( F \)-clique \( F \) such that \( p \in F \).

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 \( \alpha^{L-1} \) 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 \( \alpha \) from any tree of \( R \) leads to the root of the tree, the root belongs to a cycle colored by \( \alpha \) 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 $\alpha$. So for any $p$ from $C$ as well as from $F_1 \cap C$ holds $p\alpha^m = p$. 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 $\Gamma$ has synchronizing coloring.

The proof follows from Theorems 9 and 1.

3 Some auxiliary properties

**Lemma 11.** Suppose $p \not\in \Gamma s$. Then $p \not\in \Gamma u$ for any word $u$.

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

**Lemma 12.** Suppose $p \not\in \Gamma s$ for a word $s$ and a state $p$ of transition graph $\Gamma$ of DFA.

Then there exist two minimal integer $k$ and $r$ such that $p s^k = p s^{k+r}$. The pair of states $p, ps^r$ has 2-reset word $s^k$ and for every $i < k$ the pair of states $ps^i, ps^{r+i}$ 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 $p, ps, ..., ps^k$ on $ps^k$.

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

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

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

Then there exists a word $s$ of length of the cycle $C$ such that $rs = r$ and $|\Gamma s| < |\Gamma|$. For some state $p \in \Gamma \setminus \Gamma s$ there exists a minimal integer $k$ such that $ps^k = ps^{k+1}$. The pair of states $p, ps^k$ has 2-reset word $s^k$ and for every $i < k$ the pair of states $ps^i, ps^k$ 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 $\alpha$ can be obtained from consecutive letters on the edges of the cycle $C$. Therefore $|s|$ is equal to the length of the cycle and $rs = r$. $|\Gamma s| < |\Gamma|$ follows from $r \alpha = t \alpha$.

From $rs = r \not= t$ and $r \alpha = t \alpha$ follows that $ts = r \not= t$, whence $r = ts^i \not= t$ for any integer $i$. In the case $t \in \Gamma \setminus \Gamma s$ suppose $p = t$, and so the state $p$ is defined.

In opposite case the state $t$ has by mapping $s$ some preimage $ts^{-1}$ and in view of $ts^i \not= t$ for all $i$ there exists an integer $k$ (only one) such that the state $ts^{-k}$ belongs to $\Gamma \setminus \Gamma s$. Now suppose $p = ts^{-k}$. One has $ps^k = ps^{k+1} = r$ for $p$ from $\Gamma \setminus \Gamma s$.

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

Lemma 14. Let \( \Gamma \) 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 \( \Gamma \). \( S \) 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 \( r, p \) the first letter \( \alpha \) 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 \( r\alpha, p\alpha \). 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 \( O(n^2 q) \) 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 \( \Gamma_s \) 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 \( \Gamma_s \) with at least two sink states. The next step is the consideration of \( \Gamma_s^2 \). We unite any pair of states \( (p, r) \) and \( (r, p) \), all states \( (r, r) \) 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 \( \Gamma \) 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 \( \Gamma^2 \) and the reverse graph \( I \) of \( \Gamma \). The graph \( I \) is not deterministic for synchronizing graph \( \Gamma \). Suppose that the graph \( \Gamma \) is synchronizing, all sink states are found on the stage of checking of the synchronizability, the graph \( \Gamma^2 \) and the reverse graph \( I \) were build.

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

Let us order the set of states \( (r, p) \) 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 \( (r, p) \) 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 \( \Gamma w \) of the graph \( \Gamma \) induced by the minimal 2-reset word \( w \) of the pair of states \( r, p \). This stage begins from the shortest words \( w \) and therefore is linear for any considered pair of states \( r, p \). 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 \( \Gamma_i \) 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 \( w_1, \ldots, w_i \). Let us check pairs of states corresponding to the words from \( G \). If the pair belongs to \( \Gamma \) then the corresponding minimal reset word \( w_{i+1} \) together with its powers may be used for to find the image \( \Gamma_{i+1} \).

In the case no minimal 2-reset word of a pair from \( \Gamma_i \) was marked, let us consider the products of marked words. If some product unites a pairs of states of \( \Gamma_i \) 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 \( \Gamma \). 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 \( \Gamma_i \).

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 \( r, p \) from \( \Gamma_i \) with reset word \( w_i \). Suppose \( w_i = w_i v_i \) 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 \( \Gamma_i \) 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 \( w_1, \ldots, w_i \) such that \( w_i = u_i v_i \) is a reset word for some pair of states from \( \Gamma_{i-1} \) and \( v_i \) is a marked element of the subsemigroup \( A_{i-1} \).

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

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

Let the set \( w_1, \ldots, w_i \) 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 \( \Gamma \) 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 \( \Gamma_i \). The defect of the mapping is growing on every step. After not over than \( |\Gamma| - 1 \) steps \( \Gamma \) 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 per-
cents 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 $n=28$):

![Automaton](image)

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: \(abcabaca, acbaaaca, baab, acba, bacb\). 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 Černý 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 Černý 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 Černý sequence.

<table>
<thead>
<tr>
<th>size of the automaton</th>
<th>(n=3)</th>
<th>(n=4)</th>
<th>(n=5)</th>
<th>(n=6)</th>
<th>(n=7)</th>
<th>(n=8)</th>
<th>(n=9)</th>
<th>(n=10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((n-1)^2)</td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>25</td>
<td>36</td>
<td>49</td>
<td>64</td>
<td>81</td>
</tr>
<tr>
<td>max length, 2 letters</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>max length, 3 letters</td>
<td>0 *</td>
<td>8 *</td>
<td>15 *</td>
<td>23 *</td>
<td>32 *</td>
<td>44 *</td>
<td>58 *</td>
<td>74 *</td>
</tr>
<tr>
<td>max length, 4 letters</td>
<td>0 *</td>
<td>8 *</td>
<td>15 *</td>
<td>22 *</td>
<td>30 *</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

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 Černý 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

Dynamic Burrows-Wheeler Transform

Mikaël Salson¹*, Thierry Lecroq¹, Martine Léonard¹, and Laurent Mouchard¹,²

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 bandwidth, 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 traditionally 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 $\sigma$ 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

Proceedings of PSC 2008, Jan Holub and Jan Ždarek (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 $\Sigma$ of size $\sigma$. 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 $\Sigma$ and is smaller than any other letter of $\Sigma$.

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.** $T_i = T^{[(i+1) \mod |T|][n]}$ that will be simply denoted by $T_n^{[(i+1) \mod |T|]}$ 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, $\pi$ is the fitting sort function.

![Figure 1. $bwt(GTCT\$) = L = TT\$CG](image-url)
Remark 2. We can observe that $\pi$ 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 $\pi$. 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 $\pi(j)$.

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 $rank_A(L, i) = rank_A(F, i)$ for all $0 \leq i < 3$, 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.

\begin{table}[h]
\centering
\begin{tabular}{c|c|c|c}
$i$ & $\pi(i)$ & $(\pi(i) - 1) \mod 5$ & $\pi^{-1}((\pi(i) - 1) \mod 5)$ \\
\hline
0 & 4 & 3 & 3 \\
1 & 2 & 1 & 4 \\
2 & 0 & 4 & 0 \\
3 & 3 & 2 & 1 \\
4 & 1 & 0 & 2 \\
\end{tabular}
\caption{Using $\pi$ to recover the original word $T$.}
\end{table}

Using this formula, we obtain a permutation $0 \rightarrow 3, 3 \rightarrow 1, 1 \rightarrow 4, 4 \rightarrow 2, 2 \rightarrow 0$. 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 $U[0..i]$.

\begin{table}[h]
\centering
\begin{tabular}{c|c|c|c|c|c|c}
$i$ & $F$ & $L$ & $|F|$ & $|L|$ & $LF$ & $\pi$ \\
\hline
0 & \$ & 1 & 0 & 1 & 3 \\
1 & C & T & 4 & 1 & 4 \\
2 & G & \$ & 0 & 0 & 0 \\
3 & T & C & 1 & 1 & 1 \\
4 & T & G & 2 & 2 & 2 \\
\end{tabular}
\caption{Table $LF$: Establishing a relation between $L$ and $F$.}
\end{table}

The second $T$ in $L$ ($rank_T(L, 1) = 2$) is linked to the second $T$ in $F$ ($rank_T(F, 4) = 2$). 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 $F$.
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 $\text{bwt}(T')$. 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:

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

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), $T'^{i+1} = T[j \ldots n-1]S T[0 \ldots i-1]c T[i \ldots j-1]$, $\forall j \geq i$ meaning that $T'^{j+1}$ and $T[j]$ are sharing a common prefix $T[j \ldots n-1]S T[0 \ldots i-1]$.

**Lemma 4.** Inserting a letter $c$ at position $i$ in $T$ has no effect on the respective ranking of cyclic 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 $k < |T|$, $T[j] [0 \ldots k] \leq T[j'] [0 \ldots k]$. The prefix of $T[j]$ ending before the sentinel letter $\$ is of length $n - j < |T|$, and therefore $T[j] [0 \ldots n-j-1] \leq T[j'] [0 \ldots n-j-1]$. That is, $T[j] [0 \ldots n-j] \leq T[j'] [0 \ldots j' + n-j-1]$ (grey rectangles below). Moreover $\$, the smallest letter of $\Sigma$, 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 $T[j] [0 \ldots n-j] < T[j'] [0 \ldots n-j]$.

Since $T'[j + 1 \ldots n]S = T[j \ldots n-1]S$ and $T'[j' + 1 \ldots n + j' - j - 1] = T[j' \ldots n + j' - j]$, we have $T'[j + 1 \ldots n]S < T'[j' + 1 \ldots n + j' - j + 1]$. So $T'[j + 1 \ldots n]Su < T'[j' + 1 \ldots n + j' - j + 1]v$, for all texts $u, v$ over $\Sigma$. Finally, $T[j] < T[j'] \iff T[j]T[j'+1] < T[j'+1]$.

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 $S$ 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 $L$ → 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 $\pi$ (see [9,16]) and computing $k$ such that $\pi(k) = i$.

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

\begin{align*}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
T = C T C T G C S & \rightarrow T' = C T G C T G S
\end{align*}

(Ia): no modification.

(Ib): $T[i]$ is at position $k=3$ ($\pi(3)=2$), $L[3] \rightarrow G$.

\[
\begin{array}{cccc}
\pi & F & L & F & L \\
6 & S & C & S & C \\
5 & C & G & C & G \\
0 & C & S & C & S \\
\end{array}
\]

\[
\begin{array}{cccc}
i = 2 & C & T & \xrightarrow{\text{(Ib)}} & C & G \\
4 & G & T & G & T \\
1 & T & C & T & C \\
3 & T & C & T & C \\
\end{array}
\]

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

---

*Note: The table and diagram are placeholders for the actual input.*
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 \( c \), we now have to consider the brand new cyclic shift that starts with the added \( c \), that is \( T'[i] = cT[i] = CT[i..n-1]ST[0..i-1] \) 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 \( \text{rank}_c(L, k) \)).

Insertion of \( G \) at position \( i=2 \) in \( T \):

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
T = CTCTGC$ \rightarrow T' = CTGCTGC$ \\
\end{array}
\]

(IIa): \( T'[i] \) is inserted in the table at position \( LF[k] \).

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

\( T'[i] \) finishes with a \( G \) which is the second \( G \) in \( L \).

\( T' \) begins with this \( G \) which has to be the second \( G \) in \( F \).

After stage (IIa), we have: two \( G \)s in \( F \) and two \( G \)s in \( L \), two \( T \)s in \( F \) and two \( T \)s in \( L \).

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. \( AAG$ < AGS$ BUT ATAG$ > AGSAT \)). 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 \( \text{index} \) returns the position of a cyclic shift in the matrix.

\text{REORDER}(L, i)

1. \( j \leftarrow \text{index}(T'[i-1]) \) \( \triangleright \) Gives the position of \( T'[i-1] \)
2. \( j' \leftarrow \text{LF}[\text{index}(T'[i])] \) \( \triangleright \) Gives the computed position of \( T'[i-1] \)
3. \text{while} \( j \neq j' \) \text{do}
   4. \( \text{new}_j \leftarrow \text{LF}[j] \)
   5. \( \text{MOVEROW}(j, j') \)
   6. \( \text{UPDATELF}(j', \text{new}_j) \)
   7. \( j \leftarrow \text{new}_j \)
   8. \( j' \leftarrow \text{LF}[j'] \)

We now prove that the algorithm \text{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 \( F = $AAACCGG\), \( C[\$] = 0 \) and \( C[G] = 6 \).

\textbf{Lemma 6.} \( \forall j < i, \exists j' > j, T'[j] < T'[j'] \iff \text{index}(T'[j]) < \text{index}(T'[j']) \), after the iteration considering \( T'[j] \), in \text{REORDER}. 

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

From the previous lemma, $\forall j' \geq i + 1$ we have $T^{[i+1]}_{(j')} < T^{[j']} \iff T^{[i]} < T^{[j'-1]}$.

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

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

By definition, $T^{[j-1]}_0 = T^{[j]}_{n+1}$, let $r = \text{rank}_{T^{[j]}_n} (L, \text{index}(T^{[j]}))$. The index of $T^{[j-1]}$ is computed using LF with the following formula:

\[
\text{index}(T^{[j-1]}) = C[T^{[j-1]}_0] + r - 1.
\]

We distinguish two different cases:

- if the first letter of $T^{[j-1]}$ is different from the first one of $T^{[j]}$, then $C[T^{[j-1]}_0] \neq C[T^{[j]}_0]$. Without loss of generality, consider $T^{[j-1]}_0 < T^{[j]}_0$. By definition, $r \leq C[T^{[j-1]}_0] - C[T^{[j]}_0]$. Thus $C[T^{[j-1]}_0] + r - 1 \leq C[T^{[j]}_0] - 1$. However, the rank computed for the index of $T^{[j]}$ is strictly positive. Finally $T^{[j-1]}_0 < T^{[j]}_0; \text{index}(T^{[j-1]}) < \text{index}(T^{[j]})$.

- otherwise, both letters are equal. Then, we can write $T^{[j-1]}_0 < T^{[j]}_0 \iff T^{[j-1]}_0[1 \ldots n + 1] < T^{[j]}_0[1 \ldots n + 1] \iff T^{[j-1]}_0[1 \ldots n + 1][T^{[j-1]}_0] < T^{[j]}_0[1 \ldots n + 1][T^{[j]}_0] \iff T^{[j]}_0 < T^{[j+1]}_0$.

We know that the lemma is true for $j$, then we have $T^{[j]}_0 < T^{[j+1]}_0 \iff \text{index}(T^{[j]}) < \text{index}(T^{[j+1]})$.

Let $k = \text{index}(T^{[j]}), k' = \text{index}(T^{[j+1]}), r' = \text{rank}_{T^{[j]}_n} (L, k')$ and $c = T^{[j-1]}_0 = T^{[j]}_0$.

\[
\text{index}(T^{[j-1]}) = C[c] + \text{rank}_c(L, k) - 1 \quad \text{index}(T^{[j]}) = C[c] + \text{rank}_c(L, k') - 1
\]

We know that $T^{[j]}_0 = L_k = c$, $T^{[j+1]}_0 = L_{k'} = c$ and $k' > k$. So $\text{rank}_c(L, k') > \text{rank}_c(L, k)$ and eventually $\text{index}(T^{[j-1]}) < \text{index}(T^{[j+1]})$.

Finally, $T^{[j-1]}_0 < T^{[j]}_0; \text{index}(T^{[j-1]}) < \text{index}(T^{[j]})$. We can prove $T^{[j-1]}_0 < T^{[j]}_0 \iff \text{index}(T^{[j-1]}) < \text{index}(T^{[j]})$ 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 $\forall j, j', T^{[j]}_0 < T^{[j']}_0 \iff \text{index}(T^{[j]}) < \text{index}(T^{[j']})$. 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. $\text{index}(T^{[i]}) = \text{index}(T^{[k]}) \iff \text{index}(T^{[j]}) = \text{index}(T^{[j']})$, for $j < k < i$.

Proof. Given $\text{index}(T^{[k]})$,

\[
\text{index}(T^{[k-1]}) = C[T^{[k]}_n] + \text{rank}_{T^{[k]}_n} (L, \text{index}(T^{[k]})) = C[T^{[k]}_n] + \text{rank}_{T^{[k]}_n} (L, \text{index}(T^{[k]})) = \text{index}(T^{[k-1]})
\]

Therefore, $\text{index}(T^{[k]}) = \text{index}(T^{[k]}) \iff \text{index}(T^{[k-1]}) = \text{index}(T^{[k-1]})$.

By induction, we prove the property for each $j < k$. Consider a cyclic shift $T^{[j]}$ and $k$ the number of times $T^{[j]}_n$ 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 $T^{[j]}_n$. 

Proceedings of the Prague Stringology Conference 2008

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, $L = \text{bwt}(T')$.

### 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 $T' = T[0 \ldots i-1]S[0 \ldots m-1]T[i \ldots n]$ with $m > 1$.

The four stages can be extended as follows:

1. **(Ia)** Cyclic shifts $T'_{j}$ with $j > i + m$: unchanged.
2. **(Ib)** Cyclic shift $T'_{i+m}$: modification $L = S_{i-1}$ instead of $T_{i-1}$.
3. **(IIa)** Cyclic shifts $T'_{j}$ from $j = i + m - 1$ downto $i + 1$:
   - **insertion** $F = S_{j-1}$ and $L = S_{j-1}$.
   - $T'^{[i]}$: **insertion** $F = S_{0}$ and $L = T_{i-1}$.
4. **(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 $\text{rank}_{T_{i-1}}$ 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 $\text{rank}_{T_{i-1}}$ 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 $L(F(\ell))$ such that $L(\ell) = T_{i-1}$ and $\ell > \pi^{-1}(i)$, then we must add one to the result of $L(F(\ell))$ (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 $T' = T[0 \ldots i-1]T[i + m \ldots n]$. The four stages can be modified as follows:

1. **(Ia)** Cyclic shifts $T'_{j}$ with $j > i + m$: unchanged.
2. **(Ib)** Cyclic shift $T'^{[i+m]}$: modification $L = T_{i-1}$ instead of $T_{i+m-1}$.
3. **(IIa)** Cyclic shifts $T'_{j}$ from $j = i + m - 1$ downto $i$:
   - **deletion** of the corresponding row.
   - We still have to pay attention to $\text{rank}_{T_{i-1}}$: 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 $\text{rank}_{T_{i-1}}$.
4. **(IIb)** Cyclic shifts $T'_{j}$ with $j < i$: as presented in algorithm on page 18.
\[ \pi F L F L i \]

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 \( L[6] = C \) and \( 6 > \pi^{-1}(1) = 5 \), we must add one to the original \( LF \) value 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.

(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. \( A^m$ \rightarrow A^mC$). 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 \( \text{rank}_c \), 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.

<table>
<thead>
<tr>
<th>Entropy (H_0)</th>
<th>100 KB</th>
<th>250 KB</th>
<th>500 KB</th>
<th>1 MB</th>
<th>1 MB:100 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNA</td>
<td>1.982</td>
<td>10.12</td>
<td>9.52</td>
<td>10.26</td>
<td>10.91</td>
</tr>
<tr>
<td>English</td>
<td>4.53</td>
<td>7.75</td>
<td>7.94</td>
<td>9.03</td>
<td>10.31</td>
</tr>
<tr>
<td>Fibo</td>
<td>0.96</td>
<td>25,414.13</td>
<td>63,527.09</td>
<td>119,780.37</td>
<td>261,910.49</td>
</tr>
<tr>
<td>Random</td>
<td>6.60</td>
<td>3.89</td>
<td>4.03</td>
<td>4.21</td>
<td>4.36</td>
</tr>
<tr>
<td>Source</td>
<td>5.54</td>
<td>92.88</td>
<td>55.76</td>
<td>118.54</td>
<td>72.22</td>
</tr>
<tr>
<td>XML</td>
<td>5.23</td>
<td>26.43</td>
<td>28.84</td>
<td>34.8</td>
<td>44.08</td>
</tr>
</tbody>
</table>

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 Manisicalco 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](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 $L$:

```plaintext
RetriveSA(L)
1  j ← index(L, T[n])
2  i ← 0
3  repeat
4    SA[j] ← i
5    j ← LF[j]
6    i ← (i − 1) mod (n + 1)
7  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


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 $m \times m'$ 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...
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 \( O(n \log M) \) 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 \( O(\log M \log n) \) time and \( O(n/\log n) \) 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 \( O(\log M \log n) \) time and \( O(n/\log n) \) processors on the PRAM-EREW. On the mesh of trees though the decoder required \( O(\log^2 n) \) time. In this paper, we show how to implement \( O(\log M \log n) \) time, \( O(n/\log n) \) 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 \( m \times m' \) image \( I \) in \( x \times y \) 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 \( O(\log M \log n) \) 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 \( A_{(i-1)2^{k-1}+1,j}, A_{(i-1)2^{k-1}+2,j}, \ldots, A_{i2^{k-1}j} \), with \( i \) odd, were merged, then they will merge with areas
$A_{i,2^{k-1}+1,j}$, $A_{i,2^{k-1}+2,j}$, ..., $A_{i,(i+1)2^{k-1},j}$, if they are monochromatic with the same color. The same is done horizontally for $A_{i,(j-1)2^{k-1}+1}$, $A_{i,(j-1)2^{k-1}+2}$, ..., $A_{i,2^{k-1}}$, with $j$ odd, and $A_{i,j,2^{k-1}+1}$, $A_{i,j,2^{k-1}+2}$, ..., $A_{i,i+1,2^{k-1}}$. 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
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 in 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 \(O(\log n)\) time with \(O(n/\log n)\) processors on the PRAM EREW, since the row image size is greater than the size of the sequence. Then, still in \(O(\log n)\) time with \(O(n/\log n)\) 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 \(O(\log n)\) time and \(O(n/\log n)\) processors on the PRAM EREW.

The third phase completes the decoding. In the previous subsection, we denoted with \(A_{i,j}\) for \(1 \leq i \leq \lceil m/x \rceil\) and \(1 \leq j \leq \lceil m'/y \rceil\) 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 \(A_{i,(i-1)2^k-1+1,j}, A_{i,(i-1)2^k-1+2,j}, \ldots, A_{i,2^k-1,j}\), with \(i\) odd, decodes the areas \(A_{i,2^k-1+1,j}, A_{i,2^k-1+2,j}, \ldots, A_{i,(i+1)2^k-1,j}\), respectively. The same is done horizontally for \(A_{i,(j-1)2^k-1+1}, A_{i,(j-1)2^k-1+2}, \ldots, A_{i,j,2^k-1}\), with \(j\) odd, and \(A_{i,j,2^k-1+1}, A_{i,j,2^k-1+2}, \ldots, A_{i,(j+1)2^k-1}\). After \(O(\log M)\) steps the image is entirely decompressed. 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 decoder is realized with \(O(\log M \log n)\) time and \(O(n/\log n)\) 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 \(N \times N\) grid where a complete binary tree of processors is built on
Proceedings of the Prague Stringology Conference 2008

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

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 $m \times m'$ image. Then, the number of processors of the mesh of trees is $O(n / \log n)$ and we can store the logarithmic rectangular areas into which the parallel algorithm partitions the image into the $N \times N$ 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 $A_{(i-1)2^{k-1}+1,j}, A_{(i-1)2^{k-1}+2,j}, \ldots, A_{2^{k-1},j}$, with $i$ odd, were merged, the processor storing area $A_{2^{k-1},j}$ will broadcast to the processors storing the areas $A_{2^{k-1}+1,j}, A_{2^{k-1}+2,j}, \ldots, A_{(i+1)2^{k-1},j}$ 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 $A_{i,(j-1)2^{k-1}+1}, A_{i,(j-1)2^{k-1}+2}, \ldots, A_{i,j2^{k-1}}$, with $j$ odd, and $A_{i,j2^{k-1}+1}, A_{i,j2^{k-1}+2}, \ldots, A_{i,(j+1)2^{k-1}}$, 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 $N \times N$ pyramid is a network consisting of $\log N + 1$ two-dimensional grids with $N$ being a power of 2, each one of size $N/2^k \times N/2^k$ for $0 \leq k \leq \log N$. The grids are interconnected so that the $(i, j)$ processor on the $2^k \times 2^k$ grid is connected to processors $(2i - 1, 2j - 1), (2i - 1, 2j), (2i, 2j - 1)$ and $(2i, 2j)$ on the $2^{k+1} \times 2^{k+1}$ 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 $m \times m'$ 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 $N \times N$ 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.
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 $\ell_R$ 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 $A_{(i-1)2^{k-1}+1,j}$, $A_{(i-1)2^{k-1}+2,j}$, \ldots, $A_{2^{k-1}+1,j}$, with $i$ odd, were merged for $w_1 \leq j \leq w_2$, the processor storing area $A_{2^{k-1},w_2}$ will broadcast to the processors storing the areas $A_{i,2^{k-1}+1,j}$, $A_{i,2^{k-1}+2,j}$, \ldots, $A_{i,2^{k-1}+1,j}$ 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 $A_{i,(j-1)2^{k-1}+1}$, $A_{i,(j-1)2^{k-1}+2}$, \ldots, $A_{i,j2^{k-1}}$, with $j$ odd, were merged for $\ell_1 \leq i \leq \ell_2$, the processor storing area $A_{2^{k-1},j2^{k-1}}$ will broadcast to the processors storing the areas $A_{i,j2^{k-1}+1}$, $A_{i,j2^{k-1}+2}$, \ldots, $A_{i,(j+1)2^{k-1}}$ 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
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 inherently 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

**Figure 3.** A 4 x 4 multigrid network.
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

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

Luigi Cinque¹, Sergio De Agostino¹, and Luca Lombardi²

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 grayscale 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.
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 neighborhood 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.

\[
\begin{array}{cccccccc}
255 & 255 & 255 & 254 & 254 & 110 & 110 & 110 \\
255 & 255 & 255 & 254 & 254 & 110 & 110 & 110 \\
255 & 255 & 255 & 254 & 254 & 110 & 110 & 110 \\
255 & 255 & 255 & 254 & 254 & 110 & 110 & 110 \\
255 & 255 & 254 & 128 & 127 & 128 & 129 & 130 \\
255 & 253 & 253 & 128 & 128 & 129 & 130 & 131 \\
254 & 253 & 252 & 129 & 129 & 130 & 131 & 132 \\
253 & 252 & 251 & 130 & 130 & 130 & 254 & 255 \\
\end{array}
\]

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 \( m \times m' \) 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
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 2. Image 1.

Figure 3. Image 2.
The last component 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 4.** Image 3.

**Figure 5.** Image 4.
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 be-
tween 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 011011110111111110
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 1111110. 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 1111111 1111100 0000001 1111101 1111100 1111011 00 00010. 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 effectiveness 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.

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

**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
L. Cinque et al.: Speeding up Lossless Image Compression: Experimental Results...

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

**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.

<table>
<thead>
<tr>
<th>Image</th>
<th>1 proc.</th>
<th>2 proc.</th>
<th>4 proc.</th>
<th>8 proc.</th>
<th>16 proc.</th>
<th>32 proc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>76</td>
<td>39</td>
<td>19</td>
<td>11</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>81</td>
<td>40</td>
<td>23</td>
<td>11</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>78</td>
<td>39</td>
<td>24</td>
<td>12</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>79</td>
<td>44</td>
<td>24</td>
<td>11</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>77</td>
<td>38</td>
<td>22</td>
<td>10</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Avg.</td>
<td>78.2</td>
<td>40</td>
<td>22.4</td>
<td>11</td>
<td>5.4</td>
<td>3.2</td>
</tr>
</tbody>
</table>

**Figure 9.** BLOCK MATCHING compression 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.

<table>
<thead>
<tr>
<th>Image</th>
<th>1 proc.</th>
<th>2 proc.</th>
<th>4 proc.</th>
<th>8 proc.</th>
<th>16 proc.</th>
<th>32 proc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>43</td>
<td>22</td>
<td>11</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>44</td>
<td>22</td>
<td>12</td>
<td>7</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>43</td>
<td>22</td>
<td>15</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>43</td>
<td>30</td>
<td>12</td>
<td>7</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>41</td>
<td>32</td>
<td>15</td>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Avg.</td>
<td>42.8</td>
<td>25.6</td>
<td>13</td>
<td>6.6</td>
<td>3.4</td>
<td>2</td>
</tr>
</tbody>
</table>
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 neighborhood 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

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,
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 frequen-
cies 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 esti-
mates 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.

\[
\begin{array}{c}
\begin{array}{c}
7 \quad 6 \quad 5 \quad 4 \\
\quad 3 \quad 3 \\
\end{array} \\
\begin{array}{c}
13 \quad 9 \\
\quad 2 \quad 2 \\
\end{array} \\
\begin{array}{c}
22 \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
7 \quad 5 \\
\quad 3 \quad 2 \\
\end{array} \\
\begin{array}{c}
12 \quad 10 \\
\quad 5 \quad 5 \\
\end{array} \\
\begin{array}{c}
22 \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
7 \quad 5 \\
\quad 3 \quad 3 \\
\end{array} \\
\begin{array}{c}
12 \quad 10 \\
\quad 6 \quad 5 \\
\end{array} \\
\begin{array}{c}
22 \\
\end{array}
\end{array}
\]

\textbf{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
Table 1: Alphabet sizes

<table>
<thead>
<tr>
<th></th>
<th>1-grams</th>
<th>2-grams</th>
<th>3-grams</th>
<th>4-grams</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>52</td>
<td>808</td>
<td>6026</td>
<td>21886</td>
</tr>
<tr>
<td>French</td>
<td>131</td>
<td>2965</td>
<td>18864</td>
<td>56078</td>
</tr>
</tbody>
</table>

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.

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 $s_i$, $1 \leq i \leq n$, 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:

\begin{align*}
\text{Update after having read } P_t: \\
\text{for } i & \leftarrow 1 \text{ to } n \\
\text{add frequency of } s_i \text{ within } P_t \text{ to } f(s_i) \\
\text{subtract frequency of } s_i \text{ within } P_{t-k} \text{ from } f(s_i) \\
\text{for } i & \leftarrow 1 \text{ to } n - 1 \\
\text{if } f(s_i) > f(s_{i+1}) \text{ swap } (s_i, s_{i+1}) \\
\text{Build Huffman tree for sequence } (f(s_1), f(s_2), \ldots, f(s_n)) \text{ using two queues}
\end{align*}

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 ($k \geq 0$). 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.

<table>
<thead>
<tr>
<th>Block size</th>
<th>Compression</th>
<th>Time</th>
<th>Compression</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bub-For-1</td>
<td></td>
<td>Bub-For-5</td>
<td></td>
</tr>
<tr>
<td>English</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>4.112 5.532</td>
<td>0.13</td>
<td>4.112 5.532</td>
<td>0.63</td>
</tr>
<tr>
<td>2000</td>
<td>4.114 5.532</td>
<td>0.62</td>
<td>4.114 5.532</td>
<td>0.44</td>
</tr>
<tr>
<td>10000</td>
<td>4.123 5.533</td>
<td>2.56</td>
<td>4.123 5.533</td>
<td>1.32</td>
</tr>
<tr>
<td>French</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>4.699 6.020</td>
<td>0.27</td>
<td>4.699 6.020</td>
<td>0.49</td>
</tr>
<tr>
<td>2000</td>
<td>4.700 6.021</td>
<td>0.62</td>
<td>4.700 6.021</td>
<td>1.26</td>
</tr>
<tr>
<td>10000</td>
<td>4.705 6.022</td>
<td>2.56</td>
<td>4.705 6.022</td>
<td>1.32</td>
</tr>
</tbody>
</table>

Table 2: Dynamic compression of data packets using single characters

<table>
<thead>
<tr>
<th>Block size</th>
<th>Compression</th>
<th>Time</th>
<th>Compression</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bub-For-1</td>
<td></td>
<td>Bub-For-5</td>
<td></td>
</tr>
<tr>
<td>English</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>3.805 5.061</td>
<td>30.1</td>
<td>3.805 5.061</td>
<td>286.2</td>
</tr>
<tr>
<td>2000</td>
<td>3.805 5.061</td>
<td>30.1</td>
<td>3.805 5.061</td>
<td>286.2</td>
</tr>
<tr>
<td>10000</td>
<td>3.806 5.062</td>
<td>30.1</td>
<td>3.806 5.062</td>
<td>286.2</td>
</tr>
<tr>
<td>French</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>4.109 6.343</td>
<td>286.2</td>
<td>4.109 6.343</td>
<td>286.2</td>
</tr>
<tr>
<td>10000</td>
<td>4.108 6.342</td>
<td>286.2</td>
<td>4.108 6.342</td>
<td>286.2</td>
</tr>
</tbody>
</table>

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

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 $S \rightarrow uReRw$, 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 $S \rightarrow uAwAw$ 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.
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 $\Sigma$. The number of characters in $\Sigma$ is denoted by $|\Sigma|$. A sequence $s$ of length $n$ on $\Sigma$ is represented by $s[0]s[1] \ldots s[n-1]$, where $s[i] \in \Sigma \ \forall \ 0 \leq i < n$. We denote by $s[i,j]$ (for $j \geq i$) the sequence $s[i]s[i+1] \ldots s[j]$ 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 $\Sigma$ with a lexicographic order $\prec$ extensible to $\Sigma^*$. Let $\tilde{s} = s\$, with a special character $\$ not contained in $\Sigma$, lexicographically smaller than every element of $\Sigma$.
The suffix array, denoted by $sa$, is a permutation of $[0..n]$ such that:

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

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,

$$lcp[0] = 0,$$

and $\forall \ i \in [1,n] : lcp[i] = k$ such that

$$\tilde{s}[sa[i-1].][0,k-1] = \tilde{s}[sa[i].][0,k-1] \text{ and } \tilde{s}[sa[i-1].][k] \neq \tilde{s}[sa[i].][k].$$

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 $\omega$ where the nodes correspond to a left context of $\omega$.

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

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

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

- the root is the $\omega$-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 $s[sa[k] - 1]$. We introduce the successor and predecessor notations:

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

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

One may remark that $\text{predecessor}(i)$ 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 $\omega$ not only affect the $\omega$-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 $\text{predecessor}(k_1)$ and $\text{predecessor}(k_2)$ belong to the same $av$-interval ($a \in \Sigma$). Then $\text{predecessor}(k_1) > \text{predecessor}(k_2)$.

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

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: $\Sigma_k$ will denote the alphabet in step $k$. In each of this steps, the algorithm i) finds a repeat $R_k$ in a sequence $s^{(k)}$ defined on the alphabet $\Sigma_k$ and returns a list $O_k$ of non-overlapping occurrences of $R_k$ ii) updates the sequence $s^{(k)}$ and its associated $ESA_{DL}$ replacing the given occurrences of $R_k$ by a single new character $C_k$, thus defining a new alphabet $\Sigma_{k+1} = \Sigma_k \cup \{C_k\}$. The modified sequence is then called $s^{(k+1)}$. 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 \( R_k \) occurrence; 2) move entries with respect to the new alphabetic order; and 3) update \( lcp \) array with respect to recoding occurrences of \( R_k \) 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 \( R_k \) substituted occurrences

<table>
<thead>
<tr>
<th>entry</th>
<th>( lcp )</th>
<th>suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{prev}[j] )</td>
<td>4</td>
<td>ATAC...</td>
</tr>
<tr>
<td>( j )</td>
<td>2</td>
<td>ATGA...</td>
</tr>
<tr>
<td>( \text{next}[j] )</td>
<td>3</td>
<td>ATGT...</td>
</tr>
</tbody>
</table>

Figure 1. Deletion of entry \( j \).

By replacing the word \( R_k \) 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 \( R_k \) substituted occurrence must be deleted. Thus for \( i \) in \( O_k \) and for \( \ell \) in \( [1, |R_k| - 1] \), 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 \( \text{next} \) and \( \text{prev} \) arrays to their previous and next index:

\[
\text{next}[\text{prev}[j]] \leftarrow \text{next}[j] \quad \text{and} \quad \text{prev}[\text{next}[j]] \leftarrow \text{prev}[j].
\]

Furthermore, the \( lcp \) value of the index following \( j \) (\( lcp[\text{next}[j]] \)) 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 \( \text{next}[j] \) is equal to the minimal longest common prefix value of entries \( j \) and \( \text{next}[j] \).

An example is shown in Figure 1 where the deletion of entry \( j \) affects the \( lcp[\text{next}[j]] \) 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 (\( \text{prev}[n+1] \)).

**Algorithm 1** Delete entries at step \( k \), replacing \( R_k \) by \( C_k \)

```
delete_entries(\( ESA^{(k)}_{DL}, R_k, O_k \))
1: for \( i \in O_k \) do
2: for \( \ell \in [1, |R_k| - 1] \) do
3: \( j \leftarrow \text{isa}[i \oplus \ell] \)
4: if \( \text{next}[j] \neq END \) then
5: \( lcp[\text{next}[j]] \leftarrow \min(lcp[j], lcp[\text{next}[j]]) \)
6: end if
7: \( \text{next}[\text{prev}[j]] = \text{next}[j] \)
8: \( \text{prev}[\text{next}[j]] = \text{prev}[j] \)
9: end for
10: end for
```

2.3 Move entries, with respect to new alphabetic order

After replacing the word \( R_k \) by the new character \( C_k \), some \( ESA_{DL} \) lines may be misplaced with respect to the chosen order of \( C_k \) in \( \Sigma_{k+1} \).

Entries in the \( R_k \)-interval are potentially misplaced. Moreover, for \( v \in \Sigma_k^* \), index entries inside an \( vR_k \)-interval are misplaced if the substitution of \( R_k \) into \( C_k \) 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 \( C_k \) the largest rank in the lexicographical order of the alphabet \( \Sigma_k \), i.e. \( \forall \alpha \in \Sigma_k : \alpha \prec C_k \).

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

```plaintext
update_order\{ESA_DL^{(k)}, \mathcal{R}_k, \mathcal{O}_k, i_{start}, depth, move\}
1: if Couple \((i_{start}, depth)\) already treated during another recursion call then
2: End procedure
3: end if
4: \( i \leftarrow i_{start} \)
5: while \( i \neq END \land lcp[\text{next}[i]] \geq depth + |\mathcal{R}_k| \) do
6: \( i \leftarrow \text{next}[i] \)
7: end while
8: \( i_{end} \leftarrow i \)
9: \( \text{minLCP} \leftarrow \min_{j \in [i_{start}, i_{end}]} lcp[j] \)
10: if \( \text{move} \) then
11: while \( i \neq END \land lcp[\text{next}[i]] \geq depth \) do
12: \( i \leftarrow \text{next}[i] \)
13: end while
14: end if
15: \( i_{dest} \leftarrow i \)
16: if \( i_{end} \neq i_{dest} \) then
17: \( lcp[\text{next}[i_{end}]] \leftarrow \min(lcp[\text{next}[i_{end}]], \text{minLCP}) \)
18: \( lcp[i_{start}] \leftarrow depth \)
19: if \( i_{start} = i_{first} \land depth \neq 0 \) then
20: \( i_{first} \leftarrow \text{next}[i_{end}] \)
21: end if
22: move_group\((i_{start}, i_{end}, i_{dest})\)
23: else
24: \( lcp[i_{start}] \leftarrow \min(lcp[i_{start}], depth) \)
25: move \leftarrow false
26: end if
27: \( i \leftarrow i_{start} \)
28: while \( i \neq \text{next}[i_{end}] \) do
29: \( \text{newdepth} \leftarrow \text{depth} + (\text{if predecessor}(i) \in \mathcal{O}_k \text{ then len else 1}) \)
30: if \( \text{move} \lor (\text{sa}[\text{prev}[\text{predecessor}(i)] > \text{newdepth} \land \text{sa}[\text{prev}[\text{predecessor}(i)]] \oplus \text{newdepth} \in \mathcal{O}_k) \) then
31: update_order\{ESA_DL^{(k)}, \mathcal{R}_k, \mathcal{O}_k, \text{predecessor}(i), \text{newdepth}, i_{dest} \neq i_{end}\}
32: end if
33: \( i \leftarrow \text{next}[i] \)
34: 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 $R_k$ 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 $vR_k$-interval works as follow:

1. detects the end position of the $vR_k$-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 $vR_k$-interval is found (lines 2, 2 and 2).

Figure 2. Moves induced by substituting $GA$ by $C_1$.

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

Figure 2 shows the ESA$_{DL}$ of sequence $GAAGAAGC$, where $R_1 = GA$ is substituted by $C_1$. 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 \([i_{\text{start}}, i_{\text{end}}]\) after the position \(i_{\text{dest}}\)

\begin{algorithm}
\begin{align*}
&\text{move\_group}(\text{ESA}_{\text{DL}}^{(k)}, \mathcal{R}_k, \mathcal{O}_k, i_{\text{start}}, i_{\text{end}}, i_{\text{dest}}) \\
&1: \text{next}[\text{prev}[i_{\text{start}}]] = \text{next}[i_{\text{end}}] \\
&2: \text{prev}[\text{next}[i_{\text{end}}]] = \text{prev}[i_{\text{start}}] \\
&3: \text{next}[i_{\text{end}}] = \text{next}[i_{\text{dest}}] \\
&4: \text{prev}[\text{next}[i_{\text{dest}}]] = i_{\text{end}} \\
&5: \text{next}[i_{\text{dest}}] = \text{start} \\
&6: \text{prev}[i_{\text{start}}] = i_{\text{dest}}
\end{align*}
\end{algorithm}

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 \(TA\).

\begin{itemize}
\item \(1\) \(\underline{C}TA\text{TAC}\ldots\)
\item \(i+1\) \(\underline{C}TAT\text{TAG}\ldots\)
\item \(i+2\) \(\underline{C}TAT\text{TA}\ldots\)
\end{itemize}

and suppose that the \(TTA\)-interval containing the index \(i_{\text{sa}}[i + 2] \oplus 3\) (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 \(\text{move}\)). 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 \(\mathcal{R}_k\) occurrences Among each \(\nu\mathcal{R}_k\)-interval, suffixes starting with \(\nu\mathcal{R}_k\) where \(\mathcal{R}_k\) is not substituted (whose position does not belong to \(\mathcal{O}_k\)) may occur. The associated entries in the \(\text{ESA}_{\text{DL}}\) should not be moved with the \(\nu\mathcal{R}_k\)-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 (\(sa[i] \oplus \text{depth} \in \mathcal{O}_k\)). Once detected a non-selected occurrence, we move it to the beginning of the group (before \(i_{\text{start}}\)). As previously mentioned, this also involves modifications of the \(lcp\) array for maintaining its consistency.

2.4 Update lcp values after the substitution of \(\mathcal{R}_k\) 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 $R_k$. 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 ($R_k$-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 $lcp(\tilde{s}[sa[i]..], \tilde{s}[sa[j]..]) = \min_{\ell \in \text{next}[\text{successor}(i)], \text{successor}(j)} lcp[\ell]$

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)
1: lcp[i] ← 0
2: if prev[i] ≥ 0 then
3:   i ← sa[i]
4:   j ← sa[prev[i]]
5: while $i < n \land j < n \land s[i] = s[j]$ do
6:   i ← i ⊕ 1
7:   j ← j ⊕ 1
8:   lcp[i] ← lcp[i] + 1
9: end while
10: 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_{DL}^{(k)}, R_k, O_k)
1: q ← queue()
2: for $i \in O_k$ do
3:   recalculate_lcp(ESA_{DL}^{(k)}, isa[i])
4:   q.push((predecessor(isa[i]), 1))
5: end for
6: while not q.empty() do
7:   $(i, depth) ← q.pop$
8:   q.pop
9:   if $i ≥ 0 \land lcp[i]$ not already updated $\land lcp[i] ≥ depth$ then
10:      lcp[i] ← $\min_{j \in \text{next}[\text{successor}(prev[i])], \text{successor}(i)} lcp[j] + 1$
11:      q.push((predecessor(i), depth + 1))
12: end if
13: 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 $(i, \text{depth})$ 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 brevity, 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 nonlinear 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,
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 \textit{update}), and time spent in building \(ESA\) from scratch at each iteration with the linear algorithm from Kärkkäinen and Sanders (column \textit{K & S}) and the algorithm from Larsson and Sadakane (column \textit{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 (\texttt{fields.c}, \texttt{grammar.lsp} and \texttt{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 \texttt{Spor\_All\_2x\_fasta} files (an artificial file obtained by concatenating \texttt{Spor\_All\_fasta} with itself) from the Purdue corpus, and the \texttt{ptt5} 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 \(|O_k| - 1) \times (|w| - 1) - 1\), 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 simultaneous 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

<table>
<thead>
<tr>
<th>sequence</th>
<th>length (chars.)</th>
<th>random</th>
<th>maximal length</th>
<th>maximal compression</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sequence</td>
<td>update</td>
<td>K &amp; S</td>
<td>L &amp; S</td>
</tr>
<tr>
<td>CANTERBURY CORPUS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>alicex29.txt</td>
<td>152089</td>
<td>163</td>
<td>2812</td>
<td>1497</td>
</tr>
<tr>
<td>asyoulik.txt</td>
<td>125179</td>
<td>131</td>
<td>2111</td>
<td>1109</td>
</tr>
<tr>
<td>cp.html</td>
<td>24603</td>
<td>15</td>
<td>132</td>
<td>95</td>
</tr>
<tr>
<td>fields.c</td>
<td>11150</td>
<td>6</td>
<td>38</td>
<td>31</td>
</tr>
<tr>
<td>grammar.lisp</td>
<td>3721</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>kennedy.xls</td>
<td>1029744</td>
<td>1323</td>
<td>34905</td>
<td>12829</td>
</tr>
<tr>
<td>lect10.txt</td>
<td>426754</td>
<td>516</td>
<td>17151</td>
<td>6871</td>
</tr>
<tr>
<td>plrabn12.txt</td>
<td>48161</td>
<td>588</td>
<td>22657</td>
<td>8853</td>
</tr>
<tr>
<td>pt5</td>
<td>513216</td>
<td>1248</td>
<td>7389</td>
<td>4617</td>
</tr>
<tr>
<td>sum</td>
<td>38240</td>
<td>42</td>
<td>234</td>
<td>151</td>
</tr>
<tr>
<td>xargs.1</td>
<td>4227</td>
<td>6</td>
<td>25</td>
<td>9</td>
</tr>
<tr>
<td>LARGE CORPUS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bible.txt</td>
<td>4047392</td>
<td>5055</td>
<td>337725</td>
<td>115481</td>
</tr>
<tr>
<td>E.coli</td>
<td>4638690</td>
<td>5534</td>
<td>382636</td>
<td>151405</td>
</tr>
<tr>
<td>world192.txt</td>
<td>2473400</td>
<td>3084</td>
<td>206643</td>
<td>67079</td>
</tr>
<tr>
<td>PURDUE CORPUS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>All_Up,1M.fasta</td>
<td>1001002</td>
<td>1238</td>
<td>61657</td>
<td>24597</td>
</tr>
<tr>
<td>All_Up,400k.fasta</td>
<td>399615</td>
<td>501</td>
<td>13959</td>
<td>6777</td>
</tr>
<tr>
<td>Helden_All.fasta</td>
<td>112507</td>
<td>119</td>
<td>1511</td>
<td>963</td>
</tr>
<tr>
<td>Helden_CGN.fasta</td>
<td>32871</td>
<td>31</td>
<td>244</td>
<td>172</td>
</tr>
<tr>
<td>Spor_All_2x.fasta</td>
<td>444906</td>
<td>112</td>
<td>82</td>
<td>94</td>
</tr>
<tr>
<td>Spor_All.fasta</td>
<td>222453</td>
<td>246</td>
<td>36588</td>
<td>2107</td>
</tr>
<tr>
<td>Spor_EarlyI.fasta</td>
<td>31039</td>
<td>34</td>
<td>187</td>
<td>152</td>
</tr>
<tr>
<td>Spor_EarlyII.fasta</td>
<td>25008</td>
<td>20</td>
<td>145</td>
<td>151</td>
</tr>
<tr>
<td>Spor_Middle.fasta</td>
<td>54325</td>
<td>S 51</td>
<td>526</td>
<td>351</td>
</tr>
</tbody>
</table>

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, \( \Sigma \), 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 \( \Sigma \). 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 13.88 bytes for the regular VST, and 12.05\( n \) 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.
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 $\Sigma$, 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 $\eta_{occ}$ 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 $\Sigma$, the virtual suffix tree, including the suffix link, can be constructed in $O(n)$ time, and $O(n)$ space, independent of $\Sigma$.

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 $\Sigma$. Let $T = \alpha\beta\gamma$, for some strings $\alpha$, $\beta$, and $\gamma$ ($\alpha$ and $\gamma$ could be empty). The string $\beta$ is called a substring of $T$, $\alpha$ is called a prefix of $T$, while $\gamma$ is called a suffix of $T$. The prefix $\alpha$ is called a proper prefix of $T$ if $\alpha \neq T$. Similarly, the suffix $\gamma$ 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 $\text{label}(u, v)$) 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 $L(v) = a$. A pointer from node $u$ to node $v$ is called a suffix link. If $\alpha$ 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 $|Σ| ≤ 256$).

3.1 Example VST

We use an example sequence to explain the above definitions. The suffix tree and VST for the string missississippi$ is shown in Figure 1. Note that the string missississippi$ 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.

![Figure 1](image_url)

**Figure 1.** Suffix tree and virtual suffix tree for the string $T = \text{missississippi}$. (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.
The Virtual Suffix Tree: An Efficient Data Structure for Suffix Trees

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 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

<table>
<thead>
<tr>
<th>node</th>
<th>root</th>
<th>$N_1$</th>
<th>$N_2$</th>
<th>$N_3$</th>
<th>$N_4$</th>
<th>$N_5$</th>
<th>$N_6$</th>
<th>$N_7$</th>
<th>$N_8$</th>
<th>$N_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>sa_index</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>12</td>
<td>14</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>fchild</td>
<td>$N_1$</td>
<td>$N_5$</td>
<td>$N_8$</td>
<td>$N_7$</td>
<td>$N_6$</td>
<td>$N_9$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>elength</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>nfleaf</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>nnleaf</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 1.** VST node attributes for the example sequence $T = \text{mississippi}\$ used in Figure 1.
**Figure 2.** Example VST (solid nodes) showing left SA index (ISA) 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 $N_5 - N_1.nnleaf = N_5 - 1 = N_4$.

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 $\Sigma$, 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 $\eta_{occ}$ 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 $\text{lsa}_\text{index}$ and $\text{rsa}_\text{index}$ as bounds) to determine
all the $\eta_{occ}$ occurrences of the pattern. Reporting each occurrence can be done in
constant time, or an additional $\eta_{occ}$ 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 $\text{sa}_\text{index}$ and
$\text{elength}$ in the basic structure are now combined into one attribute called the adjusted
SA index ($\text{asa}_\text{index}$). 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 \texttt{asa\_index}, the adjusted SA index. This is done by combining the original \texttt{sa\_index} with \texttt{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 \texttt{asa\_index}, our calculations will still use the original \texttt{sa\_index}. However, this can be derived from \texttt{asa\_index} in constant time. In fact, we can observe that in practice, we need to compute the \texttt{asa\_index} for only the leftmost child node at each node-level, while keeping the original \texttt{sa\_index} for all other nodes. To determine the \texttt{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.

<table>
<thead>
<tr>
<th>NodeName</th>
<th>root</th>
<th>(N_1)</th>
<th>(N_2)</th>
<th>(N_3)</th>
<th>(N_4)</th>
<th>(N_5)</th>
<th>(N_6)</th>
<th>(N_7)</th>
<th>(N_8)</th>
<th>(N_9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sa_index</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td>9</td>
<td>12</td>
<td>4</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>fchild</td>
<td>(N_1)</td>
<td>(N_4)</td>
<td>(N_5)</td>
<td>(N_7)</td>
<td>(N_8)</td>
<td>(N_9)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>new_elength</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>nfleaf</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>nnleaf</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>asa_index</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td>9</td>
<td>12</td>
<td>4+3=7</td>
<td>10</td>
<td>13+2=15</td>
</tr>
</tbody>
</table>

**Table 2.** Node attributes in the improved VST for the example sequence, \(T = \text{mississippi}\$$. We have included \texttt{new\_elength}, so one can compare with \texttt{elength} in Table 1. However, in practice this will not be stored in the VST.

**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 ($f_{child}$, $n_{leaf}$ and $n_{nleaf}$) 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 $f_{child}$ and

![Figure 4](image-url)
\textbf{nleaf} in the leaf nodes of the VST are no longer required. We make the \texttt{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.

\subsection*{4.4 Complexity Analysis}

\textbf{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 \texttt{pTop} and \texttt{pBottom} range from 1 to $2n$. Thus the cost for the traversals is $O(n)$. Lines 18-27 in the algorithm run at most \texttt{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.

\textbf{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 \textit{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 \texttt{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:

\textbf{Lemma 8.} Given a string $T=T[1..n]$, with symbols from an alphabet $\Sigma$, the virtual suffix tree (without the suffix link) can be constructed in $O(n)$ time, and $O(n)$ space, independent of $\Sigma$. 
Table 3. Branching factor and maximum space requirement for various sample files.

| 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  |                              |

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
$SA[w.sa_index] = SA[u.sa_index] + 1$. 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: $ISA[i] = j \text{ if } SA[j] = i$, $(i, j = 1, 2, ..., n)$. 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 $u.fchild.slink$, such that $|label(u, u.fchild.slink)| = |label(w, u.fchild.slink)|$. 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)
1 SA ← COMPUTE-SUFFIXARRAY(T, n)
2 ST ← SUFFIXTREE-FROM-SUFFIXARRAY(SA)
3 ST ← ADJUST-EDGELengths(ST)
4 Initialize VST[], Q[], pTop=0, pBottom=0, curNode=root, Q[pTop]=root
5 while (pBottom >= pTop)
6   for (each childnode in curNode) do
7     if (childnode is internal node in ST ) then
8       pBottom ← pBottom + 1; Q[pBottom] ← childNode
9     if childnode is first internal node then
10       VST[pTop].fchild ← pBottom
11     end if
12     else
13       Update VST[pTop].nleaf and VST[pBottom].nleaf
14     end if
15   end for
16   pTop ← pTop + 1; curNode ← Q[pTop]
17 end while
18 for (pb ← pBottom down to 0) do
19   if (VST[pb] is leaf node) then
20     VST[pb].asa_index ← Q[pb].fchild
21   else if (Q[pb].elength=1) then
22     VST[pb].asa_index←VST[pb].fchild.asa_index + VST[pb].nleaf - Q[pb].elength
23   else
25   end if
26 end if
27 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**

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

18 for (pb ← pBottom down to 0) do
19     if (VST[pb] is leaf node) then
20         Update array M to map SA index and node VST[pb]
21     end if
26 end for
28 for (pb ← pBottom down to 0) do
29     if (VST[pb] is leaf node) then
30         VST[pb].slink ← M[ISA[VST[pb].sa_index+1]]
31     else
32         Find ancestor $w$ of VST[pb].fchild.slink s.t.
33         $|\text{label}(w, VST[pb].fchild.slink)| = |\text{label}(VST[pb], VST[pb].fchild)|$
34         Set VST[pb].slink ← $w$
35     end if
35 end for
```

Figure 5 shows the result of the suffix link algorithm when applied to the VST of our example string $T = \text{missississippi}$. 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](image-url)

**Figure 5.** Suffix link on the VST for the sample string $T = \text{missississippi}$.  

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 \(0.7n \times 4 = 2.8n\) 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

<table>
<thead>
<tr>
<th>Ratio</th>
<th>Compact</th>
<th>Regular</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst Case</td>
<td>15.50n</td>
<td>18.00n</td>
</tr>
<tr>
<td>Average Case</td>
<td>12.63n</td>
<td>14.50n</td>
</tr>
<tr>
<td>0.75</td>
<td>12.05n</td>
<td>13.80n</td>
</tr>
<tr>
<td>0.65</td>
<td>11.48n</td>
<td>13.10n</td>
</tr>
<tr>
<td>0.6</td>
<td>10.90n</td>
<td>12.40n</td>
</tr>
</tbody>
</table>

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 \(\Sigma\), the virtual suffix tree, including the suffix link, can be constructed in \(O(n)\) time and \(O(n)\) space, independent of \(\Sigma\).

**Proof.** The theorem follows directly from Lemma 8 and Lemma 9. \(\square\)

### 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 \(\Sigma\). 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 11.5\(n\) bytes without suffix links, and 15.5\(n\) 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 9.25\(n\) bytes on average without the suffix links, or 12.05\(n\) 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

Parameterized Suffix Arrays for Binary Strings

Satoshi Deguchi\textsuperscript{1}, Fumihito Higashijima\textsuperscript{1},
Hideo Bannai\textsuperscript{1}, Shunsuke Inenaga\textsuperscript{2}, and Masayuki Takeda\textsuperscript{1}

\textsuperscript{1}Department of Informatics, Kyushu University
\textsuperscript{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 $\Pi$ is the set of parameter symbols and $\Sigma$ 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 $\Sigma$, namely, it maps any $a \in \Sigma$ to $a$ itself, while symbols of $\Pi$ 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
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 $\Pi$, 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 $O(m \log(\min\{m, \pi\}))$ preprocessing time and $O(n \log(\min\{m, \pi\}))$ scanning time. They also stated that it suffices to consider strings over $\Pi$ 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 $\Pi$.

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 $\Sigma$ and $\Pi$ 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 \(\Pi = \{a, b, c\}, \Sigma = \{X, Y\}\), \(s = \text{abaXabY}\) and \(t = \text{bcbXbcY}\).
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 \([i, j] = \{i, i + 1, \ldots, j\} \subseteq \mathcal{N}\).

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

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

In the running example, \(pv(s) = 002X24Y\) with \(s = \text{abaXabY}\).

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

\[
\begin{align*}
\quad pv(s[i : j]) = v[1 : j - i + 1],
\end{align*}
\]

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 \(pv(s) = pv(t) = 002X24Y\).

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

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

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

Here, \(\infty\) denotes a value for which \(i < \infty\) for any \(i \in \mathcal{N}\). \(^1\)

\(^1\) In practice, \(n\) can be used in place of \(\infty\) as long as we are considering a single p-string of length
\(n\), and its substrings.
In the running example, $fw(s) = 242X\infty\infty Y$ with $s = abaXabY$.

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 $\pi$ 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 $\Pi$ 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 $\Pi$.

Hence, in the remainder of the paper, we consider only p-strings in $\Pi^*$. 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 $x, y \in \mathbb{N} \cup \{\infty\}$, $x \preceq_+ y \iff x \leq y$ 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 $x, y \in ([i, j] \cup \{\infty\})^*$,

$$x \preceq y \iff \exists \alpha, u, v \in ([i, j] \cup \{\infty\})^*, a, b \in [i, j] \cup \{\infty\},$$

such that $a < b$, $x = \alpha au$, $y = \alpha bv$. 

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

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

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 \(s \in \Pi^*\) 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 \(\{pv(s[i : n]) \mid 1 \leq i \leq n\}\) 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 \(\text{short} : N^* \rightarrow N^*\) to be the function such that for any string \(x \in N^n\) of length \(n\), \(\text{short}(x) = y\) where, for \(1 \leq i \leq n\),

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

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

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

\[
PLCP_s[i] = \begin{cases} 
-1 & \text{if } i = 1, \\
\lcp(pv(s[\Pi i - 1]:n), pv([s[\Pi i]:n])) & \text{if } 2 \leq i \leq n.
\end{cases}
\]

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 \( \pi \) [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 \( PLCP_s \) 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 \( s = ababaaaaabba \) and corresponding suffixes of \( fw(s) \) and \( LCP_{fw(s)} \).

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

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 \( y - i = pv(s)[y] = y - \max\{ j \mid s[y] = s[j], 1 \leq j \leq y \} \leq y - x \), which is a contradiction. Similar arguments hold for \( fw(s) \).

Lemma 20. For any \( p \)-strings \( s,t \in \Pi^* \) with \( \pi = 2 \), \( pv(s) \preceq_i pv(t) \) if and only if \( fw(s) \leq_i fw(t) \).
Figure 1. The p-suffix array \( PSA \) of \( s = \text{abaabaaabba} \) 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.

\[ \Rightarrow \] Assume \( pv(s) \prec_+ pv(t) \). If \( pv(s) \) is a prefix of \( pv(t) \), then \( fw(s) \prec_\text{'} fw(t) \) since otherwise the proof is finished.

\[ \Leftrightarrow \] Assume \( fw(s)[i] = fw(t[1 : |s|])[i] = \begin{cases} \, \, \, 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 \( i = \min \{ j \mid pv(s)[j] \prec_+ pv(t)[j] \} \), \( \ell = pv(t)[i] \) 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 \( fw(s) \prec_\text{'} fw(t) \).

Now we consider the case where \( i - \ell > 1 \). For any \( 1 \leq h < i - \ell \), we have

\[ fw(s)[h] \leq \begin{cases} \, \, \, i - \ell - h \\ i - \ell + 1 - h \end{cases} \]

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 \( fw(s) \prec_\text{'} fw(t) \).

\[ \Leftarrow \] Assume \( fw(s) \prec_\text{'} fw(t) \). 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 \( l = lcp(fw(s), fw(t)) < \min \{ |s|, |t| \} \). 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 \( 1 < l < k < l + 1 + q \), and \( s[l + 1] \neq s[k] \) for
any $l+1 < k \leq \min\{|s|, l+1+q\}$. 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 $l+2 < k \leq \min\{|s|, l+1+q\}$, 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 $\pi = 2$,

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

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

**Proof.** By similar arguments as in (⇐=) 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 $|P_{occ}|$. 
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


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.
2 Preliminaries

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

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:

$$M1q, M1d, M2q, M2d, M3q, M3d.$$ (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.

Table 1. First example of the nontransitivity effect

<table>
<thead>
<tr>
<th>Index</th>
<th>1 2 3 4 5 6 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>··· a b b a b b ···</td>
</tr>
<tr>
<td>p</td>
<td>a * * b a * a</td>
</tr>
<tr>
<td>1st Shift</td>
<td>a * * b a ···</td>
</tr>
<tr>
<td>2nd Shift</td>
<td>a * * ···</td>
</tr>
<tr>
<td>3rd Shift</td>
<td>a * ···</td>
</tr>
</tbody>
</table>

Table 2. Second example of the nontransitivity effect

<table>
<thead>
<tr>
<th>Index</th>
<th>1 2 3 4 5 6 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>··· a b a * a * a ···</td>
</tr>
<tr>
<td>p</td>
<td>a b a a a b b</td>
</tr>
<tr>
<td>Wrong Shift</td>
<td>a b a ···</td>
</tr>
<tr>
<td>Correct Shift</td>
<td>a b a a a ···</td>
</tr>
</tbody>
</table>

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 paradigm 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 \( x = x[1..n] \):

\[
i' \leftarrow m; m' \leftarrow m - 1; \\
\text{while } i' \leq n \text{ do} \\
\quad \text{Sunday-Shift; } \\
\quad \quad \text{— After Sunday-Shift exits, perform ShiftAnd-Match } \\
\quad i \leftarrow i' - m'; \\
\quad \text{ShiftAnd-Match; } \\
\quad \quad \text{— After ShiftAnd-Match exits, shift pattern right } \\
\quad i' \leftarrow i + m';
\]

**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 \( \Delta \) 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 \( \lambda \) occurs; if \( \lambda \) 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 \(\Delta\) to align \(x[i+1] = a\) with the rightmost occurrence of \(a\) in \(p\). Since there can be no occurrence in between (otherwise \(\Delta\) 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.

<table>
<thead>
<tr>
<th>(m)</th>
<th>(\Sigma)</th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

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.

\[
\begin{align*}
\text{for } i & = 1 \text{ to } |\Delta| \\
\Delta[i] & = m + 1 \\
\text{for } j & = 1 \text{ to } |\Sigma| \\
\text{if } & \text{MATCH}(p[i], \Sigma[j]) \text{ then } \Delta[p[i]] = i
\end{align*}
\]

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 $|\Sigma|$
if $\text{MATCH}(p[i], \Sigma[j])$ 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 $\Sigma'$ 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 $\text{MATCH}(p[m], x[i'])$
do
  $i' \leftarrow i' + \Delta[x[i'] + 1]$
if $i' > n$ then return

Figure 5. Sunday-Shift

$D \leftarrow 0$
repeat
  — Here and throughout this paper operator $\ll$ means shifting $D$ one position
  — towards the most significant bit and bring a 1 to the least significant bit
  $D \leftarrow (D \ll 1) \& S[x[i]]$
  if $D \& 10^m \neq 0$ then output $i$
  $i \leftarrow i + 1$
  — 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 $O\left(\frac{mn}{w}\right)$, 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:

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

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).
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
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.

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

Table 6. Details of the pattern sets used

<table>
<thead>
<tr>
<th>BMS</th>
<th>ShiftAnd</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1600</td>
<td>14390</td>
<td>1430</td>
</tr>
</tbody>
</table>

Table 7. Average microseconds/million letters in Figure 8

**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

<table>
<thead>
<tr>
<th>File Name</th>
<th>Length(bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNA0.fasta</td>
<td>40302</td>
</tr>
<tr>
<td>DNA1.fasta</td>
<td>129145</td>
</tr>
<tr>
<td>DNA2.fasta</td>
<td>282348</td>
</tr>
<tr>
<td>DNA3.fasta</td>
<td>411493</td>
</tr>
<tr>
<td>DNA4.fasta</td>
<td>798564</td>
</tr>
<tr>
<td>DNA5.fasta</td>
<td>927709</td>
</tr>
<tr>
<td>DNA6.fasta</td>
<td>1430159</td>
</tr>
<tr>
<td>DNA7.fasta</td>
<td>2228723</td>
</tr>
<tr>
<td>DNA8.fasta</td>
<td>3518496</td>
</tr>
<tr>
<td>DNA9.fasta</td>
<td>7618319</td>
</tr>
</tbody>
</table>

Table 9. Lengths of DNA text files
Table 10. Average microseconds/million letters in Figure 10

<table>
<thead>
<tr>
<th>Tests</th>
<th>Algorithms</th>
<th>BMS</th>
<th>ShiftAnd</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Length</td>
<td>990</td>
<td>4550</td>
<td>1060</td>
<td></td>
</tr>
<tr>
<td>Pattern Length</td>
<td>1600</td>
<td>14390</td>
<td>1430</td>
<td></td>
</tr>
<tr>
<td>Number of Indeterminate Letters</td>
<td>5220</td>
<td>4651</td>
<td>4841</td>
<td></td>
</tr>
<tr>
<td>DNA file</td>
<td>4531</td>
<td>4297</td>
<td>4531</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>12341</td>
<td>27888</td>
<td>11862</td>
<td></td>
</tr>
</tbody>
</table>

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


A  An Example of ShiftAnd-Sunday Algorithm

\[
\begin{align*}
  \text{x} & \quad a \quad a \quad a \quad b \quad c \quad c \quad c \ldots \\
  \rho & \quad a \quad b
\end{align*}
\]

Figure 11. Starting position

\[
\begin{align*}
  \text{x} & \quad a \quad a \quad a \quad b \quad c \quad c \quad c \ldots \\
  \rho & \quad a \quad b
\end{align*}
\]

Figure 12. After one step in Sunday-Shift

\[
\begin{align*}
  \text{D} & \quad 0 \quad 0 \quad 0 \\
  \text{t>>D} & \quad 1 \quad 0 \quad 0 \\
  \text{S\{j\}} & \quad 1 \quad 1 \quad 0 \\
  \text{S\{j\} \& \{t>>D\}} & \quad 1 \quad 0 \quad 0
\end{align*}
\]

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 \(T = T[1]T[2] \ldots T[n]\), where \(T[i] \subseteq \Sigma\) for each \(i\), and \(\Sigma\) 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 \(\kappa\). 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 musical 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, cover s of a superstring of \(x\). The notion of seeds 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 \(x\).

In this work, we study and design algorithms for these string regularities in indeterminate strings. An indeterminate string is a sequence \(T = T[1]T[2] \ldots T[n]\), where \(T[i] \subseteq \Sigma\) for each \(i\), and \(\Sigma\) is a given alphabet of potentially large size. The simplest form of indeterminate string is one in which indeterminate positions can contain only...
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 $x = uvw$, where $u, v \in Σ^*$. We denote by $x[i \ldots j]$ 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 $y \in Σ^*$. Similarly, w is a suffix of x if $x = yw$, for $y \in Σ^*$.

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 aabedef. 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 $x = x[1 \ldots n]$ and $y = y[1 \ldots m]$ such that $x[n-i+1 \ldots n] = y[1 \ldots i]$ for some $i \geq 1$ (that is, such that x has a suffix equal to a prefix of y), the string $x[1 \ldots n]y[i + 1 \ldots m]$ is said to be a superposition of x.