Proceedings of the Prague Stringology Conference 2013

Edited by Jan Holub and Jan Žd'árek

September 2013

PSC

Prague Stringology Club

`http://www.stringology.org/`

Preface

The proceedings in your hands contains the papers presented in the Prague Stringology Conference 2013 (PSC 2013) at the Czech Technical University in Prague, which organizes the event. The conference was held on September 2–4, 2013 and it 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. Sixteen papers were selected, based on originality and quality, as regular papers for presentations at the conference. This volume contains not only these selected papers but also an abstract of one invited talk "Trees and Pushdown Automata".

The Prague Stringology Conference has a long tradition. PSC 2013 is the eighteenth 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, 2008–2012 preceded this conference. The proceedings of these workshops and conferences have been published by the Czech Technical University in Prague and are available on web pages of the Prague Stringology Club. Selected contributions were published in special issues of journals the Kybernetika, the Nordic Journal of Computing, the Journal of Automata, Languages and Combinatorics, and the International Journal of Foundations of Computer Science.

The Prague Stringology Club was founded in 1996 as a research group in 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 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. As a recognition of the conference, Elsevier B.V. decided to index the conference proceedings by Scopus collection. The main product derived from this collection is Scopus.com.

I would like to thank all those who had submitted papers for PSC 2013 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 2013. 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 September 2013

Jan Holub and William F. Smyth

Conference Organisation

Program Committee

Amihood Amir	(Bar-Ilan University, Israel)
Gabriela Andrejková	(P. J. Šafárik University, Slovakia)
Maxime Crochemore	(King's College London, United Kingdom)
Simone Faro	(Università di Catania, Italy)
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)
Shunsuke Inenaga	(Kyushu University, Japan)
Shmuel T. Klein	(Bar-Ilan University, Israel)
Thierry Lecroq	(Université de Rouen, France)
Bořivoj Melichar, *Honorary chair*	(Czech Technical University in Prague, Czech Republic)
Yoan J. Pinzón	(Universidad Nacional de Colombia, Colombia)
Marie-France Sagot	(INRIA Rhône-Alpes, France)
William F. Smyth, *Co-chair*	(McMaster University, Canada)
Bruce W. Watson	(FASTAR Group (Stellenbosch University and University of Pretoria, South Africa))
Jan Žďárek	(Czech Technical University in Prague, Czech Republic)

Organising Committee

Miroslav Balík, *Co-chair*	Jan Janoušek	Ladislav Vagner
Jan Holub, *Co-chair*	Bořivoj Melichar	Jan Žďárek

External Referees

Carl Barton	Arnaud Lefebvre	Yiannis Siantos
Loek Cleophas	Yuto Nakashima	Tinus Strauss
Gabriele Fici	Takaaki Nishimoto	Shiho Sugimoto
Derrick Kourie	Gustavo Sacomoto	

Table of Contents

Graphs and Automata[⋆]
Extended abstract

Bořivoj Melichar

Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University
Thakurova 9, 160 00 Prague 6, Czech Republic
`melichar@fit.cvut.cz`

The well known Chomsky classification concerns the classification of grammars and automata. It is sketched in the following table.

Type of grammars	Type of automata
regular grammars	finite automata
context-free grammars	pushdown automata
context-sensitive grammars	linear bounded automata
unrestricted grammars	Turing machines

Algorithm processing some nonlinear structure (tree, matrix, n-dimensional array, graph, ...), contains, in some form, a statement like this:

Do traversing the structure and perform the following operations...

Such statement leads to a linearisation of the structure in question. There is a possibility to divide such process into two parts:

1. Creating a linear notation of the structure.
2. Processing the linear notation of the structure.

The question which can be asked is about properties of linear notations of some type of structures. Some of properties of such linear notations can be used for design of algorithms for their processing. The following table shows types of automata suitable as models for processing linear notations of different types of graphs.

Type of graphs	Type of automata	Discipline
"linear" graphs	finite automata	stringology
trees	pushdown automata	arbology
directed acyclic graphs	linear bounded automata	dagology
general graphs	Turing machines	?

"Linear" graphs are representations of texts. The use of finite automata for this case has been described in many publications [3].

Linear notations of trees are context-free languages. Therefore, pushdown automata can serve as good models for algorithms in arbology [1,2]. Next examples show how to transfer the knowledge from stringology to arbology.

Example 1. Pattern matching

Given string $t = a2a2a0a1a0a1a0$. The nondeterministic finite automaton for matching the string t has the transition diagram depicted in Figure 1.

⋆ This research has been supported by the Czech Science Foundation (GAČR) as project No. 13-03253S.

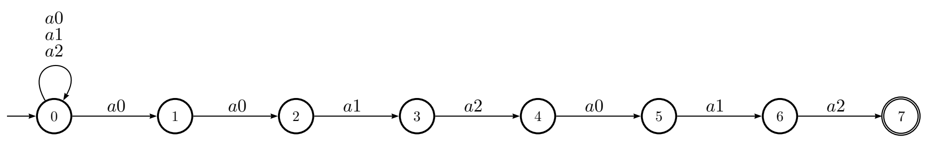
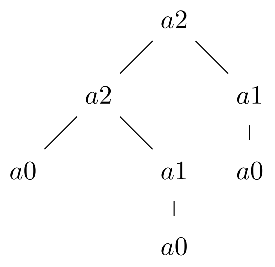


Figure 1. Transition diagram of nondeterministic finite automaton for string t from Example 1

Example 2.
String t from Example 1 is in fact the prefix notation of the tree depited in Figure 2. The transition diagram of nondeterministic subtree matching automaton is shown in Figure 3.



$$pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$$

Figure 2. Tree t from Example 2 and its prefix notation

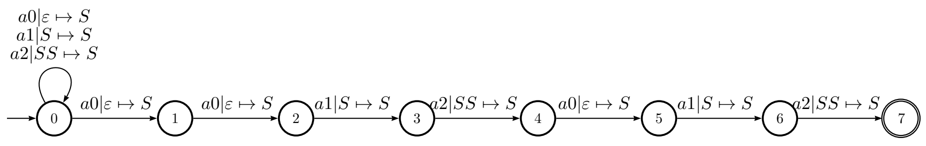


Figure 3. Transition diagram of the pushdown automaton from Example 2

Example 3.
It can be seen that shapes of transition diagrams of both automata, finite and pushdown ones, in Figures 1 and 3 are the same. They differ by pushdown operations contained in pushdown automata. The pushdown atomaton from Figure 3 is input-driven and therefore it can be determinised in the same way as finite automaton. Figure 4 shows transition diagrams of both, determistic finite and deterministic pushdown automata.

Next example shows the similarity of shapes of factor automaton and subtree pushdown automaton accepting all factors of a string and all subtrees of a tree, respectively.

Example 4. Factor and subtree automata
Figure 5 shows transition diagrams of nondeterministic factor automaton and nondeterministic subtree automaton for string $t = a2a2a0a1a0a1a0$ which is the prefix notation of tree from Figure 2.

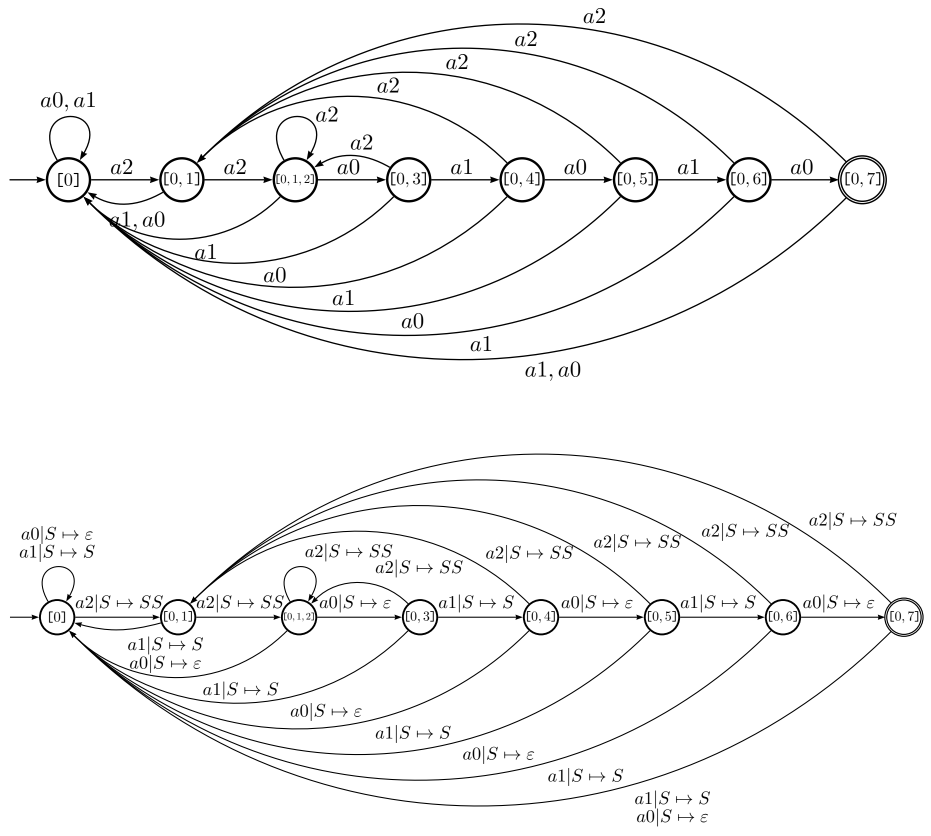


Figure 4. Transition diagrams of deterministic finite and pushdown automata from Example 2

Transition diagrams of deterministic factor and deterministic subtree pushdown automata are depicted in Figures 6 and 7.

There is a difference between these two deterministic automata. The factor automaton accepts all factors without any additional conditions. On the other hand, the subtree automaton accepts linear notations of subtrees only. Therefore transitions from states $[3, 5, 7]$ and $[5, 7]$ to states $[4, 6]$ and $[6]$ for input symbol $a1$, respectively, are omitted in the subtree automaton. The reason is that linear notations of subtrees are either $a0$ or $a1a0$ in these cases. If they are exteded by any symbol, they are no more linear notations of subtrees. The end of linear notation of a subtree is found using pushdown automaton by empty pushdown store.

Next example shows how to represent a matrix as an acyclic directed graph. Moreover, a linear representation of of this graph is shown which can be processed by linear bounded automaton. The principle can be easily extended to n-dimensional arrays [4].

Example 5. Representation of a matrix
A matrix is represented as a directed acyclic graph using relations to the right neighbour and lower neighbour. The next step is the construction of the spanning tree of this graph and adition of some pointers instead of missing edges.

References

1. *Arbology www pages*: Available on: `http://www.arbology.org`, July 2009.

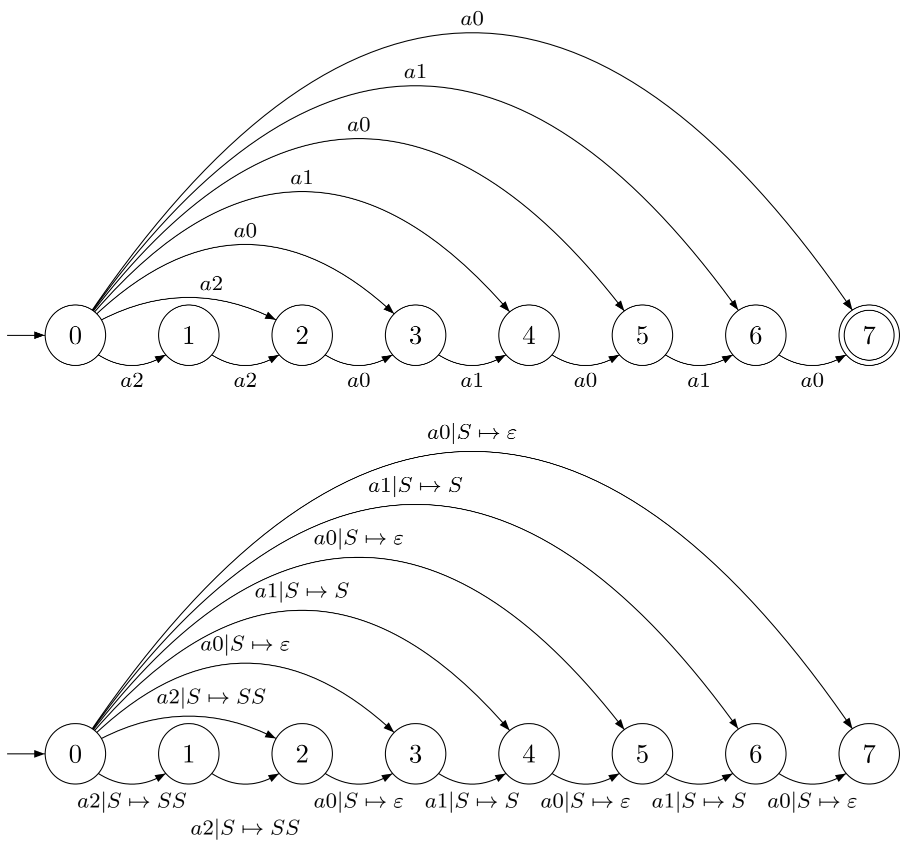


Figure 5. Transition diagram of nondeterministic factor and subtree pushdown automata for tree t in prefix notation $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 4

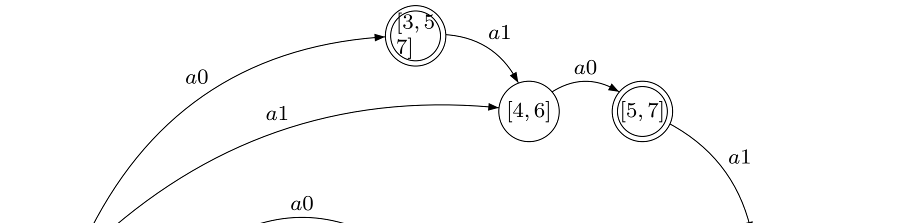


Figure 6. Transition diagram of deterministic suffix automaton for string $a2\ a2\ a0\ a1\ a0\ a1\ a0$

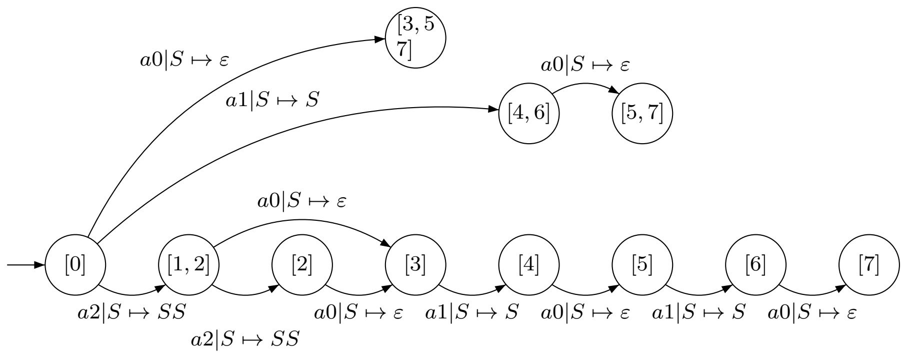

$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a0|S \mapsto \varepsilon$
[0]
[1, 2]
[2]
[3]
[4]
[5]
[6]
[7]
[3, 5 7]
[4, 6]
[5, 7]
$a2|S \mapsto SS$
$a2|S \mapsto SS$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$
$a1|S \mapsto S$
$a0|S \mapsto \varepsilon$

Figure 7. Transition diagram of deterministic subtree PDA $M_{dps}(t_1)$ for tree in prefix notation $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 5

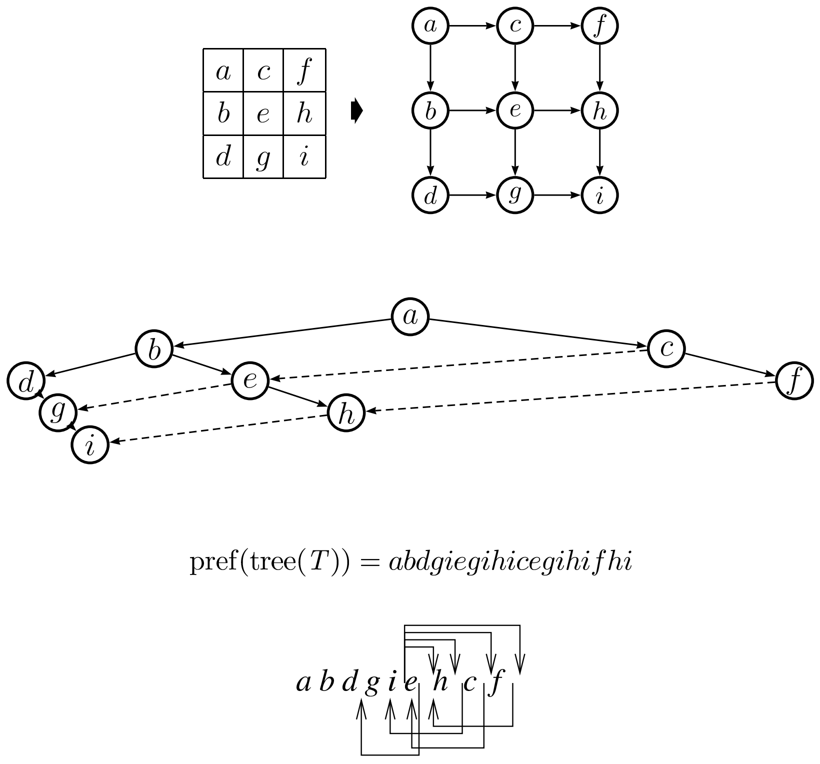

pref(tree(T)) = $abdgiegihicegihifhi$
$a\ b\ d\ g\ i\ e\ h\ c\ f$

Figure 8. Representation of a matrix: A matrix is represented as a directed acyclic graph using relations to the right neighbour and lower neighbour. The next step is the construction of the spanning tree of this graph and adition of some pointers instead of missing edges

2. B. Melichar: *Arbology: Trees and pushdown automata*, in Language and Automata Theory and Applications, A.-H. Dediu, H. Fernau, and C. Martín-Vide, eds., vol. 6031 of Lecture Notes in Computer Science, Springer-Verlag, Berlin/Heidelberg, 2010, pp. 32–49.
3. B. Melichar, J. Holub, and T. Polcar: *Text searching algorithms.* Available on: `http://www.stringology.org/athens/`, Nov. 2005.
4. J. Žďárek: *Two-dimensional Pattern Matching Using Automata Approach*, PhD thesis, Czech Technical University in Prague, 2010, Available on: `http://www.stringology.org/papers/Zdarek-PhD_thesis-2010.pdf`.

Swap Matching in Strings by Simulating Reactive Automata

Simone Faro

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

Abstract. The *pattern matching problem with swaps* consists in finding all occurrences of a pattern P in a text T, when disjoint local swaps in the pattern are allowed. In this paper we introduce a new theoretical approach to the problem based on a reactive automaton modeled after the pattern, and provide two efficient non standard simulations of the automaton, based on bit-parallelism. The first simulation can be implemented by at least 7 bitwise operations, while the second one involves only 2 bitwise operations to simulate the automaton behavior, when the input pattern satisfies particular conditions. The resulting algorithms achieve $\mathcal{O}(n)$ worst-case time complexity with patterns whose length is comparable to the word-size of the target machine.

Keywords: pattern matching with swaps, nonstandard pattern matching, combinatorial algorithms on words, bit parallelism

1 Introduction

The *string matching problem with swaps* (swap matching problem, for short) is a well-studied variant of the classic string matching problem. It consists in finding all occurrences, up to character swaps, of a pattern P of length m in a text T of length n, with P and T sequences of characters drawn from a same finite alphabet Σ of size σ. More precisely, the pattern is said to *swap-match the text at a given location* j if adjacent pattern characters can be swapped, if necessary, so as to make it identical to the substring of the text ending (or, equivalently, starting) at location j. All swaps are constrained to be disjoint, i.e., each character can be involved in at most one swap. Moreover, we make the agreement that identical adjacent characters are not allowed to be swapped.

This problem is of relevance in practical applications such as text and music retrieval, data mining, network security, and many others. Following [18], we also mention a particularly important application of the swap matching problem in biological computing, specifically in the process of translation in molecular biology, with the genetic triplets (otherwise called *codons*). In such application one wants to detect the possible positions of the start and stop codons of a mRNA in a biological sequence and find hints as to where the flanking regions are, relative to the translated mRNA region.

In the field of natural language processing the transposition of two adjacent characters in a text is a most common typing error. Thus several algorithms for the spell-checking problem are designed in order to identify swaps of characters in their matching engines.

The swap matching problem was introduced in 1995 as one of the open problems in nonstandard string matching [19]. The first nontrivial result was reported by Amir *et al.* [1], who provided a $\mathcal{O}(nm^{\frac{1}{3}} \log m)$-time algorithm in the case of alphabet sets of size 2, showing also that the case of alphabets of size exceeding 2 can be reduced to that of size 2 with a $\mathcal{O}(\log^2 \sigma)$-time overhead, subsequently reduced to $\mathcal{O}(\log \sigma)$ in the journal version [2]. Amir *et al.* [4] studied some rather restrictive cases in which a $\mathcal{O}(m \log^2 m)$-time algorithm can be obtained. More recently, Amir *et al.* [3] solved the swap matching problem in $\mathcal{O}(n \log m \log \sigma)$-time. The above solutions are all based on the fast Fourier transform (FFT).

In 2008 the first attempt to provide an efficient solution to the swap matching problem without using the FFT technique has been presented by Iliopoulos and Rahman in [18]. They introduced a new graph-theoretic approach to model the problem and devised an efficient algorithm, based on the bit-parallelism technique [5], which runs in $\mathcal{O}((n+m)\log m)$-time, in the case of short patterns.

In 2009, Cantone and Faro [9,6] presented a new efficient algorithm, named Cross-Sampling (CS), which simulates a non-deterministic automaton with $2m$ states and $3m-2$ transitions. The CS algorithm though characterized by a $\mathcal{O}(nm)$ worst-case time complexity, admits an efficient bit-parallel implementation, named Bit-Parallel-Cross-Sampling (BPCS), which achieves $\mathcal{O}(n)$ worst-case time and $\mathcal{O}(\sigma)$ space complexity in the case of short patterns fitting in few machine words.

In this paper we present a new theoretical model to solve the swap matching problem in strings, based on reactive automata [16,13]. Specifically the automaton used in our model has only m states and at most $3m-2$ transitions. Moreover it has $8m-12$ reactive links. We propose also two different non-standard simulations of the automaton based on bit parallelism. The first approach works by encoding the transitions of the automaton and leads to an algorithm with linear worst case time complexity and $\mathcal{O}(\sigma)$-space complexity, in the case of short patterns. Our second approach uses a simpler encoding and, under suitable conditions, it turns out to be very efficient in practice, achieving $\mathcal{O}(n)$ worst case time complexity and requiring $\mathcal{O}(\sigma^2)$-extra space. However in the general case it works as an oracle and needs an extra verification phase when a candidate occurrence is found. In this case its worst case time complexity is $\mathcal{O}(nm)$.

The paper is organized as follows. In Section 2 we introduce some notions and definitions. Then in Section 3 we introduce the notion of swap reactive automaton and propose two non standard simulations of it based on bit parallelism. We draw our conclusions in Section 5.

2 Notations and Definitions

Given a string $P = p_0 p_1 \cdots p_{m-1}$ of length $m \geq 0$, we represent it as a finite array $P[0\,..\,m-1]$. In particular, for $m=0$ we obtain the empty string ε. We denote by p_i (or $P[i]$) the $(i+1)$-st character of P, for $0 \leq i < m$, and by $P[i\,..\,j]$ the substring of P contained between the $(i+1)$-st and the $(j+1)$-st characters of P, for $0 \leq i \leq j < m$. For any two strings P and P' we say that P' is a prefix of P if $P' = P[0\,..\,i-1]$, for some $0 \leq i \leq m$. We denote by P_i the nonempty prefix $P[0\,..\,i]$ of P of length $i+1$, for $0 \leq i < m$.

Definition 1. *A* swap permutation *for a string P of length m is a permutation $\pi : \{0, ..., m-1\} \to \{0, ..., m-1\}$ such that:*

(a) if $\pi(i) = j$ then $\pi(j) = i$ (characters at positions i and j are swapped);
(b) for all i, $\pi(i) \in \{i-1, i, i+1\}$ (only adjacent characters can be swapped);
(c) if $\pi(i) \neq i$ then $P[\pi(i)] \neq P[i]$ (identical characters can not be swapped).

For a given string P and a swap permutation π for P, we write $\pi(P)$ to denote the *swapped version* of P, namely $\pi(P) = P[\pi(0)] \cdot P[\pi(1)] \cdots P[\pi(m-1)]$.

Definition 2. *Given a text T of length n and a pattern P of length m, P is said to* swap-match *(or to have a* swapped occurrence*) at location $j \geq m-1$ of T if there exists a swap permutation π of P such that $\pi(P)$ matches T at location j, i.e., $\pi(P) = T[j-m+1\,..\,j]$. In such a case we write $P \propto T_j$.*

It can be proved [7] that if P has a swap occurrence at location j of the text T, then the permutation π such that $\pi(P)$ matches T at location j, is unique.

A finite automaton (FA) is a 5-tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the collection of final states, Σ is an alphabet, and $\delta \subseteq (Q \times \Sigma \times Q)$ is the transition relation.

Definition 3 (Switch Reactive Transformation). *Let $\delta \subseteq (Q \times \Sigma \times Q)$ be the transition relation of an automaton A and let $\varphi \subseteq \delta$. Let T^+, T^- be two subsets of $\delta \times \delta$. A transformation $\delta \to \delta^\varphi$, for $\varphi \subseteq \delta$, is defined as follows*

$$\begin{aligned}\delta^\varphi = &(\delta \setminus \{\gamma \mid \gamma \in \delta \text{ and } \exists\, \tau \in \varphi \text{ such that } (\tau,\gamma) \in T^-\}) \\ &\cup \{\gamma \mid \gamma \in \delta \text{ and } \exists\, \tau \in \varphi \text{ such that } (\tau,\gamma) \in T^+\}\end{aligned}$$

The reactive links are intended to be applied simultaneously.

Definition 4 (Switch Reactive Automaton). *A reactive automaton is an ordinary non-deterministic automaton with a switch reactive transformation, i.e. a triple $R = (A, T^+, T^-)$ which defines the switch reactive transformation above.*

Definition 5 (Nondeterministic Run). *Let $S = s_0 s_1 \cdots s_{n-1}$ be a word on the alphabet Σ and let $R = (A, T^+, T^-)$ be a reactive automaton, where $A = (Q, q_0, \Sigma, F, \delta)$. The nondeterministic run over S is a sequence of pairs (Q_k, δ_k), for $k = 0, \ldots, n$, with $Q_k \subseteq Q$ is the set of active states, and $\delta_k \subseteq \delta$ is the set of active transitions. It can be formally defined as follows:*

$$(Q_k, \delta_k) = \begin{cases} (\{q_0\}, \delta) & \text{if } k = 0 \\ (\{q \mid (r, s_{k-1}, q) \in \delta_{k-1} \text{ with } r \in Q_{k-1}\}, \delta^\varphi_{k-1}) & \text{if } k > 0 \end{cases}$$

where $\varphi = \{(r, s_{k-1}, q) \mid (r, s_{k-1}, q) \in \delta_{k-1}$ and $r \in Q_{k-1}\}$.

We say that the string S is accepted by the reactive automaton if the nondeterministic run $\langle (Q_0,\delta_0), (Q_1,\delta_1), \ldots, (Q_n,\delta_n)\rangle$ over S is such that $Q_n \cap F \neq \emptyset$.

Finally, we recall the notation of some bitwise infix operators on computer words, namely the bitwise `and` "&", the bitwise `or` "|", and the `left shift` "$\ll$" operator (which shifts its first argument to the left by a number of bits equal to its second argument). We say that a bit *is set* to indicate that its value is equal to 1.

3 A New Algorithm Based on Reactive Automata

Reactive automata [16,13] are used to reduce dramatically the number of states in both deterministic and the non-deterministic automata. As stated by Definition 3 and Definition 4, a reactive automaton has extra links whose role is to change the behavior of the automaton itself. In this section we present a new solution for the swap matching problem in strings based on reactive automata.

In particular, we show in Section 3.1 how to construct a reactive automaton which recognizes all swap occurrence of a given input pattern and prove its correctness. Then we give two non-standard approach to simulate such automaton in Section 3.2 and in Section 3.3.

3.1 The Swap Reactive Automaton

The automaton which we use in our solution is called *swap reactive automaton*. It is defined as follows.

Definition 6 (Swap Reactive Automaton). *Let P be a pattern of length m over an alphabet Σ. The Swap Reactive Automaton (SRA) for P is a Reactive Automaton $R = (A, T^+, T^-)$, with $A = (Q, \Sigma, \delta, q_0, F)$, such that*

- *$Q = \{q_0, q_1, \ldots, q_m\}$ is the set of states;*
- *q_0 is the initial state;*
- *$F = \{q_m\}$ is the set of final states;*
- *δ is the transition relation defined as*

$$\begin{aligned}
\delta = \{&(q_i, p_i, q_{i+1}) \mid 0 \le i < m\} \cup && \textit{no swaps}\\
\{&(q_i, p_{i+1}, q_{i+1}) \mid 0 \le i < m-1 \textit{ and } p_i \neq p_{i+1}\} \cup && \textit{start of a swap}\\
\{&(q_i, p_{i-1}, q_{i+1}) \mid 1 \le i < m \textit{ and } p_i \neq p_{i-1}\} \cup && \textit{end of a swap}\\
\{&(q_0, \Sigma, q_0)\} && \textit{self loop}
\end{aligned}$$

- *T^+ is the set of (switch on) reactive links defined as*

$$\begin{aligned}
T^+ = \{&((q_i, p_i, q_{i+1}), (q_i, p_{i-1}, q_{i+1})) \in (\delta \times \delta) \mid 0 < i < m-1, \} \cup\\
\{&((q_i, p_{i+1}, q_{i+1}), (q_i, p_{i-1}, q_{i+1})) \in (\delta \times \delta) \mid 0 < i < m-1\} \cup\\
\{&((q_i, p_{i-1}, q_{i+1}), (q_i, p_i, q_{i+1})) \in (\delta \times \delta) \mid 0 < i < m-1\} \cup\\
\{&((q_i, p_{i-1}, q_{i+1}), (q_i, p_{i+1}, q_{i+1})) \in (\delta \times \delta) \mid 0 < i < m-1\}
\end{aligned}$$

- *T^- is the set of (switch off) reactive links defined as*

$$\begin{aligned}
T^- = \{&((q_i, p_i, q_{i+1}), (q_{i+1}, p_i, q_{i+2})) \in (\delta \times \delta) \mid 0 \le i < m-1\} \cup\\
\{&((q_i, p_{i-1}, q_{i+1}), (q_{i+1}, p_i, q_{i+2})) \in (\delta \times \delta) \mid 1 \le i < m-1\} \cup\\
\{&((q_i, p_{i+1}, q_{i+1}), (q_{i+1}, p_{i+1}, q_{i+2})) \in (\delta \times \delta) \mid 0 \le i < m-1\} \cup\\
\{&((q_i, p_{i+1}, q_{i+1}), (q_{i+1}, p_{i+2}, q_{i+2})) \in (\delta \times \delta) \mid 0 \le i < m-2\}
\end{aligned}$$

The swap reactive automaton of a pattern P of length m has exactly $m+1$ states, (at most) $3m-2$ transitions and (at most) $8m-12$ reactive links.

To simplify the notation we will use the symbol $\tau(i, j)$ to indicate the standard transition starting from state q_i and labeled by character p_j, i.e. $\tau(i, j) = (q_i, p_j, q_{i+1})$. Since all standard transitions of the automaton starting from state q_i, reach the state q_{i+1}, the notation defined above is not ambiguous.

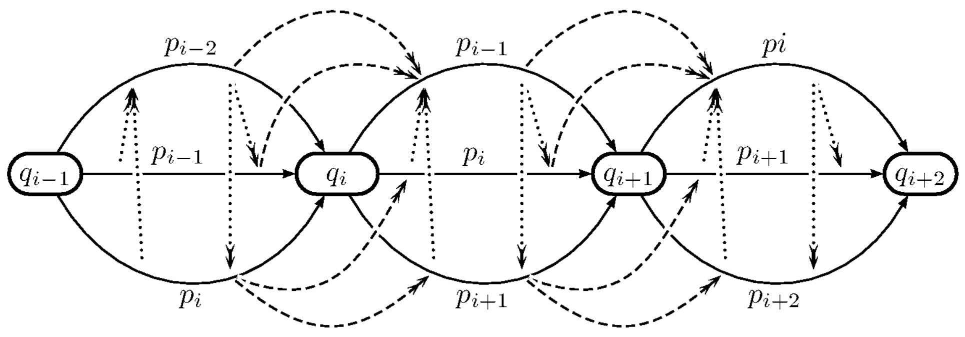


Figure 1. The general structure of a swap reactive automaton. Standard transitions in δ are represented with solid lines, reactive links in T^- are represented with dashed lines, while reactive links in T^+ are represented with dotted lines.

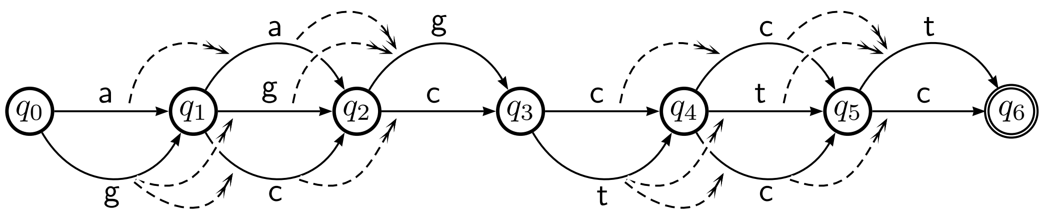


Figure 2. The swap reactive automaton for the pattern P = agcctc. Standard transitions are represented with solid lines while reactive links in T^- are represented with dashed lines. Reactive links in T^+ are not represented.

Figure 1 shows the general structure of a portion (from state q_{i-1} to state q_{i+2}) of the swap reactive automaton, while Figure 2 shows the swap reactive automaton constructed for the pattern P = agcctc. Each state has 3 standard transitions to the next state, with the exception states q_0 and q_{m-1}. Specifically state q_i, for $0 < i < m-1$, has transitions to state q_{i+1} labeled by characters p_{i-1}, p_i and p_{i+1}, respectively. Transitions labeled by character p_i are not involved in any swap, those labeled by character p_{i+1} start a new swap, while transitions labeled by character p_{i-1} end a previously started swap. Due to its external positions, state q_0 has only 2 transitions reaching state q_1. Similarly state q_{m-1} has only two transitions reaching state q_m.

When a new swap starts (with a transition from q_i to q_{i+1} labeled by p_{i+1}) two reactive links switch off the next transitions from state q_{i+1} with the exception of the transition which ends the swap. Otherwise, when a swap ends (following a transition labeled by p_{i-1}) or no swap is involved in the current transition (following a transition labeled by p_i) a reactive link switches off the next transition which ends a swap.

The reactive links in T^+ allows all transitions from q_i to q_{i+1} to be active after any step. The self loop of the initial state allows an occurrence of the pattern to begin at any position of the text.

The two following properties of an SRA trivially follows by Definition 6.

Property 7. There is no state $q_i \in Q$, with $0 \leq i < m$, such that $\tau(i,i)$ and $\tau(i,i+1)$ are both in δ, and $p_i = p_{i+1}$.

Property 8. There is no state $q_i \in Q$, with $0 \leq i < m$, such that $\tau(i,i)$ and $\tau(i,i-1)$ are both in δ, and $p_i = p_{i-1}$.

In what follows we assume that $P = p_0p_1p_2 \cdots p_{m-1}$ is a string of length m and $T = t_0t_1t_2 \cdots t_{n-1}$ is a string of length n, both over the alphabet Σ. Moreover we assume that $R = (A, T^+, T^-)$ is the SRA of P.

Lemma 9. *Let $\langle (Q_0, \delta_0), (Q_1, \delta_1), \ldots, (Q_n, \delta_n) \rangle$ be a nondeterministic run of the SRA R over the string T. If state $q_i \in Q_j$, with $i > 0$, then only one of the following relations holds*

(a) $\tau(i,i) \notin \delta_j$ *and* $\tau(i,i+1) \notin \delta_j$, *or*
(b) $\tau(i,i-1) \notin \delta_j$

Proof. Without loose in generality, we suppose that p_i, p_{i-1} and p_{i-2} are all different characters.

If $q_i \in Q_j$ then it has been reached from state q_{i-1} through one of three transitions starting from q_{i-1}, i.e. $\tau(i-1,i-2)$, $\tau(i-1,i-1)$ or $\tau(i-1,i)$. Since both $(\tau(i-1,i-2), \tau(i,i-1))$ and $(\tau(i-1,i-1), \tau(i,i-1))$ are the only reactive links in T^- starting from $\tau(i-1,i-2)$ and $\tau(i-1,i-1)$, it follows that if q_i is reached through transitions label by p_{i-2} or p_{i-1} we have that $\tau(i-1,i-1) \notin \delta_j$. Moreover $\tau(i,i)$ and $\tau(i,i+1)$ are both in δ.

Similarly, if q_i is reached through transitions label by p_i, since the reactive links $(\tau(i-1,i), \tau(i,i))$ and $(\tau(i-1,i), \tau(i,i+1))$ are the only in T^- starting from $\tau(i-1,i)$, we have that $\tau(i,i) \notin \delta_j$ and $\tau(i,i+1) \notin \delta_j$, while $\tau(i,i-1) \in \delta_j$. □

The following two technical results prove that the swap reactive automaton given in Definition 6 recognizes all and only the strings ending with an occurrence of P.

Lemma 10. *Let $\langle (Q_0, \delta_0), (Q_1, \delta_1), \ldots, (Q_n, \delta_n) \rangle$ be a nondeterministic run of the SRA R over the string T. If state $q_{i+1} \in Q_{j+1}$, for $0 < i < m$ and $0 < j < n$, then one of the following relations holds*

(a) $P_i \propto T_j$, *or*
(b) $P_{i-1} \propto T_{j-1}$ *and* $p_{i+1} = t_j$

Proof. We prove this result by induction on the value of i.

When $i = 0$ we have that $q_1 \in Q_1$, i.e. q_1 is active after we read character t_j of the text. Thus it must be $t_j = p_0$ (in which case condition (a) holds) or $t_j = p_1$ (in which case condition (b) holds).

Suppose now that the result holds for values less than i and prove it for $i > 0$. Since q_{i+1} is active after we read t_j it follows that q_i has been active after we read character t_{j-1}. This because q_{i+1} can be reached only from state q_i.

It implies by induction that

(1) $P_{i-1} \propto T_{j-1}$ or
(2) $P_{i-2} \propto T_{j-2}$ and $p_i = t_{j-1}$.

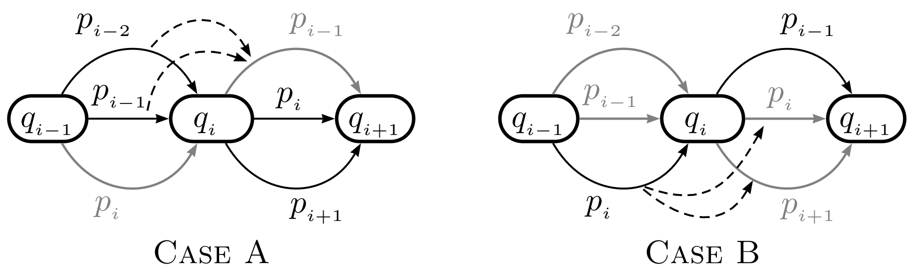


Figure 3. Two different conditions described in Lemma 10. Active transitions are represented in black lines, while switched off transitions are represented in light gray lines.

If (1) holds (see Figure 3 Case A) then state q_i has been reached through the transition labeled by p_{i-1} or through the transition labeled by p_{i-2}. Since both reactive links $(\tau(i-1,i-2), \tau(i,i-1))$ and $(\tau(i-1,i-1), \tau(i,i-1))$ are in the set T^-, it turns out that transition $\tau(i,i-1) \notin \delta_{j-1}$, before reading t_j. As a consequence, state q_{i+1} can be reached only through the transition labeled by character p_i (in which case condition (a) holds) or through transition labeled by character p_{i+1} (in which case condition (b) holds).

Otherwise, if condition (2) holds (see Figure 3 Case B), it follows that state q_i has been reached through transition labeled by character p_i. Since both reactive links $(\tau(i-1,i), \tau(i,i))$ and $(\tau(i-1,i), \tau(i,i+1))$ are in T^-, it turns out that transitions $\tau(i,i)$ and $\tau(i,i+1)$ are switched off. As a consequence state q_{i+1} can be reached only through transition $\tau(i,i-1)$, in which case condition (a) holds. □

Corollary 11. *Let $\langle(Q_0,\delta_0),(Q_1,\delta_1),\ldots,(Q_n,\delta_n)\rangle$ be a nondeterministic run of the SRA R over the string T. If state $q_m \in Q_{j+1}$ then $P \propto T_j$.*

Proof. Observe that state q_m can be reached only by state q_{m-1}, through the transition labeled by character p_{m-1} (no swap involved) or through the transition labeled by character p_{m-2} (end of a swap). The result follows by such observation and by Lemma 10. □

Lemma 12. *Let $\langle(Q_0,\delta_0),(Q_1,\delta_1),\ldots,(Q_n,\delta_n)\rangle$ be a nondeterministic run of the SRA R over the string T. If $P_i \propto T_j$ then $q_{i+1} \in Q_{j+1}$.*

Proof. We prove the lemma by induction on the value of i. For the base case, observe that if $P_0 \propto T_j$ then we have $p_0 = t_j$. Since q_0 is always active, due to the self loop, and $(q_0, p_0, q_1) \in \delta$, it follows that q_1 is active after we read t_j.

Suppose now that $i > 0$ and assume that the result holds for values less than i. The condition $P_i \propto T_j$ implies that

(1) $P_{i-2} \propto T_{j-2}$, $p_{i-1} = t_j$ and $p_i = t_{j-1}$, or
(2) $P_{i-1} \propto T_{j-1}$ and $p_i = t_j$.

If condition (1) holds then, by induction, state q_{i-1} is active after we read characters t_{j-2}. This implies that the transition $\tau(i-1,i)$ is switched on. Since $t_{j-1} = p_i$, after we read character t_{j-1} state q_i is active. Finally, observe that the reactive link $(\tau(i-1,i), \tau(i,i-1))$ is not in T^-. Thus the transition $\tau(i,i-1)$ is switched on before

```
BPSRA($P, m, T, n$)
 1. for $c \in \Sigma$ do
 2.     $M[c] \leftarrow 0$
 3. for $i \leftarrow 0$ to $m-1$ do
 4.     $M[p_i] \leftarrow M[p_i] \mid (1 \ll i)$
 5. $F \leftarrow 1 \ll (m-1)$
 6. $A \leftarrow 0$
 7. $B \leftarrow 0^{m-1}1 \ \& \ M[t_0]$
 8. $C \leftarrow 0^{m-1}1 \ \& \ M[t_1]$
 9. for $i \leftarrow 1$ to $n-1$ do
10.     $H \leftarrow (A \ll 1) \mid (M \ll 1) \mid 1$
11.     $A \leftarrow (C \ll 1) \ \& \ M[t_j]$
12.     $B \leftarrow H \ \& \ M[t_j]$
13.     $C \leftarrow H \ \& \ M[t_{j+1}]$
14.     if $((A \mid B) \ \& \ F)$ then
15.         output($i - m + 1$)
```

Figure 4. The Bit Parallel Swap Reactive Automaton Matcher for swap matching.

reading character t_j. Since $t_j = p_{i-1}$, we can conclude that after we read character t_j the state q_{i+1} is active.

Suppose now that condition (2) holds. Then we can state, by induction, that q_i is active after we read charactre t_{j-1}. This implies that the transition $\tau(i,i)$ is switched on. Since $t_j = p_i$, after we read character t_j state q_{i+1} is active. □

The following Theorem 13 trivially follows by Corollary 11 and Lemma 12

Theorem 13 (Correctness). *The swap reactive automaton R recognizes all and only the strings S over Σ such that $P \propto S$.* □

3.2 A Bit Parallel Simulation

In this section we show how to simulate the swap reactive automaton of an input pattern P, as given in Definition 6, by using bit-parallelism [5].

The bit-parallelism technique takes advantage of the intrinsic parallelism of the bit operations inside a computer word, allowing to cut down the number of operations that an algorithm performs by a factor of at most w, where w is the number of bits in the computer word. It has been extensively used in the field of exact string matching [14] for efficiently simulating non-deterministic automata. However it has also been efficiently used in the field field of multiple pattern matching [10,11,12] and approximate string matching [8,17].

In contrast with standard bit-parallel simulation of non-deterministic automata, where states of the automaton are represented by bits in a bit vector, we use bits to represent transitions of the automaton. In this context an active transition which has been just crossed during the last step is represented by a bit set to 1, while all other transitions are represented by a bit set to 0.

Let $P = p_0p_1p_2 \cdots p_{m-1}$ be a pattern of length m and let $T = t_0t_1t_2 \cdots t_{n-1}$ be a text of length n over Σ. Moreover let $R = (A, T^+, T^-)$ be the SRA of P. The representation of R uses an array M of σ bit-vectors, each of size m, where the i-th bit of $M[c]$ is set if $p_i = c$.

Automaton configurations are then encoded using 3 bit-vectors of m bits. A bit vector A encodes transitions from q_i to q_{i+1} labeled by character p_{i-1}, a bit vector B encodes transitions from q_i to q_{i+1} labeled by character p_i, while another bit vector C encodes transitions from q_i to q_{i+1} labeled by character p_{i+1}. Specifically we have that the i-th bit of A is set iff (q_i, p_{i-1}, q_{i+1}) is switched on and q_{i+1} is active, the i-th bit of B is set iff (q_i, p_i, q_{i+1}) is switched on and q_{i+1} is active and, finally, the i-th bit of C is set iff (q_i, p_{i+1}, q_{i+1}) is switched on and q_{i+1} is active.

When a search starts, the configurations of the 3 bit-vectors are initialized as $A = 0^m$, $B = (0^{m-1}1 \,\&\, M[t_0])$ and $C = (0^{m-1}1 \,\&\, M[t_1])$.

Then, the text T is scanned, character by character, from left to right and the automaton configuration is updated accordingly. Specifically transitions on character t_j are simulated by performing the following bitwise operations

(i) $A = (C \ll 1) \,\&\, M[t_{j-1}]$
(ii) $B = ((A \ll 1) \mid (B \ll 1) \mid 1) \,\&\, M[t_j]$
(iii) $C = ((A \ll 1) \mid (B \ll 1) \mid 1) \,\&\, M[t_{j+1}]$

Operation (i) ends a swap and indicates that transition $\tau(i, i-1)$ can be crossed only if $\tau(i-1, i)$ has been crossed in the previous step and $t_j = p_{i-1}$. Operations (ii) and (iii) indicates that transitions $\tau(i, i)$ and $\tau(i, i+1)$ can be crossed only if $\tau(i-1, i-1)$ or $\tau(i-1, i-2)$ have been crossed at the previous step and, moreover, $t_j = p_i$ or $t_j = p_{i+1}$, respectively.

The simulation showed above uses 7 bitwise operations for each text character scanned during the searching phase.

After we perform transition on character t_j, state q_m is active if and only if the rightmost bit in A, or in B, is active. Specifically if the test $((A \mid B) \,\&\, 10^{m-1}) \neq 0$ is true, then an occurrence has been found ending at position j of the text.

The resulting algorithm is named Bit Parallel Swap Reactive Automaton Matcher (BPSRA). Its pseudocode is shown in Figure 4.

The preprocessing phase of the BPRSA Matcher (lines 1–5) has a $\mathcal{O}(m + \sigma)$-time complexity. Its searching phase (lines 6–15) has a $\mathcal{O}(n)$-time complexity, if $m \leq w$. When $m > w$ we need to represent the whole automaton by using $3\lceil n/m \rceil$ computer words, so that the worst case time complexity is $\mathcal{O}(n\lceil n/m \rceil)$.

3.3 A More Efficient Simulation

In this section we propose a more efficient simulation of the swap reactive automaton of an input pattern P, by using bit parallelism.

As before, let $P = p_0p_1p_2 \cdots p_{m-1}$ and $T = t_0t_1t_2 \cdots t_{m-1}$ be two strings of length m over the alphabet Σ. Moreover let $R = (A, T^+, T^-)$ be the SRA of P.

Before entering into details it's convenient to give the following definition of a *string with disjoint triplets*, which we will use in the following discussion.

Definition 14 (String With Disjoint Triplets). *A string $S = s_0s_1s_2 \cdots s_{m-1}$, of length m, over an alphabet Σ, is a string with disjoint triplets (SDT) if $s_i \neq s_{i+2}$, for $i = 0, \ldots, m-3$.*

The above definition implies that in the SRA of S the standard transitions from state q_i to q_{i+1}, for $i = 0, \ldots, m-1$, are labeled by different characters.

Text	**4**	**8**	**16**	**32**
Genome Sequence	0.6080	0.2140	0.0170	0.0010
Protein Sequence	0.8420	0.6160	0.3140	0.1170
English Text	0.9380	0.8440	0.6820	0.4380
Italian Text	0.9130	0.7630	0.5100	0.2500
French Text	0.9230	0.7910	0.5930	0.3250
Chinese Text	0.9860	0.9510	0.8990	0.7750

Table 1. Relative Frequency of SDT in different text buffers.

SDT are very common strings in real data, especially in such cases where the size of the alphabet is large. For instances they are common in natural language texts where it's not common to find words with equal characters with a distance of one character. Table 1 shows the relative frequency of SDT in different text buffers and in particular on a genome sequence, on a protein sequence and on four natural language texts. For each text buffer data have been collected by extracting 10.000 random patterns of different length (ranging from 4 to 32) from the text, and computing the corresponding frequency of SDT.

With the exception of biological sequences, where an SDT is generally uncommon due to the small size of the corresponding alphabet, in the case of natural language texts the percentage of SDT are often over 70 %. In particular it turns out that English and Chinese texts have the largest percentage of SDT.

In the following we propose an efficient simulation of the swap reactive automaton of a pattern P which works properly when P is a SDT. Conversely, if P is not a SDT, then the simulation works as an *oracle*, i.e. it may recognize also strings which are not swap occurrences of the pattern. In this last case an additional naive verification must be performed. The actual advantage in using this new simulation is that it can be performed by only 2 bitwise operations for each iteration of the algorithm. This is a significant improvement compared with previous simulations where at least 7 bitwise operations are needed for each iteration of the algorithm.

In the new proposed simulation the representation of R uses an array B of σ^2 bit-vectors, each of size m, where the i-th bit of $B[c_1, c_2]$ (which we indicate as $B[c_1, c_2]_i$) is defined as

$$B[c_1, c_2]_i = \begin{cases} 1 & \text{if } \tau(i,1), \tau(i+1,2) \in \delta \text{ and } (\tau(i,1), \tau(i+1,2)) \notin T^- \\ 0 & \text{otherwise} \end{cases} \qquad (1)$$

for $c_1, c_2 \in \Sigma$, and $0 \leqslant i < m$. Roughly speaking, the matrix M encodes the couples of admissible consecutive transitions in R.

Automaton configurations are then encoded as a bit-vector D of m bits (the initial state does not need to be represented), where the i-th bit of D is set if and only if the state q_{i+1} is active.

When a search starts, the configuration D is initialized to $B[t_0, t_1]$. Then, while the string T is read from left to right, the automaton configuration is updated accordingly for each text character.

Suppose the last transition has been performed on character t_{j-1}, with $0 < j < n-1$, leading to a configuration vector D of the SRA. Then a transition on character

t_j can be implemented by the bitwise operations

$$D^{(j)} = \begin{cases} B[t_0, t_1] & \text{if } j = 1 \\ (D^{(j-1)} \ll 1) \mathbin{\&} B[t_{j-1}, t_j] & \text{otherwise} \end{cases} \tag{2}$$

It turns out that, if P is a SDT, then the simulation of the SRA described above works properly, as stated by the following lemma.

Lemma 15. *Let P be a SDT of length m and let T be a text of length n. Suppose the matrix B is initialized according to* (1) *and suppose to scan the string T, from right to left, and to perform transitions according to* (2). *After we read character t_j of the text the leftmost bit of $D^{(j)}$ is set if and only if $P \propto T_j$.*

Proof. Let R be the SRA of P. By Theorem 13 we know that R recognizes all and only the prefix of T ending with a swap occurrence of P. We prove that, after we read text character t_j, state q_i is active if and only if the i-th bit in $D^{(j)}$ is set. This will prove the thesis.

We prove it by induction on i. The base case is when $i = 1$. If state q_2 is active after we scan the first two characters of T then one of the following relations holds:

(1) $t_0 t_1 = p_0 p_1$,
(2) $t_0 t_1 = p_1 p_0$, or
(3) $t_0 t_1 = p_0 p_2$.

Observe that if $p_0 = p_1$ only transition $\tau(0,0)$ is in δ, otherwise both $\tau(0,0)$ and $\tau(0,1)$ are in δ. Moreover we have also

– $(\tau(0,0), \tau(1,1)) \notin T^-$, thus the 2nd bit of $B[p_0, p_1]$ is set;
– if $\tau(0,1) \in \delta$ then $(\tau(0,1), \tau(1,0)) \notin T^-$, thus the 2nd bit of $B[p_1, p_0]$ is set;
– $(\tau(0,0), \tau(1,2)) \notin T^-$, thus the 2nd bit of $B[p_0, p_2]$ is set;

Thus after the initialization $D^{(1)} = B[t_0, t_1]$, the second bit of $D^{(1)}$ is set, proving the base case.

Conversely, if the second bit of $B[t_0 t_1]$ is set then, by equation 1, it follows that both $\tau(0,0)$ and $\tau(1,1)$ are in δ and $(\tau(0,0), \tau(1,1)) \notin T^-$. Thus state q_2 is active after we scan $t_0 t_1$.

Suppose now that the result holds for values less than i and prove it for i. If state q_i is active after we scan character t_j, then state q_{i-1} is active just before reading character t_{j-1} and, by induction, the $(i-1)$-th bit of $D^{(j-1)}$ is active before reading character t_{j-1}. It follows that both $(q_{i-2}, t_{j-2}, q_{i-1})$ and (q_{i-1}, t_{j-1}, q_i) are in δ and that transition (q_{i-1}, t_{j-1}, q_i) is not switched off when q_{i-1} is active. Thus $((q_{i-2}, t_{j-2}, q_{i-1}), (q_{i-1}, t_{j-1}, q_i))$ is not in T^- and we can conclude that the k-th bit of $B[t_{j-2}, t_{j-1}]$ is set.

Conversely, suppose that the i-th bit of $B[t_{j-2}, t_{j-1}]$ is set. Thus just before reading character t_{j-1} the $(i-1)$-th bit of $D^{(j-1)}$ is set and, by induction, state q_{i-1} is active. It follows that the i-th bit of $B[t_{j-2}, t_{j-1}]$ is set, which implies that both $(q_{i-2}, t_{j-2}, q_{i-1})$ and (q_{i-1}, t_{j-1}, q_i) are in δ and the active link $((q_{i-2}, t_{j-2}, q_{i-1}), (q_{i-1}, t_{j-1}, q_i))$ is not in T^-. We can conclude that after we read character t_{j-1}, state q_i is active. □

The resulting algorithm is named Bit Parallel Swap Reactive Oracle (BPSRO). Its pseudocode is shown in Figure 5. The BPSRO algorithm works as the original Shift-And algorithm [5] for the exact string matching problem. During the preprocessing

```
BPSRO($P, m, T, n$)
1.  for $c_1, c_2 \in \Sigma$ do $B[c_1, c_2] \leftarrow 0$
2.  for $i = 1$ to $m - 1$ do
3.      $B[p_{i-1}, p_i] \leftarrow B[p_{i-1}, p_i] \mid (1 \ll i)$
4.      $B[p_i, p_{i-1}] \leftarrow B[p_i, p_{i-1}] \mid (1 \ll i)$
5.      if $(i < m - 1)$ then
6.          $B[p_{i-1}, p_{i+1}] \leftarrow B[p_{i-1}, p_{i+1}] \mid (1 \ll i)$
7.      if $(i > 1)$ then
8.          $B[p_{i-2}, p_i] \leftarrow B[p_{i-2}, p_i] \mid (1 \ll i)$
9.          if $(i < m - 1)$ then
10.             $B[p_{i-2}, p_{i+1}] \leftarrow B[p_{i-2}, p_{i+1}] \mid (1 \ll i)$
11. $F \leftarrow 1 \ll (m - 1)$, $D \leftarrow 0$
12. for $i \leftarrow 1$ to $n - 1$ do
13.     $D \leftarrow ((D \ll 1) \mid 1)$ & $B[t_{i-1}, t_i]$
14.     if $(D$ & $F)$ then
15.         if ($P$ is a SDT) then output($i - m + 1$)
16.         else check occurrence at position $(i - m + 1)$
```

Figure 5. The Bit Parallel Swap Reactive Oracle Matcher for swap matching.

phase the algorithm computes the matrix B of bit masks. The preprocessing phase (lines 1–11) has a $\mathcal{O}(m + \sigma^2)$-time complexity and requires $\mathcal{O}(\sigma^2)$ space.

During the searching phase the algorithm reads characters of the text, one by one, and simulates transitions on the SRA, accordingly. If the leftmost bit of D is set after we read character t_j, i.e. if $(D\&10^{m-1}) \neq 0$, then an occurrence is reported at position $j - m + 1$. The searching phase (lines 12–16) has a $\mathcal{O}(n)$-time complexity, if $m \leq w$ and P is an SDT. Assuming that P is an SDT, when $m > w$ we need to represent the whole automaton by using $3\lceil n/m \rceil$ computer words, so that the worst case time complexity is $\mathcal{O}(n\lceil n/m \rceil)$. In addition, when P is not an SDT, the algorithm works as an oracle and an additional verification phase is needed in order to check all candidate occurrences. Such a naive verification (line 16) takes $\mathcal{O}(m)$-time, so that the overall worst case time complexity of the algorithm is $\mathcal{O}(nm)$.

4 Experimental Results

In this section we briefly present experimental evaluations in order to understand the performances of the newly presented algorithms.

Specifically we compared the Bit Parallel Swap Reactive Automaton algorithm (BPSRA) and the Bit Parallel Swap Reactive Oracle algorithm (BPSRO) against the Bit Parallel Cross Sampling algorithm (BPCS) [9,6], which is one of the most efficient linear algorithm present in literature. Other practical algorithms are known in literature [7] for the swap matching problem, which show a sub-linear behavior in practical cases, however they use a backward scan of the text, an efficient technique which can be applied to almost all automata based algorithm (including ours) and which is out of the scope of the present paper.

All algorithms have been implemented in the C programming language and have been tested using the SMART tool[1], which have been provided for testing exact string

[1] The SMART tool is available online at: `http://www.dmi.unict.it/~faro/smart/`

m	2	4	8	16	32	2	4	8	16	32	2	4	8	16	32
BPCS	16.0	15.9	15.9	16.0	15.9	15.9	15.9	16.1	16.1	16.2	16.0	15.9	16.1	16.3	16.0
BPSRA	**15.4**	15.3	15.3	15.2	15.2	15.3	15.8	15.4	15.3	15.3	15.3	15.4	15.3	15.3	15.3
BPSRO	20.4	**13.7**	**11.4**	**11.2**	**11.2**	**12.0**	**11.2**	**11.4**	**11.3**	**11.3**	**12.8**	**11.5**	**11.5**	**11.3**	**11.3**
	(A) genome sequence					(B) protein sequence					(C) natural language text				

Table 2. Experimental results on (A) a genome sequence, (B) a protein sequence and (C) a natural language text.

matching algorithm [14] but allows to be adapted also for approximate string matching algorithms. The experiments were executed locally on an MacBook Pro with 4 Cores, a 2 GHz Intel Core i7 processor, 4 GB RAM 1333 MHz DDR3, 256 KB of L2 Cache and 6 MB of Cache L3. Algorithms have been compared in terms of running times, including any preprocessing time.

For the evaluation we use a genome sequence, a protein sequence and a natural language text (English language), all sequences of 4MB. The sequences are provided by the Smart research tool. In all cases the patterns were randomly extracted from the text and the value m was made ranging from 2 to 32. For each case we reported the mean over the running times of 500 runs.

Table 4 shows experimental results on the three different sequences. Running times are expressed in thousands of seconds. Best times have been boldfaced.

From the experimental results it turns out that the BPSRA algorithm has almost the same performance of the BPCS algorithm, but is slightly better in all practical cases. Both the BPCS and the BPSRA show a linear behavior. The BPSRO shows instead a decreasing trend, which is much evident for the case of short patterns. This behavior is due to the presence of a larger number of verification tests which must be run when the pattern is short and is not an SNR. It turns out moreover that in almost all cases the BPSO is faster than the other algorithms, due to its less complex simulation engine.

5 Conclusions

In this paper we have presented a new theoretical approach to solve the swap matching problem in strings. The new approach uses a reactive automaton with only m states and (at most) $3m-2$ transitions. We propose also two different approaches to simulate the automaton by using bit-parallelism.

As in the case of the Cross Sampling algorithm the new proposed approach can be extended to obtain more efficient solution by scanning the text from right to left. Our future works will consider such improvements.

References

1. A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein: *Pattern matching with swaps*, in IEEE Symp. on Foundations of Computer Science, 1997, pp. 144–153.
2. A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein: *Pattern matching with swaps*. Journal of Algorithms, 37(2) 2000, pp. 247–266.
3. A. Amir, R. Cole, R. Hariharan, M. Lewenstein, and E. Porat: *Overlap matching*. Inf. Comput., 181(1) 2003, pp. 57–74.
4. A. Amir, G. M. Landau, M. Lewenstein, and N. Lewenstein: *Efficient special cases of pattern matching with swaps*. Inf. Proc. Letters, 68(3) 1998, pp. 125–132.

5. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching.* Commun. ACM, 35(10) 1992, pp. 74–82.
6. M. Campanelli, D. Cantone, and S. Faro: *A new algorithm for efficient pattern matching with swaps*, in IWOCA 2009: 20th International Workshop on Combinatorial Algorithms, vol. 5874 of Lecture Notes in Computer Science, Springer, 2009, pp. 230–241.
7. M. Campanelli, D. Cantone, S. Faro, and E. Giaquinta: *Pattern matching with swaps in practice.* International Journal of Foundation of Computer Science, 23(2) 2012, pp. 323–342.
8. D. Cantone, S. Cristofaro, and S. Faro: *Efficient string-matching allowing for non-overlapping inversions.* Theor. Comput. Sci., 483 2013, pp. 85–95.
9. D. Cantone and S. Faro: *Pattern matching with swaps for short patterns in linear time*, in SOFSEM 2009: Theory and Practice of Computer Science, 35th Conference on Current Trends in Theory and Practice of Computer Science, vol. 5404 of Lecture Notes in Computer Science, Springer, 2009, pp. 255–266.
10. D. Cantone, S. Faro, and E. Giaquinta: *A compact representation of nondeterministic (suffix) automata for the bit-parallel approach*, in Combinatorial Pattern Matching, vol. 6129 of Lecture Notes in Computer Science, 2010, pp. 288–298.
11. D. Cantone, S. Faro, and E. Giaquinta: *A compact representation of nondeterministic (suffix) automata for the bit-parallel approach.* Inf. Comput., 213 2012, pp. 3–12.
12. D. Cantone, S. Faro, and E. Giaquinta: *On the bit-parallel simulation of the nondeterministic aho-corasick and suffix automata for a set of patterns.* J. Discrete Algorithms, 11 2012, pp. 25–36.
13. M. Crochemore and D. M. Gabbay: *Reactive automata.* Inf. Comput., 209(4) Apr. 2011, pp. 692–704.
14. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) 2013, p. 13.
15. S. Faro and M. O. Külekci: *Fast multiple string matching using streaming SIMD extensions technology*, in String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, vol. 7608 of Lecture Notes in Computer Science, Springer, 2012, pp. 217–228.
16. D. M. Gabbay: *Pillars of computer science*, Springer-Verlag, 2008, ch. Introducing reactive Kripke semantics and arc accessibility, pp. 292–341.
17. S. Grabowski, S. Faro, and E. Giaquinta: *String matching with inversions and translocations in linear average time (most of the time).* Inf. Process. Lett., 111(11) 2011, pp. 516–520.
18. C. S. Iliopoulos and M. S. Rahman: *A new model to solve the swap matching problem and efficient algorithms for short patterns*, in SOFSEM 2008, vol. 4910 of Lecture Notes in Computer Science, Springer, 2008, pp. 316–327.
19. S. Muthukrishnan: *New results and open problems related to non-standard stringology*, in Combinatorial Pattern Matching, CPM 95, vol. 937 of Lecture Notes in Computer Science, Springer, 1995, pp. 298–317.

Deciding the Density Type of a Given Regular Language[⋆]

Stavros Konstantinidis and Joshua Young

Department of Mathematics and Computing Science,
Saint Mary's University, Halifax, Nova Scotia, B3H 3C3 Canada
`s.konstantinidis@smu.ca, jyo04@hotmail.com`

Abstract. In this paper, the density of a language is the function that returns, for each n, the number of words in the language of length n. We consider the question of deciding whether the density of a given regular language L is exponential or polynomial. This question can be answered in linear time when L is given via a DFA. When L is given via an NFA, we show that L has exponential density if and only if the NFA has a strongly connected component (SCC) in which two equal length walks from the same state have different labels. This characterization leads to a simple quadratic time algorithm. However, a more elegant approach produces a linear time algorithm whose proof of correctness involves the theorem of Fine and Wilf and the greatest common divisor (gcd) of the lengths of all cycles in the SCC. We have implemented both the quadratic and linear time algorithms using the FAdo library for automata, and present results of a few test cases.

Keywords: algorithm, automaton, complexity, density, strongly connected components, regular language

1 Introduction

Following [11], we define the density of a language L to be the function that returns, for every nonnegative integer n, the number of words in L of length n. This concept is of central importance in language theory. In particular, [11] and [9] characterize regular languages of exponential density, where the characterization of [9] leads to a linear time algorithm for deciding whether a regular language is of exponential density when L is given via a *deterministic* finite automaton (DFA). This characterization is very simple: the DFA has a state that belongs to two different cycles—we note that the same idea was used in [3] in the context of encoding data into DNA languages that are described via certain DFAs.

Here, we consider the question of characterizing regular languages of exponential density when they are given via *nondeterministic* finite automata (NFAs). Our characterization is that the NFA has a strongly connected component (SCC) containing two walks of the same length, starting at the same state, and having different labels. This characterization leads to two algorithms: (i) a 'direct' quadratic time algorithm and (ii) an 'elegant' linear time algorithm that uses breadth first search (BFS) and the greatest common divisor (gcd) of the lengths of all cycles in the SCC. The proof of correctness involves a few technical facts about walk lengths in (directed) graphs, and a simple generalization of the theorem of Fine and Wilf [2].

The paper is organized as follows. Section 2 contains the basic notation and terminology about regular languages, automata and graphs. In Section 3, we consider the question of whether a given regular language L is of exponential density and present

⋆ Research supported by NSERC.

our characterization and the quadratic time algorithm. In Section 4, we present the linear time algorithm, and in Section 5 the correctness of this algorithm. In Section 6, we discuss our implementation using the FAdo library for automata [6], and results of a few test cases.

2 Basic Notation and Background

We begin this section with notation and concepts on words and languages, and then on finite automata. We use [8] as a general reference.

2.1 Sets, Words, Languages

We write $\mathbb{N}$ and $\mathbb{N}_0$ for the sets of positive integers and nonnegative integers, respectively. For a set S, we denote by $|S|$ the cardinality of S. We consider an arbitrary alphabet Σ containing at least two symbols. As usual, the set of all words over Σ is denoted by Σ^*. We write λ for the empty word and Σ^+ for the set of all nonempty words. The length of a word w is the number of alphabet symbols occurring in w and is denoted by $|w|$. For an integer $n \geq 0$, the expression $(w)^n$ is the word consisting of n copies of w. A *prefix* (resp. *suffix*) of a word w is any word u such that $w = ux$ (resp. $w = xu$) for some word x. We write $\text{Prefix}(w)$ for the set of all prefixes of w.

A language is any set of words. A word w is called an L-word if $w \in L$. As usual, for any integer $n \geq 0$, if L is a language then L^n is the language whose words consist of any n concatenated words from L. In particular Σ^n is the set of all words of length n. Also, L^* is the union of L^n, for all $n \geq 0$, and $L^+ = L^* - \{\lambda\}$. A language C is called a *block code* if all C-words are of the same length. When there is no risk of confusion, a singleton language $\{w\}$ is written as w.

A nonempty word w is called *periodic* with a period of length $g \in \mathbb{N}$, if there is a word v of length g such that $w \in \text{Prefix}(v^*)$. For example, $abbabba$ is periodic with a period of length 3, as it belongs to $\text{Prefix}((abb)^*)$. A (right) *infinite word* is a sequence $\mathbf{a} : \mathbb{N} \to \Sigma$. It is called periodic with a period of length $g \in \mathbb{N}$, if there is $g \in \mathbb{N}$ such that $\mathbf{a}(i + g) = \mathbf{a}(i)$, for all $i \in \mathbb{N}$. In this case, following [2], we write

$$\mathbf{a} = (v)^\omega,$$

where v is the word $\mathbf{a}(1) \cdots \mathbf{a}(g)$.

The *density* of a language L is the function d_L that maps every nonnegative integer n to $\text{d}_L(n) =$ the number of L-words of length n. We say that a regular language L has *exponential density* if the density of L is not polynomially upper-bounded—see below for the definition of regular language. This definition is justified by a result of [11] stating that the density of any regular language is either polynomially upper-bounded or has a subsequence of order $\Omega(2^n)$.

2.2 Automata, graphs, cycles

A complete deterministic finite automaton (complete DFA, for short) is a quintuple

$$M = (\Sigma, K, \delta, s, F)$$

such that K is the nonempty set of states, s is the start state, F is the set of final states and $\delta : K \times \Sigma \to K$ is the transition function, which can be extended as

$\delta : K \times \Sigma^* \to K$ in the usual way. If the function δ is partial, then M is not complete—we simply call it a *DFA*. A nondeterministic finite automaton (*NFA*) is a quintuple

$$M = (\Sigma, K, T, s, F)$$

such that K, s, F are as in the case of a deterministic automaton, and T is the finite set of *transitions*, which are triples of the form (p, σ, q) with $\sigma \in \Sigma$ and $p, q \in K$. In this case, we say that the *transition is going out of the state* p. A DFA is a special type of an NFA where $(p, \sigma, q) \in T$ exactly when $\delta(p, \sigma) = q$.

The NFA M can be viewed as a directed labeled graph having K as the set of vertices and any triple (p, σ, q) as a labeled arc exactly when (p, σ, q) is a transition in T. A *walk* in M is a sequence

$$(p_0, \sigma_1, p_1, \ldots, \sigma_n, p_n) \tag{1}$$

such that (p_{i-1}, σ_i, p_i) is a transition of M, for each $i = 1, \ldots, n$. In this case, the word $\sigma_1 \cdots \sigma_n$ is called the *label* of the walk. As usual, the language $L(M)$ *accepted* by M is the set of all labels that appear in walks as above such that p_0 is the start state and p_n is a final state. These languages constitute the class of *regular* languages—see [12] for more information on regular languages.

The NFA M is called *trim* if every state of M occurs in some path from the start state to a final state. The *size* of M is $|K| + |T|$, that is the number of states plus the number of transitions in M. We note that if M is trim then $|K| \leq |T| + 1$ and, therefore, the size of M is dominated by the number of transitions in M. A state in an automaton is called a *fork state* if there are at least two transitions going out of that state.

A *path* in the NFA M is a walk in which no state appears twice. A *closed walk* in M is any walk as in (1) where $p_n = p_0$. A *cycle* in the NFA is a closed walk in which only the first and last states are equal—hence, in (1) the p_i's would be unique, for $i = 1, \ldots, n$. The special cycle (p), where p is any state, is called trivial. A *strongly connected component (SCC)*, with respect to some NFA M, is a set $\mathcal{C}$ of states that is maximal with the property that there is a walk in M between any pair of states in $\mathcal{C}$. The component $\mathcal{C}$ is called *nontrivial* if there is at least one transition between two states in $\mathcal{C}$. For the sake of simplicity, we shall say that a component $\mathcal{C}$ '*contains*' a transition (or a walk) to mean that the NFA in which $\mathcal{C}$ exists contains that transition (or walk) with all states involved belonging to $\mathcal{C}$.

3 Characterizing Exponential Density

In this section we consider the following problem.

(P0) Given a regular language L, decide whether L is of exponential density.

In [9] (see also [10]) the author gives a very simple criterion for testing this property for a regular language L: it has exponential density if and only if any trim deterministic automaton accepting L has a state that belongs to two different cycles. Here we consider the case where the language is given via a nondeterministic automaton. We show that L has an exponential density if and only if any trim nondeterministic automaton accepting L has a SCC containing two walks of the same length, starting at the same state, and whose labels are different. For example, in Fig. 1, if $\sigma = a$, then the SCC has two walks from p to 3 of length 6 with different labels:

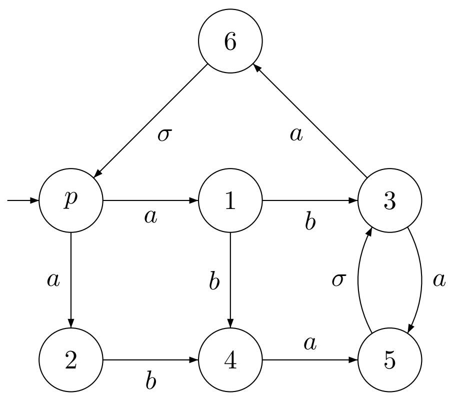


Figure 1. A strongly connected component $\mathcal{C}$ with a chosen state p, which is used as a start state of walks in $\mathcal{C}$.

$$(p, a, 1, b, 3, a, 6, \sigma, p, a, 1, b, 3) \text{ and } (p, a, 2, b, 4, a, 5, \sigma, 3, a, 5, \sigma, 3).$$

Using a 'direct' algorithm, our test can be performed in quadratic time. Note that, when the automaton is deterministic, our test is equivalent to whether a SCC contains a fork state (this is of course equivalent to the test of [9]) and can be performed in linear time. The rest of this section deals with the formalities of the above statements.

Theorem 1. *Let L be a regular language. The following statements are equivalent.*

EXP: L has exponential density.
BL2: There are two words x, y and a two-element code C whose two words have equal lengths such that $xC^*y \subseteq L$.
SCC: For every trim NFA A accepting L, there exists a strongly connected component in A containing two walks of the same length, starting at the same state, and whose labels are different.

Proof. We prove the following sequence of statements:
BL2 $\to$ EXP, SCC $\to$ BL2, EXP $\to$ SCC.

Part BL2 $\to$ EXP: Let $C = \{z_1, z_2\}$ and consider, for every $n \geq 0$, all words in L of length $|x| + |y| + \ell n$, where ℓ is the length of z_1 and z_2. As $xC^ny \subseteq L$, there are at least 2^n such words and, therefore, L must have exponential density.

Part SCC $\to$ BL2: Assume there is a state p in some SCC $\mathcal{C}$, and two walks in $\mathcal{C}$ starting at p, ending at some states q_1 and q_2, and having two *different* labels u_1, u_2 of the same length. Then, there must be two walks in $\mathcal{C}$, one from q_1 to p and the second from q_2 to p with some labels v_1, v_2, respectively. Then there are two closed walks in $\mathcal{C}$ with labels u_1v_1 and u_2v_2. Moreover, it follows that there are two closed walks in $\mathcal{C}$ with labels $z_1 = u_1v_1u_2v_2$ and $z_2 = u_2v_2u_1v_1$, which are different and of the same length, say ℓ. As the NFA A is trim, there are two paths, one from the start state to p with some label x, and the other from p to a final state with some label y. Let $C = \{z_1, z_2\}$. Then it follows that

$$xC^*y \subseteq L.$$

Part EXP $\to$ SCC: We use contraposition by assuming the negation of SCC and showing that the density of $L(A)$ is polynomially upper-bounded. So assume that in every SCC of A, any two walks starting at the same state and having the same length must have equal labels. This implies that there is no state having two outgoing transitions with different labels. First we have the following claim.

Claim 1: The assumption implies that, in every SCC $\mathcal{C}$, for every $n \geq 0$, there are at most $|\mathcal{C}|$ distinct walk labels of length n.

To see this, we first note that the claim is obvious if $\mathcal{C}$ is trivial. If $\mathcal{C}$ is nontrivial, then for every state q in $\mathcal{C}$ and any integer $n \geq 0$, there is at least one walk in $\mathcal{C}$ of length n starting at state q. At the same time, the assumption implies that, for every state q in $\mathcal{C}$ and integer $n \geq 0$, there is at most one walk label of length n. Thus, for every state q in $\mathcal{C}$ and any $n \geq 0$, there is exactly one walk label of length n starting at q, and the claim follows easily from this observation.

Next we show that the density of $L(A)$ is polynomially upper-bounded using induction on k, where k is the number of SCCs in A. For $k = 1$ this follows immediately from Claim 1. Assume the statement holds when A has at most t SCCs, for some $t \geq 1$, and consider the case where A has $k = t + 1$ SCCs. As $k \geq 2$, there must be a strongly connected component $\mathcal{D}$ with no transitions going out of $\mathcal{D}$—also, as A is trim, $\mathcal{D}$ cannot contain the start state. Consider the set L_n of all words of length n accepted by A. We shall show that $|L_n|$ is of order $O(n^\alpha)$, where α is a constant integer independent of n. First note that

$$L_n = M_n \cup K_n,$$

where M_n is the set of words of length n accepted using walks containing no state in $\mathcal{D}$, and K_n is the set of words of length n accepted using walks ending in $\mathcal{D}$. Let $\bar{\mathcal{D}}$ be the set of states not in $\mathcal{D}$, and let $q_1, \ldots, q_r$ be all states in $\bar{\mathcal{D}}$ having transitions going into $\mathcal{D}$. Let $N_1, \ldots, N_r$ be the languages accepted by the part of A that involves no states from $\mathcal{D}$ and has as final states $\{q_1\}, \ldots, \{q_r\}$, respectively. Let $N_1', \ldots, N_r'$ be the languages accepted starting, respectively, from the states $q_1, \ldots, q_r$ and then using only states in $\mathcal{D}$, where the final states of A that are in $\mathcal{D}$ are used as final states. Then, it follows that

$$K_n = (N_1N_1' \cup \cdots \cup N_rN_r') \cap \Sigma^n$$

and then

$$L_n = M_n \cup (N_1N_1' \cap \Sigma^n) \cup \cdots \cup (N_rN_r' \cap \Sigma^n).$$

By the induction hypothesis, $|M_n| = O(n^c)$, for some constant c. Now for each term $N_iN_i' \cap \Sigma^n$ we have

$$N_iN_i' \cap \Sigma^n = \bigcup_{j=0}^{n} (N_i \cap \Sigma^j)(N_i' \cap \Sigma^{n-j}).$$

Again, as N_i is accepted by an NFA having at most t SCCs we have that $|N_i \cap \Sigma^j| = O(j^{c_i})$, for some constant c_i. Also, by Claim 1, $|N_i' \cap \Sigma^{n-j}| = O(1)$. With these observations, it follows that

$$|N_iN_i' \cap \Sigma^n| = O((n+1) \times n^{c_i}) = O(n^{1+c_i}) \text{ and } |L_n| = O(n^\alpha),$$

where $\alpha = \max(c, 1 + c_1, \ldots, 1 + c_r)$ and, therefore, $L(A)$ is polynomially upper-bounded.

Now we use the previous theorem to devise a 'direct' quadratic time algorithm for deciding the density type of a given NFA language. We shall use a folklore *product construction* for labeled graphs. In particular, for any directed labeled graph $G = (V, E)$, the graph G^2 has vertices all pairs in $V \times V$ and arcs all triples of the form $((p_1, p_2), (a_1, a_2), (q_1, q_2))$ such that (p_1, a_1, q_1) and (p_2, a_2, q_2) are arcs in E. It is evident that for any walk in G^2 there are two corresponding walks in G of the same length and, conversely, for any two walks in G of the same length there is a corresponding walk in G^2. For a walk P in G^2, the *first (resp. second) corresponding walk* is made simply by the sequence of arcs formed by the first (resp. second) SCCs in the sequence of arcs in P. Thus, if

$$P = ((s_0, t_0), (a_1, b_1), (s_1, t_1), \ldots, (a_n, b_n), (s_n, t_n)),$$

then the first corresponding walk is $(s_0, a_1, s_1, \ldots, a_n, s_n)$.

Corollary 2. *There is a quadratic time algorithm for deciding the density type of a given regular language.*

Proof. The required decision algorithm is as follows.

```
algorithm ExpDensityQT($p$)
01. Make the NFA $A$ trim
02. Compute the SCCs of $A$
03. FOUND = false
04. for each SCC $G$ and while not FOUND
        05. Compute $G^2$
        06. Compute the set $Q_1$ of vertices $(p_1, p_2)$ in $G^2$ such that
                there is an arc $((p_1, p_2), (a_1, a_2), (q_1, q_2))$ with $a_1 \neq a_2$
        07. Compute the set $Q_2$ of vertices in $G^2$ of the form $(t, t)$
        08. if (there is a walk from $Q_2$ to $Q_1$) then FOUND = true
09. if (FOUND) return TRUE, else return FALSE
```

For the correctness of the algorithm we note that, at the last step, FOUND is true if and only if condition SCC of Lemma 1 is true. Indeed, if there is a walk in G^2 from some (t, t) to some $(p_1, p_2) \in Q_1$ then there is also a walk from (t, t) to some (q_1, q_2) where the last arc in the walk is $((p_1, p_2), (a_1, a_2), (q_1, q_2))$ with $a_1 \neq a_2$; hence, there must be two equal length walks in G starting at t and having different labels. Conversely, if there are two walks in G of the same length, starting at some state t and having different labels, then there are also two such walks differing on their last symbols, which implies that the algorithm will set FOUND to true when it processes the SCC G.

For the time complexity of the algorithm, first we note that Step 1 can be performed in linear time and then Step 2 also in linear time [4]. Now let n be the size of A, let k be the number of strongly connected components in the trimmed A, and let n_i be the size of the SCC i. Then $n_1 + \cdots + n_k = O(n)$. The i-th iteration of the loop requires time $O(n_i^2)$ to construct the product of the i-th SCC, which is of size $O(n_i^2)$, and then the next two steps are linear with respect to n_i^2. Also linear is the last step in the loop via a breadth-first search algorithm. So in the worst case the algorithm requires time $O(n_1^2 + \cdots + n_k^2)$.

4 Deciding the density type in linear time: the algorithm

In this section we consider a fixed SCC $\mathcal{C}$ (of some NFA) containing an arbitrarily chosen state p, which we consider *fixed*, and we show that the question of exponential density for $\mathcal{C}$ can be decided in linear time. We use the following terminology—see Fig. 1 and the example 3 further below.

- $\gcd(\mathcal{C})$ denotes the *greatest common divisor* of the lengths of all cycles in $\mathcal{C}$.
- We say that a state q in $\mathcal{C}$ *occurs at level* i, for some $i \in \mathbb{N}_0$ (when starting at state p), if there is a walk of length i from p to q.
- For each $i \in \mathbb{N}$, $A_p(i)$ denotes the *set of symbols at level* i, that is, all symbols σ such that there is a transition (q, σ, r) in $\mathcal{C}$ and state q occurs at level $i - 1$.

```
algorithm BFS($p$)
01. for each state $q$, set $\mathrm{LEV}_p(q) = ?$
02. for each $i \in \{1, \dots, N\}$, set $\mathbf{b}_p(i) = ?$
   (Note: $N$ = the number of states in the SCC)
03. Initialize a queue $Q$ to consist of $p$
04. set $\mathrm{LEV}_p(p) = 0$
05. while ($Q$ is not empty)
      06. remove $q$, the first state in $Q$
      07. for each transition $(q, \sigma, r)$
            08. set $j = \mathrm{LEV}_p(q)$
            09. if $\mathbf{b}_p(j+1) \neq ?$ and $\mathbf{b}_p(j+1) \neq \sigma$, return $\lambda$
            10. set $\mathbf{b}_p(j+1) = \sigma$
            11. if ($\mathrm{LEV}_p(r) = ?$)
                  set $\mathrm{LEV}_p(r) = j + 1$
                  append $r$ to $Q$
12. Let $k$ be the last index such that $\mathbf{b}_p(k) \neq ?$
13. return the word $\mathbf{b}_p(1) \cdots \mathbf{b}_p(k)$
```

Figure 2. This algorithm is a method of an object $\mathcal{C}$ of type "strongly connected component" containing a state p. The algorithm adds each state to the queue exactly once, and processes all transitions going out of that state. For each state q, $\mathrm{LEV}_p(q)$ is the first level at which q is encountered—this is given initially the special value '?' indicating that q has not yet been encountered. Each $\mathbf{b}_p(i)$ is the symbol found at level i (starting from state p at level 0). The algorithm returns the empty word λ if it finds a level i at which two different symbols occur, or it returns the word made by concatenating the unique symbols found at all the levels visited.

By Theorem 1, $\mathcal{C}$ has exponential density if and only if there is a level i such that $A_p(i)$ contains more than one symbol. To test this condition, we first use the breadth first search algorithm BFS(p) shown in Fig. 2. The expressions $\mathrm{LEV}_p(q)$ and $\mathbf{b}_p(i)$ are explained in that figure. In particular, $\mathrm{LEV}_p(q)$ is the length of the shortest path from p to q, and $\mathbf{b}_p(i)$ is the symbol at level i, as found by BFS(p). Then,

$$\mathbf{b}_p(i) \in A_p(i).$$

We shall show (see Theorem 5) that there is a level i_0 such that $A_p(i_0)$ contains more than one symbol, if and only if, either that level is found by BFS(p), or the word $\mathbf{b}_p$ is *not* periodic with a period of length $\gcd(\mathcal{C})$. This is the main idea for the algorithm deciding exponential density in linear time. This algorithm is shown in Fig 3.

```
algorithm ExpDensity($p$)
1. Let $\mathbf{b}_p$ = BFS($p$)
2. if ($\mathbf{b}_p = \lambda$) return TRUE
3. Let $g$ = the gcd of the cycles in the SCC
4. Let $v = \mathbf{b}_p(1) \cdots \mathbf{b}_p(g)$
5. if ($\mathbf{b}_p \notin$ Prefix($v^*$)) return TRUE
6. else return FALSE
```

Figure 3. This linear-time algorithm is a method of an object $\mathcal{C}$ of type "strongly connected component" containing a state p.

Example 3. For the SCC $\mathcal{C}$ in Fig. 1, we have that $\gcd(\mathcal{C}) = 2$. The state 4 occurs at levels $2, 8, 10, 12, \ldots$. The set $A_p(4)$ consists of σ. The algorithm BFS(p) will find that

$$\mathbf{b}_p(1) = \mathbf{b}_p(3) = a, \quad \mathbf{b}_p(2) = b, \quad \mathbf{b}_p(4) = \sigma.$$

If $\sigma = a$, then $A_p(6) = \{a, b\}$ and the density is exponential. In this case, the algorithm in Fig. 3 computes $\mathbf{b}_p = abaa$, $\gcd(\mathcal{C}) = 2$, $\mathbf{b}_p(1)\mathbf{b}_p(2) = ab$, but $abaa \notin \text{Prefix}((ab)^*)$. On the other hand, if $\sigma = b$, then the density is not exponential.

Time complexity. The algorithm in Fig. 3 runs in linear time. Indeed, BFS(p) is a linear time algorithm. The gcd g in Step 3 can be computed in linear time [1,5]. Finally, testing whether $\mathbf{b}_p \in \text{Prefix}(v^*)$ in Step 5, can also be done in linear time, as the length of $\mathbf{b}_p$ is always less than the number of states in the SCC.

5 Correctness

In this section we establish the correctness of the linear-time algorithm—see Theorem 5.

Notation. For any SCC $\mathcal{C}$ (of some NFA) containing a state p, we define the following predicates and infinite word.

- ($\mathbf{U}_\omega$): For all $i \in \mathbb{N}$: $|A_p(i)| = 1$.
- ($\mathbf{U}_{\text{bfs}}$): BFS(p) returns $\mathbf{b}_p \neq \lambda$.
- If ($\mathbf{U}_\omega$) holds, then we define $\mathbf{a}_p$ to be the infinite word made by the symbols in $A_p(1), A_p(2), \ldots$.

■

Example 4. In Fig. 1, ($\mathbf{U}_\omega$) holds if $\sigma = b$. In this case, $\mathbf{a}_p = (ab)^\omega$.

Theorem 5. *The linear time algorithm in Fig. 3 decides correctly the density type of a given SCC, that is,*

$$\neg(\mathbf{U}_\omega) \quad \leftrightarrow \quad \neg(\mathbf{U}_{\text{bfs}}) \vee \mathbf{b}_p \notin \text{Prefix}((\mathbf{b}_p(1) \cdots \mathbf{b}_p(g))^*),$$

or equivalently,

$$(\mathbf{U}_\omega) \quad \leftrightarrow \quad (\mathbf{U}_{\text{bfs}}) \wedge \mathbf{b}_p \in \text{Prefix}((\mathbf{b}_p(1) \cdots \mathbf{b}_p(g))^*),$$

where $g = \gcd(\mathcal{C})$.

How the correctness proof is presented. The 'if' part is shown in Lemma 8. This requires the first two statements of Lemma 7 about lengths of walks in $\mathcal{C}$. The 'only if' part is shown in Lemma 10. This requires the last statement of Lemma 7 and Lemma 9, which is a simple generalization of the theorem of Fine and Wilf [2]. Next we give a notation on walks, paths and cycles in a SCC, which helps make the presentation more rigorous.

Notation. For any states p, q in $\mathcal{C}$, we use the following notation.

- $[\mathcal{C}]_*^{p\to q}$: the set of all *walks* in $\mathcal{C}$ from p to q.
- $[\mathcal{C}]_*^{p\to p}$: the set of all *closed walks* in $\mathcal{C}$ starting at state p.
- $[\mathcal{C}]_0^{p\to q}$: the set of all *paths* in $\mathcal{C}$ from p to q.
- $[\mathcal{C}]_{\min}^{p\to q}$: the set of all *shortest paths* in $\mathcal{C}$ from p to q.
- $[\mathcal{C}]_0^{q\to q}$: the set of all *cycles* in $\mathcal{C}$ from q to q.
- $[\mathcal{C}]_1^{p\to p}$: the set of all *single closed walks* in $\mathcal{C}$ starting at state p. These walks start at p and end at p and they contain exactly one cycle starting at some state $q \neq p$—see the example below.

■

Example 6. Consider the SCC in Fig. 1. The cycle $(5, \sigma, 3, a, 5)$ belongs to $[\mathcal{C}]_*^{5\to 5}$ and $(p, a, 1, b, 3, a, 6, \sigma, p)$ belongs to $[\mathcal{C}]_*^{p\to p}$. The closed walk

$$\varphi = (p, a, 1, b, 3, a, 5, \sigma, 3, a, 6, \sigma, p)$$

is a single closed walk that belongs to $[\mathcal{C}]_1^{p\to p}$. Using the notation in the proof of Lemma 7.3, φ contains exactly one cycle $\varphi'' = (3, a, 5, \sigma, 3)$. If φ'' is removed from φ, then the cycle $\varphi' = (p, a, 1, b, 3, a, 6, \sigma, p)$ is obtained.

Lemma 7. *Let $\mathcal{C}$ be any SCC (of some NFA) containing the states p and q.*

1. *The lengths of all closed walks in $[\mathcal{C}]_*^{p\to p}$ are zero modulo* $\gcd(\mathcal{C})$*, that is, for every closed walk $\varphi \in [\mathcal{C}]_*^{p\to p}$, $|\varphi| \equiv 0 \pmod{\gcd(\mathcal{C})}$.*
2. *The lengths of all walks in $[\mathcal{C}]_*^{p\to q}$ are equivalent modulo* $\gcd(\mathcal{C})$*, that is, for all walks $\varphi, \psi \in [\mathcal{C}]_*^{p\to q}$, $|\varphi| \equiv |\psi| \pmod{\gcd(\mathcal{C})}$.*
3. *The greatest common divisor of the lengths of all cycles and single closed walks starting at p is equal to* $\gcd(\mathcal{C})$*, that is,*

$$\gcd\{|\varphi| : \varphi \in [\mathcal{C}]_0^{p\to p} \cup [\mathcal{C}]_1^{p\to p}\} \;=\; \gcd(\mathcal{C}).$$

Proof. For the first statement, let ψ be any cycle starting at p. By definition of $\gcd(\mathcal{C})$, $|\psi| \equiv 0 \pmod{\gcd(\mathcal{C})}$. Now consider any closed walk $\varphi \in [\mathcal{C}]_*^{p\to p}$. If φ is a cycle, then the claim holds. Else, φ contains at least one cycle. If we remove each cycle occurring in φ, then we shall obtain a cycle ψ such that $|\psi| \equiv |\varphi| \pmod{\gcd(\mathcal{C})}$, as each cycle has a length that is a multiple of $\gcd(\mathcal{C})$. Hence, $|\varphi| \equiv 0 \pmod{\gcd(\mathcal{C})}$, as required.

For the second statement, it is sufficient to show that, for every $\varphi \in [\mathcal{C}]_*^{p\to q}$, we have $|\varphi| \equiv |\psi| \pmod{\gcd(\mathcal{C})}$, where ψ is any shortest path from p to q. Let θ be any shortest path from q to p, and let φ' and ψ' be the closed walks obtained by concatenating the paths φ, θ and ψ, θ (respectively). Then, $|\varphi| - |\psi| = |\varphi'| - |\psi'|$, which is 0 modulo $\gcd(\mathcal{C})$, by the first statement.

For the third statement, we first define, for each $\varphi \in [\mathcal{C}]_1^{p\to p}$, two cycles $\varphi' \in [\mathcal{C}]_0^{p\to p}$ and $\varphi'' \in [\mathcal{C}]_0^{q\to q}$, where q is the only state, other than p, that appears twice in φ. The cycle φ'' is simply the cycle occurring inside φ and the cycle φ' is produced when φ''

is removed from φ—see Example 6. Thus, $|\varphi| = |\varphi'| + |\varphi''|$. Now let $\{\varphi_i\}_{i=1}^m$ be an enumeration of all single closed walks in $[\mathcal{C}]_1^{p\to p}$. Each φ_i contains a cycle φ_i''. In fact, by definition of $[\mathcal{C}]_1^{p\to p}$, the set $\{\varphi_i''\}_{i=1}^m$ consists of all cycles in $\mathcal{C}$ starting at a state other than p. Thus,

$$\gcd(\mathcal{C}) = \gcd(\{|\varphi_i''|\}_{i=1}^m \cup X),$$

where $X = \{|\psi| : \psi \in [\mathcal{C}]_0^{p\to p}\}$. Now let $Y = \{|\varphi| : \varphi \in [\mathcal{C}]_1^{p\to p}\}$. Using basic properties of the gcd function [7], we have that

$$\gcd(X \cup Y) = \gcd(\gcd(X), \gcd(Y)) = \gcd(\gcd(X), \gcd(Y - \{|\varphi_1|\}), |\varphi_1| - |\varphi_1'|)\},$$

as $|\varphi_1'| \in X$. Thus,

$$\gcd(X \cup Y) = \gcd(\gcd(X), \gcd(Y - \{|\varphi_1|\}), |\varphi_1''|).$$

This process can be repeated another $m-1$ times to obtain that

$$\gcd(X \cup Y) = \gcd(\gcd(X), \emptyset, \gcd\{|\varphi_i''|\}_{i=1}^m) = \gcd(X \cup \{|\varphi_i''|\}_{i=1}^m),$$

as required.

Lemma 8. *Assume that, in the BFS(p) algorithm,* $(\mathbf{U}_{\text{bfs}})$ *holds and we have* $\mathbf{b}_p \in \text{Prefix}((\mathbf{b}_p(1)\cdots\mathbf{b}_p(g))^*)$, *where* $g = \gcd(\mathcal{C})$. *Then, the following statements hold true.*

1. *For any states* q *and* q' *and shortest paths* $\psi \in [\mathcal{C}]_{\text{min}}^{p\to q}$ *and* $\psi' \in [\mathcal{C}]_{\text{min}}^{p\to q'}$, *if* ψ *and* ψ' *are of different lengths and there are transitions* (q,σ,r) *and* (q',σ',r') *with* $\sigma \neq \sigma'$, *then* $|\psi| \not\equiv |\psi'| \pmod g$
2. *The predicate* $(\mathbf{U}_\omega)$ *holds.*

Proof. For the first statement, as ψ and ψ' are shortest paths from p, BFS(p) assigns the levels $|\psi|, |\psi'|$ to $\text{LEV}_p(q), \text{LEV}_p(q')$, respectively. Also, when q and q' are removed from the queue, the symbols σ and σ' are assigned to $\mathbf{b}_p(|\psi|+1)$ and $\mathbf{b}_p(|\psi'|+1)$, respectively. Finally, as $\mathbf{b}_p \in \text{Prefix}((\mathbf{b}_p(1)\cdots\mathbf{b}_p(g))^*)$ and $\sigma \neq \sigma'$, the lengths $|\psi|$ and $|\psi'|$ cannot be equivalent $\pmod g$.

For the second statement, assume for the sake of contradiction that there is a level ℓ and two different symbols σ, σ' in $A_p(\ell)$. Then, there are two transitions of the form (q,σ,r) and (q',σ',r') such that q,q' are at level $\ell-1$. Moreover, there are two walks φ and φ' of length $\ell-1$ in $[\mathcal{C}]_*^{p\to q}$ and $[\mathcal{C}]_*^{p\to q'}$, respectively. Let ψ,ψ' be the shortest paths of BFS(p) to q,q' (respectively). By Lemma 7.2, we have

$$|\varphi| \equiv |\psi| \pmod g \ \text{ and } \ |\varphi'| \equiv |\psi'| \pmod g$$

On the other hand, the first statement implies $|\psi| \not\equiv |\psi'| \pmod g$. Therefore, $|\varphi| \not\equiv |\varphi'| \pmod g$, which contradicts $|\varphi| = |\varphi'| = \ell - 1$. Hence $(\mathbf{U}_\omega)$ holds.

Lemma 9. *For any positive integer* m *and for any words* $u_1,\ldots,u_m$,

$$\textit{if} \ \ u_1^\omega = \cdots = u_m^\omega, \quad \textit{then} \ \ u_1,\ldots,u_m \in u^*,$$

for some word u *of length* $\gcd\{|u_1|,\ldots,|u_m|\}$.

Proof. We use induction on m. The base case of $m = 1$ is trivial. Assume the claim holds for some $m \geq 1$, and consider

$$u_1^\omega = \cdots = u_m^\omega = u_{m+1}^\omega.$$

By induction hypothesis, there is a word u such that $u_i \in u^*$, for $i \in \{1, \ldots, m\}$, and $|u| = \gcd\{|u_1|, \ldots, |u_m|\}$. As $u_m^\omega = u^\omega = u_{m+1}^\omega$, the theorem of Fine and Wilf [2] implies that $u, u_{m+1} \in v^*$, for some word v of length $\gcd\{|u|, |u_{m+1}|\}$, which is equal to $\gcd\{|u_1|, \ldots, |u_m|, |u_{m+1}|\}$, by the properties of the function gcd [7].

Lemma 10. *Let $g = \gcd(\mathcal{C})$. The following statements hold.*

1. $(\mathbf{U}_\omega) \;\rightarrow\; (\mathbf{U}_{\text{bfs}}) \;\wedge\; (\mathbf{b}_p \in \text{Prefix}(\mathbf{a}_p))$
2. $(\mathbf{U}_\omega) \;\rightarrow\; \mathbf{a}_p = (\mathbf{b}_p(1) \cdots \mathbf{b}_p(g))^\omega$
3. $(\mathbf{U}_\omega) \;\rightarrow\; \mathbf{b}_p \in \text{Prefix}((\mathbf{b}_p(1) \cdots \mathbf{b}_p(g))^*)$

Proof. For the first statement, if $(\mathbf{U}_\omega)$ holds, then BFS(p) cannot find at Step 09 a level with two symbols; hence, $\mathbf{b}_p \neq \lambda$. Also, $(\mathbf{U}_\omega)$ implies that $\mathbf{a}_p$ is well defined and, as $\mathbf{b}_p(i) \in A_p(i)$, for all symbols in $\mathbf{b}_p$, we have that $\mathbf{b}_p$ is a prefix of $\mathbf{a}_p$.

For the second statement, $(\mathbf{U}_\omega)$ implies that there is exactly one symbol at level i, in any path of length i from p. Then, it follows that $\mathbf{a}_p$ is well defined and $\mathbf{a}_p = (\bar{\mathrm{w}}_\varphi)^\omega$, for every closed walk φ starting at p, where $\bar{\mathrm{w}}_\varphi$ denotes the label of the path φ. In particular, $\mathbf{a}_p = (\bar{\mathrm{w}}_\varphi)^\omega$, for every $\varphi \in [\mathcal{C}]_0^{p\to p} \cup [\mathcal{C}]_1^{p\to p}$. Then, Lemma 9 and Lemma 7.3 imply that $\bar{\mathrm{w}}_\varphi \in u^*$ for some word u of length g. Thus, $\mathbf{a}_p = u^\omega$ and, as $\mathbf{b}_p$ is a prefix of $\mathbf{a}_p$, we have that $u = \mathbf{b}_p(1) \cdots \mathbf{b}_p(g)$, as required.

The third statement follows from the previous one and the fact that $\mathbf{b}_p$ is a prefix of $\mathbf{a}_p$.

6 Implementation and testing

We have implemented both the quadratic and linear time algorithms using the FAdo library for automata [6], which is well maintained and provides several useful tools for manipulating automata. In doing so, we have also implemented a Python method

```
stronglyConnectedComponents(A)
```

receiving a parameter `A`, which is an `NFA` object with respect to FAdo, and returns a list of the SCCs of `A`, where each SCC is a list of states in `A`.

For computing the quantity $\gcd(\mathcal{C})$, one can use the depth first search (DFS) based algorithm in [1], or the breadth first search (BFS) based algorithm in [5]—in fact [5] discusses both the BFS and DFS based algorithms. In our implementation, we have adjusted the BFS algorithm in Fig 2 to compute the required $\gcd(\mathcal{C})$, in addition to the word $\mathbf{b}_p(1) \cdots \mathbf{b}_p(k)$.

Our implementation confirms the theoretical result that indeed the linear time algorithm is much faster. We have used as test cases four sequences of SCCs, which are described in Fig. 4. Each of these SCCs is implemented as an object of type `NFA` and is constructed using `NFA` methods such as

`addState()` and `addTransition()`.

When the answer is FALSE, the linear time algorithm will perform a complete BFS and then all tests in Fig. 3 to find out that $\mathbf{b}_p$ is nonempty and periodic with a period of length g. When the answer is TRUE, it is possible that the linear time

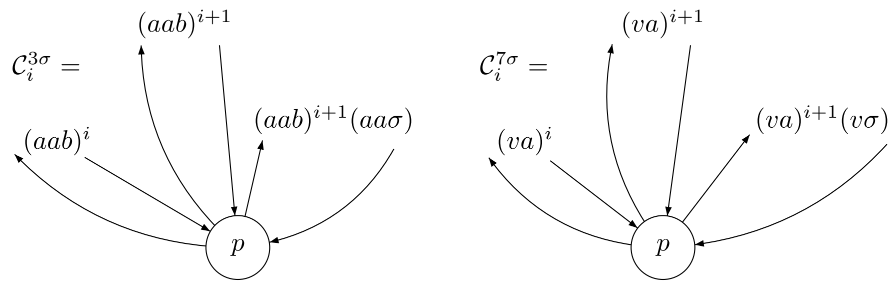


Figure 4. Four sequences of SCCs. On the left, for $\sigma = a, b$, we have the SCCs $\mathcal{C}_i^{3a}$ and $\mathcal{C}_i^{3b}$. On the right, for $\sigma = a, b$, we have the SCCs $\mathcal{C}_i^{7a}$ and $\mathcal{C}_i^{7b}$ with $v = aabbab$. Each SCC has three cycles starting at p with labels as shown in the figure. For example, for each $i \in \mathbb{N}$, the SCC $\mathcal{C}_i^{3a}$ has three cycles with labels $(aab)^i$, $(aab)^{i+1}$, $(aab)^{i+1}(aaa)$. If $\sigma = a$ the density is exponential.

algorithm will finish quickly when BFS(p) finds in Step 09 two different symbols occurring at the same level. However, we have chosen the particular test SCCs such that when the answer is TRUE, the linear time algorithm will still perform a complete BFS and then find out that $\mathbf{b}_p$ is non-periodic only when it scans the last symbol σ of the longest cycle in the SCC.

Each of the four figures in the Appendix concerns one of the two algorithms and a certain sequence $\mathcal{C}_i^{x\sigma}$ of SCCs, and shows a graph with the execution time of the algorithm $T(i)$ vs the value of the parameter i.

References

1. E. Arkin, C. Papadimitriou, and M. Yannakakis: *Modularity of cycles and paths in graphs.* Journal of the ACM, 38 1991, pp. 255–274.
2. C. Choffrut and J. Karhumäki: *Combinatorics on words*, in Rozenberg and Salomaa [8], pp. 329–438.
3. B. Cui and S. Konstantinidis: *DNA coding using the subword closure operation*, in Proceedings of 13th Inter. Meeting on DNA Computing 2007, vol. 4848 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2008, pp. 284–289.
4. S. Dasgupta, C. Papadimitriou, and U. Vazirani: *Algorithms*, McGraw-Hill, 2006.
5. J. Jarvis and D. Shier: *Graph-theoretic analysis of finite Markov chains*, in Applied Mathematical Modeling: a multidisciplinary approach, Chapman & Hall / CRC press, 2000, pp. 271–289.
6. R. Reis and N. Moreira: *FAdo: Tools for language models manipulation*, `http://www.dcc.fc.up.pt/~rvr/FAdoDoc/index.html`. Accessed in March 2013.
7. K. Rosen: *Greatest common divisors*, in Handbook of Discrete and Combinatorial Mathematics, CRC Press, Berlin, 2000.
8. G. Rozenberg and A. Salomaa, eds., *Handbook of Formal Languages, Vol. I*, Springer-Verlag, 1997.
9. A. Shur: *Combinatorial complexity of rational languages.* Discr. Anal. and Oper. Research, Ser. 1, 12(2) 2005, pp. 78–99.
10. A. Shur: *Factorial languages of low combinatorial complexity*, in In Proceedings of 19th Inter. Conf. on Developments in Language Theory. Lecture Notes in Computer Science, Vol. 4036, Springer-Verlag, Berlin, 2006, pp. 397–407.
11. A. Szilard, S. Yu, K. Zhang, and J. Shallit: *Characterizing regular languages with polynomial densities*, in Proceedings of 7th Inter. Symposium on Mathematical Foundation of Computer Science, vol. 629 of Lecture Notes in Computer Science, Springer-Verlag, London, UK, 1992, pp. 494–503.
12. S. Yu: *Regular languages*, in Rozenberg and Salomaa [8], pp. 41–110.

Appendix

This appendix consists of four graphs showing execution times of the quadratic and linear times algorithms as explained in section 6.

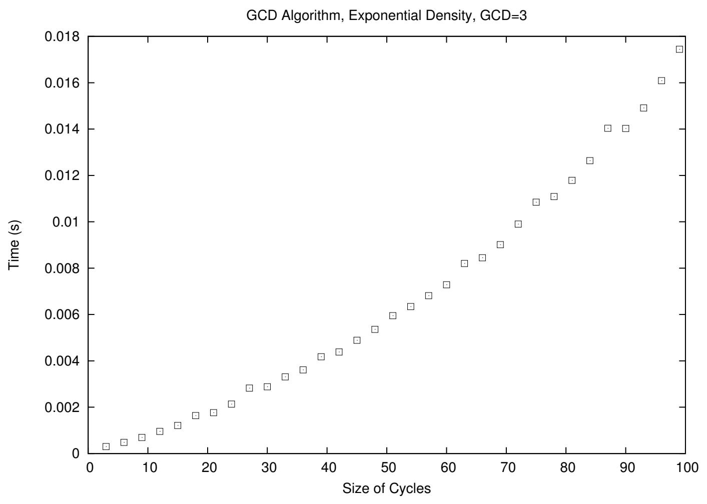


Figure 5. Execution time $T_{\rm lt}(i)$ of the *linear* time algorithm on $\mathcal{C}_i^{3a}$, for various values of i. The density type is exponential.

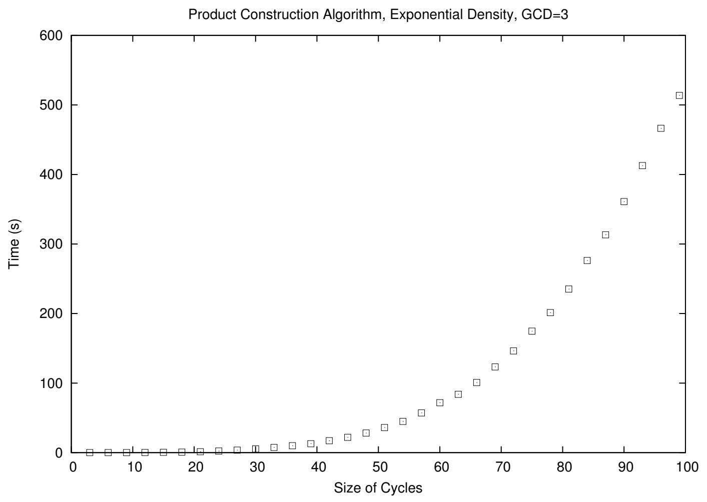


Figure 6. Execution time $T_{\rm qt}(i)$ of the *quadratic* time algorithm on $\mathcal{C}_i^{3a}$, for various values of i. The density type is exponential.

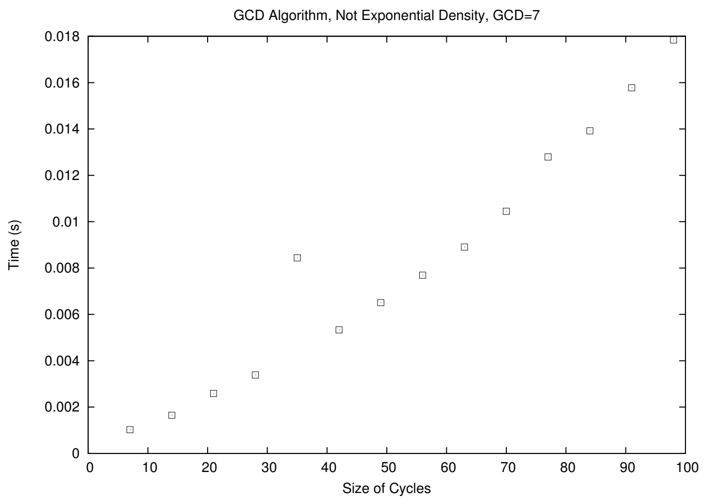


Figure 7. Execution time $T_{\mathrm{lt}}(i)$ of the *linear* time algorithm on $\mathcal{C}_i^{7a}$, for various values of i. The density type is *not* exponential.

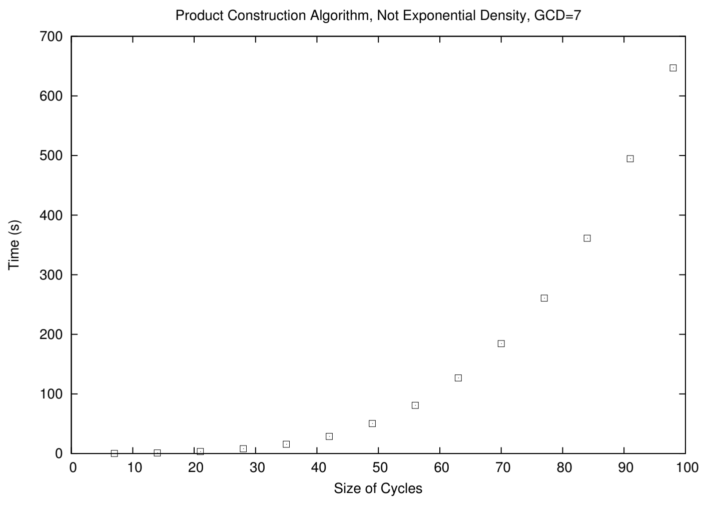


Figure 8. Execution time $T_{\mathrm{qt}}(i)$ of the *quadratic* time algorithm on $\mathcal{C}_i^{7a}$, for various values of i. The density type is *not* exponential.

On Morphisms Generating Run-Rich Strings

Kazuhiko Kusano, Kazuyuki Narisawa, and Ayumi Shinohara

Graduate School of Information Sciences, Tohoku University, Japan
{kusano@shino., narisawa@, ayumi@}ecei.tohoku.ac.jp

Abstract. A run in a string is a periodic substring which is extendable neither to the left nor to the right with the same period. Strings containing many runs are of interest. In this paper, we focus on the series of strings $\{\psi(\phi^i(\mathtt{a}))\}_{i\geq 0}$ generated by two kinds of morphisms, $\phi : \{\mathtt{a},\mathtt{b},\mathtt{c}\} \rightarrow \{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$ and $\psi : \{\mathtt{a},\mathtt{b},\mathtt{c}\} \rightarrow \{\mathtt{0},\mathtt{1}\}^*$. We reveal a simple morphism ϕ_r plays a critical role to generate run-rich strings. Combined with a morphism ψ', the strings $\{\psi'(\phi_r^i(\mathtt{a}))\}_{i\geq 0}$ achieves *exactly the same* lower bound as the current best lower bound for the maximum number $\rho(n)$ of runs in a string of length n. Moreover, combined with another morphism ψ'', the strings $\{\psi''(\phi_r^i(\mathtt{a}))\}_{i\geq 0}$ give a new lower bound for the maximum value $\sigma(n)$ of the sum of exponents of runs in a string of length n.

Keywords: run, sum of exponents, repetition, morphic word

1 Introduction

Repetitions are one of the most fundamental topics in stringology, and they are also important for practical areas, such as string processing, data compression and bioinformatics. A *run* (or *maximal repetition*) in a string is a periodic substring which is extendable neither to the left nor to the right with the same period. All repetitions in a string can be succinctly represented by runs. Strings containing many runs (we call them *run-rich strings*) are of interest to researchers. In 1999, Kolpakov and Kucherov [12] showed that the maximum number $\rho(n)$ of runs in a string of length n is $\rho(n) \leq cn$ for some constant c. Since then, a great deal of efforts have been devoted to estimate the constant c [8,17,9,18,4,3,10,16,15,19,14,13,2,5], while it is conjectured that $c < 1$. The current best upper bound for $\rho(n)/n$ is 1.029 due to Crochemore *et al.* [5] in 2011, and the current best lower bound is 0.9445757 due to Simpson [19] in 2010.

The maximum value $\sigma(n)$ of sum of exponents in runs in a string of length n is of another concern. Clearly $2\rho(n) \leq \sigma(n)$, since each exponent in a run is at least 2. The current best upper bound 4.087 and the best lower bound 2.035257 for $\sigma(n)/n$ are both given by Crochemore *et al.* [6] in 2011.

In order to provide lower bounds for $\rho(n)$ and $\sigma(n)$, various kinds of run-rich strings are shown in the literature. In 2003, Franek *et al.* [8,7] defined an ingenious run-rich strings to show a lower bound $3/(1+\sqrt{5}) = 0.9270509$ for $\rho(n)/n$. In 2008, Matsubara *et al.* [15] found a more run-rich string of length 184973 which contains 174719 runs by computer experiments, that provided a better lower bound 0.9445648. They improved it in [14] to 0.9445756 by defining a series $\{t_i\}_{i\geq 0}$ of strings. In 2010, Simpson [19] provided another series $\{s_i\}_{i\geq 0}$ of strings based on the modified Padovan words, that gives the current best lower bound 0.9445757. We note that $\{t_i\}$ also gives exactly the same bound, assuming that the recurrence formula conjectured in [14] is correct. In 2011, Crochemore *et al.* [6] showed the current best lower bound 2.035257 for $\sigma(n)/n$ by defining the strings $\{\psi_c(\phi_c^i(\mathtt{a}))\}_{i\geq 0}$ using two morphisms $\phi_c : \{\mathtt{a},\mathtt{b},\mathtt{c}\} \rightarrow \{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$ and $\psi_c : \{\mathtt{a},\mathtt{b},\mathtt{c}\} \rightarrow \{\mathtt{0},\mathtt{1}\}^*$.

$$\phi_r(\mathtt{a}) = \mathtt{abac} \qquad \phi_r(\mathtt{b}) = \mathtt{aac} \qquad \phi_r(\mathtt{c}) = \mathtt{a}$$

$$\begin{cases} h(\mathtt{a}) = \mathtt{1010010110010100101101 00} \\ h(\mathtt{b}) = \mathtt{1010010110100} \\ h(\mathtt{c}) = \mathtt{10100101} \end{cases} \qquad \begin{cases} \psi_e(\mathtt{a}) = \mathtt{101001010010} \\ \psi_e(\mathtt{b}) = \mathtt{110100} \\ \psi_e(\mathtt{c}) = \mathtt{1} \end{cases}$$

$u_i = h(\phi_r^i(\mathtt{a}))$
$\rho(u_i)/|u_i| \to 0.9445757\ (i \to \infty)$

$v_i = \psi_e(\phi_r^i(\mathtt{a}))$
$\sigma(v_{12})/|v_{12}| = 2.036982$
$\sigma\big((v_{12})^k\big)/\big|(v_{12})^k\big| \to 2.036992\ (k \to \infty)$

Figure 1. Two morphisms ϕ_r and ψ_e we discovered, and the summary of the results.

In this paper, we focus on the strings defined by the same form $\{\psi(\phi^i(\mathtt{a}))\}_{i\geq 0}$, and try to find better ones by computer experiments. We report two morphisms ϕ_r and ψ_e in Fig. 1 that we discovered. These morphisms are effective for defining run-rich strings from the following two viewpoints:

1. The strings $\{h(\phi_r^i(\mathtt{a}))\}_{i\geq 0}$ achieve *exactly the same* lower bound for $\rho(n)/n$ with the current best lower bound 0.9445757. Here, h is the morphism proposed by Simpson [19] to define the run-rich strings $\{h(p_i)\}_{i\geq 0}$ based on the *modified Padovan words* $\{p_i\}_{i\geq 0}$, and $\{h(p_i)\}_{i\geq 0}$ are the very strings that achieve the current best lower bound.
2. The strings $\{\psi_e(\phi_r^i(\mathtt{a}))\}_{i\geq 0}$ give a new lower bound 2.036992 for $\sigma(n)/n$; that is better than the current best lower bound 2.035257.

Therefore, the simple morphism ϕ_r plays a critical role to generate run-rich strings, both for the number $\rho(n)$ of runs and the sum $\sigma(n)$ of exponents of runs. Another attractive feature of ϕ_r is its simplicity, compared to the definition of the modified Padovan words.

The rest of this paper is organized as follows. In Section 2, we introduce some notations on runs. Section 3 reviews three series of strings that appeared in the literature [14,19,6], that give the current best lower bounds for $\rho(n)$ and $\sigma(n)$. We then explain in Section 4, a simple search strategy based on enumerations for finding good morphisms. In Section 5, for the strings $u_i = h(\phi_r^i(\mathtt{a}))$, we prove $\rho(u_i)/|u_i| \to 0.9445757$, that exactly equals to the current best lower bound for $\rho(n)/n$. In Section 6, we show that the lower bound for $\sigma(n)/n$ is improved to be 2.036992 by the string $\psi_e(\phi_r^{12}(\mathtt{a}))$. Section 7 concludes and discusses some future work. In Appendix, we supply some lemmas and remarks easily verified by Mathematica, for convenience.

2 Preliminaries

Let Σ be an alphabet. We denote by Σ^n the set of all strings of length n over Σ, and $|w|$ denotes the length of a string w. We denote by $w[i]$ the ith letter of w, and $w[i..j]$ is a substring $w[i]w[i+1]\cdots w[j]$ of w.

For a string w of length n and a positive integer $p \leq n$, we say that p is a *period* of w if $w[i] = w[i+p]$ holds for any $1 \leq i \leq n-p$. A string may have several periods. For instance, string abaababa has three periods 5, 7 and 8. A string w is *primitive* if w cannot be written as $w = u^k$ by any string u and any integer $k \geq 2$. A *run* (also called a maximal repetition) in a string w is an interval $[i..j]$, such that:

(1) the smallest period p of $w[i..j]$ satisfies $2p \leq j-i+1$,
(2) either $i = 1$ or $w[i-1] \neq w[i+p-1]$,

(3) either $j = n$ or $w[j+1] \neq w[j-p+1]$.

That is, *run* is a maximal repetition which is extendable neither to the left nor to the right. The (fractional) *exponent* of the run $[i..j]$ is defined as $\frac{j-i+1}{p}$. We often represent the run $[i..j]$ by a triplet $\langle i, j-i+1, p\rangle$ of the initial position, length, and the shortest period, for convenience. We denote by $Run(w)$ the set of all runs in string w. For instance, let us consider a string $w = \texttt{aabaabababab}$. It contains 4 runs; $Run(w) = \{\langle 1,2,1\rangle, \langle 4,5,1\rangle, \langle 1,7,3\rangle, \langle 5,12,2\rangle\}$. On the other hand, $\langle 1,6,3\rangle$ is not a run in w since the repetition can be extended to the right. Neither is $\langle 5,12,4\rangle$, since the smallest period of $w[5..12]$ is 2, but not 4.

We denote by $\rho(w)$ the number of runs contained in string w, and by $\sigma(w)$ the sum of exponents of all runs in string w.

Example 1. For a string $w = \texttt{aabaabaaaacaacac}$, we have

$$Run(w) = \{\langle 1,2,1\rangle, \langle 4,2,1\rangle, \langle 7,4,1\rangle, \langle 12,2,1\rangle, \langle 13,4,2\rangle, \langle 1,8,3\rangle, \langle 9,7,3\rangle\}.$$

Thus, $\rho(w) = 7$, and $\sigma(w) = \frac{2}{1} + \frac{2}{1} + \frac{4}{1} + \frac{2}{1} + \frac{4}{2} + \frac{8}{3} + \frac{7}{3} = 17$.

For a non-negative integer n, we denote by $\rho(n)$ the maximum number of runs in a string of length n, and by $\sigma(n)$ the maximum value of the sum of exponents of runs in a string of length n. That is,

$$\rho(n) = \max\{\rho(w) \mid w \in \Sigma^n\} \quad \text{and} \quad \sigma(n) = \max\{\sigma(w) \mid w \in \Sigma^n\}.$$

3 Previously Known Series of Run-Rich Strings

This section briefly reviews three series of strings containing many runs, which are defined by recursions,

The first one is due to Simpson [19], which gives the current best lower bound for the maximum number $\rho(n)$ of runs in a string of length n.

Definition 2 ([19]). *The modified Padovan words $\{p_i\}$ are defined by*

$$p_1 = \texttt{b},\quad p_2 = \texttt{a},\quad p_3 = \texttt{ac},\quad p_4 = \texttt{ba},\quad p_5 = \texttt{aca},\quad \textit{and } p_i = R(f(p_{i-5})) \textit{ for } i > 5,$$

where $R(w)$ is the reverse of w, and $f : \{\texttt{a},\texttt{b},\texttt{c}\} \to \{\texttt{a},\texttt{b},\texttt{c}\}^*$ is a morphism

$$f(\texttt{a}) = \texttt{aacab},\quad f(\texttt{b}) = \texttt{acab},\quad f(\texttt{c}) = \texttt{ac}.$$

Simpson's words $\{s_i\}$ are defined by $s_i = h(p_i)$, where $h : \{\texttt{a},\texttt{b},\texttt{c}\} \to \{\texttt{0},\texttt{1}\}^*$ is a morphism

$$\begin{aligned} h(\texttt{a}) &= \texttt{1010010110010100101101 00},\\ h(\texttt{b}) &= \texttt{1010010110100},\\ h(\texttt{c}) &= \texttt{10100101}. \end{aligned} \tag{1}$$

Theorem 3 ([19]). $\lim_{n\to\infty} \frac{\rho(n)}{n} \geq \lim_{i\to\infty} \frac{\rho(s_i)}{|s_i|} = \eta > 0.9445757,$
where η is the real root of $2693z^3 - 7714z^2 + 7379z - 2357 = 0$.

Proof. Simpson [19] proved that $\lim_{i\to\infty} \frac{\rho(s_i)}{|s_i|} = \frac{11\kappa^2+7\kappa-6}{11\kappa^2+8\kappa-6}$, where κ is the real root of $z^3 - z - 1 = 0$. We can verify $\frac{11\kappa^2+7\kappa-6}{11\kappa^2+8\kappa-6} = \eta$ easily (Lemma 16 in Appendix). □

The second one is proposed by Matsubara *et al.* [14].

Definition 4 ([14]). *Matsubara* et al.*'s words $\{t_i\}$ are defined by*

$$\begin{aligned} t_0 &= \texttt{1001010010110100101}, \\ t_1 &= \texttt{1001010010110}, \\ t_2 &= \texttt{10010100101101001010100101}, \\ t_k &= t_{k-1}\,t_{k-2} \qquad (k \bmod 3 = 0,\ k > 2), \\ t_k &= t_{k-1}\,t_{k-4} \qquad (k \bmod 3 \neq 0,\ k > 2). \end{aligned}$$

Interestingly, these strings $\{t_i\}$ give *exactly* the same lower bound as $\{s_i\}$ for $\rho(n)$.

Theorem 5 ([14][1]). $\lim_{n\to\infty} \dfrac{\rho(n)}{n} \geq \lim_{i\to\infty} \dfrac{\rho(t_i)}{|t_i|} = \eta > 0.9445757,$
where η is the real root of $2693z^3 - 7714z^2 + 7379z - 2357 = 0.$

Proof. We can verify that the value $\lim_{i\to\infty} \rho(t_i)/|t_i|$ shown in the proof of Theorem 6 in the paper [14] is *exactly* equal to η (Lemma 17 in Appendix). $\square$

The third one is introduced by Crochemore *et al.* [6], which gives the current best lower bound for the maximum value $\sigma(n)$ of the sum of exponents of runs.

Definition 6 ([6]). *Crochemore* et al.*'s words $\{c_i\}$ are defined by $c_i = \psi_c(\phi_c^i(\texttt{a}))$ using two morphisms $\phi_c : \{\texttt{a},\texttt{b},\texttt{c}\} \to \{\texttt{a},\texttt{b},\texttt{c}\}^*$ and $\psi_c : \{\texttt{a},\texttt{b},\texttt{c}\} \to \{\texttt{0},\texttt{1}\}^*$ such that*

$$\begin{aligned} &\phi_c(\texttt{a}) = \texttt{baaba}, \qquad \phi_c(\texttt{b}) = \texttt{ca}, \qquad \phi_c(\texttt{c}) = \texttt{bca}, \\ &\psi_c(\texttt{a}) = \texttt{01011}, \quad \psi_c(\texttt{b}) = \psi_c(\texttt{c}) = \texttt{01001011}. \end{aligned}$$

Theorem 7 ([6]). $\lim_{n\to\infty} \dfrac{\sigma(n)}{n} \geq \dfrac{\sigma(c_{10})}{|c_{10}|} \geq \dfrac{10599765.15}{5208071} > 2.035257.$

4 Searching for Better Morphisms

Inspired by a simple and elegant definition of Crochemore's words, we are interested in finding other series of strings defined by similar recursions, that hopefully contain more runs or larger sum of exponents.

We focus on the series $\{w_i\}$ of strings defined by $w_i = \psi(\phi^i(\texttt{a}))$ using two morphisms $\phi : \{\texttt{a},\texttt{b},\texttt{c}\} \mapsto \{\texttt{a},\texttt{b},\texttt{c}\}^*$ and $\psi : \{\texttt{a},\texttt{b},\texttt{c}\} \mapsto \{\texttt{0},\texttt{1}\}^*$, and try to find good pair of ϕ and ψ, in the sense that either $\rho(w_i)$ or $\sigma(w_i)$ is large enough.

Various approaches are possible to search for good pairs. For instance, even a simple random search might be usable. We chose the following two-phase strategy, as the search space is huge (needless to say, infinite) and we observed that inappropriate choices of ψ would never succeed to find good ϕ's.

In the first phase, we search for ϕ by fixing ψ to h defined in Eq. (1) in Definition 2. We enumerate every possible morphism ϕ in increasing order with respect to the sum $|\phi(\texttt{a})| + |\phi(\texttt{b})| + |\phi(\texttt{c})|$, and compute all runs in the string $h(\phi^i(\texttt{a}))$ whose length is reasonably long. If a good ϕ yielding many runs is found, report it. A pseudo-code is shown in Algorithm 1. At this point, we succeeded to find a good morphism ϕ_r,

[1] Strictly speaking, the general formula of $\rho(t_i)$ in the paper is derived from a recurrence formula, which is verified for $i = 0, 1, \ldots, 14$, but not formally proved.

i	$\lvert u_i \rvert$	$\rho(u_i)$	$\rho(u_i)/\lvert u_i \rvert$	i	$\lvert s_i \rvert$	$\rho(s_i)$	$\rho(s_i)/\lvert s_i \rvert$
0	**24**	**16**	**0.66666**	2	**24**	**16**	**0.66666**
1	69	56	0.81159	7	93	79	0.84946
2	218	193	0.88532	12	380	345	0.90789
3	667	616	0.92353	17	1552	1450	0.93427
4	2057	1925	0.93582	22	**6333**	**5963**	**0.94157**
5	**6333**	**5963**	**0.94157**	27	25837	24383	0.94372
6	19504	18400	0.94340	32	105405	99538	0.94433
7	60064	56711	0.94417	37	430010	406149	0.94451
8	184973	174693	0.94442	42	**1754267**	**1657007**	**0.94455**
9	569642	538041	0.94452	47	7156700	6760011	0.94457
10	**1754267**	**1657005**	**0.94455**				

Table 1. Comparison of $u_i = h\left(\phi_r^i(\mathtt{a})\right)$ with Simpson's words $s_i = h\left(p_i\right)$. Rows holding the same lengths are highlighted in **bold**, for clarity.

which achieves the same lower bound for $\rho(n)$ as the current best one. We will fully explain it in Section 5.

In the second phase, we fix ϕ to the best ϕ_r found in the first phase, and enumerate every ψ in the same way (see Algorithm 2 for a pseudo-code). We finally found a good morphism ϕ_e so that $\sigma(\psi_e(\psi_r^8(\mathtt{a})))/|\psi_e(\psi_r^8(\mathtt{a}))| = 2.03632$ clearly exceeds the current best lower bound 2.035257 for $\sigma(n)/n$. We will describe the new lower bounds in Section 6.

5 Simpler Morphism Achieving the Current Best Lower Bound for $\rho(n)$

We obtained the following morphism $\phi_r : \{\mathtt{a},\mathtt{b},\mathtt{c}\} \to \{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$,

$$\phi_r(\mathtt{a}) = \mathtt{abac}, \quad \phi_r(\mathtt{b}) = \mathtt{aac}, \quad \phi_r(\mathtt{c}) = \mathtt{a}. \tag{2}$$

Combined with the morphism h in Definition 2, we now have another good series $\{u_i\}$ of run-rich strings, defined by $u_i = h(\phi_r^i(\mathtt{a}))$. Table 1 compares $\{u_i\}$ with Simpson's words $\{s_i\}$ with respect to the length and the number of runs. While the definition of our strings $\{u_i\}$ is much simpler than that of Simpson's words $\{s_i\}$, the numbers of runs are *almost* the same; note that it is *not exactly* the same, since $|u_{10}| = |s_{42}| = 1754267$ and $\rho(u_{10}) = 1757005 < 1757007 = \rho(s_{42})$. More interestingly, however, the asymptotic value of the ratio $\rho(u_i)/|u_i|$ *exactly* coincides with that of $\rho(s_i)/|s_i|$, as we will see in Theorem 10.

We begin by obtaining a general formula representing the length $|u_i|$.

Lemma 8. *Let $L(z) = \sum_{i=0}^{\infty} |u_i| z^i$ be the ordinary generating function of the sequence $\{|u_i|\}_{i \geq 0}$ of lengths of u_i's. Then*

$$L(z) = \frac{-8z^2 - 21z - 24}{z^3 + 3z^2 + 2z - 1}.$$

Proof. Let $|w|_a$ denote the number of occurrences of a in string w. Then for any $w \in \{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$, the length $|w|$ is calculated by the sum $|w|_\mathtt{a} + |w|_\mathtt{b} + |w|_\mathtt{c}$. Let M be the *incidence matrix* (see e.g. Chapter 8.2 in [1]) of the morphism ϕ_r defined by

$$M = \begin{pmatrix} |\phi_r(\mathtt{a})|_\mathtt{a} & |\phi_r(\mathtt{b})|_\mathtt{a} & |\phi_r(\mathtt{c})|_\mathtt{a} \\ |\phi_r(\mathtt{a})|_\mathtt{b} & |\phi_r(\mathtt{b})|_\mathtt{b} & |\phi_r(\mathtt{c})|_\mathtt{b} \\ |\phi_r(\mathtt{a})|_\mathtt{c} & |\phi_r(\mathtt{b})|_\mathtt{c} & |\phi_r(\mathtt{c})|_\mathtt{c} \end{pmatrix} = \begin{pmatrix} 2\,2\,1 \\ 1\,0\,0 \\ 1\,1\,0 \end{pmatrix}.$$

Then for any string $w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^*$, it holds that

$$\begin{pmatrix} |\phi_r(w)|_{\mathtt{a}} \\ |\phi_r(w)|_{\mathtt{b}} \\ |\phi_r(w)|_{\mathtt{c}} \end{pmatrix} = M \begin{pmatrix} |w|_{\mathtt{a}} \\ |w|_{\mathtt{b}} \\ |w|_{\mathtt{c}} \end{pmatrix},$$

which induces the recurrence formula $|u_i| = 2|u_{i-1}|+3|u_{i-2}|+|u_{i-3}|$ for $i \geq 3$, since the characteristic polynomial of M is $-x^3 + 2x^2 + 3x + 1$. Taking into account the initial values $|u_0| = 24$, $|u_1| = 69$ and $|u_2| = 218$, we obtain the generating function $L(z)$ of the sequences $|u_i|$'s as we stated (see e.g. [11] for handling generating functions). See also Remark 18 in Appendix. □

Lemma 9. *Let* $R(z) = \sum_{i=0}^{\infty} \rho(u_i)\, z^i$ *be the ordinary generating function of the sequence* $\{\rho(u_i)\}_{i\geq 0}$ *of the numbers of runs in* u_i*'s. Then*

$$R(z) = \frac{-16 - 8z + 7z^2 - 5z^3 - 3z^4 - z^5 + z^6}{(1-z)^2(1+z)(-1+2z+3z^2+z^3)}.$$

Proof. By observing the sequence $\rho(u_0)$, $\rho(u_1), \ldots, \rho(u_{10})$, we found a recurrence formula would hold, as in Table 2:

$$\begin{aligned} a_{i+2} - a_i &= 25, \quad (i \geq 1), \\ a_1 = 58, &\quad a_2 = 72, \end{aligned} \tag{3}$$

where a_i is defined[2] by

$$a_i = \rho(u_{i+3}) - 2\rho(u_{i+2}) - 3\rho(u_{i+1}) - \rho(u_i)\,. \tag{4}$$

Assuming that Eq. (3) holds for any $i \geq 1$ (in this sense, the proof is incomplete yet), we can get the general term of a_i as

$$\begin{aligned} a_i &= \frac{3}{4}(-1)^i + \frac{25i}{2} + \frac{185}{4} \qquad (i \geq 1), \\ a_0 &= 46. \end{aligned}$$

Combined with Eq. (4), we get the generating function $R(z)$ of $\rho(u_i)$ as stated. See Remark 18 in Appendix. □

Theorem 10. $\displaystyle\lim_{i\to\infty} \frac{\rho(u_i)}{|u_i|} = \eta$,
where η *is the real root of* $2693z^3 - 7714z^2 + 7379z - 2357 = 0$.

Proof. By Lemma 8 and 9, we have the generating functions $L(z)$ and $R(z)$ for $|u_i|$ and $\rho(u_i)$, respectively. Lemma 19 in Appendix completes the rest. □

[2] Based on the fact that the characteristic polynomial of M is $-x^3 + 2x^2 + 3x + 1$.

i	$\rho(u_i)$	a_i	$a_{i+2}-a_i$	$a_{i+1}-a_i$
0	16	46	26	12
1	56	58	25	14
2	193	72	25	11
3	616	83	25	14
4	1925	97	25	11
5	5963	108	25	14
6	18400	122		11
7	56711	133		
8	174693			
9	538041			
10	1657005			

Table 2. Observation on the series $\{\rho(u_i)\}$ for $u_i = h\left(\phi_r^i(\mathtt{a})\right)$. If we define a_i as in Eq. (4), the difference sequence $a_{i+2}-a_i$ of order 2 seems to be a constant 25 except the initial value $a_2-a_0 = 26$. Note also that the difference sequence $a_{i+1}-a_i$ of order 1 has alternating values 14 and 11.

6 New Lower Bounds for $\sigma(n)$

In the second phase of search, we obtained the morphism $\psi_e : \{\mathtt{a},\mathtt{b},\mathtt{c}\} \to \{\mathtt{0},\mathtt{1}\}^*$,

$$\psi_e(\mathtt{a}) = \mathtt{101001010010}, \quad \psi_e(\mathtt{b}) = \mathtt{110100}, \quad \psi_e(\mathtt{c}) = \mathtt{1}.$$

Combined with the morphism ϕ_r in Eq. (2), let us define $v_i = \psi_e(\phi_r^i(\mathtt{a}))$. In this section, we will show that the strings $\{v_i\}$ give a better lower bound of the maximum sum $\sigma(n)$ of exponents of runs.

Table 3 shows the length $|v_i|$, the number $\rho(v_i)$ of runs, and the sum $\sigma(v_i)$ of exponents, together with their ratios to the length. First let us notice that the strings $\{v_i\}$ do *not* contain so many runs. In fact, we can verify $\lim_{i\to\infty} \rho(v_i)/|v_i| = 0.923118$ assuming that a similar recurrence relation as Eqs. (3), (4) holds (see Lemma 20 in Appendix for confidence), that is strictly inferior to the current best lower bound $\lim_{i\to\infty} \rho(u_i)/|u_i| = 0.9445757$.

However, on the other hand, the sum $\sigma(v_i)$ of exponents of runs in the string v_i is very large. Figure 2 illustrates the comparison of our words $v_i = \psi_e\left(\phi_r^i(\mathtt{a})\right)$ with Crochemore et al.'s words $c_i = \psi_c\left(\phi_c^i(\mathtt{a})\right)$. Apparently, $\sigma(v_i)$ for $i \geq 8$ exceeds the current best lower bound $\sigma(c_{10}) = 2.035257$.

Theorem 11. *There exist infinitely many strings w such that:*

$$\frac{\sigma(w)}{|w|} > 2.03698.$$

Proof. In Table 3, we see that $\sigma(v_{12})/|v_{12}| = 15389914.96/7555252 > 2.03698$. Thus, for any string $w = (v_{12})^k$, $k \geq 1$, we have

$$\frac{\sigma(w)}{|w|} = \frac{\sigma\left((v_{12})^k\right)}{|(v_{12})^k|} \geq \frac{k\cdot\sigma(v_{12})}{k\cdot|v_{12}|} > 2.03698,$$

since $\sigma(xy) \geq \sigma(x) + \sigma(y)$ holds for any strings x and y. □

In the rest of this section, we further push up the lower bound for $\sigma(n)$ by estimating the behavior of $\sigma(v_i)$ more carefully. It would be preferable to get a general

i	$\lvert v_i \rvert$	$\rho(v_i)$	$\frac{\rho(v_i)}{\lvert v_i \rvert}$	$\sigma(v_i)$	$\frac{\sigma(v_i)}{\lvert v_i \rvert}$	$\frac{\sigma(v_i^3)-\sigma(v_i^2)}{\lvert v_i \rvert}$
0	12	7	0.583333	14.90	1.24166	1.70238
1	31	23	0.741935	49.70	1.60322	1.94014
2	99	83	0.838384	180.88	1.82707	1.99612
3	303	268	0.884488	590.11	1.94756	2.02682
4	934	849	0.908994	1869.94	2.00208	2.03278
5	2876	2638	0.917246	5818.98	2.02329	2.03581
6	8857	8158	0.921079	17997.22	2.03197	2.03657
7	27276	25157	0.922313	55509.41	2.03510	2.03686
8	83999	77518	0.922844	171049.01	2.03632	2.03694
9	258683	238768	0.923014	526871.76	2.03674	2.03697
10	796639	735364	0.923083	1622679.68	2.03690	2.03698
11	2453326	2264678	0.923105	4997332.12	2.03696	2.03699152
12	7555252			15389914.96	2.03698	2.03699251

Table 3. Numbers of runs, and sums of exponents in runs in strings $v_i = \psi_e\left(\phi_r^i(\mathtt{a})\right)$

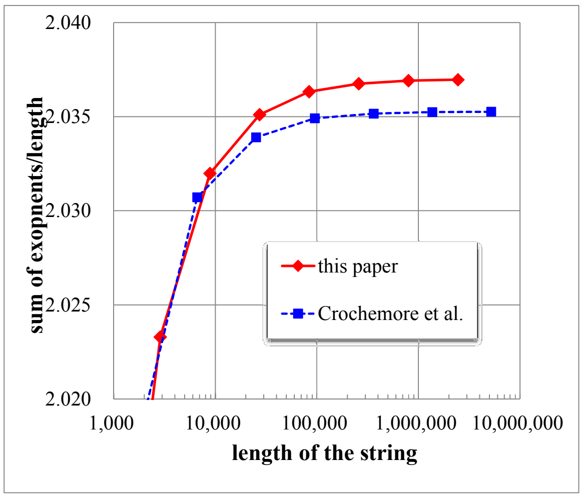


Figure 2. Comparison of the sum of exponents of runs in $v_i = \psi_e(\phi_r^i(\mathtt{a}))$ and Crochemore *et al.*'s $c_i = \psi_c(\phi_c^i(\mathtt{a}))$

formula of $\sigma(v_i)$, as similar to $\rho(u_i)$ in Section 5. Unfortunately, however, we failed to guess recurrence formulas on $\sigma(v_i)$ up to now. A part of the difficulty comes from the fact that $\sigma(v_i)$ is a fractional number, while $\rho(u_i)$ is an integer.

As an alternative approach, we consider a series of strings $\{w^k\}_{k\geq 1}$ of a run-rich string w, and compute a simple general formula for $\sigma\left(w^k\right)$. We first recall a property on runs in a string of the form w^k.

Lemma 12 ([15]). *Let $r = \langle i, l, p\rangle$ be a run in a string w^k for $k \geq 3$. If $l \geq 2|w|$, then $i = 1$ and $l = kn$, that is, $r = w^k$.*

Lemma 13. *For any string w and any $k \geq 2$,*

$$\sigma\left(w^k\right) = \left(\sigma\left(w^3\right) - \sigma\left(w^2\right)\right) \cdot k - \left(2\sigma\left(w^3\right) - 3\sigma\left(w^2\right)\right).$$

Proof. By Lemma 12, for any $k \geq 3$, the set $Run\left(w^k\right)$ consists of a single long run $\langle 1, \left|w^k\right|, p\rangle$ that covers the whole w^k, and many (possibly zero) short runs whose lengths are at most $2|w|$. Thus, we can verify that $\sigma\left(w^{k+1}\right) - \sigma\left(w^k\right) = \sigma(w^3) - \sigma(w^2)$ for any $k \geq 2$. By solving it, we get the general formula of $\sigma\left(w^k\right)$ as stated. $\square$

Theorem 14. *For any string w and any $\varepsilon > 0$, there exists a positive integer N such that for any $n \geq N$,*

$$\frac{\sigma(n)}{n} > \frac{\sigma(w^3) - \sigma(w^2)}{|w|} - \varepsilon.$$

Proof. By Lemma 13, $\sigma\left(w^k\right) = A \cdot k - B$, where $A = \sigma(w^3) - \sigma(w^2)$ and $B = 2\sigma(w^3) - 3\sigma(w^2)$. For any given $\varepsilon > 0$, we choose $N > \frac{A+B}{\varepsilon}$. For any $n \geq N$, let k be the integer satisfying $k > \frac{n}{|w|} \geq k - 1$. Notice that $k > \frac{n}{|w|} \geq \frac{N}{|w|} > \frac{A+B}{|w|\varepsilon}$. Since $\sigma(i+1) \geq \sigma(i)$ for any i, and $\left|w^{k-1}\right| = |w|(k-1)$, we have

$$\frac{\sigma(n)}{n} > \frac{\sigma(|w|(k-1))}{|w|k} \geq \frac{\sigma\left(w^{k-1}\right)}{|w|k} = \frac{A(k-1) - B}{|w|k} = \frac{A}{|w|} - \frac{A+B}{|w|k} > \frac{A}{|w|} - \varepsilon.$$

□

We now have a slightly better lower bound for $\sigma(n)$ compared to Theorem 11.

Theorem 15. *For any $\varepsilon > 0$ there exists a positive integer N such that for any $n \geq N$, $\frac{\sigma(n)}{n} > 2.036992 - \varepsilon$*

Proof. From Theorem 14 and the fact shown in Table 3, we have the bound. □

7 Concluding Remarks

We provided a new lower bound $2.036992n$ for the maximum value $\sigma(n)$ of the sum of exponents in runs in a string of length n, by exhibiting the series $\{\psi_e(\phi_r^i(\mathtt{a}))\}_{i\geq 0}$ of strings. Moreover, we also showed that the current best lower bound $0.9445757n$ for the number $\rho(n)$ of runs in a string of length n can be achieved by yet another series $\{h(\phi_r^i(\mathtt{a}))\}_{i\geq 0}$ of strings than Simpson's words $\{s_i\}_{i\geq 0}$ and Matsubara *et al.*'s words $\{t_i\}_{i\geq 0}$.

We note that the proof for Lemma 9 is incomplete for the moment, because the recurrence formula Eq. (3) is not formally proved yet for $i \geq 6$, in Table 2. We are also interested in obtaining a general formula of $\sigma(\psi_e(\phi_r^i(\mathtt{a})))$, which will yield a slightly better lower bound for $\sigma(n)$. Recall that for the standard Sturmian words, the number of runs in them can be exactly and directly computed from their *directive sequences* [3]. Similarly, it would be wonderful if we could develop a general technique to evaluate $\rho(\psi(\phi^i(\mathtt{a})))$ and $\sigma(\psi(\phi^i(\mathtt{a})))$ directly from the definition of ψ and ϕ. A natural extension of our experimental approach is to enlarge the domain of the morphism ϕ. For instance, can we get more run-rich strings $\{\psi(\phi^i(\mathtt{a}))\}_{i\geq 0}$ if we consider $\phi : \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\} \to \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}^*$?

Acknowledgments

This work was supported by KAKENHI 24106010, 23300051, 23220001, and 25560067.

Algorithm 1 find good morphism $\phi : \{\mathtt{a},\mathtt{b},\mathtt{c}\} \to \{\mathtt{a},\mathtt{b},\mathtt{c}\}^*$ by enumeration

```
$maxNum := 0$
$maxExp := 0$
**for** $N := 3$ to $\infty$ **do**
    **for** $\ell_\mathtt{a} := 1$ to $N-2$ **do**
        **for** $\ell_\mathtt{b} := 1$ to $N-\ell_\mathtt{a}-1$ **do**
            $\ell_\mathtt{c} := N-\ell_a-\ell_\mathtt{a}$
            **for** $n_\mathtt{a} := 0$ to $3^{\ell_\mathtt{a}}-1$ **do**
                **for** $n_\mathtt{a} := 0$ to $3^{\ell_\mathtt{b}}-1$ **do**
                    **for** $n_\mathtt{a} := 0$ to $3^{\ell_\mathtt{c}}-1$ **do**
                        Let $x_\mathtt{a}$ (resp. $x_\mathtt{b}$, $x_\mathtt{c}$) be the ternary representation of $n_\mathtt{a}$ (resp. $n_\mathtt{b}$, $n_\mathtt{c}$)
                          in $\ell_\mathtt{a}$ (resp. $\ell_\mathtt{b}$, $\ell_\mathtt{c}$) digits over $\{\mathtt{a},\mathtt{b},\mathtt{c}\}$
                        Let $\phi(\mathtt{a}) = x_\mathtt{a}$, $\phi(\mathtt{b}) = x_\mathtt{b}$ and $\phi(\mathtt{c}) = x_\mathtt{c}$
                        Let $w$ be the prefix of $h(\phi^k(\mathtt{a}))$ of length 10000,
                          where $k$ is the minimum integer satisfying $|h(\phi^k(\mathtt{a}))| \geq 10000$
                        **if** $\rho(w) > maxNum$ **then**
                            $maxNum := \rho(w)$ and **report** $\phi$
                        **if** $\sigma(w) > maxExp$ **then**
                            $maxExp := \sigma(w)$ and **report** $\phi$
```

Algorithm 2 find good morphism $\psi : \{\mathtt{a},\mathtt{b},\mathtt{c}\} \to \{\mathtt{0},\mathtt{1}\}^*$ by enumeration

```
$maxNum := 0$
$maxExp := 0$
**for** $N := 3$ to $\infty$ **do**
    **for** $\ell_\mathtt{a} := 1$ to $N-2$ **do**
        **for** $\ell_\mathtt{b} := 1$ to $N-\ell_\mathtt{a}-1$ **do**
            $\ell_\mathtt{c} := N-\ell_\mathtt{a}-\ell_\mathtt{b}$
            **for** $n_\mathtt{a} := 0$ to $2^{\ell_\mathtt{a}}-1$ **do**
                **for** $n_\mathtt{b} := 0$ to $2^{\ell_\mathtt{b}}-1$ **do**
                    **for** $n_\mathtt{c} := 0$ to $2^{\ell_\mathtt{c}}-1$ **do**
                        Let $y_\mathtt{a}$ (resp. $y_\mathtt{b}$, $y_\mathtt{c}$) be the binary representation of $n_\mathtt{a}$ (resp. $n_\mathtt{b}$, $n_\mathtt{c}$)
                          in $\ell_\mathtt{a}$ (resp. $\ell_\mathtt{b}$, $\ell_\mathtt{c}$) digits over $\{\mathtt{0},\mathtt{1}\}$.
                        Let $\psi(\mathtt{a}) = y_\mathtt{a}$, $\psi(\mathtt{b}) = y_\mathtt{b}$ and $\psi(\mathtt{c}) = y_\mathtt{c}$
                        Let $w$ be the prefix of $\psi(\phi_r^k(\mathtt{a}))$ of length 10000,
                          where $k$ is the minimum integer satisfying $|\psi(\phi_r^k(\mathtt{a}))| \geq 10000$
                        **if** $\rho(w) > maxNum$ **then**
                            $maxNum := \rho(w)$ and **report** $\psi$
                        **if** $\sigma(w) > maxExp$ **then**
                            $maxExp := \sigma(w)$ and **report** $\psi$
```

References

1. J.-P. Allouche and J. Shallit: *Automatic Sequences*, Cahmbridge University Press, 2003.
2. A. Baker, A. Deza, and F. Franek: *A computational framework for determining run-maximal strings*. Journal of Discrete Algorithms, 2012.
3. P. Baturo, M. Piatkowski, and W. Rytter: *The number of runs in Sturmian words*, in Proc. CIAA 2008, 2008, pp. 252–261.
4. M. Crochemore, L. Ilie, and L. Tinta: *Towards a solution to the "runs" conjecture*, in Proc. CPM 2008, vol. 5029 of LNCS, 2008, pp. 290–302.
5. M. Crochemore, L. Ilie, and L. Tinta: *The "runs" conjecture*. Theoretical Computer Science, 412 2011, pp. 2931–2941.
6. M. Crochemore, M. Kubica, J. Radoszewski, W. Rytter, and T. Waleń: *On the maximal sum of exponents of runs in a string*. Journal of Discrete Algorithms, 2011.
7. F. Franek and Q. Yang: *An asymptotic lower bound for the maximal number of runs in a string*. International Journal of Foundations of Computer Science, 19(01) 2008, pp. 195–203.

8. F. Franěk, R. Simpson, and W. Smyth: *The maximum number of runs in a string*, in Proc. AWOCA2003, 2003, pp. 26–35.
9. F. Franěk and Q. Yang: *An asymptotic lower bound for the maximal-number-of-runs function*, in Proc. Prague Stringology Conference (PSC'06), 2006, pp. 3–8.
10. M. Giraud: *Not so many runs in strings*, in Proc. LATA 2008, 2008, pp. 245–252.
11. R. L. Graham, D. E. Knuth, and O. Patashnik: *Concrete Mathematics: A Foundation for Computer Science*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd ed., 1994.
12. R. M. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in Proc. FOCS'99, 1999, pp. 596–604.
13. K. Kusano, K. Narisawa, and A. Shinohara: *Computing maximum number of runs in strings*, in String Processing and Information Retrieval, Springer, 2012, pp. 318–329.
14. W. Matsubara, K. Kusano, H. Bannai, and A. Shinohara: *A series of run-rich strings*, in Proc. LATA 2009, 2009, pp. 578–587.
15. W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and A. Shinohara: *New lower bounds for the maximum number of runs in a string*, in Proc. PSC2008, 2008, pp. 140–145.
16. S. J. Puglisi, J. simpson, and W. F. Smyth: *How many runs can a string contain?* Theoretical Computer Science, 401(1–3) 2008, pp. 165–171.
17. W. Rytter: *The number of runs in a string: Improved analysis of the linear upper bound*, in Proc. STACS 2006, vol. 3884 of LNCS, 2006, pp. 184–195.
18. W. Rytter: *The number of runs in a string.* Inf. Comput., 205(9) 2007, pp. 1459–1469.
19. J. Simpson: *Modified Padovan words and the maximum number of runs in a word.* Australasian Journal of Combinatorics, 46 2010, pp. 129–146.

A Appendix

We note some lemmas and remarks verified by Mathematica 9.0.1.

Lemma 16. $(11\kappa^2+7\kappa-6)/(11\kappa^2+8\kappa-6)=\eta$*, where* κ *is the real root of* $z^3-z-1=0$*, and* η *is the real root of* $2693z^3-7714z^2+7379z-2357=0$*.*

Proof. We can verify it as follows.

```
kappa = Solve[z^3 − z − 1 == 0, z][[1]]
```

$$\left\{z\to\frac{1}{3}\left(\frac{27}{2}-\frac{3\sqrt{69}}{2}\right)^{1/3}+\frac{\left(\frac{1}{2}\left(9+\sqrt{69}\right)\right)^{1/3}}{3^{2/3}}\right\}$$

```
FullSimplify[(11z^2 + 7z − 6)/(11z^2 + 8z − 6)/.kappa]
```

$$\text{Root}\left[-2357+7379\#1-7714\#1^2+2693\#1^3\&,1\right]$$

□

Lemma 17. *The real root* η *of* $2693z^3-7714z^2+7379z-2357=0$ *is*

$$\frac{7714-109145\sqrt[3]{\frac{2}{-27669823+9298929\sqrt{69}}}+\sqrt[3]{\frac{-27669823+9298929\sqrt{69}}{2}}}{8079}=0.9445757124$$

Proof. We can easily verify it as follows.

```
eta = Solve[2693x^3 − 7714x^2 + 7379x − 2357 == 0][[1]]
```

$$\left\{x\to\frac{7714-109145\left(\frac{2}{-27669823+9298929\sqrt{69}}\right)^{1/3}+\left(\frac{1}{2}\left(-27669823+9298929\sqrt{69}\right)\right)^{1/3}}{8079}\right\}$$

```
N[%, 10]
{x → 0.9445757124}
```

□

Remark 18. The following instructions would give a confidence that $L(z)$ (resp. $R(z)$) in Lemma 8 (resp. Lemma 9) is a generating function of $|u_i|$ (resp. $\rho(u_i)$) in Table 1.

```
Table[ SeriesCoefficient[(−8z^2 − 21z − 24)/(z^3 + 3z^2 + 2z − 1),
{z, 0, n}], {n, 0, 10}]
{24, 69, 218, 667, 2057, 6333, 19504, 60064, 184973, 569642, 1754267}

Table[SeriesCoefficient[(−16 − 8z + 7z^2 − 5z^3 − 3z^4 − z^5 + z^6)/
((1 − z)^2 ∗ (1 + z) ∗ (−1 + 2z + 3z^2 + z^3)), {z, 0, n}], {n, 0, 10}]
{16, 56, 193, 616, 1925, 5963, 18400, 56711, 174693, 538041, 1657005}
```

Lemma 19. *Assume that* $\sum_{i=0}^{\infty} |u_i| z^i = \frac{-8z^2 - 21z - 24}{z^3 + 3z^2 + 2z - 1}$, *and*
$\sum_{i=0}^{\infty} \rho(u_i)\, z^i = \frac{-16 - 8z + 7z^2 - 5z^3 - 3z^4 - z^5 + z^6}{(1-z)^2(1+z)(-1+2z+3z2+z3)}$.
Then $\lim_{i\to\infty} \frac{\rho(u_i)}{|u_i|} = \eta$, *where* η *is the real root of* $2693z^3 - 7714z^2 + 7379z - 2357 = 0$.

Proof. We can verify it as follows.

```
leng[n_]:=SeriesCoefficient[(-24-21z-8z^2)/(-1+2z+3z^2+z^3), {z, 0, n}]
run[n_]:=SeriesCoefficient[(-16-8z+7z^2-5z^3-3z^4-z^5+z^6)/((-1+z)^2(1+z)(-1+2z+3z^2+z^3)), {z, 0, n}]

Table[leng[n], {n, 0, 10}]
{24, 69, 218, 667, 2057, 6333, 19504, 60064, 184973, 569642, 1754267}

Table[run[n], {n, 0, 10}]
{16, 56, 193, 616, 1925, 5963, 18400, 56711, 174693, 538041, 1657005}

FullSimplify[Limit[run[n]/leng[n], n → Infinity]]
Root[-2357 + 7379#1 - 7714#1^2 + 2693#1^3&, 1]
```

□

Lemma 20. *Assume* $\sum_{i=0}^{\infty} |v_i| z^i = \frac{-12 - 7z - z^2}{-1 + 2z + 3z^2 + z^3}$, *and*
$\sum_{i=0}^{\infty} \rho(v_i)\, z^i = \frac{-7 - 2z - 8z^3 - 8z^4 - 2z^5 + z^6 + z^7}{(-1+z)^2(1+z)\,(-1+2z+3z^2+z^3)}$.
Then $\lim_{i\to\infty} \frac{\rho(v_i)}{|v_i|} = 0.9231182492\ldots$ *is the real root of* $175z^3 - 344z^2 + 397z - 211 = 0$.

Proof. We can easily verify it as follows.

```
leng[n_]:=SeriesCoefficient[(-12-7z-z^2)/(-1+2z+3z^2+z^3), {z, 0, n}]
run[n_]:=SeriesCoefficient[(-7-2z-8z^3-8z^4-2z^5+z^6+z^7)/((-1+z)^2(1+z)(-1+2z+3z^2+z^3)), {z, 0, n}]

Table[leng[n], {n, 0, 10}]
{12, 31, 99, 303, 934, 2876, 8857, 27276, 83999, 258683, 796639}

Table[run[n], {n, 0, 10}]
{7, 23, 83, 268, 849, 2638, 8158, 25157, 77518, 238768, 735364}

FullSimplify[Limit[run[n]/leng[n], n → Infinity]]
Root[-211 + 397#1 - 344#1^2 + 175#1^3&, 1]

N[%, 10]
0.9231182492
```

□

The Sum of Exponents of Maximal Repetitions in Standard Sturmian Words

Marcin Piątkowski

Faculty of Mathematics and Computer Science,
Nicolaus Copernicus University, Toruń, Poland
`marcin.piatkowski@mat.umk.pl`

Abstract. A maximal repetition is a non-extendable (with the same period) periodic segment in a string, in which the period repeats at least twice. In this paper we study problems related to the structure of maximal repetitions in standard Sturmian words and present the formulas for the sum of their exponents. Moreover, we show how to compute the sum of exponents of maximal repetitions in any standard Sturmian word in linear time with respect to the (total) size of its compressed representation. The presented formulas and algorithm can be easily modified to obtain the total run length of the word.

Keywords: Sturmian words, repetitions, runs, algorithm

1 Introduction

Problems related to repetitions are fundamental in combinatorics on words and many practical applications: data compression, computational biology, pattern-matching and so on, see for instance [6], [7], [10], [11], [14] and references therein. The most important type of repetitions are maximal repetitions, i.e. non-extendable (with the same period) periodic segments in a string, in which the period repeats at least twice. This paper complements the work [2], where the exact formula for the number of runs in standard Sturmian words was presented. We investigate here the structure of runs in standard Sturmian words in more details to obtain a formula for the sum of their exponents. We show also an algorithm, derived from our formula, which computes the sum of exponents of maximal repetitions in any standard word in linear time with respect to the (total) size of its compressed representation (i.e. the directive sequence).

Throughout the paper we use the standard notions of combinatorics on words. In particular, words are finite sequences over a finite set Σ of letters, called the alphabet. For a word $w = w_1 w_2 \cdots w_n$, by w_i we denote its i-th letter, by $w[i..j]$ the subword $w_i w_{i+1} \cdots w_j$, by $|w|$ its length and by $|w|_a$ the number of letters a occurring in w. The number i is a period of the word w if $w_j = w_{i+j}$ for all i with $i + j \leq |w|$. The minimal period of w is denoted by *period*(w). We say that a word w is periodic if *period*$(w) \leq \frac{|w|}{2}$. A word w is said to be *primitive* if w is not of the form z^k, where z is a nonempty word and $k \geq 2$ is a natural number.

A *maximal repetition* (a *run*, in short) in a word w is an interval $\alpha = [i..j]$, such that $w[i..j] = u^k v$ $(k \geq 2)$ is a nonempty periodic subword of w, where u is of the minimal length and v is a proper prefix (possibly empty) of u, that can not be extended (neither $w[i-1..j]$ nor $w[i..j+1]$ is a run with the period $|u|$). The factor v is called the remainder of α and the number $k + \frac{|v|}{|u|}$ is called the exponent of α. The sum of exponents of all runs in w is denoted by $\sigma(w)$. Note that the exponent of a run is a rational number, hence the value of $\sigma(w)$ does not have to be integer.

A run α can be properly included as an interval in another run β, but in this case $\mathit{period}(\alpha) < \mathit{period}(\beta)$. The value of the run $\alpha = [i..j]$ is the factor $\mathit{val}(\alpha) = w[i..j]$. When it makes no ambiguity we identify sometimes a run with its value and the period of the run $\alpha = [i..j]$ with the subword $w[i..\mathit{period}(w)]$, called also the *generator* of the repetition. The meaning will always be clear from the context. Observe that two different runs could correspond to the identical subwords, if we disregard their positions. Hence, runs are also called the maximal *positioned* repetitions.

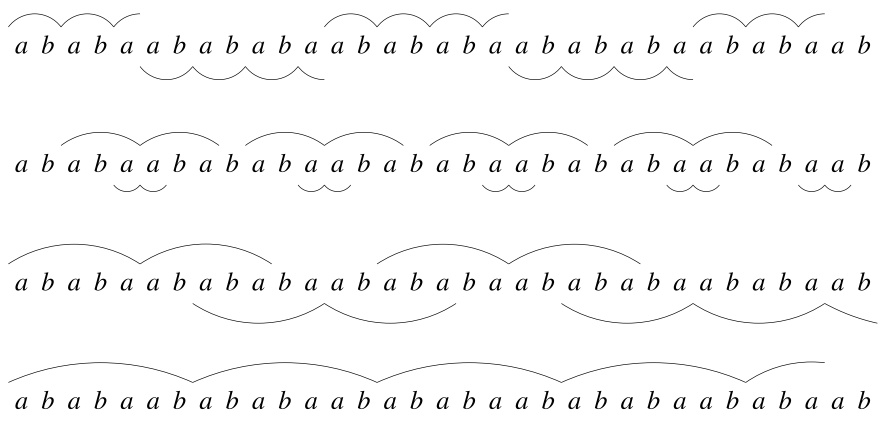

a b a b a a b a b a b a a b a b a b a a b a b a b a a b a b a a b
a b a b a a b a b a b a a b a b a b a a b a b a b a a b a b a a b
a b a b a a b a b a b a a b a b a b a a b a b a b a a b a b a a b
a b a b a a b a b a b a a b a b a b a a b a b a b a a b a b a a b

Figure 1. The structure of maximal repetitions for the example binary word.

Example 1.
Let $w = ababaabababaabababaabababaababaab$ be a binary word.
There are 5 runs with the period a and the exponent 2:

$$w[5..6] = a^2, \quad w[12..13] = a^2, \quad w[19..20] = a^2, \quad w[26..27] = a^2, \quad w[31..32] = a^2,$$

5 runs with the period ab and the exponents respectively $2\frac{1}{2}$ and $3\frac{1}{2}$:

$$w[1..5] = (ab)^2a, \qquad w[6..12] = (ab)^3a, \qquad w[13..19] = (ab)^3a,$$
$$w[20..26] = (ab)^3a, \qquad w[27..31] = (ab)^2a,$$

4 runs with the period aba and the exponent 2:

$$w[3..8] = (aba)^2, \quad w[10..15] = (aba)^2, \quad w[17..22] = (aba)^2, \quad w[24..29] = (aba)^2,$$

4 runs with the period $ababa$ and the exponents respectively 2 and $2\frac{2}{5}$:

$$w[1..10] = (ababa)^2, \qquad w[8..17] = (ababa)^2,$$
$$w[15..24] = (ababa)^2, \qquad w[22..33] = (ababa)^2ab,$$

and 1 run with the period $ababaab$ and the exponent $4\frac{3}{7}$: $w[1..31] = (ababaab)^4aba$. Altogether we have 19 runs and sum of their exponents equals $49\frac{23}{70} \approx 49.3286$, see Figure 1 for comparison.

In 1999 Kolpakov and Kucherov showed that the number of runs in a word is linear with respect to its length (see [13]). The stronger property of runs is that the sum of their exponents is also linear with respect to the length of the word. Kolpakov and Kucherov conjectured that for all w we have $\sigma(w) \leq 2 \cdot |w|$. In 2012 Crochemore with coauthors contradicted this conjecture and showed that the upper bound for $\sigma(w)$ is $2.035 \cdot |w| \leq \sigma(w) \leq 4.1 \cdot |w|$. In this paper we investigate this problem in very special class of strings – the standard Sturmian words. We present compact formulas for the sum of runs exponents and an algorithm for its efficient computation.

Recently a new measure of a string periodicity was proposed by Glen and Simpson (see [12]). The *total run length* (TRL) of a word w is the sum of the lengths of all runs in w. Since this notion is similar to the sum of exponents of maximal repetitions, our formulas and algorithm could be easily adopted to compute also the total run length of any standard Sturmian word.

The paper is organized as follows. In section 2 we introduce the definition of standard Sturmian words and some of their basic properties. Next, in section 3 we study the structure of repetitions in standard Sturmian words and present a few facts necessary in further investigation. Finally, we show an prove the formulas for the sum of exponents of maximal repetitions together with an algorithm for its fast computation. Some useful applets related to problems considered in this paper can be found on the web site:

`http://www.mat.umk.pl/~martinp/stringology/applets/`

2 Standard Sturmian words

Standard Sturmian words (standard words in short) are one of the most investigated class of strings in combinatorics on words, see for instance [1], [3], [4], [5], [15], [17], [18] and references therein. They have very compact representations in terms of sequences of integers, which has many algorithmic consequences.

The *directive sequence* is the integer sequence: $\gamma = (\gamma_0, \gamma_1, \ldots, \gamma_n)$, where $\gamma_0 \geq 0$ and $\gamma_i > 0$ for $i = 1, 2, \ldots, n$. The standard word corresponding to γ, denoted by $\mathrm{Sw}(\gamma)$, is described by the recurrences of the form:

$$x_{-1} = b, \qquad x_0 = a, \qquad \ldots, \qquad x_n = (x_{n-1})^{\gamma_{n-1}} x_{n-2}, \qquad x_{n+1} = (x_n)^{\gamma_n} x_{n-1}, \quad (1)$$

where $\mathrm{Sw}(\gamma) = x_{n+1}$. For simplicity we denote $q_i = |x_i|$.

Example 2.
Consider the directive sequence $\gamma = (1, 2, 1, 3, 1)$. We have $\mathrm{Sw}(\gamma) = x_5$, where:

$$
\begin{array}{llll}
x_{-1} = b & & & q_{-1} = 1 \\
x_0 = a & & & q_0 = 1 \\
x_1 = (x_0)^1 \cdot x_{-1} & = a \cdot b & & q_1 = 2 \\
x_2 = (x_1)^2 \cdot x_0 & = ab \cdot ab \cdot a & & q_2 = 5 \\
x_3 = (x_2)^1 \cdot x_1 & = ababa \cdot ab & & q_3 = 7 \\
x_4 = (x_3)^3 \cdot x_2 & = ababaab \cdot ababaab \cdot ababaab \cdot ababa & & q_4 = 26 \\
x_5 = (x_4)^1 \cdot x_3 & = ababaababaababaababaababa \cdot ababaab & & q_5 = 33
\end{array}
$$

The sequence of words $\{x_i\}_{i=0}^{n+1}$ is called the standard sequence. Every word occurring in a standard sequence is a standard word, and every standard word occurs in some standard sequence. We assume that the standard word given by the empty directive sequence is a and $\mathrm{Sw}(0) = b$.

Observe that for even $n > 0$ the standard word x_n has the suffix ba, and for odd $n > 0$ it has the suffix ab. Moreover, for $\gamma_0 > 0$ we have standard words starting with the letter a and for $\gamma_0 = 0$ we have standard words starting with the letter b. In fact the word $\mathrm{Sw}(0, \gamma_1, \ldots, \gamma_n)$ can be obtained from $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$ by switching the letters a and b. Without loss of generality we consider here standard words starting with the letter a, therefore we assume $\gamma_0 > 0$. Words starting with the letter b can be considered similarly.

Remark 3.
The special kind of standard words are well known Fibonacci words. They are formed by repeated concatenation in the same way that the Fibonacci numbers are formed by repeated addition. By definition Fibonacci words are standard words given by directive sequences of the form $\gamma = (1, 1, \ldots, 1)$ (n-th Fibonacci word F_n corresponds to a sequence of n ones).

The number $N = |\mathrm{Sw}(\gamma)|$ is the (real) size of the word, while $(n + 1) = |\gamma|$ can be thought as its compressed size. Observe that, by the definition of standard words, N is exponential with respect to n. Moreover, each directive sequence corresponds to a *grammar-based compression*, which consists in describing a given word by a context-free grammar G generating this (single) word. The size of the grammar G is the total length of all productions of G. In our case the size of the considered grammar is proportional to the length of the directive sequence.

2.1 Morphic reduction of standard words

The recurrent definition of standard words from equation (1) leads to their simple characterization by a composition of morphisms. Let $\gamma = (\gamma_0, \gamma_1, \ldots, \gamma_n)$ be a directive sequence. We associate with γ a sequence of morphisms $\{h_i\}_{i=0}^{n}$, defined as:

$$h_i : \begin{cases} a \longrightarrow a^{\gamma_i} b \\ b \longrightarrow a \end{cases} \qquad \text{for } 0 \le i \le n. \tag{2}$$

The following fact describes another simple method of standard word generation. It can be proven by a simple induction, see [2] for more details.

Lemma 4 (see [2]).
For $0 \le i \le n$ the morphism h_i transforms a standard word into another standard word, and we have:

$$\begin{aligned} \mathrm{Sw}(\gamma_n) &= h_n(a), \\ \mathrm{Sw}(\gamma_i, \gamma_{i+1}, \ldots, \gamma_n) &= h_i\Big(\mathrm{Sw}(\gamma_{i+1}, \gamma_{i+2}, \ldots, \gamma_n)\Big). \end{aligned}$$

As a direct corollary to Lemma 4 we have that for $\gamma = (\gamma_0, \gamma_1, \ldots, \gamma_n)$:

$$\mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n) = h_0 \circ h_1 \circ \cdots \circ h_n(a). \tag{3}$$

Moreover, the inverse morphism h_i^{-1} can be seen as a reduction of a standard word $w^{(i)} = \mathrm{Sw}(\gamma_i, \ldots, \gamma_n)$ to $w^{(i+1)} = \mathrm{Sw}(\gamma_{i+1}, \ldots, \gamma_n)$.

Recall that $|w|_a$ denotes the number of occurrences of the letter a in the word w. In the rest of this paper, for $\gamma = (\gamma_0, \ldots, \gamma_n)$ and $0 \le k \le n$, we use the following notation:

$$N_\gamma(k) \;=\; |\mathrm{Sw}(\gamma_k, \gamma_{k+1}, \ldots, \gamma_n)|_a, \tag{4}$$

which enables us to simplify the formulas for the sum of runs exponents. Observe that equations (2) and (4) imply:

$$N_\gamma(k) \;=\; \gamma_k \cdot N_\gamma(k+1) + N_\gamma(k+2). \tag{5}$$

Example 5.
Consider a directive sequence $\gamma = (1, 2, 1, 3, 1)$. We have (compare with Example 2):

$$\begin{aligned}
\mathrm{Sw}(1,2,1,3,1) &= ababaabababaababababaabababaababaab & N_\gamma(0) &= 19,\\
\mathrm{Sw}(2,1,3,1) &= aabaaabaaabaaabaaba & N_\gamma(1) &= 14,\\
\mathrm{Sw}(1,3,1) &= abababaab & N_\gamma(2) &= 5,\\
\mathrm{Sw}(3,1) &= aaaba & N_\gamma(3) &= 4,\\
\mathrm{Sw}(1) &= ab & N_\gamma(4) &= 1,\\
\mathrm{Sw}(\varepsilon) &= a & N_\gamma(5) &= 1.
\end{aligned}$$

As a straightforward corollary to equations (2), (4) and (5) we have:

Corollary 6.
The number of letters b in a word $\mathrm{Sw}(\gamma_i, \ldots, \gamma_n)$ equals $N_\gamma(i+1)$.

2.2 The m-partition of a standard word

The concept of the m-partition of a standard word is crucial in the maximal repetitions structure investigation. It allows us to divide the set of all runs in a standard word to disjoint sets depending on the length of their periods and simplify the considered problems. The following fact is a direct consequence of the recurrent definition of standard words.

Proposition 7.
Every standard word $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ can be represented as a sequence of concatenated words x_m and x_{m-1}, and has the form:

$$(i)\;\; x_m^{\alpha_1}\, x_{m-1}\, x_m^{\alpha_2}\, x_{m-1} \cdots x_m^{\alpha_s}\, x_{m-1}\, x_m \qquad \text{or} \qquad (ii)\;\; x_m^{\beta_1}\, x_{m-1}\, x_m^{\beta_2}\, x_{m-1} \cdots x_m^{\beta_s}\, x_{m-1},$$

where $\alpha_k,\ \beta_k \in \{\gamma_m,\ \gamma_m + 1\}$, $0 \le m \le n$, and x_m are as in equation (1).

Such a decomposition of a standard word w is called the *m-partition* of w. The block x_m is called the *repeating block* and x_{m-1} – the *single block*. Recall that for $m > 0$ the last two letters of x_m are ab for an odd m and ba for an even m. Therefore the m-partition of $x_{n+1} = \mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ is of the form (i) if m has the same parity as $(n+1)$, and of the form (ii) otherwise (see Example 9 and Figure 2).

Note that the 0-partition of a standard word is its decomposition into letters. Moreover, Proposition 7, Lemma 4 and equation (3) imply the following fact.

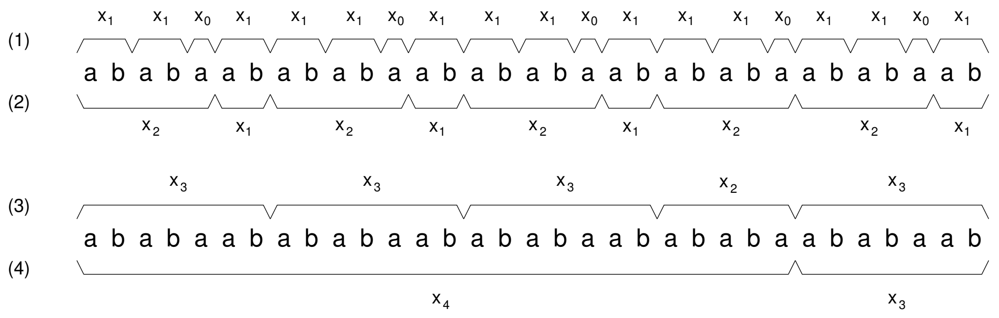

(1)
(2)
(3)
(4)
a b a b a a b a b a b a a b a b a b a a b a b a b a a b a b a a b
x1 x1 x0 x1 x1 x1 x0 x1 x1 x1 x0 x1 x1 x1 x0 x1 x1 x0 x1
x2 x1 x2 x1 x2 x1 x2 x2 x1
x3 x3 x3 x2 x3
x4 x3

Figure 2. The m-partition of the word $\mathrm{Sw}(1,2,1,3,1)$ for $1 \le m \le 4$.

Proposition 8.
The structure of occurrences of the block x_m (respectively x_{m-1}) in the m-partition of $\mathrm{Sw}(\gamma_0,\dots,\gamma_n)$ corresponds to the structure of occurrences of the letter a (respectively letter b) in $\mathrm{Sw}(\gamma_m,\dots,\gamma_n)$.

Example 9.
Consider a standard word $\mathrm{Sw}(1,2,1,3,1)$. Its m-partitions (for $1 \le m \le 4$) together with its corresponding morphic reductions are depicted in the table below. See also Figure 2 for comparison.

m	m-partition	$\mathrm{Sw}(\gamma_m,\dots,\gamma_n)$
1	$ab \cdot ab \cdot a \cdot ab \cdot ab \cdot ab \cdot a \cdot ab \cdot ab \cdot ab \cdot a \cdot ab \cdot ab \cdot ab \cdot a \cdot ab \cdot ab \cdot a \cdot ab$	$aabaaabaaabaaabaaba$
2	$ababa \cdot ab \cdot ababa \cdot ab \cdot ababa \cdot ab \cdot ababa \cdot ababa \cdot ab$	$ababababaab$
3	$ababaab \cdot ababaab \cdot ababaab \cdot ababa \cdot ababaab$	$aaaba$
4	$ababaababababaabababaababababa \cdot ababaab$	ab

3 The structure of maximal repetitions in standard words

The aim of this section is the presentation of some technical facts used further to prove the correctness of formulas for the sum of runs exponents. We start with recalling some technical facts presented in [8] and [9] related to the structure of factors in standard words.

Lemma 10 (see [9]).
Let $\gamma = (\gamma_0,\dots,\gamma_n)$ be a directive sequence. For every $0 \le k \le n$ and every $1 \le i \le \gamma_k$ the word $(x_k)^i x_{k-1}$ is primitive (i.e. is not of the form z^s, where z is nonempty and $s \ge 2$ is a natural number).

Lemma 11 (See [8]).
Let $w = \mathrm{Sw}(\gamma_0,\dots,\gamma_n)$ be a standard word and let $y \in \{a,b\}$ be a letter. For each occurrence of $y \cdot x_i$ in w, y is the last letter of the block x_{i-1} or x_i of the i-partition of w. Moreover, the type of this block is uniquely determined by y.

The following lemma is a key tool in the study of the runs structure in standard words. It is a version of Theorem 1 in [9] using a slightly different notation.

Lemma 12 (Structural Lemma).
The period of each maximal repetition in a standard word $\mathrm{Sw}(\gamma_0,\gamma_1,\dots,\gamma_n)$ is of the form x_i or $(x_i)^j x_{i-1}$, where $0 \le i \le n$, $0 < j < \gamma_i$ and x_i's are as in equation (1).

To prove the above lemma it is sufficient to show that no factor of a standard word $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ that does not satisfy the condition given there could be the generator of some repetition, see the proof of Theorem 1 in [9] for more details.

Let us denote by $\widehat{w}$ the word w with two last letters removed and by $\widetilde{w}$ the word w with two last letters exchanged. The following fact can be proven by a simple induction, see for instance [15].

Lemma 13.
Let x_i be as in equation (1) and $i > 1$. Then:

1. *We have $x_{i-1} \cdot x_i \; = \; x_i \cdot \widetilde{x_{i-1}}$,*
2. *The longest prefix of $x_{i-1} \cdot x_i$ with the period of the length q_i is of the form $x_i \cdot \widehat{x_{i-1}}$.*

Example 14.
Recall the word $\mathrm{Sw}(1, 2, 1, 3, 1)$ from Example 2, where $x_3 = ababaab$, $x_2 = ababa$. Then we have $\widetilde{x_2} = abaab$, $\widehat{x_2} = aba$ and

$$x_2 \cdot x_3 \; = \; ababa \cdot ababaab \; = \; ababaab \cdot abaab \; = \; x_3 \cdot \widetilde{x_2}.$$

Moreover, the longest prefix of $x_2 \cdot x_3$ with the period of the length q_2 is of the form:

$$\rlap{$\underbrace{}_{x_3\cdot\widehat{x_2}}$}\overbrace{a\,b\,a\,b\,a}^{x_2}\overbrace{a\,b\,a\,b\,a\,b\,a}^{x_3}$$

Observe that by equation (1) we have

$$\mathrm{Sw}(\gamma_0, \ldots, \gamma_n, 1) = (x_n)^{\gamma_n} \cdot x_{n-1} \cdot x_n \quad \text{and} \quad \mathrm{Sw}(\gamma_0, \ldots, \gamma_n + 1) = (x_n)^{\gamma_n} \cdot x_n \cdot x_{n-1}.$$

Therefore, as a straightforward corollary to the first point of Lemma 13 we get:

Corollary 15.
Standard words $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n, 1)$ and $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n + 1)$ differ only in the order of the last two letters.

See Figure 3 for an illustration of this fact. To properly count the exponents of runs in standard words we need also the following fact.

Proposition 16.
Let $w = \mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ be a standard word and $2 \leq i \leq n-2$. If x_{i-1} is the last block of the i-partition of w, then it is preceded by $(x_i)^{\gamma_i+1}$.

Proof.
Let $w = \mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ be a standard word and $2 \leq i \leq n-2$. By equation (1) we have $x_i = \mathrm{Sw}(\gamma_0, \ldots, \gamma_i)$ and $x_{i-1} = \mathrm{Sw}(\gamma_0, \ldots, \gamma_{i-1})$. Recall that x_i ends with ba for even $i > 0$ (i.e. for the odd length of a directive sequence) and with ab for odd $i > 0$ (i.e. for the even length of a directive sequence). Consider that w has the suffix $(x_i)^\alpha x_{i-1}$. Then n and i have the same parity and the number $n - m + 1$ is odd, hence the word $w^{(m)} = \mathrm{Sw}(\gamma_m, \ldots, \gamma_n)$ ends with ab. More precisely, due to Proposition 8, $w^{(m)}$ ends with $a^\alpha b$. By Lemma 4, the suffix $a^{\gamma_i} b$ of $w^{(m)}$ corresponds to the last letter a of $w^{(m+1)} = \mathrm{Sw}(\gamma_{m+1}, \ldots, \gamma_n)$. Since $n - m + 2$ is even and $w^{(m+1)}$ ends with ba, due to Lemma 4 the suffix $a^{\gamma_i} b$ of $w^{(m)}$ have to be preceded by a single occurrence of a. Therefore, we have $\alpha = \gamma_i + 1$ and this completes the proof. $\square$

4 The sum of exponents of maximal repetitions

In this section we present and prove formulas for the the sum of exponents of maximal repetitions in any standard word, that depend only on its compressed representation – the directive sequence. The following zero-one functions for testing the parity of a nonnegative integer i will be useful to simplify those formulas:

$$\mathit{even}(i) \;=\; \begin{cases} 1 & \text{for even } \; i \\ 0 & \text{for odd } \; i \end{cases} \qquad \text{and} \qquad \mathit{odd}(i) \;=\; \begin{cases} 1 & \text{for odd } \; i \\ 0 & \text{for even } \; i \end{cases}.$$

Moreover, we define an auxilary function $\mathit{\Delta}_n : \mathbf{N} \to \mathbf{N}$:

$$\mathit{\Delta}_n(i) \;=\; |n - i + 1| \bmod 2.$$

In other words, $\mathit{\Delta}_n(i) = 1$ if and only if the numbers n and i have the same parity, and $\mathit{\Delta}_n(i) = 0$ otherwise. Recall also that for simplicity we denote $|x_i| = q_i$.

The main idea of the computation of the sum of runs exponents in a standard word w is the partition of the set of all maximal repetitions in w into separate categories depending on the length of their periods. Runs in w with the period of the form x_i and $(x_i)^k x_{i-1}$ (for $1 < k < \gamma_i$), where x_i are as in equation (1), are called the runs of *type* i. We study runs of each type separately.

Let $\sigma_i(\gamma)$ denotes the sum of exponents of type i runs. Then the sum of exponents of all runs in $\mathrm{Sw}(\gamma)$ can be computed using the following theorem.

Theorem 17.
Let $\gamma = (\gamma_0, \ldots, \gamma_n)$ be a directive sequence. The sum of exponents of runs in $\mathrm{Sw}(\gamma)$ is given as:

$$\sigma(\gamma) \;=\; \sum_{i=1}^{n} \sigma_i(\gamma).$$

The detailed computation of $\sigma_i(\gamma)$ for each $0 \le i \le n$ is provided below.

4.1 The general case

We start with an investigation of a general case, i.e. maximal repetitions of the type i for $2 \le i \le n-1$. First, we consider runs with the period of the form x_i.

Lemma 18.
Let $\gamma = (\gamma_0, \ldots, \gamma_n)$ be a directive sequence and $w = \mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ be a standard word. For $2 \le i \le n-1$ the sum of exponents of runs with the period x_i in w equals:

$$\sigma'_i(\gamma) \;=\; N_\gamma(i+1) \cdot \left(\gamma_i + 1 + \frac{q_{i-1} - 2}{q_i}\right) + \Big(N_\gamma(i+2) - 1\Big) + \mathit{\Delta}_n(i)\frac{2}{q_i}. \qquad (6)$$

Proof.
Let us denote

$$w = \mathrm{Sw}(\gamma_0, \ldots, \gamma_n), \quad w^{(i)} = \mathrm{Sw}(\gamma_i, \ldots, \gamma_n) \quad \text{and} \quad w^{(i+1)} = \mathrm{Sw}(\gamma_{i+1}, \ldots, \gamma_n).$$

Due to Lemma 11, each maximal repetition with the period x_i in w is aligned to the i-partition of w, hence it corresponds to a block $(x_i)^\alpha x_{i-1}$, where $\alpha \in \{\gamma_i, \gamma_i + 1\}$.

Each internal block of this form is followed by a subsequent x_i. Due to Lemma 13, the longest prefix of $x_{i-1}x_i$ with the period x_i equals $x_i \cdot \widehat{x_{i-1}}$. Therefore, the period of the considered run repeats $\alpha+1$ times and its fractional part has the length $q_{i-1}-2$.

Consider the i-partition of w. By Proposition 8 occurrences of x_i correspond to occurrences of a in $w^{(i)}$ and occurrences of x_{i-1} correspond to occurrences of b in $w^{(i)}$. Therefore, a block $(x_i)^\alpha x_{i-1}$ correspond to the block $a^\alpha b$ in $w^{(i)}$. Moreover, due to Lemma 4, each block of the form $a^{\gamma_i+1}b$ in $w^{(i)}$ corresponds to the letter a preceded by the letter b in $w^{(i+1)}$ and each block of the form $a^{\gamma_i}b$ in $w^{(i)}$ corresponds to the letter a not preceded by the letter b in $w^{(i+1)}$.

The rightmost occurrence of $(x_i)^\alpha x_{i-1}$ have to be considered separately. Due to Proposition 7, if i and n have different parity the i-partition of w ends with $(x_i)^\alpha x_{i-1}x_i$. In this case the period of the considered repeats $\alpha+1$ times and its fractional part has the length $q_{i-1}-2$. On the other hand, if i and n have the same parity, the i-partition of w ends with $(x_i)^\alpha x_{i-1}$. Due to Proposition 16, $\alpha=\gamma_i+1$. Moreover, since x_{i-1} is a prefix of x_i, the fractional part of considered run consists of the whole word x_i and has the length q_{i-1}.

Summing up, in the computation of the sum of runs exponents, we count $\gamma_i+1+\frac{q_{i-1}-2}{q_i}$ for each occurrence of a in $w^{(i+1)}$, namely $N_\gamma(i+1)$ times, and an additional 1 for each b in $w^{(i+1)}$ (except the rightmost one), namely $N_\gamma(i+2)-1$ times. Finally, we must take care of the remainder of the rightmost run with period x_i and we obtain the statement of the lemma. See Figure 3 for the illustration of type-2 runs structure in example words and two possible remainders of the rightmost run. ◻

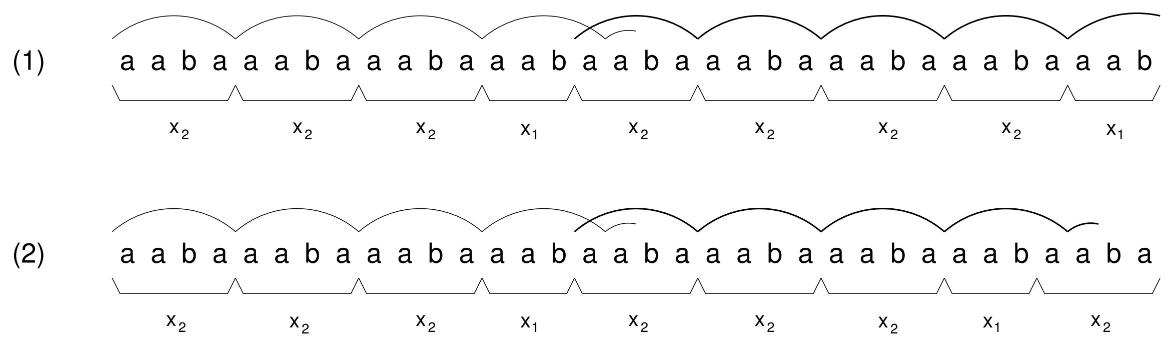

(1)
a a b a a a b a a a b a a a b a a b a a a b a a a b a a a b a a a b
x2 x2 x2 x1 x2 x2 x2 x2 x1
(2)
a a b a a a b a a a b a a a b a a b a a a b a a a b a a a b a a b a
x2 x2 x2 x1 x2 x2 x2 x1 x2

Figure 3. The structure of runs with the period x_2 in a standard word $\mathrm{Sw}(2,1,3,1,1)$ (1) compared to $\mathrm{Sw}(2,1,3,2)$ (2).

Observe that the maximal repetitions with the period of the form $(x_i)^k x_{i-1}$, where $1 \le k < \gamma_i$, appear only for $\gamma_i > 1$. The sum of exponents of such runs is given by the following fact.

Lemma 19.
Let $\gamma = (\gamma_0, \dots, \gamma_n)$ be a directive sequence and $w = \mathrm{Sw}(\gamma_0, \dots, \gamma_n)$ be a standard word. For $1 \le i \le n-1$ the sum of exponents of runs in w with the period $(x_i)^k x_{i-1}$, where $1 \le k < \gamma_i$, equals:

$$\sigma_i''(\gamma) \;=\; \Big(N_\gamma(i+1)-1\Big)\cdot\sum_{k=1}^{\gamma_i-1}\left(2+\frac{q_i-2}{k\cdot q_i+q_{i-1}}\right). \tag{7}$$

Proof.
Let $w = \mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ and $u = (x_i)^k x_{i-1}$, where $1 \leq k < \gamma_i$. Due to Lemma 11, each occurrence of u is aligned to the i-partition of w. Consider a repetition of the form u^m in w and denote it as $u^{(1)}u^{(2)} \cdots u^{(m)}$. Observe that each $u^{(2)}, \ldots, u^{(m)}$ have to be preceded by the suffix of u, namely x_{i-1}. Since each two consecutive occurrences of x_{i-1} in the i-partition of w are separated by at least γ_i occurrences of x_i and $k < \gamma_i$, the factor u cannot have more than two consecutive occurrences. Therefore, the considered run with the period u has the form $u^{(1)}u^{(2)} \cdot v$, where v is a prefix of u.

The suffix x_{i-1} of $u^{(2)}$ starts at the beginning of an x_i block followed by x_{i-1}, which appears either as block of the i-partition of w or as a prefix of a subsequent block x_i. Due to Lemma 13, the considered factor has the form $x_i \cdot x_{i-1} = x_{i-1} \cdot \widetilde{x_i}$. Therefore, the fractional part of the considered run has the length $q_i - 2$.

Observe, that occurrences of $u^{(1)}$ in w are aligned with occurrences of x_{i-1} in the i-partition of w. Therefore, each such occurrence of x_{i-1} (except the rightmost one) corresponds to $\gamma_i - 1$ runs with a period $(x_i)^k x_{i-1}$, for $1 \leq k < \gamma_i$. Due to Proposition 8, each occurrence of x_{i-1} in the i-partition of w corresponds to an occurrence of b in $\mathrm{Sw}(\gamma_i, \ldots, \gamma_n)$. Summing up exponents of all $\gamma_i - 1$ runs for each b in $\mathrm{Sw}(\gamma_i, \ldots, \gamma_n)$ (except the rightmost one), namely $N_\gamma(i+1) - 1$ occurrences, we obtain the statement of the lemma. See Figure 4 for an illustration of the structure of runs of this type. □

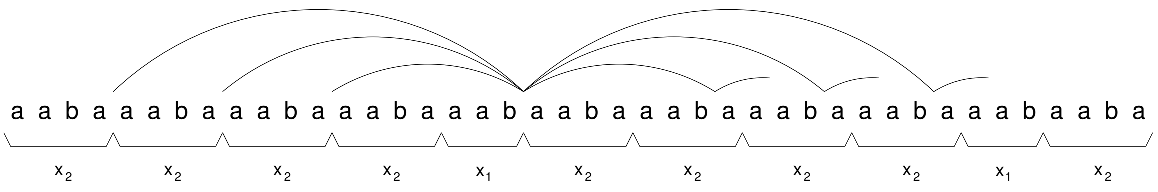

a a b a a a b a a a b a a a b a a a b a a b a a a b a a a b a a a b a a a b a a b a a b a
x2
x2
x2
x2
x1
x2
x2
x2
x2
x1
x2

Figure 4. The structure of runs with the period $(x_2)^k x_1$ $(1 \leq k \leq 3)$ in $\mathrm{Sw}(2, 1, 4, 2)$.

The complete formula for the sum of exponents of all type-i runs can be obtained by combining the formulas from Lemma 18 and Lemma 19.

Lemma 20.
Let $\gamma = (\gamma_0, \ldots, \gamma_n)$ be a directive sequence and $w = \mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ be a standard word. For $2 \leq i \leq n-1$ the sum of exponents of type i runs in w equals:

$$\begin{aligned}\sigma_i(\gamma) = N_\gamma(i+1) \cdot \left(\gamma_i + 1 + \frac{q_{i-1} - 2}{q_i}\right) + \Big(N_\gamma(i+2) - 1\Big) + \Delta_n(i)\frac{2}{q_i} \\ + \Big(N_\gamma(i+1) - 1\Big) \cdot \sum_{k=1}^{\gamma_i - 1} \left(2 + \frac{q_i}{k \cdot q_i + q_{i-1}}\right).\end{aligned} \tag{8}$$

4.2 Boundary cases

For a standard word $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ runs of types 0, 1 and n have to be investigated differently. We start with the analyze of runs of type 0, i.e. the runs with the period of the form a.

Lemma 21 (Type 0).
Let $\gamma = (\gamma_0, \ldots, \gamma_n)$ be a directive sequence and $w = \mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ be a standard

word. The sum of exponents of type 0 runs in w equals:

$$\sigma_0(\gamma) \;=\; \begin{cases} 2\Big(N_\gamma(2) - odd(n)\Big) & \text{for} \quad \gamma_0 = 1 \\ \gamma_0 N_\gamma(1) + N_\gamma(2) - odd(n) & \text{for} \quad \gamma_0 > 1 \end{cases}. \tag{9}$$

Proof.
Each standard word consists of blocks of repeated occurrences of the letter a separated by single occurrences of the letter b. The length of the blocks of the form $a \cdots a$ depends on the value of γ_0.

First assume that $\gamma_0 = 1$. In this case the word $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ consists of the blocks of two types: ab or aab and only the blocks of the second type include the runs with the period a and exponent 2. Due to Lemma 4, every such run in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ corresponds to the letter b followed by the letter a in $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$. Hence, the number of such runs equals the number of blocks ba in $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$.

Recall that for an even length of the directive sequence $|(\gamma_1, \ldots, \gamma_n)|$ (n is even) the word $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$ ends with ba and in this case the number of runs with the period a in $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$ equals the number of the letters b in $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$, namely $N_\gamma(2)$. On the other hand, for an odd length of the directive sequence $|(\gamma_1, \ldots, \gamma_n)|$ (n is odd) the word $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$ ends with ab and the last letter b does not correspond to a run in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$. In this case, the number of runs with the period a in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ is one less than the number of the letters b in $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$, namely $N_\gamma(2) - 1$. Hence, in this case the sum of type-0 runs exponents equals

$$\sigma_0(\gamma) = 2\Big(N_\gamma(2) - odd(n)\Big).$$

Assume now that $\gamma_0 > 1$. Every run with the period a in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ equals a^{γ_0} or a^{γ_0+1} and is followed by the single letter b. Due to Lemma 4, every such run in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ corresponds to the letter a in $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$. Hence in this case we have $N_\gamma(1)$ runs with the period a.

By Lemma 4 each occurrence of a in $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$ preceded by b produces a run a^{γ_0+1} in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$, and each occurrence of a in $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$ not preceded by b produces a run a^{γ_0} in $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$. Therefore, in computation of the sum of runs exponents, we count γ_0 for each a in $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$ and an additional 1 for each b. As in the previous case, for odd n, the rightmost b does not correspond to a run in $\mathrm{Sw}(\gamma_1, \ldots, \gamma_n)$. Therefore, in this case the sum of type-0 runs exponents equals

$$\sigma_0(\gamma) = \gamma_0 N_\gamma(1) + N_\gamma(2) - odd(n).$$

□

The next boundary case, strongly related to the case considered above, is the sum of exponents of runs with the period of the form x_1.

Lemma 22 (Type 1).
Let $\gamma = (\gamma_0, \ldots, \gamma_n)$ be a directive sequence and $w = \mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ be a standard word. The sum of exponents of runs with the period x_1 in w equals:

$$\sigma_1'(\gamma) \;=\; \begin{cases} \Big(N_\gamma(3) - 1\Big) \cdot \left(2 + \dfrac{\gamma_0}{\gamma_0 + 1}\right) + odd(n) \cdot \left(2 + \dfrac{1}{\gamma_0 + 1}\right) & \text{for} \;\; \gamma_1 = 1 \\ N_\gamma(2) \cdot \left(\gamma_1 + \dfrac{\gamma_0}{\gamma_0 + 1}\right) + \Big(N_\gamma(3) - 1\Big) + odd(n) \cdot \dfrac{\gamma_0 - 1}{\gamma_0 + 1} & \text{for} \;\; \gamma_1 > 1 \end{cases}. \tag{10}$$

Proof.
Let $w = \mathrm{Sw}(\gamma_0, \dots, \gamma_n)$. By definition we have $x_1 = a^{\gamma_0}b$. Therefore, the remainder of each internal run with the period x_1 has the length γ_0.

Consider the 1-partition of w. By Lemma 4 occurrences of blocks of the form $a^{\gamma_0}b$ correspond to occurrences of letters a in $\mathrm{Sw}(\gamma_1, \dots, \gamma_n)$ and occurrences of blocks of the form a to occurrences of letters b in $\mathrm{Sw}(\gamma_1, \dots, \gamma_n)$. Therefore, following similar argumentation as in proof of Lemma 21, we obtain the formula for the sum of exponents of internal runs with the period x_1 in w.

Let us now consider the rightmost run with the period x_1 in w. If n is even, w ends with $a \cdot a^{\gamma_0}b$ and this occurrence of x_1 does not correspond to a run in w. On the other hand, if n is odd, due to Proposition 16 w ends with $(a^{\gamma_0}b)^{\gamma_1+1}a$. Such a suffix corresponds to a run with the total part of exponent equal $\gamma_1 + 1$ and the remainder a, and we should include it in our formula. □

The sum of exponents of runs with the period $(x_1)^k x_0$ for $1 \le k < \gamma_1$ follows from Lemma 19. As a final step of investigation we count the sum of exponents of type-n runs.

Lemma 23 (Type n).
Let $w = \mathrm{Sw}(\gamma_0, \dots, \gamma_n)$ be a standard word. The sum of exponents of runs of type n in w is given by the formula:

$$\sigma_n(\gamma) \ = \ \begin{cases} 0 & \text{for} \quad \gamma_n = 1 \\ \gamma_n + \dfrac{q_{n-1}}{q_n} & \text{for} \quad \gamma_n > 1 \end{cases} \tag{11}$$

Proof.
We have $w = (x_n)^{\gamma_n} x_{n-1}$. Therefore, for $\gamma_n = 1$ there is no run of type n in w. On the other hand, for $\gamma_n > 1$, w contains only one run of type n. Its generator – x_n – repeats undivided γ_n times. Moreover, since x_{n-1} is a prefix of x_n, the total exponent of α equals $\gamma_n + \frac{q_{n-1}}{q_n}$. □

Now we can combine the formulas from equations (6), (7), (9), (10) and (11) and obtain the formula from Theorem 17.

4.3 Algorithm

The formulas from equations (6), (7), (9), (10) and (11) lead to simple and efficient algorithm for computation of the sum of runs exponents in any standard word. Its time complexity depends only on the coefficients of the directive sequence, which is the compressed representation of a considered word.

Theorem 24.
Let $\gamma = (\gamma_0, \dots, \gamma_n)$ be a directive sequence and $w = \mathrm{Sw}(\gamma)$ be a standard word. The sum of exponents of maximal repetitions in w can be computed in time $O(\|\gamma\|)$, where $\|\gamma\| = \gamma_0 + \gamma_1 + \dots + \gamma_n$.

Proof.
Observe that, by equations (6), (7), (9), (10) and (11), the value of each formula $\sigma_i(\gamma)$ depends only on coefficients of γ and the values of $N_\gamma(i+1)$, $N_\gamma(i+2)$, q_i and q_{i-1}.

Therefore, we can iterate through all types of runs from 0 to n computing the value of $\sigma_i(\gamma)$ and simultaneously updating the values of $N_\gamma(i+1)$, $N_\gamma(i+2)$, q_i and q_{i-1}. See Algorithm 1 for details.

The main loop of presented algorithm (lines 8-13) performs $n+1$ iterations. The most time consuming part of each iteration is the computation of the sum of exponents of maximal repetitions with the period $(x_i)^k x_{i-1}$ (line 10), namely the component

$$\sum_{k=1}^{\gamma_i-1}\left(2+\frac{q_i-2}{k\cdot q_i+q_{i-1}}\right).$$

It can be done in $O(\gamma_i)$ time. Hence, the time complexity of the whole algorithm is $O(\|\gamma\|)$, where $\|\gamma\| = \gamma_0+\gamma_1+\cdots+\gamma_n$. □

Algorithm 1: *Sum-Of-Exponents*$\big(\mathrm{Sw}(\gamma)\big)$

```
Input:   $\gamma = (\gamma_0, \ldots, \gamma_n)$
Output:  $\sigma(\gamma)$

1  $result \longleftarrow 0$;
2  $N_\gamma(n+1) \longleftarrow 1$;
3  $N_\gamma(n+2) \longleftarrow 0$;
4  $q_0 \longleftarrow 1$;
5  $q_{-1} \longleftarrow 1$;

6  for $i := 1$ to $n$ do
7      $(q_{i+1}, q_i) \longleftarrow (\gamma_i \cdot q_i, q_i)$;

8  for $i := n$ downto $0$ do
9      compute $\sigma'_i(\gamma)$;    // runs with period $x_i$;
10     compute $\sigma''_i(\gamma)$;   // runs with period $(x_i)^k x_{i-1}$;
11     $result \longleftarrow result + \sigma'_i(\gamma) + \sigma''_i(\gamma)$;
12     $(q_i, q_{i-1}) \longleftarrow (q_{i-1}, q_i - \gamma_{i-1} \cdot q_{i-1})$;
13     $(N_\gamma(i), N_\gamma(i+1)) \longleftarrow (\gamma_i \cdot N_\gamma(i) + N_\gamma(i+1), N_\gamma(i))$;

14 return $result$;
```

Final remarks

The aim of this paper was to study problems related to repetitions in standard Sturmian words – one of the most thoroughly investigated class of strings in combinatorics of words. We presented the formulas for the sum of exponents of maximal repetitions in any standard word $\mathrm{Sw}(\gamma_0,\ldots,\gamma_n)$ that depend only on its compressed representation (the directive sequence). We proposed also an algorithm based on those formulas that computes the sum of runs exponents in any standard word in linear time with respect to the (total) size of the directive sequence, i.e. in time $O(\|\gamma\|)$, where $\|\gamma\| = \gamma_0+\gamma_1+\cdots+\gamma_n$.

The notion of total run length (TRL) proposed in [12] can be considered similarly. To obtain the formulas for the total run length of a standard word we can use modified formulas for the sum of runs exponents. We only needed to multiply the total part of each exponent by the length of related period (either q_i or $k\cdot q_i+q_{i-1}$) and remove the denominator from its fractional part. The described change could be also taken into account in the presented algorithm.

In the case of the total run length computation, the component

$$\sum_{k=1}^{\gamma_i-1}\left(2+\frac{q_i-2}{k\cdot q_i+q_{i-1}}\right)$$

of equation 7 has the form

$$\sum_{k=1}^{\gamma_i-1}\Bigg((k+1)q_i+q_{i-1}-2\Bigg).$$

The above formula is a sum of an arithmetic progression, hence it can be simplified as

$$(\gamma_i-1)\frac{(\gamma_i+2)q_i+2q_i-4}{2}.$$

Therefore, in each iteration of the main loop of the modified algorithm, we have to compute the value of a single arithmetic formula and update the values of $N_\gamma(i+1)$, $N_\gamma(i+2)$, q_i and q_{i-1}. This way we obtain the algorithm computing the total run length of any standard word $\mathrm{Sw}(\gamma)$ in time $O(|\gamma|)$, where $|\gamma|$ denotes the length of the directive sequence.

References

1. J. Allouche and J. Shallit: *Automatic Sequences. Theory, Applications, Generalizations.*, Cambridge University Press, 2003.
2. P. Baturo, M. Piątkowski, and W. Rytter: *The Number of Runs in Standard Sturmian Words.* Electronic Journal of Combinatorics, 20(1) 2013.
3. P. Baturo and W. Rytter: *Compressed string-matching in standard Sturmian words.* Theoretical Computer Science, 410(30–32) 2009, pp. 2804–2810.
4. J. Berstel: *Sturmian and Episturmian words: a survey of some recent results*, in Proceedings of the 2nd international conference on Algebraic informatics, vol. 4728 of Lecture Notes in Computer Science, Springer, 2007, pp. 23–47.
5. J. Berstel, A. Lauve, C. Reutenauer, and F. Saliola: *Combinatorics on Words: Christoffel Words and Repetitions in Words*, CRM monograph series, Providence, R.I: American Mathematical Society, 2009.
6. M. Crochemore and L. Ilie: *Analysis of maximal repetitions in strings*, in Proceedings of the 32nd International Conference on Mathematical Foundations of Computer Science, vol. 4708 of Lecture Notes in Computer Science, Springer, 2007, pp. 465–476.
7. M. Crochemore, L. Ilie, and L. Tinta: *Towards a solution of the "runs" conjecture*, in Proceedings of the 19th annual symposium on Combinatorial Pattern Matching, vol. 5029 of Lecture Notes in Computer Science, Springer, 2008, pp. 290–302.
8. D. Damanik and D. Lenz: *The index of sturmian sequences.* European Journal of Combinatorics, 23(1) 2002, pp. 23–29.
9. D. Damanik and D. Lenz: *Powers in Sturmian sequences.* European Journal of Combinatorics, 24(4) 2003, pp. 377–390.
10. F. Franek, R. J. Simpson, and W. F. Smyth: *The maximum number of runs in a string*, in Proceedings of 14th Australian Workshop on Combinatiorial Algorithms, 2003, pp. 26–35.
11. F. Franek and Q. Yang: *An asymptotic lower bound for the maximal number of runs in a string.* Jnternational Journal of Foundations of Computer Science, 19(1) 2008, pp. 195–203.
12. A. Glen and J. Simpson: *The total run length of a word.* arXiv:1301.6568, 2013.
13. R. Kolpakov and G. Kucherov: *On the sum of exponents of maximal repetitions in a word*, Tech. Rep. 99-R-034, LORIA, 1999.

14. K. Kusano, W. Matsubara, A. Ishino, H. Bannai, and A. Shinobara: *New lower bound for the maximum number of runs in a string.* Computing Research Repository, abs/0804.1214 2008.
15. M. Lothaire: *Algebraic Combinatorics on Words*, vol. 90 of Encyclopedia of mathematics and its application, Cambridge University Press, 2002.
16. M. Piątkowski: *Stringology applets* . `http://www.mat.umk.pl/~martinp/stringology/applets`.
17. M. Sciortino and L. Zamboni: *Suffix automata and standard Sturmian words*, in Proceedings of the 11th International Conference on Developments in Language Theory, vol. 4588 of Lecture Notes in Computer Science, Springer, 2007, pp. 382–398.
18. J. Shallit: *Characteristic words as fixed points of homomorphisms*, Tech. Rep. CS-91-72, University of Waterloo, Department of Computer Science, 1991.

Finding Distinct Subpalindromes Online

Dmitry Kosolobov, Mikhail Rubinchik, and Arseny M. Shur

Institute of Mathematics and Computer Science, Ural Federal University,
Ekaterinburg, Russia
dkosolobov@mail.ru, mikhail.rubinchik@gmail.com, arseny.shur@usu.ru

Abstract. We exhibit an online algorithm finding all distinct palindromes inside a given string in time $\Theta(n \log |\Sigma|)$ over an ordered alphabet and in time $\Theta(n|\Sigma|)$ over an unordered alphabet. Using a reduction from a dictionary-like data structure, we prove the optimality of this algorithm in the comparison-based computation model.

Keywords: stringology, counting palindromes, subpalindromes, palindromic closure, online algorithm

1 Introduction

A *palindrome* is a string that is equal to its reversal. Palindromes are among the most interesting text regularities. During the last few decades, many algorithmic and combinatorial problems concerning palindromes were considered. For example, in the field of combinatorics on words the well-known *Sturmian words* are characterized by their palindromic complexity ([2], [4]). The *rich words* are studied in [5] (a word w with $|w|+1$ distinct subpalindromes is called *rich*). The class of rich words includes the *episturmian words* introduced in [3].

Among the algorithmic problems about palindromes, we should mention PALSTAR (check whether a word is a product of nontrivial palindromes), the problems of splitting words into a given number of palindromes, and the problems of enumerating palindromes occurring in a given word. Some results on these problems are surveyed in [1]. In this paper, we solve one enumeration problem.

There is a well known online algorithm by Manacher [7] that finds all maximal subpalindromes of a string in linear time and linear space (by a "subpalindrome" we mean a substring that is a palindrome). It is known [3] that every string of length n contains at most $n+1$ distinct subpalindromes, including the empty string. The following question arises naturally: *can one find all distinct subpalindromes of a string in linear time and space?* In [6], this question was answered in the affirmative, but with an offline algorithm. The authors stated the existence of the corresponding online algorithm as an open problem. Our main contribution is the following result.

Theorem 1. *Let Σ be a finite unordered (resp., ordered) alphabet. There exists an online algorithm which finds all distinct subpalindromes in a string over Σ in $O(n|\Sigma|)$ (resp., $O(n \log |\Sigma|)$) time and linear space. This algorithm is optimal in the comparison based computation model.*

As a by-product, we get an online linear time and space algorithm that finds, for all prefixes of a string, the lengths of their maximal suffix-palindromes and of their palindromic closures.

2 Notation and Definitions

An alphabet Σ is a finite set of letters. A *string* w over Σ is a finite sequence of letters. It is convenient to consider a string as a function $w : \{1, 2, \ldots, l\} \to \Sigma$. A *period* of w is any period of this function. The number l is the *length* of w, denoted by $|w|$. We write $w[i]$ for the i-th letter of w and abbreviate $w[i]w[i{+}1] \cdots w[j]$ by $w[i..j]$. A *substring* of w is any string u such that $u = w[i..j]$ for some i and j. Each occurrence of the substring u in w is determined by its *position* i. If $i = 1$ (resp. $j = |w|$), then u is a *prefix* (resp. *suffix*) of w. A prefix (resp. suffix) of a string w is called *proper* if it is not equal to w. The string $w[|w|]w[|w|{-}1] \cdots w[1]$ is the *reversal* of w, denoted by $\overleftarrow{w}$. A string is a *palindrome* if it coincides with its reversal. A palindrome of even (resp. odd) length is referred to as an *even* (resp. *odd*) palindrome. If a substring, a prefix or a suffix of a string is a palindrome, we call it a *subpalindrome*, a *prefix-palindrome*, or a *suffix-palindrome*, respectively. The *palindromic closure* of a string w is the shortest palindrome w' such that w is a prefix of w'.

Let $w[i..j]$ be a subpalindrome of w. The number $\lfloor (i{+}j)/2 \rfloor$ is the *center* of $w[i..j]$, and the number $\lfloor (j{-}i{+}1)/2 \rfloor$ is the *radius* of $w[i..j]$. Thus, a single letter and the empty string are palindromes of radius 0. Note that the center of the empty subpalindrome is the previous position of the string.

By an *online algorithm* for an algorithmic problem concerning strings we mean an algorithm that processes the input string w sequentially from left to right, and answers the problem for each prefix $w[1..j]$ of w after processing the letter $w[j]$.

3 Distinct subpalindromes

3.1 Suffix-Palindromes and Palindromic Closure

The problem of finding the lengths of palindromic closures for all prefixes of a string is closely related to the problem of finding all distinct subpalindromes of this string. It was conjectured in [6] that there exists an online linear time algorithm for the former problem.

Let v be the maximal suffix-palindrome of $w = uv$. It is easy to see that the palindromic closure of w equals to the string $uv\overleftarrow{u}$. An offline algorithm for finding the maximal suffix-palindromes for each prefix of the string can be found, e. g., in [1, Ch. 8]. Our online algorithm is a modification of Manacher's algorithm (see [7]).

We construct a data structure based on Manacher's algorithm. Let Δ be a boolean flag (needed to distinguish between odd and even palindromes). The data structure, denoted by man, contains a string *text* and supports the procedure man.AddLetter(c) adding a letter to the end of *text*. The function man.MaxPal returns the length of maximal odd/even (according to $\Delta = 0/1$) suffix-palindrome of *text*.

Our data structure uses the following internal variables:
n, which is the length of *text*;
i, which is the center of the maximal odd/even (according to $\Delta = 0/1$) suffix-palindrome of *text*;
Rad, which is an array of integers such that for any $j < i$ the value *Rad*[j] is equal to the radius of the maximal odd/even (according to $\Delta = 0/1$) subpalindrome with the center j. The main property of *Rad* is expressed in the following lemma (see [1, Lemma 8.1]).

Lemma 2. *Let k be an integer, $1 \le k \le Rad[i]$.*
(1) If $Rad[i{-}k] < Rad[i] - k$ then $Rad[i{+}k] = Rad[i{-}k]$;
(2) if $Rad[i{-}k] > Rad[i] - k$ then $Rad[i{+}k] = Rad[i] - k$.

At the beginning, Rad is filled with zeros, $n = 1$, $i = 2$, $text$ = "\$", where \$ is a special letter that does not appear in the input string[1].

```
1: procedure man.AddLetter($c$)
2:     $s \leftarrow i - Rad[i] + \Delta$                ▷ position of the max suf-pal of $text[1..n]$
3:     $text[n+1] \leftarrow c$
4:     while $i + Rad[i] \leqslant n$ do
5:         $Rad[i] \leftarrow \min(Rad[s{+}n{-}i{-}\Delta], n - i)$      ▷ this is $Rad[i]$ in $text[1..n]$
6:         if $i + Rad[i] = n$ and $text[i{-}Rad[i]{-}1{+}\Delta] = c$ then
7:             $Rad[i] \leftarrow Rad[i] + 1$              ▷ extending the max suf-pal
8:             break                               ▷ max suf-pal of $text[1..n{+}1]$ found
9:         $i \leftarrow i + 1$                    ▷ next candidate for the center of max suf-pal
10:    $n \leftarrow n + 1$
11: function man.MaxPal
12:     return $2Rad[i] + 1 - \Delta$
```

Theorem 3. *There exists an online linear time and space algorithm that finds the lengths of the maximal suffix-palindromes of all prefixes of a string.*

Proof. From the correctness of Manacher's algorithm (see [7]) and Lemma 2 it follows that the function man.MaxPal correctly returns the length of the maximal odd/even suffix palindrome of the processed string. For a string of length n, we call the procedure man.AddLetter n times with the parameter $\Delta = 0$ and n times with $\Delta = 1$. If one call of the procedure uses k iterations of the loop in the lines 4–9, then the value of i increases by $k{-}1$. Hence, the loop is used at most $4n$ times in total. Apart from this loop, man.AddLetter performs a constant number of operations. This gives us the required $O(n)$ time bound.

Corollary 4. *There exists an online linear time and space algorithm that finds the lengths of palindromic closured of all prefixes of a string.*

Example 5. Let $w = abadaadcaa$ and consider the state of the data structure man after the sequence of calls man.AddLetter($w[i]$), $i = 1, 2, \ldots, 10$.

$$text = \$w;$$
$$Rad = (0,1,0,1,0,0,0,0,0,0) \text{ for } \Delta = 0;$$
$$Rad = (0,0,0,0,2,0,0,0,1,0) \text{ for } \Delta = 1;$$

The calls to man.MaxPal after each call to man.AddLetter($w[i]$) return consequently the values $1,1,3,1,3,1,1,1,1,1$ for the case $\Delta = 0$ and $0,0,0,0,0,2,4,0,0,2$ for the case $\Delta = 1$.

3.2 Distinct subpalindromes

We make use of the following

[1] The strange-looking initial value of i provides the correct processing of the first letter after \$ (the while loop will be skipped and the correct values $n = i = 2$ for the next iteration will be obtained).

Lemma 6 ([6]). *Each subpalindrome of a string is the maximal suffix-palindrome of some prefix of this string.*

This lemma implies that the online algorithm designed in Sect. 3.1 finds all subpalindromes of a string. To find all *distinct* subpalindromes, we have to verify whether the maximal suffix-palindrome of a string has another occurrence in this string. Note that the direct comparison of substrings for this purpose leads to at least quadratic overall time. Instead, we will use a version of suffix tree known as *Ukkonen's tree.* To introduce it, we need some definitions.

A *trie* is a rooted labelled tree in which every edge is labelled with a letter such that all edges leading from a vertex to its children have different labels. Each vertex of the trie is associated with the string labelling the path from the root to this vertex. A trie can be "compressed" as follows: any non-branching descending path is replaced by a single edge labelled by the string equal to the label of this path. The result of this procedure is called a *compressed trie.* For a set S of strings, the *compressed trie of* S is defined by the following two properties: (i) for each string of S, there is a vertex associated it and (ii) the trie has the minimal number of vertices among all compressed tries with property (i).

A (compressed) *suffix tree* is the compressed trie of the set of all suffixes of a string. *Ukkonen's tree* is the data structure ukk containing a string and the suffix tree of this string (labels are stored as pairs of positions in the string). Ukkonen's tree allows one to add a letter to the end of the string (procedure ukk.addLetter(c)), updating the suffix tree. We also need the following parameter: the length of the minimal suffix of the processed string such that this suffix occurs in this string only once (function ukk.minUniqueSuff). Let us recall some implementation details of Ukkonen's tree for the efficient implementation of ukk.minUniqueSuff.

The update of Ukkonen's tree is based on the system of suffix links. Such a link connects a vertex associated with a word v to the vertex associated with the longest proper suffix of v. These links are also defined for "implicit" vertices (the vertices that are not in the compressed trie, but present in the corresponding trie). In particular, Ukkonen's tree supports the triple (v, e, i) such that

(1) v is a vertex (associated with some string s') of the current suffix tree,
(2) e is an edge (labelled by some string s) between v and its child,
(3) i is an integer between 0 and $|s|$,

with the property that $s's[1..i]$ is the longest suffix of the processed string that occurs in this string at least twice. This triple is crucial for fast update of Ukkonen's tree (for further details, see [8]).

Lemma 7 ([8]). *The procedure* ukk.addLetter(c) *performs n calls using $O(n)$ space and $O(n \log |\Sigma|)$ (resp., $O(n|\Sigma|)$) time in the case of ordered (resp., unordered) alphabet.*

We modify Ukkonen's tree, associating with each vertex u an additional field u.depth to store the length of the string associated with u. Maintaining this field requires a constant number of operations at the moment when u is created. Thus, this update adds $O(n)$ time and $O(n)$ space to the total cost of maintaining Ukkonen's tree. Thus, Lemma 7 holds for the modified Ukkonen's tree as well. It remains to note that ukk.minUniqueSuff $= v.\text{depth} + i + 1$.

Proof (Theorem 1: existence). The following algorithm solves the problem and has the required complexity. The algorithm uses data structures man and ukk, processing the same input string w. The next (say, nth) symbol of w is added to both structures through the procedures man.AddLetter and ukk.AddLetter. After this, we call man.MaxPal to get the length of the maximal palindromic suffix of $w[1..n]$ and ukk.MinUniqueSuff to get the length of the shortest suffix of $w[1..n]$ that never occurred in w before. The inequality man.MaxPal $\geq$ ukk.MinUniqueSuff means the detection of a new palindrome; we get its first and last positions from the structure man and output them. In the case of the opposite inequality, there is no new palindrome, and we output "—".

The required time and space bounds follow from Theorem 3 and Lemma 7.

Example 8. Consider the string $w = abadaadcaa$ again. We get the following results for $i = 1, 2, \dots, 10$:

man.MaxPal :	1	1	3	1	3	2	4	1	1	2
ukk.MinUniqueSuff :	1	1	2	1	2	2	3	1	2	3
output :	1–1	2–2	1–3	4–4	3–5	5–6	4–7	8–8	—	—

3.3 Lower bounds

Recall that a *dictionary* is a data structure D containing some set of elements and designed for the fast implementation of basic operations like checking the membership of an element in the set, deleting an existing element, or adding a new element. Below we consider an *insert-only dictionary over a set S*. In each moment, such a dictionary D contains a subset of S and supports only the operation insqry(x). This operation checks whether the element $x \in S$ is already in the dictionary; if no, it adds x to the dictionary.

Lemma 9. *Suppose that the alphabet Σ consists of indivisible elements, $n \geq |\Sigma|$, and the insert-only dictionary D over Σ is initially empty. Then the sequence of n calls of* insqry *requires, in the worst case, $\Omega(n \log |\Sigma|)$ time if Σ is ordered and $\Omega(n|\Sigma|)$ if Σ is unordered.*

Proof. Let $\Sigma = \{a_1 < a_2 < \cdots < a_m\}$ be an ordered alphabet. Assume that on some stage all letters with even numbers are in the dictionary, while all elements with odd numbers are not. Consider the next operation. In the comparison-based computation model, a query "$x \in D$?" is answered by some decision tree; each node of this tree is marked by the condition "$x < a_i$" for some i. To distinguish between a_i and a_{i+1}, the tree should contain the nodes for both a_i and a_{i+1}. Now note that for any i, exactly one of the letters a_i and a_{i+1} belongs to D. So, to answer correctly all possible queries "$x \in D$?" the decision tree should have nodes for all letters. Then the depth of this tree is $\Omega(\log m)$. Therefore, for some element $x = a_{2i}$ the number of comparisons needed to prove that $x \in D$ is $\Omega(\log m)$. After processing x, the content of the dictionary remains unchanged. The decision tree can change, but it does not matter: we again choose the next letter to be the one having an even number and requiring $\Omega(\log m)$ comparisons to prove its membership in D. Thus, our "bad" sequence of calls is as follows: it starts with insqry$(a_2), \dots,$ insqry$(a_{2\lfloor m/2 \rfloor})$, and continues with the "worst" letter, described above, on each next step. Even if the first $\lfloor m/2 \rfloor$ calls can be performed in $O(1)$ time each, the overall time is $\Omega(n \log m)$, as required.

In the case of unordered alphabet all conditions in the decision tree have the form "$x = a_i$". It is clear that if the dictionary contains $\lfloor m/2 \rfloor$ elements, the maximal number of comparisons equals $\lfloor m/2 \rfloor$ as well. Choosing the bad sequence of calls in the same way as for the ordered alphabet, we arrive at the required bound $\Omega(nm)$.

Before finishing the proof of Theorem 1 we mention the following lemma. Its proof is obvious.

Lemma 10. *Suppose that a, b are two different letters and $w = abx_1abx_2 \cdots abx_n$ is a string such that each x_i is a letter different from a and b. Then all nonempty subpalindromes of w are single letters.*

Proof (Proof of Theorem 1: lower bounds). We prove the required lower bounds reducing the problem of maintaining an insert-only dictionary to counting distinct palindromes in a string. Assume that we have a black box algorithm that processes an input string letter by letter and outputs, after each step, the number of distinct palindromes in the string read so far. The time complexity of this algorithm depends on the length n of the string at least linearly, and a linear in n algorithm does exist, as we have proved in the Sect. 3.2. Thus, we can assume that the considered black box algorithm works in time $O(n \cdot f(m))$, where m is the size of the alphabet of the processed string and the function $f(m)$ is non-decreasing.

The insert-only dictionary over a set Σ of size $m > 1$ can be maintained as follows. We pick up two letters $a, b \in \Sigma$ and mark their presence in the dictionary using two boolean variables, z_a and z_b. All other letters are processed with the aid of the mentioned black box. Let us describe how to process a sequence of n calls insqry$(x_1), \ldots,$ insqry(x_n) starting from the empty dictionary.

For each call, we first compare the current letter x_i to a and b. If $x_i = a$, then z_a is the answer to the query "$x_i \in D$?"; after answering the query we set $z_a = 1$. The case $x_i = b$ is managed in the same way.

If $x_i \notin \{a, b\}$, we feed the black box with a, b, and x_i (in this order). Then we get the output of the black box and check whether the number of distinct subpalindromes in its input string increased. By Lemma 10, the increase happens if and only if x_i appears in the input string of the black box for the first time. Thus, we can immediately answer the query "$x_i \in D$?", and, moreover, x_i is now in the dictionary.

The described algorithm performs the sequence of calls insqry$(x_1), \ldots,$ insqry(x_n) in time $O(n)$ plus the time used by the blackbox to process a string of length $\leq 3n$ over Σ. Hence, the overall time bound is $O(n \cdot f(m))$. In view of Lemma 9 we obtain $f(m) = \Omega(\log m)$ (resp., $f(m) = \Omega(m)$) in the case of ordered (resp., unordered) alphabet Σ. The required lower bounds are proved.

4 Conclusion

Our approach shows that it is hardly possible to design a linear time and space online algorithm for the discussed problem even in stronger natural computation models such as the word-RAM model or cellprobe model. The reason is the resource restrictions of dictionaries. However, up to the moment we have proved no nontrivial lower bounds for the insert-only dictionary in more sophisticated models than the comparison based model.

References

1. M. Crochemore and W. Rytter: *Jewels of Stringology*, World Scientific Publishing Co. Pte. Ltd., 2002.
2. A. de Luca: *Sturmian words: structure, combinatorics and their arithmetics.* Theoret. Comput. Sci., 183 1997, pp. 45–82.

3. X. Droubay, J. Justin, and G. Pirillo: *Episturmian words and some constructions of de luca and rauzy*. Theoret. Comput. Sci., 255 2001, pp. 539–553.
4. X. Droubay and G. Pirillo: *Palindromes and sturmian words*. Theoret. Comput. Sci., 223 1999, pp. 73–85.
5. A. Glen, J. Justin, S. Widmer, and L. Zamboni: *Palindromic richness*. European Journal of Combinatorics, 30 2009, pp. 510–531.
6. R. Groult, E. Prieur, and G. Richomme: *Counting distinct palindromes in a word in linear time*. Inform. Process. Lett., 110 2010, pp. 908–912.
7. G. Manacher: *A new linear-time on-line algorithm finding the smallest initial palindrome of a string*. J. ACM, 22(3) 1975, pp. 346–351.
8. E. Ukkonen: *On-line construction of suffix trees*. Algorithmica, 14(3) 1995, pp. 249–260.

Maximal Palindromic Factorization

Ali Alatabbi[1,⋆], Costas S. Iliopoulos[2], and M. Sohel Rahman[1,2,⋆⋆]

[1] Department of Informatics, King's College London, London WC2R 2LS, United Kingdom
[2] AℓEDA Group, Department of CSE, BUET, Dhaka-1000, Bangladesh

Abstract. A palindrome is a symmetric string, phrase, number, or other sequence of units sequence that reads the same forward and backward.
We present an algorithm for maximal palindromic factorization of a finite string by adapting an Gusfield algorithm [15] for detecting all occurrences of maximal palindromes in a string in linear time to the length of the given string then using the breadth first search (BFS) to find the maximal palindromic factorization set.
A factorization $\mathcal{F}$ of s with respect to $\mathcal{S}$ refers to a decomposition of s such that $s = s_{i_1} s_{i_2} \cdots s_{i_\ell}$ where $s_{i_j} \in \mathcal{S}$ and ℓ is minimum. In this context the set $\mathcal{S}$ is referred to as the factorization set. In this paper, we tackle the following problem. Given a string s, find the maximal palindromic factorization of s, that is a factorization of s where the factorization set is the set of all center-distinct maximal palindromes of a string s $\mathcal{MP}(s)$.

Keywords: palindromes, factorization, graph search

1 Introduction

A palindrome is a symmetric word that reads the same backward and forward. The detection of palindromes is a classical and well-studied problem in computer science, language theory and algorithm design with a lot of variants arising out of different practical scenarios. String and sequence algorithms related to palindromes have long drawn attention of stringology researchers [1,12,17,22,25,26,27,29]. Interestingly, in the seminal Knuth-Morris-Pratt paper presenting the well-known string matching algorithm [19], a problem related to palindrome recognition was also considered. In word combinatorics, for example, studies have investigated the inhabitation of palindromes in Fibonacci words or Sturmian words in general [10], [11], [14].

Manacher discovered an on-line sequential algorithm that finds all *initial* palindromes in a string [25]. A string $X[1 \dots n]$ is said to have an initial palindrome of length k if the prefix $S[1 \dots k]$ is a palindrome. Gusfield gave a linear-time algorithm to find all *maximal* palindromes (a notion we define shortly) in a string [16]. Porto and Barbosa gave an algorithm to find all *approximate* palindromes in a string [29]. Matsubara et al. solved in [27] the problem of finding all palindromes in SLP (Straight Line Programs)-compressed strings. Additionally, a number of problems on variants of palindromes have also been investigated in the literature [17,4,22]. Very recently, I *et al.* [18] worked on pattern matching problems and Chowdhury et al. [6] studied the longest common subsequence problem involving palindromes.

In this paper, we present a linear-time algorithm for computing the *maximal palindromic factorization (MPF)* of a string, that is the smallest set (minimum number of palindromic factors), such that the string is covered by that set of factors with no

[⋆] Partially supported by a Commonwealth Fellowship.
[⋆⋆] ⋆, Ali Alatabbi.

overlaps. This problem was very recently posed as an open problem in Stringology at [30].

Generic factorization process plays an important role in String Algorithms. The obvious advantage of such process is that when processing a string online, the work done on an element of the factorization can usually be skipped because already done on its previous occurrence [8]. A typical application of this concept resides in algorithms to compute repetitions in strings, such as Kolpakov and Kucherov algorithm for reporting all maximal repetitions [21], Lyndon factorization [28], have been applied in: string matching [9,2], the Burrows-Wheeler Transform [3] and LempelZiv factorization [32] have been applied in: data compression [7,13] and indeed it seems to be the only technique that leads to linear-time algorithms independently of the alphabet size [8]. Words with palindromic structure are important in DNA and RNA sequences, Biologists believe that palindromes play an important role in regulation of gene activity and other cell processes because these are often observed near promoters, introns and specific untranslated regions. Palindromic structure in DNA and RNA sequences reflects the capacity of molecules to fold [20], i.e. to form double-stranded stems, which insures a stable state of those molecules with low free energy. Identifying palindromes could help in advancing the understanding of genomic instability [5], [24], [31]. Finding common palindromes in two gene sequences can be an important criterion to compare them, and also to find common relationships between them. However, in those applications, the reversal of palindromes should be combined with the complementarity concept on nucleotides, where c is complementary to g and a is complementary to t (or to u, in case of RNA). Moreover, gapped palindromes are biologically meaningful, i.e. contain a spacer between left and right copies (see [20]). Therefore, detecting palindromes in DNA sequences is one of the challenging problems in computational biology. Researchers have also shown that based on palindrome frequency, DNA sequences can be discriminated to the level of species of origin [23]. So, finding common palindromes in two DNA sequences can be an important criterion to compare them, and also to find common relationships between them.
The rest of the paper is organized as follows. In Section 2 we give some definitions and introduce the notations used in the rest of the paper. In Section 3, we describe our algorithm for computing the maximal palindromic factorization of a given string. Finally, We will prove correctness of the algorithm and analyze its running time in Section 4 and we briefly conclude in Section 5 with some future proposals.

2 Notation and terminology

A string or sequence is a succession of zero or more symbols from an alphabet Σ of cardinality σ, where σ expresses the number of distinct characters in the alphabet. The empty string is the empty sequence (of zero length) and is denoted by ϵ. The set of all strings over the alphabet Σ including ϵ is denoted by Σ^*. The set of all non-empty strings over the alphabet Σ is denoted by Σ^+. $\Sigma^* = \Sigma^+ \cup \epsilon$. A string s of length $|s| = n$ is represented by $s[1 \ldots n]$. The i-th symbol of s is denoted by $s[i]$. A string y is a factor of s if $s = xyz$ for $x, z \in \Sigma^*$; it is a prefix of s if x is empty and a suffix of s if z is empty. We denote by $s[i \ldots j]$ the factor of s that starts at position i and ends at position j. We denote by $\tilde{s}$ the reversal of s, i.e., $\tilde{s} = s[n]\ s[n-1]\ \cdots\ s[1]$.

A palindrome is a symmetric string that reads the same forward and backward. More formally, s is called a palindrome if and only if $s = \tilde{s}$. The empty string ϵ is assumed to be a palindrome. Also note that a single character is a palindrome by definition. The following is another (equivalent) definition of a palindrome which indicates that palindrome can be of both odd and even length. A string s is a palindrome if $s = xa\tilde{x}$ where x is a string and a is either a single character or the empty string ϵ. Clearly, if a is a single character, then s is a palindrome having odd length; otherwise, it is of even length.

The radius of a palindrome s is $\frac{\lfloor s \rfloor}{2}$. In the context of a string, if we have a substring that is a palindrome, we often call it a palindromic substring. Given a string s of length n, suppose $s[i \dots j]$,with $1 \leq i \leq j \leq n$ is a palindrome, i.e., $s[i \dots j]$ is a palindromic substring of s. Then, the center of the palindromic substring $s[i \dots j]$ is $\lfloor \frac{i+j}{2} \rfloor$. A palindromic substring $s[i \dots j]$ is called the maximal palindrome at the center $\lfloor \frac{i+j}{2} \rfloor$ if no other palindromes at the center $\lfloor \frac{i+j}{2} \rfloor$ have a larger radius than $s[i \dots j]$, i.e., if $s[i-1] \neq s[j+1]$, where $i = 1$, or $j = n$. A maximal palindrome $s[i \dots j]$ is called a suffix (prefix) palindrome of s if and only if $j = n$ $(i = 1)$. We denote by $(c, r)_s$ the maximal maximal palindromic factor of a string s whose center is c and radius is r; we usually drop the subscript and use (c, r) when the string s is clear from the context. The set of all center-distinct maximal palindromes of a string s is denoted by $\mathcal{MP}(s)$. Further, for the string s, we denote the set of all *prefix palindromes* (*suffix palindromes*) as $\mathcal{PP}(s)$ $(\mathcal{SP}(s))$. We use the following result from [25,16].

Theorem 1 ([25,16]). *For any string s of length n, $\mathcal{MP}(s)$ can be computed in $O(n)$ time.*

In what follows, we assume that the elements of $\mathcal{MP}(s)$ are sorted in increasing order of centers c. Actually, the algorithm of [25] computes the elements of $\mathcal{MP}(s)$ in this order. Clearly, the set $\mathcal{PP}(s)$ and $\mathcal{SP}(s)$ can be computed easily during the computation of $\mathcal{MP}(s)$.

Suppose, we are given a set of strings $\mathcal{S} = \{s_1, s_2, \dots, s_k\}$, such that s_i is a substring of s and $1 \leq i \leq k$. A factorization $\mathcal{F}$ of s with respect to $\mathcal{S}$ refers to a decomposition of s such that $s = s_{i_1} s_{i_2} \cdots s_{i_\ell}$ where $s_{i_j} \in \mathcal{S}$ and ℓ is minimum. In this context the set $\mathcal{S}$ is referred to as the factorization set. In this paper, we tackle the following problem.

Problem 2. (Maximal Palindromic Factorization (MPF)) Given a string s, find the maximal palindromic factorization of s, that is a factorization of s where the factorization set is $\mathcal{MP}(s)$.

3 The Algorithm

In this section we present an algorithm to compute the maximal palindromic factorization of a given string s. We first present some notions required to present our algorithm. First of all, recall that we use $\mathcal{MP}(s)$ to denote the set of center distinct maximal palindromes of s. We further extend this notation as follows. We use $\mathcal{MP}(s)[i]$, where $1 \leq i \leq n$ to denote the set of maximal palindromes with center i.

Proposition 3. *The position i could be the center of at most two maximal palindromic factors, therefore; $\mathcal{MP}(s)[i]$ contains at most two elements, where $1 \leq i \leq n$, hence; there are at most $2n$ elements in $\mathcal{MP}(s)$.*

On the other hand, we use $\mathcal{MPL}(s)[i]$ to denote the set of the lengths of all maximal palindromes ending at position i,where $1 \leq i \leq n$ in s.

$$\mathcal{MPL}(s)[i] = \{2\ell - 1\ |s[i-\ell+1 \ldots i+\ell-1] \in \mathcal{MP}(s)\} \\ \cup \{2\ell'\ |s[i-\ell' \ldots i+\ell'-1] \in \mathcal{MP}(s)\} \quad (1)$$

where $1 \leq i \leq n$, with 2ℓ and $2\ell' + 1$ are the lengths of the odd and even palindromic factors respectively.

Proposition 4. *The set $\mathcal{MPL}(s)$ (Equation 1) can be computed in linear time from the set $\mathcal{MP}(s)$.*

Now we define the list $\mathcal{U}(s)$ such that for each $1 \leq i \leq n$, $\mathcal{U}(s)[i]$ stores the position j such that $j+1$ is the starting position of a maximal palindromic factors ending at i and j is the end of another maximal palindromic substring.

Clearly, this can be easily computed once we have $\mathcal{MPL}(s)$ computed.

$$\mathcal{U}[i][j] = i - \mathcal{MPL}(s)[i][j] \quad (2)$$

One can observe, from 3, that the sets $\mathcal{MPL}(s)$ and $\mathcal{U}(s)$ contain at most $2n$ elements.

Given the list $\mathcal{U}(s)$ for a string s, we define a directed graph $\mathcal{G}_s = (\mathcal{V}, \mathcal{E})$ as follows. We have $\mathcal{V} = \{i \mid 1 \leq i \leq n\}$ and $\mathcal{E} = \{(i,j) \mid j \in \mathcal{U}(s)[i]\}$. Note that (i,j) is a directed edge where the direction is from i to j. Now we can present the steps of our algorithm for computing the maximal palindromic factorization of a given string s of length n. The steps are as follows.

MPF Algorithm: Maximal Palindromic Factorization Algorithm
Input: A String s of length n
Output: Maximal Palindromic Factorization of s

```
1: Compute the set of maximal palindromes $\mathcal{MP}(s)$ and
   identify the set of prefix palindromes $\mathcal{PP}(s)$.
2: Compute the list $\mathcal{MPL}(s)$.
3: Compute the list $\mathcal{U}(s)$.
4: Construct the graph $\mathcal{G}_s = (\mathcal{V}, \mathcal{E})$.
5: Do a breadth first search on $\mathcal{G}_s$ assuming the vertex $n$ as the source.
6: Identify the shortest path $P \equiv n \rightsquigarrow v$ such that $v$ is the end position of a
   palindrome belonging to $\mathcal{PP}(s)$. Suppose $P \equiv \langle n = p_k, p_{k-1}, \ldots, p_2, p_1 = v \rangle$.
7: Return $s = s[1..p_1]\ s[p_1+1..p_2]\ \cdots\ s[p_{k-1}+1..p_k]$.
```

4 Analysis

We now have the following theorem which proves the correctness of MPF Algorithm.

Theorem 5 (Correctness and Running time). *Given a string s of length n, MPF Algorithm correctly computes the maximal palindromic factorization of s in $O(n)$ time.*

Proof. **Correctness:**
We first focus on an edge $(i,j) \in \mathcal{E}$ of the graph $\mathcal{G}_s$ constructed at Step 4 of the algorithm. By definition, this means the following:

1. There is a maximal palindrome pal_i having length ℓ_i (say) ending at position i.
2. There is a maximal palindrome pal_j having length ℓ_j (say) ending at position j.
3. $i > j$.
4. $i - \ell_i = j$.

Since, by definition, each directed edge $(i, j) \in \mathcal{E}$ is such that $i > j$, so, for a path $P \equiv \langle p_k, p_{k-1}, \ldots, p_2, p_1 \rangle$ in $\mathcal{G}_s$, we always have $p_k > p_{k-1} > \cdots > p_1$. A path $P \equiv \langle p_k, p_{k-1}, \ldots, p_2, p_1 \rangle$ can be seen as corresponding to a substring of s formed by concatenation of maximal palindromes as follows. Each edge $(p_i, p_{i-1}) \in P$ corresponds to a palindromic substring $s[p_{i-1}]s[p_{i-1}+1]s[p_{i-1}+2]\cdots s[p_i]$.

Hence, following the definition of the edges, it is clear that any path would correspond to a substring of s formed by concatenation of consecutive palindromic substrings.

In Step 5, a breadth first tree is constructed from $\mathcal{G}_s$ considering the vertex n as the source. A breadth first tree gives the shortest path from the source (in this case, n) to any other node. Now, in Step 6, MPF Algorithm identifies the set of shortest paths (say, $SPath$) between n and j such that j corresponds to a maximal palindromic prefix of s. Now the maximum palindromic factorization must contain exactly one palindrome from $\mathcal{PP}(s)$ and exactly one palindrome from $\mathcal{SP}(s)$, where ℓ is minimum. Hence, it is easy to realize that the shortest one among the paths in $SPath$ corresponds to the maximal palindromic factorization. This completes the correctness proof.

Running time:
In Step 1 the computation of $\mathcal{MP}(s)$ can be done using the algorithm of [25] in $O(n)$ time. Also, $\mathcal{PP}(s)$ and $\mathcal{SP}(s)$ can be computed easily while computing $\mathcal{MP}(s)$. The computation of $\mathcal{MPL}(s)$ and $\mathcal{U}(s)$ in Step 2 and Step 3 can be done in linear time once $\mathcal{MP}(s)$ is computed.

Now construction of the graph $\mathcal{G}_s$ is done in Step 4. There are in total n number of vertices is $\mathcal{G}_s$. The number of edges $|\mathcal{E}|$ of $\mathcal{G}_s$ depends on $\mathcal{U}(s)$. But it is easy to realize that the summation of the number of elements in all the positions of $\mathcal{U}(s)$ cannot exceed the total number of maximal palindromes. Now, since there can be at most $2n + 1$ centers, there can be just as many maximal palindromes in s. Therefore we have $|\mathcal{E}| = O(n)$.

Hence, the graph construction (Step 4) as well as the breadth first search (Step 5) can be done in $O(|\mathcal{V}| + |\mathcal{E}|) = O(n)$ time. Finally, the identification of the desired path in Step 6 can also be done easily if we do some simple bookkeeping during the breadth first search because we already have computed the sets $\mathcal{PP}(s)$ and $\mathcal{SP}(s)$ in Step 1. Hence the total running time of the algorithm is $O(n)$. And this completes the proof. □

4.1 An Illustrative Example

Suppose we are given a string $s = abbcbbcbbbcbb$. We will proceed as follows:

First we compute the set $\mathcal{MP}(s)$. For example, at position $i = 9$ there are 2 palindromes of lengthes 2 and 9 centered at position 9 of s.

Secondly, we compute the set $\mathcal{MPL}(s)$. For example, at position $i = 9$ there are 3 palindromes of lengthes 2, 5 and 8 ending at position 9 of s.

Finally, we compute $\mathcal{U}(s)$ (Table 1 shows full steps for $s = abbcbbcbbbcbb$).

Now, we can construct the graph $\mathcal{G}_s$ easily as shown in Figure 1. For example, we can see that from vertex $i = 9$ we have 3 directed edges, namely, $(9, 7), (9, 4)$ and

$(9,1)$. Our desired shortest path is $P = \langle 13, 4, 3, 1\rangle$ (corresponding edges are shown as dashed edges). So, the maximal palindromic factorization of $s = abbcbbcbbbcbb$ is as follows:

$$s[1..1]s[2..3]s[4..4]s[5..13] = a\ bb\ c\ bbcbbbcbb\,.$$

i	$\mathcal{MP}[i]$	$\mathcal{MPL}[i]$	$\mathcal{U}[i]$
1	$\mathcal{MP}[1] = \{(1,1)\}$	$\mathcal{MPL}[1] = \{1\}$	$\mathcal{U}[1] = \{0\}$
2	$\mathcal{MP}[2] = \{(2,2)\}$	$\mathcal{MPL}[2] = \{.\}$	$\mathcal{U}[2] = \{.\}$
3	$\mathcal{MP}[3] = \{(3,1)\}$	$\mathcal{MPL}[3] = \{2\}$	$\mathcal{U}[3] = \{1\}$
4	$\mathcal{MP}[4] = \{(4,5)\}$	$\mathcal{MPL}[4] = \{.\}$	$\mathcal{U}[4] = \{.\}$
5	$\mathcal{MP}[5] = \{(5,8)\}$	$\mathcal{MPL}[5] = \{.\}$	$\mathcal{U}[5] = \{.\}$
6	$\mathcal{MP}[6] = \{(6,1)\}$	$\mathcal{MPL}[6] = \{5\}$	$\mathcal{U}[6] = \{1\}$
7	$\mathcal{MP}[7] = \{(7,5)\}$	$\mathcal{MPL}[7] = \{.\}$	$\mathcal{U}[7] = \{.\}$
8	$\mathcal{MP}[8] = \{(8,2)\}$	$\mathcal{MPL}[8] = \{.\}$	$\mathcal{U}[8] = \{.\}$
9	$\mathcal{MP}[9] = \{(9,2)(9,9)\}$	$\mathcal{MPL}[9] = \{2,5,8\}$	$\mathcal{U}[9] = \{7,4,1\}$
10	$\mathcal{MP}[10] = \{(10,1)\}$	$\mathcal{MPL}[10] = \{2\}$	$\mathcal{U}[10] = \{8\}$
11	$\mathcal{MP}[11] = \{(11,5)\}$	$\mathcal{MPL}[11] = \{.\}$	$\mathcal{U}[11] = \{.\}$
12	$\mathcal{MP}[12] = \{(12,2)\}$	$\mathcal{MPL}[12] = \{.\}$	$\mathcal{U}[12] = \{.\}$
13	$\mathcal{MP}[13] = \{(13,1)\}$	$\mathcal{MPL}[13] = \{1,2,5,9\}$	$\mathcal{U}[13] = \{12,11,8,4)\}$

Table 1. Steps for computing $\mathcal{U}(s)$ and $\mathcal{MPL}(s)$ for $s = abbcbbcbbbcbb$

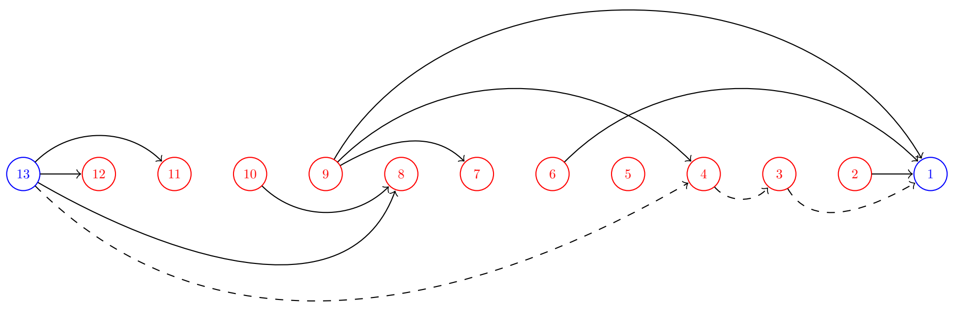


Figure 1. The graph $\mathcal{G}_s$ for $s = abbcbbcbbbcbb$

5 Conclusion

In this paper, we answer a recent question raised during StringMasters, Verona, Italy - 2013: does there exist an algorithm to compute the maximal palindromic factorization of a finite string? Namely, given a finite string, find the smallest set (minimum number of palindromic factors), such that the string is covered by that set of factors with no overlaps. We answer the previous question affirmatively by providing a linear-time algorithm that computes the *maximal palindromic factorization (MPF)* of a string (the algorithms is evaluated with respect to the length of the given string).

An immediate target will be extending the algorithm presented in 3 to biological palindromes, where the word reversal is defined in conjunction with the complementarity of nucleotide letters: $c \leftrightarrow g$ and $a \leftrightarrow t$ (or $a \leftrightarrow u$, in case of RNA). The proposed algorithm can be extended to find *maximal distinct palindromic factorization set*. We will focus on this problem in a future work. Also we will work on studying palindromic cover of string and how can it be modeled using graphs.

Acknowledgements

The authors are grateful to the anonymous reviewers for their helpful comments. This research was carried out when Rahman was visiting King's College London as a Commonwealth Fellow.

References

1. D. Breslauer and Z. Galil: *Finding all periods and initial palindromes of a string in parallel.* Algorithmica, 14 October 1995, pp. 355–366.
2. D. Breslauer, R. Grossi, and F. Mignosi: *Simple real-time constant-space string matching*, in CPM, R. Giancarlo and G. Manzini, eds., vol. 6661 of Lecture Notes in Computer Science, Springer, 2011, pp. 173–183.
3. M. Burrows and D. Wheeler: *A block-sorting lossless data compression algorithm*, tech. rep., Digital SRC Research Report 124, 1994.
4. K.-Y. Chen, P.-H. Hsu, and K.-M. Chao: *Identifying approximate palindromes in run-length encoded strings*, in Proceedings of 21st International Symposium, ISAAC 2010, Jeju, Korea, December 15-17, 2010, 2010, pp. 339–350.
5. C. Choi: *DNA palindromes found in cancer.* Genome Biology, 6 2005.
6. S. R. Chowdhury, M. M. Hasan, S. Iqbal, and M. S. Rahman: *Computing a longest common palindromic subsequence*, to appear in Fundamenta Informaticae.
7. M. Crochemore, J. Désarménien, and D. Perrin: *A note on the Burrows-Wheeler Transformation.* CoRR, abs/cs/0502073 2005.
8. M. Crochemore, L. Ilie, and W. F. Smyth: *A simple algorithm for computing the Lempel Ziv factorization*, in DCC, IEEE Computer Society, 2008, pp. 482–488.
9. M. Crochemore and D. Perrin: *Two-way string matching.* J. ACM, 38(3) 1991, pp. 651–675.
10. X. Droubay: *Palindromes in the Fibonacci word.* Inf. Process. Lett., 55(4) 1995, pp. 217–221.
11. X. Droubay and G. Pirillo: *Palindromes and Sturmian words.* Theoretical Computer Science, 223 1999, pp. 73–85.
12. Z. Galil: *Real-time algorithms for string-matching and palindrome recognition*, in Proceedings of the eighth annual ACM symposium on Theory of computing, ACM, 1976, pp. 161–173.
13. J. Y. Gil and D. A. Scott: *A bijective string sorting transform.* CoRR, abs/1201.3077 2012.
14. A. Glen: *Occurrences of palindromes in characteristic Sturmian words.* Theor. Comput. Sci., 352(1–3) 2006, pp. 31–46.
15. D. Gusfield: *Algorithms on strings, trees, and sequences: computer science and computational biology*, Cambridge University Press, New York, NY, USA, 1997.

16. D. Gusfield: *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, New York, 1997.
17. P.-H. Hsu, K.-Y. Chen, and K.-M. Chao: *Finding all approximate gapped palindromes*, in Proceedings of 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009., 2009, pp. 1084–1093.
18. T. I, I. Shunsuke, and T. Masayuki: *Palindrome pattern matching*, in Proceedings of 22nd Annual Symposium, CPM 2011, Palermo, Italy, June 27-29, 2011., 2011, pp. 232–245.
19. D. E. Knuth, J. H. M. Jr., and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal of Computing, 6(2) 1977, pp. 323–350.
20. R. Kolpakov and G. Kucherov: *Searching for gapped palindromes.*
21. R. Kolpakov and G. Kucherov: *On maximal repetitions in words.* J. Discrete Algorithms, 1 1999, pp. 159–186.
22. R. Kolpakov and G. Kucherov: *Searching for gapped palindromes.* Theoretical Computer Science, November 2009, pp. 5365–5373.
23. E. Lamprea-Burgunder, P. Ludin, and P. Mser: *Species-specific typing of dna based on palindrome frequency patterns.* DNA Research, 18 2011, pp. 117–124.
24. J. Lange, H. Skaletsky, S. K. M. van Daalen, S. L. Embry, C. M. Korver, L. G. Brown, R. D. Oates, S. Silber, S. Repping, and D. C. Page: *Isodicentric y chromosomes and sex disorders as byproducts of homologous recombination that maintains palindromes.* Cell, 138 September 2009, pp. 855–869.
25. G. Manacher: *A new linear-time on-line algorithm for finding the smallest initial palindrome of a string.* Journal of the ACM, 22 July 1975, pp. 346–351.
26. T. Martínek and M. Lexa: *Hardware acceleration of approximate palindromes searching*, in Proceedings of The International Conference on Field-Programmable Technology, 2008, pp. 65–72.
27. W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto: *Efficient algorithms to compute compressed longest common substrings and compressed palindromes.* Theoretical Computer Science, 410(8–10) March 2009, pp. 900–913.
28. G. Melançon: *Lyndon factorization of infinite words.*, in STACS, C. Puech and R. Reischuk, eds., vol. 1046 of Lecture Notes in Computer Science, Springer, 1996, pp. 147–154.
29. A. H. L. Porto and V. C. Barbosa: *Finding approximate palindromes in strings.* Pattern Recognition, 2002.
30. *StringMasters, Verona, Italy*, 2013.
31. H. Tanaka, D. A. Bergstrom, M.-C. Yao, and S. J. Tapscott: *Large dna palindromes as a common form of structural chromosome aberrations in human cancers.* Human Cell, 19(1) 2006, pp. 17–23.
32. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23(3) 1977, pp. 337–343.

Towards a Very Fast Multiple String Matching Algorithm for Short Patterns

Simone Faro[1] and M. Oğuzhan Külekci[2]

[1] Dipartimento di Matematica e Informatica, Università di Catania, Italy
[2] TÜBİTAK National Research Institute of Electronics and Cryptology, Turkey
faro@dmi.unict.it, oguzhan.kulekci@tubitak.gov.tr

Abstract. *Multiple exact string matching* is one of the fundamental problems in computer science and finds applications in many other fields, among which computational biology and intrusion detection. It turns out that short patterns appear in many instances of such problems and, in most cases, sensibly affect the performances of the algorithms. Recent solutions in the field of string matching try to exploit the power of the word RAM model to speed-up the performances of classical algorithms. In this model an algorithm operates on words of length w, grouping blocks of characters, and arithmetic and logic operations on the words take one unit of time. This study presents a first preliminary attempt to develop a filter based exact multiple string matching algorithm for searching set of short patterns by taking benefit from Intel's SSE (streaming SIMD extensions) technology. Our experimental results on small, medium, and large alphabet text files show that the proposed algorithm is competitive in the case of short patterns against other efficient solutions, which are known to be among the fastest in practice.

Keywords: multiple string matching, experimental algorithms, text-processing, short patterns, Streaming SIMD Extensions Technology, SSE

1 Introduction

In this article we consider the *multiple string matching problem* which is the problem of searching for all exact occurrences of a set of r patterns in a text t, of length n, where the text and patterns are sequences over a finite alphabet Σ.

Multiple string matching is an important problem in many application areas of computer science. For instance, in computational biology, with the availability of large amounts of DNA data, matching of nucleotide sequences has become an important application and there is an increasing demand for fast computer methods for analysis and data retrieval, e.g., in metagenomics [16,15], we have a set of short patterns which are the extracted DNA fragments of some species, and we would like to check if they exist in another living organism. Although there are various kinds of comparison tools that provide aligning and approximate matching, most of them are based on exact matching in order to speed up the process.

Another important usage of multiple pattern matching algorithms appears in network intrusion detection systems such as Snort [29] as well as in anti-virus software. Snort is a light-weight open-source NIDS which can filter packets based on predefined rules. If the packet matches a certain header rule then its payload is scanned against a set of predefined patterns associated with the header rule. The number of patterns can be in the order of a few thousands[1]. In all these applications, the speed at which pattern matching is performed critically affects the system throughput and although

[1] Snort version 2.9 contains over 2000 strings

only a small portion of such rules contains short patterns, it turns out that they sensibly affect the performance of multiple string matching algorithm [31]. Moreover another major performance bottleneck of the regarding solutions to these problems is to achieve high-speed multiple pattern matching required to detect malicious patterns of ever growing sets.

This paper presents the results of a first preliminary attempt to develop a fast and practical algorithm for the multiple exact string matching problem which focuses on sets of short patterns. The algorithm we propose, named Multiple Exact Packed String Matching algorithm (MEPSM for short), is designed using specialized word-size packed string matching instructions based on the Intel streaming SIMD extensions (SSE) technology. It can be seen as an extension of the MSSEF algorithm [20,10] that was designed for searching long patterns and has been evaluated amongst the fastest algorithms when the length of the pattern is greater than 32 characters. Thus in the present note we concentrate on solutions which could be used for searching sets of patterns shorter than 32 characters.

This work presents a preliminary result, meaning that our algorithm is still a *work in progress*. Specifically it obtains competitive results only for patterns with a length between 16 and 32, while much work has to be done for obtaining a fast solution for sets of patterns shorter than 16 characters. This will be the goal of our future work.

In Section 2, we introduce some notations and the terminology we adopt throughout the paper. We survey the most relevant existing algorithms for the multiple string matching problem in Section 3. We then present a new algorithm for the multiple string matching problem in Section 4 and report experimental results under various conditions in Section 5. Conclusions and perspectives are given in Section 6.

2 Notions and Terminology

Throughout the paper we will make use of the following notations and terminology. A string p of length $\ell > 0$ is represented as a finite array $p[0 \dots \ell - 1]$ of characters from a finite alphabet Σ of size σ. Thus $p[i]$ will denote the $(i+1)$-st character of p, and $p[i \dots j]$ will denote the *factor* (or *substring*) of p contained between the $(i+1)$-st and the $(j+1)$-st characters of p, for $0 \leq i \leq j < \ell$.

Given a set of r patterns $\mathcal{P} = \{p_0, p_1, \dots, p_{r-1}\}$, we indicate with symbol m_i the length of the pattern p_i, for $0 \leq i < r$, while the length of the shortest pattern in $\mathcal{P}$ is denoted by m', i.e. $m' = \min\{m_i \mid 0 \leq i < r\}$. The length of $\mathcal{P}$, which consists of the sum of the lengths of the $p_i s$ is denoted by m, i.e. $m = \sum_{i=0}^{r-1} m_i$.

We indicate with symbol w the number of bits in a computer word and with symbol $\gamma = \lceil \log \sigma \rceil$ the number of bits used for encoding a single character of the alphabet Σ. The number of characters of the alphabet that fit in a single word is denoted by $\alpha = \lfloor w/\gamma \rfloor$. Without loss of generality we will assume throughout the paper that γ divides w.

In chunks of α characters, any string p of length ℓ is represented by an array of blocks $P[0 \dots k-1]$ of length $k = \lceil \ell/\alpha \rceil$. Each block $P[i]$ consists of α characters of p and in particular $P[i] = p[i\alpha \dots i\alpha + \alpha - 1]$, for $0 \leq i < k$. The last block of the string $P[k-1]$ is not complete if $(\ell \mod \alpha) \neq 0$. In that case we suppose the rightmost remaining characters of the block are set to zero. Given a set of patterns $\mathcal{P}$, we define $L = \lceil m'/\alpha \rceil - 1$ as the zero-based address of the last α-character block of the shortest pattern in $\mathcal{P}$, whose individual characters are totally composed of the characters of the pattern without any padding. Actually, if the length of the shortest pattern in $\mathcal{P}$ is a multiple of α, there is no padding in the last α-characters block, and thus, $L = \lceil m'/\alpha \rceil - 1$. In the other cases, L is the index of the block preceding the last one, as the last one is not a complete block, making $L = \lceil m'/\alpha \rceil - 2$.

Although different values of α and γ are possible, in most cases we assume that $\alpha = 16$ and $\gamma = 8$, which is the most common setting while working with characters in ASCII code and in a word RAM model with 128-bit registers, available in almost all recent processors supporting single instruction multiple data (SIMD) operations.

3 Previous Results

Let $\mathcal{P} = \{p_0, p_1, \ldots, p_{r-1}\}$ be a set of r patterns, where pattern p_i has length m_i, for $0 \leq i < r$, and let t be a text of length n. Moreover let $m = \sum_{i=0}^{r-1} m_i$ and let $m' = \min\{m_i \mid 0 \leq i < m\}$ be the length of the shortest pattern of $\mathcal{P}$.

A first trivial solution to the multiple string matching problem consists of applying an exact string matching algorithm for locating each pattern in $\mathcal{P}$. If we use the well–known Knuth-Morris-Pratt algorithm (KMP) [18], whose search phase is linear in the dimension of the text, this solution has an $O(m + rn)$ worst case time complexity. However, in many practical cases it is possible to avoid reading all the characters of the text achieving sub-linear performances on average.

In a computational model, where the matching algorithm is restricted to read all the characters of the text one by one, the optimal complexity of the multiple pattern matching problem is $\mathcal{O}(m + n)$ while the optimal average complexity of the problem is $O(n \log_\sigma(rm')/m')$ [23]. Such complexities were achieved the first time by the well–known Aho-Corasick algorithm [1] and by the Set-Backward-DAWG-Matching (SBDM) algorithm [26,8], respectively. The SBNDM algorithm is based on the suffix automaton that builds an exact indexing structure for the reverse strings of $\mathcal{P}$ such as a factor automaton or a generalized suffix tree. However experimental investigations highlighted that the bottleneck of the SBDM algorithm is the construction time and space consumption of the exact indexing structure. This can be partially avoided by replacing the exact indexing structure by a factor oracle for a set of strings, which is performed in the Set Backward Oracle Matching (SBOM) algorithm [2].

Hashing is an extensively used approach in string matching [17] and also provides a simple and efficient method to design an efficient algorithm for multiple pattern matching with a sub-linear average complexity. It has been used first by Wu and Manber [34] (WM) whose algorithm constructs an index table for blocks of q characters. Their method is incorporated in the *agrep* command [32].

Recently Faro and Lecroq [13] presented an improvement of WM algorithm based on hashing and q-grams which provides good performances in practical cases. Their method is based on the combination of multiple hash functions with the aim of improving the filtering phase, i.e. to reduce the number of candidate occurrences found by the algorithm. They conduct an experimental evaluation to show the efficiency of the method for matching DNA sequences.

In the last two decades a lot of work has been made in order to exploit the power of the word RAM model of computation to speed-up string matching algorithms for a single pattern. In this model, the computer operates on words of length w, thus blocks of characters are read and processed at once. This means that usual arithmetic and logic operations on the words all take one unit of time. Most of the solutions which exploit the word RAM model are based on the *bit-parallelism* technique or on the *packed string matching* technique.

The bit-parallelism technique [3] takes advantage of the intrinsic parallelism of the bit operations inside a computer word, allowing to cut down the number of operations that an algorithm performs by a factor up to w. Bit-parallelism is particularly suitable for the efficient simulation of nondeterministic automata [7]. The Shift-Or [3] and BNDM [24] algorithms, which are the representatives of this genre, can be easily extended to the multiple patterns case by deriving the corresponding automata from the maximal trie of the set of patterns [33,25]. The resulting algorithms have a $\mathcal{O}(\sigma\lceil m/w\rceil)$-space complexity and work in $\mathcal{O}(n\lceil m/w\rceil)$ and $\mathcal{O}(n\lceil m/w\rceil m')$ worst-case searching time complexity, respectively. Another efficient solution is the MBNDM algorithm [28], which computes a superimposed pattern from the patterns of the input set when using a condensed alphabet of q characters, and performs filtering using the approach of the standard BNDM algorithm.

However, the bit-parallel encoding requires one bit per automaton state, for a total of (at most) $\lceil m/w\rceil$ computer words. Thus, as long as all the automaton states fit in a computer word, bit-parallel algorithms are extremely fast, otherwise their performances degrade as the number of states of the automaton grows. Although there have been efforts to overcome word-size limitation [19,21,5,6], their performances are still not satisfactory to meet the expectation in practice.

In the *packed string matching* technique multiple characters are packed into one larger word, so that the characters can be compared in bulk rather than individually. In this context, if the characters of a string are drawn from an alphabet of size σ, then $\lfloor w/\log\sigma\rfloor$ different characters fit in a single word, using $\lfloor\log\sigma\rfloor$ bits per character. The packing factor is $\alpha = w/\log\sigma$.

The recent study of Ben-Kiki *et al.* [4] reached the optimal $\mathcal{O}(n/\alpha + occ)$-time complexity for single string matching in $\mathcal{O}(1)$ extra space, where occ is the number of occurrences of the searched pattern. From a practical point of view a very recent algorithm by the authors [10], named Exact Packed String Matching algorithm (EPSM) turns out to be the fastest solution in the case of short patterns. When the length of the searched pattern increases, the SSEF [20] algorithm that performs filtering via the SIMD instructions becomes the best solution in many cases [11,14,12].

In the field of multiple pattern matching in [9] the authors introduced a filter based algorithm, named MSSEF, designed for long patterns, and which benefits from computers intrinsic SIMD instructions. The best and worst case time complexities of the algorithm are $\mathcal{O}(n/m)$ and $\mathcal{O}(nm)$, respectively. The gain obtained in speed via MSSEF becomes much more significant with the increasing set sizes. Hence, considering the fact that the number of malicious patterns in intrusion detection systems or anti-virus software is ever growing as well as the reads produced by next-generation sequencing platforms, the proposed algorithm is supposed to serve a good basis for massive multiple long pattern search applications on these areas.

To the best of our knowledge, packed string matching has not been explored before for multiple pattern matching, and MSSEF is the initial study of this genre.

4 A New Multiple Pattern Matching Algorithm

In this section we present a new multiple string matching algorithm for short patterns, named Multiple Exact Packed String Matching algorithm (MEPSM), and which extends the MSSEF multiple pattern matching algorithm designed for long patterns.

Along the same line of the MSSEF algorithm the MEPSM algorithm is based on a filter mechanism. It first searches the text for candidate occurrences of the patterns using a collection of fingerprint values computed in a preprocessing phase from the set of patterns $\mathcal{P}$. Then the text is scanned by extracting fingerprint values at fixed intervals and in case of a matching fingerprint at a specific position, a naive check follows at that position for all patterns which resemble the detected fingerprint value.

MEPSM is designed to be effective on sets of short patterns, where the the upper limit for the length of the shortest pattern of the set is 32 ($m' \leq 32$). The MEPSM algorithm runs in $\mathcal{O}(nm)$ worst case time complexity and use $\mathcal{O}(rm' + 2^{\alpha})$ additional space, where we remember that m' is the length of the shortest pattern in $\mathcal{P}$.

In what follows, we first describe in Section 4.1 the computational model we use for the description of our solutions. Then we describe the preprocessing phase and the searching phase of the MEPSM algorithm in Section 4.2 and in Section 4.3, respectively. We conduct a brief complexity analysis of the algorithm in Section 4.4.

4.1 The Model

In the design of our algorithm we use specialized word-size packed string matching instructions, based on the Intel streaming SIMD extensions (SSE) technology. SIMD instructions exist in many recent microprocessors supporting parallel execution of some operations on multiple data simultaneously via a set of special instructions working on limited number of special registers. Although the usage of SIMD has been explored deeply in multimedia processing, implementation of encryption/decryption algorithms, and on some scientific calculations, only in recent years it has been addressed in string matching [20,9,10].

In our model of computation we suppose that w is the number of bits in a word and σ is the size of the alphabet. When the pattern is short we process the text in chunks of ρ characters, where $\rho \leq \alpha$.

In most practical applications we have $\sigma = 256$ (ASCII code). Moreover SSE specialized instructions allow to work on 128-bit registers, thus reading and processing blocks of sixteen 8-bit characters in a single time unit (thus $\alpha = 16$). Our algorithms are allowed to scan the text in block of 4, 8 and 16 characters.

The specialized word-size packed instruction which is used by our algorithm is named pcrcf (*packed cyclic redundancy check fingerprint*).

A cyclic redundancy check (CRC) is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data. It was first proposed by W. Wesley Peterson during 1961 [27]. A CRC device calculates a short, fixed-length binary sequence, called *check value*, for each block of data to be sent or stored and appends it to the data. The check value is based on the remainder of a polynomial division of their contents.

Thus the check value of a block of data can be seen as a fingerprint of the block and can be used to evaluate the resemblance of two blocks.

Specifically the instruction $\mathsf{pcrcf}(B, k)$, computes an α-bit fingerprint r from a k-bit register B. In practical cases r is a 16-bit register, while the value of k could be 16, 32 or 64, depending on the length of the pattern.

The $\mathsf{pcrcf}(B, k)$ specialized instruction can be emulated in constant time by using the following sequence of specialized SIMD instructions

(i) $crc_{32} \leftarrow$ _mm_crc32_u16(ac, B) if B is a 16 bit register
$r \leftarrow$ (unsigned short int) crc_{32}

(ii) $crc_{32} \leftarrow$ _mm_crc32_u32(ac, B) if B is a 32 bit register
$r \leftarrow$ (unsigned short int) crc_{32}

(iii) $crc_{32} \leftarrow$ _mm_crc32_u64(ac, B) if B is a 64 bit register
$r \leftarrow$ (unsigned short int) crc_{32}

Specifically these instructions compute a 32-bit register crc_{32} which is the cyclic redundancy check of the k-bit register B. The parameter ac is a CRC additive constant. The instruction starts with the initial value in ac, accumulates a cyclic redundancy check value for B and stores the result in crc_{32}. Then a second instruction is applied in order to downsample the crc_{32} register and get the 16-bit signature of B. In our implementation we simply take the lower 16 bits of crc_{32} by casting it to an unsigned short int.

In the Intel Core i7 processors, the instructions shown above are implemented with a latency of three cycles and a throughput of one cycle.

We are now ready to describe the new multiple string matching algorithm.

4.2 The Preprocessing Phase

Given a set of patterns $\mathcal{P} = \{p_0, p_1, \ldots, p_{r-1}\}$, where pattern p_i has length m_i, let $m' = \min\{m_i \mid 0 \leq i < r\}$ denote the length of the shortest pattern in $\mathcal{P}$, and $L = \lceil m'/\rho \rceil - 1$. The preprocessing phase of the MEPSM algorithm, which is depicted in Figure 1 (on the left), consists in compiling all the possible fingerprint values of the patterns in the input set $\mathcal{P}$ according to all possible alignments with a block of ρ characters. In particular we set

$$\rho = \min\{i \mid 2^{i+1} > m\},$$

getting $\rho = 16$ when $16 \leq m < 32$, $\rho = 8$ when $8 \leq m < 16$ and $\rho = 4$ when $4 \leq m < 8$.

Thus a fingerprint value is computed for each block $p_i[j \ldots j+\rho-1]$, for $0 \leq i < r$ and $0 \leq j \leq \rho L$. The corresponding fingerprint of a block B of α characters is the α bits register returned by the instruction $\mathsf{pcrcf}(B, k)$ (where $k = \rho \times 8$).

To this purpose a table F of size 2^α is computed in order to store, for any possible fingerprint value v, the set of pairs (i, j) such that $\mathsf{pcrcf}(p_i[j \ldots j + \rho - 1], k) = v$. More formally we have, for $0 \leq v < 2^\alpha$

$$F[v] = \Big\{(i, j) \mid 0 \leq i < r, 0 \leq j \leq \alpha L \text{ and } \mathsf{wsfp}(p_i[j \ldots j + \rho - 1], k) = v\Big\}.$$

4.3 The Searching Phase

Let t be a text of length n and let $T[0 \ldots N]$ be the text t represented in blocks of ρ characters, where $N = \lceil n/\rho \rceil - 1$. Moreover let $L = \lceil m'/\rho \rceil - 1$.

The basic idea of the searching phase is to compute a fingerprint value for each block of the text $T[zL]$, where $0 \leq z < \lfloor N/L \rfloor$, to explore if it is appropriate to observe any pattern in $\mathcal{P}$ involving an alignment with the block $T[zL]$. If the fingerprint value

```
Preprocessing($\mathcal{P}, r, m', \rho$)
1.  $L \leftarrow \lceil m'/\rho \rceil - 1$
2.  for $v \leftarrow 0$ to $2^\alpha - 1$ do $F[v] \leftarrow \emptyset$
3.  for $i \leftarrow 0$ to $r - 1$ do
4.      for $j \leftarrow 0$ to $\rho L$ do
5.          $a \leftarrow p_i[j \ldots j + \rho - 1]$
6.          $v \leftarrow$ pcrcf($a, \rho \times 8$)
7.          $F[v] \leftarrow F[v] \cup \{(i, j)\}$
8.  return $F$

MEPSM($\mathcal{P}, r, t, n, \rho$)
1.  $m' \leftarrow \min\{m_i \mid 0 \leq i < r\}$
2.  $F \leftarrow$Preprocessing($\mathcal{P}, r, m', \rho$)
3.  $N \leftarrow \lceil n/\rho \rceil - 1$; $L \leftarrow \lceil m'/\rho \rceil - 1$
4.  for $s = 0$ to $N$ step $L$ do
5.      $v \leftarrow$ pcrcf($T[s], \rho \times 8$)
6.      for each $(i, j) \in F[v]$ do
7.          if $p_i = t[s\rho - j \ldots s\rho - j + m_i - 1]$ then
8.              output $(s\rho - j, i)$
```

Figure 1. The pseudo-code of the MSSEF multiple string matching algorithm.

indicates that some of the alignments are possible, then those fitting are naively checked.

The pseudo-code given in Figure 1 (on the right) depicts the skeleton of the MEPSM algorithm. The main loop investigates the blocks of the text T in steps of L blocks. If the fingerprint v computed on $T[s]$ is not empty, then the appropriate positions listed in $F[v]$ are verified accordingly.

In particular $F[v]$ contains a linked list of pairs (i, j) marking the pattern p_i and the beginning position of the pattern in the text. While investigating occurrences on $T[s]$, if $F[v]$ contains the couple (i, j), this indicates the pattern p_i may potentially begin at position $(s\rho - j)$ of the text. In that case, a complete verification is to be performed between p and $t[s\rho{-}j \ldots s\rho{-}j{+}m_i{-}1]$ via a symbol-by-symbol inspection.

4.4 Complexity Analysis

In this Section we give a brief time and space analysis of the MEPSM algorithm.

The preprocessing phase of the MSSEF algorithm requires some additional space to store the rm' possible alignments in the 2^α locations of the table F. Thus the space requirements of the algorithm is $\mathcal{O}(rm' + 2^\alpha)$. Assume $L = \lceil m'/\rho \rceil - 1$. The first loop of the preprocessing phase just initializes the table F, while the second for loop is run $L\alpha$ times. Thus, time complexity of preprocessing is $\mathcal{O}(L\rho)$ that approximates to $O(m)$.

Assume now $N = \lceil n/\rho \rceil - 1$. The searching phase of the algorithm investigates the N blocks of the text T in steps of L blocks. The total number of filtering operations is exactly N/L. At each attempt, the maximum number of verification requests is ρL, since the filter gives information about that number of appropriate alignments of the patterns.

On the other hand, if the computed fingerprint points to an empty location in F, then there is obviously no need for verification. The verification cost for a pattern $p_i \in \mathcal{P}$ is assumed to be $\mathcal{O}(m_i)$, with the brute-force checking of the pattern. Hence, in the worst case the time complexity of the verification is $\mathcal{O}(L\rho m)$, which happens when all patterns in $\mathcal{P}$ must be verified at any possible beginning position.

From these facts, the best case complexity is $\mathcal{O}(N/L)$, and worst case complexity is $\mathcal{O}((N/L)(L\rho m))$, which approximately converge to $\mathcal{O}(n/m')$ and $\mathcal{O}(nm)$ respectively.

5 Experimental results

In this section we present experimental results in order to evaluate the performances of the newly presented algorithm and to compare it against the best algorithms known in literature for multiple string matching problem.

In particular we compared the performances of the MEPSM algorithm against the fastest algorithms known in literature, and specifically:

– MBNDM(q): the Multiple Backward DAWG Matching algorithm [30,28], with values of q ranging from 3 to 8;
– WM(q, h): the Wu-Manber algorithm [34] with, values of q ranging from 3 to 8 and values of h ranging from 1 to 3.

However, in our experimental results only the best versions of the MBNDM(q) and WM(q, h) algorithms are reported, indicating the corresponding values of q and h.

All algorithms have been implemented in the C programming language and have been compiled with the GNU C Compiler, using the optimization options -O3. The experiments were executed locally on an MacBook Pro with 4 Cores, a 2 GHz Intel Core i7 processor, 4 GB RAM 1333 MHz DDR3, 256 KB of L2 Cache and 6 MB of Cache L3. Algorithms have been compared in terms of running times, including any preprocessing time, measured with a hardware cycle counter, available on modern CPUs.

For the evaluation, we use a genome sequence, a protein sequence and a natural language text (English language), all sequences of 4MB. The sequences are provided by the SMART research tool[2] and are available online for download. We have generated sets of 10, 100, 1.000 and 10.000 patterns of fixed length ℓ for the tests. In all cases the patterns were randomly extracted from the text.

For each case we reported the mean over the running times of 200 runs. Tables 1, 2, and 3 lists the timings achieved on genome, protein, and english texts, respectively. Running times are expressed in thousands of seconds. We report the mean of the overall running times and (just below) the mean of the preprocessing time. Best times have been boldfaced.

Moreover it is important to notice that, in our experimental results, the value ℓ was made ranging over the values 16 to 32, which is the range where good results have been obtained by the MEPSM algorithm.

When searching sets of shorter patterns, with a length $m < 16$, our CRC filter technique did not obtain competitive performance underlining that additional work must be done in order to achieve better results on very short patterns.

Table 1 shows experimental results obtained by searching a genome sequence. It turns out that in all cases the MEPSM algorithms obtain the best results against previous solutions. It is up to 3 times faster than the second best result. The speed up becomes more sensible when the size of the set of patterns increases, while it slightly decreases when the length of the patterns increases.

The results reported in Table 1 highlight that the CRC filter technique is particularly efficient for DNA data, improving the performances of the MEPSM algorithm.

In Table 2 results obtained by searching a protein sequence are reported. In this case the MEPSM algorithm obtains always better results only when searching for set of 100 and 1.000 patterns.

[2] The SMART tool is available online at `http://www.dmi.unict.it/~faro/smart/`

(A) m	16	18	20	22	24	26	28	30	32
WM(5,1)	5.64	5.22	4.94	4.70	4.54	4.40	4.31	4.18	4.10
	0.43	0.42	0.44	0.42	0.43	0.43	0.44	0.43	0.44
MBNDM(5)	4.42	4.44	4.44	4.45	4.44	4.42	4.45	4.44	4.45
	0.17	0.17	0.17	0.17	0.17	0.17	0.17	0.17	0.17
MEPSM	**3.89**	**3.90**	**3.89**	**3.88**	**3.13**	**3.12**	**3.13**	**3.14**	**2.78**
	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01

(B) m	16	18	20	22	24	26	28	30	32
WM(5,1)	9.78	9.38	8.96	8.73	8.60	8.42	8.22	8.10	8.07
	0.44	0.44	0.44	0.44	0.44	0.44	0.44	0.43	0.44
WM(8,1)	9.98	8.96	8.18	7.62	7.21	6.88	6.55	6.32	6.18
	0.44	0.44	0.44	0.44	0.44	0.45	0.44	0.44	0.45
MBNDM(5)	9.04	9.02	9.03	9.03	9.05	9.02	9.05	9.05	9.01
	0.21	0.21	0.22	0.21	0.21	0.21	0.21	0.21	0.21
MEPSM	**4.27**	**4.26**	**4.24**	**4.23**	**3.54**	**3.54**	**3.54**	**3.54**	**3.24**
	0.04	0.04	0.04	0.04	0.08	0.08	0.08	0.08	0.12

(C) m	16	18	20	22	24	26	28	30	32
WM(8,1)	41.44	38.26	37.24	36.00	35.08	34.24	32.79	32.42	32.09
	0.52	0.52	0.54	0.56	0.57	0.59	0.58	0.60	0.60
WM(8,2)	41.55	32.83	28.83	27.56	25.55	24.93	23.02	22.52	22.05
	0.64	0.69	0.71	0.73	0.73	0.77	0.75	0.78	0.80
MBNDM(8)	25.22	25.23	25.28	25.09	25.36	25.05	25.14	25.28	25.33
	0.39	0.39	0.39	0.39	0.39	0.40	0.39	0.40	0.40
MEPSM	**8.08**	**8.09**	**8.05**	**7.87**	**7.96**	**7.92**	**7.89**	**7.96**	**8.53**
	0.40	0.40	0.40	0.40	0.77	0.77	0.77	0.78	1.15

(D) m	16	18	20	22	24	26	28	30	32
WM(5,2)	119.29	119.49	119.68	122.00	120.50	120.28	120.53	120.82	120.94
	1.72	1.86	2.00	2.18	2.32	2.43	2.57	2.74	2.85
WM(8,2)	135.98	126.30	124.21	125.15	123.64	123.50	123.78	123.62	123.99
	1.58	1.77	2.00	2.24	2.43	2.61	2.82	3.04	3.22
MBNDM(8)	377.14	386.98	389.70	393.60	393.82	397.11	415.40	421.28	420.84
	1.34	1.37	1.37	1.38	1.39	1.40	1.45	1.46	1.46
MEPSM	**50.97**	**51.55**	**47.62**	**47.60**	**51.52**	**51.90**	**55.93**	**54.60**	**64.85**
	3.87	3.92	3.97	4.00	7.65	7.66	8.21	8.02	11.78

Table 1. Experimental results on a genome sequence of 4 MB. Running times obtained while searching for sets of (A) 10 patterns, (B) 100 patterns, (C) 1.000 patterns and (D) 10.000 patterns.

When the set of patterns is small (10 patterns) the MEPSM algorithm is outperformed by the MBNDM algorithm for short patterns. However it obtains the best results for patterns with a length greater than 22. Again the MBNDM algorithm is the best choice when the set of patterns increases to 10.000 elements. In this last case the performances of the algorithm decreases sensibly when the length of the pattern increases. This behavior is also due to the increase of the preprocessing time consumed by the algorithm.

It turns out from experimental data shown in Table 2 that protein sequences are much more difficult to be filtered by the CRC filter technique proposed in this paper.

Finally in Table 3 experimental results are reported showing the running times obtained by searching on a natural language text. When searching this type of data the MEPSM algorithm turns out to be the best solution in almost all cases. It is second

(A) m	16	18	20	22	24	26	28	30	32
WM(3,1)	5.10	4.80	4.58	4.39	4.24	4.12	4.02	3.93	3.87
	0.44	0.43	0.44	0.43	0.43	0.44	0.43	0.43	0.43
WM(6,1)	5.53	5.10	4.75	4.51	4.33	4.18	4.07	3.93	3.84
	0.43	0.43	0.44	0.43	0.43	0.43	0.43	0.42	0.42
MBNDM(3)	**3.31**	**3.30**	**3.32**	**3.31**	3.30	3.31	3.31	3.30	3.30
	0.17	0.17	0.17	0.17	0.17	0.17	0.17	0.17	0.17
MEPSM	3.89	3.86	3.88	3.88	**3.14**	**3.14**	**3.11**	**3.12**	**2.77**
	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01

(B) m	16	18	20	22	24	26	28	30	32
WM(3,1)	5.59	5.29	5.07	4.89	4.77	4.72	4.51	4.48	4.36
	0.43	0.43	0.43	0.43	0.45	0.43	0.43	0.44	0.43
WM(4,1)	6.31	5.93	5.66	5.36	5.06	5.09	4.72	4.64	4.50
	0.43	0.43	0.43	0.44	0.44	0.45	0.44	0.44	0.44
MBNDM(5)	4.34	4.34	4.34	4.34	4.34	4.36	4.35	4.35	4.36
	0.26	0.26	0.26	0.25	0.25	0.26	0.25	0.26	0.26
MEPSM	**4.00**	**3.99**	**4.01**	**4.03**	**3.33**	**3.33**	**3.34**	**3.34**	**3.02**
	0.04	0.04	0.04	0.04	0.08	0.08	0.08	0.08	0.12

(C) m	16	18	20	22	24	26	28	30	32
WM(4,1)	8.28	7.71	7.49	7.23	6.98	6.86	6.72	6.59	6.48
	0.53	0.54	0.56	0.56	0.57	0.57	0.59	0.59	0.60
WM(8,1)	9.87	8.78	8.06	7.54	7.11	6.77	6.52	6.31	6.17
	0.53	0.54	0.56	0.58	0.58	0.59	0.60	0.62	0.64
MBNDM(5)	8.47	8.52	8.50	8.51	8.57	8.58	8.51	8.49	8.64
	0.37	0.39	0.38	0.38	0.38	0.39	0.38	0.38	0.39
MBNDM(8)	7.76	7.81	7.80	7.84	7.84	7.90	7.81	7.85	7.98
	0.52	0.52	0.53	0.52	0.53	0.54	0.53	0.53	0.54
MEPSM	**5.65**	**5.67**	**5.96**	**6.04**	**5.63**	**5.62**	**5.60**	**5.61**	**5.76**
	0.40	0.40	0.40	0.41	0.79	0.79	0.79	0.79	1.18

(D) m	16	18	20	22	24	26	28	30	32
WM(8,1)	24.36	23.05	22.48	22.11	21.82	21.76	21.58	21.53	21.63
	1.54	1.65	1.78	1.91	2.03	2.16	2.25	2.39	2.49
MBNDM(8)	**19.75**	**19.68**	**19.75**	**19.76**	**19.94**	**19.95**	**20.06**	**20.60**	**20.72**
	1.51	1.51	1.51	1.52	1.55	1.56	1.59	1.60	1.62
MEPSM	22.74	22.84	27.43	27.36	31.75	31.29	32.03	33.41	42.73
	4.05	4.08	3.97	3.95	7.71	7.72	7.74	7.83	11.56

Table 2. Experimental results on a protein sequence of 4 MB. Running times obtained while searching for sets of (A) 10 patterns, (B) 100 patterns, (C) 1.000 patterns and (D) 10.000 patterns.

to the WM(q, h) algorithm only in the case of large set of long patterns ($r = 10.000$ patterns and $m \geq 22$). In all other cases the algorithms perform better than previous solutions obtaining a speed up of almost 40% in particular cases.

Table 4 summarizes the speed up ratios achieved via the new algorithms. Here a ratio equal to a value x means that the MPESM algorithm is x times faster than the best solution obtained by a previous algorithm. Thus the larger the ratios mean the better the results, while ratios less than 0 mean that the MPESM algorithm is outperformed by another algorithm.

As can be viewed from that table, the newly proposed solution are in general faster then the competitors. The most significant performance enhancement is observed on

(A) m	16	18	20	22	24	26	28	30	32
WM(5,1)	5.91	5.47	5.14	4.88	4.71	4.54	4.40	4.29	4.20
	0.42	0.43	0.43	0.43	0.43	0.43	0.43	0.43	0.44
WM(6,1)	5.85	5.39	5.05	4.77	4.64	4.46	4.30	4.20	4.12
	0.43	0.43	0.43	0.42	0.42	0.43	0.43	0.42	0.44
MBNDM(5)	4.37	4.37	4.41	4.39	4.39	4.42	4.43	4.41	4.39
	0.17	0.17	0.17	0.17	0.17	0.17	0.17	0.17	0.17
MEPSM	**3.86**	**3.87**	**3.87**	**3.85**	**3.13**	**3.12**	**3.13**	**3.11**	**2.77**
	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01

(B) m	16	18	20	22	24	26	28	30	32
WM(5,1)	7.95	7.44	6.88	6.67	6.34	6.10	5.91	5.80	5.60
	0.42	0.44	0.42	0.44	0.43	0.44	0.44	0.44	0.44
WM(7,1)	8.37	7.58	7.02	6.66	6.26	6.00	5.80	5.63	5.47
	0.39	0.41	0.41	0.40	0.40	0.40	0.41	0.40	0.40
MBNDM(5)	8.22	8.20	8.08	8.17	8.20	8.19	8.21	8.20	8.21
	0.24	0.25	0.25	0.25	0.25	0.24	0.25	0.25	0.25
MBNDM(8)	7.48	7.48	7.71	7.56	7.48	7.51	7.48	7.48	7.44
	0.28	0.28	0.29	0.28	0.28	0.28	0.27	0.27	0.28
MEPSM	**4.46**	**4.51**	**4.45**	**4.42**	**3.77**	**3.71**	**3.71**	**3.72**	**3.40**
	0.04	0.04	0.04	0.04	0.08	0.08	0.08	0.08	0.12

(C) m	16	18	20	22	24	26	28	30	32
WMQ(8,1)	21.01	18.84	17.13	15.96	15.11	14.45	14.04	13.55	13.33
	0.54	0.55	0.55	0.58	0.59	0.60	0.60	0.62	0.62
WM(8,2)	27.18	20.27	17.41	15.89	14.68	14.07	13.16	12.68	12.24
	0.67	0.70	0.72	0.76	0.77	0.80	0.82	0.86	0.85
MBNDM(5)	16.60	16.61	16.57	16.56	16.61	16.54	16.62	16.72	16.65
	0.38	0.38	0.38	0.38	0.38	0.38	0.38	0.38	0.38
MEPSM	**10.37**	**10.40**	**9.92**	**9.93**	**9.62**	**9.61**	**9.61**	**9.68**	**9.83**
	0.40	0.40	0.39	0.40	0.79	0.77	0.78	0.78	1.18

(D) m	16	18	20	22	24	26	28	30	32
WM(5,2)	91.42	86.60	84.85	82.93	83.23	80.96	80.34	79.82	80.08
	1.91	2.03	2.18	2.35	2.52	2.65	2.81	2.94	3.12
WM(8,2)	90.23	74.00	68.27	**64.27**	**62.50**	**59.98**	**58.69**	**57.97**	**57.91**
	1.70	1.92	2.16	2.40	2.65	2.86	3.08	3.32	3.54
MBNDM(8)	116.23	116.98	117.73	118.21	118.91	119.29	119.18	119.94	119.78
	1.45	1.44	1.47	1.47	1.48	1.51	1.50	1.53	1.53
MEPSM	**72.54**	**72.94**	**66.26**	67.39	74.36	76.10	75.11	76.54	85.21
	3.88	3.87	3.92	3.97	7.62	7.63	7.55	7.65	11.35

Table 3. Experimental results on an english text of 4 MB. Running times obtained while searching for sets of (A) 10 patterns, (B) 100 patterns, (C) 1.000 patterns and (D) 10.000 patterns.

genome sequences, where up to more than 3 fold increase in speed has been observed. Notice that the gain in speed is more significant in the case of a genome sequence and a natural language text.

6 Conclusions

Today, most of the commodity processors are shipped with SIMD instruction sets. Recent studies [20,9,10] benefiting from that technology have been reporting very significant results in pattern matching, where most of the time they outperform their

(A) m	16	20	24	28	32
genome	1.13	1.13	1.41	1.37	1.47
protein	0.85	0.85	1.05	1.06	1.19
nat.lang.	1.13	1.13	1.40	1.37	1.48

(B) m	16	20	24	28	32
genome	2.11	1.92	2.03	1.85	1.90
protein	1.08	1.08	1.30	1.30	1.44
nat.lang.	1.67	1.57	1.66	1.56	1.64

(C) m	16	20	24	28	32
genome	3.12	3.15	3.18	2.91	2.58
protein	1.37	1.37	1.23	1.16	1.07
nat.lang.	1.60	1.67	1.57	1.36	1.24

(D) m	16	20	24	28	32
genome	2.34	2.51	2.33	2.15	1.86
protein	0.86	0.72	0.62	0.62	0.48
nat.lang.	1.24	1.03	0.84	0.78	0.67

Table 4. The speed ups obtained via MEPSM compared with the second best results on sets of 10 (A), 100 (B), 1.000 (C) and 10.000 (D) patterns.

alternatives. This reminds us to consider using SIMD instructions in design and implementation of the algorithms in practice [22].

We have presented a new algorithm targeting patterns shorter than 32 bytes in practice. Experimental results depicted that our proposal becomes a strong alternative to the best known previous solutions when length of the patterns in the set is longer than 16 bytes. We have observed speed ups in orders of magnitudes particularly on genome sequences and English texts as can be seen from Table 4. The CRC filter scales well with the increasing size of the pattern sets.

When the length of the patterns in the set increases, the competitors start scanning quicker as their shift mechanisms improve with longer patterns, and hence, the speed ups compared with our proposal decreases. On patterns shorter than 16 bytes, the CRC filter is not very competitive in its current implementation, and thus, needs further studies to get better results.

In our future work we intend to analyze in depth the impact of the CRC filter in searching sets of short patterns with a length less than 16 characters. We are convinced that good improvements in this direction are possible.

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search.* Commun. ACM, 18(6) 1975, pp. 333–340.
2. C. Allauzen, M. Crochemore, and M. Raffinot: *Factor oracle: a new structure for pattern matching*, in Proc. of SOFSEM'99, LNCS 1725, Springer-Verlag, 1999, pp. 291–306.
3. R. Baeza-Yates and G. Gonnet: *A new approach to text searching.* Communications of the ACM, 35(10) 1992, pp. 74–82.
4. O. Ben-Kiki, P. Bille, D. Breslauer, L. Gasieniec, R. Grossi, and O. Weimann: *Optimal packed string matching*, in IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2011), vol. 13 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2011, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 423–432.
5. D. Cantone, S. Faro, and E. Giaquinta: *A compact representation of nondeterministic (suffix) automata for the bit-parallel approach*, in Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, vol. 6129 of Lecture Notes in Computer Science, Springer, 2010, pp. 288–298.
6. D. Cantone, S. Faro, and E. Giaquinta: *A compact representation of nondeterministic (suffix) automata for the bit-parallel approach.* Inf. Comput., 213 2012, pp. 3–12.

7. D. Cantone, S. Faro, and E. Giaquinta: *On the bit-parallel simulation of the nondeterministic aho-corasick and suffix automata for a set of patterns.* J. Discrete Algorithms, 11 2012, pp. 25–36.
8. M. Crochemore and W. Rytter: *Text algorithms*, Oxford University Press, 1994.
9. S. Faro and M. O. Külekci: *Fast multiple string matching using streaming SIMD extensions technology*, in String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, vol. 7608 of Lecture Notes in Computer Science, Springer, 2012, pp. 217–228.
10. S. Faro and M. O. Külekci: *Fast packed string matching for short patterns*, in Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, SIAM, 2013, pp. 113–121.
11. S. Faro and T. Lecroq: *The exact string matching problem: a comprehensive experimental evaluation.* Arxiv preprint arXiv:1012.2547, 2010.
12. S. Faro and T. Lecroq: *2001-2010: Ten years of exact string matching algorithms*, in Proceedings of the Prague Stringology Conference 2011, J. Holub and J. Zdárek, eds., Prague Stringology Club, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2011, pp. 1–2.
13. S. Faro and T. Lecroq: *Fast searching in biological sequences using multiple hash functions*, in 12th IEEE International Conference on Bioinformatics & Bioengineering, BIBE 2012, IEEE Computer Society, 2012, pp. 175–180.
14. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) 2013, p. 13.
15. S. Faro, and E. Pappalardo: *Ant-CSP: An Ant Colony Optimization Algorithm for the Closest String Problem*, in SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference on Current Trends in Theory and Practice of Computer Science, vol. 5901 of Lecture Notes in Computer Science, Springer, 2010, pp. 360–281.
16. S. Gog, K. Karhu, J. Karkkainen, V. Makinen, and N. Valimaki: *Multi-pattern matching with bidirectional indexes*, in Computing and Combinatorics, J. Gudmundsson, J. Mestre, and T. Viglas, eds., vol. 7434 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 384–395.
17. R. M. Karp and M. O. Rabin: *Efficient randomized pattern-matching algorithms.* IBM J. Res. Dev., 31(2) 1987, pp. 249–260.
18. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt: *Fast pattern matching in strings.* SIAM J. Comput., 6(1) 1977, pp. 323–350.
19. M. O. Külekci: *TARA: An algorithm for fast searching of multiple patterns on text files*, in IEEE 22nd International Symposium on Computer and Information Sciences (ISCIS), 2007, pp. 1–6.
20. M. O. Külekci: *Filter based fast matching of long patterns by using SIMD instructions*, in Proceedings of the Prague Stringology Conference, 2009, pp. 118–128.
21. M. O. Külekci: *BLIM: A new bit-parallel pattern matching algorithm overcoming computer word size limitation.* Mathematics in Computer Science, 3(4) 2010, pp. 407–420.
22. S. Ladra, O. Pedreira, J. Duato, and N. Brisaboa: *Exploiting SIMD instructions in current processors to improve classical string algorithms*, in Advances in Databases and Information Systems, T. Morzy, T. Harder, and R. Wrembel, eds., vol. 7503 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 254–267.
23. G. Navarro and K. Fredriksson: *Average complexity of exact and approximate multiple string matching.* Theor. Comput. Sci., 321(2-3) 2004, pp. 283–290.
24. G. Navarro and M. Raffinot: *A bit-parallel approach to suffix automata: Fast extended string matching*, in Combinatorial Pattern Matching, Springer, 1998, pp. 14–33.
25. G. Navarro and M. Raffinot: *Fast and flexible string matching by combining bit-parallelism and suffix automata.* ACM J. Experimental Algorithmics, 5 2000, p. 4.
26. G. Navarro and M. Raffinot: *Flexible pattern matching in strings - practical on-line search algorithms for texts and biological sequences*, Cambridge Univ. Press, 2002.
27. W.W. Peterson and D.T. Brown. *Cyclic Codes for Error Detection.* Proceedings of the IRE, 49 (1): 228–235, 1961.
28. E. Rivals, L. Salmela, P. Kiiskinen, P. Kalsi, and J. Tarhio: *Mpscan: Fast localisation of multiple reads in genomes*, in Proc. of WABI, 2009, pp. 246–260.

29. M. Roesch: *Snort - lightweight intrusion detection for networks*, in Proceedings of the 13th USENIX conference on System administration, LISA '99, Berkeley, CA, USA, 1999, USENIX Association, pp. 229–238.
30. L. Salmela and J. Tarhio: *Multi-pattern string matching with q-grams.* ACM Journal of Experimental Algorithmics, 11 2006, pp. 1–19.
31. B. Zhang, X. Chen, X. Pan and Z. Wu. *High concurrence Wu-Manber multiple patterns matching algorithm.* Proceedings of the International Symposium on Information Proces, p. 404, 2009.
32. S. Wu and U. Manber: *Agrep – a fast approximate pattern-matching tool*, in Proceedings of USENIX Winter 1992 Technical Conference, USENIX Association, 1992, pp. 153–162.
33. S. Wu and U. Manber: *Fast text searching: allowing errors.* Commun. ACM, 35(10) 1992, pp. 83–91.
34. S. Wu and U. Manber: *A fast algorithm for multi-pattern searching*, Report TR-94-17, Dep. of Computer Science, University of Arizona, Tucson, AZ, 1994.

Improved and Self-Tuned Occurrence Heuristics*

Domenico Cantone and Simone Faro

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

Abstract. In this note we present three efficient variations of the *occurrence heuristic*, adopted by many exact string matching algorithms and firstly introduced in the well-known Boyer-Moore algorithm. Our first heuristic, called *improved-occurrence heuristic*, is a simple improvement of the rule introduced by Sunday in his Quick-Search algorithm. Our second heuristic, called *worst-occurrence heuristic*, achieves its speed-up by selecting the relative position which yields the largest average advancement. Finally, our third heuristic, called *jumping-occurrence heuristic*, uses two characters for computing the next shift, whose distance allows one to maximize the average advancement. The worst-occurrence and jumping-occurrence heuristics tune their parameters according to the text characters' distribution. Experimental results show that the new proposed heuristics achieve very good results on average, especially in the case of small alphabets.

Keywords: string matching, experimental algorithms, text-processing, occurrence heuristics, frequency of characters, tuned-search approach

1 Introduction

Given a text t and a pattern p over some alphabet Σ, the *string matching problem* consists in finding *all* occurrences of the pattern p in the text t. In a computational model in which the matching algorithm is restricted to read all the characters of the text one by one, the optimal complexity is $\mathcal{O}(n)$. However, in several practical cases it is not necessary to read all text characters, achieving sublinear performances on average. The optimal average complexity is $\mathcal{O}(n/m \log \sigma)$ [18] and it is interesting to note that many of such algorithms have an even worse $\mathcal{O}(nm)$-time complexity in the worst-case [9,4,5,6,10,11,12].

This is the case for the celebrated Boyer-Moore (BM) algorithm [2], the progenitor of several algorithmic variants which aim at efficiently computing shift increments close to optimal. The Boyer-Moore algorithm computes shift increments as the maximum value suggested by the *good-suffix heuristic* and the *occurrence heuristic*, provided that both of them are applicable. However, many subsequent efficient variants of the Boyer-Moore algorithm just dropped the good-suffix heuristic and based the calculation of the shift increments only on variants of the occurrence heuristic. Some of such variants are still considered among the most efficient algorithms in practical cases (see [9]).

The occurrence heuristic uses a single character for shifting. Specifically, it states that when a mismatch is found at a given position j of the text, then the pattern can be safely shifted in such a way that its rightmost occurrence of the mismatching character in the text, if present, is aligned with the relative position j in the text.

* This work has been partially supported by project PRISMA PON04a2 A/F funded by the Italian Ministry of University within PON 2007-2013 framework.

In this paper we present three improvements of the occurrence heuristic which turn out to be more efficient in practical cases, especially in the case of small alphabets. In particular, we will introduce the following heuristics:

1. the *improved-occurrence heuristic*, which is based on the match of the rightmost character of the pattern with the corresponding character in the text;
2. the *worst-occurrence heuristic*, which selects a relative position yielding the largest average advancement according to the text characters' distribution;
3. the *jumping-occurrence heuristic*, which uses two characters for computing the shift advancements in the searching phase. The relative distance between the two characters is computed so as to maximize the average shift advancements, based on the text characters' distribution.

The paper is organized as follows. Some useful notations and terminology are preliminarily recalled in Section 2. Then, in Section 3 we briefly revise the occurrence heuristic and some of its variants. In Section 4 we present the first of our proposed occurrence heuristics, namely the improved-occurrence heuristic, and in Sections 5 and 6 we introduce the worst-occurrence and the jumping-occurrence heuristics, respectively. Finally, in Section 7 we present and comment on experimental results on the performance of our proposed heuristics in comparison with the best known algorithms present in literature based on the occurrence heuristic. Finally, we draw our conclusions in Section 8.

2 Notations and Terminology

A string p of length $|p| = m \geq 0$ over a finite alphabet Σ is represented as a finite array $p[0\,..\,m-1]$. By $p[i]$ we denote the $(i+1)$-st character of p, for $0 \leq i < m$. Likewise, by $p[i\,..\,j]$ we denote the substring of p contained between the $(i+1)$-st and the $(j+1)$-st characters of p, where $0 \leq i \leq j < m$.

Let t be a text of length n and let p be a pattern of length m. If the character $p[0]$ is aligned with the character $t[s]$ of the text, so that $p[i]$ is aligned with $t[s+i]$, for $0 \leq i \leq m-1$, we say that the pattern p has *shift* s in t. In this case, the substring $t[s\,..\,s+m-1]$ is called the *current window* of the text. If $t[s\,..\,s+m-1] = p$, we say that the shift s is *valid*. Then the *string matching problem* consists in finding all valid shifts of p in t, for given pattern p and text t.

In general, most string matching algorithms work as follows. They scan the text by sliding a text window whose size is generally equal to m. For each text window, its characters are compared with the corresponding characters of the pattern or suitable transitions are performed on some kind of automaton (this specific phase is called a *matching attempt*). After a complete match of the pattern is found or a mismatch is detected, the current window is shifted to the right by a certain number of positions. This phase is usually referred to as the *sliding window mechanism*. When the search starts, the left end of the text and of the current window are aligned. Subsequently, the sliding window mechanism is repeated until the right end of the window goes past the right end of the text. Each matching attempt can be naturally associated with the position s in the text where the current window $t[s\,..\,s+m-1]$ is positioned.

3 The Occurrence Heuristic and Some of its Variants

The well-known *occurrence heuristic* was introduced for the first time in [2] as one of the shift rules used by the Boyer-Moore algorithm. The work in [17,8] provides a uniform framework for describing all safe shifts provided by the Boyer-Moore-type pattern matching algorithms. Specifically, during a matching attempt the Boyer-Moore algorithm scans the current window (of the text) from right to left and, at the end of the matching phase, it computes the shift increment as the largest value given by the *good-suffix* and the *occurrence* heuristics.

The occurrence heuristic states that if $c = t[s+i] \neq p[i]$ is the first mismatching character (with $0 \leq i \leq m-1$), while scanning p and t (with shift s) from right to left, then p can be safely shifted in such a way that its rightmost occurrence of c, if present, is aligned with position $(s+i)$ in t (provided that such an occurrence lies in $p[0\,..\,i-1]$, otherwise the occurrence heuristic has no effect). In the case in which c does not occur in p, then p can be safely shifted just past position $(s+i)$ in t. More formally, the shift increment suggested by the occurrence heuristic is given by $(bc_p(t[s+i]) + i - m + 1)$, where, for $c \in \Sigma$, $bc_p(c) =_{\text{Def}} \min(\{k \mid 0 \leq k \leq m-1 \text{ and } p[m-k-1] = c\} \cup \{m\})$.

Observe that the table bc_p of the occurrence heuristic, for a given a pattern p of length m, can be computed in $\mathcal{O}(m+\sigma)$ time and $\mathcal{O}(\sigma)$ space, where σ is the size of the alphabet Σ.

Due to the simplicity and ease of implementation of the occurrence heuristic, some variants of the Boyer-Moore algorithm were based just on it, dropping the good-suffix heuristic. For instance, Horspool [13] suggested the following simplification of the original Boyer-Moore algorithm, which performs better in practical cases. He just dropped the good-suffix heuristic and proposed to compute shift advancements in such a way that the rightmost character $t[s+m-1]$ of the current window is aligned with its rightmost occurrence on $p[0\,..\,m-2]$, if present; otherwise the pattern is advanced just past the window. This amounts to advance the shift by $hbc_p(t[s+m-1])$ positions, where $hbc_p(c) =_{\text{Def}} \min(\{k \mid 1 \leq k \leq m-1 \text{ and } p[m-k-1] = c\} \cup \{m\})$.

The Quick-Search algorithm, presented in [16], also uses a modification of the original occurrence heuristic, much along the same lines of the Horspool algorithm. Specifically, it is based on the following observation: when a mismatching character is encountered, the pattern is always shifted to the right by at least one character, but never by more than m characters. Thus, the character $t[s+m]$ is always involved in testing for the next alignment. So, one can apply the bad character rule to $t[s+m]$, rather than to the mismatching character, possibly obtaining larger shift advancements. This corresponds to advance the shift by $qbc_p(t[s+m])$ positions, where $qbc_p(c) =_{\text{Def}} \min(\{k \mid 1 \leq k \leq m-1 \text{ and } p[m-k] = c\} \cup \{m+1\})$.

Other efficient variants of the Boyer-Moore algorithm extend the previous algorithms in that their occurrence heuristics use two characters rather than just one. For instance the Zhu-Takaoka algorithm [19] extends the Horspool algorithm by using the last two characters $t[s+m-2]$ and $t[s+m-1]$ in place of only $t[s+m-1]$. A more effective algorithm, due to Berry and Ravindran [1], extends the Quick-Search algorithm in a similar manner, by using the characters $t[s+m]$ and $t[s+m+1]$ in place of only $t[s+m]$. It is to be noticed, though, that the precomputation of the table used by an occurrence heuristic based on two text characters requires $\mathcal{O}(\sigma^2)$-space and $\mathcal{O}(m+\sigma^2)$-time complexity.

4 A Simple Improved Occurrence Heuristic

For a given shift s, the Horspool and the Quick-Search algorithms compute their shift advancements by applying the occurrence heuristic on a fixed position $s+q$ of the text, with $q = m-1$ and $q = m$, respectively. We refer to the value q as the *occurrence relative position.*

In favorable conditions, it may be possible to use an occurrence relative position $q > m$, which may lead to even larger advancements, provided that no matching can ever possibly be skipped. In such a situation, we say that the occurrence relative position q is *safe (for shifting).*

To this purpose, we begin by introducing the *generalized occurrence function* $gbc_p(i,c)$. Suppose the pattern p has shift s in the text t. For a given occurrence relative position $0 \leq i \leq 2m-1$, $gbc_p(i, t[s+i])$ is the shift advancement such that the character $t[s+i]$ is aligned with its rightmost occurrence in $p[0 \,..\, \min(i,m)-1]$, if present; otherwise $gbc_p(i, t[s+i])$ evaluates to $i+1$ (this corresponds to advance the pattern just past position $s+i$ of the text). This amounts to putting

$$gbc_p(i,c) =_{\text{Def}} \min(\{i-k \mid 0 \leq k < \min(i,m) \text{ and } p[k]=c\} \cup \{i+1\}),$$

for $c \in \Sigma$ and $i \geq 0$.[1] Plainly, $gbc_p(i,c) \geq 1$ always holds. Additionally, the shift rules of the Horspool and Quick-Search algorithms can be expressed in terms of the generalized occurrence function just defined by $hbc_p(c) = gbc_p(m-1,c)$ and $qbc_p(c) = gbc_p(m,c)$, respectively, for $c \in \Sigma$.

We will define our improved occurrence heuristic (IOH) in terms of the generalized occurrence function $gbc_p(i,c)$. Let again s be the shift of the current text window. We distinguish the following two cases:

Case $p[m-1] = t[s+m-1]$**:**
Let i_0 be the rightmost position in the substring $p[0 \,..\, m-2]$ such that $p[i_0] = p[m-1]$, provided that $p[m-1]$ occur in $p[0 \,..\, m-2]$; otherwise let i_0 be -1. Then the occurrence relative position $q_1 = 2m - i_0 - 2$ is safe for shifting, since no occurrence of the character $p[m-1]$ exists from position i_0+1 to position $m-2$. More formally, q_1 can be defined as

$$q_1 =_{\text{Def}} \min(\{2m-i-2 \mid p[i] = p[m-1] \text{ and } 0 \leq i \leq m-2\} \cup \{2m-1\})\,.$$

Case $p[m-1] \neq t[s+m-1]$**:**
In this case, let i_0 be the rightmost position in $p[0 \,..\, m-2]$ such that $p[i_0] \neq p[m-1]$, provided that $p[0 \,..\, m-2]$ contain some character distinct from $p[m-1]$, otherwise let i_0 be -1. Then the occurrence relative position $q_2 = 2m - i_0 - 2$ is safe for shifting, since no character different from $p[m-1]$ exists from position i_0+1 to position $m-2$. More formally, q_2 is defined as

$$q_2 =_{\text{Def}} \min(\{2m-i-2 \mid p[i] \neq p[m-1] \text{ and } 0 \leq i \leq m-2\} \cup \{2m-1\})\,.$$

The two occurrence relative positions q_1 and q_2 are then used by our heuristic IOH to calculate the shift advancements during the searching phase of the algorithm ImprovedOccurrenceMatcher in Figure 1, based on the following two occurrence functions

$$ibc1_p(c) =_{\text{Def}} gbc_p(q_1,c)\,, \qquad ibc2_p(c) =_{\text{Def}} gbc_p(q_2,c)\,.$$

These are computed by procedure PrecomputeIOH, shown in Figure 1, in $O(m+\sigma)$ time and $O(\sigma)$ space.

[1] A restricted variant of the generalized occurrence function gbc_p was presented in [7].

```
PrecomputeIOH($p, m, step$)
1.  for each $c \in \Sigma$ do
2.      $ibc[c] \leftarrow step + 1$
3.  for $i \leftarrow 0$ to $m - 1$ do
4.      $ibc[p[i]] \leftarrow step - i$
5.  return $ibc$
```

```
ImprovedOccurrenceMatcher($p, m, t, n$)
1.  $step_1 \leftarrow step_2 \leftarrow 2m - 1$
2.  for $i \leftarrow 0$ to $m - 2$ do
3.      if $p[i] = p[m-1]$ then
4.      then $step_1 \leftarrow 2m - i - 2$
5.      else $step_2 \leftarrow 2m - i - 2$
6.  $ibc_1 \leftarrow$ PrecomputeIOH($p, m, step_1$)
7.  $ibc_2 \leftarrow$ PrecomputeIOH($p, m, step_2$)
8.  $s \leftarrow 0$
9.  while ($s \leq n - m$) do
10.     if ($p[m-1] = t[s+m-1]$) then
11.         $i \leftarrow 0$
12.         while ($i < m$ and $p[i] = t[s+i]$) do
13.             $i \leftarrow i + 1$
14.         if ($i = m$) then Output($s$)
15.         $s \leftarrow s + ibc_1[t[s + step_1]]$
16.     else $s \leftarrow s + ibc_2[t[s + step_2]]$
```

Figure 1. A string matching algorithm based on the heuristic IOH.

5 A Self-Tuned Occurrence Heuristic

For a pattern p of length m, a text t, and a shift s, the heuristic IOH presented in the previous section computes shift advancements using the rule $ibc1_p$ or $ibc2_p$, based on two different relative positions, according to whether the last character of the pattern p matches its corresponding text character $t[s+m-1]$ or not. Differently, the Horspool and the Quick-Search algorithms compute their shift advancements by applying the occurrence heuristic on a fixed position $s+q$ of the text, with q equal, respectively, to $m-1$ and to m. In this section we will show that, given a pattern p and a text t with known character distribution, we can compute efficiently an occurrence relative position, to be called *worst-occurrence relative position*, which ensures the largest shift advancement on the average. The *worst-occurrence heuristic* (WOH) is then the corresponding occurrence heuristic based on the worst-occurrence relative position.

5.1 Finding the worst-occurrence relative position

Again, let t and p be respectively a text and a pattern over a common alphabet Σ and let $f : \Sigma \rightarrow [0,1]$ be the relative frequency function of the characters of t, so that $\sum_{c\in\Sigma} f(c) = 1$ holds.

For a given occurrence relative position $0 \leq i \leq m$, the average shift advancement of the generalized occurrence function gbc_p is given by the function

$$adv_{p,f}(i) =_{\mathrm{Def}} \sum_{c\in\Sigma} f(c) \cdot gbc_p(i, c)\,. \tag{1}$$

We then define the *worst-occurrence relative position* q^* as the smallest position $0 \leq q \leq m$ which maximizes $adv_{p,f}(q)$, i.e.,

$$q^* =_{\mathrm{Def}} \min\{q \mid 0 \leq q \leq m \text{ and } adv_{p,f}(q) = \max_{0\leq i\leq m} adv_{p,f}(i)\}\,.$$

Procedure FindWorstOccurrence in Figure 2 computes efficiently the position q^*, by exploiting the recurrence

$$adv_{p,f}(i) = \begin{cases} 1 & \text{if } i = 0 \\ adv_{p,f}(i-1) - f(p[i-1]) \cdot gbc_p(i-1, p[i-1]) + 1 & \text{if } 1 \leq i \leq m \end{cases}$$

```
FindWorstOccurrence($p, m, \Sigma, f$)
1.  for each $c \in \Sigma$ do
2.      $lp[c] \leftarrow -1$
3.  $adv \leftarrow 1$
4.  $max \leftarrow 1$
5.  $q \leftarrow 0$
6.  for $i \leftarrow 1$ to $m$ do
7.      $gbc \leftarrow i - lp[p[i-1]] - 1$
8.      $adv \leftarrow adv - f(p[i-1]) \cdot gbc + 1$
9.      $lp[p[i-1]] \leftarrow i - 1$
10.     if $(adv > max)$ then
11.         $max \leftarrow adv$
12.         $q \leftarrow i$
13. return $q$
```

```
PrecomputeWOH($p, m, q$)
1.  for each $c \in \Sigma$ do
2.      $wo[c] \leftarrow q + 1$
3.  for $i \leftarrow 0$ to $q - 1$ do
4.      $wo[p[i]] \leftarrow q - i$
5.  return $wo$

WorstOccurrenceMatcher($p, m, t, n$)
1.  $q \leftarrow$ FindWorstOccurrence($p, m, \Sigma, f$)
2.  $wo \leftarrow$ PrecomputeWOH($p, m, q$)
3.  $s \leftarrow 0$
4.  while $(s \leq n - m)$ do
5.      $i \leftarrow 0$
6.      while $(i < m$ and $p[i] = t[s+i])$ do
7.          $i \leftarrow i + 1$
8.      if $(i = m)$ then Output($s$)
9.      $s \leftarrow s + wo[t[s+q]]$
```

Figure 2. The procedure FindWorstOccurrence, the procedure PrecomputeWOH and the algorithm WorstOccurrenceMatcher.

for the calculation of the function $adv_{p,f}$ (lines 3 and 8), which, in turn, is based on the recurrence

$$gbc_p(i,c) = \begin{cases} 1 & \text{if } i = 0 \text{ or } c = p[i-1] \\ gbc_p(i-1,c) + 1 & \text{otherwise,} \end{cases}$$

for $0 \leq i \leq m$ and $c \in \Sigma$.

Notice that the entries of the generalized occurrence function gbc_p present in the above recurrence relation for $adv_{p,f}$ are only of the form $gbc_p(j, p[j])$. These can be expressed readily in terms of the *last-position* functions $lp_p^i : \Sigma \to \{-1, 0, \ldots, m-1\}$, defined (for $i = 0, 1, \ldots, m$) by

$$lp_p^i(c) =_{\text{Def}} \max(\{j \mid 0 \leq j < i \text{ and } p[j] = c\} \cup \{-1\}),$$

i.e., $lp_p^i(c)$ is the rightmost position of c in $p[0\,..\,i-1]$, if c is present in $p[0\,..\,i-1]$, otherwise $lp_p^i(c)$ is -1. In fact, we have

$$gbc_p(i, p[i]) = i - lp_p^i(p[i]),$$

for $0 \leq i \leq m-1$ (cf. line 7 of the **for-loop**).

The last-position functions can efficiently be computed during a left to right scanning of the pattern. These are maintained as a single array lp of size σ by the procedure FindWorstOccurrence. The array lp is initialized at lines 1-2 and subsequently updated at line 9 of the **for-loop**, by resorting to the recursive relation

$$lp_p^i(c) = \begin{cases} -1 & \text{if } i = 0 \\ i-1 & \text{if } i > 0 \land c = p[i-1] \\ lp_p^{i-1}(c) & \text{if } i > 0 \land c \neq p[i-1]. \end{cases}$$

It is easy to oberve that the procedure FindWorstOccurrence has an overall $\mathcal{O}(m + \sigma)$-time and $\mathcal{O}(\sigma)$-space complexity.

5.2 The worst-occurrence heuristic

The *worst-occurrence heuristic* uses the position q^* computed by the procedure FindWorstOccurrence to calculate shift advancements during the searching phase in such a way that the character $t[s+q^*]$ is aligned with its rightmost occurrence on $p[0\,..\,q^*-1]$, if present; otherwise the pattern is advanced just past position $s+q^*$ of the text. This corresponds to advance the shift by $wo_p(t[s+q^*])$ positions, where

$$wo_p(c) =_{\text{Def}} \min(\{i \mid 1 \leq i \leq q^* \text{ and } p[q^*-i] = c\} \cup \{q^*+1\})\,.$$

Observe that, for $q^* = 0$, the advancement is equal to 1. The resulting algorithm can be immediately translated into programming code (see Figure 2 for a simple implementation). The procedure PrecomputeWOH, shown in Figure 2, computes the table which implements the worst-occurrence heuristic in $\mathcal{O}(m+\sigma)$ time and $\mathcal{O}(\sigma)$ space.

5.3 Finding the relative frequency of characters

The frequency of characters in texts has often been used in string matching algorithms for speeding up the searching process [16,1,15]. Such an approach is particularly useful when one is searching texts in natural languages, whose character distributions are well studied, and therefore known in advance. However, also in the case of texts in natural languages, the exact character distribution can not be predicted, since character frequencies may depend both on the writer and on the subject. The situation may become even worse in the case of other types of sequences. In such contexts, different approaches can be adopted for retrieving good approximations of the frequency of characters in order to apply accurately the worst-occurrence heuristic presented above. Here we propose some of them.

(i) In a preprocessing phase, compute the character frequencies of an initial segment of the text (say of no more than γ characters).

(ii) Run the first γ iterations of the algorithm WorstOccurrenceMatcher, assuming *a priori* a default distribution of characters (e.g., the uniform distribution). At the same time, compute the relative frequency of the first γ characters and then recompute the occurrence heuristic according to the estimated frequency.

(iii) While running the algorithm WorstOccurrenceMatcher, keep updating the relative frequencies of the characters. At regular intervals (say of γ characters), or when the difference between the current relative frequencies and the one used in the worst-occurrence heuristic exceeds a threshold, recompute the heuristic.

From our tests, it turns out that when the distribution of characters does not vary very much along the text, a good approximation of the frequencies can be computed even for quite small values of γ in the case of strategies (i) and (ii). For instance, in our experiments reported in Section 7 we used the value $\gamma = 100$, in combination with strategy (i). When the character frequencies tend to vary very much along the text (for instance, in the case of multi-language texts or in musical sequences), strategy (iii) might be preferable. However, one must keep in mind that the overhead can sensibly affect the algorithm performance.

6 A Jumping-Occurrence Heuristic

We recall that, for a pattern p of length m, the occurrence heuristics of the Zhu-Takaoka [19] and the Berry-Ravindran [1] algorithms are based on two consecutive characters, starting at positions $m-2$ and m, respectively. In both cases, the distance between the two characters involved in the occurrence heuristics is 1. We refer to such a distance as the *occurrence jump distance.*

It may be possible that other occurrence jump distances generate larger shift advancements. We will show in this section how, given a pattern p and a text t with known character distribution, we can compute efficiently an optimal occurrence jump distance which ensures the largest shift advancements on the average. The *jumping-occurrence heuristic* will be then the occurrence heuristic based on two characters with optimal occurrence jump distance.

6.1 Finding the optimal occurrence jump distance

Again, let p be a pattern of length m. To begin with, we introduce the *generalized double occurrence function* $gbc^2_p(i,j,c_1,c_2)$ *relative to* p, with $0 \leq i \leq m$, $1 \leq j \leq m$ and $c_1, c_2 \in \Sigma$, intended to calculate the largest *safe* shift advancement for p compatible with the constraints $t[s+i] = c_1$ and $t[s+i+j] = c_2$, when p has shift s with respect to a text t. Thus, we put:

$$\begin{aligned} gbc^2_p(i,j,c_1,c_2) =_{\text{Def}} \min(&\{i-k \mid m-j \leq k < i \wedge p[k] = c_1\} \\ &\cup \{i-k \mid 0 \leq k < \min(m-j,i) \wedge p[k] = c_1 \wedge p[k+j] = c_2\} \\ &\cup \{i+j-k \mid 0 \leq k < j \wedge p[k] = c_2\} \\ &\cup \{i+j+1\}\,)\,. \end{aligned} \tag{2}$$

Plainly, $gbc^2_p(i,j,c_1,c_2) \geq 1$ always holds and it can easily be checked that

$$gbc_p(i,c_1) < i+j-m+1 \quad \Longrightarrow \quad gbc^2_p(i,j,c_1,c_2) = gbc_p(i,c_1)\,. \tag{3}$$

Additionally, the shift rules of the Zhu-Takaoka and Berry-Ravindran algorithms can be expressed in terms of the generalized double occurrence function as, respectively, $gbc^2_p(m-2,1,c_1,c_2)$ and $gbc^2_p(m,1,c_1,c_2)$.

In the following we will refer to the parameters i and j of gbc^2_p as the *relative occurrence position* and the *occurrence jump distance*, respectively. For fixed values of the relative occurrence position and the occurrence jump distance, the generalized double occurrence function can be computed in $\mathcal{O}(\sigma^2 + m\sigma)$ time and $\mathcal{O}(\sigma^2)$ space.

Let us fix, momentarily, the relative occurrence position i to $m-1$ and let $f : \Sigma \to [0,1]$ be the relative frequency of the characters in the text t. For a given $1 \leq \ell \leq m$, the probability that the generalized occurrence function gbc_p yields a shift advancement of length at least ℓ when inspecting the character at relative position $m-1$ is

$$Pr\{gbc_p(m-1,c) \geq \ell \mid c \in \Sigma\} = \sum_{\substack{c \in \Sigma \\ gbc_p(m-1,c) \geq \ell}} f(c)\,.$$

Example 1. Let $p = \mathsf{ACGAACT}$ be a pattern of $m = 7$ characters over the alphabet $\Sigma = \{\mathsf{A},\mathsf{C},\mathsf{G},\mathsf{T}\}$ of four *elements* with a relative frequency f such that $f(\mathsf{A}) = 0.3$, $f(\mathsf{C}) = 0.1$, $f(\mathsf{G}) = 0.4$ and $f(\mathsf{T}) = 0.2$. The shift advancements given by each character at the relative occurrence position $m - 1 = 6$ are $gbc_p(6, \mathsf{A}) = 2$, $gbc_p(6, \mathsf{C}) = 1$, $gbc_p(6, \mathsf{G}) = 4$, and $gbc_p(6, \mathsf{T}) = 7$, respectively. Thus, $adv_{p,f}(6) = 3.7$.

The probabilities to have a shift advancement of length at least ℓ, for $1 \leq \ell \leq 8$, are given by the following values

$$\begin{array}{ll}
Pr\{gbc6c \geq 1 \mid c \in \Sigma\} = f(\mathsf{A}) + f(\mathsf{C}) + f(\mathsf{G}) + f(\mathsf{T}) = 1; & Pr\{gbc6c \geq 5 \mid c \in \Sigma\} = f(\mathsf{T}) = 0.2 \\
Pr\{gbc6c \geq 2 \mid c \in \Sigma\} = f(\mathsf{A}) + f(\mathsf{G}) + f(\mathsf{T}) = 0.9; & Pr\{gbc6c \geq 6 \mid c \in \Sigma\} = f(\mathsf{T}) = 0.2 \\
Pr\{gbc6c \geq 3 \mid c \in \Sigma\} = f(\mathsf{G}) + f(\mathsf{T}) = 0.6; & Pr\{gbc6c \geq 7 \mid c \in \Sigma\} = f(\mathsf{T}) = 0.2 \\
Pr\{gbc6c \geq 4 \mid c \in \Sigma\} = f(\mathsf{G}) + f(\mathsf{T}) = 0.6; & Pr\{gbc6c \geq 8 \mid c \in \Sigma\} = 0\,.
\end{array}$$

Let j be a fixed relative jump distance to be used by the generalized double occurrence function gbc^2_p with relative occurrence position $m - 1$. In order for the character $t[s + m - 1 + j]$, at the relative position $m - 1 + j$, to be involved in the computation of the shift advancement by the function gbc^2_p, we must have

$$gbc_p(m - 1, t[s + m - 1]) \geq j$$

(cf. (3)). Thus, for a fixed bound $0 \leq \beta \leq 1$, the computation of the shift advancement will involve the second character with a probability of at least β if and only if its jump distance j satisfies

$$Pr\{gbc_p(m - 1, c) \geq j \mid c \in \Sigma\} \geq \beta\,.$$

This suggests to use the following relative jump distance

$$j^*_\beta =_{\text{Def}} \max\Big\{\ell \mid 1 \leq \ell \leq m \text{ and } Pr\big\{gbcm - 1c \geq \ell \mid c \in \Sigma\big\} \geq \beta\Big\}$$

in the jumping-occurrence heuristic to be presented in the next section, at least in the case in which the relative occurrence position i is $m - 1$. Plainly, the same argument can be generalized to any relative occurrence position.

In Example 1, if we set the bound $\beta = 0.5$, we obtain a relative jump distance $j^*_{0.5} = 4$. In other words, for the relative jump distance $j^*_{0.5} = 4$, the character $t[s + 10]$ will be involved in the computation of the shift advancement in at least 50% of the times, whereas in the remaining cases only the first character $t[s + 6]$ will be involved. In practical cases we set $\beta = 0.9$. This will yield, in Example 1, a relative jump distance $j^*_{0.9} = 2$.

6.2 The Jumping-Occurrence Heuristic

For a pattern p of length m, the *jumping-occurrence heuristic* makes use of the occurrence relative position q^* returned by the procedure FindWorstOccurrence described in Section 5.1. Such a position q^* and the corresponding jump distance j^*_β computed by procedure FindJumpDistance are then used by the jumping-occurrence heuristic to calculate shift advancements during the searching phase in such a way that the characters $t[s+q^*]$ and $t[s+q^*+j^*_\beta]$ are aligned with their rightmost occurrence in p. In particular, this corresponds to advance the shift by $jbc_{p,\beta}(t[s+q^*], t[s+q^*-j^*_\beta])$ positions, where

$$jbc_{p,\beta}(c_1, c_2) =_{\text{Def}} gbc^2_p(q^*, j^*_\beta, c_1, c_2)\,.$$

The resulting algorithm is shown in Figure 3. The procedure PrecomputeJOH computes the table which implements the jumping-occurrence heuristic in $\mathcal{O}(\sigma^2 + m\sigma)$ time and $\mathcal{O}(\sigma^2)$ space.

```
PrecomputeJOH($p, m, i, j$)
 1.  for each $a \in \Sigma$ do
 2.      for each $b \in \Sigma$ do
 3.          $jbc(a,b) \leftarrow i + 1 + j$
 4.  for each $a \in \Sigma$ do
 5.      for $k \leftarrow 0$ to $j - 1$ do
 6.          $jbc(a, p[k]) \leftarrow i + 1 + j - 1 - k$
 7.  for $k \leftarrow 0$ to $i + 1 - j - 1$ do
 8.          $jbc(p[k], p[k + len]) \leftarrow i + 1 - 1 - k$
 9.  for $k \leftarrow i + 1 - j$ to $m - 1$ do
10.      for each $a \in \Sigma$ do
11.          $jbc(p[k], a) \leftarrow i + 1 - 1 - j$
```

```
FindJumpDistance($p, m, i, \Sigma, f, \beta$)
 1.  for each $c \in \Sigma$ do $v[c] \leftarrow 1$
 2.  $frq \leftarrow j \leftarrow 1$
 3.  while ($frq \geq \beta$ and $j \leq i + 1$) do
 4.      if ($v[p[i + 1 - j]] = 1$) then
 5.          $v[p[i + 1 - j]] = 0$
 6.          $frq \leftarrow frq - f(p[i + 1 - j])$
 7.      $j \leftarrow j + 1$
 8.  return $j - 1$
```

```
JumpingOccurrenceMatcher($p, m, t, n$)
 1.  $i \leftarrow$ FindWorstOccurrence($p, m, \Sigma, f$)
 2.  $j \leftarrow$ FindJumpDistance($p, m, i, \Sigma, f, 0.9$)
 3.  $jbc \leftarrow$ PrecomputeJOH($p, m, i, j$)
 4.  $s \leftarrow 0$
 5.  while ($s \leq n - m$) do
 6.      $k \leftarrow 0$
 7.      while ($k < m$ and $p[k] = t[s + k]$) do  $k \leftarrow k + 1$
 8.      if ($k = m$) then Output($s$)
 9.      $s \leftarrow s + jbc(t[s + i], t[s + i + j])$
```

Figure 3. The procedure PrecomputeJOH (for computing the table implementing the jumping-occurrence heuristic), the procedure FindJumpDistance (for computing the jump relative distance for a pattern p and a relative frequency function f), and the algorithm JumpingOccurrence-Matcher.

6.3 Approximating the Optimal Jump Distance

If one knows in advance the character distribution of a given text, procedure Find-JumpDistance in Figure 3 provides an efficient way for computing the optimal jump distance. Otherwise, one can adopt any of the three different approaches outlined in Section 5.3 for computing an approximated character distribution, and then, based on this, calculate the corresponding optimal occurrence relative position and jump distance. A somewhat simplified approach, still based on the strategy (*ii*) presented in Section 5.3, which bypasses the call to procedure FindJumpDistance, can be summarized in the following steps:

- initialize to 0 an array *scnt* (shifts counter) of length m;
- compute the worst-occurrence heuristic and run the first γ iterations of the algorithm by using such a rule for shifting; in the meantime, count the shifts of length ℓ occurring in this phase, for each length $\ell = 1, \ldots, m$, by updating accordingly the entries of the array *scnt*;
- compute an approximation of the value j^*_β by putting

$$\widetilde{j^*_\beta} =_{\text{Def}} \min \left\{ j \;\middle|\; \frac{1}{\gamma} \sum_{i=1}^{j} scnt[i] \geq \beta \right\};$$

- compute the jumping-occurrence heuristic, based on the value $\widetilde{j^*_\beta}$, and resume the search from the last shift position which has been checked, using such a rule for shifting.

It turns out that a good approximation of the optimal jump distance can be obtained even with small values of the parameter γ.

7 Experimental Results

We evaluated experimentally the impact of our proposed variants of the occurrence heuristics (in combination with their corresponding matchers):

– Improved-Occurrence Matcher (in short, IOM), described in Section 4,
– Worst-Occurrence Matcher (in short, WOM), described in Section 5.2,
– Jumping-Occurrence Matcher (in short, JOM), described in Section 6.2,

by testing them against the following algorithms based on the best known implementations of the occurrence heuristic:[2]

– Horspool algorithm (in short, HOR), which uses a single character occurrence heuristic and whose advancements are computed by $gbc_p(m-1, t[s+m-1])$;
– Quick Search algorithm (in short, QS), which uses a single character occurrence heuristic and whose advancements are computed by $gbc_p(m, t[s+m])$;
– Smith algorithm (SMITH), which uses a single character heuristic, whose advancements are computed by $\max(gbc_p(m, t[s+m]), gbc_p(m-1, t[s+m-1]))$;
– Berry-Ravindran algorithm (in short, BR), which uses two characters for shifting and whose advancements are computed by $gbc^2_p(m, 1, t[s+m], t[s+m+1])$;
– Zhu-Takaoka algorithm (in short, ZT), which uses two characters for shifting and whose advancements are computed by $gbc^2_p(m, 1, t[s+m-2], t[s+m-1])$.

Our implementation of the WOM algorithm computes the frequency of characters in the searched text by using the strategy (i) described in Section 5.3, with the parameter $\gamma = 100$, whereas our implementation of the JOM algorithm is based on the approach described in Section 6.3, with the same parameter $\gamma = 100$.

All algorithms have been implemented in the C programming language and have been compiled with the GNU C Compiler, using the optimization options -O3. All experiments have been executed locally on a MacBook Pro with 4 Cores, a 2 GHz Intel Core i7 processor, 4 GB RAM 1333 MHz DDR3, 256 KB of L2 Cache, and 6 MB of Cache L3. They have been evaluated in terms of the average shift advancements and running times, including any preprocessing time, measured with a hardware cycle counter available on modern CPUs. The tests have been run on text buffers over small and large alphabets. However we report in this paper only experimental results relative to small alphabets, since the gain in running time obtained when searching texts over large alphabets is negligible. In particular, we report experimental evaluations on a random sequence over an alphabet of 2 characters, a genome sequence, and a protein sequence, all sequences of 4MB. All sequences, provided by the SMART research tool,[3] are available online for download. Patterns of length m were randomly extracted from the sequences, with m ranging over the set of values $\{2^i \mid 1 \leq i \leq 12\}$. For each case, the mean over the running times, expressed in hundredths of seconds, of 500 runs has been reported. Figure 4 shows the running times of the Jumping-Occurrence Matcher with different values of the parameter β, whereas Figure 5 reports the running times of the algorithms HOR, QS, SMITH, BR, ZT, and the matchers IOM, WOM, and JOM, implementing our new proposed occurrence heuristics. The running times in Figure 5 of the JOM algorithm refer to an implementation with the parameter $\beta = 0.9$.

[2] For each algorithm we indicate the corresponding function used for shifting when the pattern of length m is aligned with the text at a given shift s.

[3] The SMART tool is available online at `http://www.dmi.unict.it/~faro/smart/`

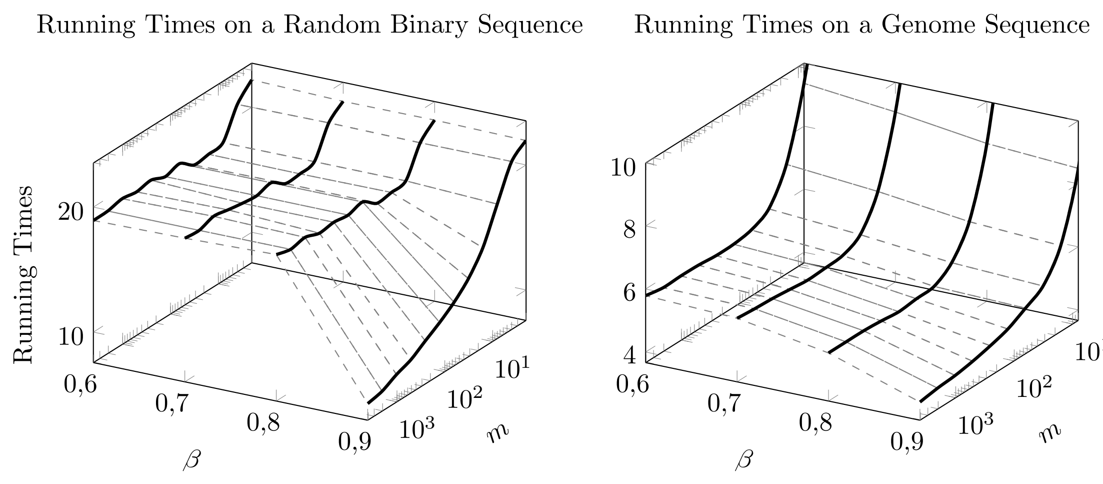


Figure 4. Running times of the Jumping-Occurrence Matcher for different values of the parameter β and pattern length m.

Running Times Evaluation

The experimental results in Figure 4 show that the choice of $\beta = 0.9$ is the best one for the jumping-occurrence heuristic in most cases. The gain in performance is more evident in the case of small alphabets or in the case of long patterns. In this latter case, the JOM algorithm with $\beta = 0.9$ is up to 50% faster. It is to be noticed, though, that in the case of large alphabets the improvement in running times is negligible.

From the experimental data in Figure 5, it follows that our proposed occurrence heuristics obtain always the best results. In particular the JOM algorithm is always the best choice for large alphabets. However, its speed-up is almost negligible in the case of large alphabets and long patterns, whereas it becomes more evident for very small alphabets, exhibiting a speed-up of more than 50% with respect to the best known algorithms. The IOM algorithm shows a very good behavior for short patterns. In fact, it turns out that it is the best solution in the case of short patterns and small alphabets, where it is more than 20% faster than other algorithms based on single character heuristics. However, its performance degrades as the length of the pattern increases. The WOM algorithm turns out to the best algorithm when the pattern is not short. Among the algorithms based on a single character occurrence heuristic, it shows an extremely fast behavior and for long patterns it is up to 50% faster than previous existing solutions. It is to be noticed that its running times are very close to those obtained by the JOM algorithm, which, however, is based on a two-characters heuristic.

Stability Evaluation

It is also useful to find out how accurately repeatable the results are. If only average running times are considered, some important details may be hidden. The SMART tool computes the stability of an algorithm as the standard deviation of the running times of the tests. The standard deviation measures the amplitude of the variation from the average, i.e., the mean of the running times. A low standard deviation indicates that the running times tend to be very close to the mean, underlying a high stability of the algorithm. On the other hand, a high standard deviation indicates that the running times are spread out over a large range of values, thus indicating a low stability.

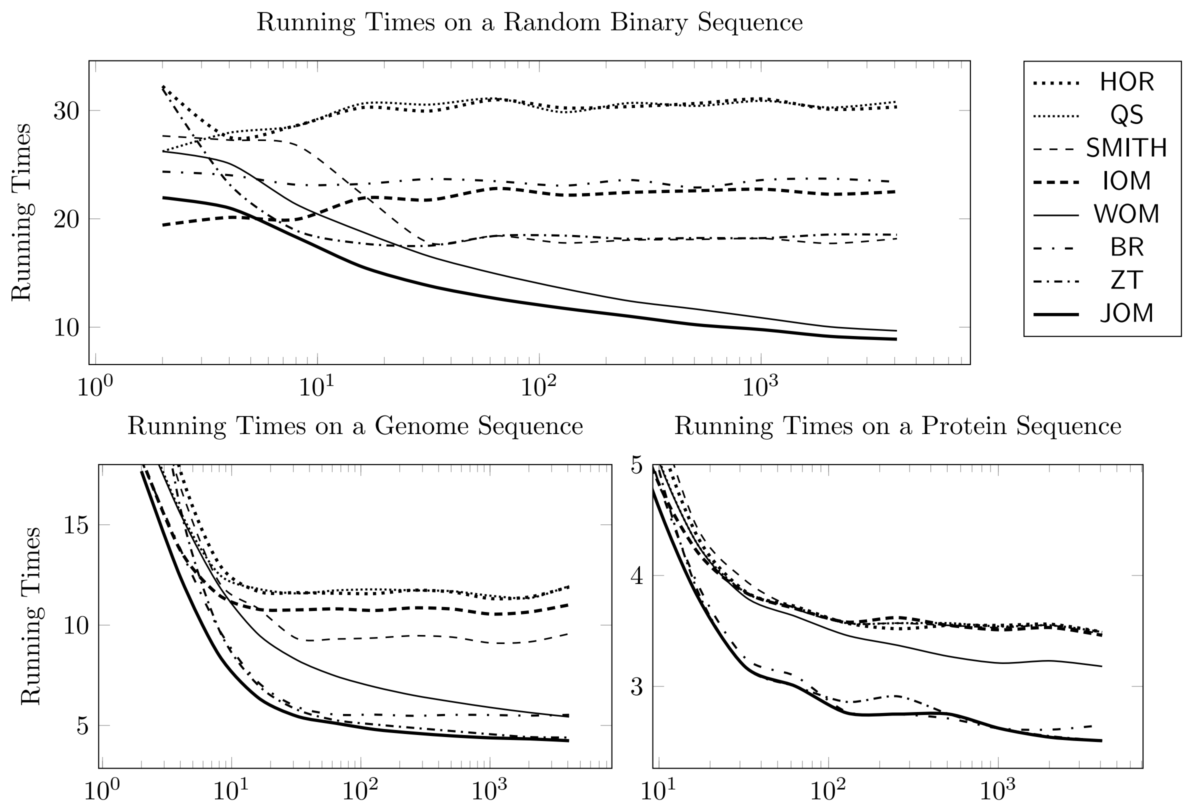


Figure 5. Running times obtained by comparing several efficient algorithms based on the occurrence heuristic shifting strategy. The x-axis represent length of patterns.

Figure 6 reports the standard deviation of the running times observed in our tests. It turns out that the WOM and the JOM heuristics are sensibly more stable than the remaining algorithms, especially in the case of long patterns and small alphabets.

While standard algorithms based on the one-character occurrence heuristic (as, for instance, HOR, QS and SMITH) become less stable as the length of the pattern increases, in some cases the algorithms based on our proposed occurrence heuristics show an opposite behavior, i.e., they become more stable as the length of the pattern increases. In particular, the IOM algorithm turns out to be the more stable algorithm in the case of short patterns, but it becomes less stable for long patterns. The converse behavior can be noticed in the case of the WOM algorithm, though we notice that the improvement in stability becomes negligible in the case of large alphabets.

Flexibility Evaluation
Flexibility is an important attribute of various types of systems. In the field of string matching, it refers to algorithms that can adapt when changes in the input data occur. Thus a string matching algorithm can be considered flexible when, for instance, it maintains good performance for both short and long patterns, or in the case of both small and large alphabets. By analyzing the running times reported in Figure 5, it turns out that the JOM algorithm is the more flexible one among the algorithms which have been tested, as it shows very good performance for all the lengths of the patterns and different sizes of the alphabet. The IOM algorithm turns out to be very efficient only for short patterns (and in some cases it is even more efficient than the JOM algorithm), but its performance degrades as the length of the pattern increases.

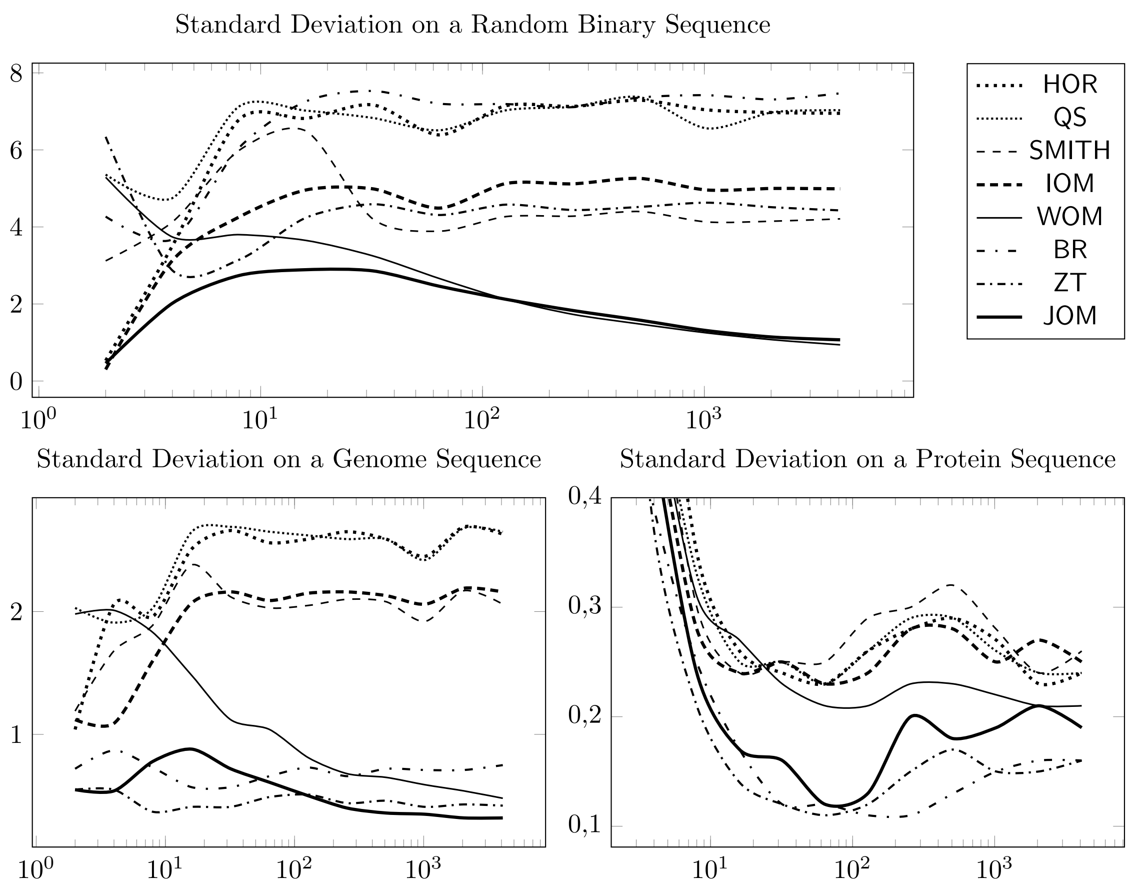


Figure 6. Standard Deviation of running times obtained by comparing several efficient algorithms based on the occurrence heuristic shifting strategy. The x-axis represent length of patterns.

An opposite observation can be done for the WOM algorithm, which maintains good performance only for medium and long patterns.

8 Conclusions

In this paper we have presented three new variations of the occurrence heuristic based on a smart computation of the relative position of the character used for computing the shift advancement. The proposed variations yield the largest average advancement, according to the characters distribution in the text. We have also shown experimental evidence that the new variants of the occurrence heuristics achieve very good results in practice, especially in the case of long patterns or small alphabets. We plan to conduct a probabilistic and a combinatorial analysis of the new proposed rules directed at giving theoretical support to the experimental evidence reported in the present work.

References

1. T. Berry and S. Ravindran: *A fast string matching algorithm and experimental results*, in Proceedings of the Prague Stringology Conference ’99, J. Holub and M. Šimánek, eds., Czech Technical University, Prague, Czech Republic, 1999, pp. 16–28, Collaborative Report DC–99–05.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) 1977, pp. 762–772.
3. D. Cantone and S. Faro: *Fast-search algorithms: New efficient variants of the Boyer-Moore pattern-matching algorithm.* Journal of Automata, Languages and Combinatorics, 10(5/6) 2005, pp. 589–608.
4. D. Cantone, S. Faro, and E. Giaquinta: *A compact representation of nondeterministic (suffix) automata for the bit-parallel approach*, in Combinatorial Pattern Matching, 21st Annual Symposium, CPM 2010, vol. 6129 of Lecture Notes in Computer Science, Springer, 2010, pp. 288–298.
5. D. Cantone, S. Faro, and E. Giaquinta: *A compact representation of nondeterministic (suffix) automata for the bit-parallel approach.* Inf. Comput., 213 2012, pp. 3–12.
6. D. Cantone, S. Faro, and E. Giaquinta: *On the bit-parallel simulation of the nondeterministic aho-corasick and suffix automata for a set of patterns.* J. Discrete Algorithms, 11 2012, pp. 25–36.
7. D. Cantone and S. Faro: *On tuning the bad-character rule: the worst-character rule*, Tech. Rep. arXiv:1012.1338v1, CoRR at arXiv.org - Cornell University Library, December 2010, Available at http://arxiv.org/abs/1012.1338.
8. L. Cleophas, B. Watson, and G. Zwaan: *A new taxonomy of sublinear right-to-left scanning keyword pattern matching algorithms.* Sci. Comput. Program., 75(11) Nov. 2010, pp. 1095–1112.
9. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) 2013, p. 13.
10. S. Faro and M. O. Külekci: *Fast multiple string matching using streaming SIMD extensions technology*, in String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, vol. 7608 of Lecture Notes in Computer Science, Springer, 2012, pp. 217–228.
11. S. Faro and M. O. Külekci: *Fast packed string matching for short patterns*, in Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, SIAM, 2013, pp. 113–121.
12. S. Faro, and E. Pappalardo: *Ant-CSP: An Ant Colony Optimization Algorithm for the Closest String Problem*, in SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference on Current Trends in Theory and Practice of Computer Science, vol. 5901 of Lecture Notes in Computer Science, Springer, 2010, pp. 360–281.
13. R. N. Horspool: *Practical fast searching in strings.* Softw. Pract. Exp., 10(6) 1980, pp. 501–506.
14. A. Hume and D. M. Sunday: *Fast string searching.* Softw. Pract. Exp., 21(11) 1991, pp. 1221–1248.
15. M. E. Nebel: *Fast string matching by using probabilities: on an optimal mismatch variant of Horspool’s algorithm.* Theor. Comput. Sci., 359(1) 2006, pp. 329–343.
16. D. M. Sunday: *A very fast substring search algorithm.* Commun. ACM, 33(8) 1990, pp. 132–142.
17. B. Watson: *Taxonomies and toolkits of regular language algorithms*, Phd. Thesis., Faculty of Computing Science, Eindhoven University of Technology, 1995.
18. A. C. Yao: *The complexity of pattern matching for a random string.* SIAM J. Comput., 8(3) 1979, pp. 368–387.
19. R. F. Zhu and T. Takaoka: *On improving the average case of the Boyer-Moore string matching algorithm.* J. Inf. Process., 10(3) 1988, pp. 173–177.

Computing Reversed Lempel-Ziv Factorization Online

Shiho Sugimoto[1], Tomohiro I[1,2], Shunsuke Inenaga[1], Hideo Bannai[1], and Masayuki Takeda[1]

[1] Department of Informatics, Kyushu University, Japan
{`shiho.sugimoto,tomohiro.i,inenaga,bannai,takeda`}`@inf.kyushu-u.ac.jp`
[2] Japan Society for the Promotion of Science (JSPS)

Abstract. Kolpakov and Kucherov proposed a variant of the Lempel-Ziv factorization, called the reversed Lempel-Ziv (RLZ) factorization (Theoretical Computer Science, 410(51):5365–5373, 2009). In this paper, we present an on-line algorithm that computes the RLZ factorization of a given string w of length n in $O(n \log^2 n)$ time using $O(n \log \sigma)$ bits of space, where $\sigma \le n$ is the alphabet size. Also, we introduce a new variant of the RLZ factorization with self-references, and present two on-line algorithms to compute this variant, in $O(n \log \sigma)$ time using $O(n \log n)$ bits of space, and in $O(n \log^2 n)$ time using $O(n \log \sigma)$ bits of space.

Keywords: reversed Lempel-Ziv factorization, on-line algorithms, suffix trees, palindromes

1 Introduction

The Lempel-Ziv (LZ) factorization of a string [21] is an important tool of data compression, and is a basis of efficient string processing algorithms [9,4] and compressed full text indices [11]. In the off-line setting where the string is static, there exist efficient algorithms to compute the LZ factorization of a given string w of length n, running in $O(n)$ time and using $O(n \log n)$ bits of space, assuming an integer alphabet. See [1] for a survey, and [8,5,7,6] for more recent results in this line of research. In the on-line setting where new characters may be appended to the end of the string, Okanohara and Sadakane [16] gave an algorithm that runs in $O(n \log^3 n)$ time using $n \log \sigma + o(n \log \sigma) + O(n)$ bits of space, where σ is the size of the alphabet. Later, Starikovskaya [18] proposed an algorithm running in $O(n \log^2 n)$ time using $O(n \log \sigma)$ bits of space, assuming $\frac{\log_\sigma N}{4}$ characters are packed in a machine word. Very recently, Yamamoto et al. [20] developed a new on-line LZ factorization algorithm running in $O(n \log n)$ time using $O(n \log \sigma)$ bits of space.

In this paper, we consider the *reversed* Lempel-Ziv factorization (RLZ in short[1]) proposed by Kolpakov and Kucherov [10], which is used as a basis of computing gapped palindromes. In the on-line setting, the RLZ factorization can be computed in $O(n \log \sigma)$ time using $O(n \log n)$ bits of space, utilizing the algorithm by Blumer et al. [3]. We present a more space-efficient solution to the same problem, which requires only $O(n \log \sigma)$ bits of working space with slightly slower $O(n \log^2 n)$ running time.

We also introduce a new, self-referencing variant of the RLZ factorization, and propose two on-line algorithms; the first one runs in $O(n \log \sigma)$ time and $O(n \log n)$ bits of space, and the second one in $O(n \log^2 n)$ time and $O(n \log \sigma)$ bits of space. A

[1] Not to be confused with the *relative* Lempel-Ziv factorization proposed in [12].

key to achieve such complexity is efficient on-line computation of the longest suffix palindrome for each prefix of the string w.

As an independent interest, we consider the relationship between the number of factors in the RLZ factorization of a string w, and the size of the smallest grammar that generates only w. It is known that the number of factors in the LZ factorization of w is a lower bound of the smallest grammar for w [17]. We show that, unfortunately, this is not the case with the RLZ factorization with or without self-references.

2 Preliminaries

2.1 Strings and model of computation

Let Σ be the alphabet of size σ. An element of Σ^* is called a string. For string $w = xyz$, x is called a prefix, y is called a substring, and z is called a suffix of w, respectively. The sets of substrings and suffixes of w are denoted by $\mathit{Substr}(w)$ and $\mathit{Suffix}(w)$, respectively. The length of string w is denoted by $|w|$. The empty string ε is a string of length 0, that is, $|\varepsilon| = 0$. For $1 \le i \le |w|$, $w[i]$ denotes the i-th character of w. For $1 \le i \le j \le |w|$, $w[i..j]$ denotes the substring of w that begins at position i and ends at position j. Let w^{rev} denote the reversed string of s, that is, $w^{\text{rev}} = w[|w|] \cdots w[2]w[1]$. For any $1 \le i \le j \le |w|$, note $w[i..j]^{\text{rev}} = w[j]w[j-1] \cdots w[i]$.

A string x is called a palindrome if $x = x^{\text{rev}}$. The *center* of a palindromic substring $w[i..j]$ of a string w is $\frac{i+j}{2}$. A palindromic substring $w[i..j]$ is called the *maximal palindrome* at the center $\frac{i+j}{2}$ if no other palindromes at the center $\frac{i+j}{2}$ have a larger radius than $w[i..j]$, i.e., if $w[i-1] \ne w[j+1]$, $i = 1$, or $j = |w|$. In particular, a maximal palindrome $w[i..|w|]$ is called a *suffix palindrome* of w.

The default base of logarithms will be 2. Our model of computation is the unit cost word RAM with the machine word size at least $\lceil \log n \rceil$ bits. We will evaluate the space complexities in bits (not in words). For an input string w of length n over an alphabet of size $\sigma \le n$, let $r = \frac{\log_\sigma n}{4} = \frac{\log n}{4 \log \sigma}$. For simplicity, assume that $\log n$ is divisible by $4 \log \sigma$, and that n is divisible by r. A string of length r, called a *meta-character*, fits in a single machine word. Thus, a meta-character can also be transparently regarded as an element in the integer alphabet $\Sigma^r = \{1, \ldots, n\}$. We assume that given $1 \le i \le n - r + 1$, any meta-character $A = w[i..i + r - 1]$ can be retrieved in constant time. We call a string on the alphabet Σ^r of meta-characters, a *meta-string*. Any string w whose length is divisible by r can be viewed as a meta-string w of length $m = \frac{n}{r}$. We write $\langle w \rangle$ when we explicitly view string w as a meta-string, where $\langle w \rangle[j] = w[(j-1)r + 1..jr]$ for each $j \in [1, m]$. Such range $[(j-1)r + 1, jr]$ of positions will be called *meta-blocks* and the beginning positions $(j-1)r + 1$ of meta-blocks will be called *block borders*. For clarity, the length m of a meta-string $\langle w \rangle$ will be denoted by $\|\langle w \rangle\|$. Note that $m \log n = n \log \sigma$.

2.2 Suffix Trees and Generalized Suffix Tries

The suffix tree [19] of string s, denoted $\mathit{STree}(s)$, is a rooted tree such that

1. Each edge is labeled with a non-empty substring of s, and each path from the root to a node spells out a substring of s;
2. Each internal node v has at least two children, and the labels of distinct out-going edges of v begin with distinct characters;

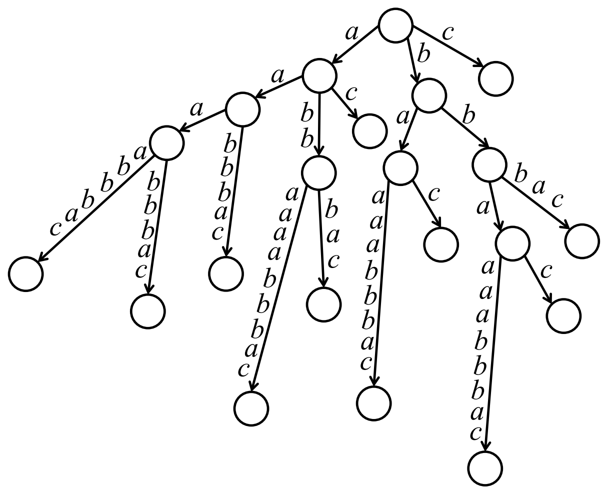


Figure 1. $STree(w)$ with $w = abbaaaabbbac$.

3. For each suffix x of w, there is a path from the root that spells out x.

The number of nodes and edges of $STree(s)$ is $O(|s|)$, and $STree(s)$ can be represented using $O(|s| \log |s|)$ bits of space, by implementing each edge label y as a pair (i, j) such that $y = s[i..j]$.

For a constant alphabet, Weiner's algorithm [19] constructs $STree(s^{\text{rev}})$ in an on-line manner from left to right, i.e., constructs $STree(s[1..j]^{\text{rev}})$ in increasing order of $j = 1, 2, \ldots, |s|$, in $O(|s|)$ time using $O(|s| \log |s|)$ bits of space. It is known that the tree of the suffix links of the directed acyclic word graph [3] of s forms $STree(s^{\text{rev}})$. Hence, for larger alphabets, we have the following:

Lemma 1 ([3]). *Given a string s, we can compute $STree(s^{\text{rev}})$ on-line from left to right, in $O(|s| \log \sigma)$ time using $O(|s| \log |s|)$ bits of space.*

In our algorithms, we will also use the generalized suffix *trie* for a set W of strings, denoted $STrie(W)$. $STrie(W)$ is a rooted tree such that

1. Each edge is labeled with a character, and each path from the root to a node spells out a substring of some string $w \in W$;
2. The labels of distinct out-going edges of each node must be different;
3. For each suffix s of each string $w \in W$, there is a path from the root that spells out s.

2.3 Reversed LZ factorization

Kolpakov and Kucherov [10] introduced the following variant of LZ77 factorization.

Definition 2 (Reversed LZ factorization without self-references). *The reversed LZ factorization of string w without self-references, denoted $RLZ(w)$, is a sequence $(f_1, f_2, \ldots, f_m)$ of non-empty substrings of w such that*

1. $w = f_1 \cdot f_2 \cdots f_m$, and
2. For any $1 \leq i \leq m$, $f_i = w[k..k + \ell_{\max} - 1]$, where $k = |f_1 \cdots f_{i-1}| + 1$ and $\ell_{\max} = \max(\{\ell \mid 1 \leq \exists t < k - \ell + 1, (w[t..t + \ell - 1])^{\text{rev}} = w[k..k + \ell - 1]\} \cup \{1\})$.

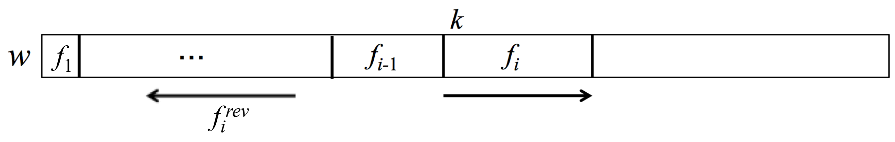


Figure 2. Let $k = |f_1 \cdots f_{i-1}| + 1$. f_i is the longest non-empty prefix of $w[k..n]$ that is also a substring of $(w[1..k-1])^{\text{rev}}$ if such exists.

Assume we have $f_1, \ldots, f_{i-1}$, and let $k = |f_1 \cdots f_{i-1}| + 1$. The above definition implies that f_i is the longest non-empty prefix of $w[k..n]$ that is also a substring of $(w[1..k-1])^{\text{rev}}$ if such exists, and $f_i = w[k]$ otherwise. See also Figure 2.

Example 3. For string $w = abbaaaabbbac$, $RLZ(w)$ consists of the following factors: $f_1 = a$, $f_2 = b$, $f_3 = ba$, $f_4 = a$, $f_5 = aabb$, $f_6 = ba$, and $f_7 = c$.

We are interested in on-line computation of $RLZ(w)$. Using Lemma 1, one can compute $RLZ(w)$ on-line in $O(n \log \sigma)$ time using $O(n \log n)$ bits of space [10], where $n = |w|$. The idea is as follows: Assume we have already computed the first j factors $f_1, f_2, \ldots, f_j$, and we have constructed $STree(w[1..l_j]^{\text{rev}})$, where $l_j = \sum_{h=1}^{j} |f_h|$. Now the next factor f_{j+1} is the longest prefix of $w[l_j + 1..n]$ that is represented by a path from the root of $STree(w[1..l_j]^{\text{rev}})$. After the computation of f_{j+1}, we update $STree(w[1..l_j]^{\text{rev}})$ to $STree(w[1..l_{j+1}]^{\text{rev}})$, using Lemma 1. In the next section, we will propose a new space-efficient on-line algorithm which requires $O(n \log^2 n)$ time using $O(n \log \sigma)$ bits of space.

We introduce yet another new variant, the reversed LZ factorization *with* self-references.

Definition 4 (Reversed LZ factorization with self-references). *The reversed LZ factorization of string w with self-references, denoted $RLZS(w)$, is a sequence $(g_1, g_2, \ldots, g_p)$ of non-empty substrings of w such that*

1. *$w = g_1 \cdot g_2 \cdots g_p$, and*
2. *For any $1 \leq i \leq p$, $g_i = w[k..k + \ell_{\max} - 1]$, where $k = |g_1 \cdots g_{i-1}| + 1$ and $\ell_{\max} = \max(\{\ell \mid 1 \leq \exists r < k, (w[r..r+\ell-1])^{\text{rev}} = w[k..k+\ell-1]\} \cup \{1\})$.*

Since r is at most $k-1$ in the above definition, g_i is the longest non-empty prefix of $w[k..n]$ that is also a substring of $(w[1..k + |g_i| - 2])^{\text{rev}}$ if such exists, and $g_i = w[k]$ otherwise. See also Figure 3.

Example 5. For string $w = abbaaaabbbac$, $RLZS(w)$ consists of the following factors: $g_1 = a$, $g_2 = b$, $g_3 = baaaabb$, $g_4 = ba$, and $g_5 = c$.

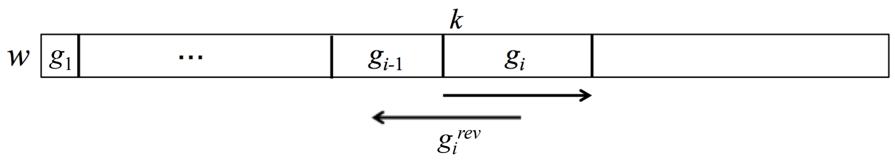


Figure 3. Let $k = |g_1 \cdots g_{i-1}| + 1$. g_i is the longest prefix of $w[k..n]$ that is also a substring of $(w[1..k + |g_i| - 2])^{\text{rev}}$ if such exists.

Note that in Definition 4 the ending position of a previous occurrence of g_i^{rev} does not have to be prior to the beginning position k of g_i, while in Definition 2 it has to, because of the constraints "$t < k - \ell + 1$". This is the difference between $RLZ(w)$ and $RLZS(w)$.

In this paper we propose two on-line algorithms to compute $RLZS(w)$; the first one runs in $O(n \log \sigma)$ time using $O(n \log n)$ bits of space, and the second one does in $O(n \log^2 n)$ time using $O(n \log \sigma)$ bits of space.

3 Computing $RLZ(w)$ in $O(n \log^2 n)$ time and $O(n \log \sigma)$ bits of space

The outline of our on-line algorithm to compute $RLZ(w)$ follows the algorithm of Starikovskaya [18] which computes Lempel-Ziv 77 factorization [21] in an on-line manner and in $O(n \log^2 n)$ time using $O(n \log \sigma)$ bits of space. The Starikovskaya algorithm maintains the suffix tree of the meta-string $\langle w \rangle$ in an on-line manner, i.e., maintains $STree(\langle w \rangle[1..k])$ in increasing order of $k = 1, 2, \ldots, n/r$, and maintains a generalized suffix trie for a set of substrings of $w[1..kr]$ of length $2r$ that begin at a block border. In contrast to the Starikovskaya algorithm, our algorithm maintains $STree((\langle w \rangle[1..k])^{\text{rev}})$ in increasing order of $k = 1, 2, \ldots, n/r$, and maintain a generalized suffix trie for a set of substrings of $w[1..kr]^{\text{rev}}$ of length $2r$ that begin at a block border.

Assume we have already computed the first $i-1$ factors $f_1, \ldots, f_{i-1}$ of $RLZ(w)$ and are computing the ith factor f_i. Let $l_i = \sum_{j=1}^{i-1} |f_j|$. This implies that we have processed $(\langle w \rangle[1..k])^{\text{rev}}$ where $k = \lceil l_i / r \rceil$, i.e., the kth meta block contains position l_i. As is the case with the Starikovskaya algorithm, our algorithm consists of two main phrases, depending on whether $|f_i| < r$ or $|f_i| \geq r$.

3.1 Algorithm for $|f_i| < r$

For any k $(1 \leq k \leq n/r)$, let W_k^{rev} denote the set of substrings of $w[1..kr]^{\text{rev}}$ of length $2r$ that begin at a block border, i.e., $W_k^{\text{rev}} = \{w[tr+1..(t+2)r]^{\text{rev}} \mid 1 \leq t \leq (k-2)\}$. We maintain $STrie(W_k^{\text{rev}})$ in an on-line manner, for $k = 1, 2, \ldots, n/r$. Note that $STrie(W_k^{\text{rev}})$ represents all substrings of $w[1..kr]^{\text{rev}}$ of length r which do not necessarily begin at a block border. Therefore, we can use $STrie(W_k^{\text{rev}})$ to determine if $|f_i| < r$, and if so, compute f_i. An example for $STrie(W_k^{\text{rev}})$ is shown in Figure 4.

A minor issue is that $STrie(W_k^{\text{rev}})$ may contain "unwanted" substrings that do not correspond to a previous occurrence of f_i^{rev} in $w[1..l_i]$, since substrings $w[(k-2)r+1..y]^{\text{rev}}$ for any $l_i < y \leq kr$ are represented by $STrie(W_k^{\text{rev}})$. In order to avoid finding such unwanted occurrences of f_i^{rev}, we associate to each node v representing a reversed substring x^{rev}, the leftmost ending position of x in $w[1..kr]$. Assume we have traversed the prefix of length $p \geq 0$ of $w[l_i+1..n]$ in the trie, and all the nodes involved in the traversal have positions smaller than $l_i + 1$. If either the node representing $w[l_i+1..l_i+p+1]$ stores a position larger than l_i or there is no node representing $w[l_i+1..l_i+p+1]$, then $f_i = w[l_i+1..l_i+p]$ if $p \geq 1$, and $f_i = w[l_i+1]$ if $p = 0$.

As is described above, f_i can be computed in $O(|f_i| \log \sigma)$ time. When $l_i + p > kr$, we insert the suffixes of a new substring $w[(k-1)r+1..(k+1)r]^{\text{rev}}$ of length $2r$ into the trie, and obtain the updated trie $STrie(W_{k+1}^{\text{rev}})$. Since there exist $\sigma^{2r} = \sigma^{\frac{\log n}{2}} = \sqrt{n}$

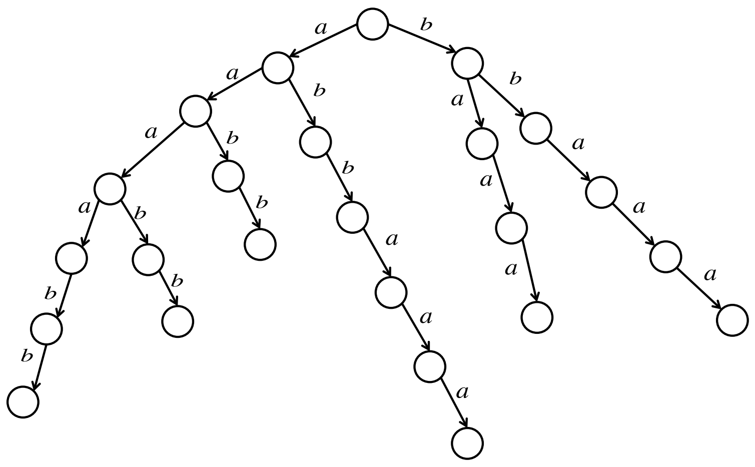


Figure 4. Let $r = 3$ and consider string $w = bba|aaa|bba|bac$, where | represents a block border. The figure shows $STrie(W_3^{\text{rev}})$ where $W_3^{\text{rev}} = \{aaaabb, abbaaa\}$.

distinct strings of length $2r$, the number of nodes in the trie is bounded by $O(\sqrt{n}r^2) = O(\sqrt{n}(\log_\sigma n)^2)$. Hence the trie requires $o(n)$ bits of space. Each update adds $O(r^2)$ new nodes and edges into the trie, taking $O(r^2 \log \sigma)$ time. Since there are n/r blocks, the total time complexity to maintain the trie is $O(nr \log \sigma) = O(n \log n)$.

The above discussion leads to the following lemma:

Lemma 6. *We can maintain in $O(n \log n)$ total time, a dynamic data structure occupying $o(n)$ bits of space that allows whether or not $|f_i| < r$ to be determined in $O(|f_i| \log \sigma)$ time, and if so, computes f_i and a previous occurrence of f_i^{rev} in $O(|f_i| \log \sigma)$ time.*

3.2 Algorithm for $|f_i| \geq r$

Assume we have found that the length of the longest prefix of $w[l_i + 1..n]$ that is represented by $STrie(W_k^{\text{rev}})$ is at least r, which implies that $|f_i| \geq r$.

For any string f and integer $0 \leq m \leq \min(|f|, r-1)$, let strings $\alpha_m(f)$, $\beta_m(f)$, $\gamma_m(f)$ satisfy $f = \alpha_m(f)\beta_m(f)\gamma_m(f)$, $|\alpha_m(f)| = m$, and $|\beta_m(f)| = j'r$ where $j' = \max\{j \geq 0 \mid m + jr \leq |f|\}$. We say that an occurrence of f in w has offset m ($0 \leq m \leq r-1$), if, in the occurrence, $\alpha_m(f)$ corresponds to a suffix of a meta-block, $\beta_m(f)$ corresponds to a sequence of meta-blocks (i.e. $\beta_m(f) \in Substr(\langle w \rangle)$), and $\gamma_m(f)$ corresponds to a prefix of a meta-block. Let f_i^m denote the longest prefix of $w[l_i+1..n]$ which has a previous occurrence in $w[1..l_i]$ with offset m. Thus, $|f_i| = \max_{0 \leq m < r} |f_i^m|$.

Our algorithm maintains two suffix trees on meta-strings, $STree((\langle w \rangle[1..k-1])^{\text{rev}})$ and $STree((\langle w \rangle[1..k])^{\text{rev}})$. Depending on the value of m, we use either $STree((\langle w \rangle[1..k-1])^{\text{rev}})$ and $STree((\langle w \rangle[1..k])^{\text{rev}})$.

If $l_i-(k-1)r \geq m$, i.e. the distance between the $(k-1)$th block border and position l_i is not less than m, then we use $STree((\langle w \rangle[1..k])^{\text{rev}})$ to find f_i^m. We associate to each internal node v of $STree((\langle w \rangle[1..k])^{\text{rev}})$ the lexicographical ranks of the leftmost and rightmost leaves in the subtree rooted at v, denoted $left(v)$ and $right(v)$, respectively. Recall that the leaves of $STree((\langle w \rangle[1..k])^{\text{rev}})$ correspond to the block borders $1, r+1, \ldots, (k-1)r+1$. Hence, $\alpha_m(f_i^m)\beta_m(f_i^m)$ occurs in $w[1..l_i]^{\text{rev}}$ iff there is a node v representing $\beta_m(f_i^m)$ and the interval $[left(v), right(v)]$ contains at least one block border b such that $w[b-m..b-1] = \alpha_m(f_i^m)$. To determine $\gamma_m(f_i^m)$, at each node v of $\mathit{STree}((\langle w\rangle[1..k])^{\mathrm{rev}})$ we maintain a trie T_v that stores the first meta-characters of the outgoing edge labels of v. Then, $\alpha_m(f_i^m)\beta_m(f_i^m)\gamma_m(f_i^m)$ occurs in $w[1..l_i]^{\mathrm{rev}}$ iff there is a node u of T_v representing $\gamma_m(f_i^m)$ and the interval $[\mathit{left}(u_1), \mathit{right}(u_2)]$ contains at least one block border b such that $w[b-m..b-1] = \alpha_m(f_i^m)$, where u_1 and u_2 are respectively the leftmost and rightmost children of u in T_v.

If $l_i - (k-1)r < m$, i.e. if the the distance between the $(k-1)$th block border and position l_i is less than m, then we use $\mathit{STree}((\langle w\rangle[1..k-1])^{\mathrm{rev}})$ to find f_i^m. This allows us to find only previous occurrences of f_i^{rev} that end before ℓ_i+1. All the other procedures follow the case where $l_i - (k-1)r \geq m$, mentioned above.

Lemma 7. *We can maintain in $O(n\log^2 n)$ total time, a dynamic data structure occupying $O(n\log\sigma)$ bits of space that allows to compute f_i with $|f_i| \geq r$ and a previous occurrence of f_i^{rev} in $O(|f_i|\log^2 n)$ time.*

Proof. Traversing the suffix tree for $\beta_m(f_i^m)$ takes $O(\frac{|f_i^m|}{r}\log n) = O(|f_i^m|\log\sigma)$ time since $\|\langle\beta_m(f_i^m)\rangle\| \leq |\frac{f_i^m}{r}|$. Also, traversing the trie for $\gamma_m(f_i^m)$ takes $O(r\log\sigma)$ time, since $|\gamma_m(f_i^m)| < r$. To assure $\beta_m(f_i^m)\gamma_m(f_i^m)$ is immediately preceded by $\alpha_m(f_i^m)$, we use the dynamic data structure proposed by Starikovskaya [18] which is based on the dynamic wavelet trees [13]. At each node v, the data structure allows us to check if the interval $[\mathit{left}(v), \mathit{right}(v)]$ contains a block border of interest in $O(\log^2 n)$ time, and to insert a new element to the data structure in $O(\log^2 n)$ time. Thus, f_i can be computed in $O(\sum_{0\leq m\leq r-1}(|f_i^m|\log\sigma + r\log\sigma + |\frac{f_i^m}{r}|\log^2 n)) = O(|f_i|\log^2 n)$. The position of a previous occurrence of f_i^{rev} can be retrieved in constant time, since each leaf of the suffix tree corresponds to a block border. Once f_i is computed, we update $\mathit{STree}((\langle w\rangle[1..k])^{\mathrm{rev}})$ to $\mathit{STree}((\langle w\rangle[1..k'])^{\mathrm{rev}})$, such that the k'th block border contains position l_{i+1} in w. Using Lemma 1, the suffix tree can be maintained in a total of $O(\frac{n}{r}\log\sigma) = O(n\log n)$ time.

It follows from Lemma 1 that the suffix tree on meta-strings requires $O(\frac{n}{r}\log n) = O(n\log\sigma)$ bits of space. Since the dynamic data structure of Starikovskaya [18] takes $O(n\log\sigma)$ bits of space, the total space complexity of our algorithm is $O(n\log\sigma)$ bits. □

The main result of this section follows from Lemma 6 and Lemma 7:

Theorem 8. *Given a string w of length n, we can compute $\mathit{RLZ}(w)$ in an on-line manner, in $O(n\log^2 n)$ time and $O(n\log\sigma)$ bits of space.*

4 On-line computation of reversed LZ factorization with self-references

In this section, we consider to compute $\mathit{RLZS}(w)$ for a given string w in an on-line manner. An interesting property of the reversed LZ factorization with self-references is that, the factorization can significantly change when a new character is appended to the end of the string. A concrete example is shown in Figure 5, which illustrates on-line computation of $\mathit{RLZS}(w)$ with $w = \mathit{abbaaaabbbac}$. Focus on the factorization of *abbaaaab*. Although there is a factor starting at position 5 in $\mathit{RLZS}(\mathit{abbaaaab})$, there is no factor starting at position 5 in $\mathit{RLZS}(\mathit{abbaaaabb})$. Below, we will characterize this with its close relationship to palindromes.

$\boxed{a\ b\ b\ a\ a\ a\ a\ b\ b\ b\ a\ c}$

$a|$
$a|b|$
$a|b|b|$
$a|b|b\ a|$
$a|b|b\ a|a|$
$a|b|b\ a|a\ a|$
$a|b|b\ a|a\ a\ a|$
$a|b|b\ a|a\ a\ a\ b|$
$a|b|b\ a\ a\ a\ a\ b\ b|$
$a|b|b\ a\ a\ a\ a\ b\ b|b|$
$a|b|b\ a\ a\ a\ a\ b\ b|b\ a|$
$a|b|b\ a\ a\ a\ a\ b\ b|b\ a|c|$

Figure 5. A snapshot of on-line computation of $RLZS(w)$ with $w = abbaaaabbbac$. For each non-empty prefix $w[1..k]$ of w, $|$ denotes the boundary of factors in $RLZS(w[1..k])$.

4.1 Computing $RLZS(w)$ in $O(n \log \sigma)$ time and $O(n \log n)$ bits of space

Let w be any string of length n. For any $1 \le j \le n$, the occurrence of substring p starting at position j is called self-referencing, if there exists j' such that $w[j'..j' + |p| - 1]^{\mathrm{rev}} = w[j..j + |p| - 1]$ and $j \le j' + |p| - 1 < j + |p| - 1$.

For any $1 \le k \le n$, let $Lpal_w(k) = \max\{k - j + 1 \mid w[j..k] = w[j..k]^{\mathrm{rev}}, 1 \le j \le k\}$. That is, $Lpal_w(k)$ is the length of the longest palindrome that ends at position k in w.

Lemma 9. *For any string w of length n and $1 \le k \le n$, let $RLZS(w[1..k-1]) = g_1, \dots, g_p$. Let $\ell_q = \sum_{h=1}^{q} |g_h|$ for any $1 \le q \le p$. Then*

$$RLZS(w[1..k]) = \begin{cases} g_1, \dots, g_p w[k] & \text{if } g_p w[k] \in Substr(w[1..\ell_{p-1}]^{\mathrm{rev}}) \text{ and } \ell_{p-1} + 1 \le d_k, \\ g_1, \dots, g_p, w[k] & \text{if } g_p w[k] \notin Substr(w[1..\ell_{p-1}]^{\mathrm{rev}}) \text{ and } \ell_{p-1} + 1 \le d_k, \\ g_1, \dots, g_j, w[\ell_j + 1..k] & \text{otherwise,} \end{cases}$$

where $d_k = k - Lpal_w(k) + 1$ and j is the minimum integer such that $\ell_j \ge d_k$.

Proof. By definition of $Lpal_w(k)$ and d_k, $w[d_k..k]$ is the longest suffix palindrome of $w[1..k]$. If $\ell_{p-1} + 1 \le d_k$, $w[\ell_{p-1} + 1..k]$ cannot be self-referencing. Hence the first and the second cases of the lemma follow. Consider the third case. Since $\ell_j \ge d_k$, $w[\ell_j + 1..k]$ is self-referencing. Since $RLZS(w[1..\ell_j]) = g_1, \dots, g_j$, the third case follows. □

See Figure 5 and focus on $RLZS(abbaaaab)$, where $g_1 = a$, $g_2 = b$, $g_3 = ba$, and $g_4 = aaab$. Consider to compute $RLZS(abbaaaabb)$. Since the longest suffix palindrome $bbaaaabb$ intersects the boundary between g_3 and g_4 of $RLZS(abbaaaab)$, the third case of Lemma 9 applies. Consequently, the new factorization $RLZS(abbaaaabb)$

consists of $g_1 = a$ and $g_2 = b$ of $RLZS(abbaaaab)$, and a new self-referencing factor $g_3 = baaaabb$.

Theorem 10. *Given a string w of length n, we can compute $RLZS(w)$ in an on-line manner, in $O(n \log \sigma)$ time and $O(n \log n)$ bits of space.*

Proof. Suppose we have already computed $RLZS(w[1..k-1])$, and we are computing $RLZS(w[1..k])$ for $1 \le k \le n$.

Assume $\ell_{p-1} + 1 \le d_k$. We check whether $g_p w[k] \in Substr(w[1..\ell_{p-1}]^{\mathrm{rev}})$ or not using $STree(w[1..\ell_{p-1}]^{\mathrm{rev}})$. If the first case of Lemma 9 applies, then we proceed to the next position $k+1$ and continue to traverse the suffix tree. If the second case of Lemma 9 applies, then we update the suffix tree for the reversed string, and proceed to computing $RLZS(w[1..k+1])$.

Assume $\ell_{p-1} + 1 > d_k$, i.e., the third case of Lemma 9 holds. For every $j < e \le p$, we remove g_e of $RLZS(w[1..k-1])$, and the last factor of $RLZS(w[1..k])$ is $w[\ell_j + 1..k]$. We then proceed to computing $RLZS(w[1..k+1])$.

As is mentioned in Section 2.3, in a total of $O(n \log \sigma)$ time and $O(n \log n)$ bits of space, we can check whether the first or the second case of Lemma 9 holds, as well as maintain the suffix tree for the reversed string on-line. In order to compute $Lpal_w(k)$ in an on-line manner, we can use Manacher's algorithm [14] which computes the maximal palindromes for all centers in w in $O(n)$ time and in an on-line manner. Since Manacher's algorithm actually maintains the center of the longest suffix palindrome of $w[1..k]$ when processing $w[1..k]$, we can easily modify the algorithm to also compute $Lpal_w(k)$ on-line. Since Manacher's algorithm needs to store the length of maximal palindromes for every center in w, it takes $O(n \log n)$ bits of space.

Finally, we show the total number of factors that are removed in the third case of Lemma 9. Once a factor that begins at position j is removed after computing $RLZS(w[1..k])$ for some k, for any $k \le k' \le n$, $RLZS(w[1..k'])$ never contains a factor starting at position j. Hence, the total number of factors that are removed in the third case is at most n. This completes the proof. □

4.2 Computing $RLZS(w)$ in $O(n \log^2 n)$ time and $O(n \log \sigma)$ bits of space

In this subsection, we present a space efficient algorithm that computes $RLZS(w)$ on-line, using only $O(n \log \sigma)$ bits of space. Note that we cannot use the method mentioned in the proof of Theorem 10, as it requires $O(n \log n)$ bits of space. Instead, we maintain a compact representation of all suffix palindromes of each prefix $w[1..k]$ of w, as follows.

For any string w of length $n \ge 1$, let $Spals(w)$ denote the set of the beginning positions of the palindromic suffixes of w, i.e.,

$$Spals(w) = \{n - |s| + 1 \mid s \in Suffix(w), s \text{ is a palindrome}\}.$$

Lemma 11 ([2,15]). *For any string w of length n, $Spals(w)$ can be represented by $O(\log n)$ arithmetic progressions.*

The above lemma implies that $Spals(w)$ can be represented by $O(\log^2 n)$ bits of space.

Lemma 12. *We can maintain $O(\log^2 n)$-bit representation of $Spals(w[1..k])$ on-line for every $1 \le k \le n$ in a total of $O(n \log n)$ time.*

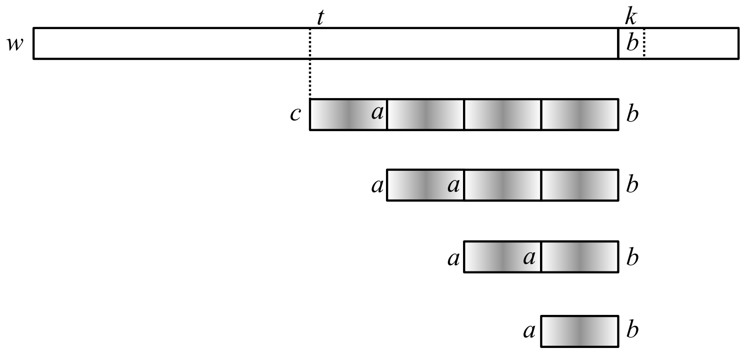


Figure 6. Illustration of Lemma 12. Let $w[t-1] = c$, $w[t+q-1] = a$, and $w[k] = b$. $w[t-1..k]$ is a suffix palindrome of $w[1..k]$ iff $c = b$, and $w[t+iq-1..k]$ is a suffix palindrome of $w[1..k]$ for any $1 \le i < m$ iff $a = b$.

Proof. We show how to efficiently update $\mathit{Spals}(w[1..k-1])$ to $\mathit{Spals}(w[1..k])$. Let S be any subset of $\mathit{Spals}(w[1..k-1])$ which is represented by a single arithmetic progression $\langle t, q, m\rangle$, where t is the first (minimum) element, q is the step, and m is the number of elements of the progression. Let s_j be the jth smallest element of S, with $1 \le j \le m$. By definition, s_j is a suffix palindrome of $w[1..k-1]$ for any j. In addition, if $m \ge 3$, then it appears that, for any $1 \le j < m$, s_j has a period q. Therefore, we can test whether the elements of S correspond to the suffix palindromes of $w[1..k]$, by two character comparisons: $w[t-1] = w[k]$ iff $t-1 \in \mathit{Spals}(w[1..k])$, and $w[t+q-1] = w[k]$ iff $t+iq-1 \notin \mathit{Spals}(w[1..k])$ for any $1 \le i < m$. (See also Figure 6.) If the extension of only one element of S becomes an element of $\mathit{Spals}(w[1..k])$, then we check if it can be merged to the adjacent arithmetic progression that contains closest smaller positions. As above, we can process each arithmetic progression in $O(1)$ time. By Lemma 11, there are $O(\log n)$ arithmetic progressions in $\mathit{Spals}(w[1..k])$ for each prefix of $w[1..k]$ of w. Consequently, for each $1 \le k \le n$ we can maintain $O(\log^2 n)$-bit representation of $\mathit{Spals}(w[1..k])$ in a total of $O(n \log n)$ time. □

The main result of this subsection follows:

Theorem 13. *Given a string w of length n, we can compute $\mathit{RLZS}(w)$ in an on-line manner, in $O(n \log^2 n)$ time and $O(n \log \sigma)$ bits of space.*

Proof. Assume that we are computing a new factor that begins at position ℓ of w. First, we use the algorithm of Theorem 8 and obtain the longest prefix f of $w[\ell..n]$ such that f^{rev} has an occurrence in $w[1..\ell-1]$. Then we apply Lemma 9 for $w[1..\ell + |f|-1]$, and if the third case holds, then we compute the self-reference factor. We use Lemma 12 to compute $\mathit{Lpal}_w(k)$ for any given position k. After computing the new factor, then we update the suffix tree of the meta-string, and proceed to computing the next factor. Overall, the algorithm takes $O(n \log^2 n)$ time and $O(n \log \sigma + \log^2 n) = O(n \log \sigma)$ bits of space. □

5 Reversed LZ factorization and smallest grammar

For any string w, the number of the LZ77 factors [21] (with/without self-references) of w is known to be a lower bound of the smallest grammar that derives only w [17].

Here we briefly show that this is not the case with the reversed LZ factorization (for either with or without self-references).

Theorem 14. *For $\sigma = 3$, there is an infinite series of strings for which the smallest grammar has size $O(\log n)$ while the size of the reversed LZ factorization is $O(n)$.*

Proof. Let $w = (abc)^{\frac{n}{3}}$. Then, $RLZ(w) = RLZS(w) = a, b, c, a, b, c, \ldots, a, b, c$, consisting of exactly n factors. On the other hand, it is easy to see that there exists a grammar of size $O(\log n)$ that generates only w. This completes the proof. □

The above theorem applies to any constant alphabet of size at least 3. When $\sigma = 1$, the size of the smallest grammar and the number of factors in $RLZ(w)$ are both $O(\log n)$, while the number of factors in $RLZS(w)$ is $O(1)$. The binary case where $\sigma = 2$ is open.

References

1. A. Al-Hafeedh, M. Crochemore, L. Ilie, J. Kopylov, W. Smyth, G. Tischler, and M. Yusufu: *A comparison of index-based Lempel-Ziv LZ77 factorization algorithms.* ACM Computing Surveys, 45(1) 2012, p. Article 5.
2. A. Apostolico, D. Breslauer, and Z. Galil: *Parallel detection of all palindromes in a string.* Theoretical Computer Science, 141(1&2) 1995, pp. 163–173.
3. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. I. Seiferas: *The smallest automaton recognizing the subwords of a text.* Theoretical Computer Science, 40 1985, pp. 31–55.
4. J.-P. Duval, R. Kolpakov, G. Kucherov, T. Lecroq, and A. Lefebvre: *Linear-time computation of local periods.* Theoretical Computer Science, 326(1-3) 2004, pp. 229–240.
5. K. Goto and H. Bannai: *Simpler and faster Lempel Ziv factorization*, in Proc. DCC 2013, 2013, pp. 133–142.
6. J. Kärkkäinen, D. Kempa, and S. J. Puglisi: *Lightweight Lempel-Ziv parsing*, in Proc. SEA 2013, 2013, pp. 139–150.
7. J. Kärkkäinen, D. Kempa, and S. J. Puglisi: *Linear time Lempel-Ziv factorization: Simple, fast, small*, in Proc. CPM 2013, 2013, pp. 189–200.
8. D. Kempa and S. J. Puglisi: *Lempel-Ziv factorization: Simple, fast, practical*, in Proc. ALENEX 2013, 2013, pp. 103–112.
9. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in Proc. FOCS 1999, 1999, pp. 596–604.
10. R. Kolpakov and G. Kucherov: *Searching for gapped palindromes.* Theoretical Computer Science, 410(51) 2009, pp. 5365–5373.
11. S. Kreft and G. Navarro: *Self-indexing based on LZ77*, in Proc. CPM 2011, 2011, pp. 41–54.
12. S. Kuruppu, S. J. Puglisi, and J. Zobel: *Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval*, in Proc. SPIRE 2010, 2010, pp. 201–206.
13. V. Mäkinen and G. Navarro: *Dynamic entropy-compressed sequences and full-text indexes.* ACM Transactions on Algorithms, 4(3) 2008.
14. G. K. Manacher: *A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string.* J. ACM, 22(3) 1975, pp. 346–351.
15. W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto: *Efficient algorithms to compute compressed longest common substrings and compressed palindromes.* Theoretical Computer Science, 410(8–10) 2009, pp. 900–913.
16. D. Okanohara and K. Sadakane: *An online algorithm for finding the longest previous factors*, in Proc. ESA 2008, 2008, pp. 696–707.
17. W. Rytter: *Application of Lempel-Ziv factorization to the approximation of grammar-based compression.* Theoretical Computer Science, 302(1-3) 2003, pp. 211–222.
18. T. A. Starikovskaya: *Computing Lempel-Ziv factorization online.*, in Proc. MFCS 2012, 2012, pp. 789–799.

19. P. Weiner: *Linear pattern-matching algorithms*, in Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory, 1973, pp. 1–11.
20. J. Yamamoto, T. I, H. Bannai, S. Inenaga, and M. Takeda: *Faster compact on-line Lempel-Ziv factorization.* CoRR, abs/1305.6095 2013.
21. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, IT-23(3) 1977, pp. 337–349.

Sorting Suffixes of a Text via its Lyndon Factorization

Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino

University of Palermo, Dipartimento di Matematica e Informatica, Italy
{sabrina,restivo,giovanna,mari}@math.unipa.it

Abstract. The process of sorting the suffixes of a text plays a fundamental role in Text Algorithms. They are used for instance in the constructions of the Burrows-Wheeler transform and the suffix array, widely used in several fields of Computer Science. For this reason, several recent researches have been devoted to finding new strategies to obtain effective methods for such a sorting. In this paper we introduce a new methodology in which an important role is played by the Lyndon factorization, so that the local suffixes inside factors detected by this factorization keep their mutual order when extended to the suffixes of the whole word. This property suggests a versatile technique that easily can be adapted to different implementative scenarios.

Keywords: sorting suffixes, BWT, suffix array, Lyndon words, Lyndon factorization

1 Introduction

The sorting of the suffixes of a text plays a fundamental role in Text Algorithms with several applications in many areas of Computer Science and Bioinformatics. For instance, it is a fundamental step, in implicit or explicit way, for the construction of the suffix array (SA) and the Burrows-Wheeler Transform (bwt). The SA, introduced in 1990 (cf. [19]), is a sorted array of all suffixes of a string, where the suffixes are identify by using their positions in the string. Several strategies that privilege the efficiency of the running time or the low memory consumption have been widely investigated (cf. [22,16]). The bwt, introduced in 1994 (cf. [6]), permutes the letters of a text according to the sorting of its cyclic rotations, making the text more compressible (cf. [2]). A recent survey on the combinatorial properties that guarantee such a compressibility after the application of bwt can be found in [25] (cf. also [23]). Moreover, in the last years the SA and the bwt, besides being important tools in Data Compression, have found many applications well beyond its original purpose (cf. [1,13,14,20,26,8,2]).

The goal of this paper is to introduce a new strategy for the sorting of the suffixes of a word that opens new scenarios of the computation of the SA and the bwt.

Our strategy uses a well known factorization of a word W called the *Lyndon factorization* and is based on a combinatorial property proved in this paper, that allows to sort the suffixes of W ("global suffixes") by using the sorting of the suffixes inside each block of the decomposition ("local suffixes").

The Lyndon factorization is based on the fact that any word W can be written uniquely as $W = L_1 L_2 \cdots L_k$, where

– the sequence $L_1, L_2, \ldots, L_k$ is non-increasing with respect to lexicographic order;
– each L_i is strictly less than any of its proper cyclic shift (Lyndon words).

This factorization was introduced in [7] and a linear time algorithm is due to Duval [11]. The intuition that the knowledge of Lyndon factorization of a text can be used for the computation of the suffix array of the text itself has been introduced in [5]. Conversely, a way to find the Lyndon factorization from the suffix array can be found in [17].

If U is a factor of a word W we say that the sorting of the local suffixes of U is *compatible* with the sorting of the global suffixes of W if the mutual order of two local suffixes in U is kept when they are extended as global suffixes. The main theorem in this paper states that if U is a concatenation of consecutive Lyndon factors, then the local suffixes in U are compatible with the global suffixes. This suggests some new algorithmic scenarios for the constructions of the SA and the *bwt*. In fact, by performing the Lyndon factorization of a word W by Duval's algorithm, one does not need to get to the end of the whole word in order to start the decomposition into Lyndon factors. Since our result allow to start the sorting of the local suffixes (compatible with the sorting of the global suffixes) as soon as the first Lyndon word is discovered, this may suggest an online algorithm, that do not require to read the entire word to start sorting. Moreover, the independence of the sorting of the local suffixes inside the different Lyndon factors of a text suggests also a possible parallel strategy to sort the global suffixes of the text itself.

In Section 2 we give the fundamental notions and results concerning combinatorics on words, the Lyndon factorization, the Burrows-Wheeler transform and the suffix array. In Section 3 we first introduce the notion of global suffix on a text and local suffix inside a factor of the text. Then we prove the compatibility between the ordering of local suffixes and the ordering of global suffixes. In Section 4 we describe an algorithm that uses the above result to incrementally construct the *bwt* of a text. Such a method can be also used to explicitly construct the SA of the text. In Section 5 we discuss about some possible improvements and developments of our method, including implementations in external memory or in place constructions. Finally, we compare our strategy for sorting suffixes with the method proposed in [12] in which a lightweight computation of the *bwt* of a text is performed by partitioning it into factors having the same length.

2 Preliminaries

Let $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ be a finite alphabet with $c_1 < c_2 < \cdots < c_\sigma$. Given a finite word $W = a_1 a_2 \cdots a_n$, $a_i \in \Sigma$ for $i = 1, \ldots, n$, a *factor* of W is written as $W[i,j] = a_i \cdots a_j$. A factor $W[1,j]$ is called a *prefix*, while a factor $W[i,n]$ is called a *suffix*. In this paper, we also denote by $suf_W(i)$ as the suffix of W starting from position i. We omit W when there is no danger of ambiguity. We say that $x, y \in \Sigma^*$ are *conjugate* (or *cyclic shift*) or *y is a conjugate of x* if $x = uv$ and $y = vu$ for some $u, v \in \Sigma^*$. Recall that conjugacy is an equivalent relation.

A *Lyndon* word is a primitive word which is also the minimum in its conjugacy class, with respect to the lexicographic order relation. In [18,11], one can find a linear algorithm that for any word $W \in \Sigma^*$ computes the Lyndon word of its conjugacy class. We call it the Lyndon word of W. Lyndon words are involved in a nice and important factorization property of words.

Theorem 1. *[7] Every word $W \in \Sigma^+$ has a unique factorization $W = L_1 L_2 \cdots L_k$ such that $L_1 \geq_{lex} \cdots \geq_{lex} L_k$ is a non-increasing sequence of Lyndon words.*

We call this factorization the *Lyndon factorization* of a word and it can be computed in linear time (see for instance [11,18]). Duval in [11] presents two variants of an algorithm of factorization of a word into Lyndon words in time linear in the length of the word. The first variant of the algorithm uses only three variables for a complete computation and it requires no more than $2n$ comparisons between two letters. The second one is slightly faster in that sense that it requires no more than $\frac{3n}{2}$ comparisons but it uses an auxiliary storage of size $\frac{n}{2}$. The basis idea for both these variants is finding each factor of the decomposition of the word W from left to right by eventually reading a long enough prefix of the next Lyndon factor.

Lyndon factorization has been realized also in parallel (cf. [3]) and in external memory (cf. [24]).

One way to define the Burrows-Wheeler Transform (*bwt*) [6] of a string W of length n (although not the most efficient way to compute it) is to construct all n cyclic shifts of W and sort them lexicographically. The output of *bwt* consists of the pair (L, I), where L is the sequence of the last character of each rotation in the sorted list and I is an integer denoting the position of the original word in the list.
Another more efficient way consists in the concatenating at the the input string W a symbol \$ that is smaller than any other letter. In this case, the *bwt* is intuitively described as follows: given a word $W \in \Sigma^*$, $bwt(W)$ is a word obtained by sorting the list of the suffixes of $W\$$ and by concatenating the symbols preceding in W each suffix in the sorted list. In both the cases, it is an invertible transform, i.e., one can recover the original text from its *bwt*.

Note that, in general, the sorting of the conjugates of a word W and the sorting of the suffixes of a word $W\$$ is different, but, as consequence of the properties of Lyndon words, when the word W is the Lyndon word, then the two sorting coincide (cf. [15, Lemma 12]). A study of the combinatorial aspects that connect these two sorting can be found in [5]. In this study an important role is played by the notion of Lyndon word.

Given a text W of length n, the suffix array (SA) for W is an array of integers of range 1 to $n+1$ specifying the lexicographic ordering of the suffixes of the string W. It will be convenient to assume that $W[n+1] = \$$, where \$ is smaller than any other letter. That is, $SA[j] = i$ if and only if $W[i, n+1]$ is the j-th suffix of W in ascending lexicographical order.

SA	bwt	$Suffixes$
12	s	\$
2	m	$a\ t\ h\ e\ m\ a\ t\ i\ c\ s\ \$$
7	m	$a\ t\ i\ c\ s\ \$$
10	i	$c\ s\ \$$
5	h	$e\ m\ a\ t\ i\ c\ s\ \$$
4	t	$h\ e\ m\ a\ t\ i\ c\ s\ \$$
9	t	$i\ c\ s\ \$$
1	\$	$m\ a\ t\ h\ e\ m\ a\ t\ i\ c\ s\ \$$
6	e	$m\ a\ t\ i\ c\ s\ \$$
10	c	$s\ \$$
2	a	$t\ h\ e\ m\ a\ t\ i\ c\ s\ \$$
8	a	$t\ i\ c\ s\ \$$

Figure 1. The table of the lexicographically sorted suffixes of the word *mathematics*\$ together the SA(*mathematics*\$) and the bwt(*mathematics*\$).

For instance, if $W = mathematics$ then $bwt(W\$) = smmihtt\$ecaa$ and $SA(W\$) = [12,\ 2,\ 7,\ 10,\ 5,\ 4,\ 9,\ 1,\ 6,\ 10,\ 2,\ 8]$. The table obtained by lexicographically sorting all the suffixes of $W\$$ is depicted in Figure 1.

3 Local and global suffixes of a text

Let $W \in \Sigma^*$ and let $W = L_1 L_2 \cdots L_k$ be its Lyndon Factorization. For each factor L_r, we denote by $first(L_r)$ and $last(L_r)$ the position of the first and the last character, respectively, of the factor L_r in W. Let u be a factor of W. We denote by $suf_u(i) = W[i, last(u)]$ and we call it *local suffix* at the position i with respect to u. Note that $suf_W(i) = W[i, n]$ and we call it *global suffix* of W at the position i. We write $suf(i)$ instead of $suf_W(i)$ when there is no danger of ambiguity.

Definition 2. *Let W be a word and let u be a factor of W. We say that the sorting of suffixes of u is* compatible *with the sorting of suffixes of W if for all i, j with $first(u) \le i < j \le last(u)$,*

$$suf_u(i) < suf_u(j) \iff suf(i) < suf(j).$$

Notice that in general taken an arbitrary factor of a word W, the sorting of its suffixes is not compatible with the sorting of the suffixes of W. Consider for instance the word $W = abababb$ and its factor $u = ababa$. Then $suf_u(1) = ababa > a = suf_u(5)$ whereas $suf(1) = abababb < abb = suf(5)$.

Theorem 3. *Let $W \in \Sigma^*$ and let $W = L_1 L_2 \cdots L_k$ be its Lyndon factorization. Let $u = L_r L_{r+1} \cdots L_s$. Then the sorting of the suffixes of u is compatible with the sorting of the suffixes of W.*

Proof. Let i and j be two indexes with $i < j$ both contained in u. We just need to prove that $suf(i) > suf(j) \iff suf_u(i) > suf_u(j)$. Let $x = W[j, last(L_s)]$ and $y = W[i, i + |x| - 1]$.

Suppose that $suf(i) > suf(j)$. Then $y \ge x$ by the definition of lexicographic order. If $y > x$ there is nothing to prove. If $x = y$, then $suf_u(j)$ is prefix of $suf_u(i)$, so by the definition of lexicographic order $suf_u(i) > suf_u(j)$.

Suppose now that $suf_u(i) > suf_u(j)$. This means that $y \ge x$. If $y > x$ there is nothing to prove. If $x = y$, the index $i + |x| - 1$ is in some Lyndon factor L_m with $r \le m \le s$, then $L_r \ge L_m \ge L_s$. We denote $z = W[i + |x|, last(L_m)]$. Then $suf(i) = xzL_{m+1} \cdots L_k > xL_{s+1} \cdots L_k = suf(j)$, since $z > L_m$ (because L_m is a Lyndon word) and $L_m \ge L_{s+1}$ (since the factorization is a sequence of non increasing factors). □

The above theorem states, in other words, that mutual order of the suffixes of W starting in two positions i and j is the same as the mutual order of the "local" suffixes starting in i and j inside the block obtained as concatenation of the consecutive Lyndon factors including i and j.

As particular case, the theorem is also true when the two suffixes start in the same Lyndon factor.

We recall that, if l_1 and l_2 denote two sorted lists of elements taken from any well ordered set, the operation $merge(l_1, l_2)$ consists in obtaining the sorted list of elements in l_1 and l_2

A consequence of previous theorem is stated in the following proposition.

Proposition 4. *Let $sort(L_1 L_2 \cdots L_l)$ and $sort(L_{l+1} L_{l+2} \cdots L_k)$ denote the sorted lists of the suffixes of $L_1 L_2 \cdots L_l$ and the suffixes $L_{l+1} L_{l+2} \cdots L_k$, respectively. Then $sort(L_1 L_2 \cdots L_k) = merge(sort(L_1 L_2 \cdots L_l), sort(L_{l+1} L_{l+2} \cdots L_k))$.*

This proposition suggests a possible strategy for sorting the list of the suffixes of some word W:

- find the Lyndon decomposition of W, $L_1 L_2 \cdots L_k$;
- find the sorted list of the suffixes of L_1 and, separately, the sorted list of the suffixes of L_2;
- merge the sorted lists in order to obtain the sorted lists of the suffixes of $L_1 L_2$;
- find the sorted list of the suffixes of L_3 and merge it to the previous sorted list;
- keep on this way until all the Lyndon factors are processed;

This kind of strategy could have several advantages: first of all, one can work online, i.e. one can start sorting suffixes as soon as the first Lyndon factor is individuated. This also allow to integrate the sorting process with the Duval's Algorithm for Lyndon decomposition that outputs Lyndon factors online as well.

The second advantage is that this kind of strategy allows parallelization, since every Lyndon factor can be processed separately for sorting its suffixes. These kind of application would require an efficient algorithm to perform the merging of two sorted lists.

A detailed algorithmic description of this method in order to obtain the *bwt* of a text is given in next section.

4 An incremental algorithm to sort suffixes of a text

In this section we propose an algorithm that incrementally constructs the suffix array SA and the Burrows-Wheeler transform *bwt* of the text W by using its Lyndon factorization. In particular, here we detail the construction of the *bwt* but an analogous reasoning can be done in order to obtain the suffix array. We assume that $L_1 L_2 \cdots L_k$ is the Lyndon factorization of the word $W[1, n]$. So $L_1 \geq L_2 \geq \cdots \geq L_k$. Such an hypothesis, although strong, is not restrictive because one can obtain the Lyndon factorization of any word in linear time (cf. [11,18]). As shown in previous section, the hypothesis that W is factorized in Lyndon words suggests to connect the problem to the sorting of the local suffixes of W to the lexicographic sorting of the global suffixes of W.

Our algorithm, called BWT_LYND, considers the input text $W[1, n]$ as logically partitioned into k blocks, where each block corresponds to a Lyndon word, and computes incrementally the $bwt(W\$)$ via k iterations, one per block of W. Each block is examined from right to left so that at iteration i we compute $bwt(L_1 \cdots L_i\$)$ given $bwt(L_1 \cdots L_{i-1}\$)$, $bwt(L_i\$)$ and $SA(L_i\$)$. Remark that the positions in $SA(L_i\$)$ range in $[first(L_i), Last(L_i) + 1]$. This means that we sum the amount $|L_1 \cdots L_{i-1}|$ to the values of the usual suffix array of $L_i\$$.

The key point of the algorithm comes from Theorem 3, because the construction of $bwt(L_1 \cdots L_i\$)$ from $bwt(L_1 \cdots L_{i-1}\$)$ requires only the insertion of the characters of L_i in $bwt(L_1 \cdots L_{i-1}\$)$ in the same mutual order as they appear in $bwt(L_i\$)$. Note that the character \$ that follows L_i is not considered in this operation.

Moreover, such an operation does not modify the mutual order of the characters already lying in $bwt(L_1 \cdots L_{i-1}\$)$.

For each block L_i with i ranging from 1 to k, the algorithm Bwt_Lynd executes the following steps:

1. Compute the $bwt(L_i\$)$ and $SA(L_i\$)$.
2. Compute the counter array $G[1, |L_i|+1]$ which stores in $G[j]$ the number of suffixes of the string $L_1 \cdots L_{i-1}\$$ which are lexicographically smaller than the j-th suffix of $L_i\$$.
3. Merge $bwt(L_1 \cdots L_{i-1}\$)$ and $bwt(L_i\$)$ in order to obtain $bwt(L_1 \cdots L_{i-1}L_i\$)$.

Example 5. Let $W = aabcabbaabaabdabbaaabbdc$. The Lyndon factorization of W is $L_1L_2L_3$, where $L_1 = aabcabb > L_2 = aabaabdabb > L_3 = aaabbdc$. Figure 2 illustrates how Step 3 of the algorithm works. Note that the positions of the suffixes in $L_2\$$ (i.e. in $SA(L_2\$)$) are shifted of $|L_1| = 7$ positions. Notice that in the algorithm Bwt_Lynd we do not actually compute the sorted list of suffixes, but we show it in Figure 2 to ease the comprehension of the algorithm. Moreover, the algorithm can be simply adapt to compute the suffix array of W, so in Figure 2 the suffix arrays are also shown.

$L_1\$$

	SA	bwt	Sorted Suffixes
	8	$\underline{b}$	$
	1	$	*aabcabb$*
	5	*c*	*abb$*
	2	*a*	*abcabb$*
	7	*b*	*b$*
	6	*a*	*bb$*
	3	*a*	*bcabb$*
	4	*b*	*cabb$*

$L_2\$$

G	SA	bwt	Sorted Suffixes
0	**11+7=18**	**b**	**$**
0	**1+7=8**	**$**	$\underline{\textbf{aabaabdabb\$}}$
2	**4+7=11**	**b**	**aabdabb$**
2	**2+7=9**	**a**	**abaabdabb$**
2	**8+7=15**	**d**	**abb$**
4	**5+7=12**	**a**	**abdabb$**
4	**10+7=17**	**b**	**b$**
5	**3+7=10**	**a**	**baabdabb$**
5	**9+7=16**	**a**	**bb$**
7	**6+7=13**	**a**	**bdabb$**
8	**7+7=14**	**b**	**dabb$**

$\Rightarrow$

$L_1L_2\$$

SA	bwt	Sorted Suffixes
18	**b**	**$**
8	$\underline{b}$	$\underline{aabaabdabb\$}$
1	$	*aabcabbaabaabdabb$*
11	**b**	**aabdabb$**
9	**a**	**abaabdabb$**
15	**d**	**abb$**
5	*c*	*abbaabaabdabb$*
2	*a*	*abcabbaabaabdabb$*
12	**a**	**abdabb$**
17	**b**	**b$**
7	*b*	*baabaabdabb$*
10	**a**	**baabdabb$**
16	**a**	**bb$**
6	*a*	*bbaabaabdabb$*
3	*a*	*bcabbaabaabdabb$*
13	**a**	**bdabb$**
4	*b*	*cabbaabaabdabb$*
14	**b**	**dabb$**

Figure 2. Iteration 2 of the computation of the bwt of the text $W = aabcabb|aabaabdabb|aaabbdc$ on the alphabet $\{a, b, c, d\}$. The two columns represent the bwts before and after the iteration. Note that the first row (the underlined letter) in the table relative to $L_1\$$ and the second row (the underlined suffix) in the table relative to $L_2\$$ flow into the second row in the table relative to $L_1L_2\$$. Indeed, the suffix $aabaabdabb\$$ is preceded by the symbol b in $L_1L_2\$$. We use distinct style fonts for each Lyndon word.

Step 1 can be executed in linear time $O(|L_i|)$, if $bwt(L_i\$)$ and $SA(L_i\$)$ are stored in internal memory (see [22,16]).

During Step 2, the algorithm uses the functions C and $rank$ described as follows. For any character $x \in \Sigma$, let $C(u, x)$ denote the number of characters in u that are smaller than x, and let $rank(u, x, t)$ denote the number of occurrences of x in $u[1, t]$. Such functions have been introduced in [13] for the FM-index. For sake of simplicity we can firstly construct the array $A[1, |L_i| + 1]$ which stores in $A[j]$ the number of suffixes of the string $L_1 \cdots L_{i-1}\$$ which are lexicographically smaller than the suffix of $L_i\$$ starting at the position j. Remark that we set $A[1] = 0$ because $L_i[1, |L_i|]\$$ has the same rank of $\$$ between the suffixes of $L_1 \cdots L_{i-1}\$$ and it is preceded by the same symbol $L_{i-1}(|L_{i-1}|)$ in $L_1 \cdots L_{i-1}L_i\$$. Consequently, in our algorithm considers the suffixes $L_i[1, |L_i|]\$$ and the suffix $\$$ (of the string $L_1 \cdots L_{i-1}\$$) as the same suffix. It is easy to prove that the value $A[|L_i| + 1]$ is 0. The array A is computed from the position $|L_i|$ to 2 by using Proposition 6.

Proposition 6. *Let j be a integer ranging from $|L_i|$ to 2 and let $A[j + 1]$ be the number of suffixes of $L_1 \cdots L_{i-1}\$$ lexicographically smaller than $L_i[j + 1, |L_i|]\$$. Let c be the first symbol of the suffix $L_i[j, |L_i|]\$$. Then,*

$$A[j] = C(bwt(L_1 \cdots L_{i-1}\$), c) + rank(bwt(L_1 \cdots L_{i-1}\$), c, A[j + 1]).$$

Proof. Since c is the first symbol of the suffix $L_i[j, |L_i|]\$$, then $L_i[j, |L_i|]\$ = cL_i[j + 1, |L_i|]\$$. All the suffixes of $L_1 \cdots L_{i-1}\$$ starting with a symbol smaller than c are lexicographically smaller than $L_i[j, |L_i|]\$$. The number of such suffixes is given by $C(bwt(L_1 \cdots L_{i-1}\$), c)$. Let us count now the number of suffixes that starting with c and are smaller than $L_i[j, |L_i|]\$$. This is equivalent to counting how many c's occur in $bwt(L_1 \cdots L_{i-1}\$)[1, A[j+1]]$. Such a value is given by $rank(bwt(L_1 \cdots L_{i-1}\$), c, A[j+1])$. □

It is easy to verify that we can obtain the array G by using the array A and the suffix array $SA(L_i\$)$, i.e. $G[i] = A[SA(L_i\$)[i]]$. Note that the array G contains the partial sums of the values of the *gap* array used in [9,12]. However, we could directly compute the array G by using the notion of inverse suffix array ISA[1]. Step 2 could be realized in $O(\sum_{j=1,\ldots,i} |L_j|)$ time because we can build a data structure supporting $O(1)$ time rank queries over $bwt(L_1 \cdots L_{i-1}\$)$. The same time complexity is obtained if the rank queries are executed over $bwt(L_i\$)$.

Step 3 uses G to create the new array $bwt(L_1 \cdots L_i\$)$ by merging $bwt(L_i\$)$ with the $bwt(L_1 \cdots L_{i-1}\$)$ computed at the previous iteration. Such a step implicitly constructs the lexicographically sorted list of suffixes starting in $L_1 \cdots L_{i-1}$ and extending up to end of L_i together with the suffixes of L_i. In order to do this we keep the mutual order between the suffixes of $L_1 \cdots L_{i-1}\$$ and $L_i\$$ thanks to Theorem 3. From the definition of the array G, it follows that the first two positions of the array $bwt(L_1 \cdots L_i\$)$ are the first symbol of $bwt(L_i\$)$ and the first symbol of $bwt(L_1 \cdots L_{i-1}\$)$, respectively. For $j = 3, \ldots, |L_i|$ we copy $G[j]$ values from $bwt(L_1 \cdots L_{i-1}\$)$ followed by the value $bwt(L_i\$)[j]$. It is easy to see that the time complexity of Step 3 is $O(\sum_{j=1,\ldots,i} |L_j|)$, too.

From the description of the algorithm and by proceeding by induction, one can prove the following proposition.

Proposition 7. *At the end of the iteration k, Algorithm* Bwt_Lynd *correctly computes $bwt(L_1 \cdots L_k\$)$. Each iteration i runs in $O(\sum_{j=1,\ldots,i} |L_j|)$ time. The overall time complexity is $O(k^2 M)$, where $M = \max_{i=1,\ldots,k}(|L_i|)$.*

[1] The inverse suffix array ISA of a word $W\$$ is the inverse permutation of SA, i.e., $ISA[SA[i]] = i$ for all $i \in [1, |w| + 1]$. The value $ISA[j]$ is the lexicographical rank of the suffix starting at the position j.

5 Discussions and conclusions

The goal of this paper is to propose a new strategy to compute the *bwt* and the SA of a text by decomposing it into Lyndon factors and by using the compatibility relation between the sorting of its local and global suffixes. At the moment, the quadratic cost of the algorithm could make it impractical. However, from one hand, in order to improve our algorithm, efficient dynamic data structure for the rank operations and for the insertion operations could be used. Navarro and Nekrich's recent result [21] on optimal representations of dynamic sequences shows that one can insert symbols at arbitrary positions and compute the rank function in the optimal time $O(\frac{\log n}{\log \log n})$ within essentially $nH_0(s) + O(n)$ bits of space, for a sequence s of length n. On the other hand, our technique, differently from other approaches in which partitions of the text are performed, is quite versatile so that it easily can be adapted to different implementative scenarios.

For instance, in [12] the authors describe an algorithm, called BWTE, that logically partitions the input text W of length n into blocks of the same length m, i.e. $W = T_{n/m}T_{n/m-1}\cdots T_1$ and computes incrementally the *bwt* of W via n/m iterations, one per block of W. Text blocks are examined from right to left so that at iteration $h+1$, they compute and store on disk $bwt(T_{h+1}\cdots T_1)$ given $bwt(T_h\cdots T_1)$. In this case the mutual order of the suffixes in each block depends on the order of the suffixes of the next block. Our algorithm BWT_LYND builds the *bwt* of a text or its SA by scanning the text *from left to right* and it could run online, i.e. while the Lyndon factorization is realized. One of the advantages is that adding new text to the end does not imply to compute again the mutual order of the suffixes of the text analyzed before, unless for the suffixes of the last Lyndon word that could change by adding characters on the right. Moreover, as described in the previous section, the text could be partitioned into several sequences of consecutive blocks of Lyndon words, and the algorithm can be applied *in parallel* to each of those sequences. Furthermore, also the Lyndon factorization can be performed in parallel, as shown in [3]. Alternatively, since we read each symbol only once, also an in-place computation could be suggested by the strategy proposed in [10], in which the space occupied by text W is used to store the $bwt(W)$.

Finally, in the description of the algorithm we did not mention the used workspace. In fact, it could depend on the time-space trade-off that one should reach. For instance, the methodologies used in [4,12] where disk data access are executed only via sequential scans could be adapted in order to obtain a lightweight version of the algorithm. An external memory algorithm for the Lyndon factorization can be found in [24]. We remark that that the method proposed in [12] could be integrated into BWT_LYND in the sense that one can apply BWTE to compute at each iteration the *bwt* and the SA of each block of the Lyndon partition.

In conclusion, our method seems lay out the path towards a new approach to the problem of sorting the suffixes of a text in which partitioning the text by using its combinatorial properties allows it to tackle the problem in local portions of the text in order to extend efficiently solutions to a global dimension.

References

1. M. ABOUELHODA, S. KURTZ, AND E. OHLEBUSCH: *The enhanced suffix array and its applications to genome analysis*, in Algorithms in Bioinformatics, vol. 2452 of LNCS, Springer Berlin Heidelberg, 2002, pp. 449–463.

2. D. Adjeroh, T. Bell, and A. Mukherjee: *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*, Springer Publishing Company, Incorporated, first ed., 2008.
3. A. Apostolico and M. Crochemore: *Fast parallel lyndon factorization with applications.* Mathematical systems theory, 28(2) 1995, pp. 89–108.
4. M. J. Bauer, A. J. Cox, and G. Rosone: *Lightweight algorithms for constructing and inverting the BWT of string collections.* Theoretical Computer Science, 483(0) 2013, pp. 134–148.
5. S. Bonomo, S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino: *Suffixes, Conjugates and Lyndon words*, in DLT, vol. 7907 of LNCS, Springer, 2013, pp. 131–142.
6. M. Burrows and D. J. Wheeler: *A block sorting data compression algorithm*, tech. rep., DIGITAL System Research Center, 1994.
7. K. T. Chen, R. H. Fox, and R. C. Lyndon: *Free differential calculus. IV. The quotient groups of the lower central series.* Ann. of Math. (2), 68 1958, pp. 81–95.
8. A. J. Cox, T. Jakobi, G. Rosone, and O. B. Schulz-Trieglaff: *Comparing DNA sequence collections by direct comparison of compressed text indexes*, in WABI, vol. 7534 LNBI of LNCS, Springer, 2012, pp. 214–224.
9. A. Crauser and P. Ferragina: *A theoretical and experimental study on the construction of suffix arrays in external memory.* Algorithmica, 32(1) 2002, pp. 1–35.
10. M. Crochemore, R. Grossi, J. Kärkkäinen, and G. Landau: *Constant-Space Comparison-Based Algorithm for Computing the Burrows-Wheeler Transform*, in CPM, LNCS, Springer, 2013, In press.
11. J.-P. Duval: *Factorizing words over an ordered alphabet.* Journal of Algorithms, 4(4) 1983, pp. 363–381.
12. P. Ferragina, T. Gagie, and G. Manzini: *Lightweight Data Indexing and Compression in External Memory.* Algorithmica, 63(3) 2012, pp. 707–730.
13. P. Ferragina and G. Manzini: *Opportunistic data structures with applications*, in FOCS 2000, IEEE Computer Society, 2000, pp. 390–398.
14. P. Ferragina and G. Manzini: *An experimental study of an opportunistic index*, in SODA 2001, SIAM, 2001, pp. 269–278.
15. R. Giancarlo, A. Restivo, and M. Sciortino: *From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization.* Theoret. Comput. Sci., 387(3) 2007, pp. 236–248.
16. R. Grossi: *A quick tour on suffix arrays and compressed suffix arrays.* Theoretical Computer Science, 412(27) 2011, pp. 2964–2973.
17. C. Hohlweg and C. Reutenauer: *Lyndon words, permutations and trees.* Theoretical Computer Science, 307(1) 2003, pp. 173–178.
18. M. Lothaire: *Applied Combinatorics on Words (Encyclopedia of Mathematics and its Applications)*, Cambridge University Press, New York, NY, USA, 2005.
19. U. Manber and G. Myers: *Suffix arrays: a new method for on-line string searches*, in Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms, SODA '90, Philadelphia, PA, USA, 1990, SIAM, pp. 319–327.
20. S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino: *A new combinatorial approach to sequence comparison.* Theory Comput. Syst., 42(3) 2008, pp. 411–429.
21. G. Navarro and Y. Nekrich: *Optimal dynamic sequence representations*, in Proc. 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2013, pp. 865–876.
22. S. J. Puglisi, W. F. Smyth, and A. H. Turpin: *A taxonomy of suffix array construction algorithms.* ACM Comput. Surv., 39 2007.
23. A. Restivo and G. Rosone: *Balancing and clustering of words in the Burrows-Wheeler transform.* Theoret. Comput. Sci., 412(27) 2011, pp. 3019–3032.
24. K. Roh, M. Crochemore, C. S. Iliopoulos, and K. Park: *External memory algorithms for string problems.* Fundam. Inf., 84(1) 2008, pp. 17–32.
25. G. Rosone and M. Sciortino: *The Burrows-Wheeler Transform between Data Compression and Combinatorics on Words*, in CiE, vol. 7921 of LNCS, Springer, 2013, In press.
26. J. T. Simpson and R. Durbin: *Efficient construction of an assembly string graph using the FM-index.* Bioinformatics, 26(12) 2010, pp. i367–i373.

Optimal Partitioning of Data Chunks in Deduplication Systems

Michael Hirsch[1], Ariel Ish-Shalom[1], and Shmuel T. Klein[2]

[1] IBM — Diligent, Atidim Industrial Park, Tel Aviv 61580, Israel
{michael.hirsch, arielish}@il.ibm.com

[2] Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
tomi@cs.biu.ac.il

Abstract. Deduplication is a special case of data compression in which repeated chunks of data are stored only once. For very large chunks, this process may be applied even if the chunks are similar and not necessarily identical, and then the encoding of duplicate data consists of a sequence of pointers to matching parts. However, not all the pointers are worth being kept, as they incur some storage overhead. A linear, sub-optimal solution of this partition problem is presented, followed by an optimal solution with cubic time complexity and requiring quadratic space.

1 Introduction and Background

Large backup and storage systems need to process ever increasing amounts of data, and standard lossless data compression methods may not be able to cope with it. On the other hand, the use of classical compression may be an overkill, since backup data has generally the property that only a small fraction of it is changed between consecutive backup generations. This calls for a special form of data compression, known as *deduplication*, which tries to store repetitive data only once. The challenge is, of course, to locate as much of the duplicated data as possible.

A general paradigm to achieve this goal could be the following. Partition the input database, which is often called the *repository*, into fixed or variable sized blocks, called *chunks*, apply a cryptographically strong hash function on each of these input chunks, and store the different hash values, along with the address of the corresponding chunk, in a fast to access data structure, like a hash table or a B-Tree [6,7]. When a fresh copy of the data is given, e.g., for a weekly or even daily backup, the new data, often called a *version*, is also partitioned into similar chunks, and a chunk is only kept if the corresponding hash value is not stored yet. Otherwise it is replaced by a pointer to the already stored copy.

A major dilemma is to decide what the (average) chunk size should be, as if it is too small, the number of chunks and the accompanying overhead might be too large; on the other hand, the larger the chunks, the lower is the probability of finding identical ones, reducing the potential deduplication benefits. Note that systems based on using hashing functions are generally only able to detect *identical* chunks, because most hashing functions are designed with the specific aim that even small changes in the argument should imply substantive changes in the hashed values. This lead to the idea of devising deduplication systems based on *similarity* rather than identity, thereby allowing the use of considerably larger chunks, as in the IBM ProtecTIER product, described in [1]. An extension of this similarity based deduplication system to an environment using small sized chunks has been presented last year at this conference [2]. We now focus again on systems using very large chunks, and shall deal with the following problem implied by it.

While a single pointer is sufficient for the compression of an identical chunk, the case of similar chunks is more involved. Similarity might imply that most of the data of the version chunk can be copied from the repository, but the data to be copied is not necessarily contiguous and might appear in various chunks; moreover, even if several pointers refer to the same repository chunk, they could point to locations that are scattered throughout it. In fact, the encoding of a compressed chunk will be a sequence of various copy items, interspersed with stretches of new data. If one considers quite long chunks, say, of the order of 16MB, and adds to this the fact that the new data can be as short as a single byte, the conclusion is that the number of elements in the encoding of a single chunk may be large.

This situation is aggravated in a typical scenario of a backup system, which stores several consecutive *generations* of almost the same data. There might only be minor changes between adjacent generations, but these changes have a cumulative effect, leading to chunks that are increasingly fragmented into smaller and smaller copy and non-copy items. However, storing the data needed to reconstruct a highly fragmented chunk may itself create a compression problem.

In the next section, we define the specific problem dealt with herein, namely finding an optimal partition of a chunk into matching and non-matching parts. Section 3 then suggests a sub-optimal, yet linear, algorithm, and Section 4 an optimal one, requiring cubic time. Section 5 brings a few improvements. We opted for suggesting only a theoretical framework, so there is no experimental section, which is justified in the conclusion.

2 Definition of the problem

We thus consider applying a filtering stage after having located all the matching parts, which should eliminate those parts of the compressed data that will ultimately not be worth being kept, because the required overhead might be larger than the compression gain. The input to this part of the process is a chunk of data and a *list of matches*, each consisting of a pair of pointers, one to the given version chunk, one into the repository, and the size of the matching substring. The expected output is a partition of the given chunk into a sequence of mismatching and matching blocks. The compressed form of the chunk will then consist of a copy of the mismatching parts, and of pointers describing where the matching parts can be found.

A simplistic solution would of course be to build the output by just copying the input, that is, accept exactly the partition found by listing all the matches. But this would ignore the fact that at least a part of the matches are not worth being kept, as they might cause a too high degree of fragmentation. The challenge is therefore to decide which matches should be kept, and which should be ignored.

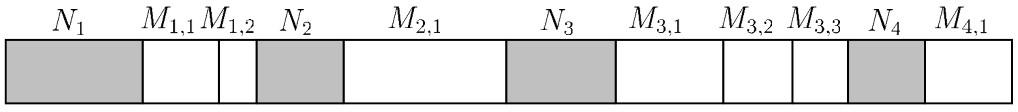


Figure 1. Schematic representation of the partition of a data chunk

Figure 1 shows a possible partition of a data chunk into alternating areas of non-matches and matches. The non-matches, represented by the grey rectangles, contain **N**ew data and are indexed $N_1, N_2, \ldots, N_k$. The **M**atches, drawn as the white rectangles, contain data that has previously appeared in the repository, and will be stored by means of pointers of the form (address, length); the matching parts between the non-matching blocks N_i and N_{i+1} are indexed $M_{i,1}, M_{i,2}, \ldots M_{i,j_i}$. Non-matching parts cannot be consecutive — this is new data, and any stretch of such new characters is considered a single new part. The matching parts, on the other hand, may consist of several different sub-parts that are located in different places on the disk; each sub-part needs therefore a pointer of its own.

We consider two functions defined on these matching and non-matching parts. A cost function $c()$ giving the price we incur for storing the pointers in the meta-data; typically, but not necessarily, all pointers are of fixed length E (in our implementation, $E = 24$ bytes), that is $c(N_i) = c(M_{\ell,j}) = 24$ for all indexes, so that actually, the cost for the meta-data depends only on the number of parts, which is $k + \sum_{t=1}^{k} j_t$. In other implementations, the pointers may undergo another layer of compression, e.g., Huffman coding, resulting in variable length elements.

The second function $s()$ measures, for each part, the size of the data on the disk. So we have that $s(N_i)$ will be just the number of bytes of the non-matching part, as these new bytes have to be stored physically somewhere, and $s(M_{\ell,j}) = 0$, since no new data is written to the disk for a matching part. However, we shall define $s(M_{\ell,j}) =$ length for a block $M_{\ell,j}$ that is stored by means of a pointer (address, length), which means that the size will be defined as the number of bytes written to the disk in case we decide to ignore the fact that $M_{\ell,j}$ has occurred earlier and thus has a matching part already in the repository.

The compressed data consists of the items written to the disk plus the pointers in the meta-data, but these cannot necessarily be traded one to one, as storage space for the meta-data will generally be more expensive. We shall assume that there exists a multiplicative factor F such that, in our calculations, we can count one byte of meta-data as equivalent to F bytes of data written to the disk. This factor need not be constant and may dynamically depend on several run-time parameters. Practically, F will be stored in a variable and may be updated when necessary, but we shall use it in the sequel as if it were a constant.

Given the above notations, the size of the compressed file is then

$$F \cdot \left[\sum_{i=1}^{k} \left(c(N_i) + \sum_{t=1}^{j_i} c(M_{i,t}) \right) \right] + \sum_{i=1}^{k} s(N_i),$$

and in the particular case of fixed length pointers of size E, which we shall assume, for simplicity, in the sequel:

$$F \cdot E \cdot \left(k + \sum_{t=1}^{k} j_t \right) + \sum_{i=1}^{k} s(N_i), \qquad (1)$$

whereas the uncompressed file has size

$$\sum_{i=1}^{k} \left(s(N_i) + \sum_{t=1}^{j_i} s(M_{i,t}) \right).$$

The optimization problem we consider is based on the fact that the partition we obtain as input may be altered. The non-matching parts N_i can obviously not be touched, so the only degree of freedom we have is to decide, for each of the matching parts $M_{i,j}$, whether the corresponding pointer should be kept, or whether we opt to ignore the match and treat this part as if it were non-matching. There is a priori nothing to be gained from such a decision: the pointer in the meta-data is changed from matching to non-matching, but incurs the same cost, and some data has been added to the disk, so there will always be a loss.

The following example shows that nevertheless, there can also be a gain in certain cases. Consider the block $M_{1,2}$ in Figure 1. If we decide to ignore its matching counterpart, the data of $M_{1,2}$ has to be written to the disk, but it is contiguous with the data of N_2. The two parts may therefore be fusioned, which reduces the number of meta-data entries by one. This will result in a gain if

$$s(M_{1,2}) < F \cdot E.$$

Moreover, if indeed we decide to consider $M_{1,2}$ as a non-matching block, this will leave $M_{1,1}$ as a single match between two non-matches. In this case, ignoring the match may allow to unify the three blocks $N_1, M_{1,1}, N_2$, reducing the number of meta-data entries by two. This will be worthwhile even if

$$s(M_{1,1}) < 2\, F \cdot E.$$

More generally, any extremal matching blocks (those touching on at least one of their sides with a non-match) may be candidates for such a fusion, which can trigger even further unifications like in the example. But these are not the only cases: even non-extremal blocks may profit from unification. This is not true for a single matching blocks, whose both neighbors are also matching, like $M_{3,2}$ in Figure 1, because we add data to the disk, but do not remove any meta-data, just change one of the entries. But there might be a stretch of several matching blocks that can profit from unification.

It should be noted that devising a new partition is not only a matter of trading a byte of meta-data versus F bytes of disk data. Reducing the number of entries in the meta-data has also an effect of the time complexity, since each entry requires an additional read operation. Many compression algorithms have to deal with such time/space tradeoffs, and for our purpose, we shall assume that the factor F already takes also the time complexity into account, that is, F reflects our estimation of how many bytes of disk space we are ready to pay in order to save one byte of meta-data, considering all aspects, including space, CPU and I/O.

The challenge is therefore to come up with an efficient, and if possible, optimal way to select an appropriate subset of the input partition which minimizes the size of the compressed file as measured by equation (1).

3 Linear sub-optimal algorithm

The following algorithm is a first solution attempt. The partition it produces is not necessarily optimal, but the complexity is linear with the number of elements N_i and $M_{i,j}$. The algorithm uses as main data structure a doubly linked list L, the elements of which represent the matching or non-matching data blocks defined above, so their initial number is $n = k + \sum_{t=1}^{k} j_t$. Each element p of the list L has the following fields:

- **status**(p) – indicating whether the element pointing p is matching (M), non-matching (NM), or a sentinel element (S) for smoother programming

- **prev**(p) – pointing to the predecessor of p
- **succ**(p) – pointing to the successor of p
- **size**(p) – if status(p) = NM, this is the number of non-matching bytes; if status(p) = M, this is the length of the element to be copied; if status(p) = S, size(p) is not defined.
- **data**(p) – defined only if status(p) = NM, in which case it contains the new data not found in the repository; if status(p) = M, nothing will be stored in data(p), but we shall refer by DATA(p) to the bytes pointed to by the (address, length) pointer.

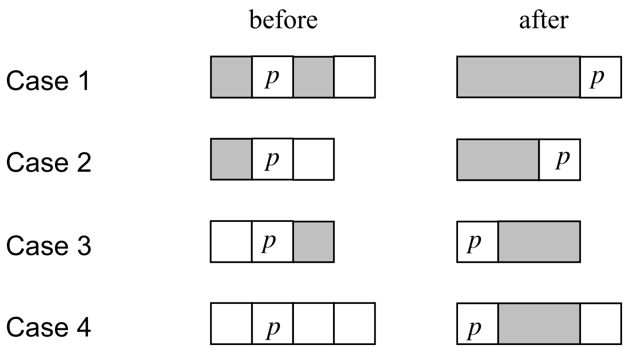


Figure 2. Different cases dealt by the algorithm

We first add sentinel elements at the beginning and end of the list, which avoids the necessity to check at each step whether successors and predecessors exist. The main idea is then to scan the list of items with a pointer p and perform local substitutions according to the contexts, if possible. If the current item is of type NM, it is skipped. If it is a matching item, we consider 5 disjoint cases.

1. Case 1: The item pointed to by p is surrounded by NM items. In this case, all 3 elements can be merged into one, if appropriate, that is, if **size**(p)$< 2FE$.
2. Case 2: The item pointed to by p is preceded by an NM item; it can then be merged into the preceding item, if appropriate. Note that if several consecutive items can be merged, this is dealt with in the following iterations.
3. Case 3: The item pointed to by p is followed by an NM item; this case is symmetric to Case 2 .
4. Case 4: The item pointed to by p is surrounded by M items. We then check whether two M items can be merged into one NM item. Longer chains of M items are considered in the following iterations, though then in Case 3.
5. Case 5: No substitution is possible, just advance p to its successor.

The four first cases are schematically represented in Figure 2, where as before, NM items appear in grey and M items in white. As part of the actions to be performed in each case, the pointer p has to be repositioned. In the first 2 cases, p will point to the item following the newly merged block, so the next iteration will take us to Case 2, and in the last 2 cases, p will point to the item preceding the newly merged block, so the next iteration will take us to Case 3.

It therefore follows that the main pointer of the procedure may also move backwards, which could result in an unbounded number of iterations. But in each iteration, either the pointer is advanced by one step, or the overall number of items is reduced

```
$p \longleftarrow$ succ(TOP)
while succ($p$) $\neq$ NULL
    if status($p$) $\neq$ M then
        $p \longleftarrow$ succ($p$)
    else

        if status(prev($p$)) $=$ NM and status(succ($p$)) $=$ NM and size($p$) $< 2\,F\,E$ then
                    // Case 1
            $q \longleftarrow$ prev($p$)
            size($q$) $\longleftarrow$ size($q$) $+$ size($p$) $+$ size(succ($p$))
            $q \longleftarrow$ succ(succ($p$))
            delete succ($p$) from $L$
            delete ($p$) from $L$
            $p \longleftarrow q$

        else if status(prev($p$)) $=$ NM and size($p$) $< F\,E$ then
                    // Case 2
            $q \longleftarrow$ prev($p$)
            size($q$) $\longleftarrow$ size($q$) $+$ size($p$)
            $q \longleftarrow$ succ($p$)
            delete ($p$) from $L$
            $p \longleftarrow q$

        else if status(succ($p$)) $=$ NM and size($p$) $< F\,E$ then
                    // Case 3
            $q \longleftarrow$ succ($p$)
            size($q$) $\longleftarrow$ size($q$) $+$ size($p$)
            $q \longleftarrow$ prev($p$)
            delete ($p$) from $L$
            $p \longleftarrow q$

        else if status(prev($p$)) $\neq$ NM and status(succ(succ($p$))) $\neq$ NM
                and size($p$) $+$ size(succ($p$)) $< F\,E$ then
                    // Case 4
            status($p$) $\longleftarrow$ NM
            size($p$) $\longleftarrow$ size($p$) $+$ size(succ($p$))
            $q \longleftarrow$ prev($p$)
            delete succ($p$) from $L$
            $p \longleftarrow q$

        else
            $p \longleftarrow$ succ($p$)
```

Figure 3. Linear sub-optimal algorithm

by one, which bounds the global complexity to be at most $2n$ iterations, each requiring $O(1)$ commands. Note, however, that this solution is not necessarily optimal, as sequences of consecutive blocks are substituted greedily by pairs. It may happen that 3 consecutive M items could be merged, but considered as two pairs, none of them will result in a substitution. The formal algorithm is given in Figure 3.

4 Optimal solution of the partition problem

We now turn to an optimal solution of the partition problem. The solution will be applied individually on each sequence of consecutive M-items, surrounded on both ends by NM-items, since these cannot be altered, and the only possible transformation is to declare matching blocks as if they were non-matching. Therefore the originally given NM-items will appear also in the final optimal solution, so we can concentrate on each sub-part on its own. Consider then the (matching) elements as indexed $1, 2, \ldots, n$, and the non-matching delimiters as indexed 0 and $n+1$.

Notation: we shall return the required partition in the form of a bit-string of length n, with the bit in position i being set to 1 if the i-th element should be of type NM, and set to 0 if the i-th element should be of type M. This notation implies immediately that the number of possible solutions is 2^n, so that an exhaustive search of this exponential number of alternatives is ruled out.

The basis for a non-exponential solution is the fact that the optimal partition can be split into sub-parts, each of which has to be optimal for the corresponding subranges. We can thus get the solution for a given range by trying all the possible splits into, say, two sub-parts. Such recursive definitions call for resolving them by means of dynamic programming [4]. The tricky part here is that the optimal solution for the range (i, j), might depend on whether its bordering elements, indexed $i-1$ and $j+1$, are of type matching or non-matching, so the optimal solution for range (i, j) might depend on the optimal solution on the neighboring ranges.

The optimal partition will thus be built by means of a two-dimensional dynamic programming table $C[i, j]$, and the optimal partition will be stored in a similar table PS, so that $PS[i, j]$ holds the optimal partition for the given parameters, which is a bit-string of length $j-i+1$. For $1 \leq i \leq j \leq n$, we define $C[i, j]$ as the global cost of the optimal partition of the sub-sequence of elements $i, i+1, \ldots, j-1, j$, when the surrounding elements $i-1$ and $j+1$ are of type NM. This cost will be given in bytes and reflects the size of the data on disk for NM-items, plus the size of the meta-data for all the elements, using the equivalence factor explained above, that is, each meta-data entry incurs a cost of FE bytes. Once the table is filled up, the cost of the optimal solution we seek is stored in $C[1, n]$ and the corresponding partition is given in $PS[1, n]$.

The basis of the calculation will be the individual items themselves stored in the main diagonal of the matrix, $C[i, i]$ for $1 \leq i \leq n$, as well as the elements just below the diagonal, $C[i, i-1]$. The following iterations will then be ordered by increasing difference between i and j. We shall thus first deal with all sequences of two adjacent elements, then 3, etc. When calculating the optimal solution for a sequence of ℓ adjacent elements, we can use our knowledge of the optimal solutions for all shorter sub-sequences. If fact, for a sequence of length $\ell = j-i+1$, we only need to check the sum of the costs of all possible partitions of this range into two subranges, that is the cost for $(i, k-1)$ plus that of $(k+1, j)$ for $i < k < j$. We initialize the cost for each subrange by the possibility of leaving all the n elements of type matching.

More specifically, the formal algorithm is given in Figure 4 and the line numbers below refer to this figure. Lines 1 and 3 initialize the table for ranges of size 0, that is, of type $[i+1, i]$, giving them a cost 0. The corresponding bit-string are Λ, which denotes the empty string. Lines 4–7 deal with singletons of type $[i, i]$. Since we assume that the surrounding elements are both of type NM, we have to compare the size $s(i)$ of the matching element with the cost of defining it as non-matching, and letting it be absorbed by the neighboring NM items. In that case, two elements of the meta-data can be saved, which is checked in line 4.

```
1  $C[n+1,n] \longleftarrow 0$    $PS[n+1,n] \longleftarrow \Lambda$
2  for $i \longleftarrow$ 1 to $n$
3    $C[i,i-1] \longleftarrow 0$    $PS[i,i-1] \longleftarrow \Lambda$
4    if $s(i) - FE < FE$ then
5      $C[i,i] \longleftarrow s(i) - FE$    $PS[i,i] \longleftarrow$ '1'
6    else
7      $C[i,i] \longleftarrow FE$    $PS[i,i] \longleftarrow$ '0'
8  end for $i$

9  for $diff \longleftarrow$ 1 to $n-1$
10   for $i \longleftarrow$ 1 to $n - diff$
11     $j \longleftarrow i + diff$
12     $C[i,j] \longleftarrow (diff+1)FE$
13     $PS[i,j] \longleftarrow$ '000$\cdots$0'   //(length $diff+1$)

14     $OK \longleftarrow 0$
15     for $k \longleftarrow i$ to $j$
16       if $k = j$ then
17         $L \longleftarrow 1$
18       else
19         $L \longleftarrow$ left($PS[k+1,j]$)
20       if $k = i$ then
21         $R \longleftarrow 1$
22       else
23         $R \longleftarrow$ right($PS[i,k-1]$)

24       $newcost \longleftarrow C[i,k-1] + C[k+1,j] + s(k) + (1-L-R)FE$
25       if $newcost < C[i,j]$
26         $C[i,j] \longleftarrow newcost$
27         $OK \longleftarrow k$
28     end for $k$

29     if $OK > 0$ then
30         $PS[i,j] \longleftarrow PS[i,OK-1] \;\|\; '1' \;\|\; PS[OK+1,j]$
31   end for $i$
32 end for $diff$
```

Figure 4. Optimal algorithm

The main loop starts then on line 9. The table is filled primarily by diagonals, each corresponding to a constant difference $\mathit{diff} = j - i$, and within each diagonal, by increasing i. Line 11 redefines j just for notational convenience.

In lines 12–13, the table entries are given default values, corresponding to the extreme case of all $\mathit{diff} + 1$ elements in the range between and including i and j remaining matching as initially given in the input. This corresponds to a bitstring of $\mathit{diff} + 1$ zeroes '000· · ·0' in PS. As to the cost of the default partition, we have to store $\mathit{diff} + 1$ meta data blocks, at the total price of $(\mathit{diff} + 1)FE$.

After having initialized the table, the loop starting in line 15 tries to partition the range (i, j) into two sub-pieces. The idea is to consider two possibilities for the optimal partition of the range $[i, j]$: either all the $\mathit{diff}+1$ elements should remain matching, as we assume in the default setting initializing the $C[i, j]$ value in line 12, or there is at least one element k, with $i \le k \le j$, which in the optimal partition should be turned into an NM-element. The optimal solution is then obtained by solving the problem recursively on the remaining sub-ranges $(i, k - 1)$ and $(k + 1, j)$. The advantage of this definition is that the surrounding elements of the sub-ranges, $i - 1$ and k for $(i, k - 1)$, and k and $j + 1$ for $(i, k - 1)$, are again both of type NM, so the same table C can be used.

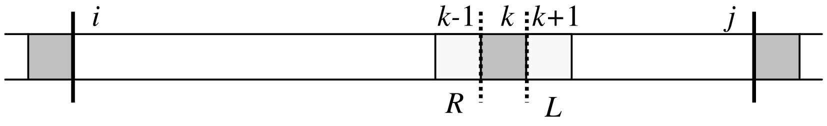


Figure 5. Schematic representation of a partition of a sub-range

However, to combine the optimal solutions of the sub-ranges into an optimal solution for the entire range, one needs to know whether the elements adjacent to the separating element indexed k are of type M or NM. For if one or both of them are NM, they can be merged with the separating element itself, so the meta-data decreases by one or two elements, reducing the price by FE or $2FE$. Let L denote the leftmost element of the right range $[k + 1, j]$, and R the rightmost element of the left range $[i, k - 1]$. These values are assigned in lines 16–23, including extremal values. The general case is depicted in Figure 5. We thus need a function $f(L, R)$, giving the number of additional meta-data elements needed as function of the type 0 or 1, corresponding to M or NM, of the bordering elements L and R. This function should give values according to Table 1. A possible function is thus $f(L, R) = 1 - L - R$, which explains the definition of the *newcost* in line 24.

L	R	$f(L, R)$
1	1	-1
0	1	0
1	0	0
0	0	1

Table 1. Values for $f(L, R)$

We check the sum of the costs of the optimal solutions of the sub-problems plus the cost of the separating element, and keep the smallest such sum, over all the possible partition points k, in the table entry $C[i,j]$. In other words,

$$C[i,j] \leftarrow \min \begin{cases} (\mathit{diff}+1)FE, \\ \min\limits_{i \le k \le j} \left(C[i,k-1] + C[k+1,j] + s(k) + (1-L-R)FE\right). \end{cases}$$

OK stores the value of k for which the optimal partition has been found, i.e., that with minimum cost. If the default value has been changed, the optimal solution, expressed as a bitstring of length $\mathit{diff}+1$, is obtained in line 30 by concatenating the bitstrings corresponding to the optimal solutions of the subranges and between them the string $'1'$ corresponding to the element indexed k. The operator $\|$ denotes concatenation.

The complexity of evaluating the table is dominated by the loops starting at line 9. There are three nested loops, and the loop on k goes from i to $j-1 = i + \mathit{diff} - 1$, so it is executed diff times for each possible value of diff and i. The total number of iterations is therefore

$$\sum_{i=1}^{n-1} i(n-i) = \left[n\frac{n(n-1)}{2} - \frac{(n-1)n(2n-1)}{6}\right] = \frac{1}{6}(n^3 - n).$$

Such a cubic number of iterations might be prohibitive, even though the coefficient of n^3 is at most 0.17. Recall that n, the input parameter of the number of consecutive blocks dealt with in each call to the program for the optimal partition, is the number of consecutive matching items between two non-matching ones. In terms of our bit-string notation: the result of applying the deduplication algorithm of a large input chunk is a sequence of matching or non-matching items, which we denoted by a bit-string of the form, e.g., 1001000101110000000100... The optimal partition algorithm is then invoked for each of the 0-bit runs, which, on the given example, are of lengths 2, 3, 1, 0, 0, 7, etc. There is of course no need to call the procedure when $n = 0$.

5 Improvements

5.1 Reducing the time complexity

If certain values of n are too large, one may try to reduce the complexity a priori by applying a preliminary filtering heuristic that will not impair the optimal solution. For example, one could consider the maximal possible gain from declaring a matching item (0) to be non-matching (1). This happens if the two adjacent blocks are non-matching themselves, and then all 3 items could be merged into a single one. The savings would then be equivalent to $2FE$ bytes, which have to be counterbalanced by the loss of $s(i)$ bytes that are not referenced anymore, so have to be stored explicitly. Thus, if $s(i) > 2FE$, the ith M-element will surely not be transformed into an NM-element. It follows that $s(i) > 2FE$ is a sufficient condition for keeping the value of the ith bit in the optimal partition as 0.

The heuristic will then scan all the input items and check this condition for each 0-item. If the condition holds, the element can be declared to remain of type 0, which partitions the rest of the elements into two parts. For example, if the middle element of n is thereby declared as keeping its 0-status, we have split the n elements into two parts of size $n/2$ each, so the complexity is reduced from $\frac{1}{6}n^3$ to $2\frac{1}{6}\left(\frac{n}{2}\right)^3 = \frac{1}{24}n^3$. Returning to the example bit-string above 10**0**10**0**0101110000**00**00100..., if the boldfaced elements are those fixed by the heuristic in their 0-status, the algorithm will be invoked with lengths 1, 1, 1, 1, 3, 2, etc. Theoretically, the worst case didn't change, even after applying this heuristic, but in practice, the largest values of n might be much smaller.

There remains a technical problem: the optimal partition evaluated in $C[i,j]$ is based on the assumption that the surrounding elements $i-1$ and $j+1$ were of type 1, and if the above heuristic is applied, this assumption is not necessarily true. Two approaches are possible to confront this problem. We could use the value of $C[i,j]$ and the corresponding partition in $PS[i,j]$ and adapt it locally to the cases if one of the surrounding elements is 0. For example, if the rightmost bit in $PS[i,j]$ is 0, and bit $j+1$ is also 0, then no adaptation is needed; but if the rightmost bit in $PS[i,j]$ is 1, and bit $j+1$ is 0, then the optimal value $C[i,j]$ took into account that elements j and $j+1$ were merged, which is not true in our case, so the value of $C[i,j]$ has to be increased by one meta-data element, that is by FE. A similar adaptation is needed for the left extremity, element $i-1$. Such an adaptation is not necessary optimal, since it might be possible that, had we known that the surrounding elements are not both 1, an altogether different solution will be optimal.

As a second approach, we could extend the definitions of the $C[i,j]$ and $PS[i,j]$ tables to be 4-dimensional, with $C[i,j,L,R]$ being the cost of the optimal partition of the elements $i, i+1, \ldots, j$, under the assumption that the bordering elements $i-1$ and $j+1$ are of type L and R, respectively, where $L, R \in \{0,1\}$. Similarly, $PS[i,j,L,R]$ will hold the optimal partition for the given parameters. There are only four possibilities for L and R: $LR \in \{00, 01, 10, 11\}$, and the total size of each table is therefore only $2n^2$.

As above, one tries to partition the range (i,j) into two pieces, just without a separating element as before. The ranges will be (i,k) and $(k+1,j)$, for some $i \leq k < j$. L and R still denote the elements to the left of i and to the right of j, respectively, but we also need the bordering elements of the subranges, which again can be of type M or NM, denoted by 0 or 1, respectively. We therefore need to iterate on the possible internal left and right values IL and IR. It might be easiest to understand the notation by referring to the schema in Figure 7. The left subrange, (i,k), is delimited on its left by L and on its right by IL, whereas the right subrange, $(k+1,j)$, is delimited on its left by IR and on its right by R. The notation thus refers each bordering element to the position of the corresponding subrange, rather than to its own position, which is why IL appears in the figure to the right of IR.

Iterating of the four possibilities for (IL, IR), we have to check for consistency. Suppose, for example, that we consider $IL = 0$. That means that we are looking for the optimal partition of the left range (i,k), under the condition that the bordering elements are L and $IL = 0$. But we have also to check that the complementing optimal solution of the right range $(k+1,j)$ is such that its leftmost bit is indeed 0. A similar consistency check verifies that the optimal solution for the right range $(k+1,j)$ is taken for the given value of IR and that indeed, the rightmost bit of the string corresponding to the left range (i,k) is consistent with this IR value. If there is consistency, we check the sum of the costs of the optimal solutions of the sub-problems, and keep the smallest such sum, over all the possible partition points k. If there is no consistency for any k, the default value of keeping all bits as 0 is chosen. We omit here the formal algorithm and the details.

```
$C[n+1,n] \longleftarrow 0$   $LT[n+1,n] \longleftarrow 1$   $RT[n+1,n] \longleftarrow 1$
for $i \longleftarrow 1$ to $n$
  $C[i,i-1] \longleftarrow 0$   $LT[i,i-1] \longleftarrow 1$   $RT[i,i-1] \longleftarrow 1$
  if $s(i) - FE < FE$ then
    $C[i,i] \longleftarrow s(i) - FE$   $S[i,i] \longleftarrow i$
    $LT[i,i] \longleftarrow 1$   $RT[i,i] \longleftarrow 1$
  else
    $C[i,i] \longleftarrow FE$
    $LT[i,i] \longleftarrow 0$   $RT[i,i] \longleftarrow 0$
end for $i$

for $diff \longleftarrow 1$ to $n-1$
  for $i \longleftarrow 1$ to $n - diff$
    $j \longleftarrow i + diff$
    $C[i,j] \longleftarrow (diff + 1)FE$
    $LT[i,j] \longleftarrow 0$   $RT[i,j] \longleftarrow 0$

    $OK \longleftarrow 0$
    for $k \longleftarrow i$ to $j$
      $L \longleftarrow LT[k+1,j]$
      $R \longleftarrow RT[i,k-1]$
      $newcost \longleftarrow C[i,k-1] + C[k+1,j] + s(k) + (1-L-R)FE$
      if $newcost < C[i,j]$
        $C[i,j] \longleftarrow newcost$
        $OK \longleftarrow k$
    end for $k$

    $S[i,j] \longleftarrow OK$

    if $OK > 0$ then
      $LT[i,j] \longleftarrow LT[i,OK-1]$   $RT[i,j] \longleftarrow RT[OK+1,j]$
  end for $i$
end for $diff$
```

Figure 6. Optimal algorithm with reduced space complexity

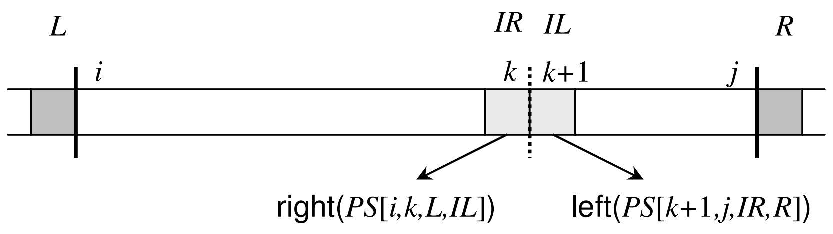


Figure 7. Schematic representation of an alternative partition of a sub-range

5.2 Reducing the space complexity

While the time complexity is $\theta(n^3)$, the $C[i,j]$ table needs only n^2 space. But the strings stored in the $PS[i,j]$ table are of length $j-i+1$, so that the space for $PS[i,j]$ is also $\theta(n^3)$. We can reduce this and store only $O(1)$ for each entry at the cost of not giving the optimal partition explicitly, but providing enough information for the optimal partition to be built in linear time, similarly to what has been done in [5].

The key to this reduction is storing in $PS[i,j]$ (which we call now $S[i,j]$ to avoid confusions) not the string itself, but the value OK at which the range $[i,j]$ has been split in an optimal way (line 27), or leaving it undefined, if no such value OK exists. Since the string $PS[i,j]$ served also to provide information on its extremal elements (left and right in lines 19 and 23 of the algorithm in Figure 4), these elements have now to be saved in tables LT and RT on their own. The updated algorithm is given in Figure 6.

To build the optimal solution, we initialize a vector A with n zeros, and then change selected values according to the values in the $S[i,j]$ matrix, using the recursive procedure Fill_Sol, given in Figure 8. It will be invoked by Fill_Sol$(A,1,n)$. The total running time of the recursion is clearly bounded by n.

```
Fill_Sol($A, i, j$)
  if $j \geq i$ AND $S[i,j]$ is defined
    $k \longleftarrow S[i,j]$
    $A[k] \longleftarrow 1$
    Fill_Sol($A, i, k-1$)
    Fill_Sol($A, k+1, j$)
```

Figure 8. Construction of the optimal solution

6 Conclusion

Papers presenting new compression schemes usually contain experimental sections reporting on tests of the suggested algorithms. But while there are well established test cases which have been agreed upon in the compression community, like the Calgary or the Canterbury [3] corpora, there is no equivalent for deduplication tests. The reason is mainly that the performance does not depend on the nature of the files, but rather on the their repetitiveness. Thus even a file containing random data, which cannot be compressed, may still profit from deduplication if it appears more than once in the repository.

There is therefore no possibility to find data that could be deemed to be representative, which is why we have preferred to leave this article on the theoretic level. We nevertheless collect statistics on the performance of the new methods when applied on a large deduplication system. The experimental results will be presented as examples only, without claiming that one could extrapolate from them information on the performance in general. These results will be presented in an extended version of this paper.

References

1. Aronovich L., Asher R., Bachmat E., Bitner H., Hirsch M., Klein S.T., The Design of a Similarity Based Deduplication System, *Proc. SYSTOR'09*, Haifa, (2009) 1–14.
2. Aronovich L., Asher R., Harnik D., Hirsch M., Klein S.T., Toaff Y., Similarity based Deduplication with small data chunks, *Proc. Prague Stringology Conference PSC–2012*, Prague, (2012) 3–17.
3. http://corpus.canterbury.ac.nz/
4. Cormen T.H., Leiserson C.E., Rivest R.L., *Introduction to Algorithms*, MIT Press, 1990.
5. Klein S.T., On the Use of Negation in Boolean IR Queries, *Information Processing & Management* **45** (2009) 298–311.
6. Quinlan S., Dorward S., Venti: A New Approach to Archival Storage, *Proceedings of FAST'02, the 1st USENIX Conference on File and Storage Technologies* , Monterey, CA (2002) 89–101.
7. Zhu B., Li K., Patterson H., Avoiding the Disk Bottleneck in the Data Domain Deduplication File System, *Proceedings of FAST'08, the 6th USENIX Conference on File and Storage Technologies*, San Jose, CA (2008) 279–292.

Parallel Suffix Array Construction by Accelerated Sampling[⋆]

Matthew Felice Pace and Alexander Tiskin

DIMAP and Department of Computer Science
University of Warwick, Coventry, CV4 7AL, UK
matthewfp,tiskin@dcs.warwick.ac.uk

Abstract. A deterministic BSP algorithm for constructing the suffix array of a given string is presented, based on a technique that we call *accelerated sampling*. It runs in optimal $O(\frac{n}{p})$ local computation and communication, and requires a near optimal $O(\log \log p)$ supersteps. The algorithm provides an improvement over the synchronisation costs of existing algorithms, and reinforces the importance of the sampling technique.

Keywords: BSP, suffix array, accelerated sampling

1 Introduction

Suffix arrays are a fundamental data structure in the string processing field and have been researched extensively since their introduction by Manber and Myers [11,14].

Definition 1. *Given a string $x = x[0] \cdots x[n-1]$ of length $n \geq 1$, defined over an alphabet Σ, the suffix array problem is that of constructing the suffix array $SA_x = SA_x[0] \cdots SA_x[n-1]$ of x, which holds the ordering of all the suffixes $s_i = x[i] \cdots x[n-1]$ of x in ascending lexicographical order; i.e. $SA_x[j] = i$ iff s_i is the j^{th} suffix of x in ascending lexicographical order.*

1.1 Notation, Assumptions and Restrictions

We assume zero-based indexing throughout the paper, and that the set of natural numbers $\mathbb{N}$ includes zero. For any $i, j \in \mathbb{N}$, we use the notation $[i : j]$ to denote the set $\{a \in \mathbb{N} \mid i \leq a \leq j\}$, and $[i : j)$ to denote $\{a \in \mathbb{N} \mid i \leq a < j\}$. This notation is extended to substrings by denoting the substrings of string x, of size n, by $x\,[i : j) = x[i] \cdots x[j-1]$, for $i \in [0 : n)$ and $j > i$.

The input to the algorithms to be presented in this paper is restricted to strings defined over the alphabet $\Sigma = [0 : n)$, where n is the size of the input string. This allows us to use counting sort [3] throughout when sorting characters, in order to keep the running time linear in the size of the input. We also use counting sort in conjunction with the radix sorting technique [3].

The end of any string is assumed to be marked by an end sentinel, typically denoted by '\$', that precedes all the characters in the alphabet order. Therefore, to mark the end of the string and to ensure that any substring $x\,[i : j)$ is well defined, we adopt the padding convention $x[k] = -1$, for $k \geq n$.

⋆ Research supported by the Centre for Discrete Mathematics and its Applications (DIMAP), University of Warwick, EPSRC award EP/D063191/1

Note that the suffix array construction algorithms to be presented in Sections 3 and 5 can also be applied to any string X, of size n, over an indexed alphabet Σ' [14,16], which is defined as follows:

- Σ' is a totally ordered set.
- an array A can be defined, such that, $\forall\sigma \in \Sigma'$, $A[\sigma]$ can be accessed in constant time.
- $|\Sigma'| \leq n$.

Commonly used indexed alphabets include the ASCII alphabet and the DNA bases. It should also be noted that any string X, of size n, over a totally ordered alphabet can be encoded as a string X', of size n, over integers. This is achieved by sorting the characters of X, removing any duplicates, and assigning a rank to each character. The string X' is then constructed, such that it is identical to X except that each character of X is replaced by its rank in the sorted array of characters. However, sorting the characters of X could require $O(n \log n)$ time, depending on the nature of the alphabet over which X is defined.

Let $x_1 \odot x_2$ denote the concatenation of strings x_1 and x_2. Then, for any set of integers A, $\bigodot_{i \in A} x_i$ is the concatenation of the strings indexed by the elements of A, in ascending index order. Throughout the paper we use $|b|$ to denote the size of an array or string b. To omit $\lceil\cdot\rceil$ operations, we assume that real numbers are rounded up to the nearest integer.

1.2 Problem Overview

As previously stated, the suffix array problem is that of constructing the suffix array of a given string. The example in Table 1 shows the suffix array of string X, of size 12, over an indexed alphabet of a subset of the ASCII characters, written as string X' over $\Sigma = [0 : 12)$.

	0	1	2	3	4	5	6	7	8	9	10	11	12
$X =$	a	c	b	a	a	c	e	d	b	b	e	a	\$
$X' =$	0	2	1	0	0	2	4	3	1	1	4	0	−1
$SA_X =$	11	3	0	4	2	8	9	1	5	7	10	6	

Table 1. Suffix array of a string X over an indexed alphabet, written as string X' over $\Sigma = [0 : 12)$

The problem is, by definition, directly related to the sorting problem. In fact, if all the characters of the input string are distinct, then the suffix array is obtained by sorting the strings' characters and returning the indices of the characters in their sorted order. In general, if the characters of the string are not distinct, the naive solution is to radix sort all the suffixes, which runs in $O(n^2)$ time if counting sort is used to sort the characters at each level of the radix sort. However, numerous algorithms exist that improve on this. The first such algorithm was presented by Manber and Myers [11] and required $O(n \log n)$ time. The running time was reduced to $O(n)$ through three separate algorithms by Kärkkäinen and Sanders [4], Kim et al. [7], and Ko and Aluru [8]. A number of other algorithms exist with a higher theoretical worst case running time but faster running time in practice, as discussed in [14]. However, the study of these is beyond the scope of this work.

The idea behind the algorithms having linear theoretical worst case running time is to use recursion as follows:

1. Divide the indices of the input string x into two nonempty disjoint sets. Form strings x' and y' from the characters indexed by the elements of each set. Recursively construct $SA_{x'}$.
2. Use $SA_{x'}$ to construct $SA_{y'}$.
3. Merge $SA_{x'}$ and $SA_{y'}$ to obtain SA_x.

The aim of this paper is to investigate the suffix array problem in the Bulk Synchronous Parallel (BSP) model, on a p processor distributed memory system. As in the sequential setting, the naive solution to the problem is to radix sort all the suffixes of the string. Shi and Shaeffer [15] provide a comparison based parallel sorting algorithm, using a technique known as regular sampling that is then adapted by Chan and Dehne [1] for integer sorting. However, using such a technique to sort the suffixes of a given string of size n leads to a parallel algorithm with $O(\frac{n^2}{p})$ local computation cost, which is clearly inefficient.

Kärkkäinen et al. [5] give a brief overview of a BSP suffix array construction algorithm having optimal $O(\frac{n}{p})$ local computation and communication costs and requiring $O(\log^2 p)$ supersteps. They also present similar algorithms for various computation models including the PRAM model. Kulla and Sanders [9] show that the BSP algorithm presented in [5] requires $O(\log p)$ supersteps and discuss their experimental evaluation of the algorithm.

In this paper we reduce the number of supersteps required to a near optimal $O(\log\log p)$, while keeping the local computation and communication costs optimal. The algorithm is based on a technique that we call *accelerated sampling*. This technique was introduced (without a name) by Tiskin [18] for the parallel selection problem. An accelerated sampling algorithm is a recursive algorithm that samples the data at each level of recursion, changing the sampling frequency at a carefully chosen rate as the algorithm progresses.

1.3 Paper Structure

The rest of the paper is structured as follows. The next section provides an overview of the concept of difference covers. A description of the sequential suffix array construction algorithm presented in [5] is given in Section 3. This is a generalised version of the algorithm of [4], which is known as the DC3 algorithm. An overview of the BSP model is provided in Section 4. In Section 5 we present our parallel suffix array construction algorithm, based on the accelerated sampling technique, building on top of the detailed algorithm description given in Section 3. A detailed description is given since, as opposed to the parallel DC3 algorithms of [5,9], we do not assume a fixed input parameter $v = 3$ in all the levels of recursion, so a more general version of the algorithm is required. A detailed analysis of our proposed parallel suffix array construction algorithm in the BSP model is then presented. The last section offers some concluding views and discusses possible future work.

2 Difference Covers

The suffix array construction algorithms to be presented in this paper make use of the concept of difference covers [2,6,13]. Given a positive integer v, let $\mathbb{Z}_v$ denote the set of integers $[0 : v)$. A set $D \subseteq \mathbb{Z}_v$ can be defined such that for any $z \in \mathbb{Z}_v$, there exist $a, b \in D$ such that $z \equiv a - b \pmod{v}$. Such a set D is known as a *difference*

v	$5\dots13$	$14\dots73$	$74\dots181$	$182\dots337$	$338\dots541$	$\dots$	1024	$\dots$	2048	$\dots$
$\lvert D_v\rvert$	4	10	16	22	28		40		58	

Table 2. Size of the difference cover obtained using the algorithm in [2] for various values of v

cover of $\mathbb{Z}_v$, or *difference cover modulo* v. For example, $\{0,1,2\}$, $\{0,1,3\}$, $\{0,2,3\}$ and $\{1,2,3\}$ are valid difference covers of $\mathbb{Z}_4$ while no other proper subset of $\mathbb{Z}_4$ is.

Colbourn and Ling [2] present a method for obtaining, for any v, a difference cover D of $\mathbb{Z}_v$ in time $O(\sqrt{v})$, where $|D| = 6r+4$, $r = \frac{-36+\sqrt{48+96v}}{48}$. Hence, $|D| \le \sqrt{1.5v}+6$. Note that, in general, for any v and any difference cover D of $\mathbb{Z}_v$, $|D| \ge \frac{1+\sqrt{4v-3}}{2}$, since we must have $|D|(|D|-1)+1 \ge v$. Therefore, the size of the difference cover obtained by using the algorithm in [2] is optimal up to a multiplicative constant.

For technical reasons, discussed in Section 3, the algorithms to be presented in this paper require that $0 \notin D$. This does not represent a restriction since, for any v and difference cover D of $\mathbb{Z}_v$, for all $z \in \mathbb{Z}_v$ the set $D' = \{(d-z) \bmod v \mid d \in D\}$ is also a difference cover of $\mathbb{Z}_v$ (see e.g. [13]).

Furthermore, we require that $|D| < v$. Since the minimum size of a difference cover constructed using the method of [2] is 4, we only use this method for $v \ge 5$. For $v = 3$ and $v = 4$ we use the difference covers $\{1,2\}$ and $\{1,2,3\}$ respectively. Table 2 shows the size of the difference cover obtained using the algorithm of [2] for various values of v.

The following simple lemma is also required to ensure the correctness of the algorithms to be presented.

Lemma 2. [5] *If D is a difference cover of $\mathbb{Z}_v$, and i and j are integers, then there exists $l \in [0:v)$ such that $(i+l) \bmod v$ and $(j+l) \bmod v$ are both in D.*

For any difference cover D of $\mathbb{Z}_v$ and integer $n \ge v$, a *difference cover sample* is defined as $C = \{i \in [0:n) \mid i \bmod v \in D\}$. The index set C is a v-periodic sample of $[0:n)$, as defined in [5]. The fact that difference cover samples are periodic allows them to be used for efficient suffix sorting on a given string.

3 Sequential Algorithm

Kärkkäinen and Sanders [4] present a sequential recursive algorithm that constructs the suffix array of a given string x, of size n, using the difference cover $\{1,2\}$ of $\mathbb{Z}_3$, in time $O(n)$. This algorithm is generally known as the DC3 algorithm. Kärkkäinen et al. [5] then generalise the DC3 algorithm such that the suffix array of x can be constructed using a difference cover D of $\mathbb{Z}_v$, for any arbitrary choice of $v \in [3:n]$, in time $O(vn)$. Clearly, setting $v = 3$ results in a running time of $O(n)$, with a small multiplicative constant. As v approaches n the running time approaches $O(n^2)$, and when $v = n$ the algorithm is simply a complex version of the naive suffix array construction algorithm. However, by initially letting $v = 3$ and increasing the value of v at a carefully chosen rate in every subsequent level of recursion, we can reduce the total number of recursion levels required for the algorithm to terminate, while still keeping the total running time linear in the size of the input string. This technique can be used to decrease the number of supersteps required by the parallel suffix array construction algorithm in the BSP model. This is discussed further in Section 5. The detailed sequential algorithm presented in [5] proceeds as follows:

Algorithm 1. *Sequential Suffix Array Construction*
Parameters: integer n; integer $v \in [3 : n]$.
Input: string $x = x[0] \cdots x[n-1]$ over alphabet $\Sigma = [0 : n)$.
Output: suffix array $SA_x = SA_x[0] \cdots SA_x[n-1]$.
Description:
Recursion base

We sort x using counting sort, in time $O(n)$. If all the characters of x are distinct we return, for each character, in the sorted order, the index of the character in x, i.e. SA_x. Otherwise, the following stages are performed:

Stage 0 - Sample construction and initialisation

Construct the difference cover D of $\mathbb{Z}_v$ as discussed in Section 2. Then, for each $k \in [0 : v)$, define the set $B_k = \{i \in [0 : n) \mid i \bmod v = k\}$. This partitions the set of indices of x into v sets of size about $\frac{n}{v}$. The difference cover sample $C = \bigcup_{k \in D} B_k$ is then constructed. For $i \in C$, we call the characters $x[i]$ *sample characters* and the suffixes s_i *sample suffixes*. We denote by S_k, $k \in [0 : v)$, the set of suffixes s_i, $i \in B_k$.

Furthermore, an array *rank* of size $n + v$ is declared and initialised by $rank[0] = \ldots = rank[n + v - 1] = -1$. This array will be used to store the rank of the sample suffixes of x in the suffix array returned by the recursive call made later in Stage 1. While only $|C|$ elements of *rank* will be used, and in fact a smaller array could be used to hold these values. However, we use a larger array to avoid complex indexing schemes relating elements in *rank* to characters in x.

Stage 1 - Sort the sample suffixes

Let $\overline{\Sigma}$ be an alphabet of super-characters, which are defined to be in 1-1 correspondence with the distinct substrings of x of length v, i.e. super-character $\overline{x\,[i : i+v)}$ corresponds to the substring $x\,[i : i+v)$, for all $i \in C$. Therefore, $\overline{\Sigma} \subseteq (\Sigma \cup \{-1\})^v$. Recall from Section 1 that, due to the padding convention, any substring $x\,[i : j)$ is well-defined, for $i \in [0 : n)$ and $j > i$, and, therefore, any super-character $\overline{x\,[i : j)}$ is also well-defined.

For each $k \in D$, we now define a string of super-characters X_k over $\overline{\Sigma}$, where $X_k = \bigodot_{i \in B_k} \overline{x\,[i : i+v)}$ and $|X_k| = \frac{n}{v}$. Then, we construct the string of super-characters $X = \bigodot_{k \in D} X_k$, with $|X| = |D|\frac{n}{v}$. Note that for each k, the suffixes of X_k correspond to the set of suffixes S_k. The last super-character of X_k ends with one or more -1 sentinel elements, since 0 is not allowed to be in the difference cover. Therefore, each suffix of X corresponds to a different sample suffix of x, followed by one or more -1 sentinel characters followed by other characters that do not affect the lexicographic order of the suffixes of X. Note that, if 0 was allowed in the difference cover and n was a multiple of v, then the last super-character of X_k would not end with -1.

Recall from Section 1 that since the input to the algorithm is a string over integers, the string of super-characters X can be encoded as string X' over $\Sigma' = [0 : |X|)$ using radix sorting, in time $O(v|X|)$, where $|X'| = |X| = |D|\frac{n}{v}$. The order of the suffixes of X, i.e. the suffix array of X, can then be found by recursively calling the algorithm on the string X' over Σ', with parameters $|X'|$ and v', where v' can be chosen arbitrarily from the range $\left[3 : \min\left((1-\epsilon)\frac{v^2}{|D|}, |X'|\right)\right]$, for some fixed $\epsilon > 0$. Thus, v' becomes the value of v in the subsequent recursion level. The bound $v' \leq (1-\epsilon)\frac{v^2}{|D|}$ follows from the fact that we want the work done in the current level to be greater than the work done in the subsequent recursion level.

Recall from Section 2 that we require $|D| < v$. This ensures that $|X| < n$, so the algorithm is guaranteed to terminate, since each recursive call is always made on a shorter string.

When the recursive call returns with $SA_{X'}$, this holds the ordering of all the suffixes of X', i.e. the rank of the sample suffixes of x within the ordered set of sample suffixes. Then, for $i \in C$, the rank of s_i in $SA_{X'}$ is recorded in $rank[i]$. The order of the sample suffixes within each set S_k, $k \in D$, is also found from $SA_{X'}$.

The total cost of this stage is dominated by the radix sorting procedure required to encode string X into X' over $\Sigma' = [0 : |X|)$, which runs in time $O(|D|n)$.

Note that we can now compare any pair of suffixes by the result of Lemma 2. However, this is not sufficient to sort the suffixes of x in linear time, since each non-sample suffix of x would have to be compared to, possibly, all the other suffixes of x using different values of l. Instead, we perform the following.

Stage 2 - Find the order of the non-sample suffixes within each set S_k**,** $k \in \mathbb{Z}_v \setminus D$

For each $k \in \mathbb{Z}_v \setminus D$, consider any $l_k \in [1 : v)$ such that $(k + l_k) \bmod v \in D$. For every character $x[i]$, $i \in [0 : n) \setminus C$, define the tuple $t_i = (x[i], x[i + 1], \ldots, x[i + l_k - 1], rank[i + l_k])$, where $k = i \mod v$. Note that $rank[i + l_k]$ is defined for each i, since $rank[a]$, for all $a \in C$, has been found in the previous stage and $rank[a] = -1$ for all $a \geq n$.

Then, for each set B_k, $k \in \mathbb{Z}_v \setminus D$, construct the sequence of tuples $(t_i)_{i \in B_k}$. Each of the $v - |D|$ constructed sequences has about $\frac{n}{v}$ tuples, with each tuple having less than v elements. The order of the suffixes within S_k is then obtained by independently sorting every sequence of tuples $(t_i)_{i \in B_k}$, using radix sorting.

The total computation cost of this stage is dominated by the cost of radix sorting all the sequences, i.e. $O\left((v - |D|)\, n\right) = O(vn)$.

Stage 3 - Sort all suffixes by first v ***characters***

Note that in the previous stages the order of every suffix within each set S_k, $k \in [0 : v)$, has been found. Now, let S^α be the set of suffixes starting with α, for $\alpha \in (\Sigma \cup \{-1\})^v$. Then, every set S^α is composed of ordered subsets S_k^α, where $S_k^\alpha = S^\alpha \bigcap S_k$.

All the suffixes s_i, $i \in [0 : n)$, are partitioned into the sets S^α by representing each suffix by the substring $x\,[i : i + v)$, and sorting these substrings using radix sort in time $O(vn)$.

Stage 4 - Merge and complete the suffix ordering

For all $\alpha \in \Sigma^v$, the total order within set S^α can be obtained by merging the subsets S_k^α, $k \in \mathbb{Z}_v$. This comparison-based v-way merging stage uses the fact that all the suffixes in x^α start with the same substring α, in conjunction with Lemma 1. Due to this lemma, a value $l \in [0 : v)$ exists such that for any i, j the comparison of suffixes s_i, s_j only requires the comparison of $rank[i + l]$ and $rank[j + l]$. Having already partitioned the suffixes into sets S^α and found the order of the suffixes within each set S_k, $k \in [0, v)$, the suffix array can be fully constructed through this merging process in time $O(n \log v) = O(vn)$. □

All the stages of the algorithm can be completed in time $O(vn)$, and the recursive call is made on a string of size at most $\frac{4}{5}n$, which corresponds to $|D| = 4$, $v = 5$. Note that a smaller difference cover of $\mathbb{Z}_5$ exists, but as discussed in section 2 we use the algorithm presented in [2] to construct the difference cover of $\mathbb{Z}_v$ for $v \geq 5$. This leads to an overall running time of $O(vn)$.

4 BSP model

The *bulk-synchronous parallel* (BSP) computation model [19,12] was introduced by Valiant in 1990, and has been widely studied ever since. The model was introduced with the aim of bridging the gap between the hardware development of parallel systems and the design of algorithms on such systems, by separating the system processors from the communication network. Crucially, it treats the underlying communication medium as a fully abstract communication network providing point-to-point communication in a strictly synchronous fashion. This allows the model to be architecture independent, promoting the design of scalable and portable parallel algorithms, while also allowing for simplified algorithm cost analysis based on a limited number of parameters.

A BSP machine consists of p processors, each with its local primary and secondary memory, connected together through a communication network that allows for point-to-point communication and is equipped with an efficient barrier synchronisation mechanism. It is assumed that the processors are homogeneous and can perform an elementary operation per unit time. The communication network is able to send and receive a word of data to and from every processor in g time units, i.e. g is the inverse bandwidth of the network. Finally, the machine allows the processors to be synchronised every l time units. The machine is fully specified using only parameters p, g, l, and is denoted by $BSP(p, g, l)$.

An algorithm in the BSP model consists of a series of *supersteps*, or *synchronisation steps*. In a single superstep, each processor performs a number of, possibly overlapping, computation and communication steps in an asynchronous fashion. However, a processor is only allowed to perform operations on data that was available to it at the start of the superstep. Therefore, in a single superstep, a processor can send and receive any amount of data, however, any received data can only be operated on in the following superstep. At the end of a superstep, barrier synchronisation is used to ensure that each processor is finished with all of its computation and data transfer.

The cost of a BSP superstep on a $BSP(p, g, l)$ machine can be computed as follows. Let $work_i$ be the number of elementary operations performed by processor $\pi \in [0 : p)$, in this superstep. Then, the *local computation cost* w of this superstep is given by $w = \max_{\pi\in[0:p)}(work_\pi)$. Let h_π^{out} and h_π^{in} be the maximum number of data units sent and received, respectively, by processor $\pi \in [0 : p)$, in this superstep. Then, the *communication cost* h of this superstep is defined as $h = \max_{\pi\in[0:p)}(h_\pi^{out})+\max_{\pi\in[0:p)}(h_\pi^{in})$. Therefore, the total cost of the superstep is $w + h \cdot g + l$. The total cost of a BSP algorithm with S supersteps, with local computation costs w_s and communication costs h_s, $s \in [0 : S)$, is $W + H \cdot g + S \cdot l$, where $W = \sum_{s=0}^{S-1} w_s$ is the total local computation cost and $H = \sum_{s=0}^{S-1} h_s$ is the total communication cost.

The main principle of efficient BSP algorithm design is the minimisation of the algorithm's parameters W, H, and S. These values typically depend on the number of processors p and the problem size.

5 BSP Algorithm

Along with the sequential suffix array construction algorithm, described in Section 3, Kärkkäinen et al. [5] discuss the design of the algorithm on various computation models, including the BSP model. They give a brief overview of a parallel suffix array construction algorithm, running on a $BSP(p, g, l)$ machine, with optimal $O(\frac{n}{p})$

local computation and communication costs and requiring $O(\log^2 p)$ supersteps. The algorithm is based on the sequential algorithm described in Section 3 with parameter $v = 3$ used in every level of recursion. A number of existing parallel sorting and merging algorithms are used to achieve this result. Kulla and Sanders [9] show that this parallel algorithm actually requires $O(\log p)$ supersteps, implement it on a 64 dual-core processor machine and discuss the obtained results.

The algorithm described in Section 3 initially solves the suffix array problem on a sample of the suffixes of the input string, in order to gain information that is then used to efficiently sort all the suffixes. Sampling techniques are widely used in various fields ranging from statistics to engineering to computer science. A number of parallel algorithms exist that use sampling to efficiently solve problems, such as the sorting [15,1] and convex hull [17] algorithms. In [18], Tiskin presents a BSP algorithm for the selection problem, in which, not only is the data sampled, but, the sampling rate is increased at a carefully chosen rate in successive levels of recursion. This reduces the number of supersteps required by the parallel selection algorithm from the previous upper bound of $O(\log p)$ to a near optimal $O(\log \log p)$, while keeping the local computation and communication costs optimal.

In this section we make use of this technique, which we call *accelerated sampling*, to achieve the same synchronisation costs for our parallel suffix array construction algorithm, while, again, keeping the local computation and communication costs optimal. In contrast with [18], in our algorithm the sampling frequency has to be decreased, rather than increased, in successive levels of recursion. This is achieved by increasing the parameter v in successive levels of recursion. Since, as opposed to the previous work in [5,9], the presented algorithm does not assume a fixed parameter $v = 3$ the algorithm is described in great detail in order to cater for this generality, building on the description given in Section 3.

The algorithms to be presented in this section are designed to run on a $BSP(p, g, l)$ machine. We denote the sub-array of an array a assigned to processor $\pi \in [0 : p)$ by a_π and extend this notation to sets, i.e. we denote by A_π the subset of a set A assigned to processor π.

In the suffix array construction algorithm to be presented, we make extensive use of the parallel integer stable sorting algorithm introduced in [1]. This algorithm is based on the parallel sorting by regular sampling algorithm [15], but uses radix sorting to locally sort the input, removing the extra cost associated with comparison sorting. Given an array y having m distinct integers, such that each integer is represented by at most κ digits, the algorithm returns all the elements of y sorted in ascending order. Since the presented suffix array construction algorithm runs on strings over $\Sigma = \mathbb{N} \cup \{-1\}$, then we can use the same algorithm, which we refer to as the parallel string sorting algorithm, to sort an array of m strings or tuples, each of fixed length κ. In this case, the algorithm has $O(\kappa \frac{m}{p})$ local computation and communication costs and requires $O(1)$ supersteps.

Algorithm 2. *Parallel String Sorting*
Parameters: integer $m \geq p^3$; integer κ.
Input: array of strings $y = y[0] \cdots y[m-1]$, with each string of size κ over $\Sigma = \mathbb{N} \cup \{-1\}$.
Output: array y ordered in ascending lexicographical order.

Description:

The input array y is assumed to be equally distributed among the p processors, with every processor $\pi \in [0 : p-2]$, assigned the elements $y\left[\frac{m}{p}\pi : \frac{m}{p}(\pi+1)\right)$, and processor $p-1$ assigned elements $y\left[\frac{m}{p}(p-1) : m\right)$. Note that each processor holds $\frac{m}{p}$ elements, except the last processor $p-1$, which may hold fewer elements. We call this type of distribution of elements among the p processors a *block distribution.*

Each processor π first locally sorts sub-array y_π, using radix sorting, and then chooses $p+1$ equally spaced samples from the sorted sub-array, including the minimum and maximum values of y_π. These samples, which we call *primary samples*, are sent to processor 0. Having received $(p+1)p$ primary samples, each of which is a string of length κ, processor 0 locally sorts these samples, using radix sorting, and chooses $p+1$ sub-samples, including the minimum and maximum values of the primary samples. These chosen sub-samples, which we call *secondary samples*, partition the elements of y into p blocks $Y_0, \ldots, Y_{p-1}$. The secondary samples are broadcast to every processor, and each processor π then uses the secondary samples to partition its sub-array y_π into the p sub-blocks $Y_{0,\pi}, \ldots, Y_{p-1,\pi}$. Each processor π collects the sub-blocks $Y_{\pi,\chi}$ from processors $\chi \in [0 : p)$, i.e. all the elements of Y_π, and locally sorts these elements using radix sorting. The array y is now sorted in ascending lexicographic order, however, it might not be equally distributed among the processors, so an extra step is performed to ensure that each processor has $\frac{m}{p}$ elements of the sorted array. Note that each primary and secondary sample also has the index of the sample in y attached to it, so that any ties can be broken. Also note that the size of each block is bounded by $O(p)$ so the partitioning of the elements among the processors is balanced.□

The parallel suffix array construction algorithm presented below requires that the input string x of size n be equally distributed among the p processors, using a block distribution, prior to the algorithm being called. We denote by I_π the subset of the index set $[0 : n)$ that indexes x_π, $\pi \in [0 : p)$, i.e. $I_\pi = \left[\frac{n}{p}\pi : \frac{n}{p}(\pi+1)\right)$, for $\pi \in [0 : p-2]$, and $I_{p-1} = \left[\frac{n}{p}(p-1) : n\right)$. Finally, we use the same indexing for a and a_π, i.e. $a[i] = a_\pi[i]$. The algorithm is initially called on string x of length n, with parameters n and $v = 3$.

Algorithm 3. *Parallel Suffix Array Construction*
Parameters: integer $n \geq p^4$; integer $v \in [3 : n]$.
Input: string $x = x[0] \cdots x[n-1]$ over alphabet $\Sigma = [0 : n)$.
Output: suffix array $SA_x = SA_x[0] \cdots SA_x[n-1]$.
Description:
Recursion base

Recall that if all the characters of x are distinct, then SA_x can be obtained by sorting the characters of x in ascending order. Therefore, we call Algorithm 2 on string x with parameters $m = n$ and $\kappa = 1$. When the algorithm returns with the sorted array of characters, which we call x', each processor π holds the sub-array x'_π, of size $\frac{n}{p}$, and checks for character uniqueness in its sub-array. If all the characters in each sub-array are distinct, then, each processor $\pi \in [0 : p-2]$, checks with its neighbour $\pi+1$ to ensure that $x'[\frac{n}{p}(\pi+1)-1] \neq x'[\frac{n}{p}(\pi+1)]$. If every character is distinct then each character in the sorted array x' is replaced by its index in x and x' is returned. However, if at any point in this process identical characters are found, then the following stages are performed:

Stage 0 - Sample construction and initialisation

Every processor π, constructs the difference cover D of $\mathbb{Z}_v$ as discussed in Section 2. Then, each processor π, for each $k \in [0 : v)$, defines the subset $B_{k\pi} = \{i \in I_\pi \mid i \bmod v = k\}$. This partitions each set of indices B_k into p subsets of size about $\frac{n}{pv}$. The subset C_π of the difference cover sample C is then constructed by every processor π, such that $C_\pi = \cup_{k \in D} B_{k\pi}$. We denote by $S_{k\pi}$, $k \in [0 : v)$ and $\pi \in [0 : p)$, the set of suffixes s_i, $i \in B_{k\pi}$.

To ensure that each processor is able to locally construct its subset of super-characters in the next stage, we require that every processor $\pi \in [1 : p)$ sends the first $v - 1$ characters of x_π to processor $\pi - 1$.

Finally, every processor π also declares the array $rank_\pi$, of size $\frac{n}{p} + v$ for $\pi \in [0 : p - 2]$, and size $n - \frac{n}{p}(p-1) + v$ for $\pi = p-1$. Each element of $rank_\pi$ is initialised by -1. Note that the size of each $rank_\pi$, $\pi \in [0 : p - 2]$, allows each processor to store a copy of the first v elements of $rank_{\pi+1}$ in order to be able to locally construct the tuples associated with all the non-sample characters in x_π.

Stage 1 - Sort the sample suffixes

For every processor π, we define, for each $k \in D$, the substring of super-characters $X_{k\pi} = \bigodot_{i \in B_{k\pi}} \overline{x\,[i : i + v)}$, such that the size of $X_{k\pi}$ is about $\frac{n}{pv}$. Note that every substring $x\,[i : i + v)$ is locally available for all $i \in C_\pi$, due to the padding convention and the distribution of x among the processors. Then, construct the string of super-characters X, as discussed in Section 3. This string is distributed among the p processors using a block distribution, with each processor having around $|D|\frac{n}{pv}$ super-characters. Note that it is not necessary to actually construct X, since the position of each $X_{k\pi}$, and, therefore, the index of each super-character $\overline{x\,[i : i + v)}$, $i \in C$, in X can be calculated by every processor π. However, this is done for simplicity. Algorithm 2 is then called on string X with parameters $m = |D|\frac{n}{v}$ and $\kappa = v$. After sorting, a rank is assigned to each super-character in its sorted order, with any identical super-characters given the same rank, and the string X' is constructed as discussed in Section 3. Note that X' is already equally distributed among the processors.

The algorithm is then called recursively on the string X' with parameters $n = |X'|$ and $v' = v^{5/4}$, where v' is the value of v in the subsequent recursion level. If $|X'| \leq \frac{n}{p}$, then X' is sent to processor 0, which calls the sequential suffix array algorithm on X' with parameters $n = |X'|$ and $v = 3$. A detailed discussion on the assignment $v' = v^{5/4}$ and its impact on the synchronisation costs of the algorithm is given later in this section.

When the recursive call returns with $SA_{X'}$, the rank of each s_i in $SA_{X'}$, $i \in C_{k\pi}$, $\pi \in [0 : p)$, is recorded in $rank_\pi$. Also, a copy of the first v elements of $rank_\pi$, for $\pi \in [1 : p)$, is kept in $rank_{\pi-1}$. The order of each suffix s_i within each set S_k, $k \in D$, is stored by each processor π, for $i \in I_\pi$.

Stage 2 - Find the order of the non-sample suffixes within each set S_k, $k \in \mathbb{Z}_v \setminus D$

For each $k \in \mathbb{Z}_v \setminus D$, consider any $l_k \in [1 : v)$ such that $(k + l_k) \bmod v \in D$. We define the tuple $t_i = (x[i], x[i + 1], \ldots, x[i + l_k - 1], rank[i + l_k])$, for each character $x[i]$, $i \in I_\pi \setminus C_\pi$, $\pi \in [0 : p)$ and $k = i \mod v$. Note that every character in the tuple t_i, $i \in I_\pi$, is locally available on processor π.

Then, every processor $\pi \in [0 : p)$ constructs the subsequence of tuples $(t_i)_{i \in B_{k\pi}}$, for each subset $B_{k\pi}$, $k \in \mathbb{Z}_v \setminus D$. Therefore, each sequence $(t_i)_{i \in B_k}$ is the concatenation of the subsequences $(t_i)_{i \in B_{k\pi}}$ in ascending order of π. Recall from Section 3, that the number of sequences $(t_i)_{i \in B_k}$ to be sorted is $v - |D|$, and that each sequence contains $\frac{n}{v}$ tuples, of length at most v; i.e. each processor holds about $\frac{n}{vp}$ tuples of each sequence.

Each sequence is then sorted independently using Algorithm 2 with parameters $m = \frac{n}{v}$ and κ being the length of the tuples in the sequence, which is at most v. After each sequence is sorted, the order of each non-sample suffix s_i, $i \in I_\pi$, within each set S_k, $k \in \mathbb{Z}_v \setminus D$, is stored by each processor π.

Stage 3 - Sort all suffixes by first v characters

Let each suffix s_i, $i \in [0:n)$, of x be represented by the substrings $x[i : i+v)$. These substrings are stably sorted using Algorithm 2 with parameters $m = n$ and $\kappa = v$. After sorting, the suffixes of x will have been partitioned into the sets S^α, $\alpha \in (\Sigma \cup \{-1\})^v$, as discussed in Section 3.

Stage 4 - Merge and complete the suffix ordering

Recall from Section 3 that each set S^α, $\alpha \in (\Sigma \cup \{-1\})^v$, is partitioned into at most v subsets S^α_k, $k \in [0:v)$, and that the order of the suffixes within each such subset has been found in the previous stages. Ordering a set S^α is achieved through a v-way merging procedure based on Lemma 2. For every two subsets $S^\alpha_{k'}$ and $S^\alpha_{k''}$, $k', k'' \in [0:v)$, we choose any $l \in [0:v)$ such that $(k'+l) \bmod v$ and $(k''+l) \bmod v$ are both in D. Then, comparing two suffixes $s_i \in S_{k'}$ and $s_j \in S_{k''}$ only requires the comparison of $rank[i+l]$ and $rank[j+l]$.

Therefore, in order to sort S^α we require, for each element of S^α, the rank of the element within the sorted subset S_k it belongs to and at most $|D|$ values from the array $rank$. Hence, at most $(|D|+1)\frac{n}{p}$ values need to be received by each processor. Note that the rank of each suffix s_i, $i \in [0:n)$, within the set S_k, $i \bmod v = k$, is stored on processor π, $i \in I_\pi$, as is $rank[i+l]$, for all $l \in [0:v)$.

After the sorting procedure in the previous stage, the suffixes of a set S^α, $\alpha \in \Sigma^v$, are contiguous and can be either contained within a single processor, or span two or more processors. If S^α is contained within one processor, then all the subsets of S^α are locally merged. If the set spans two processors $\pi', \pi'' \in [0:p)$, then, for each of the suffixes $s_i \in S^\alpha$, $i \in [0:n)$, on processor π'', the values required to merge the suffixes into the ordered set S^α are sent to processor π', which then constructs the order set. Otherwise, if S^α spans more than two processors, the following procedure, based on the parallel sorting by regular sampling technique, is performed.

The set S^α is block distributed among the p processors. Again, note that the actual suffixes $s_i \in S^\alpha$, $i \in [0:n)$, are not communicated, but only the values required by the merging process are, i.e. at most $|D|+1$ values for each suffix in S^α. Each processor locally sorts its assigned elements of S^α, using the v-way merging procedure, and chooses $p+1$ equally spaced primary samples from the sorted elements, including the minimum and maximum elements. Every primary sample is sent to one of the p processors that is chosen as the designated processor. Therefore, this designated processor receives $(p+1)p$ primary samples, which it sorts locally using the v-way merging procedure. It then chooses $p+1$ equally spaced secondary samples from the merged primary samples, including the minimum and maximum primary samples, that partition S^α into p blocks. These secondary samples are broadcast to the p processors such that each processor can partition its assigned elements into p sub-blocks. Every processor then collects all the sub-blocks that make up a unique block and locally merges the received elements. Finally, send the ordered set S^α back to the processors it originally spanned. Note that the size of each block is bounded by $O(p)$ so the partitioning of the elements among the processors is balanced.

Round i	v_i	$\lvert D_i\rvert$	n_i	Total Work = $O(v_i \cdot n_i)$
0	v	$O\left(v^{\frac{1}{2}}\right)$	n	$O\left(v \cdot n\right)$
1	$v^{\frac{5}{4}}$	$O\left(\left(v^{\frac{5}{4}}\right)^{\frac{1}{2}}\right)$	$O\left(v^{\frac{1}{2}} \cdot v^{-1} \cdot n\right) = O\left(v^{-\frac{1}{2}} \cdot n\right)$	$O\left(v^{\frac{3}{4}} \cdot n\right)$
2	$v^{\left(\frac{5}{4}\right)^2}$	$O\left((v^{\left(\frac{5}{4}\right)^2})^{\frac{1}{2}}\right)$	$O\left(\left(v^{\left(\frac{5}{4}\right)}\right)^{\frac{1}{2}} \cdot v^{-\frac{5}{4}} \cdot v^{-\frac{1}{2}} \cdot n\right) = O\left(v^{-\frac{9}{8}} \cdot n\right)$	$O\left(v^{\frac{7}{16}} \cdot n\right)$
i	$v^{\left(\frac{5}{4}\right)^i}$	$O\left(v^{\left(\frac{5}{4}\right)^i\left(\frac{1}{2}\right)}\right)$	$O\left(v^{-2\left(\frac{5}{4}\right)^i+2} \cdot n\right)$	$O\left(v^{-\left(\frac{5}{4}\right)^i+2} \cdot n\right)$
$\log_{\frac{5}{4}}\left(\log_v p^{\frac{1}{2}}+1\right)$	$v \cdot p^{\frac{1}{2}}$	$O\left(v^{\frac{1}{2}} \cdot p^{\frac{1}{4}}\right)$	$O\left(p^{-1} \cdot n\right)$	$O\left(v \cdot p^{-\frac{1}{2}} \cdot n\right)$

Table 3. Algorithm analysis

After all the sets S^α have been sorted, all the suffixes of x have been ordered and the suffix array is returned. $\square$

5.1 Algorithmic Analysis

The presented suffix array construction algorithms are recursive, and the number of levels of recursion required for the algorithms to terminate depends on the factor by which the size of the input string is reduced in successive recursive calls. While the number of levels of recursion does not influence the running time of the sequential algorithm, in BSP this determines the synchronisation costs of the algorithm, and, therefore, we want to reduce it to a minimum. Before detailing the costs of each stage of the algorithm we explain how changing the sample size at each subsequent level of recursion results in $O(\log\log p)$ levels of recursion.

We refer to the i^{th} level of recursion of the algorithm as round i, $i \geq 0$. Then, we denote by n_i, v_i and D_i the size of the input string, the parameter v and the difference cover D of $\mathbb{Z}_{v_i}$, respectively, in round i.

Recall from Section 2 that the maximum size of a difference cover D of $\mathbb{Z}_v$, for any positive integer v, that can be found in time $O(\sqrt{v})$ is $\sqrt{1.5v}+6$, i.e. $|D| = O(v^{1/2})$. Therefore, for the sake of simplicity, in our cost analysis we assume that $|D_i| = O\left({v_i}^{1/2}\right)$.

Changing the parameter v in successive recursive calls affects the sampling rate and the size of the input string. Let v_0 and n_0 be the parameters v and n given in the initial call to the algorithm, while $|D_0| = O\left({v_0}^{1/2}\right)$. Then, in round $i \geq 1$, $v_i = {v_{i-1}}^{5/4}$, $|D_i| = O\left(|D_i|^{5/4}\right)$ and $n_i = n_{i-1}v^{(-1/2)(5/4)^{i-1}}$. Note that $n_i = nv^{\sum_{k=1}^{i} -1/2(5/4)^{k-1}}$, i.e. the exponent of the term v is a geometric series with $a = -\frac{1}{2}$ and $r = \frac{5}{4}$. The analysis given in Table 3 illustrates how these values change in successive recursion levels. Recall from Section 3 that, the cost of each level of recursion in the sequential algorithm is $O(v_i n_i)$. Therefore, the table also shows that the order of work done decreases in subsequent recursive levels.

The results in Table 3 clearly show that if the algorithm is initially called on a string of size n, with parameter $v = 3$, on a $BSP(p, g, l)$ machine, then the size of the input converges towards $\frac{n}{p}$ super-exponentially. In fact, after $\log_{5/4}(\log_3 p^{1/2} + 1) = O(\log\log p)$ levels of recursion, the size of the input string is $O(\frac{n}{p})$, and in the subsequent level of recursion the suffix array is computed sequentially on processor 0. Note that the value $\frac{5}{4}$ as a power of v is not the only one possible. In fact, any value $1 < a < \frac{3}{2}$ can be used, but $a = \frac{5}{4}$ is used for simplicity.

Finally, recall that in Section 3 we require $v_i \leq n_i$ for the sequential algorithm. However, in our parallel algorithm, since we require that $n_0 > p^4$ and we set $v_0 = 3$, this will always be the case in the first $O(\log\log p)$ levels of recursion, at which point the algorithm is called sequentially on a single processor. Therefore, this bound is not required in our parallel algorithm.

Having determined the number of recursive calls required by the algorithm, the cost of each stage is now analysed. In the recursion base, the costs are dominated by those of Algorithm 2, i.e. $O(\frac{n_i}{p})$ local computation and communication cost.

In stage 0, constructing the difference cover D_i has local computation costs $O(\sqrt{v_i})$, while constructing the subsets C_π, independently for each processor $\pi \in [0 : p)$, has $O(|D_i|\frac{n_i}{pv_i})$ local computation cost. Passing the first $v-1$ characters of x_π, $\pi \in [1 : p)$, from processor π to $\pi - 1$ has $O(v)$ local computation and communication costs. Finally, initialising $rank_\pi$ requires $O(\frac{n_i}{p} + v_i)$ work. Only $O(1)$ supersteps are required.

In stage 1, the costs are dominated by the construction of the string of super-characters X and the call to Algorithm 2, leading to $O(|D_i|\frac{n_i}{p})$ local computation and communication costs and requires $O(1)$ supersteps.

In stage 2, the costs are again dominated by the call to Algorithm 2 for each sequence of tuples. The number of sequences to be sorted is $v_i - |D_i|$, which is always less than p. Therefore, we can use a different designated processor for each call to Algorithm 2. The size of each sequence is $\frac{n_i}{v_i}$, and the size of each tuple is at most v_i. Therefore, each processor has $O(\frac{n_i}{v_i p})$ tuples from each sequence, i.e. $O(\frac{n_i}{p})$ tuples in total. Therefore, sorting all these tuples independently for each sequence has $O(v_i\frac{n_i}{p})$ local computation and communication costs and requires $O(1)$ supersteps.

The cost of stage 3 is simply the cost of Algorithm 2 on a string of size n_i with $\kappa = v_i$, i.e. $O(v_i\frac{n_i}{p})$ local computation and communication costs and $O(1)$ supersteps.

In stage 4, obtaining, for each suffix of x, the information required to sort each set S^α using a v-way merging procedure has $O(|D_i|\frac{n_i}{p})$ local computation and communication costs. Then, sorting a set S^α that is contained on a single processor has $O(|S^\alpha|v_i)$ local computation costs, and no communication is required. Note that in this case $|S^\alpha| < \frac{n_i}{p}$. If S^α spans two processors, then we send all the elements of the set to one of the two processors. Therefore, since each processor has $\frac{n_i}{p}$ suffixes, then, $2 \leq |S^\alpha| \leq 2\frac{n_i}{p}$, and the costs of sorting this set are $O(v_i\frac{n_i}{p})$ local computation, $O(|D_i|\frac{n_i}{p})$ communication and $O(1)$ supersteps.

Finally, if a set S^α spans more than 2 processors, then $|S^\alpha| > \frac{n_i}{p}$. Therefore, the number of such sets is less than p. In this case a technique based on parallel sorting by regular sampling on p processors is performed, choosing a different designated processor for each such set S^α. In fact, the only difference between the two techniques is that v-way merging is used, instead of radix sorting, to locally sort the suffixes. Since the v-way merging procedure on n elements has the same asymptotic costs as the radix sorting procedure on an array of n strings each of size v, over an alphabet $\Sigma = \mathbb{N} \cup \{-1\}$, then the local computation cost for this procedure is $O(v_i\frac{n_i}{p})$ and the communication cost is $O(|D_i|\frac{n_i}{p})$. Since each such set can be merged independently in parallel, then a constant number supersteps is required. Note that we could merge any set spanning $p' > 2$ processors on the p' processors instead of distributing the set across all the p processors, however we choose not to do this for the sake of simplicity.

In the i^{th} level of recursion, each stage has $O(v_i\frac{n_i}{p})$ local computation and communication costs and requires $O(1)$ supersteps. The presented parallel suffix array construction algorithm is initially called on a string of size n with parameter $v = 3$, so the local computation and communication costs are $O(\frac{n}{p})$ in round 0. In round $\log_{5/4}(\log_3 p^{1/2} + 1)$ these costs are $O(\frac{n}{p^{3/2}})$. Note that after this round the algorithm is called sequentially on a string of length less than $\frac{n}{p}$. Also note that, as shown in table 3, $O(v_i n_i)$ decreases super-exponentially in each successive level of recursion, and, therefore, the order of work done in each such level also decreases super-exponentially. Therefore, the algorithm has $O(\frac{n}{p})$ local computation and communication costs and requires $O(\log \log p)$ supersteps.

Finally, recall that Algorithm 2 requires slackness, $m \geq p^3$. Since in the critical round Algorithm 2 is called in stage 2 on $O(p^{\frac{1}{2}})$ sequences of size $\frac{n}{p^{3/2}}$, we require that $n \geq p^4$. Note that this slackness can be reduced by sorting the sequences locally if each sequence fits on a separate processor, however, such detail is beyond the scope of this paper and will be given in a full version of this paper.

6 Conclusion

In this paper we have presented a deterministic BSP algorithm for the construction of the suffix array of a given string. The algorithm runs in optimal $O(\frac{n}{p})$ local computation and communication, and requires a near optimal $O(\log \log p)$ supersteps.

The method of regular sampling has been used to solve the sorting [15,1], and 2D and 3D convex hulls [17] problems. Random sampling has been used to solve the maximal matching problem and provide an approximation to the minimum cut problem [10] in a parallel context. An extension of the regular sampling technique, which we call *accelerated sampling*, was introduced by Tiskin [18] to improve the synchronisation upper bound of the BSP algorithm for the selection problem. The same technique was used here to improve the synchronisation upper bounds of the suffix array problem. Accelerated sampling is a theoretically interesting technique, allowing, in specific cases, for an exponential factor improvement in the number of supersteps required over existing algorithms.

It is still an open question whether the synchronisation cost of the suffix array problem and the selection problem can be reduced to the optimal $O(1)$ while still having optimal local computation and communication costs. Another open question is whether further applications of the sampling technique, whether regular, random or accelerated, are possible.

References

1. A. Chan and F. Dehne: *A note on coarse grained parallel integer sorting.* Parallel Processing Letters, 9(4) 1999, pp. 533–538.
2. C. J. Colbourn and A. C. H. Ling: *Quorums from difference covers.* Information Processing Letters, 75(1-2) July 2000, pp. 9–12.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein: *Introduction to Algorithms*, MIT Press, 3rd ed., 2009.
4. J. Kärkkäinen and P. Sanders: *Simple linear work suffix array construction*, in Proceedings of the 30th International Conference on Automata, Languages and Programming, 2003, pp. 943–955.
5. J. Kärkkäinen, P. Sanders, and S. Burkhardt: *Linear work suffix array construction.* Journal of the ACM, 53(6) Nov. 2006, pp. 918–936.

6. J. Kilian, S. Kipnis, and C. E. Leiserson: *The organization of permutation architectures with bused interconnections.* IEEE Transactions on Computers, 39(11) November 1990, pp. 1346–1358.
7. D. K. Kim, J. S. Sim, H. Park, and K. Park: *Linear-time construction of suffix arrays*, in Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching, 2003, pp. 186–199.
8. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*, in Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching, 2003, pp. 200–210.
9. F. Kulla and P. Sanders: *Scalable parallel suffix array construction.* Parallel Computing, 33(9) 2007, pp. 605–612.
10. S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii: *Filtering: A method for solving graph problems in MapReduce*, in Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures, 2011, pp. 85–94.
11. U. Manber and G. Myers: *Suffix arrays: A new method for on-line string searches*, in Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 319–327.
12. W. McColl: *Scalable computing*, in Computer Science Today, J. van Leeuwen, ed., vol. 1000 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 1995, pp. 46–61.
13. C. Mereghetti and B. Palano: *The complexity of minimum difference cover.* Journal of Discrete Algorithms, 4(2) June 2006, pp. 239–254.
14. S. J. Puglisi, W. F. Smyth, and A. H. Turpin: *A taxonomy of suffix array construction algorithms.* ACM Computing Surveys, 39(2) July 2007.
15. H. Shi and J. Schaeffer: *Parallel sorting by regular sampling.* Journal of Parallel Distributed Computing, 14(4) Apr. 1992, pp. 361–372.
16. W. F. Smyth: *Computing Patterns in Strings*, Addison-Wesley, April 2003.
17. A. Tiskin: *Parallel convex hull computation by generalised regular sampling*, in Proceedings of the 8th International Euro-Par Conference on Parallel Processing, 2002, pp. 392–399.
18. A. Tiskin: *Parallel selection by regular sampling*, in Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II, 2010, pp. 393–399.
19. L. G. Valiant: *A bridging model for parallel computation.* Communications of the ACM, 33 August 1990, pp. 103–111.

Compact Complete Inverted Files for Texts and Directed Acyclic Graphs Based on Sequence Binary Decision Diagrams

Shuhei Denzumi[1], Koji Tsuda[2,3], Hiroki Arimura[1], and Shin-ichi Minato[1,3]

[1] Graduate School of IST, Hokkaido University, Sapporo, Japan
[2] AIST Computational Biology Research Center, Tokyo, Japan
[3] JST ERATO MINATO Discrete Structure Manipulation System Project, Sapporo, Japan
{denzumi, arim, minato}@ist.hokudai.ac.jp, koji.tsuda@aist.go.jp

Abstract. A complete inverted file is an abstract data type that provides functions for text retrieval. Using it, we can retrieve frequencies and occurrences of strings for given texts. There have been various complete inverted files for texts. However, complete inverted files for graphs have not been studied. In this paper, we define complete inverted files based on sequence binary decision diagrams (SDD) for directed acyclic graphs (DAG). Directed acyclic graphs are given as sequence binary decision diagrams. We propose new complete inverted files called PosFSDD and PosFSDDdag for a text and a DAG, respectively. We also present algorithms to construct them and to retrieve occurrence information from them. Computational experiments are executed to show the efficiency of PosFSDDs.

1 Introduction

Recent emergence of massive text and sequence data has increased the importance of string processing technologies. In particular, complete inverted files for efficient text retrieval and analysis has attracted much attention in many applications such as bioinformatics, natural language processing, and sequence mining. A complete inverted file for a text w is a data structure that stores all factors of w allowing three functions; *find*, *freq*, and *locations*. In many real applications, indices that store occurrence information are highly required. Sequence binary decision diagrams (sequence BDDs or SDDs, for short) are compact representation for manipulating sets of strings, proposed by Loekito, *et al.* [7]. In this paper, we consider the problem of constructing a complete inverted file on SDD framework. We define complete inverted files on SDDs, named PosFSDD (See Fig. 2), and propose an algorithm to construct a PosFSDD from an input text. We also define a complete inverted file for a directed acyclic graph (DAG) and present an efficient construction algorithm to construct a PosFSDDdag from an input DAG, which is given as an SDD. There is research on construction factor automata from automata [10]. On the other hand, complete inverted files for graphs have not been studied. We can construct complete inverted files for multiple texts by concatenating them on existing data structures. However, those methods cannot deal with very large number of strings such that DAGs can represent by sharing its subgraphs. For example, regular expressions without infinite loop and human genomes with many replacements can be represented much more compactly by DAGs than by explicit representations. We also show some experimental results for real data. Our method will be useful for wide variety of pattern matching applications and sequence mining.

Shuhei Denzumi, Koji Tsuda, Hiroki Arimura, Shin-ichi Minato: Compact Complete Inverted Files for Texts and Directed Acyclic Graphs Based on Sequence Binary Decision Diagrams, pp. 157–167.

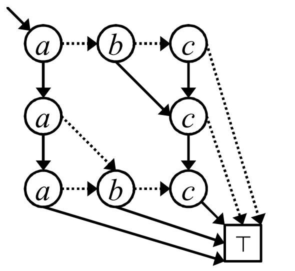


Figure 1. An SDD for the language {ϵ, *aaa*, *aab*, *aac*, *ab*, *ac*, *b*, *bcc*, *c*, *ccc*}. Circles denote nonterminals. Squares denote terminals. The 0-terminal $\bot$ and 0-edges coming to $\bot$ are omitted.

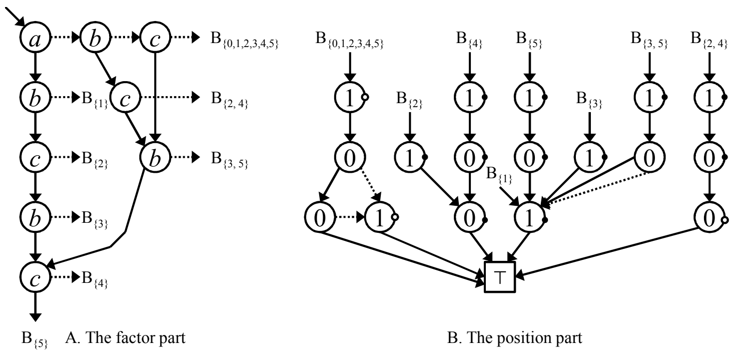


Figure 2. An example of a complete inverted file based on SDD, PosFSDD, for $w = abcbc$. The 0-terminal $\bot$ is omitted. All 0-edges coming to $\bot$ and $\top$ are indicated by a small black dot and white dot on the right side, respectively.

2 Preliminaries

2.1 Strings and string sets

Let $\Sigma = \{a, b, \ldots\}$ be a countable alphabet of symbols, for which the equality $=_\Sigma$ and a strict total order $\prec_\Sigma$, such that $a \prec_\Sigma b \prec_\Sigma \cdots$, are defined on Σ. We often omit the subscript Σ if no confusion arises. A *string* on Σ of length $n \geq 0$ is a sequence $s = a_1 \cdots a_n$ of symbols, where $|s| = n$ is called the *length* of s and for every $i = 1, \ldots, n$, $s[i] = a_i \in \Sigma$ is called the i-th symbol of s for $1 \leq i \leq |s|$.

Let ϵ be the *empty string* of length zero, and Σ^* be the set of *all possibly empty finite strings*. For strings $x = a_1 \cdots a_m$ and $y = b_1 \cdots b_n$, we define the *concatenation* of x and y by $x \cdot y = xy = a_1 \cdots a_m b_1 \cdots b_n$. For any symbol $\alpha \in \Sigma$, let $\alpha \cdot L = \{\alpha\} \cdot L = \{\, \alpha x \mid x \in L \,\}$. We denote the reversed string of x by $x^R = x[|x|] \cdots x[1]$. For a string s, if $s = xyz$ for $x, y, z \in \Sigma^*$, then we call x, y, and z a *prefix*, a *factor*, and a *suffix* of s, respectively. The sets of prefixes, factors, and suffixes of a string s are denoted by $\mathit{Prefix}(s)$, $\mathit{Factor}(s)$, and $\mathit{Suffix}(s)$, respectively. Given a set S of strings, let the sets of prefixes, factors, and suffixes of the strings in S be denoted by $\mathit{PREFIX}(S)$, $\mathit{FACTOR}(S)$, and $\mathit{SUFFIX}(S)$, respectively.

For any $x \in \mathit{Factor}(w)$, $\mathit{epos}_w(x)$ denotes the set of all positions in w immediately following the occurrences of x and $\mathit{bpos}_w(x)$ denotes the set of all positions immediately preceding occurrences of x. We denote binary representation of an integer i by $\mathit{binstr}(i) \in \{\mathit{0}, \mathit{1}\}^*$ where the leading *0*s are omitted. Therefore, $\mathit{binstr}(0) = \epsilon$. If $\mathit{0}, \mathit{1} \in \Sigma$, $a \prec \mathit{0} \prec \mathit{1}$ for any symbol $a \in \Sigma$.

2.2 Finite Automata

We presume a basic knowledge of the automata theory. For comprehensive introduction to it, see [5,11] for example. A (partial) *deterministic finite automaton DFA* is represented by a quintuple $A = \langle \Sigma, \Gamma, \delta, q_0, F \rangle$, where Σ is the input alphabet, Γ is the *state set*, δ is the *partial transition function* from $\Gamma \times \Sigma$ to Γ, $q_0 \in \Gamma$ is the *initial state* and $F \subseteq \Gamma$ is the set of *acceptance states*. The partial function δ can be regarded as a subset $\delta \subseteq \Gamma \times \Sigma \times \Gamma$. We define the *size* of a DFA A, denoted by $|A|$, as the number of labeled edges in A, i.e., $|A| = |\delta|$.

The set of strings that lead the automaton A from a state q to an acceptance state is denoted by $L_A(q)$. The *language* $L(A)$ accepted by A is $L_A(q_0)$. We say that ADFAs A and A' are *equivalent* if $L(A) = L(A')$. A *minimal DFA* has no state q such that $L_A(q) = \emptyset$ and no distinct states q' and q'' such that $L_A(q') = L_A(q'')$. Since we are concerned with finite languages, all DFAs discussed in this section are *acyclic* DFAs (ADFA, for short).

2.3 Sequence binary decision diagrams

In this subsection, we briefly give a formalization of sequence BDDs, introduced by Loekito, Bailey, and Pei [7], and related concepts for further discussion. Let $\mathsf{dom} = \{u, v, v_1, v_2, \ldots\}$ be a countable set, where each element is called a *node*, and let Σ be a countable alphabet with which a strict total order $\prec_\Sigma$ is associated. A *labeled binary directed acyclic graph* (*labeled binary DAG*) is a directed acyclic graph (DAG) in which every node has out-degree either zero (terminal) or two (non-terminal), where each non-terminal node has a pair of distinguished edges called the 0-edge and the 1-edge . We call the nodes pointed to by the 0- and 1-edges the 0-child and the 1-child, respectively. We define the *subgraph of S rooted at node v* by the connected subgraph of S reachable from v and denote it by $S(v)$.

Roughly speaking, a *sequence binary decision diagram* [7] on Σ is a node-labeled binary DAG that encodes an acyclic DFA on Σ in the *leftmost child and right-sibling* (*LCRS*, for short) *representation* (see, e.g., [1,6]), where the 0-child and 1-child of a non-terminal node correspond to its leftmost child and the right sibling, respectively. Formally, sequence binary decision diagram is defined as follows.

Definition 1. *Let Σ be an alphabet. A sequence binary decision diagram (a* sequence BDD*, for short) is a DAG $S = \langle \Sigma, V, \tau, \bot, \top, \boldsymbol{r} \rangle$ satisfying the following conditions:*

- *$V = V(S) \subseteq \mathsf{dom}$ is a finite set of nodes and every node has unique ID,*
- *$\boldsymbol{r} \in V$ is a distinguished node called the* root *of S.*
- *$\bot$ and $\top \in V$ are distinguished nodes called the* 0- *and* 1-terminal*, respectively. The nodes in $V_N = V \backslash \{\bot, \top\}$ are called* non-terminals.
- *$\tau : V_N \to \Sigma \times V^2$ is the function that assigns to each $v \in V_N$ the triple $\tau(v) = \langle v.lab, v.0, v.1 \rangle$, called the* node triple *for v. Then, the triple indicates that (i) $v.lab \in \Sigma$ is the label of v, (ii) $v.0 \in V$ is the child, called the* 0-child*, that is pointed to by a* 0-edge *from v, and (iii) $v.1 \in V$ is the child, called the* 1-child*, that is pointed to by a* 1-edge *from v.*
- *S must be* acyclic *in its* 0- *and* 1-*edges, that is, there exists some strict partial order $\succ_V$ on V such that for any $v \in V_{\mathrm{N}}$, both of $v \succ_V v.0$ and $v \succ_V v.1$ hold.*
- *S must be* 0-ordered*, that is, for every non-terminal node v, if $v.0$ is a non-terminal node then $v.lab \prec_\Sigma (v.0).lab$ must hold. This means that siblings are deterministically ordered from left to right by $\prec_\Sigma$ on their labels when S is interpreted as an acyclic DFA in the LCRS representation.*

In the figures of this paper, the terminals/nonterminals are denoted by squares/circles, and the 0/1-edges are denoted by dotted/solid lines.

In the above definition, S is said to be *well-defined* if it is both acyclic and 0-ordered. We define the *size* $|S|$ of S by the number of non-terminals in S, i.e., $|S| = |V_{\mathrm{N}}| = |V| - 2$. In the rest of this paper, we often abbreviate a sequence BDD as *SDD* if no confusion arises.

An important class of sequence BDDs is that of reduced sequence BDDs [7], which is a syntactic normal form of SDDs defined as follows.

Definition 2 (reduced SDD [7]). *An sequence BDD is said to be* reduced *if it satisfies the following two conditions:*

- Node-sharing rule*: For any non-terminal nodes $u, v \in V_N$, $\tau(u) = \tau(v)$ implies $u = v$, i.e., no distinct non-terminal nodes have the same triple.*
- Zero-suppress rule*: For any non-terminal nodes $v \in V_N$, $v.1 \neq \bot$ holds, i.e., no non-terminal node has the* 0*-terminal as its* 1*-child.*

The above two rules were originally introduced by Minato [8] for ZDDs [6]. A sequence BDD S defines its language $L(S)$ in the following way. The language of a sequence BDD S is the language assigned to its root $\mathbf{r}$.

Definition 3 (language). *To each node $v \in V$, we inductively assign a language $L_S(v)$ w.r.t. $\succ_V$ as follows: (i) $L_S(\bot) = \emptyset$; (ii) $L_S(\top) = \{\varepsilon\}$; (iii) $L_S(v) = L_S(v.0) \cup (v.lab) \cdot L_S(v.1)$.*

In Fig. 1, we show an example of SDD for the language $\{\epsilon, aaa, aab, aac, ab, ac, b, bcc, c, ccc\}$.

In sequence BDD environment, we can create a new subgraph by combining one or more existing subgraphs in an arbitrary way. As an invariant, all subgraphs are maintained as minimal. In the environment, We use two hash tables *uniqtable* and *cache*, explained below. The first table *uniqtable*, called the unique node table, assigns a nonterminal node $v = \mathit{uniqtable}(c, v_0, v_1)$ to a given triple $t = \langle c, v_0, v_1\rangle$ of a symbol c and a pair of nodes v_0 and v_1. This table is maintained such that it is a function from all triples $t \in \Sigma \times V^2$ to the nonterminal node v in V such that $\tau(v) = t$. If such a node does not exist, *uniqtable* returns *null*. We define a procedure Getnode(c, v_0, v_1) that returns a node with the triple $\langle c, v_0, v_1\rangle$. If there is such a node in V, Getnode returns it. Otherwise, it creates such a node and returns it. The Getnode checks the two reduction rules by using the *uniqtable* to avoid creating duplicated nodes. The second table *cache*, called the operation cache, is used for a user to memorize the invocation pattern "$op(v_1, \ldots, v_k)$" of a user-defined operation *op* and the associated return value $u = op(v_1, \ldots, v_k)$, where each v_i, $i = 1, \ldots, k$ is an existing node in V.

For two given SDDs P and Q, we can compute a SDD R such that R is the language obtained from primitive set operations, union, intersection and difference, on the languages $L(P)$ and $L(Q)$ by recursive algorithms [4]. In addition, concatenation of languages can be computed by Concat in Fig. 3. Using these algorithms, we can construct SDDs for sets of strings of exponential size such as regular expressions without infinite repeats.

2.4 Complete Inverted File

The notion of an inverted file for a textual database is common in the literature on information retrieval, but precise definitions of this concept vary. We use the following definition. Given a finite alphabet Σ, and a text word $w \in \Sigma^*$, a complete inverted file for (Σ, w) is an abstract data type that implements the following functions:

- (1) *find*: $\Sigma^* \to \mathit{Factor}(w)$, where *find*$(x)$ is the longest prefix y of x such that $x \in \mathit{Factor}(w)$ and y occurs in w, that is, $x = yz$, $x, y, z \in \Sigma^*$, and y is a factor of a text w.

```
Global variable: $uniqtable, cache$: hash tables.

Proc Concat($P, Q$: SBDD):
Return: $R$: SDD;
1:  if ($P = \bot$ or $Q = \bot$) return $\bot$;
2:  else if ($P = \top$) return $Q$;
3:  else if ($Q = \top$) return $P$;
4:  else if (($R \leftarrow cache[\text{“}Concat(P, Q)\text{”}]$) exists) return $R$;
5:  else
6:      $\langle x, P_0, P_1 \rangle \leftarrow \tau(P)$;
7:      $R \leftarrow \mathsf{Getnode}(P.lab, \bot, \mathsf{Concat}(P_1, Q))$;
8:      $R \leftarrow \mathsf{Union}(R, \mathsf{Concat}(P_0, Q))$;
9:      $cache[\text{“}Concat(P, Q)\text{”}] \leftarrow R$;
10:     return $R$;
```

Figure 3. An algorithm Concat that constructs the SDD for the language $L(P) \cdot L(Q)$, for given SDDs P and Q.

– (2) *freq*: $Factor(w) \to \mathbb{N}$, where $freq(x)$ is the number of times x occurs as a factor of the text w.
– (3) *locations*: $Factor(w) \to \mathbb{N}^*$, where $locations(x)$ is the set of end positions within the text in which x occurs.

In this paper, we consider the problem of constructing a complete inverted file for a text w. The function $locations(x)$ returns the SDD that represents set of integers $epos_w(x)$ as set of binary strings in our method. We describe SDDs can implement complete inverted files compactly.

Example 4. Let $w = abaababa$ be a given text. Then, $find(baabbaab) = baab$, $freq(ba) = 3$, and $locations(ba) = \{3, 6, 8\}$.

3 Position Factor SDD

We begin with a brief look at some aspects of the factor structure of a fixed, arbitrary word w. In particular, for each factor x of w we will be interested in the set of positions in w at the ends of occurrences of x. We describe the basic data structure used to implement a complete inverted file for a text w based on an SDD.

In our method, occurrence positions are represented as a set of binary strings instead of simple a list of integers. If a factor x occurs at position i, our inverted file stores $x \cdot binstr(i)$. That is, a factor x of w is followed by its occurrence positions in the complete inverted file. Then, we can know the occurrences of x after traversing the path corresponding to x. All equivalent subgraphs are online minimized automatically by always using Getnode when a node with some triple is needed. Therefore, the subgraphs which represent binary strings also share their equivalent subgraphs and become compact.

Definition 5. *Let w be any string. Then, we define two languages.*

– $\mathcal{L}_{epos}(w) = \{x \cdot binstr(k) : x \in Factor(w), k \in epos_w(x)\}$,
– $\mathcal{L}_{bpos}(w) = \{x^R \cdot binstr(k) : x \in Factor(w), k \in bpos_w(x)\}$.

Definition 6. *The Position Factor SDD (PosFSDD) of $w \in \Sigma^*$ is the SDD $F = \langle \Sigma \cup \{0, 1\}, V, \tau, \bot, \top, \boldsymbol{r} \rangle$ such that $L(\boldsymbol{r}) = \mathcal{L}_{epos}$.*

The PosFSDD for $w = abcbc$ is given in Figure 2. Note that the SDDs that represent binary strings play a role analogous to the identification pointers in the compact DAWG [3].

Theorem 7. *Using PosFSDD $F = \langle \Sigma \cup \{0,1\}, V, \tau, \bot, \top, \boldsymbol{r} \rangle$ for a word $w \in \Sigma^*$, for any word $x \in \Sigma^*$, $y = find(x)$ can be determined in time $O(|\Sigma||x|)$. For any $x \in Factor(w)$, $freq(x)$ can be determined in time $O(|\Sigma||x|)$ if* $\mathsf{Card}(\boldsymbol{r})$ *is already executed at least once.*

Proof. To implement *find*, we begin at the root **r** and trace a path corresponding to the letters of x as long as possible. This "search path" is determined and continues until the longest prefix y of x in $Factor(w)$ has been found. To implement *freq*, we note that $freq(x) = |\{z : xz \in \mathcal{L}_{epos}(w)\}| = |epos_w(x)|$ for any $x \in Factor(w)$. The algorithm Card computes the cardinality of the language that each SDD node represents and stores each result in *cache* [6]. So, $freq(x)$ can be obtained by following the procedure of *find* and then returning the result of Card of the node stored in the *cache*. $\mathsf{Card}(\boldsymbol{r})$ is executed in linear time to the input SDD size. Since this node represents the language $M = \{z : xz \in \mathcal{L}_{epos}(w)\}$, we can obtain the node that represents $\{b : b \in M, b \in \{\mathit{0}, \mathit{1}\}^*\}$ by traversing 0-edges until getting a node labeled by *0* or *1*. Clearly all queries are $O(|\Sigma||x|)$. □

Our algorithm to construct $\mathsf{PosFSDD}$ is described in Fig. 4. The union operation is computed in $O(|P||Q|)$ time for two SDDs P and Q [4]. In fig. 5, we shows the algorithm BinSDD(k) that constructs an SDD that represents a binary representation of a natural number k. That is, $L(\mathsf{BinSDD}(k))$ is $\{binstr(k)\}$. We can also construct an SDD for $\mathcal{L}_{bpos}$ with some modification of $\mathsf{BuildPosFSDD}$. That is swapping $|w|$ with 0 in line 1 and line 5, and changing the for loop in line 2 from descending order $|w|, \ldots, 1$ to ascending order $1, \ldots, |w|$.

For a given text w and its factor x, it takes $O(freq(x) \log w)$ time to compute occurrence list of x after obtaining the SDD for $locations(x)$, because occurrences are represented as binary strings and every node has just one label. On the other hand, there are advantages due to SDD representation, especially when $freq(x)$ is large. A list of integers in ordinary representation requires $O(freq(x))$ space and time to examine all positions. By sharing structures, these positions can be represented compactly in our method. As a result, execution times for various operations are improved. For example, for given two factors x and y, finding the positions that both occur within l symbols is computed with some modifications. At first, we construct SDD for $\mathcal{L}'_{epos}(w) = \{x \cdot binstr(k+j) : x \in Factor(w), k \in epos_w(x), 0 \leq j \leq l\}$. Next, obtain the SDDs for $locations(x)$ and $locations(y)$. Then, the positions we want are computed by the intersection operation of these two SDDs.

4 Position FSDD for SDD

We now show our algorithm that constructs a complete inverted file for a directed acyclic graph given as an SDD. First we note that the complete inverted file for an SDD S is defined as follows. In our method, we use node identifiers (IDs) instead of positions for ordinary texts, and factors correspond to paths in the input SDD.

Given an SDD S, a complete inverted file for S for it is an abstract data type that implements the following functions:

```
Global variable: $uniqtable, cache$: hash tables.

Proc BuildPosFSDD($w$: string):
Return: $F$: PosFSDD;
1: $P_{|w|} \leftarrow$ BinSDD($|w|$), $F_{|w|} \leftarrow P_{|w|}$;
2: for $i = |w|, \ldots, 1$
3:     $P_{i-1} \leftarrow$ Getnode($w[i]$, BinSDD($i-1$), $P_i$);
4:     $F_{i-1} \leftarrow$ Union($F_i, P_i$);
5: return $F_0$;
```

Figure 4. An algorithm BuildPosFSDD for constructing the PosFSDD of an input string w.

```
Global variable: $uniqtable, cache$: hash tables.

Proc BinSDD($k$: natural number):
Return: $B$: SDD such that $L(B) = \{binstr(k)\}$;
1: return BinSDD0($k, \lfloor \log_2(k+1) \rfloor$);
Proc BinSDD0($k, l$: natural number):
Return: $B$: SDD that $L(B) = \{l$ length binary string of $k\}$;
2: if ($l = 0$) return $\top$;
3: else if ($B \leftarrow cache[$"$BinSDD(k,l)$"$]$ exists) return $B$;
4: else
5:     if ($k \& (1 << l) \neq 0$) $B \leftarrow$ Getnode($1, \perp$, BinSDD0($k \& ((1 << l) - 1), l - 1$));
6:     else $B \leftarrow$ Getnode($0, \perp$, BinSDD0($k \& ((1 << l) - 1), l - 1$));
7:     $cache[$"$BinSDD(k,l)$"$] \leftarrow B$;
8: return $B$;
```

Figure 5. An algorithm BinSDD for constructing the SDD for $\{binstr(k)\}$. Bitwise AND operation and bit left shift operaton are denoted by & and <<, respectively.

```
Global variable: $uniqtable, cache$: hash tables.

Proc AppendID($P$: SDD):
Return: $R$: SDD such that $L(R) = \{x \cdot binstr(P.ID) : x \in FACTOR(L(Q))$,
$P$ is a SDD node reachable from root via the path corresponding to $x$
 and traversing 0-edges$\}$;
1: if ($P = \perp$) return BinSDD(0);
2: else if ($P = \top$) return BinSDD(1);
3: else if ($R \leftarrow cache[$"$AppendID(P)$"$]$ exists) return $R$;
4: else
5:     $\langle x, P_0, P_1 \rangle \leftarrow \tau(P)$;
6:     $R \leftarrow$ Union(Getnode($x$, AppendID($P_0$), AppendID($P_1$)), BinSDD($P.id$));
7:     $cache[$"$AppendID(P)$"$] \leftarrow R$;
8: return $R$;
```

Figure 6. An algorithm AppendID for constructing the SDD with node IDs by binary strings.

– (1) *find*: $\Sigma^* \rightarrow FACTOR(L(S))$, where *find*$(x)$ is the longest prefix y of x such that $x \in FACTOR(L(S))$ and y occurs in $L(S)$, that is, y is a factor of a string in $L(S)$.

```
Global variable: $uniqtable, cache$: hash tables.

Proc BuildPosFSDDdag($S$: SDD):
Return: $F$: Position FSDDdag for $S$;
 1: return BuildPosFSDDdag0(AppendID($S$));
Proc BuildPosFSDDdag0($P$: SDD):
Return: $G$: SDD such that $L(G) = \{z : z \in SUFFIX(L(P)), z \in \Sigma^+ \cdot \{0,1\}^*\}$;
 1: if ($P = \bot$ or $P = \top$) return $P$;
 2: else if ($G \leftarrow cache[$"BuildPosFSDDdag(P)"$]$ exists) return $G$;
 3: else
 4:     $\langle x, P_0, P_1 \rangle \leftarrow \tau(P)$;
 5:     if ($x \in \{0, 1\}$) return $P$;
 6:     $G \leftarrow$ BuildPosFSDDdag0($P_0$) $\cup$ BuildPosFSDDdag0($P_1$) $\cup$ Getnode($x, \bot, P_1$) ;
 7:     $cache[$"BuildPosFSDDdag(P)"$] \leftarrow G$;
 8: return $G$;
```

Figure 7. An algorithm constructs the PosFSDDdag for the input SDD S. Union operations are denoted by $\cup$.

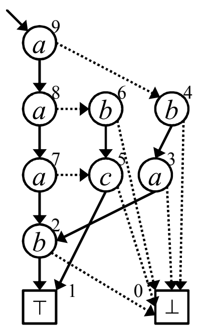


Figure 8. An SDD for $\{aaab, aac, abc, bab\}$. Node IDs are given on the side of each nodes.

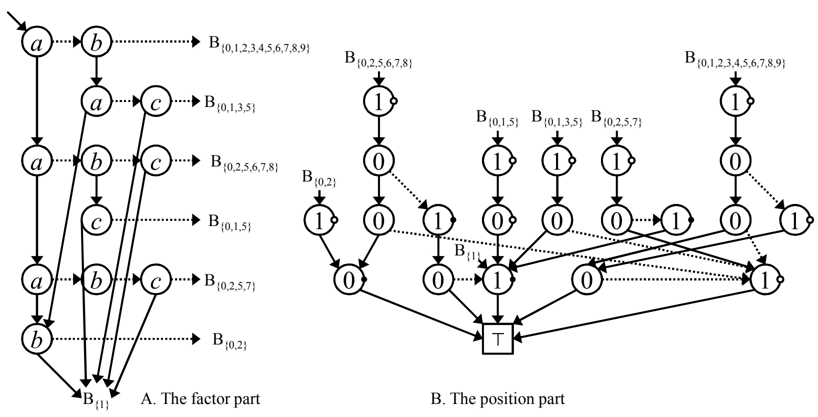


Figure 9. An example of a complete inverted file based on SDD, PosFSDDdag, for the SDD in Fig. 8. The 0-terminal $\bot$ is omitted. All 0-edges incoming to $\bot$ and $\top$ are indicated by a small black dot and white dot on the right side of a node, respectively.

– (2) *freq*: $FACTOR(L(S)) \rightarrow \mathbb{N}$, where $freq(x)$ is the number of nodes reachable by paths corresponding to x that begins from any nodes in S.
– (3) *locations*: $FACTOR(L(S)) \rightarrow \mathbb{N}^*$, where $locations(x)$ is the set of IDs of nodes in S to which paths lead that corresponding to x.

In our method, the set of node IDs that *locations* returns is represented by an SDD for the set of binary strings of the IDs.

Let S be an SDD. For any $x \in FACTOR(L(S))$, $enode_S(x)$ denotes the set of all IDs of nodes in S following the paths corresponding to x and traversing some 0-edges, $bnode_S(x)$ denotes the set of all IDs of nodes in S which represent a language M such that $x \in PREFIX(M)$.

Definition 8. *We define* $\mathcal{L}_{enode}(S) = \{x \cdot binstr(i) : x \in FACTOR(L(S)), i \in enode_S(x)\}$, *and* $\mathcal{L}_{bnode}(S) = \{x^R \cdot binstr(i) : x \in FACTOR(L(S)), i \in bnode_S(x)\}$. *The PosFSDDdag for* S *is the SDD* G *such that* $L(G) = \mathcal{L}_{enode}(S)$.

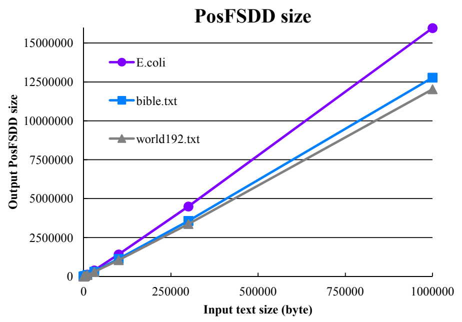


Figure 10. SDD size of PosFSDD with increasing length of input string.

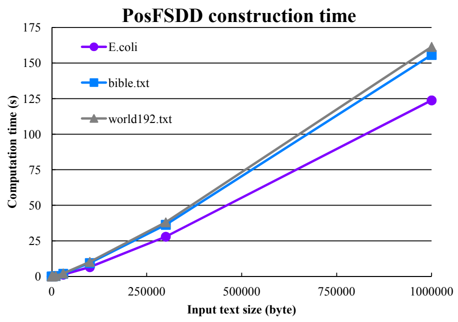


Figure 11. Computation time of BuildPosFSDD with increasing length of input string.

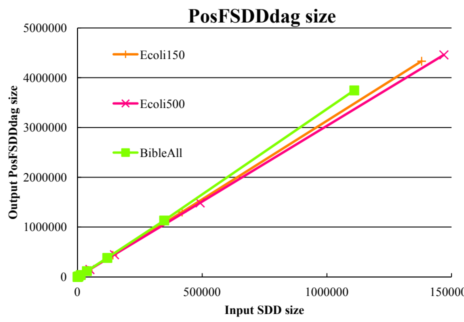


Figure 12. SDD size of PosFSDDdag with increasing input SDD size.

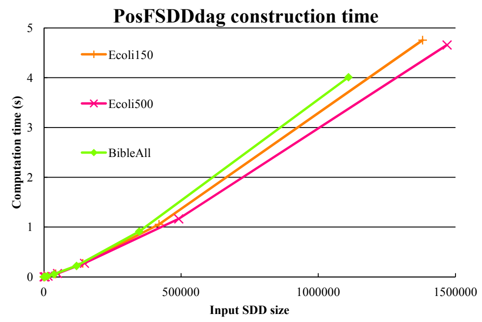


Figure 13. Computation time of BuildPosFSDDdag with increasing input SDD size.

The PosFSDDdag for an SDD S such that $L(S) = \{aaab, aac, abc, bab\}$ is given in Fig. 9, and Fig. 8 shows the input SDD.

Theorem 9. *Using PosFSDDdag G, for any word $x \in \Sigma^*$, $y = find(x)$ can be determined in time $O(|\Sigma||x|)$. For any $x \in FACTOR(L(S))$ can be determined in time $O(|\Sigma||x|)$.*

Proof. We can implement *find*, *freq* and *locations* as in PosFSDD for a text. □

Fig. 7 shows an algorithm to build the PosFSDDdag for an SDD S. The algorithm in Fig. 6 is used for prepocessing of PosFSDDdag. The basic action of the algorithm for an SDD S is to construct the PosFSDDdag for each node recursively, synchronized with the depth-first traversal of S. We can construct reversed version of the PosFSDDdag. It allows for the computation of the exact number of paths corresponding to queries. It also allows for returning the node IDs at which the paths begin. Such an SDD is constructed by executing BuildPosFSDDdag after appliying the algorithm that construct an SDD for reversed $L(S)$, which is proposed by Aoki *et al.* [2].

First, we append SDDs for node IDs to the input by AppendID. Next, we construct reversed SDD of it, but we do not reverse the SDDs that represent node IDs as binary strings. Then, we can construct the SDD for $\mathcal{L}_{bnode}(S)$ by execute BuildPosFSDDdag0 on the obtained SDD.

5 Experimental Results

Setting: In the experiments, we used the following data sets. As real data sets, we used `E.coli`, `bible.txt`, and `world192.txt` obtained from the Canterbury corpus[1].

[1] `http://corpus.canterbury.ac.nz/resources/`

From these data sets, we obtained the following derived data sets: BibleAll is the set of all lines drawn from `bible.txt`. Ecoli150 and Ecoli500 are the set of factors drawn from `E.coli` by cutting the whole sequence at every 150-th or 500-th letter, respectively. We made subsets of these data sets by randomly taking l lines varying $l = 10, 30, 100, \dots$ for BibleAll, Ecoli150, and Ecoli500.

We implemented our shared and reduced SDD environment on the top of the *SAPPORO BDD package* [9] for BDDs and ZDDs written in C and C++, where each node is encoded in a 64-bit integer and a node triple occupies approximately 50 to 55 bytes on average including hash entries in *uniqtable*. We performed experiments on a machine that consists of eight quad-core 3.1 GHz Intel Xeon CPU E7-8837 SE processors (i.e, 32 CPU cores in total) and 1 TB DDR2 memory shared among cores. For PosFSDD and PosFSDDdag construction, we implemented BinSDD, BuildPosFSDD, AppendID, and BuildPosFSDDdag.

Experiment 1: PosFSDD construction. First, Fig. 10, and Fig. 11 show the results. From Fig. 10, we see that PosFSDDs are almost $O(n \log n)$ size for n length text. The number of nodes are between $12n$ to $15n$. As is illustrated in Fig. 11, the proposed BuildPosFSDD runs in $O(n \log n)$.

Experiment 2: PosFSDDdag construction. Fig. 12 demonstrates that the PosFSDDdags are close to linear in the size of the input SDDs. The number of nodes are almost twice as that of the input SDD. As can be seen from Fig.13, BuildPosFS-DDdag runs in almost $O(N \log N)$ time for N sized input SDDs, practically.

6 Conclusions

We proposed PosFSDD that is a complete inverted file for a text based on SDD. We also defined complete inverted files for directed acyclic graphs and implemented it as PosFSDDdag. They allow all queries to be solved in $O(|\Sigma||x|)$ time for n sized input. We gave algorithms that construct PosFSDD and PosFSDDdag. From the experimental results, their sizes are compact and our algorithms BuildPosFSDD and BuildPosFSDDdag run in almost $O(n \log n)$ time. The exact size bound of PosFSDD and the exact time complexity of our algorithms are not obvious. To propose more efficient algorithms is our future work. Position restricted search with PosFSDD is also a challenging problem.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
2. H. Aoki, S. Yamashita, and S. Minato: *An efficient algorithm for constructing a sequence binary decision diagram representing a set of reversed sequences*, in Proceedings of the 2011 IEEE International Conference on Granular Computing (GrC'2011), IEEE, 2011, pp. 54–59.
3. A. Blumer, J. Blumer, D. Haussler, R. M. McConnell, and A. Ehrenfeucht: *Complete inverted files for efficient text retrieval and analysis.* J. ACM, 34(3) 1987, pp. 578–595.
4. S. Denzumi, R. Yoshinaka, H. Arimura, and S. Minato: *Notes on sequence binary decision diagrams: Relationship to acyclic automata and complexities of binary set operations*, in Proceedings of the Prague Stringology Conference 2011 (PSC'11), J. Holub and J. Žďárek, eds., Czech Technical University in Prague, 2011, pp. 147–161.
5. J. E. Hopcroft, R. Motwani, and J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 3rd. ed., 2006.
6. D. E. Knuth: *The Art of Computer Programming, volume 4, fascicle 1, Bitwise Tricks & Techniques; Binary Decision Diagrams*, Addison-Wesley, 2009.

7. E. Loekito, J. Bailey, and J. Pei: *A binary decision diagram based approach for mining frequent subsequences*. Knowledge and Information Systems, 24(2) 2010, pp. 235–268.
8. S. Minato: *Zero-suppressed BDDs and their applications*. Software Tools for Technology Transfer, 3(2) 2001, pp. 156–170.
9. S. Minato: *SAPPORO BDD package*. Division of Computer Science, Hokkaido University, 2011, unreleased.
10. M. Mohri, P. Moreno, and E. Weinstein: *Factor automata of automata and applications*, in Proceedings of the 12th International Conference on Implementation and Application of Automata (CIAA'07), LNCS 4783, Springer, 2007, pp. 168–179.
11. D. Perrin: *Finite automata*, in Handbook of Theoretical Computer Science,, J. van Leuwen, ed., vol. B. Formal Models and Semantics, Elsevier, 1990, pp. 1–57.

Crochemore's String Matching Algorithm: Simplification, Extensions, Applications[⋆]

Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi

Department of Computer Science,
University of Helsinki
Helsinki, Finland
{*firstname.lastname*}@cs.helsinki.fi

Abstract. We address the problem of string matching in the special case where the pattern is very long. First, constant extra space algorithms are desirable with long patterns, and we describe a simplified version of Crochemore's algorithm retaining its linear time complexity and constant extra space usage. Second, long patterns are unlikely to occur in the text at all. Thus we define a generalization of string matching called Longest Prefix Matching that asks for the occurrences of the longest prefix of the pattern occurring in the text at least once, and modify the simplified Crochemore's algorithm to solve this problem. Finally, we define and solve the problem of Sparse Longest Prefix Matching that is useful when the pattern has to be split into multiple pieces because it is too long to be processed in one piece. These problems are motivated by and have application in Lempel-Ziv (LZ77) factorization.

1 Introduction

String matching, the problem of finding all the occurrences of a string $\mathsf{Y}[0..m)$ (the pattern) in a larger string $\mathsf{X}[0..n)$ (the text)[1] is a foundational problem in computer science, and has applications throughout modern computer software. Several algorithms that are optimal in space ($\mathrm{O}(1)$ extra space) as well as in time ($\mathrm{O}(n+m)$) were discovered in the 80's and 90's [11,10,2,3]. In practice, these algorithms, though optimal in theory, are greatly outperformed by algorithms that use $\mathrm{O}(m)$ extra space [5], and thus to date have been mostly a theoretical curiousity. Our own interest in constant extra space algorithms is, however, a practical one: the space-efficient computation of the LZ77 factorization of a string [20]. To our knowledge, this is the first practical application to make use of these optimal string matching techniques, and the first time they have been applied to problems beyond simple pattern matching.

The LZ77 factorization of large strings has many important applications these days, for example in compression [6,12] and indexing [9,8,18] of large text collections (see [16] for more applications of LZ77). The factorization can be computed in linear time but at the cost of using a lot of space [8,15]. We have recently introduced a more space-efficient approach running in $\mathrm{O}(nd)$ time while using $\mathrm{O}(n/d)$ space. The space requirement is in addition to the text when operating in main memory [14], and in total when using external memory [13]. However, very long phrases are a problem for the basic approach and have to be processed differently. There are at most $\mathrm{O}(d)$ of such long phrases and thus we can afford to spend $\mathrm{O}(n)$ time for each. As mentioned in [14,13], the long phrase computation is based on a modified Crochemore's algorithm

⋆ This research is partially supported by the Academy of Finland thorugh grant 118653 (ALGODAN) and grant 250345 (CoECGR).

1 We write $[i..j)$ as a shorthand for $[i..j-1]$.

for string matching but it is never described in more detail. In this paper, we formulate the long phrase computation task as two more general formal problems and show how to solve them by modifying Crochemore's algorithm.

The critical operation in computing the factorization is to find the longest prefix of suffix $\mathsf{X}[i..n)$ that occurs at the some earlier position $j < i$ in X. If the length of this prefix is ℓ, then the next factor will be a prefix of $\mathsf{X}[i+\ell..n)$. If we consider suffix $\mathsf{X}[i..n)$ as a pattern, we can formulate the operation as a special case of the following more general problem.

Definition 1. *Given two strings, a text and a pattern, the* Longest Prefix Matching *problem is to find the length of the longest prefix of the pattern that occurs in the text and to report all occurrences of that prefix in the text.*

From now on we will only consider this general problem. However, consistent with the application in LZ factorization, we focus on the case where the pattern and even the matching prefix is extremely long.

Note that if the pattern as a whole occurs in the text, the output is the occurrences of the pattern. Thus Longest Prefix Matching is a generalization of standard exact string matching. String matching algorithms based on matching pattern prefixes such as Knuth–Morris–Pratt (KMP) [17] can be easily modified to perform Longest Prefix Matching, while others such as Boyer–Moore [1] cannot. However, when the pattern is very long, the space requirement of the data structures built during KMP preprocessing can become a problem. Among the constant extra space algorithms that we are aware of, Crochemore's algorithm [2] is the only one based on matching pattern prefixes. Thus it is the basis of our solution to the Longest Prefix Matching problem. Crochemore's algorithm, and particularly its analysis, is quite complicated. Our first contribution is a simplified version of the algorithm that retains the linear time complexity and constant extra space usage. We then generalize the simple version to solve the Longest Prefix Matching problem in the same time and space complexity.

Even Crochemore's algorithm needs fast access to the full pattern, but in the external memory context the pattern length may even exceed the size of the available memory. To deal with this case, we split the pattern into blocks $\mathsf{Y} = \mathsf{Y}[0..M)\mathsf{Y}[M..2M)\ldots$ that are small enough to fit in memory. We start with longest prefix matching for the first block. If the full block occurs in the text, we then process the second block but considering only occurrences that start where an occurrence of the previous block ends. We continue to process further blocks in the same way as long as necessary. The matching problem for the second and further blocks can be formulated as the following general problem:

Definition 2. *Given two strings, a text and a pattern, and an ascending sequence of text positions, the* Sparse Longest Prefix Matching *problem is to find the length of the longest prefix of the pattern that occurs in the text starting at one of the specified positions and to report all such occurrences.*

We generalize Crochemore's algorithm to solve this problem too.

2 Preliminaries

Strings. Consider a string $\mathsf{X} = \mathsf{X}[0..n-1] = \mathsf{X}[0]\mathsf{X}[1]\cdots\mathsf{X}[n-1]$ of $|\mathsf{X}| = n$ symbols drawn from an ordered alphabet Σ of size σ. For $i = 0, \ldots, n-1$ we write X_i to denote the *suffix* of X of length $n-i$, that is $\mathsf{X}_i = \mathsf{X}[i..n-1] = \mathsf{X}[i]\mathsf{X}[i+1]\cdots\mathsf{X}[n-1]$. The lexicographically maximal among all suffixes of X is denoted $\mathsf{MS}(\mathsf{X})$. By $\mathsf{lcp}(\mathsf{X},\mathsf{Y})$ we denote the length of the longest common prefix of X and Y. A string Y is said to be a *border* of X if Y is both a prefix and a suffix of X. A string is called *border-free* if it has no borders, except itself and the empty string.

Periods. A positive integer p is called a *period* of X if $\mathsf{X}[i] = \mathsf{X}[i+p]$ for any $i \in [0..n-p)$. The shortest period of X is denoted $per(\mathsf{X})$. We say that X is *k-periodic* if $per(\mathsf{X}) \leq |\mathsf{X}|/k$. Throughout we use a classic result about periodicity due to Fine and Wilf [7].

Lemma 1 (Weak Periodicity Lemma) *If a string X has periods p and q that satisfy $p+q \leq |\mathsf{X}|$ then X also has period* $\gcd(p,q)$.

3 Simplified Crochemore's Algorithm

Crochemore's algorithm resembles in many ways the famous Morris-Pratt [19] (MP in short) algorithm[1]. At a generic step it attempts to match the pattern Y against the suffix X_i of the text by computing $\ell = \mathsf{lcp}(\mathsf{X}_i,\mathsf{Y})$ and checking whether $\ell = m$. After that it determines the next position $i+q$ in the text at which the pattern may occur. The value of ℓ is then either set to zero or - if partial information about $\mathsf{lcp}(\mathsf{X}_{i+q},\mathsf{Y})$ is known - to a positive value in order to speed up the next lcp query. Note that any shift length q satisfying $q \leq per(\mathsf{Y}[0..\ell))$ is safe, i.e., prevents from missing an occurrence of Y due to the following fact.

Observation 2 *Assume $\mathsf{X}_i[0..\ell) = \mathsf{Y}[0..\ell)$. Then for any $k \in [1..per(\mathsf{Y}[0..\ell)))$ it holds* $\mathsf{lcp}(\mathsf{X}_{i+k},\mathsf{Y}) = \mathsf{lcp}(\mathsf{Y}_k,\mathsf{Y}) < \ell - k$.

The main difference between MP and Crochemore's algorithm is the choice of shift length q and how it is computed. MP precomputes and stores $per(\mathsf{Y}[0..i))$ for all $i \in [1..m]$, and always sets $q = per(\mathsf{Y}[0..\ell))$ (or $q = 1$ if $\ell = 0$). Crochemore's algorithm uses only O(1) extra space in addition to the text and the pattern (which are treated as read-only) thus cannot afford to store these values. Instead, as the computation of $\mathsf{lcp}(\mathsf{X}_i,\mathsf{Y})$ is taking place, it is simultaneously computing the lexicographically maximal suffix (together with its shortest period) of the growing pattern prefix that matches the text.

Fig. 1 shows an algorithm, called `UpdateMS`, that updates the maximal suffix computation when the prefix match is extended by one character. It is based on properties of maximal suffixes observed by Duval [4] and detailed in the following theorem.

[1] We point out that the original Crochemore's algorithm performs slightly more complicated shifts than MP making it closer to KMP [17] algorithm.

```
Function UpdateMS(Y, $\ell, s, p$)
Input: a string Y and integers $\ell, s, p$ such that
        $\mathsf{MS}(\mathsf{Y}[0..\ell)) = \mathsf{Y}[s..\ell)$ and $p = per(\mathsf{Y}[s..\ell))$.
Output: a triple $(\ell+1, s, p)$ such that
        $\mathsf{MS}(\mathsf{Y}[0..\ell+1)) = \mathsf{Y}[s..\ell+1)$ and $p = per(\mathsf{Y}[s..\ell+1))$.
1:  if $\ell = 0$ then
2:      return $(1, 0, 1)$
3:  $i \leftarrow \ell$
4:  while $i < \ell + 1$ do
        // $\mathsf{MS}(\mathsf{Y}[0..i)) = \mathsf{Y}[s..i)$ and $p = per(\mathsf{Y}[s..i))$
        // $\mathsf{A} = \mathsf{Y}[s..s+p)$ and $\mathsf{B} = \mathsf{Y}[i - (i-s) \bmod p..i)$
5:      if $\mathsf{Y}[i-p] > \mathsf{Y}[i]$ then // Theorem 3, case (3)
6:          $i \leftarrow i - (i-s) \bmod p$
7:          $s \leftarrow i$
8:          $p \leftarrow 1$
9:      elsif $\mathsf{Y}[i-p] < \mathsf{Y}[i]$ then // Theorem 3, case (2)
10:         $p \leftarrow i - s + 1$
11:     $i \leftarrow i + 1$
12: return $(\ell+1, s, p)$
```

Figure 1. A procedure extending the matching pattern prefix by one letter simultaneously updating its maximal suffix and associated shortest period.

Theorem 3 *Let* $\mathsf{Y} = \mathsf{PA}^k\mathsf{B}$ *where* $\mathsf{M} := \mathsf{MS}(\mathsf{Y}) = \mathsf{A}^k\mathsf{B}$ *and* $|\mathsf{B}| < |\mathsf{A}| = p := per(\mathsf{MS}(\mathsf{Y}))$. *Suppose* $a \in \Sigma$ *is such that* $\mathsf{B}a$ *is a prefix of* A *and* b *is an arbitrary character. Then* $\mathsf{M}_b := \mathsf{MS}(\mathsf{Y}b)$ *and* $p_b := per(\mathsf{M}_b)$ *satisfy*

$$\mathsf{M}_b = \mathsf{M}b \text{ and } p_b = p \qquad \text{if } a = b \tag{1}$$
$$\mathsf{M}_b = \mathsf{M}b \text{ and } p_b = |\mathsf{M}b| \qquad \text{if } a < b \tag{2}$$
$$\mathsf{M}_b = \mathsf{MS}(\mathsf{B}b) \qquad \text{if } a > b \tag{3}$$

A key to easily proving this theorem is a simple fact about maximal suffixes:

Lemma 4 *Let* $\mathsf{Y} = \mathsf{PA}^k\mathsf{B}$, *where* $\mathsf{MS}(\mathsf{Y}) = \mathsf{A}^k\mathsf{B}$ *and* $|\mathsf{B}| < |\mathsf{A}| = per(\mathsf{MS}(\mathsf{Y}))$. *The string* A *is border-free.*

Observe that each step of the while loop on line 4 in `UpdateMS` increases the value of the non-decreasing expression $i + s$. The final and initial values of i differ exactly by one. Hence we can make the following observation.

Observation 5 *The cost of* `UpdateMS` *is* $O(\Delta s)$.

The key property of maximal suffixes is the connection between $per(\mathsf{Y}[0..\ell))$ and $per(\mathsf{MS}(\mathsf{Y}[0..\ell)))$. In certain (easy to recognize) situations the two values are equal. We will now give a precise description of this connection.

We point out that a superset of the properties stated next is proven in [2]. However, our version of the algorithm requires a smaller number of (slightly simpler, both in terms of the claim and the proof) formal statements and we leave the proofs to present the algorithm description standalone.

```
Algorithm Match($\mathsf{X}, n, \mathsf{Y}, m$)
Input: strings $\mathsf{X}[0..n)$ (text) and $\mathsf{Y}[0..m)$ (pattern).
Output: the set $\mathcal{S} = \{i \in [0..n) \mid \mathsf{X}[i..i+m) = \mathsf{Y}\}$.
1:  $\mathcal{S} \leftarrow \emptyset$
2:  $i \leftarrow \ell \leftarrow p \leftarrow s \leftarrow 0$
3:  while $i < n$ do
4:      while $i + \ell < n$ and $\ell < m$ and $\mathsf{X}[i+\ell] = \mathsf{Y}[\ell]$ do
5:          $(\ell, s, p) \leftarrow$ UpdateMS($\mathsf{Y}, \ell, s, p$)
        // $\ell = \mathsf{lcp}(\mathsf{X}_i, \mathsf{Y})$
6:      if $\ell = m$ then
7:          $\mathcal{S} \leftarrow \mathcal{S} \cup \{i\}$
        // $\mathsf{MS}(\mathsf{Y}[0..\ell)) = \mathsf{Y}[s..\ell)$ and $p = per(\mathsf{Y}[s..\ell))$
8:      if $p \le \ell/3$ and $\mathsf{Y}[0..s) = \mathsf{Y}[p..p+s)$ then    // $per(\mathsf{Y}[0..\ell)) = p$
9:          $i \leftarrow i + p$
10:         $\ell \leftarrow \ell - p$
11:     else    // $per(\mathsf{Y}[0..\ell)) > \ell/3$
12:         $i \leftarrow i + \lfloor \ell/3 \rfloor + 1$
13:         $(\ell, s, p) \leftarrow (0, 0, 0)$
14: return $\mathcal{S}$
```

Figure 2. The main procedure of the simplified Crochemore's algorithm.

Lemma 6 *Let* $\mathsf{Y} = \mathsf{PA}^k\mathsf{B}$ *where* $\mathsf{MS}(\mathsf{Y}) = \mathsf{A}^k\mathsf{B}$ *and* $|\mathsf{B}| < |\mathsf{A}| = p := per(\mathsf{MS}(\mathsf{Y}))$. *Then:*

1. $|\mathsf{P}| < per(\mathsf{Y})$
2. $per(\mathsf{Y}) = per(\mathsf{MS}(\mathsf{Y}))$ *iff* P *is a suffix of* A
3. *if* Y *is 3-periodic then* $per(\mathsf{Y}) = per(\mathsf{MS}(\mathsf{Y}))$

Proof. Let $p' = per(\mathsf{Y})$.

1. Otherwise $\mathsf{A}^k\mathsf{B}$ occurs in Y p' positions earlier, thus is not a maximal suffix.

2. A prefix of Y of length $|\mathsf{P}| + p'$ has a border of length $|\mathsf{P}|$. If $p' = p$ position $|\mathsf{P}| + p$ coincides with the end of A.

The opposite implication follows from the definition of a period.

3. Clearly $p \le p'$ as $\mathsf{A}^k\mathsf{B}$ is a factor of Y. Suppose $p < p'$ and observe that $\mathsf{A}^k\mathsf{B}$ has periods p and p'. Moreover, $3p' \le |\mathsf{Y}|$ and $|\mathsf{P}| < p'$ imply $|\mathsf{A}^k\mathsf{B}| > 2p' > p + p'$ hence from Lemma 1 $\mathsf{A}^k\mathsf{B}$ has also period $p'' := \gcd(p, p')$. But $\mathsf{A}^k\mathsf{B}$ contains an occurrence of $\mathsf{Y}[0..p')$ as a factor thus $\mathsf{Y}[0..p')$ has period $p'' < p'$ and so (since $p'' \mid p'$) the whole Y as well, contradicting the definition of p'.

We immediately obtain the following result (for Y as in Lemma 6).

Corollary 7 Y *is 3-periodic iff* $p \le |\mathsf{Y}|/3$ *and* P *is a suffix of* A.

The pseudo-code of the matching procedure is given in Fig. 2. After computing $\ell = \mathsf{lcp}(\mathsf{X}_i, \mathsf{Y})$ we test if $\mathsf{Y}[0..\ell)$ is 3-periodic using Corollary 7. If it is not, we can safely set $q := \lfloor \ell/3 \rfloor$ and $\ell := 0$. Otherwise, from Lemma 6, we know that $p := per(\mathsf{MS}(\mathsf{Y}[0..\ell))) = per(\mathsf{Y}[0..\ell))$ thus we set $q := p$ and decrease the match length ℓ by p, because the definition of the period implies that we can skip the first $\ell - p$ characters when computing $\mathsf{lcp}(\mathsf{X}_{i+p}, \mathsf{Y})$.

However, now the problem is obtaining the starting position of the maximal suffix of $\mathsf{Y}[0..\ell - p)$ and its shortest period. As explained in the next Lemma, it turns out that both the starting position and the shortest period of the new maximal suffix stay the same.

This is in contrast with the original Crochemore's algorithm, where 3 cases are considered when performing the shift, each with more involved formulas expressing maximal possible shifts. It results in a tight upper bound on the number of comparisons, but at the cost of intricate complexity analysis and the need for more formal statements.

Lemma 8 *Assume* Y *is a 3-periodic string of length* ℓ. *Let* $\mathsf{MS}(\mathsf{Y}) = \mathsf{Y}_s$ *and* $per(\mathsf{Y}_s) = p$. *Then for* $\mathsf{Y}' := \mathsf{Y}[0..\ell - p)$ *we have* $\mathsf{MS}(\mathsf{Y}') = \mathsf{Y}'_s$ *and* $per(\mathsf{Y}'_s) = p$.

Proof. Suppose $\mathsf{MS}(\mathsf{Y}') = \mathsf{Y}'_{s'}$ for $s' \neq s$. The only case that does not immediately yields $\mathsf{Y}_{s'} > \mathsf{Y}_s$ (contradicting $\mathsf{MS}(\mathsf{Y}) = \mathsf{Y}_s$) is when $s' < s$ and Y'_s is a prefix of $\mathsf{Y}'_{s'}$. It is also its suffix, thus $\mathsf{Y}'_{s'}$ has a period $s - s'$. It also has period p and inequalities $|\mathsf{Y}'| \geq 2p$, $s' < s < p$ (recall Lemma 6(1)) imply $|\mathsf{Y}'_{s'}| \geq 2p - s' > p + (s - s')$, thus from Lemma 1 $\mathsf{Y}'_{s'}$ has period $p' := \gcd(p, s - s') < p$. But $\mathsf{Y}'_{s'}$ contains an occurrence of $\mathsf{Y}[0..p)$ hence $\mathsf{Y}[0..p)$ must also have period p', and since $p' \mid p$ the whole Y as well, a contradiction.

Clearly $per(\mathsf{Y}'_s) \leq p$, as Y'_s is a factor of Y. It cannot be $p' := per(\mathsf{Y}'_s) < p$ because then $\mathsf{Y}'_s[p'..p)$ is a border of $\mathsf{Y}_s[0..p)$ which is impossible by Lemma 4.

Theorem 9 `Match` *runs in* $\mathrm{O}(n + m)$ *time and uses* $\mathrm{O}(1)$ *extra space.*

Proof. Clearly only a constant number of integer variables are used throughout the computation and neither the text nor the pattern are modified.

Each step of the while loop in line 3 increases the value of the non-decreasing expression $3i + \ell$, thus it is executed at most $3n + m = \mathrm{O}(n + m)$ times.

The total cost of `UpdateMS` is bounded by the total increase of s (Observation 5). The maximal value of s is $m - 1$ and it can only decrease in line 13. But since $s < \ell$ and the decrease is always followed by increasing i by $\lfloor \ell/3 \rfloor + 1 > s/3$, s can overall increase by at most $3n + m = \mathrm{O}(n + m)$.

Finally, we divide the checks $\mathsf{Y}[0..s) = \mathsf{Y}[p..p+s)$ into two groups. If the condition in line 8 evaluates to true we have $s < per(\mathsf{Y}[0..\ell)) = p$ (see Lemma 6) and i is immediately increased by p (line 9), thus the total cost of such checks is $\mathrm{O}(n)$. Otherwise i is incremented by $\lfloor \ell/3 \rfloor + 1 > s/3$ (line 12). The maximal value of i is n, thus such checks overall cost at most is $3n = \mathrm{O}(n)$.

4 Extensions

4.1 Longest Prefix Matching

We search a pattern Y inside X and keep track of the length $\ell_{\max}$ of the longest matching prefix of Y found so far. The pseudo-code, which is a straightforward modification of the `Match` procedure is given in Fig. 3. During the computation we maintain a set of text positions $\mathcal{S}$ such that $j \in \mathcal{S}$ iff $\mathsf{lcp}(\mathsf{X}_j, \mathsf{Y}) = \ell_{\max}$.

```
Algorithm LongestPrefixMatch($\mathsf{X}, n, \mathsf{Y}, m$)
Input: strings $\mathsf{X}[0..n)$ (text) and $\mathsf{Y}[0..m)$ (pattern).
Output: $\ell_{\max} = \max_{i\in[0..n)} \mathsf{lcp}(\mathsf{X}_i, \mathsf{Y})$ and $\mathcal{S} = \{j \in [0..n) \mid \mathsf{lcp}(\mathsf{X}_j, \mathsf{Y}) = \ell_{\max}\}$.
1:  $\mathcal{S} \leftarrow \emptyset$
2:  $i \leftarrow p \leftarrow s \leftarrow \ell \leftarrow \ell_{\max} \leftarrow 0$
3:  while $i < n$ do
4:      while $i + \ell < n$ and $\ell < m$ and $\mathsf{X}[i + \ell] = \mathsf{Y}[\ell]$ do
5:          $(\ell, s, p) \leftarrow$ UpdateMS($\mathsf{Y}, \ell, s, p$)
6:      if $\ell > \ell_{\max}$ then
7:          $\mathcal{S} \leftarrow \{i\}$
8:          $\ell_{\max} \leftarrow \ell$
9:      elsif $\ell = \ell_{\max}$ then
10:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{i\}$
11:     if $p \le \ell/3$ and $\mathsf{Y}[0..s) = \mathsf{Y}[p..p + s)$ then
12:         $i \leftarrow i + p$; $\ell \leftarrow \ell - p$
13:     else
14:         $i \leftarrow i + \lfloor \ell/3 \rfloor + 1$; $(\ell, s, p) \leftarrow (0, 0, 0)$
15: return $(\ell_{\max}, \mathcal{S})$
```

Figure 3. The basic algorithm solving Longest Prefix Matching problem.

Theorem 10 *The algorithm* `LongestPrefixMatch` *solves the Longest Prefix Matching problem in linear time.*

Proof. The time complexity follows from Theorem 9.

To prove its correctness, observe that after line 10 we have $\ell_{\max} \ge \ell$. The shift that follows is not longer than $per(\mathsf{Y}[0..\ell))$, so from Observation 2 all positions j that we skip satisfy $\mathsf{lcp}(\mathsf{X}_j, \mathsf{Y}) < \ell \le \ell_{\max}$, i.e., we only omit the candidates for $\ell_{\max}$ that would not change its value nor end up in $\mathcal{S}$.

Note that the peak size of set $\mathcal{S}$ can be much larger than the final output. For instance when $\mathsf{X} = a^q b$ and $\mathsf{Y} = ab$ the size of $\mathcal{S}$ reaches $q - 1$ but the final $\mathcal{S}$ satisfies $|\mathcal{S}| = 1$.

It is possible to get rid of this overhead as follows. First run `LongestPrefixMatch` but only record the length $\ell_{\max}$. Then, in the second run, collect exclusively the elements on the final set $\mathcal{S}$, which can now be easily recognized. We have proved the following

Theorem 11 *It is possible to solve the Longest Prefix Matching problem in linear time and using only constant extra space in addition to the input and the output.*

4.2 Sparse Longest Prefix Matching

Let $\mathcal{P}$ be the ascending sequence of text positions given in addition to the text X and the pattern Y. In order to solve the sparse variant of the problem, we proceed exactly the same as in the basic version, but only execute lines 4-10 if $i \in \mathcal{P}$. We call this modified algorithm `SparseLongestPrefixMatch`.

Theorem 12 *The algorithm* `SparseLongestPrefixMatch` *solves the Sparse Longest Prefix Matching problem in linear time.*

Proof. The condition $i \in \mathcal{P}$ can be checked in constant time since i never decreases and the elements of $\mathcal{P}$ are given in ascending order. The analysis from Theorem 9 applies to the rest of the algorithm.

In order to prove its correctness observe that whenever we are about to execute lines 4-10 the condition $\ell \leq \ell_{\max}$ is satisfied, even if $i \notin \mathcal{P}$. This is because ℓ increases *only* for positions $i \in \mathcal{P}$ and any such increase is immediately recorded in lines 6-10. Therefore the argument from Theorem 10 also applies here, i.e., the positions in the text that are not inspected would never contribute to the answer.

An identical technique as for `LongestPrefixMatch` can be applied to reduce the memory overhead caused by the large peak size of $\mathcal{S}$ yielding

Theorem 13 *It is possible to solve the Sparse Longest Prefix Matching problem in linear time and using only constant extra space in addition to the input and the output.*

References

1. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Communications of the ACM, 20(10) 1977, pp. 762–772.
2. M. Crochemore: *String-matching on ordered alphabets.* Theoretical Computer Science, 92 1992, pp. 33–47.
3. M. Crochemore and W. Rytter: *Sqares, cubes, and time-space efficient string searching.* Algorithmica, 13(5) 1995, pp. 405–425.
4. J.-P. Duval: *Factorizing words over an ordered alphabet.* Journal of Algorithms, 4(4) 1983, pp. 363–381.
5. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Computing Surveys, 45(2) 2013, article 13.
6. P. Ferragina and G. Manzini: *On compressing the textual web*, in WSDM, ACM, 2010, pp. 391–400.
7. N. J. Fine and H. S. Wilf: *Uniqueness theorems for periodic functions.* Proceedings of the American Mathematical Society, 16(1) 1965, pp. 109–114.
8. T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi: *A faster grammar-based self-index*, in LATA, vol. 7183 of LNCS, Springer, 2012, pp. 240–251.
9. T. Gagie, P. Gawrychowski, and S. J. Puglisi: *Faster approximate pattern matching in compressed repetitive texts*, in ISAAC, vol. 7074 of LNCS, Springer, 2011, pp. 653–662.
10. Z. Galil and J. Seiferas: *Time-space optimal string matching.* Journal of Computer and System Sciences, 26 1983, pp. 280–294.
11. Z. Galil and J. I. Seiferas: *Saving space in fast string-matching.* SIAM Journal on Computing, 9(2) 1980, pp. 417–438.
12. C. Hoobin, S. J. Puglisi, and J. Zobel: *Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections.* Proceedings of the VLDB Endowment, 5(3) 2011, pp. 265–273.
13. J. Kärkkäinen, D. Kempa, and S. J. Puglisi: *Lempel-Ziv parsing in external memory.* Manuscript, `http://arxiv.org/abs/1307.1428`, 2013.
14. J. Kärkkäinen, D. Kempa, and S. J. Puglisi: *Lightweight Lempel-Ziv parsing*, in SEA, vol. 7933 of LNCS, Springer, 2013, pp. 139–150.
15. J. Kärkkäinen, D. Kempa, and S. J. Puglisi: *Linear time Lempel-Ziv factorization: Simple, fast, small*, in CPM, vol. 7922 of LNCS, Springer, 2013, pp. 189–200.
16. D. Kempa and S. J. Puglisi: *Lempel-Ziv factorization: simple, fast, practical*, in ALENEX, SIAM, 2013, pp. 103–112.
17. D. Knuth, J. H. Morris, and V. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
18. S. Kreft and G. Navarro: *Self-indexing based on LZ77*, in CPM, vol. 6661 of LNCS, Springer, 2011, pp. 41–54.
19. J. H. Morris, Jr and V. R. Pratt: *A linear pattern-matching algorithm*, Report 40, University of California, Berkeley, 1970.
20. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23(3) 1977, pp. 337–343.

Weak Factor Automata: Comparing (Failure) Oracles and Storacles

Loek Cleophas[1], Derrick G. Kourie[1], and Bruce W. Watson[2]

[1] FASTAR Research Group, Department of Computer Science, University of Pretoria, Private Bag X20, 0028 Hatfield, Pretoria, Republic of South Africa
{loek,derrick}@fastar.org

[2] FASTAR Research Group, Department of Information Science, Stellenbosch University, Private Bag X1, 7602 Matieland, Republic of South Africa
bruce@fastar.org
http://www.fastar.org

Abstract. The *factor oracle* [3] is a data structure for weak factor recognition. It is a deterministic finite automaton (DFA) built on a string p of length m that is acyclic, recognizes at least all factors of p, has $m+1$ states which are all final, is homogeneous, and has m to $2m-1$ transitions. The *factor storacle* [6] is an alternative automaton that satisfies the same properties, except that its number of transitions may be larger than $2m-1$, although it is conjectured to be linear with an upper bound of at most $3m$. In [14] (among others), we described the concept of a *failure automaton* i.e. a *failure DFA* (FDFA), in which so-called failure transitions are used to reduce the total number of transitions and thus reduce representation space compared to the use of a DFA. We modify factor oracle and storacle construction algorithms to introduce failure arcs *during* the respective automata's construction. We thus end up with four deterministic automata types for weak factor recognition: factor oracle, factor storacle, failure factor oracle, and failure factor storacle. We compare them empirically in terms of size. The results show that despite the relative simplicity of (failure) factor (st)oracles, the failure versions show additional savings of 2–7% in number of transitions, for generated keywords of length 5–9, and of e.g. 5–9% for English words of lengths around 9–15. This may already be substantial in memory-restricted settings such as hardware implementations of automata. The results indicate the gains increase for longer keywords, which seems promising for applications in DNA processing and intrusion detection. Furthermore, our results provide a rather negative result on storacles: apart from rare cases, factor storacles do not have fewer transitions than factor oracles, and similarly for failure factor storacles versus failure factor oracles.

Keywords: factor oracle, approximate automaton, failure automaton, weak factor recognition, pattern matching

1 Introduction

The *factor oracle* is a data structure for weak factor recognition. It is an automaton built from a string p of length m that (a) is acyclic, (b) recognizes at least all factors of p, (c) has $m+1$ states (which are all final), and (d) has m (at least, one for each letter in p) to $2m-1$ transitions (cf. [3]). In addition, (e) the resulting automaton is homogeneous, i.e. for every state, all of its incoming transitions are on the same symbol. An example factor oracle is given in Figure 1. Factor oracles are introduced in [3] as an alternative to the use of exact factor recognition in many on-line keyword pattern matching algorithms. In such algorithms, a window on a text is read backward while attempting to match a keyword factor. When this fails, the window is shifted using the information of the longest factor matched and the mismatching character.

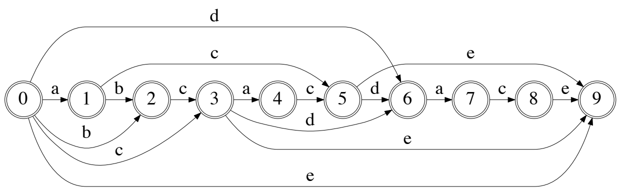


Figure 1: Factor oracle (with initial state 0) recognizing a superset of $\mathbf{fact}(p)$ (including for example $cace \notin \mathbf{fact}(p)$), for $p = abcacdace$. The automaton has 17 transitions.

Instead of an automaton recognizing exactly all factors of the keyword, it is possible to use a factor oracle: although it recognizes more strings than just the factors and thus might read backwards longer than necessary, it cannot miss any matches. The advantage of using factor oracles is that they are easier to construct and take less space to represent compared to automata that were previously used in these factor-based algorithms, such as suffix, factor and subsequence automata. This is due to the latter automata lacking one or more of the essential properties of the factor oracle.

In [7], we presented an alternative construction algorithm for factor oracles. This algorithm was based on considering the suffixes of the string p in order of decreasing length. While being $\mathcal{O}(m^2)$ and not linear like the algorithm in [3], this construction is easier to understand. (It also makes some of the factor oracle's properties immediately obvious, while making some others harder to prove.) An extended version of [7] appears as [8] and in the Master's thesis [10, Chapter 4]. In those versions, some properties of the language of a factor oracle are discussed as well. The thesis also discusses the implementation of the factor oracle in the SPARE Time toolkit. A further extended and revised version of the work appears in [9]. The language of a factor oracle was finally characterized completely in a paper by Mancheron and Moan [15]. Related to the factor oracle, the *suffix oracle*—in which only those states corresponding to a suffix of p are marked final—is introduced in [3]. In [5], the authors present a statistical average-case analysis on the size of factor and suffix oracles.

In [6] we presented the *factor storacle*, short for *shortest forward transition factor oracle*. The factor storacle is an alternative automaton that satisfies the same properties as the factor oracle does, except property (d) mentioned earlier: in contrast to the case of the factor oracle for the same keyword, the factor storacle's number of transitions may be larger than $2m-1$, although it is conjectured to be linear with an upper bound of at most $3m$. We presented a construction algorithm for factor storacles as well as a limited empirical comparison of factor oracles and factor storacles, showing the maximum numbers of transitions the factor oracle and factor storacle for particular string lengths may have, and leading to the conjecture mentioned above. For certain keywords, the factor storacle has a *smaller* number of transitions than the factor oracle, although such cases turn out to be rare, as we empirically show in this paper. Figure 2 shows an example factor storacle (having one less transition than the corresponding factor oracle depicted in Figure 1).

In [14], we described the concept of a *failure automaton* i.e. a *failure DFA* (FDFA). In such an automaton so-called failure transitions are used to reduce the total number of transitions compared to a DFA for the same language. This is done to reduce the space needed to represent the automaton compared to the space usage of a DFA representation. Björklund et al. in [4] recently showed that even without changing the

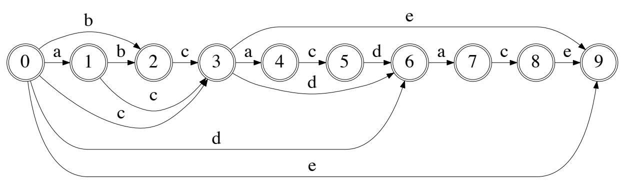


Figure 2: Factor storacle (with initial state 0) recognizing a superset of $\mathbf{fact}(p)$ (including e.g. $abce \notin \mathbf{fact}(p)$), for $p = abcacdace$. The automaton has 16 transitions.

state set from DFA to FDFA, the problem of minimizing the number of transitions by replacing symbol transitions by failure transitions is unfortunately NP-complete, although it can be approximated efficiently within a factor of $\frac{2}{3}$.

The concepts of factor oracle and factor storacle serve to reduce memory usage compared to a factor automaton, while the concept of an FDFA does the same for the general DFA case, albeit in different ways. It is therefore of interest to combine the basic ideas and empirically investigate the results. In the current paper, we thus combine the ideas of factor oracle and storacle on the one hand, and failure automata on the other hand. We modify the factor oracle and factor storacle construction algorithms to introduce failure arcs *during* the respective automata's construction. We thus end up with four kinds of deterministic automata for weak factor recognition: the factor oracle, factor storacle, failure factor oracle, and failure factor storacle. We compare them empirically in terms of size, using both randomly generated keywords as well as English dictionary keywords for the construction process.

After discussing preliminaries, we consider suffix-based factor oracle and factor storacle construction in Section 2. We present our previously existing construction algorithms for these two cases. In Section 3 we present our modified algorithms, directly constructing the failure factor oracle and the failure factor storacle respectively, and we discuss the properties of these two automata types. Section 4 presents and analyses our preliminary benchmarking results in comparing the four resulting automata types; these results focus on size of the resulting automata in terms of number of (symbol and failure) transitions. Section 5 provides concluding remarks as well as a discussion of ideas for future research in this subject area.

2 Suffix-based Construction of the Factor Oracle and Factor Storacle

Formally, a *string* $p = p_1 \cdots p_m$ of length m is a sequence of characters from an alphabet V. A string u is a *factor* (resp. *prefix*, *suffix*) of a string v if $v = sut$ (resp. $v = ut$, $v = su$), for $s, t \in V^*$. We will use $\mathbf{pref}(p)$, $\mathbf{suff}(p)$ and $\mathbf{fact}(p)$ for the set of prefixes, suffixes and factors of p respectively. A prefix (resp. suffix or factor) is a *proper* prefix (resp. suffix or factor) of a string p if it does not equal p.

In Algorithm 1 the factor oracle construction algorithm given in [7,9] is repeated. In steps 1 to 4 the algorithm constructs a 'skeleton' automaton for p—recognizing $\mathbf{pref}(p)$. In steps 5 to 8, it then considers, in decreasing order of length, each proper suffix $p_i \cdots p_m$ of p. For each such suffix, it determines the longest prefix recognised by the automaton to date—i.e. the longest path starting from state 0 and ending in some state j that spells out $p_i \cdots p_k$ $(i-1 \leq k \leq m)$. If such a suffix $p_i \cdots p_m$ is already recognized (i.e. if $k = m$), then no transition needs to be constructed. If on the other hand the complete suffix is not yet recognized—i.e. if $p_i \cdots p_k$ is the longest prefix recognised where $k < m$) and if the recognition path ends at state j—then a transition is inserted from state j to state $k + 1$. It can be easily shown that the language recognised by the resulting automaton is a superset of $\mathbf{pref}(\mathbf{suff}(p)) = \mathbf{fact}(p)$.

Algorithm 1 Build_Oracle($p = p_1p_2 \cdots p_m$)

```
1: for $i$ from 0 to $m$ do
2:     Create a new final state $i$
3: for $i$ from 0 to $m - 1$ do
4:     Create a new transition from $i$ to $i + 1$ on symbol $p_{i+1}$
5: for $i$ from 2 to $m$ do
6:     Let the longest path from state 0 that spells a prefix of $p_i \cdots p_m$ end in state $j$ and spell out
       $p_i \cdots p_k$ $(i - 1 \le k \le m)$
7:     if $k \neq m$ then
8:         Build a new transition from $j$ to $k + 1$ on symbol $p_{k+1}$
```

This algorithm is $\mathcal{O}(m^2)$. The factor oracle on p built using this algorithm is referred to as Oracle(p) and the language recognized by it as $\mathbf{factoracle}(p)$.

Our factor storacle construction algorithm, presented in [6], is similar to our factor oracle construction algorithm. It is reproduced in Algorithm 2. It also constructs a 'skeleton' automaton for p—recognizing $\mathbf{pref}(p)$—and then also constructs a path for each of the proper suffixes of p in order of decreasing length, such that eventually at least $\mathbf{pref}(\mathbf{suff}(p)) = \mathbf{fact}(p)$ is recognized. If such a suffix of p is already recognized, no transition needs to be constructed. If on the other hand the complete suffix is not yet recognized there is a longest prefix of such a suffix that is recognized.

A transition on the next, non-recognized symbol is then created, from the state in which this longest prefix of the suffix is recognized. Instead of creating such a transition to the unique state from which the remainder of that suffix is known to be recognized, as is done in the factor oracle construction above, this transition is constructed to go to the next state from the current state onward that has an incoming transition on the non-recognized symbol. That is, the factor storacle construction algorithm in such a case constructs the shortest forward transition that keeps the automaton homogeneous. This procedure of creating transitions is repeated while the complete suffix is not yet recognized.

Algorithm 2 Build_Storacle($p = p_1p_2 \cdots p_m$)

```
1: for $i$ from 0 to $m$ do
2:     Create a new final state $i$
3: for $i$ from 0 to $m - 1$ do
4:     Create a new transition from $i$ to $i + 1$ on symbol $p_{i+1}$
5: for $i$ from 2 to $m$ do
6:     Let the longest path from state 0 that spells a prefix of $p_i \cdots p_m$ end in state $j$ and spell out
       $p_i \cdots p_k$ $(i - 1 \le k \le m)$
7:     while $k \neq m$ do
8:         Let the first state from state $j$ onward that has an incoming transition on $p_{k+1}$ be state $l$
           $(j < l <= k + 1)$
9:         Build a new transition from $j$ to $l$ on symbol $p_l (= p_{k+1})$
10:        Let the longest path from state 0 that spells a prefix of $p_i \cdots p_m$ end in state $j$ and spell
           out $p_i \cdots p_k$ $(i - 1 \le k \le m)$
```

This algorithm is $\mathcal{O}(m^3)$, although that is a coarse upper bound. The factor storacle on p built using this algorithm is referred to as Storacle(p) and the language recognized by it as **factstoracle**(p).

As stated in [6], the difference between this algorithm and the $\mathcal{O}(m^2)$ factor oracle construction algorithm originates from the choice of the target of the first (if any) newly created transition for each proper suffix:

– In this algorithm, that newly created transition leads to the next state (from a particular state onward) that has an incoming transition on the non-recognized symbol. This procedure may then need to be repeated for further symbols of the suffix to be recognized.
– In the case of the factor oracle construction, the newly created transition leads to the unique state from which the remainder of the suffix leads to the last state of the automaton—thus immediately guaranteeing that the entire suffix is recognized.

We summarize the most important properties of factor oracles and factor storacles. All of these were known before; some proofs are therefore omitted or sketched, and can be found in e.g. [3,7,9,6]. The first properties mentioned correspond to properties (a)–(c) and (e) from the introduction, and hold for factor oracles and factor storacles.

Property 1. Oracle(p) and Storacle(p) are acyclic automata.
Proof idea: For the factor oracle, it is obvious that transitions created are always forward ones; for the factor storacle, it can be shown that the transitions created may be different from those created for the factor oracle, but are still forward ones.

Property 2. **fact**(p) $\subseteq$ **factoracle**(p) and **fact**(p) $\subseteq$ **factstoracle**(p).

Property 3. For p of length m, Oracle(p) and Storacle(p) each have exactly $m + 1$ states.

Property 4 (Homogeneousness). All transitions reaching a state i of Oracle(p) and Storacle(p) are labeled by p_i.

Furthermore, factor oracles and factor storacles satisfy the following obvious property:

Property 5 (Weak determinism). For each state of Storacle(p) or Oracle(p), no two outgoing transitions of the state are labeled by the same symbol.

As stated before, property (d), the remaining property enumerated in the introduction, only holds for factor oracles, while a weaker property holds for factor storacles.

Property 6. For p of length m, Oracle(p) has between m and $2m - 1$ transitions.

Since the factor storacle construction algorithm we presented might create multiple transitions per proper suffix of the keyword, this property does not hold for factor storacles. [6] showed the following very coarse upper bound on the total number of transitions of the factor storacle:

Property 7. For p of length m, Storacle(p) has between m and $m(m+1)/2$ transitions.
Proof: The lower bound follows from the second **for**-loop of the algorithm. Disregarding any properties of the keyword and alphabet used (except for the keyword's length

m), an upper bound of $m(m+1)/2$ can be proven in at least two ways. Firstly, the sum of the lengths of all the suffixes of a keyword of length m, including the keyword itself, equals $m(m+1)/2$. Secondly, since all transitions are forward transitions, and the factor storacle is kept homogeneous, there can be at most one transition between each pair of states, hence at most $(|Q|-1)|Q|/2$ in total, and this equals $m(m+1)/2$ since $m = |Q| - 1$. □

[6] also conjectured a linear upper bound on the number of transitions of the factor oracle, based on empirical evidence; experiments generating all keywords of length m out of an alphabet of size $|m|$ (modulo renaming of alphabet symbols) showed that the upper bound increases from at most $2m$ for lengths up to $m = 7$ to $2m + 5$ for length $m = 12$, i.e. grows linearly in the range of the experiments.

Conjecture 8. For p of length m, Storacle(p) has a linear number of transitions, bounded above by $3m$.

3 Suffix-based Construction of the Failure Factor Oracle and Failure Factor Storacle

In [14], we give a general algorithm for constructing a *failure deterministic finite automaton* (FDFA) based on a given deterministic finite automaton (DFA). The algorithm ensures that the constructed FDFA is language-equivalent to the given DFA. Such an FDFA in essence forms a generalization of the failure function Aho-Corasick automaton [2,17]: from a finite set of keywords (as in normal Aho-Corasick), to the general/arbitrary regular language case. In essence, an FDFA is a DFA, but may have so-called failure transitions apart from normal symbol transitions. Such transitions are introduced to save space: under certain conditions, a single failure transition can be used as default instead of multiple symbol transitions. These failure transitions are represented by the function $\mathfrak{f}$ in the definition below.

Definition 9 (FDFA [14]). *$\mathcal{F} = (Q, \Sigma, \delta, \mathfrak{f}, F, s)$ is an FDFA if $\mathfrak{f} : Q \to Q$ is a possibly partial function and $\mathcal{D} = (Q, \Sigma, \delta, F, s)$ is a DFA.*

As with a DFA, a simple string recognition algorithm can be used to determine whether or not a given string is part of the FDFA's language. The algorithm corresponds to that of a DFA by consuming an input symbol and moving to a next state if there is an out-transition from the current state on the current input symbol. However, if there is no such out-transition, but a failure transition, then the failure transition determines the new state, but the current input symbol is not consumed.

The above FDFA definition may lead to complications in the presence of certain types of cycles in the failure function. More precisely, cycles in which, for one or more symbols, no state in the cycle has an out-transition labeled by this symbol are problematic for the associated FDFA string recognition algorithm. [14] called these *divergent failure cycles*, and ensured that the FDFA construction algorithm presented simply does not create such divergent failure cycles. In the present setting, no failure cycles are created at all, circumventing the potential complications altogether.

In [14], FDFAs were created by taking (complete) DFAs and transforming them. Here, we introduce failure transitions *during* construction of weak factor automata. We do so by slight modifications of the factor oracle and factor storacle construction algorithms presented before. These modifications lead to construction algorithms for what we call the *failure factor oracle* and the *failure factor storacle* respectively. It should be noted that the resulting automata are not necessarily language-equivalent to the original factor oracle or factor storacle respectively, but we are not concerned with such language-equivalence here: what matters is that the resulting automata recognize at least all factors of the given keyword.

Algorithm 3, the failure factor oracle construction algorithm, is similar to the factor oracle one, Algorithm 1. The main differences are in lines 6 and 8–11: in line 6, from state j processing continues with (0 or more) existing failure transitions leading to a state j', to prevent constructing a failure transition (in line 9) from a state that already has an outgoing failure transition. (Note that the recognition path that leads to state j may also contain failure transitions—another implicit difference to Algorithm 1.) In line 9, in case $k > j'$, instead of a symbol transition from j to $k+1$ on symbol p_{k+1}, a failure transition from j' to k is constructed. Note that a transition on symbol p_{k+1} from state k to state $k+1$ will exist, due to lines 1–4 of the algorithm, and hence processing of $p_i \cdots p_k p_{k+1}$ will have the automaton end up in state $k+1$, just as it would in the original factor oracle.

Our initial version of the algorithm did not have the inner **if**-statement, assuming $k > j'$ to always hold inside the outer **if**-statement, and therefore always building a new failure transition from j' to k, keeping the automaton acyclic. For the large data sets we used in the experiments reported further on in this paper, this holds true, but it is not true in general: with increasing keyword length, it becomes possible in rare cases for $k > j'$ not to hold. In some such cases, cycles of failure transitions arise, which in some cases lead to divergent failure cycles and even live-lock of the construction algorithm. The **else**-case of lines 10–11 ensures that this does not happen, by creating an appropriate non-forward *symbol* transition instead of a non-forward failure transition. The failure factor oracle in general thus does not have the acyclicity property of the factor oracle, but our initial experiments with sets of longer keywords (on a DNA alphabet) show such cases to be rare ($< 0.001\%$ of 749920 keywords tested of length 16 rising to ca. 1% of 5935 keywords tested of length 1024).

Algorithm 3 Build_Failure_Oracle($p = p_1 p_2 \cdots p_m$)

```
 1: for $i$ from 0 to $m$ do
 2:     Create a new final state $i$
 3: for $i$ from 0 to $m-1$ do
 4:     Create a new transition from $i$ to $i+1$ on symbol $p_{i+1}$
 5: for $i$ from 2 to $m$ do
 6:     Let the longest recognized prefix of $p_i \cdots p_m$ be recognized in state $j$ and spell out $p_i \cdots p_k$
        ($i-1 \le k \le m$), and let the longest failure transition path from $j$ end in state $j'$
 7:     if $k \ne m$ then
 8:         if $k > j'$ then
 9:             Build a new failure transition from $j'$ to $k$
10:         else
11:             Build a new symbol transition on symbol $p_{k+1}$ from $j'$ to $k+1$
```

This algorithm is $\mathcal{O}(m^2)$. The failure factor oracle on p built using this algorithm is referred to as FailureOracle(p). It is easy to show that, apart from acyclicity, the properties of the factor oracle mentioned previously do hold for the failure version.

The failure factor storacle construction algorithm, Algorithm 4, is similar to the factor storacle construction algorithm, Algorithm 2. The main differences are in lines 6 and 8–13: on lines 6 and 13, as for the failure factor oracle construction above, processing continues with existing failure transitions, to prevent constructing one from a state that already has an outgoing failure transition; on lines 9 and 10, instead of a symbol transition from j to l on symbol p_{k+1}, a failure transition from j' to $l-1$ ($j' < l <= k+1$) is constructed. Note that a path (possibly using failure transitions) to process symbol p_{k+1} from that state l may not exist, and processing of the current suffix thus has to continue, as in Algorithm 2. As with Algorithm 3, an **else**-case is added to prevent divergent failure cycles from arising in case $k \leq j'$.

Algorithm 4 Build_Failure_Storacle($p = p_1 p_2 \cdots p_m$)

```
 1: for $i$ from 0 to $m$ do
 2:     Create a new final state $i$
 3: for $i$ from 0 to $m-1$ do
 4:     Create a new transition from $i$ to $i+1$ on symbol $p_{i+1}$
 5: for $i$ from 2 to $m$ do
 6:     Let the longest recognized prefix of $p_i \cdots p_m$ be recognized in state $j$ and spell out $p_i \cdots p_k$
        ($i-1 \leq k \leq m$), and let the longest failure transition path from $j$ end in state $j'$
 7:     while $k \neq m$ do
 8:         if $k > j'$ then
 9:             Let the first state from state $j'$ onward that has an incoming transition on $p_{k+1}$ be state
                $l$ ($j' < l <= k+1$)
10:             Build a new failure transition from $j'$ to $l-1$
11:         else
12:             Build a new symbol transition on symbol $p_{k+1}$ from $j'$ to $k+1$
13:         Let the longest recognized prefix of $p_i \cdots p_m$ be recognized in state $j$ and spell out $p_i \cdots p_k$
            ($i-1 \leq k \leq m$), and let the longest failure transition path from $j$ end in state $j'$
```

This algorithm is $\mathcal{O}(m^3)$, although that is a coarse upper bound. The failure factor storacle on p built using this algorithm is referred to as FailureStoracle(p).

Figure 3 depicts an example of a failure factor oracle and a failure factor storacle in one: for this particular keyword, the automata happen to be equivalent. Figures 4a and 4b depict the case of keyword *abcaab*, for which the automata differ.

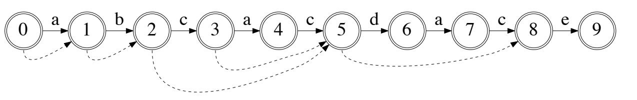
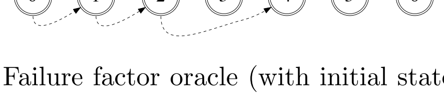
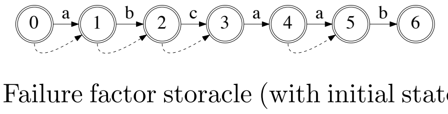


Figure 3: Failure factor oracle and failure factor storacle (with initial state 0) recognizing a superset of **fact**(p) (including for example *cace* $\notin$ **fact**(p), and *acace* not recognized by Oracle(p)), for $p = abcacdace$. The automaton has 14 transitions.



(a) Failure factor oracle (with initial state 0). The automaton has 9 transitions.



(b) Failure factor storacle (with initial state 0). The automaton has 10 transitions.

Figure 4: Failure factor oracle and failure factor storacle for $p = abcaab$.

As with the factor oracle and its failure version, the properties of the factor storacle other than acyclicity can easily be shown to hold for the failure factor storacle.

4 Empirical results

We implemented the four construction algorithms in Java, and ran benchmarks on an 1.7 GHz Intel Core i5 with 4 GB of 1333 MHz DDR3 RAM, running OS X 10.8.3. Two sets of data were used for the benchmarks, one consisting of generated strings of certain lengths, and one consisting of English words of widely varying lengths.

The first set consists of all generated strings of length m over an alphabet of size m, for values of m in the range of 4..9. (Strings of length < 4 are not considered, as for every string of such length the factor oracle and factor storacle do not differ.)

Figure 5 shows the distributions of the number of transitions for the first data set, for $m = 4..9$. As can be seen from the figure, the number of automata for particular numbers of transitions may vary drastically, from near 0 to almost m^m. Note that absent bars indicate values of 0 (i.e. no automata/keyword result in automata of a particular type with the given number of transitions), while bars represented by a flat line (i.e. bars seemingly of height 0) in fact indicate small but non-0 numbers of keywords/automata having the given number of transitions. As keyword length grows, it becomes easier to see that storacle versions of the automata are typically outperformed by oracle versions of the automata in terms of number of transitions. The graphs also show the *average* number of transitions per automata type for each of $m = 4..9$ (using dashed vertical lines). The average number of transitions for factor oracle and factor storacle on the one hand, and for their failure versions on the other hand, are fairly close, particularly for small word lengths, causing the dotted lines to overlap in the figures. Closely looking at the graph for e.g. $m = 8$ or $m = 9$ however shows that in fact four average lines are represented in each graph. The averages for factor oracles on the one hand and failure factor oracles on the other hand show that the use of failure version may lead to savings increasing from 1.5% for $m = 4$ up to 6.4% for $m = 9$, and suggest such savings may (sublinearly) increase further for longer keywords.

It is noteworthy that (for keyword lengths $m = 5..9$) only the factor storacle breaks the upper bound of $2m-1$ transitions established for the factor oracle; neither failure version breaks this barrier—and the failure factor oracle cannot for any keyword length, as it has the same transition set upper bound as the factor oracle. However, it is likely that the failure factor storacle will break the $2m - 1$ barrier as keyword length increases.

To make the experimental results more insightful, Figure 6 shows histograms for the *difference* in the number of transitions between factor oracles on the one hand and factor storacles, failure factor oracles or failure factor storacles on the other hand, again for all words of lengths $m = 4..9$ over alphabets of size m. For ease of understanding and comparison, the scale used here is a logarithmic one, and labels are in terms of percentages of all (automata for) keywords of a given length.

A number of interesting observations can be made from Figure 6:

– Comparing factor oracles and factor storacles, it turns out that in most cases, they are the same size. In a reasonable number of cases, growing to ca. 13% for $m = 9$, does the factor oracle have one or more transitions less than the factor storacle. What is remarkable is that only in rare cases does the factor storacle beat the

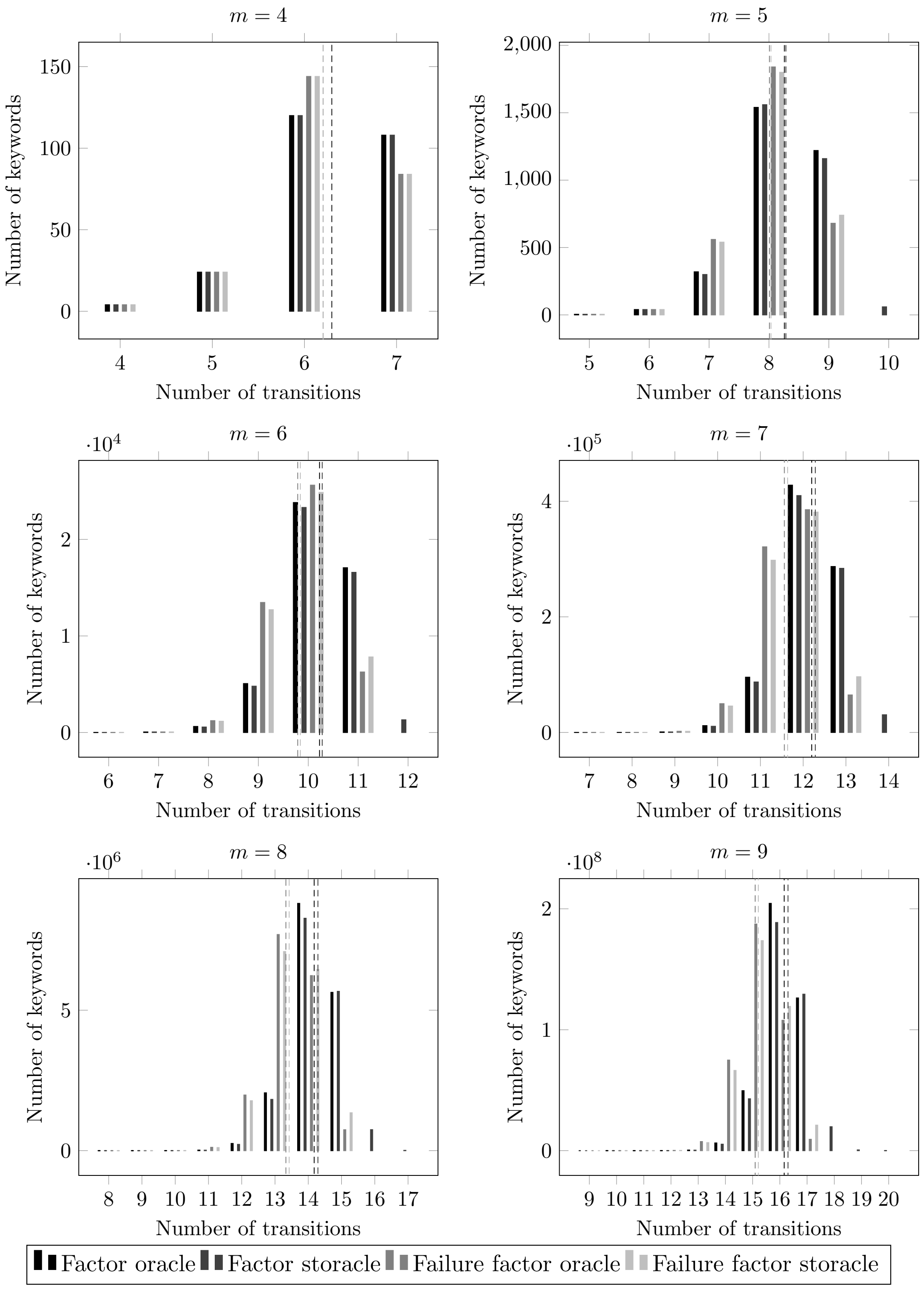


Figure 5: Distribution of number of transitions for the four automaton types, for all words of lengths $m = 4..9$ over alphabets of size m.

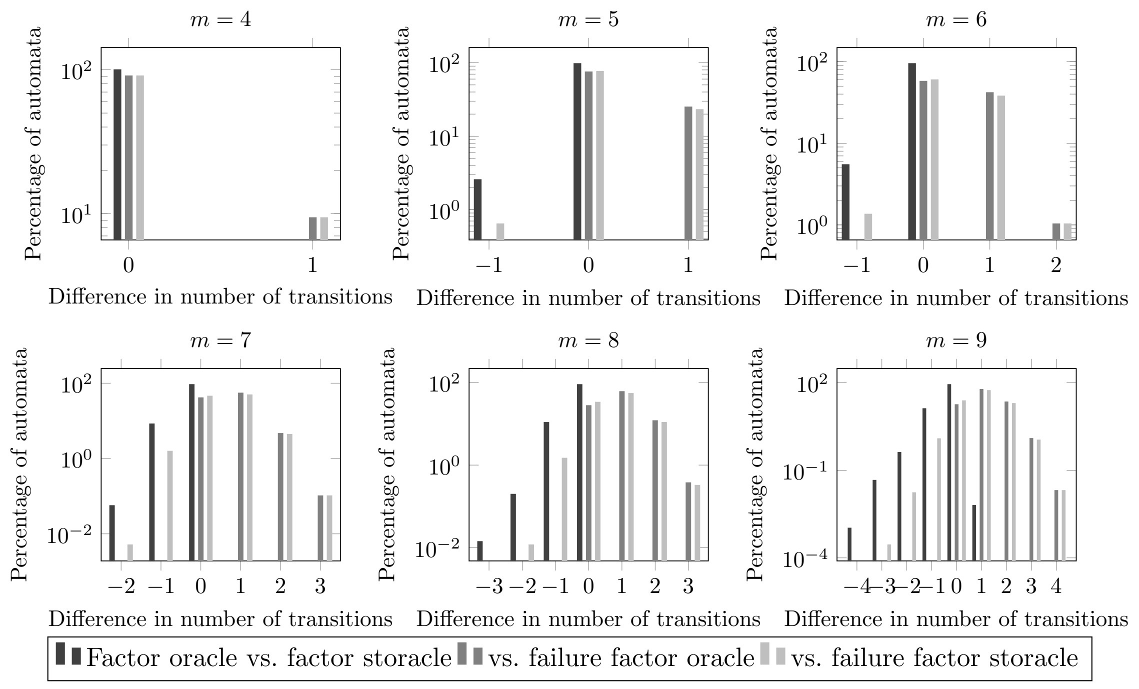


Figure 6: Distribution of difference in number of transitions for factor oracle versus each of the other three automaton types, for all words of lengths $m = 4..9$ over alphabets of size m.

factor oracle in size: no such cases occur for lengths $m = 4..8$, and it only happens in 0.006244% of cases for $m = 9$. As recollected in [6], the factor storacle was originally found by accident for $p = abcacdace$, for which it is *smaller* than the corresponding factor oracle. It thus turns out that this was somewhat of a lucky encounter as such cases are very rare. This also means that the use of a factor storacle is probably not advisable in general, compared to using a factor oracle.

– Comparing factor oracles and failure factor oracles, the experiments indicate that the failure version never performs worse than the original in terms of size, and frequently helps to reduce automaton size by 1 or 2 transitions for the keyword lengths considered. Larger savings seem to occur less frequently, although it is expected this will improve for longer keywords.
– Comparing failure factor oracles and failure factor storacles, the observation made for the non-failure cases above seem to hold true: it appears that failure factor oracles are preferable to failure factor storacles.

The second data set was obtained from [1]:

“A list of 109582 English words compiled and corrected in 1991 from lists obtained from the Interociter bulletin board. The original read.me file said that the list came from Public Brand Software. This word list includes inflected forms, such as plural nouns and the -s, -ed and -ing forms of verbs.”

As for the set of generated strings, words of length < 4 were ignored. Figure 7 shows the distribution of the set (including words of length < 4).

Figure 8 shows the distribution of factor oracle and failure factor oracle sizes for the English words for the cases of words of lengths $m = 5, 7, 9, 11, 13$, and 15.

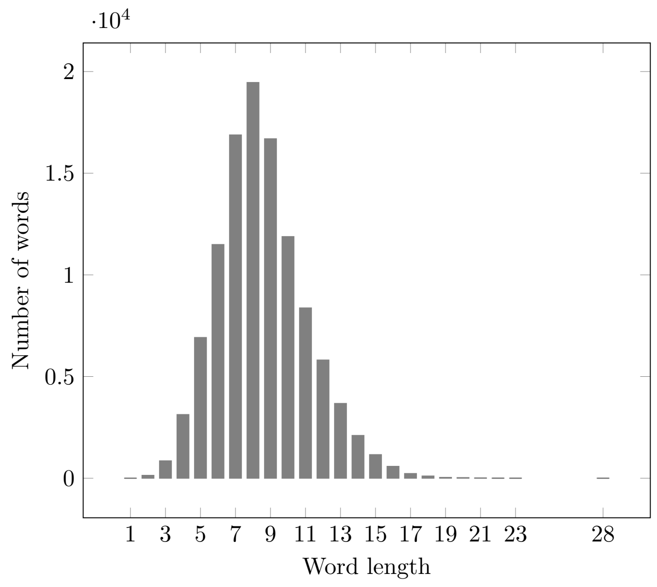


Figure 7: Distribution of lengths of the list of English words. No words of lengths 24−27 or of length over 28 occur. Words of length < 4 were not used for benchmarking.

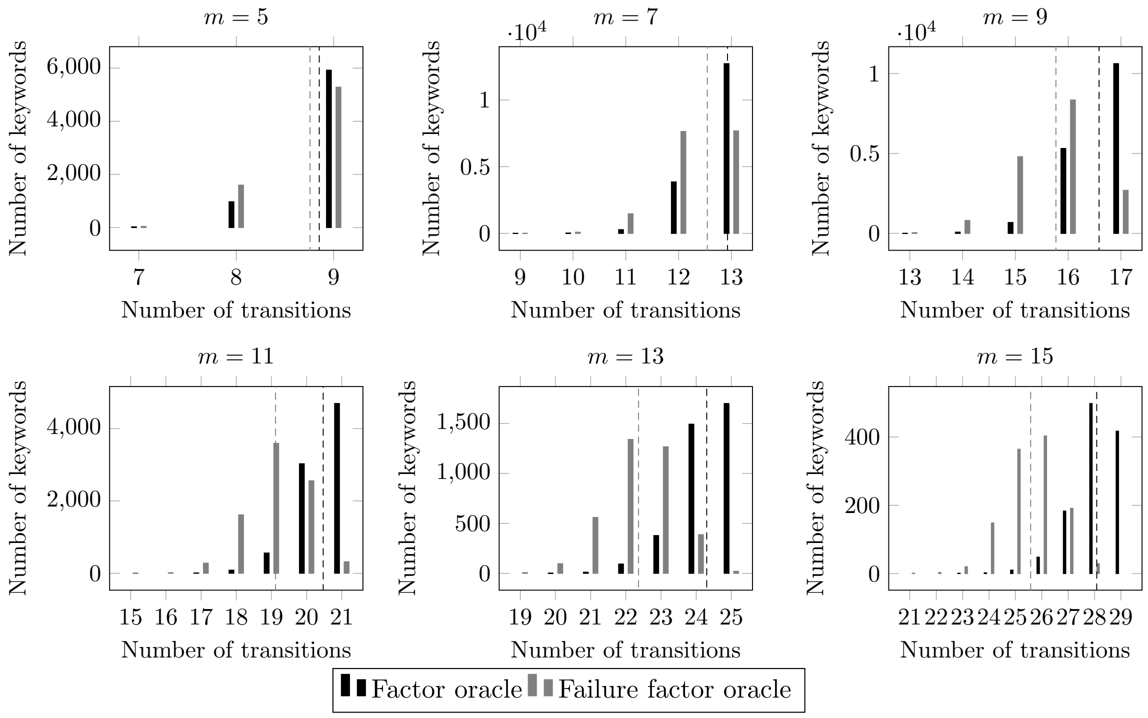


Figure 8: Distribution of number of transitions for the factor oracle and failure factor oracle automaton types, for English words of lengths $m = 5$ (6919 words), 7 (16882 words), 9 (16693 words), 11 (8374 words), 13 (3676 words), and 15 (1159 words).

Figure 9 shows the distribution of the difference in the number of transitions between the two automata kinds for the same data set. In that figure, the scale used is again a logarithmic one, and labels are in terms of percentages of all the (automata for) English words of a given length. (Results for the storacles versions are omitted, as

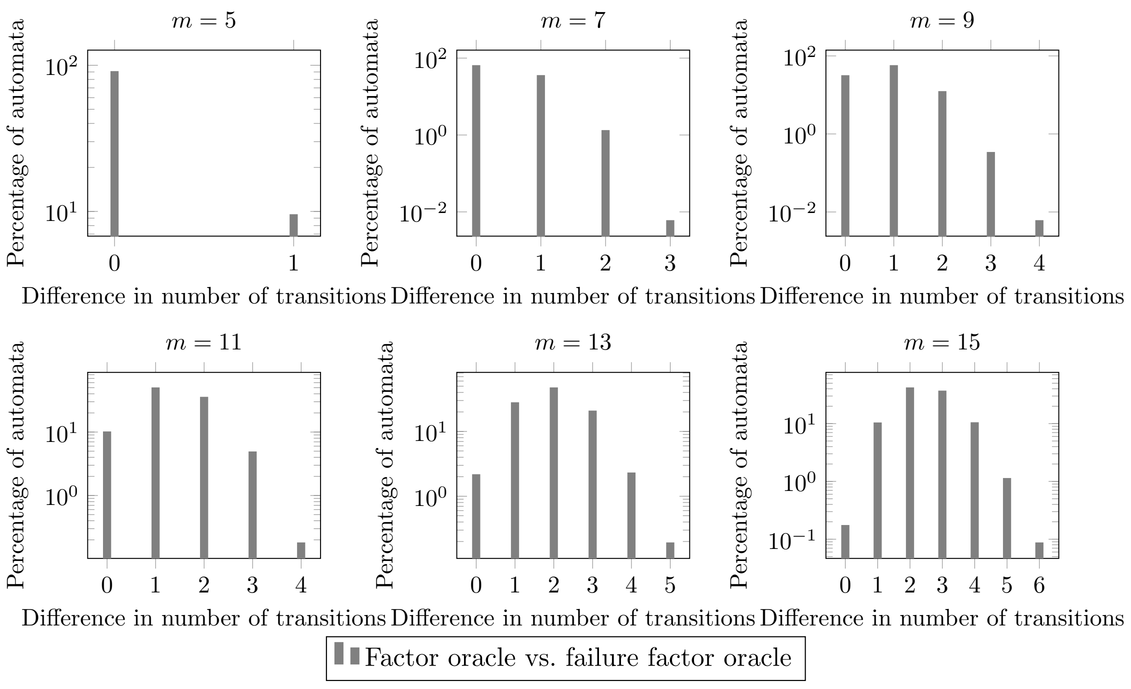


Figure 9: Distribution of difference in number of transitions for factor oracles versus failure factor oracles, for English words of lengths $m = 5$ (6919 words), 7 (16882 words), 9 (16693 words), 11 (8374 words), 13 (3676 words), and 15 (1159 words).

the discussion of results for the sets of generated strings indicated these automata are unlikely to be smaller than the corresponding oracle versions.) Comparing the results with those for the sets of generated strings, the following observations can be made:

– As was the case with the sets of generated strings, the use of failure transitions saves transitions compared to the non-failure automata versions. Comparing failure factor oracles to factor oracles, the savings are 1.07% on average for length 5, 4.932% for length 9, and 8.913% for length 15. Contrasting this with the results on the set of generated strings, the savings for a given word length are smaller, but as was the case there, the percentage of savings increases with increasing word length. This indicates that the use of failure transitions in weak factor automata may show particular promise in pattern matching for DNA processing and network intrusion detection, where longer patterns are typically being used.
– Compared to the results on the sets of generated strings, the results show fewer or no cases where the number of transitions is fairly close to the lower bound of m. This makes sense, as a language such as English has relatively few words with lots of repetition, while the generated sets contained many such strings, e.g. *aaaaaaa*, *abcabcabc*, *abccabccc* etc.
– As before for the sets of generated strings, the failure factor oracle for a given word never has more transitions, and often has fewer transitions than the corresponding factor oracle does. The distributions of differences also clearly show that with increasing word length, the distribution shifts further from a difference of 0, i.e. for longer words, the savings in number of transitions by the use of failure arcs increases.

5 Conclusions and Future Work

We have presented two new kinds of weak factor automata, based on modifications of two algorithms for constructing factor oracles and factor storacles. These new kinds of automata combine the use of failure transitions with the concept of the factor oracle and storacle, respectively.

Our experimental evaluation, with both generated strings of lengths up to 9 and English words of various lengths, showed that factor storacles and their failure versions are rarely competitive to factor oracles and their failure versions respectively. The results also show that with increasing word length, the savings in using failure factor oracles instead of factor oracles increase, to roughly between 5–9% for English words of lengths 9 and 15. Although not substantial, in restricted memory settings such savings may be useful. The increase in savings with increasing keyword length suggests more substantial savings may occur in the setting of DNA processing or intrusion detection, where patterns are typically longer than in natural languages.

A number of open questions w.r.t. the failure factor (st)oracles remain:

– What is the upper bound on the number of transitions for the failure factor oracle and failure factor storacle?
– How does the use of failure transitions in the failure automata constructions change the language accepted by the underlying non-failure automata constructions? Preferably the language accepted should not become much larger, as the use of weak factor automata in pattern matching applications becomes less efficient when more non-factors are accepted by such automata.
– While [3] introduced factor oracles for use in a particular pattern matching algorithm, it would be interesting to see how factor (st)oracles and failure factor (st)oracles can be used in a very different algorithm skeleton, such as the efficient *dead-zone* algorithm [16,18].
– The four types of oracle automata discussed here potentially accept *more* than just the factors of p, making them a type of *super* automaton. An alternative general technique for constructing super automata is discussed in [12,13,11]. How would a super automaton for the factors of p, constructed with those generalized techniques, perform against the (failure) factor (st)oracle of this paper?
– Language-preserving FDFA construction algorithms such as those in [14] and [4] could be applied to the constructed factor (st)oracles. Would the resulting failure factor (st)oracles differ significantly from those discussed in the present research?

References

1. *English wordlist*: `http://www.sil.org/linguistics/wordlists/english`.
2. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search.* Communications of the ACM, 18 1975, pp. 333–340.
3. C. Allauzen, M. Crochemore, and M. Raffinot: *Efficient Experimental String Matching by Weak Factor Recognition*, in Proceedings of the 12th conference on Combinatorial Pattern Matching, vol. 2089 of LNCS, 2001, pp. 51–72.
4. H. Björklund, J. Björklund, and N. Zechner: *Compact representation of finite automata with failure transitions*, Tech. Rep. UMINF 13.11, Umeå University, 2013.
5. J. Bourdon and I. Rusu: *Statistical properties of factor oracles.* Journal of Discrete Algorithms, 9(1) March 2011, pp. 57–66.
6. L. Cleophas and B. W. Watson: *On Factor Storacles: an Alternative to Factor Oracles?*, in Festschrift for Bořivoj Melichar, Department of Theoretical Computer Science, Czech Technical University, Prague, August 2012.

7. L. Cleophas, G. Zwaan, and B. W. Watson: *Constructing Factor Oracles*, in Proceedings of the Prague Stringology Conference 2003, Department of Computer Science and Engineering, Czech Technical University, Prague, September 2003.
8. L. Cleophas, G. Zwaan, and B. W. Watson: *Constructing Factor Oracles*, Tech. Rep. 04/01, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, January 2004.
9. L. Cleophas, G. Zwaan, and B. W. Watson: *Constructing Factor Oracles.* Journal of Automata, Languages and Combinatorics, 10(5/6) 2005, pp. 627–640.
10. L. G. W. A. Cleophas: *Towards SPARE Time: A New Taxonomy and Toolkit of Keyword Pattern Matching Algorithms*, Master's thesis, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, August 2003.
11. W. Coetser: *Finite state automaton construction through regular expression hashing*, Master's thesis, University of Pretoria, 2009.
12. W. Coetser, D. G. Kourie, and B. W. Watson: *On regular expression hashing to reduce FA size*, in Proceedings of the Prague Stringology Conference 2008, J. Holub and J. Žd'árek, eds., Czech Technical University in Prague, Czech Republic, 2008, pp. 227–241.
13. W. Coetser, D. G. Kourie, and B. W. Watson: *On regular expression hashing to reduce FA size.* International Journal of Foundations of Computer Science, 20(6) 2009, pp. 1069–1086.
14. D. G. Kourie, B. W. Watson, L. Cleophas, and F. Venter: *Failure Deterministic Finite Automata*, in Proceedings of the Prague Stringology Conference 2012, Department of Theoretical Computer Science, Czech Technical University, Prague, September 2012.
15. A. Mancheron and C. Moan: *Combinatorial Characterization of the Language Recognized by Factor and Suffix Oracles*, in Proceedings of the Prague Stringology Conference 2004, Department of Computer Science and Engineering, Czech Technical University, Prague, August 2004.
16. M. Mauch, B. W. Watson, D. G. Kourie, and T. Strauss: *Performance assessment of dead-zone single keyword pattern matching*, in Proceedings of the Symposium of the South African Institute for Computer Scientists and Information Technologists, H. Gelderblom and H. Lotriet, eds., ACM International Conference Proceedings, Centurion, South Africa, Oct. 2012, pp. 59–68.
17. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Faculty of Computing Science, Technische Universiteit Eindhoven, September 1995.
18. B. W. Watson, D. G. Kourie, and T. Strauss: *A sequential recursive implementation of dead-zone single keyword pattern matching*, in Proceedings of the 23rd International Workshop on Combinatorial Algorithms (IWOCA), S. Arumugam and W. F. Smyth, eds., vol. 7643 of Lecture Notes in Computer Science (LNCS), Heidelberg, Germany, 2012, Springer, pp. 236–248.

Degenerate String Reconstruction from Cover Arrays

(Extended Abstract)

Dipankar Ranjan Baisya, Mir Md. Faysal, and M. Sohel Rahman

AℓEDA Group
Department of Computer Science and Engineering (CSE)
Bangladesh University of Engineering and Technology (BUET)
Dhaka 1000
Bangladesh
`msrahman@cse.buet.ac.bd`

Abstract. Regularities in degenerate strings have recently been a matter of interest because of their use in the fields of molecular biology, musical text analysis, cryptanalysis and so on. In this paper, we study the problem of reconstructing a degenerate string from a cover array. We present two efficient algorithms to reconstruct a degenerate string from a valid cover array one using an unbounded alphabet and the other using minimum sized alphabet.

Keywords: degenerate strings, string reconstruction, algorithms

1 Introduction

A degenerate string (also referred to as an indeterminate string in the literature) is a generalization of a (regular) string, in which each position contains either a single character or a nonempty set of characters. The problems of degenerate pattern matching [9–11, 15] and finding regularities in degenerate strings [1, 2, 4, 8, 14] have been addressed with great enthusiasm over the last decade. Authors in [4] described the way of finding all covers of an indeterminate string in $O(n)$ time on average. Another interesting avenue for research is to explore the problem of inferring a string given some arbitrary data structure (e.g., array, tree etc.) related to some of these regularities. However, despite several results on regular string inference in the literature [5–7, 12] the problem of degenerate string inference is yet to be explored extensively. To the best of our knowledge the only work on this topic is the recent work of Nazeen et al. [13] where the authors presented string inference algorithms considering border arrays of degenerate strings. The authors in [13] mentioned that similar inference algorithms for cover arrays of degenerate strings could be worth-investigating as a future research topic. Inspired by the future research direction mentioned there, in this paper, we first present an algorithm for degenerate string reconstruction from an input cover array using an unbounded alphabet. Then we modify this algorithm such that it uses a least sized alphabet. Notably, the problem of inferring (regular) strings from cover arrays has already been tackled in [12].

The rest of this paper is organized as follows. Section 2 presents some definitions and notations. Section 3 discusses some important properties of a cover array and extends those in the context of degenerate strings. In Section 4 we describe the algorithms and related results. Finally, we briefly conclude in Section 5.

2 Preliminaries

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 $W \in \Sigma^*$, is a *substring* of X if $X = UWV$, where $U, V \in \Sigma^*$. Conversely, X is called a *superstring* of W. We denote by $X[i..j]$ the substring of X that starts at position i and ends at position j. A string $W \in \Sigma$ is a *prefix* (*suffix*) of X if $X = WY$ ($X = YW$), for $Y \in \Sigma^*$. A string W is a *subsequence* of X (or X 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 *abcabbcde*. For a given set S of strings, a string W is called a common subsequence of S if W is a subsequence of every string in S.

A string U is a *period* of X if X is a prefix of U^k for some positive integer k, or equivalently if X is a prefix of UX. The *period* of X is the shortest period of X. For example, if $X =$ *abcabcab*, then *abc*, *abcabc* and the string X itself are periods of X, while *abc* is the *period* of X.

A degenerate string is a sequence $X = X[1]X[2] \cdots X[n]$, where $X[i] \subseteq \Sigma$ for all i, and Σ is a given alphabet of fixed size. A position of a degenerate string may match more than one elements from the alphabet Σ; such a position is said to have a *non-solid* symbol. If in a position we have only one element of Σ, then we refer to this position as *solid.* The definition of length for degenerate strings is the same as for regular strings: a degenerate string X has length n, when X has n positions, where each position can be either solid or non-solid. We represent non-solid positions using [..] and solid positions omitting [..]. The example in Table 1 identifies the solid and non-solid positions of a degenerate string.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$X =$	a	a	[abc]	a	[ac]	b	c	a	a	[ac]	b	a	c	[abc]	a	[bc]

Table 1. An example of a degenerate string

Table 1 presents a degenerate string having non-solid symbols at Positions 3, 5, 10, 14 and 16. The rest of the positions contain solid symbols. Let λ_i, $|\lambda_i| \geq 2, 1 \leq i \leq s$, be pairwise distinct subsets of Σ. We form a new alphabet $\Sigma' = \Sigma \cup \lambda_1, \lambda_2, \ldots, \lambda_s$ and define a new relation *match* ($\approx$) on Σ' as follows:

Type 1. for every $\mu_1, \mu_2 \in \Sigma, \mu_1 \approx \mu_2$ if and only if $\mu_1 = \mu_2$;
Type 2. for every $\mu \in \Sigma$ and every $\lambda \in \Sigma' - \Sigma, \mu \approx \lambda$ if and only if $\mu \in \lambda$;
Type 3. for every $\lambda_i, \lambda_j \in \Sigma' - \Sigma, \lambda_i \approx \lambda_j$ if and only if $\lambda_i \cap \lambda_j \neq \emptyset$.

Observe that the relation match ($\approx$) is reflexive and symmetric but not necessarily transitive. For example, if $\lambda = [a, b]$, then we have $a \approx \lambda$ and $b \approx \lambda$. But clearly $a \not\approx b$.

From the example in Table 1, we have a Type 1 match between Positions 2 and 4, as both positions are solid and contain the letter a. Positions 3 and 6 give a match of Type 2 as the letter b is contained in the non-solid symbol [abc]. A match of Type 3 can be found between Positions 3 and 5, as the symbols at these two positions have a and c common. Although Positions 5 and 3 match and Positions 3 and 6 match, Positions 5 and 6 do not match, illustrating the non-transitivity of the matching operation for degenerate strings.

Cover is an interesting regularity in strings that in some sense generalizes the concept of quasiperiodicity [3]. We say that a string S covers a string U if every letter of U is contained in some occurrence of S as a substring of U. Then S is called a cover of U. Clearly, S must be a (proper) substring of to be a (proper) cover of U. Although a string can be considered to be a cover of itself, we follow the convention in the literature and consider only the proper covers. The *cover array* C of a regular string $X[1..n]$, is a data structure used to store the length of the longest proper cover of every prefix of X. So for all $i \in \{1..n\}$, $C[i]$ stores the length of the longest proper cover of $X[1..i]$ or 0. In fact, since every cover of any cover of X is also a cover of X, it turns out that, the cover array C compactly describes all the covers of every prefix of X. For every prefix $X[1..i]$ of X, the following sequence

$$C^1[i], C^2[i], \ldots, C^m[i] \tag{1}$$

is well defined and monotonically decreasing to $C^m[i] = 0$ for some $m \geq 1$ and this sequence identifies every cover of $X[1..i]$. Here, $C^k[i]$ is the length of the kth longest cover of $X[1..i]$, for $1 \leq k \leq m$.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$X =$	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a
$C =$	0	0	0	0	0	3	0	3	0	5	6	0	5	6	0	8	9	10	11

Table 2. Cover array of *abaababaabaababaaba*

From Table 2 we can see that, cover array of X, has the entries $C[19] = 11,\ C[11] = 6,\ C[6] = 3$ *and* $C[3] = 0$ representing the three covers of X having length $11,\ 6$ and 3 respectively.

The definition for cover is the same for both regular and degenerate strings. However, the definition of the cover array for a degenerate string changes. For a degenerate string, $C[i]$ stores the list of the lengths of the covers of $X[1..i]$. More elaborately, each $C[i]$ is a list $\langle C^p[i] \rangle$ such that $1 \leq p \leq |C[i]|$, where $C^p[i]$ denotes the pth largest cover of $X[1..i]$. As the matching operations of degenerate strings are not transitive, cover array algorithms for regular strings cannot be readily extended to degenerate strings.

Index	1	2	3	4	5	6
$X =$	a	b	a	[ab]	[ab]	a
$C =$	0	0	0	2	3	4
					2	3

Table 3. Cover array of $aba[ab][ab]a$

Also, Sequence 1, does not fully apply to the covers of degenerate strings. From Table 2 and Sequence 1, the degenerate string X should have two covers of length 4 and 2, as $C[6]$ contains 4 and $C[4]$ contains 2. However, as can be seen from Table 2, X has covers of length 4 and 3, but not of length 2. So for covers of degenerate strings Sequence 1 gives wrong information. Note that, the space requirement for representing the cover array of a degenerate string of length n is $O(n^2)$.

3 Basic Validation of a cover array

In this section, we discuss some basic properties of a valid cover array. The properties discussed here are mostly in the context of reconstruction of a (degenerate) string from a given cover array while we scan/proceed one position at a time from left to right. i.e., in an online fashion. Further validation properties will be discussed in the following section where we describe the algorithms. For $i \geq 2$, we say an integer j is a *candidate-length* (i.e., "candidate" to be the length) of a cover of $X[1..i]$, if $j \in \{1, \ldots, i-1\}$. Thus candidate-lengths of covers of $X[1..i]$ is $\pi_i = \{1, 2, \ldots, i-1\}$. We say that an array $C[1..n]$ is a valid cover array if and only if it is the cover array of at least one degenerate string X with n positions (i.e., having length n). We also use the notion of an equivalence of strings based on their cover arrays as follows. We say that two strings $X_1[1..n]$ and $X_2[1..n]$ are *C-Equivalent* if and only if both of them have the same cover array $C[1..n]$. Given a degenerate string X of length n on alphabet Σ, we define $\Sigma_i \subseteq \Sigma$ to be the set of symbols used by the prefix $X[1..i]$. Further we say that a symbol $\psi \in \Sigma_i - \Sigma_{i-1}$ is *required by* $C[i]$.

Clearly, the only cover of $X[1]$ is necessarily an empty word. Thus we must have $C[1] = 0$, irrespective of any strings. Also, as has been discussed above, the list of lengths of the nonempty covers $C[i]$ of $X[1..i]$ is taken from π_i. We now present the following useful observation, argument and theorem.

Observation 1 *Suppose $C[1..n]$ is the cover array of a string $X[1..n]$. Then the following hold true.*

a. If $1 \in C[i]$, then $1 \in C[j]\ \forall\ 1 < j \leq i-1,\ i > 1$.
b. If $f \in C[i]$, and $f \in C[j]$ such that $j \geq i$ then $j - i \leq f$.
c. If $C[k] = 0$ for $1 \leq k \leq m$, then $C[m+1] \leq m-1$.

Lemma 1. *Suppose $C[1..n]$ is the cover array of a string $X[1..n]$. If $1 \in C[i]$, then $\{1, 2, \ldots, i-1\} \subseteq C[i],\ i > 1$.*

Proof. Proof will be provided in the journal version. □

Lemma 2. *Suppose we have a cover array C. Suppose we have correctly reconstructed a degenerate string X_1 of length $i-1$ based on $C[1..i-1]$. Also assume that we have also correctly reconstructed X_2 of length i for $C[1..i]$. Further, suppose that $Z = \Sigma_{X_2} - \Sigma_{X_1}$. Then the following hold true:*

a. Suppose, $|Z| = 1$ and $Z = \{\psi\}$. Also, assume that ψ is required for $C^p[i]$. Then ψ can only be put at the following positions of $X_1 : \{C^p[i], 2 \times C^p[i], 3 \times C^p[i] \ldots\}$ to get X_2
b. Suppose $|Z| > 1$ and $Z = \{\psi_1, \ldots, \psi_k\}$. Also, assume that $\psi_j,\ 1 \leq j \leq k$ is required for $C^{p_j}[i]$
   i. Then $\psi_1, \psi_2, \ldots, \psi_k$ can only be put at the following positions of X_1: $\{C^{p_j}[i], 2 \times C^{p_j}[i], 3 \times C^{p_j}[i], \ldots\}$
   ii. Assume that λ is the non-solid character containing all letters of Z. We use λ_k to denote the character containing $\psi_1, \ldots, \psi_k$. So, $\lambda_1 = \psi_1$ and hence is a solid character and $\lambda_k = [\psi_1..\psi_k]$. Further assume that $X_2^k = X_1'[1..i-1]\lambda_k$. We get $X_1'[1..i-1]$ by placing the new characters $\psi_i,\ \psi_i \in \lambda_k$ at the aforementioned specific Positions of X_1. Then all of $X_2^i[1..i-1],\ 1 \leq i \leq k$ along with X_1 are $C-Equivalent$.
c. X_1 and $X_2[1..i-1]$ are $C-Equivalent$.

Proof. Proof will be provided in the journal version. □

Now we are ready to present and prove the following important theorem.

i	1	2	3	4	5	6
$C[i]$	0	1	2	3	4	2
			1	2	3	
				1	2	
					1	

Table 4. An example of cover array of Degenerate string

i	1	2	3	4	5
$X[i]$	a	a	a	a	a

Table 5. Degenerate string of cover array up to Position 5

i	1	2	3	4	5	6
$X[i]$	a	a	a	a	a	b
		b		b		

Table 6. Degenerate string of cover array up to Position 6

Theorem 2 *Suppose $C[1..n]$ is an array of $n \geq 1$ lists of integers. If the following condition (Condition 1) is satisfied, then $C[1..n]$ is a cover array of some degenerate string $X[1..n]$.*

Condition 1

a. $C[1] \ = \ 0$
b. $C[i] \subseteq \{0\} \cup \pi_i, \ for \ 2 \leq i \leq n.$
  i. If $X[1..i]$ has the empty word for its only cover then we have $C[i] = \{0\}$
  ii. If $X[1..i]$ has nonempty covers then $C[i] = \{j | j \in \pi_i$ and $X[i] \approx X[j]\}$

Proof. Proof will be provided in the journal version. □

4 Our Algorithms

4.1 *CrAyDSRUn*

Our problem is to reconstruct a degenerate string of length n, given a valid cover array C. In this section, we focus on an unbounded alphabet and propose an algorithm called ***CrAyDSRUn*** (Cover Array Degenerate String Reconstruction from Unbounded Alphabet) for this problem. Given an array $C[1..n]$, ***CrAyDSRUn*** determines whether $C[1..n]$ is a valid cover array for at least one degenerate string and if so, it constructs one such degenerate string. Before presenting the algorithm, we first need to present some relevant definitions and notions.

Assume that, we have successfully reconstructed $X[1..i-1]$. We use ψ_i to denote the new set of characters introduced in $X[i]$, i.e., $\psi_i = \Sigma_i - \Sigma_{i-1}$. Now, we want to extend $X[1..i-1]$ to get $X[1..i]$ based won $C[1..i]$. Suppose that $a \in C[i]$. So, we must have a cover of length a for $X[1..i]$. Also if we need a new character, we have to place that it at Position i and other necessary positions of $X[1..i-1]$ (See Lemma 2). We denote by A'_i the set of symbols that are not allowed only at Position i, i.e., $A'_i \ = \ \bigcup_{j \in \pi_i - C[i]} X[j]$. On the other hand, we denote by A_i the set of symbols that can be assigned to $X[i]$. We now have the following lemma.

Lemma 3. *For every degenerate string* $X[1..i]$ *the following hold true:*

a. If $C^p[i] \neq 0$ *for* $1 \leq p \leq |C[i]|$ *then*

$$X[i] \approx X[C^p[i]],\ C^p[i] \in \pi_i \text{ and } A_i = \psi_i \cup \left(\bigcup_{1 \leq p \leq |C[i]|} (X[C^p[i]] - A'_i) \right)$$

b. if $C^p[i] = 0$ *is the only entry of* $C^p[i]$, *then* $C^p[i] \notin \pi_i,\ \ A_i = \psi_i$ *and* $|A_i| = |\psi_i| = 1$

Proof. Proof will be provided in the journal version. □

We note that our string inference algorithm follows a similar approach used in [13] to reconstruct degenerate strings from border arrays. The main differences lie in manipulating A_i, A'_i, validity checking of X and placing appropriate characters at appropriate positions. The steps of ***CrAyDSRUn*** are formally presented in Algorithm 1 (in Appendix). We assume that, we have an array α representing an unbounded alphabet from which we take the *basic* letters i.e., the non-degenerate letters from the alphabet Σ. The ***CrAyDSRUn*** algorithm takes an array $C[1..n]$ as input. It first checks the trivial validity condition whether $C[1] = 0$ or not; subsequently for every position $2 \leq i \leq n$, it checks whether $C^p[i] \in \pi_i,\ i \leq p \leq |C[i]|$. Algorithm ***CrAyDSRUn*** returns the input cover array as invalid as soon as it finds a violation of the conditions checked above. As long as the result of the above checking is positive, Algorithm ***CrAyDSRUn*** constructs A'_i and A_i for each position $2 \leq i \leq n$. To keep track of the alphabet size of each prefix $X[1..i]$, our algorithm uses an array k where $k[i] = |\Sigma_i|$.

To manipulate A'_i, we use function $getInvalidChar(Position,\ CoverValue)$ that takes two parameters. $Position$ refers to the position of the cover array and $CoverValue$ refers to one of the values of that position. To manipulate A_i, we use function $getProbableValidChar(Position,\ CoverValue)$. If a $CoverValue$ appears for the first time in C at Position i, then our algorithm will extend the string such that there is a cover of length equal to $CoverValue$ by putting the characters in $X[covervalue]$ at $X[i]$ provided that the positions of $(X[i-1]$ *and* $X[CoverValue - 1])$, $(X[i-2]$ *and* $X[CoverValue - 2]), \ldots,$ $(X[i - (CoverValue - 1)]$ *and* $X[1])$ have at least one common character. Otherwise, the cover array is invalid and the function returns indicating that (see Lemma 4 later).

If a $Covervalue$ appears previously in C, then our algorithm uses a variable $lastpos$ to hold the immediate previous position of $CoverValue$ in the cover array. In this case, our algorithm will extend the string such that there is a cover of length equal to $CoverValue$ by putting the common characters in $X[CoverValue]$ *and* $X[lastpos]$ at Position i provided that the positions of $(X[CoverValue-1],\ X[lastpos-1]$ *and* $X[i-1]), \ldots, (X[1],\ X[lastpos-(CoverValue-1)]$ *and* $X[i-(CoverValue-1)])$ have at least one common character. Otherwise, the cover array is invalid and the function returns indicating that (see Lemma 5 later).

Now, we focus our attention on how we can effectively check whether multiple positions have common characters among them. To find whether there exists common character at two positions, we use Bit Vector technique [4]. In our algorithm, we use ν to indicate Bit Vector. Although we are reconstructing over an unbounded alphabet, when we compare between two positions for common characters we have already placed characters in those positions previously. We will also create the Bit Vector again if new characters arrive at that position. If two positions have common characters then we save this record in a two dimensional array H. For example, if

Positions a and b have common characters then we mark the entry of $H[a][b]$. We will update H incrementally. For example, for Position 2, we need to check Position 1 and 2 whether they have common characters or not. Again, for Position 3, we need to check Positions 1 and 3 and Positions 2 and 3 whether they have common characters or not. So we can see for Position 3, there are two entries to update in H namely $H[1][3]$ and $H[2][3]$. It is notable that for any Position i, all the entries of $H[1][1]$, $H[1][2]$, …, $H[i-1][i-1]$ will remain unchanged. Because even if new character arrives, it will be placed in the positions stated in Lemma 2 and according to that lemma $X[1..i-1]$ will still be $C - Equivalent$. Now for Position n, we have at most $n-1$ entries such as $H[1][n], H[2][n], H[3][n], \dots, H[n-1][n]$ to update. That is how, we have pre-computed H while placing characters in X. If we need to check whether there exists common characters between three positions namely a, b, c, we need to AND the Bit Vector of these three positions. If the result of this AND is non-zero then we can say there exists common character between Positions a, b and c.

In order to place characters of A_i in proper positions our algorithm uses function $PlaceCharInProperPosition(Position, CoverValue, necessarychar)$. Here $necessarychar$ indicates the necessary character to fill the positions of X. Whenever some $C^p[i] \neq 0$, ***CrAyDSRUn*** puts a character $v \in A_i$ into $X[i]$; v is also included in $X[C^p[i]]$ and $X[j]$ *where* $2 \leq j < i$ *and* $C^p[i] \in C[j]$ if $v \notin \Sigma_i$. It is notable that we have included character set $\Sigma_i - A_i - A'_i$ at Position i. We can safely add those characters because they are not invalid at Position i. By adding these characters we make sure that we do not need to add any more characters in the previous positions of Position i if no new characters arrive. We now report the following Observations which basically support the correctness of our approach.

Lemma 4. *Given a cover array $C[1..n]$, suppose $C[i] = \ell$ (we need to have a cover of length ℓ at Position i) such that $\ell \notin \{C[1] \cup C[2] \cup \cdots \cup C[i-1]\}$ and $X[1] \cap X[i-(\ell-1)] \neq \phi$, $X[2] \cap X[i-(\ell-2)] \neq \phi, \dots,$ $X[\ell-1] \cap X[i-1] \neq \phi$ then we must include $X[\ell] - A'_i$ at position i. If in any one of the above intersection returns ϕ then the input cover array is not valid.*

Proof. Proof will be provided in the journal version. □

Lemma 5. *Given a cover array $C[1..n]$, suppose $C[i] = \ell$ (we need to have a cover of length ℓ at Position i) such that $\ell \in \{C[1] \cup C[2] \cup \cdots \cup C[i-1]\}$ and let p be the immediate previous position of i where $\ell \in C[p]$ and $1 \leq p \leq (i-1)$ and $\{X[1] \cap X[p-(\ell-1)] \cap X[i-(\ell-1)]\} \neq \phi$, $\{X[2] \cap X[p-(\ell-2)] \cap X[i-(\ell-2)]\} \neq \phi, \dots,$ $\{X[\ell-1] \cap X[p-1] \cap X[i-1]\} \neq \phi$, then we must include $X[\ell] \cap X[p] - A'_i$ at position i. If in any one of the above intersection returns ϕ then the input cover array is invalid. Because if any one of the intersection returns ϕ, then we can not have a cover of length ℓ at Position i.*

Proof. Proof will be provided in the journal version. □

Based on the above discussions we have the following theorem.

Theorem 3 *Given a valid cover array $C[1..n]$, the algorithm* **CrAyDSRUn** *checks for its validity at every position and as long as it is valid it reconstructs a degenerate string $X[1..n]$ on an unbounded alphabet for which $C[1..n]$ is a cover array.*

Proof. Proof will be provided in the journal version. □

Now we focus on the complexity of algorithm ***CrAyDSRUn***. We start with the following theorem.

Theorem 4 *Algorithm* **CrAyDSRUn** *runs in $O(N\ |\Sigma|)$ time, where N is the product of string length and maximum list length of cover array C.*

Proof. Proof will be provided in the journal version. □

Theorem 5 *Algorithm* **CrAyDSRUn** *runs in linear time on average.*

Proof. Proof will be provided in the journal version. □

Algorithm 1 ***CrAyDSRUn***(C,n)

```
1: if $C[1] \neq \{0\}$ then
2:     return $C$ invalid at index $1$
3: end if
4: $X[1] \leftarrow \{\alpha[1]\}$
5: $k[1] \leftarrow 1$
6: for $i \leftarrow 2$ to $n$ do
7:     $k[i] \leftarrow k[i-1]$
8:     $A \leftarrow \phi$
9:     $X[i] \leftarrow \phi$
10:    $\pi \leftarrow \{1..i-1\}$
11:    $A'_i \leftarrow \phi$
12:    for all $k,\ \pi - C[i]$ do
13:        $getInvalidChar(i,k)$
14:    end for
15:    $probablevalidchar \leftarrow \phi$
16:    for $j \leftarrow 1$ to $|C[i]|$ do
17:        if $C^j[i] \neq \{0\}$ then
18:            if $C^j[i] \notin \pi_i$ then
19:                return $invalid\ at\ index\ i$
20:            end if
21:            if $1 \in C[i]$ then
22:                if $Observation\ 1a\ \&\ Lemma\ 1\ not\ satisfied$ then
23:                    return $invalid\ at\ index\ i$
24:                end if
25:            end if
26:            if $Observation\ 1b\ not\ satisfied\ at\ position\ i$ then
27:                return $invalid\ at\ index\ i$
28:            end if
29:            if $Observation\ 1c\ not\ satisfied\ at\ position\ i$ then
30:                return $invalid\ at\ index\ i$
31:            end if
32:            $getProbableValidChar(i, C^j[i])$
33:            $A \leftarrow probablevalidchar - A'_i$
34:            if $A \neq \phi$ then
35:                $X[i] \leftarrow X[i] \cup A$
36:                if $\Sigma_i - A_i - A'_i \neq \phi$ then
37:                    $Add\ \Sigma_i - A_i - A'_i\ at\ position\ i$
38:                end if
39:                $update\ \nu\ position\ i$
40:            else
41:                $k[i] \leftarrow k[i-1]+1$
42:                $A \leftarrow \{\alpha[k[i]]\}$
43:                $placecharacterinproperposition(i, C^j[i], A)$
44:            end if
45:        else
46:            $k[i] \leftarrow k[i-1]+1$
47:            $A \leftarrow \{\alpha[k[i]]\}$
48:            $X[i] \leftarrow X[i] \cup A$
49:            $update\ \nu\ at\ postition\ i$
50:        end if
51:    end for
52: end for
53: return $X$
```

```
function getInvalidChar($position, covervalue$)
    $d \leftarrow covervalue - 1$
    $k \leftarrow position - 1$
    $flag \leftarrow 0$
    $flag2 \leftarrow 0$
    $lastpos \leftarrow 0$
    $lastpos \leftarrow find\ immediate\ previous\ position\ i\ of\ position\ where\ covervalue \in\ c[i] \&\ 1 \leq i \leq position - 1$
    if $covervalue = 1$ then
        if $covervalue \in c[k]$ then
            $A'_i \leftarrow\ A'_i \cup X[covervalue]$
            return $A'_i$
        else
            return $false$
        end if
    else if $lastpos = 0$ then
        for $i \leftarrow 1\ to\ covervalue - 1$ do
            $p \leftarrow \phi$
            $b \leftarrow position \% covervalue$
            $p \leftarrow X[i] \cap X[i + b]$
            if $p = \phi$ then
                $flag \leftarrow 1$
                $break$
            end if
        end for
    else if $lastpos \neq\ 0$ then
        $k \leftarrow position\ -\ covervalue\ +\ 1$
        $j \leftarrow\ 1$
        for $i \leftarrow\ lastpos - covervalue + 1\ to\ lastpos - 1$ do
            $p \leftarrow \phi$
            $p \leftarrow X[i] \cap X[k] \cap X[j]$
            $j + +$
            $k + +$
            if $p = \phi$ then
                $flag2 \leftarrow 1$
                $break$
            end if
        end for
    end if
    if $lastpos\ = 0 \&\ flag = 0$ then
        $A'_i \leftarrow A'_i \cup X[covervalue]$
        return $A'_i$
    else if $lastpos\ = 0 \& flag = 1$ then
        return $false$
    end if
    if $lastpos \neq 0 \&\ flag2 = 0$ then
        $A'_i \leftarrow A'_i \cup (X[covervalue] \cap\ X[lastpos])$
        return $A'_i$
    else if $lastpos \neq 0 \&\ flag2 = 1$ then
        return $false$
    end if
end function
```

```
function getProbaleValidChar($position, covervalue$)
    $d \leftarrow covervalue - 1$
    $k \leftarrow position - 1$
    $lastpos \leftarrow 0$
    $flag \leftarrow 0$
    $flag2 \leftarrow 0$
    $lastpos \leftarrow find\ immediate\ previous\ position\ i\ of\ position\ where\ covervalue \in\ c[i] \&\ 1 \leq i \leq position - 1$
    if $covervalue = 1$ then
        if $covervalue \in c[k]$ then
            $probablevalidchar \leftarrow\ probablevalidchar \cup X[covervalue]$
            return
        else
            return $invalid\ at\ position$
        end if
    else if $lastpos = 0$ then
        for $i \leftarrow 1\ to\ covervalue - 1$ do
            $p \leftarrow \phi$
            $b \leftarrow position \% covervalue$
            $p \leftarrow X[i] \cap X[i + b]$
            if $p = \phi$ then
```

```
                $flag \leftarrow 1$
                $break$
            end if
        end for
    else if $lastpos \neq 0$ then
        $k \leftarrow position - covervalue + 1$
        $j \leftarrow 1$
        for $i \leftarrow lastpos - covervalue + 1$ to $lastpos - 1$ do
            $p \leftarrow \phi$
            $p \leftarrow X[i] \cap X[k] \cap X[j]$
            $j++$
            $k++$
            if $p = \phi$ then
                $flag2 \leftarrow 1$
                $break$
            end if
        end for
    end if
    if $lastpos = 0 \& flag = 0$ then
        $probablevalidchar \leftarrow probablevalidchar \cup X[covervalue]$
        return $probablevalidchar$
    else if $lastpos = 0 \& flag = 1$ then
        return $invalid\ at\ position$
    end if
    if $lastpos \neq 0 \& flag2 = 0$ then
        $probablevalidchar \leftarrow probablevalidchar \cup (X[covervalue] \cap X[lastpos])$
        return $probablevalidchar$
    else if $lastpos \neq 0 \& flag2 = 1$ then
        return $invalid\ at\ position$
    end if
end function
function PlaceCharInProperPosition($position, covervalue, necessarychar$)
    $b \leftarrow position \% covervalue$
    if $b \neq 0$ then
        for ($i \leftarrow covervalue$; $i \leq position$; $i \leftarrow i + b$) do
            $X[i] \leftarrow X[i] \cup necessarychar$
            $update\ \nu\ at\ position\ i$
        end for
    else if $b = 0$ then
        for ($i \leftarrow covervalue$; $i \leq position$; $i \leftarrow i + covervalue$) do
            $X[i] \leftarrow X[i] \cup necessarychar$
            $update\ \nu\ at\ position\ i$
        end for
    end if
end function
```

Table 7 shows an example run of the algorithm.

4.2 *CrAyDSRin*

Now we present a modified version of algorithm ***CrAyDSRUn*** that reconstructs a degenerate string using a minimum sized alphabet. We call this algorithm ***CrAyDSRin*** (Cover Array Degenerate String Reconstruction from Minimal Alphabet). As before, suppose we are reconstructing a degenerate string $X = X[1..n]$ from a cover array $C[1..n]$ and assume that we have successfully reconstructed $X[1..i-1]$. Now, we want to extend $X[1..i-1]$ to get $X[1..i]$ based on $C[1..i]$. Recall from Section 4.1 that, we use A'_i and A_i to denote the set of symbols that, respectively, are not allowed and allowed to be assigned to $X[i]$. Now we present an extended version of Lemma 3.b below.

Lemma 6. *For every degenerate string $X[1..i]$, if $C^p[i] = 0$ is the only entry in $C[i]$, then $C^p[i] \notin \pi_i$ and $A_i = \psi_i \cup (\Sigma_{i-1} - A'_i)$.*

Proof. Proof will be provided in the journal version. □

Algorithm 2 CrAyDSRIn(C,n)

```
if $C[1] \neq \{0\}$ then
    return $C$ invalid at index 1
end if
$X[1] \leftarrow \{\alpha[1]\}$
$k[1] \leftarrow 1$
for $i \leftarrow 2$ to $n$ do
    $k[i] \leftarrow k[i-1]$
    $A \leftarrow \phi$
    $X[i] \leftarrow \phi$
    $\pi \leftarrow \{1..i-1\}$
    $A'_i \leftarrow \phi$
    for all $k,\ \pi - C[i]$ do
        $getInvalidChar(i,k)$
    end for
    $probablevalidchar \leftarrow \phi$
    for $j \leftarrow 1$ to $|C[i]|$ do
        if $C^j[i] \neq \{0\}$ then
            if $C^j[i] \notin \pi_i$ then
                return invalid at index $i$
            end if
            if $1 \in C[i]$ then
                if Observation 1a & Lemma 1 not satisfied then
                    return invalid at index $i$
                end if
            end if
            if Observation 1b not satisfied at position $i$ then
                return invalid at index $i$
            end if
            if Observation 1c not satisfied at position $i$ then
                return invalid at index $i$
            end if
            $getProbableValidChar(i, C^j[i])$
            $A \leftarrow probablevalidchar - A'_i$
            if $A \neq \phi$ then
                $X[i] \leftarrow X[i] \cup A$
                if $\Sigma_i - A_i - A'_i \neq \phi$ then
                    Add $\Sigma_i - A_i - A'_i$ at position $i$
                end if
                update $\nu$ position $i$
            else
                if $j = 1$ then
                    $k[i] \leftarrow k[i-1] + 1$
                    $A \leftarrow \{\alpha[k[i]]\}$
                else
                    for $m \leftarrow 1$ to $j-1$ do
                        if $C^j[i] \in C[C^m[i]]$ then
                            $A \leftarrow A \cup (X[C^m[i]] - A'_i)$
                            break
                        end if
                    end for
                    if $m = j$ then
                        $k[i] \leftarrow k[i] + 1$
                        $A \leftarrow \{\alpha[k[i]]\}$
                    end if
                end if
                $placecharainproperposition(i, C^j[i], A)$
            end if
        else
            $A \leftarrow \alpha[1..k[i]] - A'_i$
            if $A = \phi$ then
                $k[i] \leftarrow k[i] + 1$
                $A \leftarrow \{\alpha[k[i]]\}$
            end if
            $X[i] \leftarrow X[i] \cup A$
            update $\nu$ at position $i$
        end if
    end for
end for
return $X$
```

Itn	i	1	2	3	4	5	6	7	8	9	$k[i]$	Explanation
0	$X[i]$	a									k[1]=1	
1	$X[i]$	a	a								k[2]=1	$\pi_2 = \{1\}$
												$A'_2 = \phi,\ A_2 = \{a\}$
2	$X[i]$	a	a	a							k[3]=1	$\pi_3 = \{1, 2\}$
												$A'_3 = \phi,\ A_3 = \{a\}$
3	$X[i]$	a	a	a	a						k[4]=1	$\pi_4 = \{1, 2, 3\}$,
												$A'_4 = \phi,\ A_4 = \{a\}$
4	$X[i]$	a	a	a	a	a					k[5]=1	$\pi_5 = \{1, 2, 3, 4\}$,
												$A'_5 = \phi,\ A_5 = \{a\}$
5	$X[i]$	a	a	a	a	a	b				k[6]=3	$\pi_6 = \{1, 2, 3, 4, 5\},\ A'_6 = \{a\}$
					b		c					for $c^1[6] = 4$ place new symbol 'b' in position 4, and 6
			c		c							for $c^2[6] = 2$ place new symbol 'c' in position 2,4, and 6
												$A_6 = \psi_6 = \{b, c\}$
6	$X[i]$	a	a	a	a	a	b	a			k[7]=3	$\pi_7 = \{1, 2, 3, 4, 5, 6\},\ A'_7 = \phi$
					b		c	b				$A_7 = \{a\},\ \Sigma_7 - A_7 - A'_7 = \{b, c\}$
			c		c			c				
7	$X[i]$	a	a	a	a	a	b	a	b		k[8]=3	$\pi_8 = \{1, 2, 3, 4, 5, 6, 7\},\ A'_8 = \{c\}$,
			c		b		c	b	a			$A_8 = \{b\},\ \Sigma_8 - A_8 - A'_8 = \{a\}$
					c			c				
8	$X[i]$	a	a	a	a	a	b	a	b	d	k[9]=4	$\pi_9 = \{1, 2, 3, 4, 5, 6, 7\},\ A'_9 = \{a, b, c\}$
			c		b		c	b	a			$A_9 = \psi_9 = \{d\}$
					c			c				

Table 7. An example run of algorithm ***CrAyDSRUn***

The algorithm ***CrAyDSRin*** is formally presented in Algorhitm 2. ***CrAyDSRin*** algorithm works exactly like ***CrAyDSRUn*** algorithm except for that it computes A_i slightly differently. In particular, it computes A_i following Lemmas 3.a and 6 (instead of Lemma 3.b).

Lemma 7. *Let $X[1..n]$ be a degenerate string and $k[1..n]$ be the array computed by the algorithm* **CrAyDSRin** *given a valid cover array C. Then, for $1 \leq i \leq n$ we have $k[i] = |\Sigma_{i-1} \cup A_i| = k[i-1] + |A_i| - |\Sigma_{i-1} \cap A_i|$.*

Proof. The proof immediately follows from the algorithm ***CrAyDSRin*** and Lemma 6. □

Lemma 8. *Suppose given a valid cover array $C[1..n]$, the algorithm* **CrAyDSRin** *returns an degenerate string $X[1..n]$ and computes the array $k[1..n]$. Then, the minimum cardinality of an alphabet required to build each prefix $X[1..i]$ is equal to $k[i]$.*

Proof. Proof will be provided in the journal version. □

The above discussion can be summarized in the following theorem.

Theorem 6 *Given a cover array $C[1..n]$ the algorithm* **CrAyDSRin** *checks for its validity at every position and as long as it is valid it reconstructs an indeterminate string $X[1..n]$ on a minimum sized alphabet for which $C[1..n]$ is a cover array.*

The runtime analysis of algorithm ***CrAyDSRin*** follows readily from the analysis of algorithm ***CrAyDSRUn***. The only extra work the former does is the calculation of $\Sigma_{i-1} - A'_i$ which can be done in $O(m|\Sigma|)$ time. Therefore we have the following results.

Theorem 7 *Algorithm* **CrAyDSRin** *runs in $O(N|\Sigma|)$ time, where N is the the product of string length and maximum list length of the cover array C.*

Corollary 9. *Algorithm* **CrAyDSRin** *runs in linear time on average.*

Itn	i	1	2	3	4	5	6	7	8	9	$k[i]$	Explanation
0	$X[i]$	a									k[1]=1	
1	$X[i]$	a	a								k[2]=1	$\pi_2 = \{1\}$
												$A'_2 =\phi,\ A_2 = \{a\}$
2	$X[i]$	a	a	a							k[3]=1	$\pi_3 = \{1,2\}$
												$A'_3 =\phi,\ A_3 = \{a\}$
3	$X[i]$	a	a	a	a						k[4]=1	$\pi_4 = \{1,2,3\}$
												$A'_4 =\phi,\ A_4 = \{a\}$
4	$X[i]$	a	a	a	a	a					k[5]=1	$\pi_5 = \{1,2,3,4\}$
												$A'_5 =\phi,\ A_5 = \{a\}$
5	$X[i]$	a	a	a	a	a	b				k[6]=2	$\pi_6 = \{1,2,3,4,5\},\ \ A'_6 =\{a\}$
			b		b							for $c^1[6] = 4$ place new symbol 'b' in position 4 and 6
												for $c^2[6]\ = 2,\ c^2[6] \in c[c^1[6]]$
												$A_6 = \psi_6 = \{b\}$
6	$X[i]$	a	a	a	a	a	b	a			k[7]=2	$\pi_7 = \{1,2,3,4,5,6\},\ \ A'_7 =\phi$
			b		b			b				$A_7 = \{a\},\ \Sigma_7 - A_7 - A'_7 = \{b\}$
7	$X[i]$	a	a	a	a	a	b	a	c		k[8]=3	$\pi_8 = \{1,2,3,4,5,6,7\},\ \ A'_8 =\{b\},$
			b		b		c	b	a			for $c^1[8] = 6$ place new symbol 'c' in position 6 and 8
					c							for $c^2[8] = 2,\ c^2[8] \in c[c^1[8]]$
												$A_8 = \psi_8 = \{c\},\ \Sigma_8 - A_8 - A'_8 = \{a\}$
8	$X[i]$	a	a	a	a	a	b	a	c	c	k[9]=3	$\pi_9 = \{1,2,3,4,5,6,7,8\},\ \ A'_9 =\{a,b\}$
			b		b		c	b	a			$A_9 = \{a,b,c\} - \{a,b\}\ = \{c\}$
					c							

Table 8. An example run of algorithm CrAyDSin

Table 8 shows an example run of **CrAyDSRin** *Algorithm.*

5 Conclusion

In this paper, we have presented efficient algorithms for inferring a degenerate string given a valid cover array. We have presented two algorithms both of which returns a degenerate string from a given cover array, if the cover array is valid. Our first algorithm infers a degenerate string on an unbounded alphabet satisfying the cover array and our second algorithm infers a degenerate string on a least size. Future research may be carried out for devising similar reconstruction algorithms for degenerate strings considering other data structures (e.g., seed array).

References

1. P. Antoniou, M. Crochemore, C. S. Iliopoulos, I. Jayasekera, and G. M. Landau: *Conservative string covering of indeterminate strings.* Proceedings of the Prague Stringology Conference 2008, 2008, pp. 108–115.
2. P. Antoniou, C. S. Iliopoulos, I. Jayasekera, and W. Rytter: *Computing epetitive structures in indeterminate strings.* Proceedings of the 3rd IAPR International Conference on Pattern Recognition in Bioinformatics (PRIB 2008), 2008, pp. 108–115.
3. A. Apostolico and A. Ehrenfeucht: *Efficient detection of quasiperiodicities in strings.* tcs, 119(2) 1993, pp. 247–265.
4. M. F. Bari, M. S. Rahman, and R. Shahriyar: *Finding all covers of an indeterminate string in $O(n)$ time on average*, in Proceedings of the Prague Stringology Conference 2009, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2009, pp. 263–271.
5. M. Crochemore, C. S. Iliopoulos, S. P. Pissis, and G. Tischler: *Cover array string reconstruction*, in CPM'10, 2010, pp. 251–259.
6. J.-P. Duval, T. Lecroq, and A. Lefebvre: *Border array on bounded alphabet.* Journal of Automata, Languages and Combinatorics, 2005, pp. 51–60.
7. F. Franek, S. Gao, W. Lu, P. J. Ryan, W. F. Smyth, Y. Sun, and L. Yang: *Verifying a border array in linear time.* J. Comb. Math. Comb. Comput, 42 2000, pp. 223–236.
8. J. Holub and W. F. Smyth: *Algorithms on indeterminate strings.* Miller, M., Park, K. (eds.): Proceedings of the 14th Australasian Workshop on Combinatorial Algorithms AWOCA'03, 2003, pp. 36–45.
9. J. Holub, W. F. Smyth, and S. Wang: *Fast pattern-matching on indeterminate strings.* Journal of Discrete Algorithms, 6(1) 2008, pp. 37–50.
10. C. S. Iliopoulos, M. Mohamed, L. Mouchard, K. G. Perdikuri, W. F. Smyth, and A. K. Tsakalidis: *String regularities with don't cares.* Nordic Journal of Computing, 10(1) 2003, pp. 40–51.
11. C. S. Iliopoulos, L. Mouchard, and M. S. Rahman: *A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching.* Mathematics in Computer Science, 1(4) 2008, pp. 557–569.
12. T. M. Moosa, S. Nazeen, M. S. Rahman, and R. Reaz: *Linear time inference of strings from cover arrays using a binary alphabet – (extended abstract)*, in WALCOM'12, 2012, pp. 160–172.
13. S. Nazeen, M. S. Rahman, and R. Reaz: *Indeterminate string inference algorithms.* J. Discrete Algorithms, 2012, pp. 23–34.
14. W. F. Smyth and S. Wang: *New perspectives on the prefix array*, in SPIRE, A. Amir, A. Turpin, and A. Moffat, eds., vol. 5280 of Lecture Notes in Computer Science, Springer, 2008, pp. 133–143.
15. W. F. Smyth and S. Wang: *An adaptive hybrid pattern-matching algorithm on indeterminate strings.* Int. J. Found. Comput. Sci., 2009, pp. 985–1004.

Author Index

Proceedings of the Prague Stringology Conference 2013
Edited by Jan Holub and Jan Žďárek
Published by: Prague Stringology Club
Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9, Praha 6, 160 00, Czech Republic.

ISBN 978-80-01-05330-0

URL: `http://www.stringology.org/`
E-mail: `psc@stringology.org` Phone: +420-2-2435-9811

Printed by: Česká technika – Nakladatelství ČVUT
Thákurova 550/1, Praha 6, 160 41, Czech Republic