Proceedings of the Prague Stringology Conference 2017

Edited by Jan Holub and Jan Žďárek

August 2017

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

ISBN 978-80-01-06193-0

Preface

The proceedings in your hands contains a collection of papers presented in the Prague Stringology Conference 2017 (PSC 2017) held on August 28–30, 2017 at the Czech Technical University in Prague, which organizes the event. The conference focused on stringology, i.e., a discipline concerned with algorithmic processing of strings and sequences, and related topics.

The submitted papers were reviewed by the program committee subject to originality and quality. The fourteen papers in this proceedings made the cut and were selected for regular presentation at the conference. In addition, this volume contains an abstract of the invited talk "Compressed Random-Access Memory and Dynamic Succinct Data Structures" by Simon Puglisi.

The Prague Stringology Conference has a long tradition. PSC 2017 is the twenty-first PSC conference. In the years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conferences (PSC's) in 2001–2006, 2008–2016 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 have been regularity published in special issues of journals the Kybernetika, the Nordic Journal of Computing, the Journal of Automata, Languages and Combinatorics, the International Journal of Foundations of Computer Science, and the Discrete Applied Mathematics.

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.

We would like to thank all those who had submitted papers for PSC 2017 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 2017. Last, but not least, our thanks go to the members of the organizing committee for ensuring such a smooth running of the conference.

In Prague, Czech Republic
on August 2017

Jan Holub and Shunsuke Inenaga

Conference Organisation

Program Committee

Amihood Amir	(Bar-Ilan University, Israel)
Gabriela Andrejková	(P. J. Šafárik University, Slovakia)
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, *Co-chair*	(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	(McMaster University, Canada)
Bruce W. Watson	(FASTAR Group/Stellenbosch University, South Africa)
Jan Žďárek	(Czech Technical University in Prague, Czech Republic)

Organising Committee

Miroslav Balík, *Co-chair*	Bořivoj Melichar	Jan Trávníček
Ondřej Guth,	Radomír Polách	Jan Žďárek
Jan Holub, *Co-chair*		

External Referees

Keisuke Goto	Dominik Köppl	Arnaud Lefebvre
Yannick Guesnet		Panagiotis Charalampopoulos

Table of Contents

Invited Talk

Contributed Talks

Online Recognition of Dictionary with One Gap *by Amihood Amir, Avivit Levy, Ely Porat, and B. Riva Shalom* 3

Range Queries Using Huffman Wavelet Trees *by Gilad Baruch, Shmuel T. Klein, and Dana Shapira* 18

Dynamic Succinct Data Structures and Compressed Random Access Memory*

(*Abstract*)

Simon J. Puglisi

Helsinki Institute for Information Technology,
Department of Computer Science, University of Helsinki,
P.O. Box 68, FI-00014, Finland
puglisi@cs.helsinki.fi

Keywords: succinct data structures, dynamic data structures, trie, hash table

In the past 20 years, succinct and compact data structures have matured to the point that practical implementations of them now underpin several data intensive software processes, e.g, for DNA sequence assembly and search in bioinformatics. The tacit assumption with the vast majority of results to date has been that the data structure and the underlying data remain static — practical dynamic compact data structures are still very much in their infancy. This talk will focus on some recent forays into the development of practical dynamic succinct data structures, including dynamic succinct tries, compact hash tables, and compressed arrays.

This is joint work with Andreas Poyias, Rajeev Raman, and Bella Zhukova.

* This work was supported by the Academy of Finland via grant 294143.

Simon J. Puglisi: Dynamic Succinct Data Structures and Compressed Random Access Memory, p. 1.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

Online Recognition of Dictionary with One Gap

Amihood Amir[1,2], Avivit Levy[3], Ely Porat[1], and B. Riva Shalom[3]

[1] Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel.
E-mail: {amir, porately}@cs.biu.ac.il
[2] Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218.
[3] Department of Software Engineering, Shenkar College, Ramat-Gan 52526, Israel.
E-mail: {avivitlevy, rivash}@shenkar.ac.il

Abstract. We formalize and examine the online Dictionary Recognition with One Gap problem (DROG) which is the following. Preprocess a dictionary D of d patterns, where each pattern contains a special *gap* symbol that can match any string, so that given a text that arrives online, a character at a time, we can report all the patterns from D that have not been reported yet and are suffixes of the text that has arrived so far, before the next character arrives. The gap symbols are associated with *bounds* determining the possible lengths of matching strings. Online DROG captures the difficulty in a bottleneck procedure for cyber-security, as many digital signatures of viruses manifest themselves as patterns with a single gap.
Following the work of [4] on the closely related online Dictionary Matching with One Gap problem (DMOG), we provide algorithms whose time cost depends linearly on $\delta(G_D)$, where G_D is a bipartite graph that captures the structure of D and $\delta(G_D)$ is the *degeneracy* of this graph. These algorithms are of practical interest since although $\delta(G_D)$ can be as large as $\sqrt{d}$, and even larger if G_D is a multi-graph, it is typically a very small constant in practice. Finally, when $\delta(G_D)$ is large we describe other efficient solutions.

1 Introduction

Cyber-security is a critical modern challenge. Network intrusion detection systems (NIDS) perform protocol analysis, content searching, recognizing and matching, in order to detect harmful software. Such malware may appear non-contiguously, scattered across several packets, which necessitates matching *gapped* patterns.

A *gapped pattern* P is one of the form $P_1\ \{\alpha, \beta\}\ P_2$, where each subpattern P_1, P_2 is a string over alphabet Σ, and $\{\alpha, \beta\}$ matches any substring of length at least α and at most β, which are called the *gap bounds*. Gapped patterns may contain more than one gap, however, those considered in NIDS systems typically have at most *one* gap, and are a serious bottleneck in such applications [22,4]. Therefore, an efficient solution for this case is of special interest.

Though the gapped pattern matching problem arose over 20 years ago in computational biology applications [20,14] and has been revisited many times in the intervening years (e.g. [19,8,17,7,12,21,23]), network intrusion detection systems applications necessitate a different generalization of the problem. These applications motivate the *dictionary matching with one gap* (DMOG) problem defined by [4], which is a variant of the well-studied dictionary matching problem (see, e.g. [1,2,9,3,11]). The dictionary, which is the set of d gapped patterns to be detected, could be quite large.

The DMOG problem was, therefore, studied [6,15,4] for both the offline and the online settings. Lower bounds on the complexity of this problem as well as (almost) matching upper bounds were described in [4]. These lower bounds expose a hidden parameter of input dictionary that sheds light on the reason why this problem has

Amihood Amir, Avivit Levy, Ely Porat, B. Riva Shalom: Online Recognition of Dictionary with One Gap, pp. 3–17.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

resisted many researcher's attempts at finding a definitive efficient solution on the one hand, while on the other hand, enables describing the solutions in terms of this parameter. We elaborate on this issue in Section 2.

The definition of the DMOG problem requires reporting all occurrences of the dictionary patterns. This is a necessary requirement in order to remove all viruses from a given source. However, the size of the input may be quite large if dictionary patterns occur many times in the source. The process of malware detection is required to be very fast, and in many cases we would prefer a faster scan in order to determine whether the source stream is infected by viruses or not. We would also like to know which viruses attacked the source in case it is affected, so that an appropriate (slower) exhaustive infection recovery procedure can be applied on the source. Motivated by this need of NIDS applications, we focus in this paper on the recognition of the set of viruses that exists in the source, and formally define the *Dictionary Recognition with One Gap problem (DROG)* as follows:

Definition 1. The Dictionary Recognition with One Gap problem (DROG) *is:*
Input: *A text T of length $|T|$ over alphabet Σ, and a dictionary D of d gapped patterns $P_1, \ldots, P_d$ over alphabet Σ, where each pattern has at most one gap.*
Output: The maximal subset $S \subseteq D$, where pattern $P_i \in S$ appears at least once in T.

We study the more practical *online* DROG problem. The dictionary D can be preprocessed in advance, resulting in a data structure. Given this data structure the text T is presented one character at a time, and when a character arrives only the subset of patterns with a match ending at this character *that were not previously reported* should be reported before the next character arrives. Two cost measures are of interest: a preprocessing time and a time per character.

	Preprocess Time	Total Query Time	Algorithm Type	Remark
[16]	none	$\tilde{O}(\|T\| + \|D\|)$	online	reports only first occurrence
[23]	$O(\|D\|)$	$\tilde{O}(\|T\| + d)$	online	reports only first occurrence
[13]	$O(\|D\|)$	$O(\|T\| \cdot lsc + socc)$	online	reports one occurrence per pattern and location
[5]	$\tilde{O}(\|D\|)$	$\tilde{O}(\|T\|(\beta - \alpha) + op)$	offline	DMOG
[15]	$O(\|D\|)$	$\tilde{O}(\|T\|(\beta^* - \alpha^*) + op)$	offline	DMOG
[4]	$O(\|D\|)$	$\tilde{O}(\|T\| \cdot \delta(G_D) \cdot lsc + op)$	online	DMOG
[4]	$O(\|D\|)$ $O(\|D\|)$	$\Omega(\|T\| \cdot \delta(G_D)^{1-o(1)} + op)$ $\Omega(\|T\| \cdot (\beta - \alpha)^{1-o(1)} + op)$	online or offline	DMOG
This paper	$O(\|D\|)$	$\tilde{O}(\|T\| \cdot \delta(G_D) \cdot lsc + d)$	online	DROG

Table 1. Comparison of previous work and some new results. The parameters: lsc is the longest suffix chain of subpatterns in D, $socc$ is the number of subpatterns occurrences in T, op is the number of pattern occurrences in T, α^* and β^* are the minimum left and maximum right gap borders in the non-uniformly bounded case, $\delta(G_D)$ is the degeneracy of the graph G_D representing dictionary D.

Previous Work. Finding efficient solutions for the problem has proven to be a difficult algorithmic challenge as little progress has been obtained even though many researchers in the pattern matching community and the industry have tackled it. Table 1 describes a summary and comparison of previous work. It illustrates that previous formalizations of the problem until that of [4], either do not enable detection of all intrusions or are incapable of detecting them in an online setting, and therefore, are inadequate for NIDS applications. Table 1 also demonstrates that the upper bounds of [4] for the DMOG problem are essentially optimal (assuming some popular conjectures). Most importantly, Table 1 demonstrates that no previous work has been done on the DROG problem as formalized in this paper.

Our Results. Our goal in this paper is that the time per character cost would be independent of the number of occurrences of dictionary patterns in the text. This is a nontrivial requirement as we can no longer afford costly operations that were accounted for the size of the output for the detection of dictionary patterns at query time in the online DMOG solutions of [4]. In our case such costly operations can be afforded for newly detected patterns only. This raises the difficulty of limiting the detection process to the dynamically changing set of yet undetected dictionary patterns.

Paper Organization. In Section 2 we give a brief review of the solutions to the online Dictionary Matching with One Gap (DMOG) problem suggested by [4]. Section 3 describes our solution for the online Dictionary Recognition with One Gap (DROG) problem, which is based on the solutions described in Section 2 for the DMOG problem with changes and adoptions in order to fulfill the requirement of the DROG problem. Section 4 concludes the paper and poses some open problems.

2 An Overview of the DMOG Solutions

In this section we give a brief description of the DMOG solutions of [4]. The reader who is familiar with their ideas and techniques can skip this section.

The Bipartite Graph G_D**.** The first baseline idea of their solutions is to represent the dictionary as a graph $G_D = (V, E)$, where the subpatterns are the vertices, and there is an edge $(u, v) \in E$ if and only if there is a pattern $P \in D$, where P^1 is associated with node u and P^2 is associated with v. Moreover, the graph $G_D = (V, E)$ is converted to a bipartite graph by creating two copies of V called L (the left vertices) and R (the right vertices) in the following way. For every edge $(u, v) \in E$, an edge is added to the bipartite graph from $u_L \in L$ to $v_R \in R$, where u_L is a copy of u and v_R is a copy of v.

Graph Orientations. The next baseline idea is to preprocess G_D using linear time greedy algorithm suggested by Chiba and Nishizeki [10] to obtain a $\delta(G_D)$-orientation of the graph G_D, where an orientation of an undirected graph $G = (V, E)$ is called a *c-orientation* if every vertex has out-degree at most $c \geq 1$. The orientation is viewed as assigning "responsibility" for all data transfers occurring on an edge to one of its endpoints, depending on the direction of the edge in the orientation (regardless of the actual direction of the edge in the input graph G_D). The notation of an edge $e = (u, v)$ is as oriented from u to v, while e could be directed either from u to v or from v to u. The vertex u is called a *responsible-neighbour* of v and v an *assigned-neighbour*

of u. The notion of graph *degeneracy* $\delta(G_D)$ is defined as follows. The degeneracy of an undirected graph $G = (V, E)$ is $\delta(G) = \max_{U \subseteq V} \min_{u \in U} d_{G_U}(u)$, where d_{G_U} is the degree of u in the subgraph of G induced by U. In words, the degeneracy of G is the largest minimum degree of any subgraph of G. A non-multi graph G with m edges has $\delta(G) = O(\sqrt{m})$, and a clique has $\delta(G) = \Theta(\sqrt{m})$. The degeneracy of a multi-graph can be much higher.

Subpatterns Detection Mechanism. An Aho-Corasick (AC) Automaton [1] is used for determining when a subpattern arrives using a standard binary encoding technique, so that each character arrival costs $O(\log |\Sigma|)$ worst-case time for recognizing the arrival of a dictionary subpattern. For simplicity of exposition, $|\Sigma|$ is assumed to be constant. Since each arriving character may correspond to the arrival of several subpatterns when a subpattern is a proper suffix of another, the complexities are phrased in terms of lsc, which is the maximum number of vertices in the graph that arrive due to a character arrival. The lsc factor was used even in solutions for simplified relaxations of the DMOG problem [13]. Another issue is that, since subpatterns may be long, a delay must be accommodated in the time a vertex corresponding to a second subpattern is treated as if it has arrived, thus inducing a minor additive space usage.

Two variants of the gapped dictionary are considered having either uniformly bounded gap borders or non-uniformly bounded gap borders. In the former case, all gapped patterns of the dictionary have the same gaps borders $\{\alpha, \beta\}$, whereas in the latter, every pattern P_i has its own gap borders $\{\alpha_i, \beta_i\}$. Two sets of solutions are described: for sparse graphs, where $\delta(G_D) = o(\sqrt{d})$, and for dense graphs. The solutions for these four cases are described hereafter.

2.1 DMOG for Sparse Graphs

Uniformly Bounded Gaps. The data structures used in this case are:

1. For each vertex $v \in R$, a list $\mathcal{L}_v$ maintaining all responsible-neighbours of v, $u \in L$, that arrived at least α and at most β time units ago.
2. For each vertex $u \in L$, an ordered list of time stamps τ_u of the times u arrived within the appropriate gap to the current time unit (text index).
3. The list $\mathcal{L}_\beta$ of delayed vertices $u \in L$ for at least α time units before they are considered.

The $\mathcal{L}_v$ lists are updated by deleting u nodes that arrived more than β time units ago and inserting u nodes that just arrived α time units ago and do not appear already in the data structure. Therefore, when an appearance of node v is detected, all the patterns $u\{\alpha, \beta\}v$ for $u \in \mathcal{L}_v$ are reported according to the time stamps in τ_u, as the output includes all appearances of the gapped patterns. In addition, the edges for which v is their responsible-neighbour are scanned, and those for which the assigned-neighbour u is marked as arrived, are reported.

The removal of $u \in L$ from $\mathcal{L}_v$ must be delayed by at least $m_v - 1$ time units, where m_v is the length of the substring represented by v. If u is removed from $\mathcal{L}_v$ after a delay of $m_v - 1$, then we may be forced to remove a large number of such vertices at a given time. Therefore, the removal of u is delayed by $M - 1$ time units, where M is the length of the longest subpattern that corresponds to a vertex in R.

Time and Space Complexity: [4] show that using the above data structures, the $DMOG$ problem with uniformly bounded gap borders can be solved in $O(|D|)$ preprocessing time, $O(\delta(G_D)\cdot lsc+op)$ time per text character, where op is the number of patterns that are reported due to the character arriving, and $O(|D|+lsc\cdot(\beta-\alpha+M)+\alpha)$ space.

Non-Uniformly Bounded Gaps. In the case of non-uniformly bounded gaps, each edge $e=(u,v)$ has its own boundaries $\{\alpha_e,\beta_e\}$, yielding a multi-graph. Let α^* and β^* be the minimum left and maximum right gap borders in the non-uniformly bounded dictionary. A framework similar to the previous subsection is used, yet, instead of the list $\mathcal{L}_v$, a fully dynamic data structure S_v supporting 4-sided 2-dimensional orthogonal range reporting queries, is used for saving the occurrences of responsible neighbour of v.

For each responsible-neighbour $u\in L$ of v, that arrived in the active window in time t, where $e=(u,v)$, the point $(t+\alpha_e+1, t+\beta_e+1)$ is inserted into S_v, yielding the information saved is the intervals in which an occurrence of v implies an occurrence of a gapped pattern. When a vertex $v\in R$ arrives at time t, a range query $[0,t]\times[t,\infty]$ over S_v returns the points that have (x,y) coordinates in the given range, thus a pattern appearance.

To implement S_v, a Mortensen's data structure [18] is used. It supports the set of $|S_v|$ points from $\mathbb{R}^2$ with $O(|S_v|\log^{7/8+\epsilon}|S_v|)$ words of space, insertion and deletion time of $O(\log^{7/8+\epsilon}|S_v|)$ and $O(\frac{\log|S_v|}{\log\log|S_v|}+op)$ time for range reporting queries on S_v, where op is the size of the output.

Time and Space Complexity: [4] show that using the above, the $DMOG$ problem with non-uniformly bounded gap borders on a graph G with m edges (gapped patterns) and n vertices can be solved with $O(|D|)$ preprocessing time, $\tilde{O}(\delta(G)+op)$ time per query vertex, where op is the number of edges reported due to the vertex arriving, and $\tilde{O}(m+\delta(G)(\beta^*-\alpha^*)+\alpha^*)$ space.

2.2 DMOG for Dense Graphs

In the case of dense graphs where $\delta(G_D)=\Omega(\sqrt{d})$, the solutions described above require $O(lsc\cdot\sqrt{d})$ time. For such cases a different method for orienting the graph is suggested by [4], referred to as a *threshold* orientation, where a vertex in G_D is defined as *heavy* if it has more than $\sqrt{d/lsc}$ neighbours, and *light* otherwise. Hence, the number of heavy vertices is less than $\sqrt{lsc\cdot d}$. An edge where at least one of its endpoints is light is oriented to leave the light vertex. For such edges the algorithms from the previous subsection are applied in $\tilde{O}(lsc+\sqrt{lsc\cdot d}+op)$ time complexity.

Reporting edges between two heavy vertices is done differently. Although the number of vertices from L that arrive at the same time can be as large as lsc and the number of neighbours of each such vertex can be very large, the number of heavy vertices in R is still less than $\sqrt{lsc\cdot d}$. So [4] use a batched scan on all vertices of R to keep the time cost low. The vertices from L are ordered in a tree T according to suffix relations between the subpatterns associated with the vertices, where a vertex u is an ancestor of a vertex u' if and only if the subpattern associated with u is a suffix of the subpattern associated with u'.

Uniformly Bounded Gaps. Let $R = \{v_1, v_2, \ldots\}$, where $|R| = O(\sqrt{lsc \cdot d})$, since we only deal with heavy vertices.

For this case, the $\mathcal{L}_v, \mathcal{L}_\beta, \tau_u$ data structures are used as well as the framework of the solution to $DMOG$ for bounded gaps in sparse graphs, as described in Subsection 2.1. In addition, in order to add vertices that are suffix of each other to $\mathcal{L}_v$ in a single operation, the following data structures are also used:

1. For each vertices $u \in L$ and $v_i \in R$ such that $e = (u, v_i) \in E$, **a pointer** $next(e)$ is set to an edge $e' = (u', v_i)$ where u' is the lowest proper ancestor of u in T such that there is an edge from u' to v_i. If no such vertex u' exists then $next(e) = null$. All these pointers can be added in linear time, and their space usage is linear.
2. For each vertex $u \in L$, **an array** $A_u[]$ of size $|R|$ is built, where $A_u[i]$ is a pointer to a list of edges connecting all ancestors of u in T(which may be u) to v_i. If $e = (u, v_i) \in E$, then $A_u[i]$ points to $e = (u, v_i)$, and the list of all ancestors of u in T that have edges touching v_i is obtained through the $next(\cdot)$ pointers. Similarly, if there is no edge (u, v_i) then the entry of $A_u[i]$ points to the edge (u', v_i) where u' is the lowest proper ancestor of u in T such that there is an edge from u' to v_i. If no such edge exists then $A_u[i] = null$.

The $A_u[]$s arrays are constructed by filling $A_u[i]$ while consulting $A_{u'}[]$, where u' is a proper ancestor of u and $A_{u'}[]$ was already filled. In order to reduce space usage of the A_u arrays, the A_u arrays are constructed during preprocessing time only for specially chosen $O(\sqrt{\frac{d}{lsc}})$ vertices so that the time cost for constructing the rest of the A_u arrays online is $O(\sqrt{lsc \cdot d})$.

Time and Space Complexity: [4] show that the DMOG problem with one gap and uniform gap borders can be solved with $O(|D|)$ preprocessing time, $O(lsc + \sqrt{lsc \cdot d} + op)$ time per text character, and $O(|D| + lsc(\beta - \alpha + M) + \alpha)$ space.

Non-Uniformly Bounded Gaps. Recall that for the non-uniform gaps, the gap boundaries of an edge e are denoted by α_e and β_e. For the case of dense graphs and unbounded gaps [4] used different data structures:

1. For each $e = (u, v_i) \in E$, **an array** $next_e$ of size $\beta_e - \alpha_e + 1$ is maintained, where for $\alpha_e \leq j \leq \beta_e$, $next_e[j]$ points to an edge $e' = (u', v_i)$ such that u' is the lowest ancestor of u in T (possibly u itself) such that there is an edge $e' = (u', v_i)$ where $\alpha_{e'} \leq j \leq \beta_{e'}$ and the pointers $next_e[j]$ do not form a loop. If no such edge exists then $next_e[j] = null$. (For simplicity, the indices of the array are treated as starting from α_e and ending at β_e)
2. For each pair of vertices $u \in L$ and $v_i \in R$, **an array** $W_{u,i}$ of size $\beta^* - \alpha^* + 1$ is maintained. If $e = (u, v_i) \in E$, then $W_{u,i}[j]$ is a pointer to e if its gap boundaries include j, and to $next_e[j]$ otherwise.The remaining entries of $W_{u,i}[j]$ are null.
3. For each vertex $v_i \in R$, **a cyclic *active window* array** AW_i of size $\beta^* - \alpha^* + M + 1$ is maintained, where $AW_i[j]$ is a pointer to a list of lists of edges that all need to be reported if v_i appears in $j - 1$ time units from now.

The total space usage for the $next_e$ pointer arrays is $\rho := \sum_{e \in E}(\beta_e - \alpha_e + 1) \leq d(\beta^* - \alpha^*)$, and they can be constructed in $O(\rho)$ time. Yet, if all the arrays $W_{u,i}$ are computed in the preprocessing, the time and space would be $O(lsc \cdot d(\beta^* - \alpha^*))$. Therefore, [4] reduce this cost by postponing the construction of a carefully chosen part of them to the query time.

Time and Space Complexity: [4] show that the DMOG problem with non-uniform gap borders can be solved with $O(|D| + d(\beta^* - \alpha^*))$ preprocessing time, $\tilde{O}(lsc + \sqrt{lsc \cdot d}(\beta^* - \alpha^* + M) + op)$ time per query text character, and $\tilde{O}(|D| + d(\beta^* - \alpha^*) + \sqrt{lsc \cdot d}(\beta^* - \alpha^* + M) + \alpha^*)$ space.

3 Solving Online Dictionary Recognition with One Gap

Our solution follows the framework of [4] showing that it is possible to make changes in their algorithms in order to solve the DROG problem. Recall that the definition of the DROG problem requires reporting only a single appearance of each gapped pattern in the dictionary, where each gapped pattern is represented as an edge in the bipartite graph G_D. Our basic idea is, therefore, quite simple: in order to avoid repetitious reports of an edge (u, v), after the first time an edge is reported we delete it from the graph, thus assuring that u will not be inserted to the data structures maintaining v's responsible neighbours again. In order to avoid considering vertices whose associated edges are all reported, we add two counters to each vertex v, *lcount* and *rcount*, which are initialized with the number of responsible neighbours of v and with the number of neighbours that v is responsible for, respectively. Each report and deletion of an edge (u, v) implies a decrease in $lcount(u)$ and in $rcount(v)$.

The remaining task we should carefully take care of is to assure that for every edge (u, v), u appears only once in the data structure of v, so that reporting the edge will be unique even if v occurs again. This task is nontrivial since in some cases it is not possible to save a single copy of u in the data structure of v (otherwise, we might miss an occurrence of a dictionary gapped pattern), therefore, the deletion of the edge (u, v) requires a deletion of additional appearances of u from the data structure associated with v. This should be done without causing an unbearable overhead in the time complexity.

In the following subsections we consider the four solutions described in Section 2, giving for each of them a tailored adaptation for the online *Dictionary Recognition with One Gap* problem.

3.1 DROG for Sparse Graphs

Uniformly Bounded Gaps. Following the framework described in Subsection 2.1, We use the $\mathcal{L}_v$ lists to maintain at most a single appearance of the responsible neighbours of v. Hence, when going over the list and reporting edges in case v occurred, each edge is reported once without scanning the τ_u list of u appearance times. Therefore, the solution to the DROG problem for uniformly bounded gap borders is identical to the solution for DMOG with the additional task of deleting an edge after its first report, updating the relevant *rcount* and *lcount* and considering only vertices whose *rcount* and *lcount* values are non zero.

This immediately gives Theorem 2.

Theorem 2. *The online DROG problem with uniformly bounded gap borders on a graph G_D with m edges and n vertices can be solved in $O(m + n)$ preprocessing time, $O(lsc \cdot \delta(G_D) + op^*)$ time per query vertex, where op^* is the number of new distinct dictionary patterns reported due to a character arrival, and $O(m + \beta)$ space.*

Non Uniformly Bounded Gaps. In this case, for every vertex $v \in R$, the data structure S_v supporting 4-sided 2-dimensional orthogonal range reporting queries saves the occurrences of responsible neighbours of v, as in Subsection 2.1. Now, the deletion of a reported edge from G_D is not sufficient in order to assure a unique report of edge occurrence, since the same vertex u can be represented by several points in a certain S_v data structure due to several arrival times. If several points are within the query bounds, the range query will return all these occurrences of the edge (u, v). We need to avoid such redundant reports.

In order to avoid an increase in the time complexity, we modify the algorithm as follows. When a vertex u arrives at time t, each assigned-neighbour v such that $e = (u, v)$, the point $(t + \alpha_e + 1, t + \beta_e + 1)$ is inserted to the data structure S_v, representing a time interval in which an occurrence of v yields an occurrence of the edge e. Our modification is to unite every two points representing overlapping or adjacent intervals into a single point. This procedure is delicately performed, as the intervals may be separated again, when one of the occurrences of u represented by the interval becomes irrelevant – when located beyond the gaps bounds. An example of the union effect is depicted in Figure 1. In addition, all intervals associated with the same edge are linked, in order to enable deleting all of them, when the edge is reported due to one of the intervals.

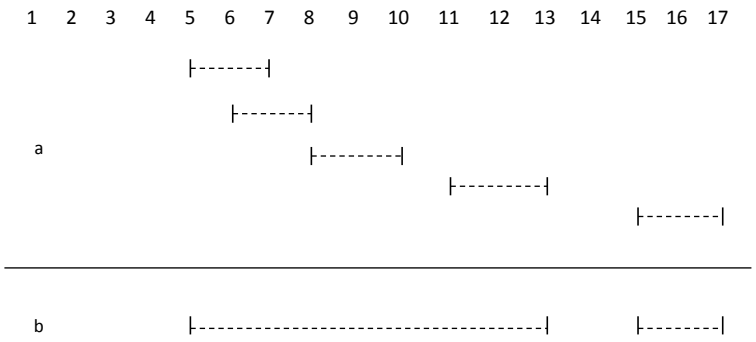


Figure 1. Consider $e = (bbb\{2-4\}a)$ and $\tau_u = \{2, 3, 5, 8, 12\}$. (a) depicts the intervals represented by points that are inserted to S_a by the DMOG algorithm. (b) depicts the intervals represented by points that are inserted to S_a by the DROG algorithm.

The modified algorithm is:
For an arrived vertex at query time t, do:

1. If the arrived vertex is $v \in R$, such that $rcount(v) \neq 0$,
   (a) A range query $[0, t] \times [t, \infty]$ is performed over S_v. Edges representing the range output are reported.
   (b) Edges for which v is their responsible-neighbour are scanned, and those for which the assigned-neighbour u is marked as arrived are reported according to a search in their time stamp.

(c) For every reported edge $e = (u, v)$,
  i. e is deleted from G_D,
  ii. Every point associated with e is deleted from S_v,
  iii. $lcount(u)$ and $rcount(v)$ are decreased by one.

2. If the arrived vertex is $u \in L$, such that $lcount(u) \neq 0$,
  (a) u is inserted to $\mathcal{L}_{\beta^*}$,
  (b) For each assigned-neighbour v such that $e = (u, v) \in E$,
    i. If τ_u is empty or $t + \alpha_e > t' + \beta_e + 1$, where t' is the newest time stamp in τ_u, then $(t + \alpha_e + 1, t + \beta_e + 1)$ is inserted to S_v.
    ii. Else, let (x, y) be the last point associated with e that was inserted to S_v, then delete (x, y) from S_v and insert $(x, t + \beta_e + 1)$ to S_v.
3. For vertices $u \in L$ arriving exactly $\alpha^* + 1$ time units before time t, such that $lcount(u) \neq 0$,
  (a) u is marked as arrived,
  (b) $t - \alpha^* - 1$ is added to τ_u.
4. For vertices $u \in L$ arriving exactly $\beta^* + M + 1$ time units before time t, such that $lcount(u) \neq 0$,
  (a) u is removed from $\mathcal{L}_{\beta^*}$,
  (b) The time stamp $t - \beta_* - M - 1$ is removed from τ_u,
  (c) For each assigned-neighbour v, such that $e = (u, v) \in E$,
    i. Let t' be the oldest time stamp in τ_u,
    ii. If τ_u is empty or $t-\beta^*-M+\beta_e < t'+\alpha_e$, then the point $(t-\beta^*+\alpha_e, t-\beta^*+\beta_e)$ is deleted from S_v,
    iii. Else, let (x', y') be the first point associated with e in S_v, delete (x', y') from S_v and insert $(t' + \alpha_e + 1, y')$ to S_v.

Theorem 3 follows.

Theorem 3. *The online DROG problem with non-uniformly bounded gap borders can be solved in $O(|D|)$ preprocessing time, $\tilde{O}(\delta(G_D) \cdot lsc + op^*)$ time per text character, where op^* is the number of new distinct dictionary patterns reported due to character arrival, and $\tilde{O}(|D| + lsc \cdot \delta(G_D)(\beta^* - \alpha^* + M) + \alpha^*)$ space.*

Proof. Correctness: The strategy of inserting a point $(x, y) = (t + \alpha_e + 1, t + \beta_e + 1)$ to S_v when u appears at time t and $e = (u, v)$, implies that when v appears at time t^*, the range query $[0, t^*][t^*, \infty]$ is performed giving all points (x, y) where $x \leq t^*$ and $t^* \leq y$. Thus, all overlapping intervals associated with edge e including point t^* are reported. Our method of uniting all these intervals to a single interval ensures a single report due to an occurrence of e and the occurrence of v at time t^*. Note that only points representing intervals related to the same edge are united to the same point, and therefore, to the same gap since several possible gaps between some vertices u and v define distinct edges. Hence, it is only necessary to consider the start point of a new interval with the endpoint of the last interval included in S_v.

Uniting overlapping intervals is sufficient for a unique report of an edge e, yet if there exist several intervals representing e in S_v due to later occurrences of u, they are deleted when e is reported, in order to avoid additional redundant report of e in future arrivals of v.

Time complexity: As the framework of [4] is used, the time complexity is in accordance with a decrease due to the fact that our total output size is limited to d. The

modifications require only extra $O(1)$ operations in all cases except for the necessity to delete all points representing intervals associated with e from S_v after reporting e. However, as deletion from S_v requires the same time as the insertion to it, we account a deletion of a point by its insertion that was already accounted in the time complexity of the algorithm.

In addition, uniting points representing overlapping intervals and deleting points associated with a reported edges may decrease the time complexity of each insertion, deletion and range query, as their time complexity depends on $|S_v|$ in [18], which we use. Moreover, the range query requires $O(\frac{\log |S_v|}{\log \log |S_v|} + op)$ time, where op is the size of the output of the query, so by reducing the total output size to be at most d in all queries, we reduce the time complexity.

The space complexity is identical to that of the DMOG solution. □

3.2 DROG for Dense Graphs

Uniformly Bounded Gaps. We follow the framework of Subsection 2.2 of constructing the A_u arrays, and inserting A_u to $\mathcal{L}_{v_i}$ when u arrives, where $e = (u, v_i) \in E$. After reporting an edge, it is deleted from G_D, yet, as in the previous subsection, this deletion is not sufficient for preventing repetitious report of the edge e, since additional occurrences of v_i or of u'', where the subpattern associated with u is a suffix of the subpattern associated with u'', can yield additional reports of e. Nevertheless, by analyzing carefully the problematic scenarios this difficulty can be overcome. The modified algorithm follows.

When a vertex arrives at query time t:

1. If the arrived vertex is $v_i \in R$, such that $rcount(v_i) \neq 0$,
   (a) For every $A_u[i] \in \mathcal{L}_{v_i}$ where $A_u[i]$ contains a pointer to edge e,
      i. if e appeared in the appropriate time frame according the τ_u list (since the tail of $\mathcal{L}_{v_i}$ should be skipped), then **while** $e \neq null$
         A. Edge e is reported once,
         B. $e = next(e)$
      ii. The edges for which v_i is their responsible-neighbour are scanned, and those for which the assigned-neighbour u is marked as arrived are reported.
   (b) For every reported edge $e = (u, v_i)$
      i. e is deleted from E_D,
      ii. $A_u[i] = null$,
      iii. $A_u[i]$ is deleted from $\mathcal{L}_{v_i}$,
      iv. $lcount(u)$ and $rcount(v_i)$ are decreased by one.
2. If the arrived vertex is $u \in L$, such that $lcount(u) \neq 0$, then u is inserted into $\mathcal{L}_\beta$.
3. For vertices $u \in L$ arriving exactly $\alpha + 1$ time units before time t, such that $lcount(u) \neq 0$,
   (a) For each v_i, where $A_u[i] \neq null$,
      i. $A_u[i]$ is added to the beginning of $\mathcal{L}_{v_i}$,
      ii. if it was already in the reporting list $\mathcal{L}_{v_i}$ then the older copy is removed from $\mathcal{L}_{v_i}$.
   (b) u is marked as arrived,
   (c) $t - \alpha - 1$ is added to τ_u.
4. For vertices $u \in L$ arriving exactly $\beta + M$ time units before time t, such that $lcount(u) \neq null$,

(a) u is removed from $\mathcal{L}_\beta$,
(b) The time stamp $t - \beta - M$ is removed from τ_u,
(c) For each v_i, where $A_u[i] \neq null$, $A_u[i]$ is deleted from $\mathcal{L}_{v_i}$.

This gives Theorem 4.

Theorem 4. *The online DROG problem with uniform gap borders can be solved in $O(|D|)$ preprocessing time, $O(lsc+\sqrt{lsc \cdot d}+op^*)$ time per text character, where op^* is the number of new distinct dictionary patterns reported due to the character arriving, and $O(|D| + lsc(\beta - \alpha + M) + \alpha)$ space.*

Proof. Correctness: According to the algorithm each $A_u[i]$ is uniquely added to $\mathcal{L}_{v_i}$, thus, in case edge $e = (u, v_i)$ is detected, it is reported once from the current $\mathcal{L}_{v_i}$. The deletion of $A_u[i]$ from $\mathcal{L}_{v_i}$ after reporting $e = (u, v_i)$ prevents another report of e upon additional occurrences of v_i. By adding the requirement of overriding $A_u[i]$ by $null$, it is guaranteed that no additional report of e, either by another occurrence of u or by an occurrence of u'', where the subpattern associated by u is a suffix of the subpattern associated by u''. In the latter case, if the subpattern associated by u'' occurred after the subpattern associated by u in the text, $\mathcal{L}_{v_i}$ contains $A_{u''}[i]$, thus, $e = (u, v_i)$ is included in the list derived from $A_{u''}[i]$. When v_i occurs within a proper gap from the occurrence of u'', all the edges in the list derived from $A_{u''}[i]$ are reported. Nevertheless, our modified algorithm requires that before reporting edge e, the array $A_u[i]$ associated with e is checked to be non null, if $A_u[i] = null$ the report of the edges derived from the occurrence of u'' is terminated, as the null implies that all edges associated with u as well as its suffixes and v_i were already reported if they existed. In case u'' occurred before u, the edge $e = (u, v_i)$ is reported by the list derived from $A_{u''}[i]$, yet when e is reported the algorithm deletes $A_u[i]$ from $\mathcal{L}_{v_i}$, so the occurrence of u is not checked again with regard to v_i. Hence, a single occurrence of (u, v_i) is reported.

Time Complexity: The main modifications of the original DMOG algorithm for uniformly bounded gaps are the check whether $A_u[i] \neq null$ before reporting the edge $e = (u, v_i)$, and the procedure of deleting an edge after its report. We show that both these additions to the algorithm do not increase its time complexity. First, note that every check of $A_u[i]$ can be attributed to a reported edge. Either $A_u[i]$ is found to be non $null$, due to locating u in the text and reporting (u, v_i), or $A_u[i]$ is found to be $null$ due to a report of a $e'' = (u'', v_i)$, where u is a suffix of u''. Even in the latter case, the check operation can be accounted for by the report of the (u'', v_i), as a termination of a loop reporting the edges derived from the fact that some $A_u[i]$ already occurred in the report process, thus the loop reaches an already reported edge implying that all the following edges in the list were already reported. Second, a reported edge indeed induces the deletion operations required to preserve the uniqueness of the edges reported. However, the number of such operations is constant and can be accounted for the reported occurrence of an edge. Also note, that updating $A_u[i] = null$ after reporting the associated edge does not change the efficiency of the A_u arrays construction, as it implies that all edges associated with u and its suffixes and v_i were already reported, so the list starting in a predecessor of u comes to an end, as no further suffixes should be considered with node v_i. The rest of the time complexity analysis is the same as that of [4] except for the *op* factor which is replaced by a total of at most d reports due to a single report of every edge.

The space complexity is unchanged. □

Non-Uniformly Bounded Gaps. Following the same basic idea, after reporting an edge $e = (u, v_i)$ it is deleted from G_D and $W_{u,i}[j]$ should be assigned *null* for $\alpha_e \leq j \leq \beta_e$, so that additional occurrence of u are not inserted to the active window of v_i, $AW_{v_i}[]$. In addition, other occurrences of u already in $AW_{v_i}[]$ may derive a repetitious report of e due to an additional occurrence of v_i. In the case of uniformly bounded gaps, we handled the problem by deleting all appearances of $e = (u, v_i)$ from the data structure $\mathcal{L}_{v_i}$ of located neighbours of v_i. However, dealing with non-uniformly bounded gaps is more complicated, since when edge (u, v_i) is reported, some edges $e' = (u', v_i)$, where u' represents a subpattern that is a suffix of the subpattern represented by u, are reported too while other such e's are not reported due to different gap boundaries. Consequently, we cannot automatically delete all appearances of u from $W_{u,i}[j]$ and from the data structure $AW_{v_i}[]$ of located neighbours of v_i, since it may cause misses of occurrence of some suffixes of u, as a node occurrence implies all the suffixes of the subpattern associated by that node occurred as well, so a deletion of the occurrence of u causes the suffixes of u to have no indication of it.

For example, consider the dictionary appearing in Figure 2 and suppose e_1 was reported due to a gap of size 2 between the subpatterns. Hence e_1, e_4 are reported, as e_4 is included in $next_{e_1}[2]$ list. If e_1 and e_4 are deleted from every $AW_c[j]$ or $W_{baa,c}[j]$, it causes and implicit deletion of e_2 and e_3 from these locations, since both are present in $AW_c[4-6]$ implicitly through the $next_{e_1}[3-5]$ lists. In case c occurs again with gap of $4-6$ locations, edges e_3 and e_2 should be reported, as it may be their only occurrence in the text. We, therefore, only assign $W_{baa,c}[3]$ to *null*, and $W_{baa,c}[j \neq 3]$ is updated to the longest suffix of baa, u'', where $e'' = (u'', c) \in E$, e'' appears in the list emanating from $next_{e_1}[j]$ and e'' was has not been reported yet. $AW_c[j]$ is updated accordingly. Our modified algorithm follows.

When a vertex arrives at query time t, the data structures are updated as follows.

1. For each $v_i \in R$, such that $rcount(v_i) \neq 0$,
   (a) the active window array AW_i is shifted by one position by incrementing its starting position in a cyclic manner,
   (b) $AW_i[\beta^* - \alpha^* + M + 1]$ is cleared (It may have been reported in the previous query when it was $AW_i[1]$).
2. If the arrived vertex is $v_i \in R$, such that $rcount(v_i) \neq 0$, then for every $W_{u,i}[j] \in AW_i[1]$,
   (a) $[e, j] = [(u, v_i), j] (= W_{u,i}[j], j)$,
   (b) Report & Update($[e, j]$)
      i. Edge e is reported,
      ii. $W_{u,i}[j] = null$,
      iii. $e' = next_e[j]$
      iv. if $e' \neq null$, then Report & Update($[e', j]$),
      v. For $f = \alpha_e$ to β_e, such that $f \neq j$, if $e' = next_e[f]$ and e' was reported, then
         A. $next_e[f] = next_{e'}[f]$,
         B. If $W_{u,v_i}[f]$ contains a pointer to e then $W_{u,v_i}[f] = next_e[f]$.
3. If the arrived vertex is $u \in L$, such that $lcount(u) \neq 0$, then
   For each $v_i \in R$ and each j, such that $W_{u,i}[j] \neq null$,
   (a) $(W_{u,i}[j], j)$ is inserted to the list at $AW_i[j + m_{v_i}]$, where m_{v_i} is the length of the subpattern associated with v_i, (since in $j + m_{v_i}$ time units from now, if $v_i \in R$ arrives, the edges pointed to by $W_{u,i}[j]$ should be reported).

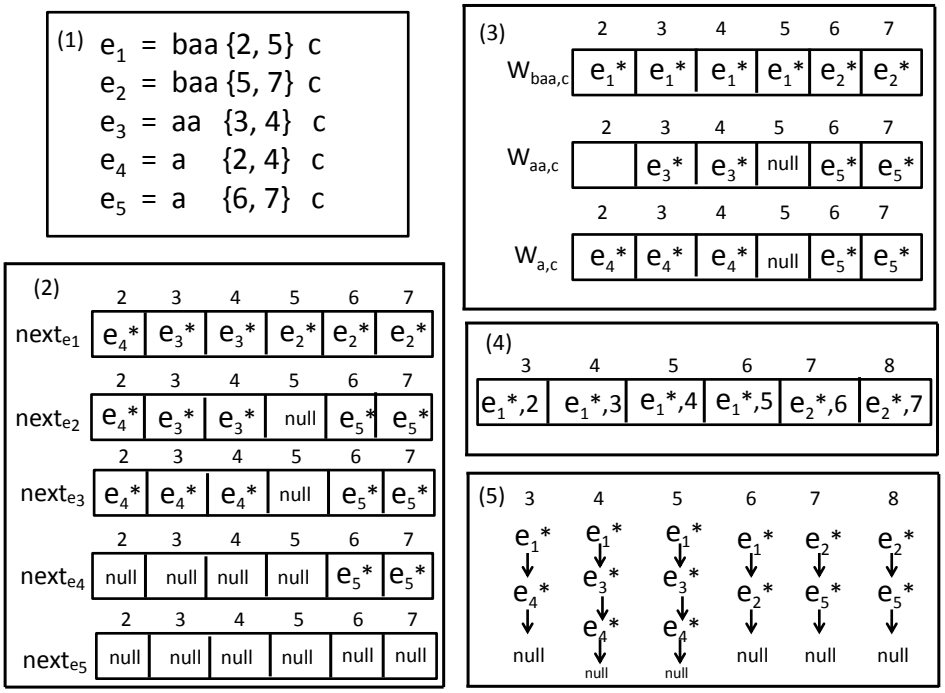


Figure 2. Consider the dictionary in (1). Its *next* arrays appear on (2), its W_u, v_i arrays appear on (3), AW_c after *baa* was found appears on (4), and the implicit lists that are saved in AW_c appear in (5). The * denotes a pointer to an edge.

Theorem 5 follows.

Theorem 5. *The online DROG problem with non-uniform gap borders can be solved in* $O(|D| + d(\beta^* - \alpha^*))$ *preprocessing time,* $\tilde{O}(lsc + \sqrt{lsc \cdot d}(\beta^* - \alpha^* + M) + op^*)$ *time per query text character, where* op^* *is the number of new distinct dictionary patterns reported due to the character arriving, and* $\tilde{O}(|D| + d(\beta^* - \alpha^*) + \sqrt{lsc \cdot d}(\beta^* - \alpha^* + M) + \alpha^*)$ *space.*

Proof. Correctness: To avoid repetitious reports of an edge e, after e is reported once due to a gap of size j', e is deleted both from G_D and from $W_{u^*,v_i}[j']$, where $(u^*, v_i) = e$ or $(u, v_i) = e$ and u represents a subpattern that is a suffix of the one represented by u^*. As described before, the deletion from $W_{u^*,v_i}[j]$, $j \neq j'$, yields an update to the array entry of the first edge to the $next_e[j]$ list that has not been reported yet, thus ensuring that a reported edge (u, v_i) is reported again. Nevertheless, unreported edges are not overlooked, even if it is of the form (u', v_i), where u' represents a suffix of the subpattern represented by u. Such edges are indicated by u in the $W_{u,v_i}[]$ array on indices where the gaps of the edges overlap.

Time Complexity: Due to the recursive nature of the updating procedure, every update of $W_{u,v_i}[j]$ requires $O(1)$ time, as the arrays of all u's are already updated, where u' represents a suffix of the subpattern represented by u. Recall from Subsection 2.2 that there are at most $\sqrt{lsc \cdot d}$ nodes currently in L, so that each report of an edge requires $O(\sqrt{lsc \cdot d}(\beta^* - \alpha^*))$ time. Such a time requirement already exists in the original $DMOG$ solution. The rest of the time complexity analysis is the same as

that of [4], except for the *op* factor which is replaced by a total of at most d reports due to the single report of every edge.

The space complexity is identical to that of the $DMOG$ solution. □

4 Conclusion and Open Problems

In this paper we give the first formalization of the Dictionary Recognition with One Gap (DROG) problem and give practical solutions for this problem in the online setting required for NIDS applications. Some open problems are:

- Can some of the factors in these solutions be reduced?
- Can these solutions be adapted to take care of a dictionary with subpatterns having more than one gap?

Since the DROG problem is a crucial bottleneck procedure in NIDS applications these open problems should be addressed in the future.

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: An aid to bibliographic search.* Commun. ACM, 18(6) 1975, pp. 333–340.
2. A. Amir, M. Farach, R. M. Idury, J. A. L. Poutré, and A. A. Schäffer: *Improved dynamic dictionary matching.* Inf. Comput., 119(2) 1995, pp. 258–282.
3. A. Amir, D. Keselman, G. M. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh: *Text indexing and dictionary matching with one error.* J. Algorithms, 37(2) 2000, pp. 309–325.
4. A. Amir, T. Kopelowitz, A. Levy, S. Pettie, E. Porat, and B. R. Shalom: *Mind the gap: Essentially optimal algorithms for online dictionary matching with one gap*, in 27th International Symposium on Algorithms and Computation, ISAAC 2016, December 12-14, 2016, Sydney, Australia, 2016, pp. 12:1–12:12.
5. A. Amir, A. Levy, E. Porat, and B. R. Shalom: *Dictionary matching with one gap*, in CPM, 2014, pp. 11–20.
6. A. Amir, A. Levy, E. Porat, and B. R. Shalom: *Dictionary matching with a few gaps.* Theor. Comput. Sci., 589 2015, pp. 34–46.
7. P. Bille, I. L. Gørtz, H. W. Vildhøj, and D. K. Wind: *String matching with variable length gaps.* Theor. Comput. Sci., 443 2012, pp. 25–34.
8. P. Bille and M. Thorup: *Regular expression matching with multi-strings and intervals*, in Proc. of SODA, 2010, pp. 1297–1308.
9. G. S. Brodal and L. Gasieniec: *Approximate dictionary queries*, in Proc. of CPM, 1996, pp. 65–74.
10. N. Chiba and T. Nishizeki: *Arboricity and subgraph listing algorithms.* SIAM J. Comput., 14(1) 1985, pp. 210–223.
11. R. Cole, L. Gottlieb, and M. Lewenstein: *Dictionary matching and indexing with errors and don't cares*, in Proc. of STOC, 2004, pp. 91–100.
12. K. Fredriksson and S. Grabowski: *Efficient algorithms for pattern matching with general gaps, character classes, and transposition invariance.* Inf. Retr., 11(4) 2008, pp. 335–357.
13. T. Haapasalo, P. Silvasti, S. Sippu, and E. Soisalon-Soininen: *Online dictionary matching with variable-length gaps*, in Proc. of SEA, 2011, pp. 76–87.
14. K. Hofmann, P. Bucher, L. Falquet, and A. Bairoch: *The PROSITE database, its status in 1999.* Nucleic Acids Research, 27(1) 1999, pp. 215–219.
15. W.-K. Hon, T.-W. Lam, R. Shah, S. V. Thankachan, H.-F. Ting, and Y. Yang: *Dictionary matching with uneven gaps*, in CPM, 2015, pp. 247–260.
16. G. Kucherov and M. Rusinowitch: *Matching a set of strings with variable length don't cares.* Theor. Comput. Sci., 178(1-2) 1997, pp. 129–154.
17. M. Morgante, A. Policriti, N. Vitacolonna, and A. Zuccolo: *Structured motifs search.* Journal of Computational Biology, 12(8) 2005, pp. 1065–1082.

18. C. W. Mortensen: *Fully dynamic orthogonal range reporting on RAM.* SIAM J. Comput., 35(6) 2006, pp. 1494–1525.
19. E. W. Myers: *A four russians algorithm for regular expression pattern matching.* J. ACM, 39(2) 1992, pp. 430–448.
20. G. Myers and G. Mehldau: *A system for pattern matching applications on biosequences.* CABIOS, 9(3) 1993, pp. 299–314.
21. G. Navarro and M. Raffinot: *Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching.* Journal of Computational Biology, 10(6) 2003, pp. 903–923.
22. Verint: *Personal communication*, 2013.
23. M. Zhang, Y. Zhang, and L. Hu: *A faster algorithm for matching a set of patterns with variable length don't cares.* Inf. Process. Lett., 110(6) 2010, pp. 216–220.

Range Queries Using Huffman Wavelet Trees

Gilad Baruch[1], Shmuel T. Klein[1], and Dana Shapira[2]

[1] Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
gilad.baruch@gmail.com, tomi@cs.biu.ac.il

[2] Dept. of Computer Science, Ariel University, Ariel 40700, Israel
shapird@ariel.ac.il

Abstract. A Wavelet Tree (WT) is a compact data structure which is used in order to perform various well defined operations directly on the compressed form of a file. Many algorithms that are based on WTs consider balanced binary trees as their shape. However, when non uniform repetitions occur in the underlying data, it may be better to use a Huffman structure, rather than a balanced tree, improving both storage and average processing time. We study distinct range queries and several related problems that may benefit from this change and present theoretical and empirical improvements in time and space complexities.

1 Introduction

Given an array A of n elements from an alphabet Σ, and indices *low* and *high*, consider the problem named *Distinct Range Queries* that returns the d distinct elements in $A[low, high]$. Here and below, $A[i, j]$ denotes the sub-array of A, consisting of the consecutive elements $A[i], A[i+1], \ldots, A[j]$, for $i \leq j$. For example, if $A = \texttt{xxxABRACADABRAyyyyy}$, then $n = 19$, and for range $[4, 14]$, we have $d = 5$ and the sought elements are $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}, \texttt{R}\}$. The goal is to preprocess A and generate a bounded amount of auxiliary information so that given a specific range, the query could be answered efficiently. There are several applications that use such queries. To mention just one, consider the case a list of the most traded stocks for the past n days is given, and one wishes to calculate the set of most traded stocks in some specific period of time, e.g., two months ago.

A trivial solution, without preprocessing, sorts the elements in the given range of size r, and computes the set of distinct elements by sequentially rescanning the sorted range in time $r \log r = O(n \log n)$, and without auxiliary storage.

A possible solution with preprocessing and auxiliary storage, would use a sliding window of size r, $1 \leq r \leq n$. Given a fixed range of size r, it will first compute, in $O(r)$ processing time, the set of distinct elements in the prefix $A[1, r]$ of the array, based on a constant time computation of the corresponding set for $A[1, r-1]$, $r \geq 2$. A table of size $|\Sigma|\lceil \log n \rceil$ bits will store the number of occurrences of each character in $A[1, r]$. The algorithm then slides the window of fixed size r, one character at a time, and compares the outgoing character to the incoming one, that is, it compares the first character of the current sliding range to the character just after that range. If these characters are equal, the set of distinct elements does not change. Otherwise, the new set of distinct elements can be determined in constant time by updating the table, for a total of $O(n - r + 1)$ time to process the entire array. The algorithm repeats this process for every r, $1 \leq r \leq n$, and stores the answer for every range

Gilad Baruch, Shmuel T. Klein, Dana Shapira: Range Queries Using Huffman Wavelet Trees, pp. 18–29.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

$[i, j]$, $1 \leq i, j \leq n$. Thus, this solution uses $(|\Sigma| n^2 \log n)$ memory space and $O(n^2)$ preprocessing time, but then answers the range query in constant time.

Another line of investigation considers *Wavelet trees*, defined by Grossi et al. [12]. A Wavelet tree (WT) T for an array A of n elements is a full binary tree whose leaves are labeled by the elements of Σ, and the internal nodes store bitmaps. The bitmap at the root contains n bits, in which the i^{th} bit is set to 0 or 1 depending on whether $A[i]$ is the label of a leaf that is stored in the left or right subtree of T. Each internal node v of T, is itself the root of a WT T_v for the *subarray* of A consisting only of the labels of the leaves of T_v, which are not necessarily consecutive elements of the array A. Balanced WTs can be constructed in $O(n \log |\Sigma|)$ time and require $n \log |\Sigma|(1 + o(1))$ bits.

The data structures associated with a WT for general prefix codes require some amount of additional storage (compared to the memory usage of the compressed file itself). Given a text string of length n over an alphabet Σ, the space required by Grossi et al.'s implementation can be bounded by $nH_h + O(\frac{n \log \log n}{\log_{|\Sigma|} n})$ bits, for all $h \geq 0$, where H_h denotes the hth-order empirical entropy of the text, which is at most $\log |\Sigma|$; processing time is just $O(m \log |\Sigma| + \text{polylog}(n))$ for searching any pattern sequence of length m. Multiary WTs replace the bitmaps by sequences over sublogarithmic sized alphabets in order to reduce the $O(\log |\Sigma|)$ height of binary WTs, and obtain the same space as the binary ones, but their times are reduced by an $O(\log \log n)$ factor. If the alphabet Σ is small enough, say $|\Sigma| = O(\text{polylog}(n))$, the tree height is a constant and so are the query times.

Many algorithms that are based on WTs consider balanced binary trees as their shape, that is, during the construction of each of the subtrees of the WT, the corresponding set of elements of Σ is split, at each stage, into two subsets of equal size, ± 1. However, when repetitions occur in the underlying data, it may be better using a Huffman structure, rather than a balanced tree, as suggested already in [16], improving both storage and average processing time. The contribution of this paper is to formalize this approach and conduct some empirical studies supporting its efficiency. Let H denote the zeroth order entropy of the given elements in A, and d the number of distinct elements in the range of the query, we show how to answer distinct range queries using a Huffman based WT, in $O(d(H + 1))$ processing time on average, and only $O(n(H + 1))$ auxiliary storage.

The rest of the paper is organized as follows. Section 2 reports on previous research. Section 3 presents the algorithm for solving the *distinct range query* problem by means of a Huffman WT. Section 4 considers other problems that can benefit from the use of Huffman WTs rather than balanced ones. Section 5 brings preliminary empirical evidence that using Huffman WTs may enhance processing time as well as storage usage as compared to the corresponding balanced WT.

2 Previous Work

Previous work has focused mainly on finding the k^{th} element in a given range, also named *Range Selection Queries*, and specifically on *Range Median Queries* in which k is equal to $\frac{n}{2}$. Krizanc et al. [14] presented the first preprocessing solution for *mode* and *median* queries, the mode of a given set being its most frequent element. In addition to mode and median range queries on lists, they also considered the general

settings of *path queries*, in which the input is given as a node labeled tree, and the query consists of two nodes. For the mode query they suggest an $O(n^{\epsilon} \log n)$ time and $O(n^{2-2\epsilon})$ space algorithm, where $0 < \epsilon < \frac{1}{2}$, while the median query could be answered in constant time using an $O(\frac{n^2 \log\log n}{\log n})$ space algorithm. For the median query, Petersen [18] improves the space to $O(\frac{n^2 \log^{(k)} n}{\log n})$, still answering the query in constant time, where k is a constant and $\log^{(k)}$ is the k times iterated logarithm. Unlike the near quadratic space of Petersen, the best known linear space solution is due to Chan et al. [4] and requires $O(\sqrt{\frac{n}{\log n}})$ query time.

Range Least Frequent Element Queries on arrays were studied by Chan et al. in [5], and improved by Durocher et al. in [6]. Durocher et al. [7] study the *Range Majority Query* problem, which asks to report the mode in $A[low, high]$ only if the mode occurs more than half of the times in the range. Given a real number $0 < \tau \leq 1$, Navarro et al. [17] consider a generalization where any element occurring a fraction of times larger than τ in $A[low, high]$ can be reported. Thus a majority corresponds to $\tau = \frac{1}{2}$. They prove a lower bound of $\Omega(n\lceil\log(\frac{1}{\tau})\rceil)$ bits, without storing A, for any data structure supporting τ majorities within any range, and present a data structure that returns a single position of each τ-majority, and obtains this space lower bound, in running time $O(\frac{1}{\tau} \log\log_w(\frac{1}{\tau}) \log n)$, on a RAM machine with word size w. As extension, Huffman WTs can also be used when considering Range Least Frequent Element Queries and Range Majority Queries, yielding an improvement as can be found in Table 1.

A problem related to the range selection queries is *Range Rank Queries* (or range dominance queries), where, given indices i, j and a value e, the goal is to return the number of elements from $A[i, j]$ that are less than or equal to e (dominated by e). Brodal et al. [3] designed a static linear space data structure that supports both range selection and range rank queries in $O(\log n / \log\log n)$ time. In [2] the authors suggest a linear space and $O(n \log n)$ preprocessing time solution to the median range queries problem, with the same time complexity per query. Their data structure sorts the input elements and places them in the leaves of a balanced binary search tree. Consider a search for the k^{th} smallest element in $A[i, j]$. If the left subtree of the root contains k or more elements from $A[i, j]$ then it contains the k^{th} smallest element from $A[i, j]$. If not, the sought element is in the right subtree. Each node of the tree stores the prefix sum such that the number of elements from $A[1, j]$ contained in the left subtree can be determined for any j. The space is then reduced to $O(n)$ using rank and select data structures defined as:

$\mathsf{rank}_\sigma(A, i)$ – returns the **number** of occurrences of $\sigma \in \Sigma$ in A up to and including position i;
$\mathsf{select}_\sigma(A, i)$ – returns the **position** of the ith occurrence of $\sigma \in \Sigma$ in A.

Given a range $[low, high]$ and an element x, the *Range Counting Query* problem is counting the number of occurrences of x in $A[low, high]$. Krizanc et al. [14] use a series of sorted arrays, one for each element in Σ. The array for element x, denoted by A_x, contains the indices $1 \leq i \leq n$ such that $a_i = x$ in sorted order. Given a range $[low, high]$ and an element x, binary search is applied on A_x in order to find the indices ℓ and h of low and $high$, respectively. The number of occurrences of x is then $h - \ell + 1$. This solution uses $O(n)$ words of storage and $O(\log n)$ processing

time. It should be noted that a space of $O(n)$ words is equal to $O(n \log n)$ bits in the word-RAM model, in which a word size is $\Theta(\log n)$. By applying the predecessor data structure of van Emde Boas [8] instead of binary search, Range Counting Queries over the integer alphabet $[1..u]$ can be answered in $O(\log \log u)$ time using $O(u \log u)$ bits. If the length of the string is much smaller than the alphabet size, i.e., if $n \ll u$, then *Y-fast tries* can be used, with $O(\log \log n)$ time using $O(n \log n)$ bits [20].

Muthukrishnan [15] solved the *Distinct range query* problem, also called the *colored range listing* problem, as part of a solution to the *document listing problem* for listing all distinct documents containing a given pattern. His solution is based on defining an additional array C, so that $C[k]$ is the largest value $i < k$ such that $A[i] = A[k]$, or 0 if there is no such i. $A[k]$ is then the first occurrence of this element in the range $A[i, j]$ if and only if $C[k] < i$. Thus, if the minimum value in $C[i, j]$ is $C[k]$, the element $A[k]$ is reported as a new element in the range if and only if $C[k] < i$. All other distinct elements in the (original) range are reported by recursively applying the same method on the sub-arrays $C[i, k-1]$ and $C[k+1, j]$. The constant time *Range Minimum Queries (RMQ)* data structure, due to Gabow et al. [9] is used for a total of $O(d)$ time and $O(n \log n)$ space, where d is the number of distinct elements.

Välimäki and Mäkinen [19] reduce the space of Muthukrishnan's data structure by means of a multiary wavelet tree, using $O(n \log |\Sigma|)$ bits and $O(d \frac{\log(|\Sigma|}{\log \log n})$ time. Their idea is based on the rank and select data structures used in the internal nodes of the mulitary wavelet tree. They give an alternative way for computing the value $C[k]$ used in Muthukrishnan's solution as $C[k] = \mathsf{select}_{A[k]}(A, \mathsf{rank}_{A[k]}(A, k) - 1)$.

Gagie et al. [10] eliminate the use of RMQ's and suggest a binary WT for solving range quantile queries and distinct range queries, using the same size of auxiliary space and $O(d \log |\Sigma|)$ processing time. In particular, range counting queries are solved by them in $O(\log |\Sigma|)$ time. Unlike this solution which is based on a binary balanced Wavelet tree, we examine the use of the Huffman tree that corresponds to the number of occurrences of the items in A as the structure of the WT.

Concentrating on the shape of the WT was recently done by Klein and Shapira [13] and Baruch et al. [1], where a pruning strategy was applied to the WTs in order to reduce the overhead of the additional storage used by the data structures for processing the stored bitmaps. Moreover, the average path lengths corresponding to the codewords was also decreased, thus implying a reduction of the average random access time.

Table 1 summarizes the results. The variable $w = \Omega(\log n)$ stands for the word size.

3 Distinct Range Queries

Recall that the binary tree T_C corresponding to a prefix code C is defined as follows: we imagine that every edge pointing to a left child is labeled 0 and every edge pointing to a right child is labeled 1; each node v is associated with the bit string obtained by concatenating the labels on the edges on the path from the root to v; finally, T_C is defined as the binary tree for which the set of bit strings associated with its leaves is the code C.

WTs can be defined for a text array over any prefix code and the tree structure is inherited from the tree usually associated with the code. Considering the WT as

	Processing Time	*Space (bits)*
Distinct Range Queries		
Välimäki et al. [19]	$O(d\frac{\log(\lvert\Sigma\rvert)}{\log\log n})$	$O(n\log\lvert\Sigma\rvert)$
Gagie et al. [10]	$O(d\log\lvert\Sigma\rvert)$	$O(n\log\lvert\Sigma\rvert)$
Section 3	$O(d(H+1))$ average time	$O(n(H+1))$
Range Counting Queries		
Krizanc et al. [14]	$O(\log\log n)$	$O(n\log n)$
Gagie et al. [10]	$O(\log\lvert\Sigma\rvert)$	$O(n\log\lvert\Sigma\rvert)$
Section 4	$O(H+1)$ average time	$O(n(H+1))$
Range Mode Queries		
Petersen [18]	$O(1)$	$O(\frac{n^2\log^{(k)}n}{\log n})$
Chan et al. [4]	$O(\sqrt{n/\log n})$	$O(n\log n)$
Section 4	$O(d(H+1))$ average time	$O(n(H+1))$
Range Least Frequent Element Queries		
Chan et al. [5]	$O(\sqrt{n})$	$O(n\log n)$
Durocher et al. [6]	$O(\sqrt{n/w})$	$O(n\log n)$
Section 4	$O(d(H+1))$ average time	$O(n(H+1))$
Range Majority Queries		
Chan et al. [7]	$O(1)$	$O(n\log n)$
Section 4	$O(H+1)$ average time	$O(n(H+1))$

Table 1. Time and space complexities for range queries.

associated with the prefix code, rather than with the text array itself, yields the following equivalent definition, as alternative to the one given in the introduction. The root holds the bitmap obtained by concatenating the *first* bit of each of the sequence of codewords in the order they appear in the encoded text. The left and right children of the root hold, respectively, the bitmaps obtained by concatenating, again in the given order, the *second* bit of each of the codewords starting with 0, respectively with 1. This process is repeated similarly on the next levels: the grandchildren of the root hold the bitmaps obtained by concatenating the *third* bit of the sequence of codewords starting, respectively, with 00, 01, 10 or 11, if they exist at all, etc.

The bitmaps in the nodes of the WT can be stored as a single bit stream by concatenating them in order of any predetermined top-down tree traversal, such as depth-first or breadth-first. No delimiters between the individual bitmaps are required, since we can restore the tree topology along with the bitmaps lengths at each node once the size n of the text is given in the header of the file.

Let the weights $\{w_1, w_2, \ldots, w_k\}$ be the number of occurrences of the individual characters in $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$, respectively. It is well known that Huffman's encoding is optimal, and assigns codeword lengths $\{\ell_1, \ell_2, \ldots, \ell_k\}$ so that $W = \sum_{i=1}^{k} w_i\ell_i$ is minimal. Let us assume that $\sigma_1, \ldots, \sigma_k \in \Sigma$ occur $\{w'_1, w'_2, \ldots, w'_k\}$ times in $A[low, high]$ ($w'_i = 0$ for characters that do not occur in the given range). A Huffman based WT requires only $O(W)$ space and $O(\sum_{i=1}^{k} w'_i\ell_i)$ processing time. Notice the following:

1. There are d non zero terms in $\sum_{i=1}^{k} w'_i\ell_i$;
2. $W \leq n\log\lvert\Sigma\rvert$;
3. $\sum_{i=1}^{k} w'_i\ell_i \leq d\log\lvert\Sigma\rvert$;

The last two points indicate that Huffman based WTs may improve both space and processing time of the WTs of Gagie et al. [10].

The algorithm for extracting the distinct elements in the range $[low, high]$ of an array A by means of a Huffman WT rooted by v_{root} is given in Algorithm 1, using the function call distinct(v_{root}, low, $high$). B_v denotes the bitmap belonging to vertex v of the Wavelet tree. The variables num_0 and num_1 are assigned the number of 0s and 1s in the given range in lines 3.1 and 3.2, respectively, by subtracting the number of 0s/1s up to the beginning of the range from the number of 0s/1s up to the end of the range. Branching left or right depends on whether there are 0s or 1s in the current range. If num_0 is greater than 0, the process continues on the left subtree, and if num_1 is greater than 0, it continues (also) on the right subtree. Computing the new range in the following bitmap is done by applying the rank operation on both ends of the current range. As a side effect, when processing a leaf v, the number of occurrences of the corresponding element is also computed, based on the number of 0s or 1s in the parent node of v.

Distinct$(v, low, high)$
1 $num \leftarrow high - low + 1$
2 if v is a leaf
2.1 output element corresponding to v and its frequency num
2.2 return
3 else
3.1 $num_0 \leftarrow rank_0(B_v, high) - rank_0(B_v, low - 1)$
3.2 $num_1 \leftarrow num - num_0$
3.3 if $num_0 > 0$
3.3.1 Distinct $(left(v), rank_0(B_v, low - 1) + 1, rank_0(B_v, high))$
3.4 if $num_1 > 0$
3.4.1 Distinct $(right(v), rank_1(B_v, low - 1) + 1, rank_1(B_v, high))$

Algorithm 1. Extracting the distinct elements of $A[low, high]$ from a Wavelet tree.

Consider for example the tree in Figure 1, which represents a Wavelet tree for some array A. Assume that the substring of A from position 4 to position 14 contains `abracadabra` and consider the query with $low = 4$ and $high = 14$. Note that the leaves are sorted from left to right according to the number of their occurrences in the entire array A. At the beginning we are looking for the leftmost leaf corresponding to an element that occurs in the given range. There are 0s in the given range in the bitmap stored in the root, meaning that the range contains elements corresponding to the left subtree, thus v is assigned the left child of the root. The new range is computed to be from 3 to 7, according to the number of 0s $num_0 = 5$ in the range $[4, 14]$ in the bitmap of the root, and the number of 0s preceding the range, which is 2 in this example. As all bits in the range $[3, 7]$ in the bitmap of the left child of the root are 1s, the element `e` does not occur in the range, and the left subtree can be skipped, going directly to the right child of the left child of the root. The new range is computed to be [2,6], and as the corresponding bitmap is all 0s, the algorithm continues with the left child, and character `a` with frequency 5 is reported. This process continues until all elements of the range are reported, skipping subtrees that do not contain leaves with labels in the range.

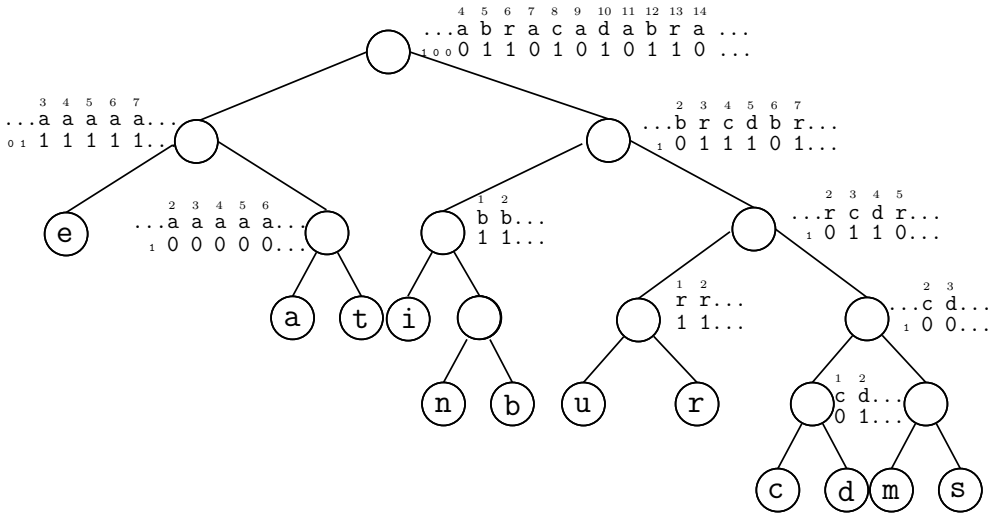


Figure 1. A Range Query on the Wavelet tree induced by the canonical Huffman tree corresponding to the frequencies {20, 9, 9, 9, 5, 5, 5, 5, 2, 2, 2, 2} of {e, a, t, i, n, b, u, r, c, d, m, s}, respectively.

As mentioned in Section 2, the algorithm of Gagie et al. [10] for Distinct Range Queries, runs in $O(d \log |\Sigma|)$ time, and uses $O(n \log |\Sigma|)$ space. It is important to note that given a specific range, the running time $O(d(H+1))$ of Algorithm 1, could be longer than the $O(d \log |\Sigma|)$, suggested by Gagie. This happens when the distribution of the characters within the given range significantly deviates from this distribution in the entire text. However, the improvement of the average running time is based on the assumption that there is no such discrepancy between the partial range and one spanning the entire text, resulting in a reduction in running time. Nevertheless, the storage of the entire WT requires generally less space than a balanced WT, and only if the distribution of the character frequencies is close to uniform, both will produce an $O(n \log |\Sigma|)$ space data structure.

Another interesting bound can be derived on the worst case running time of Algorithm 1. The Range Distinct Elements algorithm runs on the Huffman tree, possibly skipping several subtrees in case the relevant bitmap contains only 0s or only 1s. In the worst case, when all characters of Σ appear in the given range, the entire Huffman tree is processed. Thus, the running time is bounded by the total number of nodes in the Huffman tree, which is $O(|\Sigma|)$, and may be independent of n.

The results can be summarized in the following theorem:

THEOREM 1: *There exists a data structure of size $O(W)$ bits which can be built in $O(W)$ time, that answers distinct range queries on $A[i,j]$ for $1 \leq i \leq j \leq n$ in $O(d(H+1))$ average time.*

4 Range Mode, Range Least, Range Counting, and Range Majority Queries

The operation $rank_\sigma(A, i)$ is defined as computing the number of occurrences of σ in A up to position i. This can be adapted quite easily in order to compute the number of occurrences of σ in a given range $[low, high]$ by simply calculating $rank_\sigma(A, high) - rank_\sigma(A, low - 1)$. A WT can be used to compute $rank_\sigma(A, i)$ in time proportional

to the length of the path starting at the root and ending at the leaf corresponding to σ. Using a Huffman based WT, this time is $O(H+1)$ on average, where the WT occupies $O(W)$ bits. Though the $O(\log\log n)$ time algorithm of Krizanc et al. [14] for Range Counting Queries is usually faster than the $O(H+1)$ average time of our suggested algorithm, their $O(n\log n)$ memory space is larger than the $O(W)$ space we use.

Given a range $[low, high]$, the *Range Mode Query* reports the most frequent element in $A[low, high]$, or one of them if there are several. As mentioned above, Chan et al. [4] present an $O(\sqrt{\frac{n}{\log n}})$ query time algorithm for this problem, using $O(n\log n)$ bits for storage. We note that the problem of finding the mode of a given range can also be solved by using a balanced Wavelet tree, by computing Range Counting Queries for each distinct element in the range. This solution suggests a method requiring $O(d\log|\Sigma|)$ processing time and $O(n\log|\Sigma|)$ space. By applying Huffman shaped WTs, the time is reduced to $O(d(H+1))$ and to only $O(W)$ space. In more details, the algorithm presented for Distinct Range Queries can also be used to solve Range Mode Queries, no matter whether the underlying shape of the Wavelet tree is balanced or Huffman. As described above, as a side effect of this algorithm, the number of occurrences of each element is also computed each time a leaf is processed. We can therefore answer Range Mode Queries in time $O(d(H+1))$, using a Huffman shaped WT, and in both cases the times are bounded by $O(|\Sigma|)$.

Note that if an unbounded alphabet Σ is assumed, the traditional WT and the Huffman shaped WT algorithms are worse than the $O(\sqrt{n/\log n})$ of Chan et al., but reduce the processing time in the case of a finite alphabet. However, the WTs algorithms may still be useful when the number of distinct elements d in the given range is small, e.g., when $d = \log n$, which can happen even in the case of an unbounded alphabet. Moreover, in the bounded and unbounded cases, using WTs needs only $O(n\log|\Sigma|)$ and $O(W)$ space for traditional and Huffman shaped Wavelet trees, respectively, as compared to $O(n\log n)$ of Chan et al.. The same discussion applies also to a symmetric problem named *Range Least Frequent Element*.

The algorithm for solving Range Majority Queries in a given range $[low, high]$ of an array A, by means of a Huffman WT rooted at v_{root}, is given in Algorithm 2, using the function call majority(v_{root}, low, $high$, $(high-low+1)/2$). As the majority depends on the number of elements in the original range, the last argument of the function giving the majority bound is passed through all recursive calls. The variables B_v, num_0 and num_1 are the same as in Algorithm 1. Branching left or right depends on whether the number of 0s or 1s is greater than the required target value $m = (high-low+1)/2$ in the current range. This time, at most one of the subtrees will be processed. If num_0 is greater than m, the process continues on the left subtree, otherwise, if num_1 is greater than m, it continues on the right subtree. If neither of num_0 and num_1 are greater than m, there is no majority element in A, and the process terminates after reporting so. This algorithm runs in $H+1$ time on average, unlike the constant time of Durocher et al. [7]. However, it only uses $O(W) \leq O(n(H+1))$ space rather than $O(n\log n)$.

Gagie et al. [10] use a balanced WT for finding the k^{th} element in time $O(\log|\Sigma|)$ and $O(n\log n)$ space. In our paradigm the elements are sorted by frequencies in the *entire* array, thus the problem is now finding the k^{th} frequent element in a given range. In fact, the same algorithm can be used on a Huffman shaped WT, and produces an

```
majority(v, low, high, m)
1    num ← high − low + 1
2    if v is a leaf
2.1        output element corresponding to v
2.2        return
3    else
3.1        num_0 ← rank_0(B_v, high) − rank_0(B_v, low − 1)
3.2        num_1 ← num − num_0
3.3        if num_0 ≥ m
3.3.1          Majority (left(v), rank_0(B_v, low − 1) + 1, rank_0(B_v, high), m)
3.4        else if num_1 ≥ m
3.4.1          Majority (right(v), rank_1(B_v, low − 1) + 1, rank_1(B_v, high), m)
3.5        else
3.5.1          output "no Majority in Range"
3.5.2          return
```

Algorithm 2. Majority Query on $A[low, high]$.

average running time of $O(H+1)$ and only $O(W) \leq O(n(H+1))$ space. The algorithm is similar to Algorithm 2.

5 Experimental Results

For our preliminary experiments we considered two different files of different languages and alphabet sizes. The Bible (King James version) in English, *ebib*, in which the text was stripped of all punctuation signs, and the French version of the European Union's JOC corpus, *ftxt*, which is a collection of pairs of questions and answers on various topics used in the ARCADE evaluation project. Our implementation used the *Succinct Data Structure Library* [11], which is an open-source library implementing succinct data structures efficiently in C++. All experiments were conducted on a machine running 64 bit Linux Ubuntu with an Intel Core i7-4720 at 2.60 GHz processor, 6144K L3 cache size of the CPU, and 4 GB of main memory.

The files were encoded as a sequence of characters as well as a sequence of *words* (a maximal sequence of non whitespace characters), producing two different alphabets, a small and a large one. Table 2 presents some information on the data files involved. The second column presents the original file sizes in MB. The third and fourth columns give the number of elements in the character alphabet (chars) and the word alphabet (words), respectively. The size of the word alphabet is given in thousands of (different) words. The number of words in the file, including repetitions, is given in the fifth column, in millions.

Our first experiment compares the processing times for the distinct elements range query problem, using balanced and Huffman WTs. The range sizes were chosen as a series of increasing powers of 2, starting with 1 and up to the size of 256. For each of the test files and range sizes, the range query was run 1000 times, with randomly chosen starting points. The displayed plots are the averages over these runs. Figures 2 and 3 present the processing times for our dataset for the alphabet of characters and

File	size	chars	words	Words in text
	(MB)		(in thousands)	(in millions)
ebib	3.5	53	11	0.6
ftxt	7.6	132	75	1.2

Table 2. Information about the used datasets

words, respectively. The plots are given on a log scale, showing the processing time, in microseconds, as function of the range size, measured in number of characters.

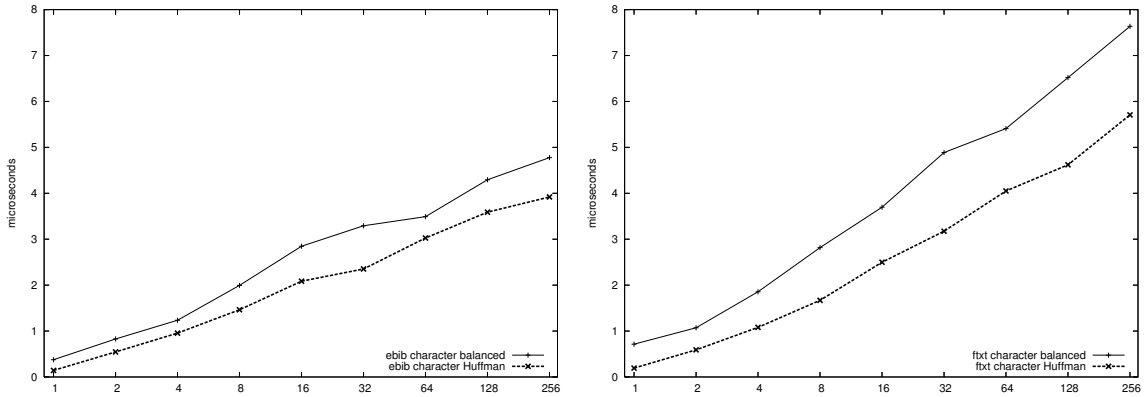


Figure 2. Processing time as function of the range size with character alphabet.

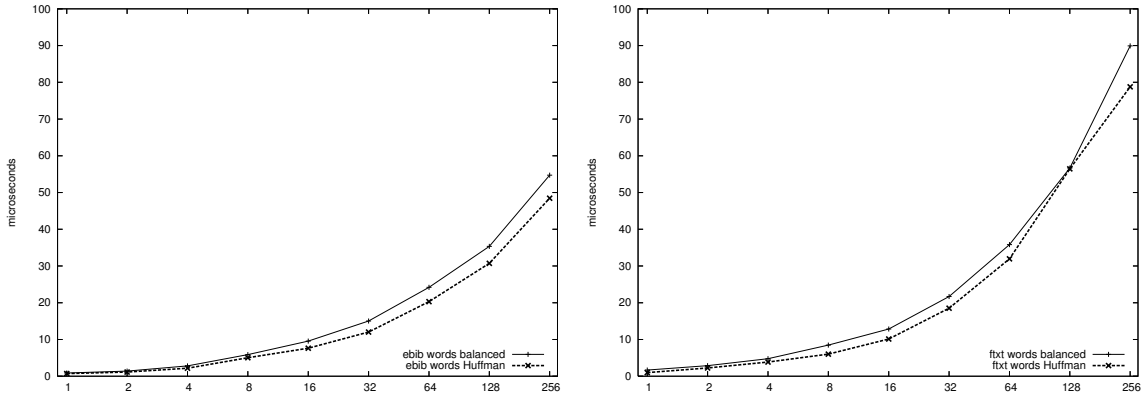


Figure 3. Processing time as function of the range size with word alphabet.

As can be seen, processing the Huffman WT is consistently faster than processing the balanced one, for ranges up to 256. The ratio of the improvement of Huffman over balanced WTs reduces as the ranges become longer. This can be explained by the fact that the probability that longer ranges include also less frequent characters becomes higher, requiring longer processing times for the deeper leaves. Thus, there

are cases in which for a given range the running time of the balanced WT can be faster than the Huffman one, and the advantage of the Huffman structure vanishes.

In the following table we present the storage usage in MBs of balanced versus Huffman WTs on both our datasets, and for the two kinds of alphabets. As expected, the storage of the entire Huffman WT, including the rank and select data structures, requires less space than the corresponding balanced WT, because of the skewed probabilities of the underlying alphabets. Although we expected that the word based WTs will generally save space as compared to that corresponding to characters, this is not the case for the Huffman WTs on *ftxt*. This can be explained by the overhead requirements of the rank and select data structures that are needed for a larger set of nodes.

File	**Character alphabet**		**Word alphabet**	
	Balanced	Huffman	Balanced	Huffman
ebib	3.92	2.77	2.54	2.01
ftxt	11.38	7.16	9.69	8.34

Table 3. Comparison of storage usage.

References

1. Baruch, G. and Klein, S. T. and Shapira, D., A Space Efficient Direct Access Data Structure, *The Journal of Discrete Algorithms*, (2017) 26–37.
2. G.S. Brodal, B. Gfeller, A.G. Jørgensen, P. Sanders, Towards Optimal Range Median, *Theoretical Computer Science, Special issue of ICALP'09*, **412**(24) (2011) 2588–2601.
3. G.S. Brodal, A.G. Jørgensen, Data Structures for Range Median Queries, *Algorithms and Computation*, (2009) 822-831.
4. T.M. Chan, S. Durocher, K.G. Larsen, J. Morrison, B.T. Wilkinson, Linear-Space Data Structures for Range Mode Query in Arrays, *Theory Comput. Syst.*, (2014) 719–741.
5. T.M. Chan, S. Durocher, M. Skala, B.T. Wilkinson, Linear-Space Data Structures for Range Minority Query in Arrays. *In Proc. SWAT*, **7357** (2012) 295–306.
6. S. Durocher, R. Shah, M. Skala, S.V. Thankachan, Linear-Space Data Structures for Range Frequency Queries on Arrays and Trees, *MFCS*, (2013) 325–336.
7. S. Durocher, M. He, J.I. Munro, P.K. Nicholson, M. Skala, Range majority in constant time and linear space, *Inf. Comput.*, (2013) 169–179.
8. P. van Emde Boas, Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space, *Information Processing Letters*, **6**(3) (1977) 80–82.
9. H. N. Gabow, J. L. Bentely, R. E. Tarjan Scaling and related techniques for geometry problems, *Proc. STOC*, (1984) 135–143.
10. T. Gagie, S.J. Puglisi, A. Turpin, Range Quantile Queries: Another Virtue of Wavelet Trees, *SPIRE '09 Proceedings of the 16th International Symposium on String Processing and Information Retrieval* (2009) 1–6.
11. S. Gog, T. Beller, A. Moffat, M. Petri, From theory to practice: plug and play with succinct data structures, *13th International Symposium on Experimental Algorithms, (SEA 2014)*, Copenhagen (2014) 326–337.

12. R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes, *Proceedings of the 14th Annual SIAM/ACM Symposium on Discrete Algorithms (SODA)* (2003) 841–850.
13. Klein, S. T. and Shapira, D., Random access to Fibonacci codes, *Discrete Applied Mathematics*, **212**, (2016) 115–128.
14. D. Krizanc, P. Morin, M.H.M. Smid, Range mode and range median queries on list and trees, *Nordic Journal of Computing*, **12** (2005) 1–17.
15. S. Muthukrishnan, Efficient algorithms for document retrieval problems. *Proc. SODA'02*, (2002) 657–666.
16. G. Navarro, Wavelet Trees for All, *Journal of Discrete Algorithms*, **25** (2014) 2–20.
17. G. Navarro, S.V. Thankachan, Optimal Encodings for Range Majority Queries, *Algorithmica*, **74** (2016) 1082–1098.
18. H. Petersen, Improved bounds for range mode and range median queries, *Proc. of the 34th Conference on Current Trends in Theory and Practice of Computer Science*, (2008) 418–423.
19. N. Välimäki, V. Mäkinen, Space-efficient algorithms for document retrieval, *Proc. CPM*, (2007) 205–215.
20. D. E. Willard Log-logarithmic worst-case range queries are possible in space $\Theta(N)$ *proc. Information Processing Letters*, (1983) **17**(2) 81–84.

Regular Expressions with Backreferences Re-examined

Martin Berglund[1,2] and Brink van der Merwe[3]

[1] Department of Information Science, Stellenbosch University, South Africa
[2] Center for AI Research, CSIR, Stellenbosch University, South Africa
[3] Department of Computer Science, Stellenbosch University, South Africa

Abstract. Most modern regular expression matching libraries (one of the rare exceptions being Google's RE2) allow backreferences, operations which bind a substring to a variable allowing it to be matched again verbatim. However, different implementations not only vary in the syntax permitted when using backreferences, but both implementations and definitions in the literature offer up a number of different variants on how backreferences match. Our aim is to compare the various flavors by considering the formal languages that each can describe, resulting in the establishment of a hierarchy of language classes. Beyond the hierarchy itself, some complexity results are given, and as part of the effort on comparing language classes new pumping lemmas are established, and old ones extended to new classes.

1 Introduction

Regular expressions as used and implemented in practice are vastly different from their traditional theoretic counterpart, both in semantics (driven by the features offered), and expectations of performance. Even when not using the more complex features the performance profile of practical regular expression matching is a fairly deep subject matter, which has seen theoretical study only fairly recently, such as in [2] and [7]. In this paper we focus on regular expressions with backreferences (rewbr for short), an advanced feature which is available in most regular expression matching libraries. This subject matter has seen some study in the literature, we will refer frequently to [1], [4], and [5], but each paper has its own definition of a rewbr and its semantics ([4] has two), and many implementations disagree with all of them (the definition given by Aho in [1] is common however), and with each other.

A backreference is placed in a regular expression to indicate that the substring matched by some specified capturing group (where capturing group is synonymous with parenthesized subexpression), should be matched again at the position (or positions) where the backreference is placed. In the Java programming language we denote by `\i` that the substring most recently matched by the ith capturing group should be matched by the backreference again, where capturing groups are numbered from 1 onwards, based on the relative position of their left parenthesis when reading the regular expression from left to right. For example, `[0-9]+\.\d*(\d+)\1+` can be used to match recurring decimal numbers, such as 0.33, 0.818181 and 0.04555, since the subexpression `(\d+)` captures some sequence of digits in the input string and the backreference `\1+` instructs the matcher to match this sequence again, one or more times. Similarly, the regular expression `(.+)\1` matches strings of the form `ww`, i.e. producing the non-context-free reduplication property.

Long and complicated regular expressions may be hard to read and maintain as adding or removing capturing groups changes the numbers of all groups following the

Martin Berglund, Brink van der Merwe: Regular Expressions with Backreferences Re-examined, pp. 30–41.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

modification. Python's `re` module was the first to offer a solution in terms named capturing groups and backreferences — `(?P<name>group)` captures the match of the subexpression `group` into `name`, whereas a backreference to the contents of this capturing group is done with `(?P=name)`. In some implementations it is then possible to *reuse* the same label for *different* capturing groups (e.g. Python and .NET both allow naming of groups, but .NET allows reusing names where Python does not), which opens possibilities obviously not available when simply numbering capturing groups from left to right. Also, regular expression matchers use different conventions in terms of how matching is defined when encountering a backreference without having captured a substring with a label corresponding to the backreference. These subtle differences in syntax and semantics allowed in rewbr influence the classes of languages described, as well as the relative succinctness of the rewbr variants. It is thus clear that a thorough comparison of rewbr variants is needed if further study is to be possible, which forms a big part of our contribution.

This paper uses as starting point the definitions and results, on rewbr, from [1], [4], [6] and [5]. In particular, the structure of the definition of matching semantics of rewbr is taken from [6], and the pumping lemma from [4] (for rewbr) provides the intuition for our own more general pumping lemmas.

The outline of the paper is as follows. After providing the necessary notation and definitions in the next section, we first give some improvements on past complexity results (demonstrating some differences between the classes), we then develop various pumping lemmas and then describe the relationships between the language classes obtained when considering the variants of rewbr as found in theory and practice.

2 Notation and Definitions

We use Σ and Φ as finite input and backreference alphabets respectively, with these (possibly empty) alphabets being disjoint. Also, $\emptyset$ and ε denote the empty set and word respectively, and $\mathbb{N}$ the set of natural numbers including 0. To improve readability, we sometimes denote $v_1 = w_1, \ldots, v_n = w_n$ as $(v_1, \ldots, v_n) = (w_1, \ldots, w_n)$. For a string w over Σ (or any other alphabet), we denote by $|w|$ the length of w, i.e. the number of occurrences of symbols from Σ in w, and more generally, if $\Sigma' \subseteq \Sigma$, then $|w|_{\Sigma'}$ is the number of occurrences of symbols from Σ' in w. For sets A and B of strings, the concatenations $A \cdot B$ is defined as usual as $\{w_1 w_2 \mid w_1 \in A, w_2 \in B\}$ and the Kleene closure of A, denoted as A^*, as $\{\varepsilon\} \cup (\cup_{i=1}^{\infty} A^i)$, where A^i is the concatenation of A with itself using the concatenation operator $(i-1)$ times.

Definition 1. *A* regular expression with backreferences (rewbr) *over input alphabet Σ and backreference alphabet Φ is defined inductively as:*

(1) $\emptyset$, or an element of $\Sigma \cup \{\varepsilon\}$, or
(2) an expression of the form $(E_1 \mid E_2)$, $(E_1 \cdot E_2)$, or, $(E_1{}^*)$, the Kleene closure, *for any rewbr E_1 and E_2, or,*
(3) for $\phi \in \Phi$, the expression $[_\phi E]_\phi$, i.e. a capturing group *labeled by ϕ, or $\uparrow_\phi$, a* backreference *to a (possibly non-existing) capturing group labeled by ϕ.*

Let $rewbr_{\Sigma,\Phi}$ denote all rewbr over input alphabet Σ and backreference alphabet Φ. The subset of rewbr obtained by using only (1) and (2) above (i.e. regular expressions over Σ without backreferences), is denoted by $\mathfrak{R}_\Sigma$.

We use Σ and Φ to indicate a generic input and backreference alphabet respectively, without stating it explicitly.

As usual, parenthesis may be elided from rewbr by using the rule that Kleene closure ' * ' takes precedence over concatenation ' · ', which takes precedence over union ' | '. In addition, outermost parenthesis may be dropped and $E_1 \cdot E_2$ abbreviated as E_1E_2. The brackets which denote a capturing group may not be elided (except if no corresponding backreference appears in the rewbr). Also, for $E \in \text{rewbr}_{\Sigma,\Phi}$, we use E^+ as abbreviation for $E \cdot E^*$.

When $E \in \mathfrak{R}_\Sigma$, the language described by E, denoted as $\mathcal{L}(E)$, is defined inductively as usual, i.e. $\mathcal{L}(\emptyset) = \emptyset, \mathcal{L}(a) = \{a\}$ for $a \in \Sigma \cup \{\varepsilon\}$, $\mathcal{L}(E_1 \,|\, E_2) = \mathcal{L}(E_1) \cup \mathcal{L}(E_2)$, $\mathcal{L}(E_1 \cdot E_2) = \mathcal{L}(E_1) \cdot \mathcal{L}(E_2)$ and $\mathcal{L}(E_1{}^*) = \mathcal{L}(E_1)^*$.

Following [6] and [5], we use *ref-words* (short for *reference words*), to define the matching semantics of rewbr, instead of using the approach of Câmpeanu et al. in [4]. Câmpeanu and his co-authors used parse trees as mechanism to describe the way in which a string is matched in terms of which substring of the input string is captured by which subexpression of the rewbr. A backreference then matches a substring that is equal to the closest matched substring w' to its left in the input string, where w' was matched/captured by a subexpression labeled by the same symbol as used by the backreference. The ref-words and parse tree approaches of arriving at matching semantics are indeed equivalent. We use the ref-words approach, explained next, since it allows us to show that the various pumping lemmas for rewbr is a direct consequence of the regular language counterpart.

Let $\Sigma_\Phi = \Sigma \cup \{\phi, (_\phi,)_\phi \mid \phi \in \Phi\}$ and $w \in (\Sigma_\Phi)^*$. Then if ϕ appears in w and no subword of the form $(_\phi u)_\phi$ appears to the left of ϕ (in w), we say that the ϕ *is unbound*. We define a function $D_\varepsilon : (\Sigma_\Phi)^* \to \Sigma^*$ by using the following steps to obtain $\mathcal{D}_\varepsilon(w)$ from w. First replace all instances of ϕ by ε if ϕ is unbound. Next replace iteratively, ϕ by u, if $(_\phi u)_\phi$ is the closest subword to the left of ϕ in w starting and ending with '$(_\phi$' and '$)_\phi$' respectively, and $u \in \Sigma^*$. Finally, delete all symbols from $\{(_\phi,)_\phi \mid \phi \in \Phi\}$. The order in which the replacements are made in the second step, has no effect on the final word obtained, and thus we may assume it is made from left to right.

On occasion we are interested in the image of a specific substring in an input string under D_ε, which obviously depends on the prefix to the left of the substring of interest. We denote by $D_{\varepsilon,w}(w')$ the string obtained by removing the prefix $D_\varepsilon(w)$ from $D_\varepsilon(ww')$.

Similarly to D_ε, we let $D_\emptyset : (\Sigma_\Phi)^* \to \Sigma^*$ be the partial function with $D_\emptyset(w)$ undefined if w contains an unbound reference, and being equal to $D_\varepsilon(w)$ otherwise. The partial function $D_{\emptyset,w}(w')$ is defined in the same way as $D_{\varepsilon,w}(w')$.

We denote by $\mathbb{B}(\Sigma_\Phi)$ the subset of $(\Sigma_\Phi)^*$ of strings with well-balanced parenthesis from $\{(_\phi,)_\phi \mid \phi \in \Phi\}$. In our exposition we only use the case where the function and partial function D_ε and $D_\emptyset$ are applied on strings which are prefixes of words in $\mathbb{B}(\Sigma_\Phi)$, and in fact, D_ε and $D_\emptyset$ are mostly applied on strings from $\mathbb{B}(\Sigma_\Phi)$.

For $E \in \text{rewbr}_{\Sigma,\Phi}$, we denote by ref-regex(E) the regular expression $E' \in \mathfrak{R}_{\Sigma_\Phi}$ obtained by replacing $\uparrow_\phi, [_\phi,]_\phi$ with $\phi, (_\phi,)_\phi$ respectively.

Example 2. Let $\Sigma = \{a\}$, $\Phi = \{0, 1\}$, and $E = ([_0 a \uparrow_1]_0 [_1 \uparrow_0 \uparrow_0]_1)^*$. Then $E' = \text{ref-regex}(E) = ((_0 a 1)_0 (_1 00)_1)^*$ and for $w' = (_0 a 1)_0 (_1 00)_1 (_0 a 1)_0 (_1 00)_1 \in \mathcal{L}(E')$ we

have that $D_\varepsilon(w') = a^{12}$, which is obtained by rewriting w' as follows:

$$\begin{aligned} w' &\to ({}_0a)_0({}_10^2)_1({}_0a1)_0({}_10^2)_1 \to ({}_0a)_0({}_1a^2)_1({}_0a1)_0({}_10^2)_1 \\ &\to ({}_0a)_0({}_1a^2)_1({}_0a^3)_0({}_10^2)_1 \to ({}_0a)_0({}_1a^2)_1({}_0a^3)_0({}_1a^6)_1 \\ &\to a^{12} \end{aligned}$$

The partial function $D_\emptyset$ is undefined on all of $\mathcal{L}(E')$, with the exception of ε, since all non-empty strings in $\mathcal{L}(E')$ has $({}_0a1)_0({}_100)_1$ as prefix, in which the 1 in the substring $({}_0a1)_0$ is unbound.

Remark 3. Note that $|D_\varepsilon(w)| \leq 2^{|w|}$ (and similarly for $D_\emptyset(w)$), since each time we substitute a backreference ϕ with a substring w' (where $w' \in \Sigma^*$), we at most double the length of the string. More generally, we have $|D_{\varepsilon,w_1}(w_2)| \leq \max(1, |D_\varepsilon(w_1)|)2^{|w_2|} \leq 2^{|w_1|+|w_2|}$, where the first inequality follows from the fact that in computing $D_{\varepsilon,w_1}(w_2)$ from w_2 by substitution, we may immediately use captured substrings of w_1 (if $w_1 \neq \varepsilon$) for substitution as we process w_2 from left to right.

Definition 4. *For $E \in rewbr_{\Sigma,\Phi}$, we define the language described by E based on ε-*semantics *and $\emptyset$-semantics, and denoted by $\mathcal{L}_\varepsilon(E)$ and $\mathcal{L}_\emptyset(E)$ respectively, as follows:*

- $\mathcal{L}_\varepsilon(E) = \{D_\varepsilon(w) \mid w \in \mathcal{L}(\textit{ref-regex}(E))\}$
- $\mathcal{L}_\emptyset(E) = \{D_\emptyset(w) \mid w \in \mathcal{L}(\textit{ref-regex}(E))$ *and $D_\emptyset(w)$ is defined* $\}$

Definition 5. *We obtain variants of rewbr by using $\mathcal{L}_\varepsilon(r)$ or $\mathcal{L}_\emptyset(r)$ or syntactically restricting the rewbr we consider in $rewbr_{\Sigma,\Phi}$ by not allowing more than one occurrence of $[_\phi$ for each ϕ (i.e. capturing labels may not be repeated) in the rewbr we consider. The four variants are then:*

	No label repetitions	***May repeat labels***
ε-semantics	*Câmpeanu-Salomaa-Yu semi-regex [4]*	*Freydenberger-Schmid [5]*
∅-semantics	*Java, Python*	*Aho [1], Boost, PCRE, .NET*

A fifth variant is the extended (non-semi) regexes of [4], which additionally require that $\uparrow_\phi$ only occur to the right of the occurrence of $]_\phi$ in the rewbr.

When we distinguish between these variants we call them (going left-to-right top-to-bottom) semi-CSY-, FS-, Java-, and Aho-style, with the addition of CSY-style to refer to the full (non-semi) regexes of [4]. We denote by $\mathbb{L}_x$ the class of languages matched by an x-style variant rewbr.

Example 6. The expression $\uparrow_1[_1\Sigma^*]_1$ can be interpreted as a semi-CSY-, FS-, Java-, or Aho-style rewbr, but not a CSY-style one (as $\uparrow_1$ occurs before $]_1$). However, the semi-CSY- and FS-style rewbr described by that expression matches Σ^*, whereas the Java- and Aho-style ones match $\emptyset$, by the difference between ε- and $\emptyset$-semantics set out in Definition 4. Meanwhile the expression $([_1a^*]_1 \,|[_1(b \,|\, c)]_1)\uparrow_1$ can only describe either an FS- or Aho-style rewbr (as it repeats labels), but in this instance they match the same language, as the final $\uparrow_1$ always refers to something bound, eliminating the distinction between D_ε and $D_\emptyset$.

Remark 7. The (CSY- and Java-style) restriction of not allowing repeated labels, leads to unnatural closure properties of the respective classes of rewbr, since if $E = F(G \,|\, H)$ is of CSY- or Java-style, and F contains a capturing group, then in $E' = (FG \,|\, FG)$ a label is repeated, and E' is thus not CSY- or Java-style.

Remark 8. The additional restriction used to obtain CSY-style rewbr can be used in conjunction with the other four variants to obtain eight variants of rewbr in total, but by doing so, we end up with an additional three variants which appear to not have been considered before in literature, nor been used in practical matching software, making them of little interest to us.

Remark 9. In [5], Freydenberger and Schmid disallowed rewbr with subexpressions of the form $[_\phi \cdots \uparrow_\phi \cdots]_\phi$ (i.e. backreferences within a capturing group using the same label), since their memory automaton model, which provides a state machine equivalent formalism for the class of languages equivalent to FS-style rewbr (with this additionally stated constraint), has a memory location for each capture symbol, but it is not possible to update a memory location (of a memory automaton) and use its previous content at the same time. We, however, do not consider this restriction.

Remark 10. Notice that rewbr (independent of the choice of variant from Definition 5) are exponentially more succinct than regular expressions for some languages, for example the family

$$E_n = [_0 a]_0 [_1 \uparrow_0 \uparrow_0]_1 \cdots [_n \uparrow_{n-1} \uparrow_{n-1}]_n$$

has $\mathcal{L}(E_n) = \{a^{2^{n+1}-1}\}$, which is exponential in the length of the expression itself. By contrast, a regular expression is always at least as long as the shortest string it matches.

Next we define a generalization of the syntactic constraint that was used to define the CSY subclass of semi-CSY-style rewbr.

Definition 11. *For $E \in rewbr_{\Sigma,\Phi}$, we define the relation $\sim_E$ on Φ as $\phi \sim_E \phi'$ for $\phi, \phi' \in \Phi$, if E contains a subexpression of the form $[_\phi \cdots \uparrow_{\phi'} \cdots]_\phi$. Let $\approx_E$ be the transitive closure of $\sim_E$. Then E is* non-circular *if it is not the case that $\phi \approx_E \phi$ for any $\phi \in \Phi$.*

In a similar way as in the definition above, we define when strings in $B(\Sigma_\Phi)$ are non-circular, and note that if $w \in \mathcal{L}(\text{ref-regex}(E))$, with E non-circular, then w is also non-circular. Note that the rewbr in Example 2 is circular, while E_n in Remark 10 is non-circular.

Remark 12. The class of CSY-style rewbr has the unnatural closure (or more precisely, non-closure) property that if we start with E of CSY-style and replace in E a subexpression of the form $(F_1|F_2)$ by $(F_2|F_1)$ to obtain E', then E' might no longer be of CSY-style (but of course still semi-CSY-style). This makes it clear that non-circular rewbr (or non-circular semi-CSY) is a more natural subclass of rewbr to consider.

3 The Complexity of Backreference Matching

It is shown already in [1] that matching a rewbr to a string is NP-complete in general. In that proof a reduction from Vertex Cover is performed, with a large alphabet. As usual the alphabet can be reduced to a binary one by straightforward encoding of symbols, but we take one step further and prove that the matching problem for rewbr is NP-complete even for a unary alphabet.

Theorem 13. *Uniform membership testing a rewbr (independent of the choice of semantics from Definition 5) over alphabet Σ is NP-complete even for $|\Sigma| = 1$.*

Proof. We demonstrate this by a reduction from SATISFIABILITY (deciding satisfiability of propositional formulas on conjunctive normal form). For any instance of such a formula, $c_1 \wedge \cdots \wedge c_n$ over the variables $x_1, \ldots, x_m$, first, for each clause c_i construct the rewbr r_i as the union of backreferences for every literal in the disjunction. That is, if $c_i = x_3 \vee \overline{x_7} \vee \overline{x_9}$ (where $\overline{x}$ represents the literal negating the variable x, here viewed as a single symbol) then $r_i = \uparrow_{x_3} | \uparrow_{\overline{x_7}} | \uparrow_{\overline{x_9}}$. Then construct the rewbr:

$$R = ([_{x_1} a]_{x_1} | [_{\overline{x_1}} a]_{\overline{x_1}}) \cdots ([_{x_m} a]_{x_m} | [_{\overline{x_m}} a]_{\overline{x_m}}) r_1 \cdots r_n$$

We then argue that $a^{m+n} \in \mathcal{L}(R)$ if and only if the formula is satisfiable. This is straightforward: clearly at most $m + n$ symbols can be read (as the expression is a concatenation of $m + n$ unions). The initial sequence of unions corresponding to variables will read m symbols, in the process defining a capture *either* x_i *or* $\overline{x_i}$ for each i. The only way to read another n symbols is if every union contains at least one backreference to a literal which was chosen in the first phase. This corresponds precisely to assigning truth values to the variables, and requiring that each disjunction in the original formula contains at least one true literal.

The problem is *in* NP for all alphabet sized by a straightforward search argument over the expression. Starting at the left hand side of the expression nondeterministically search for a path through the expression in the obvious way, consuming symbols from the string as needed. If the right of the expression is reached with the entire string consumed the search accepts. This search can be restricted to polynomially many steps by rejecting whenever it would visit a position in the expression twice without either consuming a symbol or passing through a previously unvisited capturing group (i.e. defining $\uparrow_\phi$ for some ϕ where it was previously undefined) in an intervening step. The latter case is necessary for Java- and Aho-style semantics when matching e.g. $([_1 a^*]_1 | [_2 b^*]_2)^* \uparrow_1 \uparrow_2 c$ to the string c, having to repeat the first Kleene closure twice to get $\uparrow_1$ and $\uparrow_2$ initialized. This easily gives a bound of $|E|^2|w|$ (heavily overestimating), as there are $|E|$ positions, and no more than $|E|$ capturing groups which may get defined. □

Using the above result we can further demonstrate that some of the rewbr semantics we consider also give rise to a difficult emptiness problem.

Theorem 14. *For Java- and Aho-style semantics uniform membership testing and emptiness checking is NP-complete, even for $\Sigma = \emptyset$.*

Proof. For the empty alphabet emptiness checking and membership testing is equivalent, as the only string that can be in the language matched is ε. Use the same reduction that was shown in Theorem 13, but remove all as from the rewbr R. Now $\varepsilon \in \mathcal{L}_\emptyset(R)$ if and only if the formula is satisfiable.

This is easy to see, as Java- and Aho-style rewbr have the $\emptyset$-semantics defined by the $D_\emptyset$ function, which does not permit $\uparrow_{x_i}$ to match anything if it is unbound (i.e. if the capturing group labeled x_i has not been matched to something). Therefore each clause must contain some literal chosen to match in the first part of the expression (despite the captures simply being the empty string), which again simulates assigning truth values to the variables.

Membership is in NP for all alphabet sizes by the argument in Theorem 13. Emptiness is also in NP by a similar search argument, simply ignoring what symbols

are being consumed. As some expressions may contain only long strings (see e.g. Remark 10) a witness string must not be explicitly constructed, but it is sufficient for the search to track which capturing groups have been visited, capturing *some* string, not caring *which*. □

Remark 15. It should be clear that (semi-)CSY-/FS-style semantics have linear-time emptiness-checking (and therefore membership testing with $|\Sigma| = 0$), as an expression is only empty if it is a concatenation with one empty sub-expression, or is a union with both sub-expressions empty, or it equals $\emptyset$. In practical implementations $\emptyset$ is seldom even available, as it has very limited usefulness, making practical emptiness-checking constant time, since then no CSY-/FS-style expression is empty).

It is reasonably obvious, by practical use if nothing else, that the difficulty of rewbr matching is not insurmountable. If used with care, capturing in contexts where the ambiguity is low (i.e. the number of options for capturing is limited) the performance impact can be minimized. Practical regular expression libraries (all mentioned here) often have operators specifically aimed at managing such ambiguity, see for example [3]. A deeper study of the fixed parameter complexity of matching will, however, be left as future work in this paper.

4 Pumping Lemmas with Backreference Matching

The pumping lemma given in [4] is a useful tool for finding languages that cannot be matched by CSY-style rewbr. It is used in the next section to show that $\mathbb{L}_{\text{CSY}} \subsetneq \mathbb{L}_{\text{semi-CSY}}$. First we recall the definition of the pumping lemma, which we will then consider in the context of the additional semantics treated here, to then introduce a more generalized pumping lemma.

Lemma 16 (from [4]). *For every $L \in \mathbb{L}_{CSY}$ (i.e. any language matched by some CSY-style rewbr) there exists a constant k such that if $w \in L$ with $|w| > k$, then there is a decomposition $w = x_0 v x_1 v x_2 \cdots v x_n$, for some $n \geq 1$, such that:*

- $|x_0 v| < k$;
- $|v| \geq 1$; *and,*
- $x_0 v^i x_1 v^i x_2 \cdots v^i x_n \in \mathcal{L}(E)$ *for all* $i \geq 1$.

First we note that this pumping lemma does not apply to most of the other styles considered here. We satisfy ourselves with proving that it does not hold for semi-CSY-style, extending the proof to FS- and Aho-style is straightforward, but Lemma 22 will later on achieve the same result by demonstrating that languages matched by semi-CSY-style forms a subclass of FS- and Aho-style.

Lemma 17. *The pumping lemma of [4] does not hold for semi-CSY-style rewbr.*

Proof. This follows from there being exponentially growing languages matched by semi-CSY-style rewbr. Let $E = ([_\alpha\uparrow_\beta\uparrow_\beta]_\alpha[_\beta a\alpha]_\beta)^*$. Then $L = \mathcal{L}_\varepsilon(E) = \{a^{2^n-1} \mid n \geq 1\}$ and $L \in \mathbb{L}_{\text{semi-CSY}}$. The result now follows by observing that the pumping lemma recalled in Lemma 16 does not hold for L, as it implies there would exist some k, n and v such that $a^{k+i\cdot n|v|} \in L$ for $i \geq 1$, which precludes strict exponential growth. □

However, this pumping lemma does hold for Java-style rewbr.

Lemma 18. *The pumping lemma of [4] also holds for Java-style rewbr.*

Proof. The intuition for why Java-style rewbr differs from semi-CSY-style in this regard is that, while there may be circular capturing groups in Java, the first capture in the cycle must be possible to perform without using any of the other backreferences in the cycle (as they will be unbound). Since the capturing labels cannot repeat, the option of not using any backreference in the capturing sub-expression will then remain on every subsequent repetition of the cycle, making it possible to "restart" the cycle at will. A formal argument follows.

Let E be a Java-style rewbr, set $k = 2^{|E|}$, to match a string w with $|w| > k$ some Kleene closure must be repeated at least once (matching a backreference may at most double the length of the string matched, see e.g. Remark 3, and, obviously, E contains at most linearly many backreferences). Fix one particular match for w, and take the Kleene closure which first repeats a full match of its enclosed subexpression, $x_0 v x_1 v x_2 \cdots v x_n$, where the first v is the substring matched by the the first repetition of the F subgroup, and each following v is produced by a backreference to that initial matching of the subgroup (obviously n may be one if the capture is never referred to). Then we argue that in Java-style semantics $x_0 v^i x_1 v^i x_2 \cdots v^i x_n \in \mathcal{L}(E)$ for all $i \geq 1$.

This is the case as, by assumption, the match of v was the first entry into F, and as unbound backreferences do not match in Java-style, and capture group labels may not repeat, this means the match of v used a path through F on which no backreference is used which is subsequently assigned by a capture inside F (as such a backreference would have had to be undefined). This means that if F is repeated, it will be able to match v again, any number of times, without changing any capturing group contents (performing the same captures as it did the first time), using the same path through F in each instance. The remaining vs, down the line, are produced by backreferences and need no special argument. □

In the next three lemmas we develop a more general pumping lemma for $\mathbb{L}_{\text{CSY}}$.

Lemma 19. *For $L \in \mathbb{L}_{CSY}$ there exists a constant k_L such that if $w_0 w \in L$ with $|w| \geq k_L \max(1, |w_0|)$, then we have strings u, x, y, z such that $uxyz, y \in \mathbb{B}(\Sigma_\Phi)$, with:*

- $(D_\varepsilon(u), D_{\varepsilon,u}(xyz)) = (w_0, w)$ *and thus* $D_\varepsilon(uxyz) = w_0 w$;
- $|D_{\varepsilon,u}(xy)| \leq k_L \max(1, |w_0|)$;
- $D_{\varepsilon,uxy^i}(y) = D_{\varepsilon,ux}(y) \neq \varepsilon$ *for all* $i \geq 0$; *and*
- $D_\varepsilon(uxy^*z) \subseteq L$.

Proof. Assume $L = \mathcal{L}_\varepsilon(E)$, $E' = \text{ref-regex}(E)$, $p > |E'|$ (i.e. p is a pumping constant for the regular language $\mathcal{L}(E')$) and $k_L = 2^p$. The result now follows from the relationship between regular languages and $\mathbb{L}_{\text{CSY}}$ via the function D_ε, Remark 3 and the pumping lemma applied to the regular language $\mathcal{L}(E')$. Next the details. We have u, u' with $(D_\varepsilon(u), D_{\varepsilon,u}(u')) = (w_0, w)$ and $uu' = \mathcal{L}(E')$. From Remark 3, $|u'| \geq p$. The pumping lemma for $\mathcal{L}(E')$ implies we have x, y, z with $u' = xyz$, $uxy^*z \subseteq \mathcal{L}(E') \subseteq \mathbb{B}(\Sigma_\Phi)$ and thus $D_\varepsilon(uxy^*z) \subseteq L$. We may assume y is matched by F, with F^* a subexpression of E' and F not containing Kleene stars. To get $|D_{\emptyset,ux}(y)| \geq 1$, we consider all possible non-empty substrings y of u' matched by an F with F^* a subexpression of E', and if for all of them $|D_{\emptyset,ux}(y)| = 0$, we would get a contradiction to $|w| \geq k_L \max(1, |w_0|)$. By picking the first possible y (to the left) with $|D_{\emptyset,ux}(y)| \geq 1$, we also ensure $|D_{\varepsilon,u}(xy)| \leq k_L \max(1, |w_0|)$, by using Remark 3.

Finally, to obtain $D_{\varepsilon,uxy^i}(y) = D_{\varepsilon,ux}(y)$ for all $i \geq 0$, we need the CSY assumption that backreferences do not appear before corresponding capturing subexpressions

in E, which implies that if $y = y_1\phi y_2$, then y_1 contains a substring of the form $({}_\phi y')_\phi$. This is enough to ensure $D_{\varepsilon,uxy^i}(y) = D_{\varepsilon,ux}(y)$. □

Lemma 20. *Assume x, y, z are strings with $xyz, y \in \mathbb{B}(\Sigma_\Phi)$ and xyz non-circular with $(D_\varepsilon(x), D_{\varepsilon,x}(y)) = (x_0, v)$. Also assume if $y = y_1\phi y_2$, then y_1 contains a substring of the form $({}_\phi y')_\phi$. Then for $i \geq 1$, $D_\varepsilon(xy^iz) = x_0v^ix_1v^ix_2 \cdots v^ix_n$, for some strings $x_1, \ldots, x_n$ where $n \geq 1$.*

Proof. If $xyz = w_0({}_\phi w_1 y w_2)_\phi w_3$, where w_3 has $(n-1)$ occurrences of the symbol ϕ before any substring of the form $({}_\phi w)_\phi$ (which includes the case of w_3 not having a substring of the form $({}_\phi w)_\phi$), then $D_\varepsilon(xy^iz)$ is as specified. Otherwise, if y is not properly contained in a substring of the form $({}_\phi w)_\phi$, we have $n = 1$. □

To see that the non-circular requirement is necessary in Lemma 20, take $x = z = \varepsilon$ and $y = ({}_0a1)_0({}_1 00)_1$. Then $D_\varepsilon(y) = a^3, D_\varepsilon(y^2) = a^{12}$, $D_\varepsilon(y^3) = a^{33}$, and in general, $|D_\varepsilon(y^i)| \geq 3^i$. Also, if $(x, y, z) = (({}_0a)_0, 0({}_0b)_0, \varepsilon)$, then $D_\varepsilon(xyz) = a^2b$, while $D_\varepsilon(xy^{i+1}z)) = a^2bb^{2i}$, and thus the reason for assuming if $y = y_1\phi y_2$, then y_1 contains a substring of the form $({}_\phi y')_\phi$.

Lemma 21. *For $L \in \mathbb{L}_{CSY}$ there is a constant k_L such that if $w_0w \in \mathbb{L}$ with $|w| \geq k_L \max(1, |w_0|)$, then there are strings $x_0, \ldots, x_n, v$, for some $n \geq 1$, with $|v| \geq 1$, so that we have:*

- *$w = x_0vx_1vx_2 \cdots vx_n$; and*
- *$w_0x_0v^ix_1v^ix_2 \cdots v^ix_n \in L$ for all $i \geq 1$.*

Proof. From Lemma 19 we have u, x, y, z with $(D_\varepsilon(u), D_{\varepsilon,u}(xyz)) = (w_0, w)$ (and $D_\varepsilon(uxyz) = ww_0$), $D_{\varepsilon,uxy^i}(y) = v \neq \varepsilon$ for $i \geq 0$, with $D_\varepsilon(uxy^*z) \subseteq L$. $L \in \mathbb{L}_{\text{CSY}}$ implies we can use Lemma 20 and conclude that there is some $n \geq 1$ so that $D_\varepsilon(uxy^iz) = w_0x_0v^ix_1v^ix_2 \cdots v^ix_n \in L$, for $i \geq 1$. □

Beyond its general usefulness, Lemma 21 will be used to distinguish between the language classes matched by CSY- and Java-style rewbr in Lemma 25.

5 Language Hierarchies

In the previous section several containment relationships between the language classes which can be matched by the different styles of rewbr were established, in this section we refine this further. Let us begin by combining and summarizing a few straightforward relations with what was already established in previous sections.

Lemma 22. *The following inclusions hold: $\mathbb{L}_{CSY} \subsetneq \mathbb{L}_{semi\text{-}CSY} \subseteq \mathbb{L}_{FS} \subseteq \mathbb{L}_{Aho}$, and in addition, $\mathbb{L}_{semi\text{-}CSY} \not\subseteq \mathbb{L}_{Java} \subseteq \mathbb{L}_{Aho}$.*

Proof. $\mathbb{L}_{\text{CSY}} \subseteq \mathbb{L}_{\text{semi-CSY}} \subseteq \mathbb{L}_{\text{FS}}$ and $\mathbb{L}_{\text{Java}} \subseteq \mathbb{L}_{\text{Aho}}$ follow directly by Definition 5, by explicit restrictions placed on the styles. $\mathbb{L}_{\text{semi-CSY}} \not\subseteq \mathbb{L}_{\text{CSY}}$ and $\mathbb{L}_{\text{semi-CSY}} \not\subseteq \mathbb{L}_{\text{Java}}$ is shown in Lemma 17. Finally, $\mathbb{L}_{\text{FS}} \subseteq \mathbb{L}_{\text{Aho}}$ since Aho-style can simulate FS-style, i.e. $\mathbb{L}_{\text{FS}} \subseteq \mathbb{L}_{\text{Aho}}$, since if $E \in \text{rewbr}_{\Sigma,\Phi}$ and $E' = [_{\phi_1}]_{\phi_1} \cdots [_{\phi_n}]_{\phi_n} E$, where $\uparrow_{\phi_1}, \ldots, \uparrow_{\phi_n}$ are all of the distinct backreference symbols in E, then $\mathcal{L}_\varepsilon(E) = \mathcal{L}_\emptyset(E')$. □

As a first additional piece of the puzzle we demonstrate that the two most powerful formalisms in the hierarchy are actually equivalent.

Lemma 23. $\mathbb{L}_{FS} = \mathbb{L}_{Aho}$.

Proof. Lemma 22 already demonstrates that $\mathbb{L}_{\text{FS}} \subseteq \mathbb{L}_{\text{Aho}}$, so all that is still needed is to establish that $\mathbb{L}_{\text{Aho}} \subseteq \mathbb{L}_{\text{FS}}$. Let $\mathcal{A}$ be Aho-style rewbr E with the property that if $w \in \mathcal{L}(\text{ref-regex}(E))$, then w has no unbound reference. Thus $\mathcal{L}_\emptyset(E) = \mathcal{L}_\varepsilon(E)$ for $E \in \mathcal{A}$. Let F be a rewbr of Aho-style. We show that there exists $F' \in \mathcal{A}$ with $\mathcal{L}_\emptyset(F) = \mathcal{L}_\emptyset(F') = \mathcal{L}_\varepsilon(F')$ and thus $\mathbb{L}_{\text{Aho}} \subseteq \mathbb{L}_{\text{FS}}$. Let $[_{\phi_1}F_1]_{\phi_1}, \ldots, [_{\phi_k}F_k]_{\phi_k}$ be all capturing subexpressions in F. We replace subexpressions in F as follows:

- subexpressions of the form $\bar{F}^*$ are replaced by $(\varepsilon \mid \bar{F}^+)$; and
- $(\bar{F} \mid \bar{G})^*$ is replaced by $\bar{F}^+\bar{G}(\bar{F} \mid \bar{G})^* \mid \bar{G}^+\bar{F}(\bar{F} \mid \bar{G})^*)$.

After these replacements we use the fact that concatenation distribute over union to obtain $F' = (H_1 \mid \ldots \mid H_l)$, with $\mathcal{L}_\emptyset(F) = \mathcal{L}_\emptyset(F')$. Each H_i is such that if $\uparrow_{\phi_k}$ is a subexpression of H_i, then H_i is a concatenation of subexpressions, one of which is $H' = [_{\phi_k}F_k]_{\phi_k}$, and this subexpression H' appears in the concatenation of subexpressions forming H_i, before the subexpression containing $\uparrow_\phi$. Thus during a match with H_i, a subexpression of the form $[_{\phi_k}F_k]_{\phi_k}$ must be used to match a substring of the input string, before encountering $\uparrow_{\phi_k}$. This property of the H_i's ensures that $F' \in \mathcal{A}$. □

Example 24. Here we illustrate part of the construction used in the proof of Lemma 23 to turn an Aho-style rewbr into a language equivalent FS-style rewbr. Let Π_n be the set of $n!$ permutations on $\{1, \ldots, n\}$, and $P_n = \{a_{\pi(1)} \cdots a_{\pi(n)} \mid \pi \in \Pi_n\}$. Then $\mathcal{L}_\emptyset(E_n) = P_n$, where E_n is the Aho-style (in fact Java-style) rewbr given by:

$$(a_1({}_1\varepsilon)_1 \mid \ldots \mid a_n({}_n\varepsilon)_n)^n\uparrow_{\phi_1} \ldots \uparrow_{\phi_n}$$

Since E_n contains no Kleene star operators, when we use the procedure described in the proof of the previous theorem, we simply have to distribute concatenation over union to obtain an FS-style (more precisely, semi-CSY-style) rewbr F_n, with $\mathcal{L}_\emptyset(E_n) = \mathcal{L}_\varepsilon(F_n)$. When we do this, we obtain that F_n is the union of $n!$ subexpressions of the following form, for all $\pi \in \Pi_n$:

$$a_{\pi(1)}({}_{\pi(1)}\varepsilon)_{\pi(1)} \cdots a_{\pi(n)}({}_{\pi(1)}\varepsilon)_{\pi(n)}\uparrow_{\phi_1} \cdots \uparrow_{\phi_n}$$

Note that when distributing concatenation over union, we get many more subexpressions which are all of form $b_1 \cdots b_n\uparrow_{\phi_1} \cdots \uparrow_{\phi_n}$, with $b_i \in \{a_1, \ldots, a_n\}$, for $1 \leq i \leq n$. But when some of the b_i's are equal (i.e when we have backreferences to non-existing capturing groups), the languages represented by these subexpressions are empty in Aho-style, and they are thus not used in F_n. The semi-CSY-style rewbr F_n is of course more complicated than necessary to describe P_n, but it remains open if a more succinct rewbr of semi-CSY-style or even FS-style exists for the language P_n, than simply taking the union of all $n!$ subexpressions of the form $a_{\pi(1)} \cdots a_{\pi(n)}$.

Further, while Java does fulfill the original pumping lemma recalled in Lemma 16, it does in fact not fulfill Lemma 21, the generalized pumping lemma for $\mathbb{L}_{\text{CSY}}$.

Lemma 25. $\mathbb{L}_{Java} \not\subseteq \mathbb{L}_{CSY}$.

Proof. Take the Java-style rewbr $E = ([_0a \mid \uparrow_0 a]_0 b)^*$, for which we have $\mathcal{L}(E) = (\{(ab)^i \mid i \geq 0\} \cup \{a^{2^0}b \cdots a^{2^i}b \mid i \geq 0\})^*$. We argue that Lemma 21 does not hold for $\mathcal{L}(E)$, by assuming the contrary and taking $w_0 = ab$ and $w = a^{2^1}b \cdots a^{2^j}b$, with $|w| \geq 2k_{\mathcal{L}(E)}$. Let u be the non-empty pumping substring. Clearly any u consisting

of only as does not work, but choosing a string containing a b will under pumping also give rise to a substring of the form $\cdots a^l b a^l b \cdots$, which is not in $\mathcal{L}(E)$ for any l except for $l = 1$, however, that corresponds to $u = ab$, and the only place where that substring could be pumped in this particular string would be in the initial prefix, but as that prefix is taken by w_0 that is not available as a choice. We thus conclude that $\mathbb{L}_{\text{Java}} \not\subseteq \mathbb{L}_{\text{CSY}}$. □

We are with these results in hand ready to summarize the containment results for the classes of languages matched by the various variants of rewbr.

Theorem 26. *The following inclusions hold.*

$$\mathbb{L}_{CSY} \subsetneq \mathbb{L}_{semi\text{-}CSY} \subsetneq \mathbb{L}_{FS} = \mathbb{L}_{Aho}$$
$$\mathbb{L}_{CSY} \not\supseteq \mathbb{L}_{Java} \subsetneq \mathbb{L}_{FS}$$

Proof. Combine Lemmas 22 (in turn using Lemma 17), 23, and 25.

6 Conclusions and Future Work

These initial definitions, pumping lemmas, and inclusion proofs create a solid foundation, there are still numerous avenues for further investigation available:

- The inclusions of Theorem 26 paint a fairly clear picture, but it does remain to show whether the non-inclusions are one side of the classes being incomparable, or, seemingly more likely, whether they can be expanded into containments. That is, it seems a reasonable conjecture that a completed result should read

  $$\mathbb{L}_{\text{CSY}} \subsetneq \mathbb{L}_{\text{Java}} \subsetneq \mathbb{L}_{\text{semi-CSY}} \subsetneq \mathbb{L}_{\text{FS}} = \mathbb{L}_{\text{Aho}},$$

  but the actual inclusions of $\mathbb{L}_{\text{CSY}}$ in $\mathbb{L}_{\text{Java}}$ and $\mathbb{L}_{\text{Java}}$ in $\mathbb{L}_{\text{semi-CSY}}$ remain to be demonstrated.
- Pumping lemmas which generalize to semi-CSY-, FS- and Aho-style rewbr should also be found, it may be that Lemma 19 can be adapted to these cases with some minor restatements and additional argument.
- While differences between the language classes matched seems the most important point from a theoretical perspective, it may for practical purposes be almost more important to determine the relative succinctness of the rewbr variants. The exponential growth exhibited by some of the classes, demonstrating the differences, is likely not languages of very great interest for practical matching. However, how compactly some of the languages within the intersection of the language classes can be described could inform choices for future implementations (e.g. if Aho-style is not more succinct on interesting cases it may be inadvisable to accept the additional power offered by the variant).
- Finally, the most important practical questions is no doubt matching time complexity. While we give a refinement on the hardness of matching with rewbr in Section 3 there is much more that can be done attacking this problem. Applying parameterized complexity theory to study which aspects of the problem cause the seeming high complexity seems a promising avenue, as practical use suggests that suitably limited use of backreferences make for matching performance which is in fact entirely tractable.

More broadly the area of practical regular expressions remains teeming with poorly understood extensions and common use cases which require study to form a solid theoretical foundation for practical string matching.

References

1. A. V. Aho: in Handbook of Theoretical Computer Science (Vol. A), J. van Leeuwen, ed., MIT Press, Cambridge, MA, USA, 1990, ch. Algorithms for Finding Patterns in Strings, pp. 255–300.
2. M. Berglund, F. Drewes, and B. van der Merwe: *Analyzing catastrophic backtracking behavior in practical regular expression matching*, in Automata and Formal Languages, Z. Ésik and Z. Fülöp, eds., vol. 151 of Electronic Proceedings in Theoretical Computer Science, 2014, pp. 109–123.
3. M. Berglund, B. van der Merwe, B. Watson, and N. Weideman: *On the semantics of atomic subgroups in practical regular expressions*, in Implementation and Application of Automata, A. Carayol and C. Nicaud, eds., vol. 10329 of Lecture Notes in Computer Science, Springer, 2017, pp. 14–26.
4. C. Câmpeanu, K. Salomaa, and S. Yu: *A formal study of practical regular expressions.* International Journal of Foundations of Computer Science, 14(6) 2003, pp. 1007–1018.
5. D. D. Freydenberger and M. L. Schmid: *Deterministic regular expressions with back-references*, in 34th Symposium on Theoretical Aspects of Computer Science (STACS 2017), H. Vollmer and B. Vallée, eds., vol. 66 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2017, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 33:1–33:14.
6. M. L. Schmid: *Characterising REGEX languages by regular languages equipped with factor-referencing*, in Developments in Language Theory - 18th International Conference, DLT 2014, Ekaterinburg, Russia, August 26-29, 2014. Proceedings, A. M. Shur and M. V. Volkov, eds., vol. 8633 of Lecture Notes in Computer Science, Springer, 2014, pp. 142–153.
7. N. Weideman, B. van der Merwe, M. Berglund, and B. Watson: *Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA*, in Implementation and Application of Automata, Y. Han and K. Salomaa, eds., vol. 9705 of Lecture Notes in Computer Science, Springer, 2016, pp. 322–334.

Speeding Up String Matching by Weak Factor Recognition

Domenico Cantone[1], Simone Faro[1], and Arianna Pavone[2]

[1] Università di Catania, Viale A. Doria 6, 95125 Catania, Italy
[2] Università di Messina, Via Concezione 6, 98122 Messina, Italy

Abstract. String matching is the problem of finding all the substrings of a text which match a given pattern. It is one of the most investigated problems in computer science, mainly due to its very diverse applications in several fields. Recently, much research in the string matching field has focused on the efficiency and flexibility of the searching procedure and quite effective techniques have been proposed for speeding up the existing solutions. In this context, algorithms based on factors recognition are among the best solutions.
In this paper, we present a simple and very efficient algorithm for string matching based on a weak factor recognition and hashing. Our algorithm has a quadratic worst-case running time. However, despite its quadratic complexity, experimental results show that our algorithm obtains in most cases the best running times when compared, under various conditions, against the most effective algorithms present in literature. In the case of small alphabets and long patterns, the gain in running times reaches 28%. This makes our proposed algorithm one of the most flexible solutions in practical cases.

Keywords: string matching, text processing, design and analysis of algorithms, experimental evaluation

1 Introduction

The *exact string matching problem* is one of the most studied problem in computer science. It consists in finding all the (possibly overlapping) occurrences of an input pattern x in a text y, over a given alphabet Σ of size σ. A huge number of solutions has been devised since the 1980s [6,16] and, despite such a wide literature, much work has been produced in the last few years, indicating that the need for efficient solutions to this problem is still high.

Solutions to the exact string matching problem can be divided in two classes: *counting* solutions simply return the number of occurrences of the pattern in the text, whereas *reporting* solutions provide also the exact positions at which the pattern occurs. Solutions in the first class are in general faster than the ones in the second class. In this paper we are interested in algorithms belonging to the class of reporting solutions.

From a theoretical point of view, the exact string matching problem has been studied extensively. If we denote by m and n the lengths of the pattern and of the text, respectively, the problem can be solved in $\mathcal{O}(n)$ worst-case time complexity [18]. However, in many practical cases it is possible to avoid reading all the characters of the text, thus achieving sublinear performances on the average. The optimal average $\mathcal{O}(\frac{n \log_\sigma m}{m})$ time complexity [22] has been reached for the first time by the Backward DAWG Matching algorithm [7] (BDM). However, all algorithms with a sublinear average behaviour may have to possibly read all the text characters in the worst

Domenico Cantone, Simone Faro, Arianna Pavone: Speeding Up String Matching by Weak Factor Recognition, pp. 42–50.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

case. It is interesting to note that many of those algorithms have an $\mathcal{O}(nm)$-time complexity in the worst-case. Interested readers can refer to [6,13,16] for a detailed survey of the most efficient solutions to the problem.

The BDM algorithm computes the Directed Acyclic Word Graph (DAWG) of the reverse x^{R} of the pattern x. Such graph is an automaton which recognizes all and only the factors of x^{R}, and can be computed in $\mathcal{O}(m)$ time. During the searching phase, the BDM algorithm moves a window of size m on the text. For each new position of the window, the automaton of x^{R} is used to search for a factor of x from the right to the left of the window. The basic idea of the BDM algorithm is that when the backward search fails on a letter c after reading a word u, then cu can not be a factor of p, so that moving the window just after c is safe. In addition, the algorithm maintains the length of the last recognized suffix of x^{R}, which is a prefix of the pattern. If a suffix of length m is recognized, then an occurrence of the pattern is reported.

We say that the DAWG of a string performs an *exact factor recognition* since the accepted language coincides exactly with the set of the factors of the string. On the other hand, we say that a structure performs a *weak factor recognition* when it is able to recognize *at least* all the factors of the string, but maybe something more. For instance, the Factor Oracle [1] of a string x performs a weak factor recognition of the factors of x. It is an automaton which recognizes all the factors of x acting like an oracle: if a string is accepted by the automaton, it *may be* a factor of x. However, all the factors of x are accepted. Due to its relaxed recognition approach, the Factor Oracle can be constructed and handled using less resources than the DAWG, both in terms of space and time.

The Backward Oracle Matching algorithm [1] (BOM) works in the same way as the BDM algorithm, but makes use of the Factor Oracle of the reverse pattern, in place of the DAWG. In practical cases, the resulting algorithm performs better than the BDM algorithm [16].

Both BDM and BOM algorithms have been recently improved in various way. For instance, very fast BDM-like algorithms based on the bit-parallel simulation of the nondeterministic factor automaton [2] have been presented in [20], whereas efficient extensions of the BOM algorithm appeared in [11].

In this paper we present a new fast string matching algorithm based on a(n) (even more) weak factor recognition approach. Our solution uses a hash function to recognize all the factors of the input pattern. Such method leads to a simple and very fast recognition mechanism and makes the algorithm very effective in practical cases. In Section 2, we introduce and analyze our proposed algorithm, whereas in Section 3 we compare experimentally its performance against the most effective solutions present in the literature. Finally, we draw our conclusions in Section 4.

2 An Efficient Weak-Factor-Recognition Approach

In this section we present an efficient algorithm for the exact string matching problem based on a weak-factor-recognition approach with hashing. Though the resulting algorithm has a quadratic worst-case time complexity, on average it shows a sublinear behaviour.

Let x be a pattern of length m and y a text of length n. In addition, let us assume that both strings x and y are drawn from a common alphabet Σ of size σ. Our proposed algorithm, named *Weak Factor Recognition* (WFR) is able to count

and report all the occurrences of x in y. It consists in a preprocessing and a searching phase. These are described in detail in the following sections.

2.1 The Preprocessing Phase

During the preprocessing phase, all subsequences of the pattern x are indexed to facilitate their search during the searching phase. Specifically, we define a hash function $h : \Sigma^* \to \{0 \,..\, 2^\alpha - 1\}$, which associates an integer value $0 \leq v < 2^\alpha$ (for a given bound α)[1] with any string over the alphabet Σ. Here, we shall make the assumption that each character $c \in \Sigma$ can be handled as an integer value, so that arithmetic operations can be performed on characters. For instance, in many practical applications, input strings can be handled as sequences of ASCII characters. Thus each character can be seen as an 8-bit value corresponding to its ASCII code.

For each string $x \in \Sigma^*$ of length $m \geq 0$, the value of $h(x)$ is recursively defined as follows

$$h(x) := \begin{cases} 0 & \text{if } m = 0 \\ (h(x[1 \,..\, m-1]) \times 2 + x[0]) \mod 2^\alpha & \text{otherwise.} \end{cases}$$

Observe that, for each string $x \in \Sigma^*$, we have $0 \leq h(x) < 2^\alpha$.

The preprocessing phase of our algorithm, which is reported in Fig. 1 (on the left), consists in computing the hash values of all possible substrings of the pattern x.

A bit vector F of size 2^α is maintained for storing the hash values corresponding to the factors of x. Thus, if z is a factor of x, then the bit at position $h(z)$ in F is set (i.e., $F[h(z)] := 1$), otherwise it is set to 0. More formally, for each value v in the bit vector, with $0 \leq v < 2^\alpha$, we have

$$F[v] := \begin{cases} 1 & \text{if } h(x[i \,..\, j]) = v, \text{ for some } 0 \leq i \leq j < m \\ 0 & \text{otherwise.} \end{cases}$$

Given two strings $x, z \in \Sigma^*$, it is easy to prove that if z is a factor of x then $F[h(z)] = 1$; on the other hand, when $F[h(z)] = 1$, in general we can not conclude that z is a factor of x.

Let w be the number of bits in a computer word of the target machine. Then the bit vector F can be implemented as a table of $2^\alpha/w$ words.[2] The procedure SetBit(F, i) and the function TestBit(F, i) (both reported in Fig. 1, on the left) are used to quickly set and query, respectively, the bit at position i in the vector F. Such procedures are very fast and can be executed in constant time.

Since the set of all nonempty factors of a string x of length m has size m^2, the preprocessing phase of the algorithm requires $\mathcal{O}(2^\alpha)$ space and $\mathcal{O}(m^2)$ time.

2.2 The Searching Phase

As in the BDM and BOM algorithms, during the searching phase a window of size m is opened on the text, starting at position 0. After each attempt, the window is shifted to the right until the end of the text is reached. During an attempt at a given position

[1] In our setting, the value α has been fixed to 16, so that each hash value fits into a single 16-bit register.

[2] In our setting, we have $w = 8$ and F has been implemented as a table of $8,192$ chars, corresponding to a bit-vector of 65,536 bits.

```
SetBit(F, v)
1. p ← ⌊v/w⌋
2. b ← v mod w
3. F[p] ← F[p] or (1 ≪ b)

TestBit(F, v)
1. p ← ⌊v/w⌋
2. b ← v mod w
3. return (F[p] and (1 ≪ b)) ≠ 0

Preprocessing(x, m)
1. for v ← 0 to 2^α − 1 do
2.    F[v] ← 0
3. for i ← m − 1 downto 0 do
4.    v ← 0
5.    for j ← i downto 0 do
6.       v ← (v ≪ 2) + x[j]
7.       SetBit(F, v)
8. return F

Check(x, m, y, i)
1. k ← 0
2. while (k < m and x[k] = y[i + k]) do
3.    k ← k + 1
4. if k = m then return true
5. return false

Wfr(x, m, y, n,)
1. F ←Preprocessing(x, m)
2. j ← m − 1
3. while (j < n) do
4.    v ← y[j]
5.    i ← j − m + 1
6.    while (j > i and TestBit(F, v)) do
7.       j ← j − 1
8.       v ← (v ≪ 2) + y[j]
9.    if (j = i and TestBit(F, v)) then
10.      if Check(x, m, y, i) then return i
11.   j ← j + m
```

Figure 1. The pseudo-code of the Wfr algorithm and of some auxiliary procedures.

i of the text, the current window is opened on the substring $y[i \,..\, j]$ of the text, with $j = i + m - 1$. Our algorithm starts computing the hash value $h(y[j])$ corresponding to the rightmost character of the window. If the corresponding bit in F is set, then such substring may be a factor of x. In this case, the algorithm computes the hash value of the subsequent substring, namely, $h(y[j-1 \,..\, j])$.

More precisely, the hash value $y[j-k \,..\, j]$ of the suffixes of the window is computed for increasing values of k, until k reaches the value m or until the corresponding bit in F is not set.

Observe that by using the following relation

$$h(y[j-k \,..\, j]) = \Big((h(y[j-k+1 \,..\, j]) \ll 1) + y[j-k] \Big) \mod 2^{\alpha} ,$$

the hash value of the suffix $y[j-k \,..\, j]$ can be computed in constant time in terms of $h(y[j-k+1 \,..\, j])$.

When an attempt ends up with $k = m$, a naive check is performed in order to verify whether the substring $y[i \,..\, j]$ matches the pattern (see procedure Check shown in Fig. 1). Such verification can obviously be performed in $\mathcal{O}(m)$ time. In this case, the shift advancement is of a single character to the right.

Table 1 shows the average number of occurrences (α value) versus the average number of verifications (β value) for every 1024 Kb. Values have been computed during the searching phase in our experimental tests described in Section 3. Notice that the number of exceeding verifications is negligible and, in most cases, equal to 0.

The pseudo-code provided in Fig. 1 (on the right) reports the skeleton of the algorithm. If a naive check were performed after each attempt of the algorithm, then a shift of one position would be performed at each iteration. This leads to a $\mathcal{O}(nm)$ worst-case time complexity. However, the experimental results reported in Section 3 show that, in practical cases, the Wfr algorithm has a sublinear behaviour.

m	4	8	16	32	64	128	256	512	1024
Genome-α	4068,40	23,20	0,20	0,20	0,20	0,20	0,20	0,20	0,20
Genome-β	4068,40	24,40	0,20	0,20	0,20	0,20	0,20	0,20	0,20
Protein-α	17,00	0,20	0,20	0,20	0,20	0,20	0,20	0,20	0,20
Protein-β	21,40	0,20	0,20	0,20	0,20	0,20	0,20	0,20	0,20
English-α	1275,80	28,60	2,00	0,40	0,20	0,20	0,20	0,20	0,20
English-β	1280,40	28,80	2,20	0,40	0,20	0,20	0,20	0,20	0,20

Table 1. The average number of occurrences (α value) versus the average number of verifications (β value) for every 1024 Kb. Values have been computed in the searching phase of the experimental tests described in Section 3.

2.3 Some Improvements

Practical improvements of the WFR algorithm can be obtained by means of a *chained-loop* on the characters of the pattern in the implementation of the searching phase. Such a technique consists in dropping the call to TESTBIT in the while-loop at line 6, while computing the hash value. The test is performed only every k cycles, for a fixed value of k. This leads to a fast computation of the hash values even if the corresponding shifts are shorter on average.

For instance, if k is set to 2, then lines 4, 7, and 8 of the WFR algorithm are implemented in the following way:

4. $v \leftarrow (y[j] \ll 1) + y[j-1]$
...
7. $j \leftarrow j - 2$
8. $v \leftarrow (v \ll 4) + (y[j] \ll 2) + y[j-1]$

The resulting algorithm maintains the same space and time complexity, but in practice it shows a sensible increase of its performance, as shown in the next section.

3 Experimental Results

We report the experimental results of the performance evaluation of the WFR algorithm and its variants with a k-chained-loop against the most efficient solutions present in literature for the online exact string matching problem. Specifically, the following 15 algorithms (implemented in 79 variants, depending on the values of their parameters) have been compared:

- AOSOq: the Average-Optimal variant [17] of the Shift-Or algorithm [2] using q-grams, with $1 \leq q \leq 6$;
- BNDMq: the Backward-Nondeterministic-DAWG-Matching algorithm [20] implemented using q-grams, with $1 \leq q \leq 8$;
- BSDMq: the Backward-SNR-DAWG-Matching algorithm [14] using condensed alphabets with groups of q characters, with $1 \leq q \leq 8$;
- BXSq: the Backward-Nondeterministic-DAWG-Matching algorithm [20] with Extended Shift [8] implemented using q-grams, with $1 \leq q \leq 8$;
- EBOM: the extended version [11] of the BOM algorithm [1];
- FSBNDMqs: the Forward Simplified version [21,11] of the BNDM algorithm [20] implemented using q-grams s-forward characters (with $1 \leq q \leq 8$ and $1 \leq s \leq 6$);

- KBNDM: the Factorized variant [5] BNDM algorithm [20];
- SBNDMq: the Simplified version of the Backward-Nondeterministic-DAWG-Matching algorithm [1] implemented using q-grams, with $1 \leq q \leq 8$;
- FS-w: the Multiple Windows version [15] of the Fast Search algorithm [3] implemented using w sliding windows, with $2 \leq w \leq 6$;
- HASHq: the Hashing algorithm [19] using q-grams, with $3 \leq q \leq 5$;
- IOM: the Improved Occurrence Matcher [4]
- WOM: the Worst Occurrence Matcher [4];
- JOM: the Jumping Occurrence Matcher [4];
- WFR: the new Weak Factors Recognition algorithm;
- WFRq: the new Weak Factors Recognition variants implemented with a k-chained-loop (with $2 \leq k \leq 4$);

For the sake of completeness, we evaluated also the following two string matching algorithms for *counting* occurrences (however, we did not take them into account in our comparison since they simply count the number of matching occurrences):

- EPSM: the Exact Packed String Matching algorithm [10];
- TSOq: the Two-Way variant of [9] the Shift-Or algorithm [2] implemented with a loop unrolling of q characters, with $q = 5$;

All algorithms have been implemented in the C programming language and have been tested using the Smart tool [12].[3] All experiments have been executed locally on a MacBook Pro with 4 Cores, a 2 GHz Intel Core i7 processor, 16 GB RAM 1600 MHz DDR3, 256 KB of L2 Cache and 6 MB of Cache L3. All algorithms have been compared in terms of their running times, including any preprocessing time.

We report experimental evaluations on three real data sets (see Tables 2, 3, and 4). Specifically, we used a genome sequence, a protein sequence, and an English text. All sequences have a length of 5 MB; they are provided by theSmart research tool and are available online for download.

In the experimental evaluation, patterns of length m were randomly extracted from the sequences, with m ranging over the set of values $\{2^i \mid 2 \leq i \leq 10\}$. For each case, the mean over the running times (expressed in hundredths of seconds) of 500 runs has been reported.

The following tables summarize the running times of our evaluations. Each table is divided into four blocks. The first and the second block present the most effective algorithms known in literature based on automata and comparison of characters, respectively. The best results among these two sets of algorithms have been bold-faced in order to easily locate them. The third block contains the running times of our newly proposed algorithm and its variant, including the speed up (in percentage) obtained against the best running time in the first two blocks. Positive values indicate a breaking of the running time whereas a negative percentage represent a performance improvement. Running times which represent an improvement of the performance have been bold-faced.

The last block reports the running times obtained by the best two algorithms for *counting* occurrences (however, as already remarked, these have not been included in our comparison).

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

m	4	8	16	32	64	128	256	512	1024
AOSOq	16.98$^{(2)}$	9.63$^{(2)}$	3.93$^{(4)}$	3.39$^{(4)}$	2.98$^{(6)}$	2.97$^{(6)}$	2.99$^{(6)}$	3.00$^{(6)}$	3.03$^{(6)}$
BNDMq	11.13$^{(4)}$	4.10$^{(4)}$	2.99$^{(4)}$	2.47$^{(4)}$	2.38$^{(4)}$	2.39$^{(4)}$	2.41$^{(4)}$	2.47$^{(4)}$	2.45$^{(4)}$
BSDMq	8.37$^{(4)}$	**3.71**$^{(4)}$	**2.78**$^{(4)}$	**2.46**$^{(4)}$	**2.25**$^{(8)}$	**2.15**$^{(8)}$	**2.11**$^{(8)}$	**2.16**$^{(6)}$	**2.11**$^{(6)}$
BXSq	11.86$^{(2)}$	4.78$^{(4)}$	3.25$^{(4)}$	2.53$^{(6)}$	2.50$^{(6)}$	2.52$^{(4)}$	2.49$^{(4)}$	2.55$^{(4)}$	2.54$^{(4)}$
EBOM	7.72	7.15	5.66	4.10	3.17	2.67	2.40	2.32	2.41
FSBNDMqs	**6.46**$^{(3,1)}$	3.87$^{(4,1)}$	2.94$^{(4,1)}$	2.38$^{(4,1)}$	2.35$^{(6,2)}$	2.31$^{(6,1)}$	2.33$^{(6,1)}$	2.38$^{(3,1)}$	2.37$^{(6,1)}$
KBNDM	10.88	8.21	6.15	4.17	3.27	3.09	3.10	3.13	3.14
SBNDMq	8.75$^{(2)}$	3.95$^{(4)}$	2.97$^{(4)}$	2.47$^{(4)}$	2.39$^{(4)}$	2.39$^{(4)}$	2.36$^{(4)}$	2.38$^{(4)}$	2.38$^{(4)}$
FS-w	12.33$^{(2)}$	9.39$^{(2)}$	7.76$^{(2)}$	6.89$^{(2)}$	6.16$^{(2)}$	5.63$^{(2)}$	5.06$^{(2)}$	4.73$^{(2)}$	4.42$^{(2)}$
FJS	18.60	16.69	16.96	15.96	16.09	16.80	16.71	16.61	16.59
HASHq	18.09$^{(3)}$	7.68$^{(3)}$	4.67$^{(5)}$	3.31$^{(5)}$	2.78$^{(5)}$	2.60$^{(5)}$	2.63$^{(5)}$	2.51$^{(5)}$	2.40$^{(5)}$
IOM	14.41	11.88	11.08	11.17	11.17	11.13	11.03	11.03	10.98
WOM	16.69	12.48	9.88	8.61	7.75	7.16	6.72	6.29	6.11
WFR	13.85	8.77	5.70	3.73	2.69	2.28	1.98	1.72	1.57
WFRq	8.67$^{(2)}$	4.42$^{(4)}$	2.98$^{(4)}$	**2.36**$^{(4)}$	**2.08**$^{(4)}$	**1.97**$^{(4)}$	**1.86**$^{(4)}$	**1.62**$^{(4)}$	**1.52**$^{(4)}$
speed-up	+34%	+19%	+7.1%	-4.0%	-7.5%	-8.3%	-11%	-25%	-28%
EPSM	5.87	3.72	2.50	1.93	1.75	1.72	1.66	1.62	1.65
TSOq	5.54$^{(5)}$	3.85$^{(5)}$	3.08$^{(5)}$	2.42$^{(5)}$	2.05$^{(5)}$	-	-	-	-

Table 2. Experimental results on a genome sequence.

m	4	8	16	32	64	128	256	512	1024
AOSOq	10.80$^{(2)}$	4.27$^{(4)}$	3.84$^{(4)}$	3.81$^{(4)}$	3.18$^{(4)}$	3.17$^{(4)}$	3.16$^{(4)}$	3.16$^{(4)}$	3.16$^{(4)}$
BNDMq	12.20$^{(4)}$	4.29$^{(4)}$	3.06$^{(4)}$	2.46$^{(4)}$	2.45$^{(4)}$	2.43$^{(4)}$	2.42$^{(4)}$	2.40$^{(4)}$	2.40$^{(4)}$
BSDMq	4.68$^{(2)}$	3.71$^{(2)}$	2.75$^{(4)}$	2.35$^{(4)}$	**2.06**$^{(4)}$	**1.98**$^{(4)}$	**1.97**$^{(4)}$	**1.97**$^{(4)}$	**1.94**$^{(4)}$
BXSq	6.91$^{(2)}$	4.29$^{(2)}$	3.12$^{(2)}$	2.52$^{(2)}$	2.48$^{(2)}$	2.52$^{(2)}$	2.50$^{(2)}$	2.51$^{(2)}$	2.52$^{(2)}$
EBOM	**3.87**	**2.94**	**2.57**	**2.29**	2.11	2.18	2.20	2.24	2.42
FSBNDMqs	4.32$^{(2,0)}$	3.28$^{(2,0)}$	2.59$^{(3,1)}$	2.26$^{(3,1)}$	2.22$^{(3,1)}$	2.25$^{(3,1)}$	2.25$^{(3,1)}$	2.20$^{(3,1)}$	2.26$^{(3,1)}$
KBNDM	7.46	4.97	3.81	3.24	3.04	3.01	2.95	2.96	2.95
SBNDMq	5.25$^{(2)}$	3.67$^{(2)}$	2.79$^{(2)}$	2.34$^{(2)}$	2.45$^{(4)}$	2.41$^{(4)}$	2.42$^{(4)}$	2.41$^{(4)}$	2.40$^{(4)}$
FS-w	6.18$^{(2)}$	4.33$^{(2)}$	3.55$^{(2)}$	3.20$^{(2)}$	3.05$^{(2)}$	2.94$^{(2)}$	2.90$^{(2)}$	2.87$^{(2)}$	2.86$^{(2)}$
FJS	9.68	18.54	4.18	3.02	2.92	2.89	2.82	3.16	4.11
HASHq	19.92$^{(3)}$	8.36$^{(3)}$	5.05$^{(3)}$	3.75$^{(5)}$	3.19$^{(5)}$	2.99$^{(5)}$	2.92$^{(5)}$	2.76$^{(5)}$	2.66$^{(5)}$
IOM	8.87	6.36	5.02	4.41	4.04	3.92	3.86	3.86	3.79
WOM	9.31	6.61	5.13	4.32	4.03	3.72	3.56	3.43	3.33
WFR	6.79	5.80	4.43	3.21	2.65	2.38	2.12	1.87	1.70
WFRq	4.85$^{(2)}$	3.69$^{(2)}$	2.98$^{(4)}$	2.36$^{(4)}$	**2.03**$^{(4)}$	**1.93**$^{(4)}$	**1.89**$^{(4)}$	**1.75**$^{(4)}$	**1.66**$^{(4)}$
speed-up	+25%	+25%	+15%	+3.0%	-1.4%	-2.5%	-4.0%	-11%	-14%
EPSM	6.67	5.55	2.77	2.16	1.91	1.91	1.90	1.83	1.86
TSOq	5.41$^{(5)}$	3.90$^{(5)}$	3.29$^{(5)}$	2.59$^{(5)}$	2.17$^{(5)}$	-	-	-	-

Table 3. Experimental results on a protein sequence.

m	4	8	16	32	64	128	256	512	1024
AOSOq	$11.14^{(2)}$	$4.58^{(4)}$	$3.89^{(4)}$	$3.76^{(4)}$	$3.16^{(6)}$	$3.16^{(6)}$	$3.18^{(6)}$	$3.21^{(6)}$	$3.16^{(6)}$
BNDMq	$12.30^{(4)}$	$4.35^{(4)}$	$3.17^{(4)}$	$2.49^{(4)}$	$2.53^{(4)}$	$2.52^{(4)}$	$2.51^{(4)}$	$2.54^{(4)}$	$2.51^{(4)}$
BSDMq	$4.73^{(2)}$	$3.85^{(2)}$	$\mathbf{2.86}^{(4)}$	$\mathbf{2.35}^{(4)}$	$\mathbf{2.20}^{(4)}$	$\mathbf{2.09}^{(4)}$	$\mathbf{2.07}^{(4)}$	$\mathbf{2.02}^{(4)}$	$\mathbf{2.00}^{(4)}$
BXSq	$7.38^{(2)}$	$4.85^{(2)}$	$3.43^{(4)}$	$2.59^{(4)}$	$2.59^{(4)}$	$2.64^{(4)}$	$2.62^{(4)}$	$2.62^{(4)}$	$2.63^{(4)}$
EBOM	**4.33**	**3.47**	3.05	2.74	2.54	2.51	2.40	2.40	2.57
FSBNDMqs	$4.66^{(2,0)}$	$3.55^{(3,1)}$	$2.77^{(3,1)}$	$2.39^{(3,1)}$	$2.39^{(3,1)}$	$2.38^{(3,1)}$	$2.41^{(3,1)}$	$2.42^{(3,1)}$	$2.43^{(3,1)}$
KBNDM	7.84	5.49	4.22	3.59	3.28	3.08	3.04	3.03	3.03
SBNDMq	$5.75^{(2)}$	$4.18^{(2)}$	$3.13^{(4)}$	$2.43^{(4)}$	$2.52^{(4)}$	$2.50^{(4)}$	$2.52^{(4)}$	$2.51^{(4)}$	$2.52^{(4)}$
FS-w	$6.05^{(6)}$	$4.25^{(6)}$	$3.39^{(6)}$	$2.89^{(6)}$	$2.73^{(6)}$	$2.54^{(6)}$	$2.43^{(6)}$	$2.40^{(6)}$	$2.39^{(6)}$
FJS	7.06	25.33	3.68	2.95	2.96	2.81	3.18	3.42	3.83
HASHq	$19.96^{(3)}$	$8.34^{(3)}$	$5.02^{(3)}$	$3.68^{(5)}$	$3.17^{(5)}$	$2.95^{(5)}$	$2.96^{(5)}$	$2.76^{(5)}$	$2.65^{(5)}$
IOM	9.37	6.67	5.26	4.38	3.96	3.73	3.47	3.30	3.20
WOM	9.98	7.01	5.28	4.32	3.91	3.53	3.25	3.11	3.02
WFR	8.25	6.47	4.67	3.61	2.78	2.47	2.17	1.89	1.75
WFRq	$5.20^{(4)}$	$3.89^{(4)}$	$3.08^{(4)}$	$2.42^{(4)}$	$\mathbf{2.08}^{(4)}$	$\mathbf{1.97}^{(4)}$	$\mathbf{1.91}^{(4)}$	$\mathbf{1.76}^{(4)}$	$\mathbf{1.69}^{(4)}$
speed-up	+20%	+12%	+7.6%	+2.9%	-5.4%	-5.7%	-7.72%	-12%	-15%
EPSM	6.72	6.36	2.86	2.13	1.94	1.94	1.92	1.86	1.87
TSOq	$5.54^{(5)}$	$4.05^{(5)}$	$3.26^{(5)}$	$2.61^{(5)}$	$2.23^{(5)}$	-	-	-	-

Table 4. Experimental results on a natural language sequence.

Experimental results show that the BSDMq algorithm obtains the best running times among previous solutions, especially in the case of long patterns. However it is second to the EBOM algorithm in the case of short patterns.

Our proposed WFR algorithm performs well in several cases and turns out to be competitive against previous solutions. It even turns out to be faster than the BSDMq algorithm in the case of very long patterns ($m \geq 256$), since the shift performed by the WFR algorithm are longer on average than the shifts performed by the BSDMq algorithm.

When the WFR algorithm is implemented using unchained-loop, the performance increases further. Specifically, the WFRq algorithm turns out to be the fastest solution for patterns with a moderate length and for long patterns ($m \geq 32$). Better performances are obtained in the case of small alphabets, where the gain is up to 25 %, whereas in the case of large alphabets the gain is up to 14 %.

4 Conclusions

In this paper we investigated a weak-factor-recognition approach to the exact string matching problem and devised an algorithm which, despite its quadratic worst case time complexity, shows a sublinear behaviour in practical cases. Experimental results show that under suitable conditions, our algorithm obtains better running times than the most efficient algorithms known in literature. It would be interesting to investigate whether multiple hashing functions can be used to reduce the number of false positives in the searching phase, in order to obtain better results. A deeper analysis of the implemented hash function and of the implemented data structure will be performed in future works.

Acknowledgments

This work has been supported by G.N.C.S., Istituto Nazionale di Alta Matematica "Francesco Severi".

References

1. C. Allauzen, M. Crochemore, M. Raffinot. Factor oracle: a new structure for pattern matching. *in SOFSEM'99, Lecture Notes in Computer Science*, Vol. 1725, pages 291–306, 1999.
2. R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.
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):589–608, 2005.
4. D. Cantone and S. Faro. Improved and Self-Tuned Occurrence Heuristics. *Journal of Discrete Algorithms*, 28:73–84, 2014.
5. D. Cantone, S. Faro, and E. Giaquinta. A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. *Inf. Comput.*, 213:3–12, 2012.
6. C. Charras and T. Lecroq. *Handbook of exact string matching algorithms*. King's College, 2004.
7. M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12(4):247–267, 1994.
8. B. Durian, H. Peltola, L. Salmela, and J. Tarhio. Bit-parallel search algorithms for long patterns. In *SEA, Lecture Notes in Computer Science*, vol. 6049, pages 129–140, 2010.
9. B. Durian, T. Chhabra, S.S. Ghuman, T. Hirvola, H. Peltola, J. Tarhio. Improved Two-Way Bit-parallel Search. In *Proc. of Stringology*, pages 71–83, 2014.
10. S Faro and O. Külekci. Fast and Flexible Packed String Matching. *Journal of Discrete Algorithms*, 28:61–72, 2014.
11. S. Faro and T. Lecroq. Efficient Variants of the Backward-Oracle-Matching Algorithm. *Int. J. Found. Comput. Sci.* 20(6):967–984, 2009.
12. S. Faro, T. Lecroq, S. Borzì, S. Di Mauro, A. Maggio. The String Matching Algorithms Research Tool. In *Proc. of Stringology*, pages 99–111, 2016.
13. S. Faro and T. Lecroq. The exact string matching problem: a comprehensive experimental evaluation. *CoRR*, abs/1012.2547, 2010.
14. S. Faro and T. Lecroq. A Fast Suffix Automata Based Algorithm for Exact Online String Matching. In *CIAA, Lecture Notes in Computer Science*, vol. 7381, pages 149–158, 2012.
15. S. Faro and T. Lecroq. A Multiple Sliding Windows Approach to Speed Up String Matching Algorithms. In *SEA, Lecture Notes in Computer Science*, vol. 7276, pages 172–183, 2012.
16. S. Faro and T. Lecroq. The exact online string matching problem: a review of the most recent results. *ACM Computing Surveys*, 45(2): Article No. 13, 2013.
17. K. Fredriksson and S. Grabowski. Practical and Optimal String Matching. *SPIRE, Lecture Notes in Computer Science*, vol. 3772, pages 376–387, 2005.
18. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.
19. T. Lecroq. Fast exact string matching algorithms. *Inf. Process. Lett.*, 102(6):229–235, 2007.
20. G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *CPM, Lecture Notes in Computer Science*, vol. 1448, pages 14–33, 1998.
21. H. Peltola, J. Tarhio. Variations of Forward-SBNDM. In *Proc. of Stringology*, pages 3–14, 2011.
22. A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.

Counting Mismatches with SIMD

Fernando J. Fiori*, Waltteri Pakalén, and Jorma Tarhio

Department of Computer Science
Aalto University, P.O.B. 15400
FI-00076 Aalto, Finland
fiorifj@gmail.com, waltteri.pakalen@aalto.fi, tarhio@iki.fi

Abstract. We consider the k mismatches version of approximate string matching for a single pattern and multiple patterns. For these problems we present new algorithms utilizing the SIMD (Single Instruction Multiple Data) instruction set extensions for patterns of up to 32 characters. We apply SIMD computation in two ways: in counting of mismatches and in calculation of fingerprints. We demonstrate the competitiveness of our solutions by practical experiments.

1 Introduction

The *string matching problem* is defined as follows: given a pattern $P = p_0 \cdots p_{m-1}$ and a text $T = t_0 \cdots t_{n-1}$ in an alphabet Σ, find all the occurrences of P in T. In this paper we consider the k *mismatches* variation of the problem where P' is an occurrence of P, if $|P'| = |P|$ holds and P' has at most k mismatches with P. The mismatch distance of two strings of equal length is also called the *Hamming distance*.

There are numerous good solutions for the k mismatches problem, see e.g. Navarro's survey [26]. In this article, we introduce new algorithms for the problem. Besides the single pattern problem, we also consider the multiple pattern variation. Our solutions utilize SIMD (Single Instruction Multiple Data) instruction set extensions [17,20]. We apply SIMD computation in two ways: in counting of mismatches and in calculation of fingerprints a.k.a. hash values. Our emphasis is on the practical efficiency of the algorithms and we show the competitiveness of the new algorithms by practical experiments. Our new algorithms for the single pattern problem are faster than reference methods in most cases tested, and our multiple pattern algorithm beats Fredriksson and Navarro's algorithm [13] with a wide margin.

The rest of the paper is organized as follows. Section 2 reviews earlier solutions. Section 3 introduces SIMD computation and the SIMD techniques applied. Section 4 and 5 describe the new algorithms, Section 6 presents the results of practical experiments, and Section 7 concludes the article.

2 Earlier Solutions

There are many algorithms for the string matching with k mismatches problem. Most of them solve the single pattern variation, whereas only few exist for the multiple pattern counterpart. Naively, an algorithm for the single pattern variation can be extended to solve the multiple pattern variation by executing it separately for each pattern. In the following, we will review earlier solutions to these variations.

* Visitor from Universidad Nacional de Rosario, Rosario, Santa Fe, Argentina.

Fernando J. Fiori, Waltteri Pakalén, Jorma Tarhio: Counting Mismatches with SIMD, pp. 51–61.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

2.1 Single String Matching with k Mismatches

A naive algorithm works in $O(mn)$ time in the worst case and in $O(kn)$ time on average if individual characters in P and T are chosen independently and uniformly from the alphabet Σ.

Baeza-Yates and Gonnet [3] presented Shift-Add (SA), the first bit-parallel algorithm for the k mismatches problem. Shift-Add works in linear time for short patterns $m \leq w/\lceil \log_2(k+1)+1 \rceil$ where w is the width of the computer word. Shift-Add is still competitive for short patterns and large k [15]. Ďurian et al. [9] presented variations of SA, TuSA and TwSA, which process the alignment window backwards.

Approximate Boyer–Moore (ABM) by Tarhio and Ukkonen [30] is a generalization of the Boyer–Moore–Horspool algorithm [18] to approximate string matching. In ABM, shifting is based on a q-gram, $q = k + 1$. Liu et al. [24] tuned ABM for small alphabets. Their algorithm applies wider q-grams and is called FAAST (short for a Fast Algorithm for Approximate STring matching). Salmela et al. [28] designed an enhanced version of FAAST. We call this algorithm EF. In Sect. 4.2 we will integrate SIMD computation with EF.

Approximate BNDM (ABNDM) by Navarro and Raffinot [27] is based on the BNDM algorithm [27] for exact string matching. BNDM simulates the suffix automaton of the reversed pattern with bit-parallelism. ABNDM as well as ABM, FAAST, EF, and TwSA achieve a sublinear running time on average in the case of favorable problem parameters.

The Baeza-Yates–Perlberg algorithm (BYP) [5] is based on a partitioning scheme, where at least one of P's substrings of length $l = \lfloor m/(k+1) \rfloor$ is exactly present in an approximate occurrence of P. In the preprocessing phase it splits the pattern into subpatterns of length l, and then it performs a multiple exact string matching search of these subpatterns. Whenever one of the subpatterns is found, it checks if there is an approximate pattern match with Ukkonen's dynamic algorithm [31]. In Sect. 4.3 we will integrate SIMD computation with BYP.

Besides practically oriented algorithms mentioned above, there are other solutions to the k mismatches problem: the kangaroo method [14,23], the algorithms based on the fast Fourier transform and marking [1,2,12], and the $O(nk^2 \log k/m + n \text{ polylog } m)$ solution presented by Clifford et al. [7], which is the best theoretical result.

2.2 Multiple String Matching with k Mismatches

The first algorithm for multiple string matching with k mismatches was presented by Muth and Manber [25]. For $k = 1$ they preprocess all the strings that result of taking one character out of every pattern, and compute a hash value for each of them, storing it in a table. This amounts to an $O(rm)$ preprocessing time for r patterns. For a text window of m characters, they compute m hash values in the same way as for the patterns. If there is a match in the hash table, a naive verification follows. The average search time is $O(mn(1 + rm^2/M))$, where M is the size of the hash table. If $M = \Omega(rm^2)$, this results in $O(mn)$. In this way, the total cost of the algorithm is $O(m(r + n))$. However, if $k > 1$ they have to preprocess all the strings that result from taking k characters out of every pattern, amounting for a total time complexity of $O(m^k(r + n))$. Hence, it allows only very small k in practice, but the overall complexity is rather independent of the number of patterns to search, given that n is usually much larger than r.

Later, Baeza-Yates and Navarro [4] designed an algorithm which is based on a similar partition scheme as in the BYP algorithm. They split every pattern into subpatterns of length $l = \lfloor m/(k+1) \rfloor$ and perform an exact multiple pattern search. Whenever there is a subpattern occurrence, they check for the entire pattern with an approximate single pattern matching algorithm.

The fastest algorithm to date is Fredriksson and Navarro's algorithm [13], which is optimal on average. It places a window over the text, in which q-grams are read in a backwards order. Whenever an occurrence is impossible, the window is shifted past the read q-grams. The average complexity of the algorithm is $O((k + \log_\sigma(rm))n/m)$ for $\alpha < 1/2 - O(1/\sqrt{\sigma})$, where $\alpha = k/m$ is the difference ratio.

3 SIMD Techniques

SIMD [20] is a type of parallel architecture that allows one instruction to be operated on multiple data items at the same time. Initially, SIMD was used in multimedia, especially in processing images or audio files. SIMD instructions have since found applications in other areas such as cryptography. Recently, they have also been applied to string matching [6,10,21,22,29].

Streaming SIMD Extensions comprise of SIMD instruction sets supported by modern processors which allow computation on vectors of length 16 bytes in the case of SSE2 and 32 bytes in the case of AVX2. In the near future, one can process 64 bytes with AVX-512. The instructions operate on such vectors stored in special registers. As one instruction is performed on all the data in these vectors, it is considered SIMD computation.

Next, we describe our techniques to use SIMD in the new algorithms. In the descriptions, the SSE2 instructions are listed for 16 bytes (= 128 bits). There are corresponding AVX instructions for 32 bytes (= 256 bits). We assume that a byte represents one character.

3.1 Counting of Mismatches

Counting mismatches is an usual operation in approximate string matching. It can be done with the instructions *simd-cmpeq*(x, y) and *simd-popcount*(x) explained below. In practice, we also need the instruction *simd-load*(x), which is an intrinsic function of the compiler formally defined as

$$\_\_m128i \ \_mm\_loadu\_si128(x).$$

This instruction loads 16 bytes from the address x to a SIMD register given as the left-hand side of an assignment statement. The instruction *simd-cmpeq*(x, y) is formally

$$\_mm\_movemask\_epi8(\_mm\_cmpeq\_epi8(\_\_m128i\ x, \_\_m128i\ y)).$$

The instruction $\_mm\_cmpeq\_epi8$ compares 16 bytes in x and y bytewise for equality and stores the result. The instruction $\_mm\_movemask\_epi8$ creates a bitvector from the most significant bit of each byte of the parameter. The instruction *simd-popcount*(x) counts the number of on bits in x and is formally

$$\_mm\_popcnt\_u32(x).$$

The *simd-cmpeq*(x, y) instruction, therefore, makes it possible to compare up to α characters at the same time, where α is 16 or 32. The result is a bitvector of the pairwise comparisons. Lastly, a popcount operation on the result tells the number of matching characters.

3.2 CRC as a Fingerprint

There are many filtration methods for approximate string matching. Those methods contain two phases which are usually interleaved. The filtration phase selects match candidates and the checking phase verifies them. The former often entails the calculation of a fingerprint or a hash value from a q-gram, with which precomputed tables are accessed. Such a calculation can be performed with the *simd-crc*(x) instruction. A similar instruction was first used by Faro and Külekci [10,11] in exact string matching.

The instruction *simd-crc*(x) returns a b-bit value by first calculating a 32-bit cyclic redundancy checksum (CRC) of a 64-bit value, and then taking the b least significant bits of the CRC. It is formally

$$\_mm\_crc32\_u64(x)\ \&\ mask,$$

where x is a 64-bit integer, $mask$ is $2^b - 1$, and '&' is bitparallel *and*. Based on our experiments, the best value of b depends on the problem parameters.

4 Improved Solutions – Single Pattern

4.1 Variations of Naive

A straightforward approach to string matching with k mismatches is the naive counting of mismatches. Alg. 1 is the pseudocode of the naive algorithm ANS (short for Approximate Naive with SIMD). ANS counts the character matches with P starting from the $n - m + 1$ first positions of the text. According to our experiments (see Section 6), it is clearly faster than both the classic Shift-Add [3] and TuSA [9].

Algorithm 1: ANS
$x \leftarrow$ *simd-load*$(p_0 \cdots p_{m-1})$
for $i \leftarrow 0$ to $n - m$ do
  $y \leftarrow$ *simd-load*$(t_i \cdots t_{i+\alpha-1})$
  $t \leftarrow$ *simd-cmpeq*(x, y)
  if *simd-popcount*$(t) \geq m - k$ then $occ \leftarrow occ + 1$

There is a way to make ANS even faster when α is 16. We preprocess the condition *simd-popcount*$(t) \geq m - k$ to a Boolean array D for each vector t of 16 bits. Then the last line of ANS is changed to

$$\text{if } D[t] \text{ then } occ \leftarrow occ + 1$$

We call this variation ANS2. ANS2 is about 30% faster than ANS in our experiments in Sect. 6.

For longer patterns, $16 < m \leq 32$, the last line will be

$$\text{if } D[t\ \&\ mask] \text{ then if } \textit{simd-popcount}(t) \geq m - k \text{ then } occ \leftarrow occ + 1$$

where $mask$ is $2^{16}-1$. In other words, the first 16 characters of the pattern are tested first.

In our test environment (see Sect. 6), the computation of D takes about 2 ms, which is tolerable. Note that the preprocessing time would grow exponentially if D were extended for wider vectors. The speed of ANS does not depend on k, while the speed of ANS2 obviously decreases when k approaches m for $m > 16$. Moreover, we observed a further decrease in practice, as discussed in Sect. 6.1.

Besides the *simd-cmpeq* instruction and other basic SIMD commands, the SIMD architecture comprises of several aggregation operations for string processing. However, they are too slow for the k mismatches problem on those processors we have tested. Hirvola [16] implemented several algorithms similar to ANS with PCMP and STTNI instructions, but all those algorithms are clearly slower than ANS and TuSA.

4.2 EF Enhanced with SIMD

EF contains a filtration and a checking phase. The checking method can be replaced with ANS2 (see Sect. 4.1), while the fingerprint computation of the filtration method can be replaced with the CRC fingerprint technique of Sect. 3. Alg. 2 shows the pseudocode of EF.

Algorithm 2: EF
$s \leftarrow m-1$
while $s < n$ do
  $f \leftarrow \sum_{i=0}^{q-1} map(t_{s-i}) * 4^i$
  if $M[f] \leq k$ then
    $c \leftarrow M[f]$
    for $i \leftarrow 1$ to $m-q$ do
      if $t_{s-q-i+1} \neq p_{m-q-i}$ then
        $c \leftarrow c+1$
        if $c > k$ then break
    if $c \leq k$ then $occ \leftarrow occ+1$
  $s \leftarrow s + S_q[f]$

For each q-gram $u_0 \cdots u_{q-1}$, the preprocessing phase of EF computes the Hamming distance with the end of all prefixes of the pattern. With this information, a shift table S_q can be constructed (see details in [28]). M is another precomputed table. M gives the Hamming distance of a q-gram against the last q-gram of the pattern. Whenever $M[t_{s-q+1} \cdots t_s] > k$ holds, the algorithm shifts forward without processing the alignment window further. Both the tables are accessed with the fingerprint $f \leftarrow \sum_{i=0}^{q-1} map(t_{s-i}) * 4^i$, where the function map maps each DNA character to an integer in $\{0,1,2,3\}$.

Alg. 3 is the pseudocode of EFS, which is EF enhanced with SIMD computation for $m \leq 16$. The array D is computed in the same way as for ANS2. For longer patterns, $16 < m \leq 32$, the required change is the same as in the case of ANS2.

4.3 BYP Enhanced with SIMD

BYP looks for exact occurrences of substrings of length $l = \lfloor m/(k+1) \rfloor$ (called subpatterns from now on) of the pattern in the text. To achieve this, we employed a tuned version of MEPSM algorithm [11] for exact multiple string matching. MEPSM

Algorithm 3: EFS
$x \leftarrow$ *simd-load*$(p_0 \cdots p_{m-1})$
$s \leftarrow m - 1$
while $s < n$ do
  $f \leftarrow$ *simd-crc*$(t_{s-q+1} \cdots t_s)$
  if $M[f] \leq k$ then
    $y \leftarrow$ *simd-load*$(t_{s-m+1} \cdots t_s)$
    $t \leftarrow$ *simd-cmpeq*(x, y)
    if $D[t]$ then $occ \leftarrow occ + 1$
  $s \leftarrow s + S_q[f]$

reports subpattern occurences, which are later verified by ANS2 (see Sect. 4.1). Let us call the total algorithm BYPS.

MEPSM computes the CRC fingerprint of every q-gram of each subpattern, where $q \leq l$ is a parameter of MEPSM. The information about which q-gram the fingerprint belongs to is stored in a table. Afterwards, during the search, the algorithm looks for matching fingerprints of q-grams in the text. Whenever a subpattern occurrence candidate is found, it is naively verified and reported in case of a match. After each q-gram analysis, the algorithm shifts forwards by $l - q + 1$ characters.

We tuned MEPSM by setting q as large as possible, which causes less fingerprint collisions. Conversely, larger q reduces shifts between alignments. However, this trade-off showed to be really satisfactory, especially in the case of small subpatterns.

Our algorithm has the practical limitation that $4 \leq l \leq 32$ must hold for l, as the performance drops substantially otherwise.

5 Improved Solution – Multiple Patterns

We have extended BYPS algorithm to work with multiple patterns. The new algorithm MBYPS works as follows:

1. In the preprocessing, we split every pattern into subpatterns of length l. Then we compute the CRC fingerprint of every q-gram of each subpattern, where $q \leq l$ is a parameter of the MEPSM algorithm. The fingerprint is used to access a table that stores information about which subpattern of which pattern it was computed from.
2. In the search, we compute the fingerprint of a q-gram in the text, with which we fetch the corresponding information from the table. We perform a shift of $l-q+1$ characters in the text after analysing each q-gram, which is the maximum number of characters we can skip.
3. For every subpattern associated with the fingerprint, we naively check if it appears exactly at this point. If it does, a possible occurrence of the corresponding pattern is reported.
4. Every time a match candidate of a pattern is found, we use an approximate single pattern matching algorithm to verify it.

For the phase of exact multiple string matching, we use the tuned version of MEPSM as in BYPS. For the phase of approximate single string matching, we use ANS2 for $m \leq 32$. For longer patterns, another method such as Ukkonen's dynamic algorithm [31] should be used.

The phase of approximate single string matching requires the occurrences of the subpatterns to be ordered, so as to avoid re-verifying an occurrence. However, MEPSM does not guarantee ordering. This has been solved by executing the approximate single pattern matching algorithm in a larger window. If a subpattern occurrence is found at position x in the text, we check for an approximate pattern occurrence from position $x-(m-l)$ to $x+m$. Thus, once an occurrence of a pattern has been found, a newer occurrence will never precede it positionally, as shown next.

Justification. Let x and y be the text positions of an old and a new q-gram occurrence respectively, which correspond to exact subpatterns occurrences. These subpatterns are placed in text positions s_x and s_y respectively, such that $x-(l-q) \leq s_x \leq x$ and $y-(l-q) \leq s_y \leq y$. Then the patterns which contain such subpatterns occur at positions p_x and p_y respectively, such that $s_x-(m-l) \leq p_x \leq s_x$ and $s_y-(m-l) \leq p_y \leq s_y$. But we perform our approximate pattern matching search from $s_x-(m-l)$ onwards. So we need to check that:

$$s_x-(m-l) \leq p_y$$

Which is valid if:

$$s_x-(m-l) \leq s_y-(m-l) \iff s_x \leq s_y$$

Which is true if:

$$x \leq y-(l-q) \tag{1}$$

It could happen that $x=y$ but MEPSM reports first the highest occurrences of a determined supattern (i.e. it preserves ordering of occurrences of q-grams for the same subpattern). So $x+(l-q+1) \leq y$ because we skip $l-q+1$ bytes after analysing a q-gram. Then (1) is true if:

$$x \leq x+(l-q+1) \iff q-1 \leq l$$

Which is true because q is the size of the q-grams of the subpatterns of length l.

6 Experiments

The tests were run on Intel Core i7-6500U 2.5 GHz with 16 GiB memory. This processor has SSE2 and AVX2, but not AVX-512. Programs were written in the C programming language and compiled with gcc 5.4.0 using -O3 optimization level. All the algorithms were implemented and tested in the testing framework of Hume and Sunday [19].

We used two texts: DNA (the genome of E. Coli, 4.6 MB) and English (the KJV Bible, 4.0 MB) for testing. The texts were taken from the Smart corpus[1]. Sets of patterns of various lengths were randomly taken from each text. In the case of single pattern matching, each set contains 200 patterns.

A word of warning. Our experimental results hold on the processor we used in our tests. It is possible that future processors will give different results, if the relative speed of instructions will change. In exact string matching we have encountered such a development several times. For example, in the case of English text, SBNDM2 [8] is 75% faster than ufast-rev-md2 [19] on our test processor for $m=5$, but the situation is almost reversed on a 20 years older processor Pentium 75: ufast-rev-md2 is 47% faster than SBNDM2!

[1] https://www.dmi.unict.it/~faro/smart/

6.1 Single String Matching with k Mismatches

The new algorithms were compared with the following earlier algorithms: SA [3], TuSA [9], TwSA [9], EF [28] and BYP [5]. According to tests by Hirvola [16], TwSA was the best for English data. According to tests by Salmela et al. [28], EF was the best for DNA data.

The results are shown in Table 1 with the best times bolded. We can observe that ANS2 is the best for several parameter combinations on both DNA and English. Meanwhile, EFS and BYPS are the best for some cases with small k on DNA, and TwSA is the best on English for some combinations when $k > 1$ and $m \geq 16$.

Like SA, ANS and ANS2 work for all possible values of k, and ANS does so at an almost constant speed independent of the value of k. On the contrary, TuSA and TwSA are limited to small values of k for long patterns. For example, they only work for $k < 4$ in the case of $m = 20$. Furthermore, the speed of TwSA degrades as k grows. EF, EFS, BYP and BYPS exhibit similar behavior, with k affecting their speed. Despite this, the growth of k can be tolerated given that m is large enough, i.e. when we have a large difference ratios. Some timings of BYPS have been omitted because it does not work for $l < 4$.

ANS2 is the best for small patterns across both texts with every value of k. As the pattern length increases, EFS overtakes ANS2 on DNA, and TwSA overtakes ANS2 on English up to a small value of k. Once k surpasses this value, ANS2 becomes the best again.

Lastly, in Sect. 4.1 it was stated that the speed of ANS2 obviously decreases when k approaches m for $m > 16$. Beyond that, however, we observed a peak in the speed of ANS2 for $m > 16$, as depicted in Figure 1. We tried two different compilers, and multiple compilation options, but the peak persisted. It is conjectured to be caused by branch mispredictions. Thus, ANS is the better choice over ANS2 for $k > 7$ on DNA, and $k > 11$ on English when $m > 16$.

BYPS has also been tested for longer patterns. According to our experiments and following the same line as stated by Baeza-Yates and Perlberg in [5], BYP and BYPS obtain their best results for high difference ratios.

Table 1. Search times (in seconds) of algorithms for approximate matching with k mismatches.

	$m = 8$			$m = 12$			$m = 16$			$m = 20$			
k	1	2	3	1	2	3	1	2	3	1	2	3	Σ
SA	1.99	2.00	1.99	1.99	1.99	2.01	2.02	1.99	1.99	1.99	1.99	1.99	DNA
TuSA	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	
TwSA	1.73	2.31	2.63	1.16	1.54	1.85	0.88	1.15	1.39	0.71	0.92	1.12	
ANS	1.13	1.13	1.13	1.13	1.13	1.13	1.13	1.13	1.13	1.13	1.13	1.13	
ANS2	**0.76**	**0.76**	**0.76**	0.76	**0.76**	**0.76**	0.83	**0.85**	**0.90**	0.82	0.83	**0.86**	
EF	1.70	2.46	4.27	1.08	1.51	2.55	0.81	1.14	1.94	0.66	0.95	1.71	
EFS	1.09	1.78	3.76	**0.71**	1.13	2.12	0.55	0.91	1.73	0.46	**0.79**	1.63	
BYP	4.95	-	-	4.54	7.20	-	4.15	6.28	9.58	4.15	6.02	8.05	
BYPS	1.56	-	-	1.48	1.79	-	**0.42**	1.53	2.21	**0.41**	1.57	1.62	
SA	1.74	1.74	1.74	1.74	1.74	1.74	1.74	1.74	1.74	1.74	1.74	1.73	English
TuSA	1.45	1.45	1.45	1.45	1.45	1.45	1.45	1.45	1.45	1.45	1.45	1.45	
TwSA	0.78	1.14	1.52	0.61	0.83	1.07	0.49	**0.65**	0.82	0.43	**0.54**	**0.67**	
ANS	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	0.99	1.01	0.99	
ANS2	0.72	**0.66**	**0.67**	0.66	**0.66**	**0.66**	0.66	0.66	**0.66**	0.72	0.72	0.72	
BYP	1.39	-	-	1.03	1.83	-	0.86	1.50	2.16	0.72	1.27	1.83	
BYPS	**0.65**	-	-	**0.43**	0.78	-	**0.32**	0.82	0.96	**0.31**	0.55	0.87	

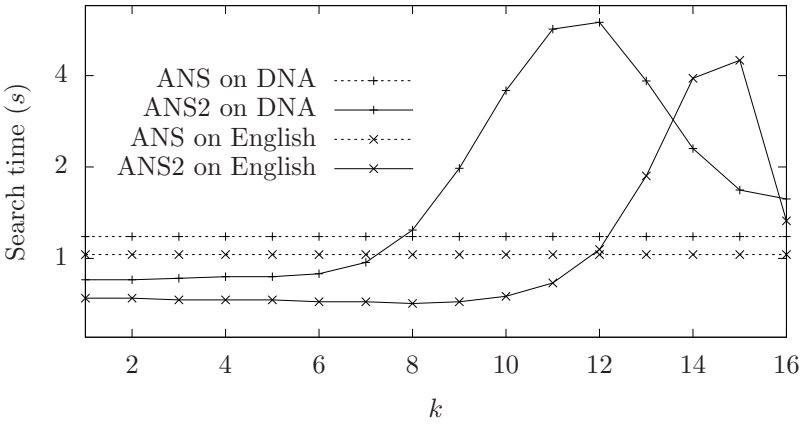


Figure 1. Search times of ANS and ANS2 as a function of k for $m = 20$.

6.2 Multiple String Matching with k Mismatches

We compare our new MBYPS algorithm with Fredriksson and Navarro's algorithm (FN) [13]. We used sets of 10, 100 and 1000 patterns. The results show that our algorithm outperforms FN in all cases. There is a larger difference for larger sets of patterns, and for larger difference ratios. We also ran tests on a Protein sequence and obtained similar results.

It is worth mentioning that we performed thorough testing to choose the best parameters for FN in each case. For DNA we obtained the same tuning mentioned in [13] as the best configuration.

Table 2. Search times (in seconds) for multiple approximate matching with k mismatches.

	m=8	$m = 16$			$m = 24$			$m = 32$				
k	1	1	2	3	1	2	3	1	2	3	r	Σ
FN	0.129	0.018	0.120	0.396	0.006	0.009	0.015	0.004	0.006	0.008	10	DNA
MBYPS	0.031	0.007	0.019	0.066	0.002	0.008	0.013	0.001	0.003	0.008		
FN	1.132	0.165	0.996	4.635	0.012	0.032	0.086	0.007	0.016	0.034	100	
MBYPS	0.240	0.010	0.154	0.585	0.003	0.012	0.069	0.002	0.005	0.013		
FN	11.220	1.697	10.222	44.030	0.098	0.364	1.044	0.059	0.158	0.344	1000	
MBYPS	2.574	0.033	1.695	6.797	0.012	0.055	0.695	0.011	0.025	0.075		
FN	0.026	0.008	0.014	0.027	0.005	0.008	0.012	0.004	0.006	0.009	10	English
MBYPS	0.013	0.006	0.008	0.008	0.001	0.006	0.007	0.001	0.002	0.006		
FN	0.143	0.043	0.173	0.406	0.023	0.104	0.174	0.021	0.084	0.125	100	
MBYPS	0.046	0.009	0.048	0.093	0.002	0.009	0.021	0.001	0.003	0.010		
FN	1.723	0.373	1.212	4.224	0.201	0.491	1.028	0.148	0.333	0.563	1000	
MBYPS	0.314	0.022	0.348	0.984	0.008	0.032	0.167	0.007	0.018	0.045		

7 Concluding Remarks

We have demonstrated that simple SIMD solutions are competitive in searching for approximate single pattern matches within the Hamming distance for patterns $|P| \leq 32$. In Sect. 4.1 and Sect. 4.2, we showed that the algorithms for naive counting of mismatches can be used as a checking method for single pattern filtration algorithms.

Meanwhile, the fingerprint calculation of a filtration method can be replaced with the CRC fingerprint technique of Sect. 3.2.

We have also presented an effective way of using the SIMD techniques for approximate multiple string matching in Sect. 5. The resulting algorithm is substantially faster than the previous most competitive algorithm across multiple alphabets.

When AVX-512 will become widely available, it may be possible to achieve better speed-ups, because compare and mask instructions have been merged.

References

1. K. Abrahamson: *Generalized string matching*. SIAM Journal on Computing, 16(6) 1987, pp. 1039–1051.
2. A. Amir, M. Lewenstein, and E. Porat: *Faster algorithms for string matching with k mismatches*. Journal of Algorithms, 50(2) 2004, pp. 257–275.
3. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching*. Communications of the ACM, 35(10) 1992, pp. 74–82.
4. R. Baeza-Yates and G. Navarro: *New and faster filters for multiple approximate string matching*. Random Structures & Algorithms, 20(1) 2002, pp. 23–49.
5. R. Baeza-Yates and C. Perlberg: *Fast and practical approximate string matching*. Information Processing Letters, 59(1) 1996, pp. 21–27.
6. T. Chhabra, S. Faro, M. O. Külekci, and J. Tarhio: *Engineering order-preserving pattern matching with SIMD parallelism*. Software: Practice and Experience, 47(5) 2017, pp. 731–739.
7. R. Clifford, A. Fontaine, E. Porat, B. Sach, and T. Starikovskaya: *The k-mismatch problem revisited*, in Proc. 27th ACM-SIAM Symposium on Discrete Algorithms, SIAM, 2016, pp. 2039–2052
8. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching*. Information Processing Letters, 110(4) 2010, pp. 148–152.
9. B. Ďurian, T. Chhabra, S. Guman, T. Hirvola, H. Peltola, and J. Tarhio: *Improved two-way bit-parallel search*, in Proceedings of the Prague Stringology Conference, Prague, Czech Republic, 2014, pp. 71–83
10. S. Faro and M. O. Külekci: *Fast packed string matching for short patterns*, in Proc. 15th Meeting on Algorithm Engineering and Experiments, SIAM, 2013, pp. 113–121.
11. S. Faro and M. O. Külekci: *Towards a very fast multiple string matching algorithm for short patterns*, in Proceedings of the Prague Stringology Conference, Prague, Czech Republic, 2013, pp. 78–91.
12. K. Fredriksson and S. Grabowski: *Exploiting word-level parallelism for fast convolutions and their applications in approximate string matching*. European Journal of Combinatorics, 34(1) 2013, pp. 38–51.
13. K. Fredriksson and G. Navarro: *Average-optimal single and multiple approximate string matching*. ACM Journal of Experimental Algorithmics, 9 2004.
14. Z. Galil and R. Giancarlo: *Improved string matching with k mismatches*. ACM SIGACT News, 17(4) 1986, pp. 52–54.
15. S. Grabowski and K. Fredriksson: *Bit-parallel string matching under Hamming distance in $O(n\lceil m/w\rceil)$ worst case time*. Information Processing Letters, 105(5) 2008, pp. 182–187.
16. T. Hirvola: *Bit-parallel approximate string matching under Hamming distance*. Master's Thesis, Aalto University, 2016. http://urn.fi/URN:NBN:fi:aalto-201608263081
17. M. Hassaballah, S. Omran, and Y. B. Mahdy: *A review of SIMD multimedia extensions and their usage in scientific and engineering applications*. The Computer Journal, 51(6) 2008, pp. 630–649.
18. R. N. Horspool: *Practical fast searching in strings*. Software: Practice and Experience, 10(6) 1980, pp. 501–506.
19. A. Hume and D. Sunday: *Fast string searching*. Software: Practice and Experience, 21(11) 1991, pp. 1221–1248.
20. Intel: *Intel (R) 64 and IA-32 architectures software developer's manual*. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (Retrieved in May 2017).

21. M. O. Külekci: *Filter based fast matching of long patterns by using SIMD instructions*, in Proceedings of the Prague Stringology Conference, Prague, Czech Republic, 2009, pp. 118–128.
22. S. Ladra, O. Pedreira, J. Duato, and N. R. Brisaboa: *Exploiting SIMD instructions in current processors to improve classical string algorithms*, in Proc. 16th East European Conference on Advances in Databases and Information Systems, LNCS, vol. 7503, Springer, 2012, pp. 254–267.
23. G. Landau and U. Vishkin: *Efficient string matching with k mismatches*. Theoretical Computer Science, 43 1986, pp. 239–249.
24. Z. Liu, X. Chen, J. Borneman, and T. Jiang: *A fast algorithm for approximate string matching on gene sequences*, in Proc. 16th Symposium on Combinatorial Pattern Matching, LNCS, vol. 3537, Springer, Berlin, 2005, pp. 79–90.
25. R. Muth and U. Manber: *Approximate multiple string search*, in Proc. 7th Symposium on Combinatorial Pattern Matching, LNCS, vol. 1075, Springer, Berlin, 1996, pp. 75–86.
26. G. Navarro: *A guided tour to approximate string matching*. ACM Computing Surveys, 33(1) 2001, pp. 31–88.
27. G. Navarro and M. Raffinot: *Fast and flexible string matching by combining bit-parallelism and suffix automata*. Journal of Experimental Algorithmics, 5 2000, pp. 4.
28. L. Salmela, J. Tarhio, and P. Kalsi: *Approximate Boyer-Moore string matching for small alphabets*. Algorithmica, 58(3) 2010, pp. 591–609.
29. J. Tarhio, J. Holub, and E. Giaquinta: *Technology beats algorithms (in exact string matching)*. To appear in: Software: Practice and Experience.
30. J. Tarhio and E. Ukkonen: *Approximate Boyer-Moore string matching*. SIAM Journal on Computing, 22(2) 1993, pp. 243–260.
31. E. Ukkonen: *Finding approximate patterns in strings*. Journal of Algorithms, 6(1) 1985, pp. 132–137.

Dismantling DivSufSort*

Johannes Fischer and Florian Kurpicz

Dept. of Computer Science, Technische Universität Dortmund, Germany
johannes.fischer@cs.tu-dortmund.de, florian.kurpicz@tu-dortmund.de

Abstract. We give the first concise description of the fastest known suffix sorting algorithm in main memory, the *DivSufSort* by Yuta Mori. We then present an extension that also computes the LCP-array, which is competive with the fastest known LCP-array construction algorithm.

Keywords: text indexing; suffix sorting; algorithm engineering

1 Introduction

The suffix array [12] is arguably one of the most interesting and versatile data structure in stringology. Despite the plethora of theoretical and practical papers on suffix sorting (see the two overview articles [18,3] for an overview up to 2007/2012), the text indexing community faces the curiosity that the fastest and most space-conscious way to construct the suffix array is by an algorithm called *DivSufSort* (coded by Yuta Mori), which has only appeared as (almost undocumented) source code, and has never been described in an academic context. The speed and its space-consciousness make DivSufSort still the method of choice in many software systems, e.g. in bioinformatics libraries[1], and in the *succinct data structures library (sdsl)* [5].

The starting point of this article was that we wanted to get a better understanding of DivSufSort's functionality and the reasons for its advantages in performance, but we could not find any arguments for this neither in the literature nor in the documentation. We therefore dove into the source code (consisting of more than 1,000 LOCs) ourselves, and want to communicate our findings in this article. We point out that just very recently Labeit et al. [10] parallelized DivSufSort, making it also the fastest *parallel* suffix array construction algorithm (on all instances but one). We think that this successful parallelization adds another reason for why a deeper study of DivSufSort is worthwile.

Our Contributions and Outline. This article pursues two goals: First, it gives a concise description of the DivSufSort-algorithm (Sect. 3), so that readers wishing to understand or modify the source code have an easy-to-use reference at hand. Second (Sect. 4), we provide and describe our own enhancement of DivSufSort that also computes related and equally important information, the array of longest common prefixes of lexicographically adjacent suffixes (*LCP-array* for short). We test our implementation empirically on a well-accepted testbed and prove it competitive with existing implementations, sometimes even little faster.

To help the reader link our description to the implementation, we show relevant excerpts from the original code[2], along with their original line numbers in the source

* This work was supported by the German Research Foundation (DFG), priority programme "Algorithms for Big Data" (SPP 1736).

[1] https://github.com/NVlabs/nvbio, last seen 05.07.2017

[2] https://github.com/y-256/libdivsufsort, last seen 05.07.2017

Johannes Fischer, Florian Kurpicz: Dismantling DivSufSort, pp. 62–76.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$\mathsf{T}[i]$	c	d	c	d	c	d	c	d	c	c	d	d	$
$\mathsf{type}(i)$	B*	A	B*	A	B*	A	B*	A	B	B*	A	A	A

Figure 1: Classification of T= cdcdcdcdccdd$ (our running example).

code (difsufsort.c, sssort.c, and trsort.c). In the following, we use a slanted font for variables that also appear verbatim in the source code; e.g., T for the text.

2 Preliminaries

Let $\mathsf{T}= \mathsf{T}[0]\,\mathsf{T}[1]\ldots\mathsf{T}[\mathsf{n}-1]$ be a *text* of length n consisting of characters from an ordered *alphabet* Σ of size $\sigma = |\Sigma|$. For integers $0 \le i \le j \le \mathsf{n}$, the notation $[i,j)$ represents the integers from i to $j-1$, and $\mathsf{T}[i,j)$ the *substring* $\mathsf{T}[i]\ldots\mathsf{T}[j-1]$. We call $S_i = \mathsf{T}[i,\mathsf{n})$ the i-th *suffix* of T. The *suffix array* SA of a text T of length n is a permutation of $[0,\mathsf{n})$ such that $S_{\mathsf{SA}[i]} < S_{\mathsf{SA}[i+1]}$ for all $0 \le i < \mathsf{n}-1$. In SA, all suffixes starting with the same character $\mathsf{c0} \in \Sigma$ form a contiguous interval called $\mathsf{c0}$-*bucket*. The same is true for all suffixes starting with the same two characters $\mathsf{c0},\mathsf{c1} \in \Sigma$. We call the corresponding intervals $(\mathsf{c0},\mathsf{c1})$-*buckets*. The *inverse suffix array* ISA is the inverse permutation of SA. The *longest common prefix* of two suffixes S_i and S_j is $\mathsf{lcp}(i,j) = \max\{s \ge 0\colon \mathsf{T}[i,i+s) = \mathsf{T}[j,j+s)\}$. The *longest common prefix array* LCP of T contains the longest common prefixes of the lexicographically consecutive suffixes, i.e., $\mathsf{LCP}[0] = 0$ and $\mathsf{LCP}[i] = \mathsf{lcp}(\mathsf{SA}[i-1],\mathsf{SA}[i])$ for all $1 \le i \le \mathsf{n}-1$.

We classify all suffixes as follows (a technique first introduced by [7]; see Figure 1). The suffix S_i is an A-suffix (or "S_i has type A") if $\mathsf{T}[i] > \mathsf{T}[i+1]$ or $i = \mathsf{n}-1$. If $\mathsf{T}[i] < \mathsf{T}[i+1]$, then S_i is a B-suffix (or "has type B"). Last, if $\mathsf{T}[i] = \mathsf{T}[i+1]$ then S_i has the same type as S_{i+1}.[3] We further distinguish B-suffixes: if S_i has type B and S_{i+1} has type A, then suffix S_i is also a B*-suffix. Note that there are at most $\frac{\mathsf{n}}{2}$ B*-suffixes. The definition of types implies restrictions on how the suffixes are distributed within one bucket: A $(\mathsf{c0},\mathsf{c1})$-bucket cannot contain A-suffixes if $\mathsf{c0} < \mathsf{c1}$, and it cannot contain B-suffixes if $\mathsf{c0} > \mathsf{c1}$. If $\mathsf{c0} = \mathsf{c1}$ it cannot contain B*-suffixes. The classification also induces a partial order among the suffixes (see also Fig. 2):

Lemma 1. *Let S_i and S_j be two suffixes. Then*

1. *$S_i < S_j$ if S_i has type A, S_j has type B and $\mathsf{T}[i] = \mathsf{T}[j]$, and*
2. *$S_i < S_j$ if S_i has type B*, S_j has type B but not type B* and $\mathsf{T}[i,i+1] = \mathsf{T}[j,j+1]$.*

Proof. A- and B-suffixes can only occur together in a $(\mathsf{c0},\mathsf{c0})$-bucket. Assume that S_i and S_j start with $\mathsf{c0c0}$ followed by a (possibly empty) sequence of $\mathsf{c0}$'s and S_i, S_j have type A, B, resp. Let $u = \mathsf{T}[i+\mathsf{lcp}(i,j)]$ and $v = \mathsf{T}[j+\mathsf{lcp}(i,j)]$ be the first characters where the suffixes differ. Therefore, $u \le \mathsf{c0}$ and $v \ge \mathsf{c0}$. Since the characters differ, at least one of the inequalities is strict. The argument for the second case works analogously. □

Given two consecutive B*-suffixes S_i and S_j (i.e., there is no B*-suffix S_k such that $i < k < j$), we call the substring $\mathsf{T}[i,j+2)$ *B*-substring*. Also, for the last B*-suffix S_i (i.e., there is no B*-suffix S_k with $i < k < \mathsf{n}$), the substring $\mathsf{T}[i,\mathsf{n})$ is also called a B*-substring.

[3] This differs from [7], where S_i is always a B-suffix if $\mathsf{T}[i] = \mathsf{T}[i+1]$.

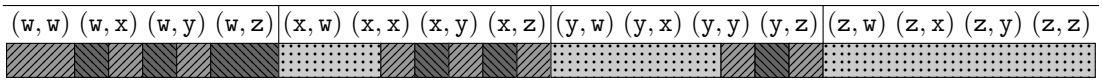


Figure 2: Position of the suffix types within the (c0, c1)-buckets for $\Sigma = \{\mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$. Light gray (▨) areas represent positions of A-suffixes, gray (▨) areas represent positions of B-suffixes, and dark gray (▨) areas represent positions of B*-suffixes.

3 DivSufSort

In this section we describe DivSufSort based on its current implementation (libdivsufsort v2.0.2). The algorithm consists of three phases:

- First, we identify the types of all suffixes and compute the corresponding c0- and (c0, c1)-bucket borders. This requires one scan of the text.
- Next, we sort all B*-suffixes and place them at their correct position in SA. This is the most complicated part, as we first have to sort the B*-substrings in-place. Then, we use the ranks of the sorted B*-substrings to sort the corresponding B*-suffixes.
- In the last step, we scan SA twice to induce the correct position of all remaining suffixes. (We first scan from right to left to induce all B-suffixes, followed by a scan from left to right, inducing all A-suffixes.)

Throughout the computation we utilize two additional arrays to store information about the buckets: BUCKET_A (for A-suffixes) and BUCKET_B (for B- and B*-suffixes) of size σ and σ^2, resp. The former is used to store values associated with A-suffixes and is accessed by only one character. The latter is used to store values associated with B- and B*-suffixes and is accessed by two characters. BUCKET_B[c0, c1] is short for BUCKET_B[|c0| · σ + |c1|] and BUCKET_BSTAR[c0, c1] is short for BUCKET_B[|c1| · σ + |c0|], where $|\alpha|$ denotes the rank of α in the alphabet Σ. Information about both suffixes can be stored in the same array (Figure 3), as there are no B*-suffixes in (c0, c0)-buckets and no B-suffixes in (c0, c1)-buckets for c0 > c1. We denote the number of B*-suffixes by m.

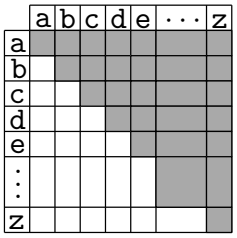


Figure 3: BUCKET_B (gray) and BUCKET_BSTAR represented as a 2-dimensional array.

3.1 Initializing DivSufSort

The initialization of DivSufSort is listed in divsufsort.c. First, we scan T from right to left (line 60), determine the type of each suffix and store the sizes of the corresponding buckets in BUCKET_A, BUCKET_B and BUCKET_BSTAR (lines 62, 69 and 65). In addition, we store the text position of each B*-suffix at the end of SA such that SA[n − m..n) contains the text positions of all B*-suffixes (line 66). We call this part of the suffix array PAb with PAb[i] = SA[n − m + i] for all $0 \le i <$ m (line 94), see Figure 4 (a) and (b).

Next (lines 81 to 90), we compute the prefix sum of BUCKET_A and BUCKET_BSTAR, such that BUCKET_A[c0] contains the leftmost position of each c0-bucket and BUCKET_BSTAR[c0, c1] contains the rightmost position of the corresponding B*-suffixes with respect only to other B*-suffixes, i.e., the positions are in

i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	0	0	0	0	0	0	0	0	0	2	4	6	9

(a)

	$	c	d	(c,c)	(c,d)
BUCKET_A	1	0	6	-	-
BUCKET_B	-	-	-	1	-
BUCKET_BSTAR	-	-	-	-	5

(b)

	$	c		d									
i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	0	0	0	0	0	0	0	0	0	2	4	6	9

(c)

	$	c	d	(c,c)	(c,d)
BUCKET_A	**0**	**1**	**7**	-	-
BUCKET_B	-	-	-	1	-
BUCKET_BSTAR	-	-	-	-	5

(d)

	$	c		d									
i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	4	0	1	2	3	0	0	0	0	2	4	6	9

(e)

	$	c	d	(c,c)	(c,d)
BUCKET_A	0	1	7	-	-
BUCKET_B	-	-	-	1	-
BUCKET_BSTAR	-	-	-	-	**0**

(f)

Figure 4: SA and the buckets after the first scan of T are shown in (a) and (b). PAb (dark gray ■ in (a), (c) and (e)) contains the text positions of all B*-suffixes in *text order*. The buckets (b) contain the number of suffixes beginning with the corresponding characters. In (d), they are updated such the first position of each c0-bucket is stored in BUCKET_A[c0] (bold entires). The SA does not change during this update, see (c). In (e) we stored references to the text positions in SA$[0..m-1]$ (light gray ■) and update the corresponding BUCKET_BSTAR with the first position in SA$[0..m-1]$ (bold entry in (f)).

the interval $[0, m)$, see Figures 4 (c) and (d), where (c) remains unchanged. During the sorting step, we do not sort the text positions. Instead we sort *references* to these positions. These references are stored in SA$[0..m)$ (line 97). During this step, BUCKET_BSTAR$[c0, c1]$ is updated (line 97), such that it now contains the leftmost reference corresponding to a B*-suffix in the $(c0, c1)$-bucket within the interval $[0, m)$. The reference to the last B*-suffix is put at the beginning of its corresponding bucket (line 100). This reference is a special case as it has no successor in PAb that is required for the comparison of two B*-substrings, see Figure 4 (e) and (f).

3.2 Sorting the B*-Suffixes

In this section, we describe how the B*-suffixes are sorted in three steps. First, all B*-substrings are sorted independently for each $(c0, c1)$-bucket (lines 134 to 142) using functions defined in sssort.c. Then (second step starting at line 146), a partial ISA (named ISAb) is computed, containing the ranks of the partially sorted B*-suffixes (sorted by their initial B*-substrings). Using these ranks we compute the lexicographical order of all B*-suffixes adopting an approach similar to prefix doubling, in the last step using functions defined in trsort.c (line 159). We augment the approach with *repetition detection* as introduced by Maniscalco and Puglisi [13].

Sorting the B*-Substrings. All B*-substrings in a BUCKET_BSTAR are sorted independently and in-place. The interval of SA that has not been used yet (SA[m..n − m)) serves as a buffer during the sorting (line 133). We refer to this part of SA as buf with buf$[i]$ = SA[m + i] for all $0 \leq i <$ n − 2m. This part of DivSufSort can be executed in parallel by sorting the BUCKET_BSTAR in parallel, i.e., all B*-substring in one BUCKET_BSTAR are sorted sequentially, but multiple BUCKET_BSTAR are processed in parallel (see divsufsort.c, lines 105 to 131). Here, each process gets a buffer of size $\frac{|\mathsf{buf}|}{\mathsf{p}}$, where p is the number of processes. All following line numbers in this subsection refer to sssort.c.

In the default configuration we only sort 1024 elements at once (see SS_BLOCKSIZE, e.g., line 763). If the size of buf is smaller than 1024 or the size of the current bucket, the bucket is divided in smaller subbuckets which are then sorted and merged (see line 767, splitting due to the buffer size and the loop at line 770 splitting with respect to the number of elements). Lines 789 to 802 are used to merge the last considered subbuckets. If the currently sorted bucket contains the last B*-substring it is moved to the corresponding position (lines 811 and 813).

The heavy lifting is done by the function ss_mintrosort that is an implementation of *Introspective Sort* (ISS) [16]. It sorts all B*-substring within the interval [`first`, `last`] (line 310). ISS uses *Multikey Quicksort* (MKQS) [1] and *Heapsort* (HS). MKQS is used $\lfloor \lg(\texttt{last} - \texttt{first}) \rfloor$ times to sort an interval before HS is used (if there are still elements in the interval that have been equal to the pivot each time, see line 333). MKQS divides each interval into three subintervals with respect to a pivot element. The first subinterval contains all substrings whose k-th character is smaller than the pivot, the second subinterval contains all substrings whose k-th character is equal to the pivot, and the last subinterval contains all substrings whose k-th character is greater than the pivot. We call k the depth of the current iteration (line 332). ISS is not implemented recursively; instead, a stack is used to keep track of the unsorted subintervals and the smaller subintervals are always processed first. This guarantees a maximum stack size of $\lg \ell$, where ℓ is the initial interval size [15, p. 67]. The subintervals containing the substrings whose k-th character is not equal to the pivot are sorted using MKQS $\lfloor \lg(\texttt{last} - \texttt{first}) \rfloor$ times before using HS, where now `last` and `first` refer to the first and last positions of these intervals (lines 414 and 428).

Whenever an unsorted (sub)bucket is smaller than a threshold (8 in the default configuration), *Insertionsort* (IS) is used to sort the bucket and mark it sorted (line 326). Whenever we compare two B*-Substrings during IS, we use the function ss_compare that compares two B*-substrings starting at the current depth and compares the substrings character by character.

Throughout the sorting of the B*-substrings, substrings that cannot be fully sorted, i.e. B*-substrings that are equal, are marked by storing their bitwise negated reference (line 178). Only the first reference of such an interval is stored normally to identify the beginning of an interval of unsorted substrings (line 178). There are B*-suffixes that are not sorted completely by their initial B*-substrings e.g., in our example T= cdcdcdcdccdd$ the B*-substring cdcd occurs three times – see Figure 5. Therefore, we cannot determine the order of the corresponding B*-suffixes just using their initial B*-substring. The idea of sorting the suffixes in a (c0, c1)-bucket up to a certain depth is similar to the approach of Manzini and Ferragina [14], who sort the suffixes up to a certain LCP-value.

	$		c					d					
i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	3	0	$\tilde{1}$	$\tilde{2}$	4	0	0	0	0	2	4	6	9

(a)

Ref.	Text Pos.	B⋆-substring
3	6	cdcc
0	0	cdcd
1	2	cdcd
2	4	cdcd
4	9	cdd$

(b)

Figure 5: The lexicographically sorted references of the B⋆-substrings in SA[0..m − 1] (light gray in (a)). For readability we write $\tilde{i}$ if i is bitwise negated ($\tilde{i} < 0$ for all $0 \leq i \leq$ n). The content of the buckets is not changed in this step. The references, their corresponding text positions and the B⋆-substrings are shown in (b).

i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	**-1**	0	1	2	**-1**	3	3	3	0	4	4	6	9

Figure 6: ISAb contains the inverse suffix array of the sorted B⋆-substrings. ISAb[i] = SA[m + i] for all $0 \leq i <$ m (dark gray in SA). If m > $\frac{n}{3}$, ISAb overlaps with PAb. This does not matter, as we do not require the text positions at this point any more. While computing ISAb, we also mark completely sorted intervals in SA[0..m − 1]. The leftmost position of a sorted interval of length ℓ is changed to -ℓ (see SA[0] and SA[4] where we store -1 as the sorted intervals contain one entry).

Computing the Partial Inverse Suffix Array. After the B⋆-substrings are sorted, we compute the ISA for the partially sorted B⋆-substrings (lines 146 to 156). The inverse suffix array for the B⋆-suffixes is stored in SA[m..2m) and referred to as ISAb with ISAb[i] = SA[m + i]. ISAb[i] contains the *rank* of the i-th B⋆-suffix, i.e., the number of lexicographically smaller B⋆-suffixes. All references to line numbers in this subsection refer to divsufsort.c. We scan the SA[0..m) from right to left (line 146) and distinguish between bitwise negated references (values < 0, starting at line 154) and non-negated references (values ≥ 0, starting at line 147). In the first case, we have reached an interval where we have references of suffixes which could not be sorted comparing only the B⋆-substring. We assign each of those suffixes the greatest feasible rank, i.e., m − i, where i is the number of lexicographically greater suffixes (similar to Larsson and Sadakane [11]). In addition we also store the bitwise negation of the references, i.e., the original reference. In the other case (a value ≥ 0) we simply assign the correct rank to the B⋆-suffix. Whenever we scan an interval of completely sorted B⋆-suffixes, we mark the first position of the interval in SA[0..m) with $-k$, where k is the size of the interval (line 150). Now we can identify all sorted intervals as they start with a negative value whose absolute value is the length of the interval.

In our example (see Figure 6) we have two fully sorted intervals of length 1 at SA[0] and SA[4], and an only partially sorted interval in SA[1..3].

Sorting the B⋆-Suffixes. In the last part of the B⋆-suffix sorting in DivSufSort we compute the correct ranks of all B⋆-suffixes and store them in ISAb. During this step,

we only require information about the ranks of the suffixes and have no random access to the text, i.e., PAb is not required any more. All line numbers in this section refer to trsort.c. Using ISAb, we compute the ranks of all B*-suffixes using an approach similar to prefix doubling [11]. Instead of doubling the length of the suffixes we double the number of considered B*-substrings that can have an arbitrary length (line 563). Here, ISAd$[i]$ refers to the rank of the $i + 2^k$-th B*-suffix, where k is the current iteration of the doubling algorithm. Obviously, we need to update the ranks when we double the number of considered substrings, i.e., compute the new ranks for the B*-suffixes. Since the ranks in the ISA are given in text order, we can access the rank of the next (in text order) B*-substring for any given substring.

Repetition Detection. The sorting that uses the new ranks as keys is done using Quicksort (QS), which also allows us to use the *repetition* detection introduced by Maniscalco and Puglisi [13] (see line 452 for the identification and the function tr_copy for the computation of the correct ranks). A repetition in T is a substring $\mathsf{T}[i, i+rp]$ with $r \geq 2, p \geq 0$ and $i, i + rp \in [0, \mathsf{n})$ such that $\mathsf{T}[i, i + p) = \mathsf{T}[i + p, i + 2p) = \cdots = \mathsf{T}[i + (r - 1)p, i + rp)$. Those repetitions are a problem if S_i is a B*-suffix, since then S_{kp} is a $B^\star$-suffix for all $k \leq r$. We can simply sort all those suffixes by looking at the first character not belonging to the repetition ($\mathsf{T}[i+rp+l] \neq \mathsf{T}[i+l]$). If $\mathsf{T}[i+rp+l] < \mathsf{T}[i+l]$ then $\mathsf{T}[i+(r-1)p+1, i+rp] < \mathsf{T}[(i-1)+(r-1)p+1, (i-1)+rp]$ for all $1 < i \leq r$. The analogous case is true for $\mathsf{T}[i + rp + l] > \mathsf{T}[i + l]$, i.e., $\mathsf{T}[i+(r-1)p+1, i+rp] > \mathsf{T}[(i-1)+(r-1)p+1, (i-1)+rp]$ for all $1 < i \leq r$. This is done in lines 276 (and 282), where we increase (and decrease) the ranks of all suffixes in the repetition. The identification of a repetition is supported by QS. QS divides each interval into three subintervals (like MKQS). We chose the median rank of the B*-suffixes that are considered during this doubling step as the pivot element for QS (line 455). If the (current) rank of the first B*-suffix in the subinterval (considered in this doubling step) is equal to the pivot element, i.e., ISAb$[i]$ = ISAd$[i]$ where i is the first B*-suffix in the interval, then we have found a repetition (line 452, where tr_ilg denotes the logarithm, i.e., the number of iterations until HS is used instead of QS).

Now we have computed the ISA of all B*-suffixes (stored in ISAb), i.e., we have all B*-suffixes in lexicographic order. From this point on, all line numbers refer to divsufsort.c, again. Next (see loop starting at line 162), we scan T from right to left, and when we read the i-th B*-suffix at position j, we store j at position SA[ISAb$[i]$]. Since we use the B*-suffixes to induce the B-suffixes (and we do not want to induce A-suffixes during the first inducing phase) we store the bitwise negation of j if S_{j-1} has type A (line 167). Figures 7a and 7b show the transition in SA[0..m) for our example. Now, SA[0..m) contains the text positions of all B*-suffixes in lexicographic order. Next (see loop beginning at line 173), we need to put these text positions at their correct position in SA[0..n) (line 182). While doing so, we update BUCKET_B and BUCKET_BSTAR such that they contain the rightmost position of the corresponding buckets (lines 177 and 185). Figures 7c and 7d show this step for our running example.

3.3 Inducing the *A*- and *B*-suffixes

Due to the types of the suffixes, we know that in any (c0, c1)-bucket the A-suffixes are lexicographically smaller than the B-suffixes, and that B*-suffixes are lexicographically smaller than B-suffixes. We also know that in lexicographic order, all consecutive intervals of B-suffixes are left of at least one B*-suffix and all A-suffixes are right of at

i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	-1	-4	1	0	-1	3	2	1	0	4	4	6	9

(a)

i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	$\tilde{6}$	$\tilde{4}$	$\tilde{2}$	0	9	3	2	1	0	4	4	6	9

(b)

	$	c						d					
i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	$\tilde{6}$	$\tilde{4}$	$\tilde{6}$	$\tilde{4}$	$\tilde{2}$	0	9	1	0	4	4	6	9

(c)

	$	c	d	(c,c)	(c,d)
BUCKET_A	0	1	7	-	-
BUCKET_B	-	-	-	**1**	**6**
BUCKET_BSTAR	-	-	-	-	**1**

(d)

Figure 7: ISAb (dark gray in (a) and (b)) contains the ranks of all B*-suffixes. The lexicographically sorted text positions of the B*-suffixes are shown light gray in (b). Each text position i is bitwise negated if S_{i-1} has type A. In (c) all text positions of the B*-suffixes are at their correct position in SA[0..n − 1] (light gray). The buckets (d) contain the leftmost position of the corresponding suffixes.

least one B-suffix – see Figure 2. Now we scan SA twice: once from right to left where all B-suffixes are induced (we can skip all parts of SA containing only A-suffixes), and then from left to right to induce all A-suffixes (see Figure 8 for an example of the entire inducing process). All following line numbers refer to difsufsort.c. A step-by-step example is given in Figure 8.

During the inducing of the B-suffixes, i.e., the first scan of SA (see loop starting at line 205), whenever we read an entry i in SA such that $i > 0$ (line 211), we store the entry $i-1$ at the rightmost free position (a position in which a correct text position has not been stored yet) in the $(\mathsf{T}[i-1], \mathsf{T}[i])$-bucket (line 220). If $\mathsf{T}[i-2] > \mathsf{T}[i-1]$, then S_{i-2} is an A-suffix, which is not induced during the first scan, but the bitwise negated value of $i-1$ is stored instead (line 217). Every position is overwritten with its bitwise negated value. If the position was already bitwise negated, i.e., it has been induced and the corresponding suffix has type A, it is considered during the next scan (line 226) and it is ignored otherwise. After the first traversal, all suffixes that have been used for inducing are represented by their bitwise negated position whereas all other suffixes are represented by their position, i.e., a positive integer. It should be noted that all induced suffixes are lexicographically smaller than the suffix they are induced from: if we induce from a $(\mathsf{c0}, \mathsf{c1})$-bucket, we know that $\mathsf{c0} \leq \mathsf{c1}$, since we are considering B-suffixes. In addition, we can only induce in $(\mathsf{c0}, \mathsf{c1})$-buckets with $\mathsf{c1} \leq \mathsf{c0}$, as only B-suffixes are considered during this traversal.

Before SA is scanned a second time, $\mathsf{n}-1$ is stored at the beginning of the $\mathsf{T}[\mathsf{n}-1]$-bucket (line 234). If $S_{\mathsf{n}-2}$ has type A, we store $\mathsf{n}-1$ (we want to induce $S_{\mathsf{n}-2}$ during the second scan). Otherwise, we store the bitwise negation of $\mathsf{n}-1$.

During the second scan of SA (see loop starting at line 236), whenever an entry i of SA is smaller than 0 it is overwritten by its bitwise negated value, i.e., the

i	0	1	2	3	4	5	6	7	8	9	10	11	12
T[i]	c	d	c	d	c	d	c	d	c	c	d	d	$
SA[i]	12	8	6	4	2	0	9	11	7	5	3	1	10

Figure 9: The final SA of T= cdcdcdcdccdd$.

		Scanned Interval												BUCKET_A[$]	BUCKET_A[c]	BUCKET_A[d]	BUCKET_B[c, c]	BUCKET_BSTAR[c, d]	
i	0	1	2	3	4	5	6	7	8	9	10	11	12						
SA[i]	$\tilde{6}$	$\tilde{4}$	$\tilde{6}$	$\tilde{4}$	$\tilde{2}$	0	9	1	0	4	4	6	9	0	1	7	1	1	First Induction Phase
SA[i]	$\tilde{6}$	$\tilde{4}$	$\tilde{6}$	$\tilde{4}$	$\tilde{2}$	0	9	1	0	4	4	6	9	0	1	7	1	1	
SA[i]	$\tilde{6}$	**$\tilde{8}$**	$\tilde{6}$	$\tilde{4}$	$\tilde{2}$	0	**$\tilde{9}$**	1	0	4	4	6	9	0	1	7	**0**	1	
SA[i]	$\tilde{6}$	$\tilde{8}$	$\tilde{6}$	$\tilde{4}$	$\tilde{2}$	**$\tilde{0}$**	$\tilde{9}$	1	0	4	4	6	9	0	1	7	0	1	
SA[i]	$\tilde{6}$	$\tilde{8}$	$\tilde{6}$	$\tilde{4}$	**2**	$\tilde{0}$	$\tilde{9}$	1	0	4	4	6	9	0	1	7	0	1	
SA[i]	$\tilde{6}$	$\tilde{8}$	$\tilde{6}$	**4**	2	$\tilde{0}$	$\tilde{9}$	1	0	4	4	6	9	0	1	7	0	1	
SA[i]	$\tilde{6}$	$\tilde{8}$	**6**	4	2	$\tilde{0}$	$\tilde{9}$	1	0	4	4	6	9	0	1	7	0	1	
SA[i]	$\tilde{6}$	**8**	6	4	2	$\tilde{0}$	$\tilde{9}$	1	0	4	4	6	9	0	1	7	0	1	
SA[i]	**12**	8	6	4	2	$\tilde{0}$	$\tilde{9}$	1	0	4	4	6	9	0	1	7	0	1	
SA[i]	12	8	6	4	2	$\tilde{0}$	$\tilde{9}$	1	0	4	4	6	9	**1**	1	7	0	1	Second Induction Phase
SA[i]	12	8	6	4	2	$\tilde{0}$	$\tilde{9}$	**11**	0	4	4	6	9	1	1	**8**	0	1	
SA[i]	12	8	6	4	2	$\tilde{0}$	$\tilde{9}$	11	**7**	4	4	6	9	1	1	**9**	0	1	
SA[i]	12	8	6	4	2	$\tilde{0}$	$\tilde{9}$	11	7	**5**	4	6	9	1	1	**10**	0	1	
SA[i]	12	8	6	4	2	$\tilde{0}$	$\tilde{9}$	11	7	5	**3**	6	9	1	1	**11**	0	1	
SA[i]	12	8	6	4	2	$\tilde{0}$	$\tilde{9}$	11	7	5	3	**1**	9	1	1	**12**	0	1	
SA[i]	12	8	6	4	2	**0**	$\tilde{9}$	11	7	5	3	1	9	1	1	12	0	1	
SA[i]	12	8	6	4	2	0	**9**	11	7	5	3	1	9	1	1	12	0	1	
SA[i]	12	8	6	4	2	0	9	11	7	5	3	1	**10**	1	1	**13**	0	1	

Figure 8: During the first phase, we induce B-suffixes and only scan intervals where B- and B*-suffixes occur. Each of those intervals ends left of the succeeding c0-bucket. Its borders are stored in the corresponding BUCKET_BSTAR (boxed entries, the right border is not part of the interval). After the first phase we put the last suffix at the beginning of its corresponding bucket. During the second phase we scan the whole array, as we also store the bitwise negation of all entries that have already been used for inducing. The currently considered entry is marked light gray (■). The entries highlighted dark gray (■) are the positions where a value is induced. The bucket that contains the position is highlighted in the same color. Entries that have changed are bold in the following row.

position of the suffix in the correct position in the suffix array (line 249). Whenever $i > 0$ (line 237) the suffix S_{i-1} is induced at the leftmost free position in the $\mathsf{T}[i-1]$-bucket (line 243). Since all remaining suffixes are induced during this scan it is sufficient to identify the border using the $\mathsf{c0}$-buckets, i.e., the value stored in $\mathsf{BUCKET\_A}[\mathsf{c0}]$. If the induced suffix would induce a B-suffix, its bitwise negated value is induced instead (line 240). At the end of the traversal SA contains the indices of all suffixes in lexicographic order.

4 Inducing the LCP-Array

We now show how to modify DivSufSort such that it also computes the LCP-array in addition to SA. To do so, we extend DivSufSort at three points of the computation of SA. First, we need to compute the LCP-values of all B*-suffixes. Next, during the inducing step, we also induce the LCP-values for A- and B-suffixes. For this we utilize a technique also described in [4,2] that allows us to answer RMQs on LCP using only a stack [6]. Last, we compute the LCP-values of suffixes at the border of buckets, as those values cannot be induced.

Recall that the LCP-value of two arbitrary suffixes S_i and S_j is denoted by $\mathsf{lcp}(i,j)$. We need the following additional definition: Given an array A of length ℓ and $0 \leq i \leq j \leq \ell$, a *range minimum query* $\mathsf{RMQ}_A[i,j]$ asks for the minimum in A in the interval $[i,j]$, in symbols: $\mathsf{RMQ}_A[i,j] = \min\{A[k] \colon i \leq k \leq j\}$.

4.1 Computing the LCP-Values of the B*-Suffixes

During the sorting of the B*-suffixes (right before the B*-suffixes are put at their correct position in $\mathsf{SA}[0..\mathsf{n}]$), all lexicographically sorted B*-suffixes are in $\mathsf{SA}[0..\mathsf{m}]$. There are two cases regarding m (the number of B*-suffixes). If $\mathsf{m} > \frac{\mathsf{n}}{3}$, we have overwritten the text positions of the B*-suffixes in PAb with ISAb. In this case we must compute the LCP-values naively.[4] Otherwise (we still know the text positions of all B*-suffixes), we compute their LCP-values using a sparse version of the Φ-algorithm [8], based on Observation 2, which was also used implicitly in [4,2].

Observation 2 *If $S_i, S_{i'}, S_j$ and $S_{j'}$ are B*-suffixes such that $i < i', j < j'$ and there is no other B*-suffix S_k such that $i < k < i'$ or $j < k < j'$, then $\mathsf{lcp}(i',j') \geq \mathsf{lcp}(i,j) - (i'-i)$.*

This is possible as we know the distance (in the text) of two B*-suffixes, i.e., $\mathsf{PAb}[i] - \mathsf{PAb}[j]$ is the distance of the i-th and j-th B*-suffix with $1 \leq i \leq j \leq \mathsf{m}$. See Figure 10 for an Example. Algorithm 1 shows the *sparse* version of the Φ-algorithm. The difference to the original algorithm [8] is that the next considered suffix is an arbitrary number of character shorter than the previous one, which results in Observation 2. The computation of the LCP-values does not require any additional memory except for the n words for LCP, where we temporarily store additional data.

First (lines 1 to 4 of Algorithm 1), we fill the PHI (stored in $\mathsf{LCP}[\mathsf{m}..2\mathsf{m}]$) such that $\mathsf{PHI}[i]$ contains the text position of the suffix that is lexicographically consecutive to the i-th suffix (text position). In $\mathsf{DELTA}[i]$ (stored in $\mathsf{LCP}[\mathsf{n}-\mathsf{m}..\mathsf{n}]$) we store the text distance of the i-th and $(i+1)$-th B*-suffix (text occurrence), i.e., $\mathsf{PAb}[i+1] - \mathsf{PAb}[i]$. Then (lines 5 to 8), we compute the sparse LCP-array using Observation 2. As we store the LCP-values in PHI in text order, we need to rewrite them to LCP (line 9).

[4] For all tested instances (see Section 5) $\mathsf{m} \leq \frac{\mathsf{n}}{3}$.

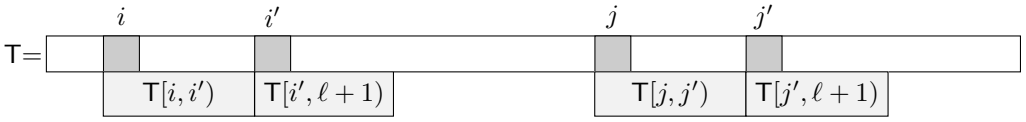


Figure 10: Let $S_i, S_j, S_{i'}$ and $S_{j'}$ be B*-suffixes such that there is no B*-suffix S_k with $i < k < i'$ or $j < k < j'$, and let the LCP-value of S_i and S_j be $\ell = \mathsf{lcp}(i, j) + i$. Then the LCP-value of $S_{i'}$ and $S_{j'}$ is $\mathsf{lcp}(i', j') = \ell - i' = \mathsf{lcp}(i, j) - (i' - i)$.

4.2 Inducing the LCP-Values in Addition to the SA

During the inducing of the B-suffixes, whenever a suffix is induced at position u in SA and there is already a suffix at position $u + 1$ in the same $(\mathsf{c0}, \mathsf{c1})$-bucket, there are two cases:

1. The suffixes $S_{\mathsf{SA}[u]}$ and $S_{\mathsf{SA}[u+1]}$ have been induced from suffixes $S_{\mathsf{SA}[v]}, S_{\mathsf{SA}[w]}$ in the same $(\mathsf{c0}, \mathsf{c1})$-bucket; in this case $\mathsf{LCP}[u + 1] = \mathsf{RMQ}_{\mathsf{LCP}}[v + 1, w] + 1$.
2. Otherwise, the LCP-value is either 1 or 2, depending on the c0-buckets $S_{\mathsf{SA}[v]}, S_{\mathsf{SA}[w]}$ are. If they are in the same bucket the LCP-value is 2 and 1 if not.

The computation of the LCP-values during the inducing of the A-suffixes works analogously. This leads to the following observation for the general case:

Observation 3 *Let* $\mathsf{SA}[u] = i, \mathsf{SA}[u + 1] = j, \mathsf{SA}[v] = i + 1$ *and* $\mathsf{SA}[w] = j + 1$ *such that* S_i *and* S_j *are in the same* c0*-bucket and* $u + 1 < v, w$ *or* $w, v < u$*. Then* $\mathsf{LCP}[u + 1] = \mathsf{RMQ}_{\mathsf{LCP}}[\min\{v, w\} + 1, \max\{v, w\}] + 1$.

Not all LCP-values can be induced this way. The missing cases are covered in the next section. Instead of using a dynamic RMQ data structure, we can answer the RMQs using a *min-stack* [2,4,6]. We only need to consider RMQs for suffixes from the same $(\mathsf{c0}, \mathsf{c1})$-bucket. To this end, we build the min-stack while scanning an interval $[\mathsf{first}, \mathsf{last}]$ (from right to left) of the LCP-array. An entry on the min-stack consist of tuple $\langle k, \mathsf{LCP}[k]\rangle$. Initially, the tuple $\langle \mathsf{n}, -1\rangle$ is on the min-stack. To update the min-stack at position $i \in [\mathsf{first}, \mathsf{last}]$ we look at the top of the min-stack and remove the tuple $\langle k, \mathsf{LCP}[k]\rangle$ if $\mathsf{LCP}[k] \geq \mathsf{LCP}[i]$. We repeat this process until no tuple is removed. Then we add $\langle i, \mathsf{LCP}[i]\rangle$ to the min-stack.

Now we want to answer $\mathsf{RMQ}_{\mathsf{LCP}}[i, j]$ with $\mathsf{first} \leq i < j \leq \mathsf{last}$. (It should be noted that at this point we have not added $\langle i, \mathsf{LCP}[i]\rangle$ to the min-stack or have removed

Algorithm 1: Sparse Φ-Algorithm

Input : T, m, SA, ISAb= SA[m..2m − 1], PAb= SA[n − m..n − 1] and LCP, PHI= LCP[m..2m − 1] PHI= DELTA[n − m..n − 1].
Output : LCP[0..m − 1] contains the LCP-values of the B*-suffixes.

```
1 PHI[SA[0]] = −1
2 for i = 1; i ≤ m − 1; i = i + 1 do
3 |   PHI[SA[i]] = SA[i − 1]
4 |   DELTA[i − 1] = PAb[i] − PAb[i + 1]
5 for i = 0, p = 0; i < m; i = i + 1 do
6 |   while T[PAb[i] + p + 1] = T[PAb[PHI[i]] + p + 1] do
7 |   |   p = p + 1
8 |   PHI[i] = p and p = max{0, p − DELTA[i]}
9 for i = 0; i < m; i = j + 1 do  LCP[ISAb[i]] = PHI[i];
```

any tuple from the min-stack in the process of adding it to the min-stack.) To this end, we scan the min-stack from top to bottom, until we find two consecutive tuples $\langle k, \mathsf{LCP}[k]\rangle$, $\langle k', \mathsf{LCP}[k']\rangle$ such that $k' > j$. Then, $\mathsf{RMQ}_{\mathsf{LCP}}[i,j] = \mathsf{LCP}[k]$. If we scan from left to right, the min-stack works analogously. The only difference is that the initial tuple is $\langle -1, -1\rangle$ and we search for the two consecutive tuples until $k' < j$

The min-stack is reseted whenever we arrive at a new $(\mathsf{c0}, \mathsf{c1})$-bucket, i.e., we only keep the $\langle \mathsf{n}, -1\rangle$-tuple. In the implementation, the min-stack is realized using a single array and a reference to its current top.

i	0	1	2	3	4	5	6
$\mathsf{A}[i]$	4	2	0	1	4	3	2

(a)

i	6	5	4	3	2	1	0
	$\langle 6,2\rangle$	$\langle 5,3\rangle$	$\langle 4,4\rangle$				
	$\langle \mathsf{n},-1\rangle$	$\langle 6,2\rangle$	$\langle 5,3\rangle$	$\langle 3,1\rangle$	$\langle 2,0\rangle$	$\langle 1,2\rangle$	$\langle 1,2\rangle$
		$\langle \mathsf{n},-1\rangle$	$\langle 6,2\rangle$	$\langle \mathsf{n},-1\rangle$	$\langle \mathsf{n},-1\rangle$	$\langle 2,0\rangle$	$\langle 2,0\rangle$
			$\langle \mathsf{n},-1\rangle$			$\langle \mathsf{n},-1\rangle$	$\langle \mathsf{n},-1\rangle$

(b)

Figure 11: The min-stack for each *current* position i (b) while scanning A (a) from right to left. A tuple (p, v) contains the position p of the value v. For the current position i the stack can be used to answer RMQs of the type $\mathsf{RMQ}_{\mathsf{A}}[i,j]$ with $j \geq i$ by looking at elements from the top until a position k with $k \geq j$ is found.

In addition to the min-stack, we require for each $\mathsf{c0}$-bucket the position of where the last suffix has been induced from. This is the position we look for when querying the min-stack.

4.3 Special Cases during LCP Induction

There are three special cases where the LCP-value cannot be induced using the min-stack (or RMQs in general). The first case occurs if a suffix is induced next to a B*-suffix. The inducing can happen to the left or right of the already placed B*-suffix. The former case is easy as there cannot be an A- or B-suffix to the left of a B*-suffix in the same $(\mathsf{c0}, \mathsf{c1})$-bucket. Therefore, we only need to check whether the suffixes are in the same $\mathsf{c0}$-bucket to compute the LCP-value for the B*-suffix, which is either 0 or 1. The other case (a suffix is induced to the right of a B*-suffix) is more demanding, as the LCP-value must be computed. Fortunately, this can be done more sophisticated than by naive comparison of the suffixes. First, we check whether both the B*-suffix S_i and the B-suffix S_j are in the same $(\mathsf{c0}, \mathsf{c1})$-bucket. If not, the LCP-value is 1 if they occur in the same $\mathsf{c0}$-bucket, and 0 otherwise. However, if they occur in the same $(\mathsf{c0}, \mathsf{c1})$-bucket, we know that S_i has a prefix $\mathsf{c0c1}d$, $d \in \Sigma$, such that $\mathsf{c0} < \mathsf{c1} \geq d$, and that S_j has a prefix $\mathsf{c0c1}e$, $e \in \Sigma$, such that $\mathsf{c0} < \mathsf{c1} \leq e$. Hence, the LCP-value is $\max\{k \geq 0 \colon \mathsf{T}[i+1, i+k+2) = \mathsf{T}[j+1, j+k+2)\} + 1$, i.e., the first appearance of a character not equal to $\mathsf{c1}$ in either suffix. In the last case (an A-suffix is induced next to a B-suffix) the LCP-value can be determined in an analogous way.

5 Experiments with LCP-Construction

We implemented the modified DivSufSort in C and compiled it using `gcc` version 6.2 with the compiler options `-DNDEBUG`, `-O3` and `-march=native`. Our implementation

is available from `https://github.com/kurpicz/libdivsufsort`. We ran all experiments on a computer equipped with an Intel Core i5-4670 processor and 16 GiB RAM, using only a single core.

We evaluated our algorithm on the *Pizza & Chili* Corpus[5] and compared our implementation to the following LCP-construction algorithms (using the same compiler options): *KLAAP* [9] is the first linear-time LCP-construction algorithm. The *Φ*-algorithm [8] is an alternative to KLAAP that reduces cache-misses. *Inducing+SAIS* [4] is an LCP-construction algorithm (using similar ideas as in this paper) based on SAIS [17], and *naive* scans the suffix array and checks two consecutive suffixes character by character.

We also looked at LCP-construction algorithms requiring the *Burrows-Wheeler transform*, i.e., *GO* and *GO2* by Gog and Ohlebusch [6]. Since these algorithms are only available in the *succinct data structure library* (SDSL) [5], which has an emphasis on a low memory footprint, the running times are affected by that.

The results of our experiments can be found in Table 1. As a brief summary, our practical tests show that *Φ* (see column 1) is the fastest LCP-construction algorithm if SA is already given, while our new implementation (column 6) is faster than the only other inducing-based approach (last 2 columns).

6 Conclusions

We presented a detailed description of *DivSufSort* that has not been available albeit its wide use in different applications. We linked interesting approaches, e.g., the repetition detection, to the corresponding lines in the source code and to the original literature.

Compared with SAIS, the other popular suffix array construction algorithm based on inducing, DivSufSort is faster. We ascribe this to the two main differences between DivSufSort and SAIS: First, the sorting of the initial suffixes in SAIS (the ones that cannot be induced) is done by recursively applying the algorithm (and renaming the initial suffixes), which is slower in practice than the string-sorting and prefix doubling-like approach used by DivSufSort (which also employs techniques like repetition detection to further decrease runtime). Second, the classification of the initial suffixes differs: while the suffixes that have to be sorted initially in SAIS can be displaced during the inducing of the SA, they are not moved again in DivSufSort. This also allows DivSufSort to skip parts (containing only A-suffixes) of the SA during the first induction phase.

In addition, we showed that the LCP-array can be computed during the inducing of the suffix array in DivSufSort. This approach is faster than the previous known inducing LCP-construction algorithm based on SAIS [4], and competitive with the *Φ*-algorithm, i.e, the fastest pure LCP-construction algorithms.

[5] `http://pizzachili.dcc.uchile.cl/`, last seen 05.07.2017

		LCP given SA (and BWT if necessary)							SA		SA + LCP	
	Text	Φ [8]	KLAAP [9]	naive	GO [6]	GO2 [6]	inducing [this paper]	inducing [4]	DivSufSort	SAIS [17]	inducing + DivSufSort [this paper]	inducing + SAIS [4]
20 MB	dna	**0.77**	0.91	1.180	6.46	2.65	0.78	1.12	1.45	1.71	2.23	2.83
	english	**0.61**	0.77	44.72	7.90	4.03	0.64	0.91	1.45	1.65	2.09	2.56
	dblp.xml	0.54	0.55	1.640	2.56	3.92	**0.53**	0.82	1.06	1.29	1.59	2.11
	sources	**0.54**	0.57	1.530	2.87	4.26	0.57	0.85	1.07	1.41	1.64	2.26
	proteins	**0.60**	0.67	4.190	5.46	3.24	0.66	0.96	1.51	1.79	2.17	2.75
50 MB	dna	**2.02**	2.360	3.240	16.25	14.43	2.06	2.96	3.88	4.57	5.94	7.53
	english	**1.70**	2.080	65.85	15.41	12.76	1.88	2.65	3.83	4.56	5.71	7.21
	dblp.xml	1.41	1.45	4.370	9.490	9.370	**1.39**	2.17	2.93	3.53	4.32	5.70
	sources	**1.45**	1.49	6.950	10.06	10.15	1.51	2.26	2.87	3.77	4.38	6.03
	proteins	**1.77**	2.01	6.560	14.38	15.74	1.87	2.83	4.55	5.27	6.42	8.10
100 MB	dna	**4.11**	4.75	6.590	26.03	26.62	4.24	5.95	8.23	9.44	12.47	15.39
	english	**3.56**	4.28	185.9	32.57	28.09	4.02	5.62	7.96	9.49	11.98	15.11
	dblp.xml	2.85	2.89	9.040	19.91	21.49	**2.82**	4.41	6.19	7.22	9.010	11.63
	sources	**2.93**	3.02	39.85	24.92	24.46	3.07	4.62	5.98	7.72	9.050	12.34
	proteins	**3.56**	4.09	16.99	30.89	28.12	3.96	5.86	9.91	10.96	13.87	16.82
200 MB	dna	**8.25**	10.0	17.36	76.11	79.02	8.64	12.02	17.41	19.18	26.05	31.20
	english	**7.23**	8.70	1070	72.58	73.75	8.25	11.49	16.80	19.39	25.05	30.88
	dblp.xml	**5.75**	6.28	18.23	49.97	52.91	5.77	9.120	12.99	14.72	18.76	23.84
	sources	**5.98**	6.23	52.60	61.61	59.01	6.37	9.700	12.63	16.01	19.00	25.71
	proteins	**6.86**	7.94	42.60	78.78	77.40	8.33	11.82	19.73	21.65	28.06	33.47

Table 1: The first seven columns contain the times solely for the computation of LCP. Since the inducing algorithms are interleaved with the computation of SA, we subtracted the time to compute SA with the corresponding inducing approach ("inducing [this paper]" and "inducing [4]"). *GO* and *GO2* require the BWT in addition to SA; the time to compute BWT is also not included. The last two columns show the time to compute SA and LCP using the inducing approach. All times are in seconds, and are the average over 21 runs on the same input.

References

1. J. L. Bentley and R. Sedgewick: *Fast algorithms for sorting and searching strings*, in SODA, ACM/SIAM, 1997, pp. 360–369.
2. T. Bingmann, J. Fischer, and V. Osipov: *Inducing suffix and lcp arrays in external memory.*, in ALENEX, SIAM, 2013, pp. 88–102.
3. J. Dhaliwal, S. J. Puglisi, and A. Turpin: *Trends in suffix sorting: A survey of low memory algorithms*, in Proc. ACSC, Australian Computer Society, 2012, pp. 91–98.
4. J. Fischer: *Inducing the LCP-array*, in Proc. WADS, vol. 6844 of LNCS, Springer, 2011, pp. 374–385.
5. S. Gog, T. Beller, A. Moffat, and M. Petri: *From theory to practice: Plug and play with succinct data structures*, in Proc. SEA, vol. 8504 of LNCS, Springer, 2014, pp. 326–337.
6. S. Gog and E. Ohlebusch: *Fast and lightweight LCP-array construction algorithms*, in Proc. ALENEX, SIAM, 2011, pp. 25–34.
7. H. Itoh and H. Tanaka: *An efficient method for in memory construction of suffix arrays*, in Proc. SPIRE/CRIWG, IEEE Press, 1999, pp. 81–88.
8. J. Kärkkäinen, G. Manzini, and S. J. Puglisi: *Permuted longest-common-prefix array*, in Proc. CPM, vol. 5577 of LNCS, Springer, 2009, pp. 181–192.
9. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in Proc. CPM, vol. 2089 of LNCS, Springer, 2001, pp. 181–192.
10. J. Labeit, J. Shun, and G. E. Blelloch: *Parallel lightweight wavelet tree, suffix array and fm-index construction*, in Data Compression Conference (DCC), IEEE, 2016, pp. 33–42.
11. N. J. Larsson and K. Sadakane: *Faster suffix sorting.* Theor. Comput. Sci., 387(3) 2007, pp. 258–272.
12. U. Manber and G. Myers: *Suffix arrays: a new method for on-line string searches.* siam Journal on Computing, 22(5) 1993, pp. 935–948.
13. M. A. Maniscalco and S. J. Puglisi: *An efficient, versatile approach to suffix sorting.* ACM J. Experimental Algorithmics, 12 2008, p. Article no. 1.2.
14. G. Manzini and P. Ferragina: *Engineering a lightweight suffix array construction algorithm.* Algorithmica, 40(1) 2004, pp. 33–50.
15. K. Mehlhorn: *Data Structures and Algorithms 1: Sorting and Searching*, vol. 1 of EATCS Monographs on Theoretical Computer Science, Springer, 1984.
16. D. R. Musser: *Introspective sorting and selection algorithms.* Softw., Pract. Exper., 27(8) 1997, pp. 983–993.
17. G. Nong, S. Zhang, and W. H. Chan: *Linear suffix array construction by almost pure induced-sorting*, in Proc. DCC, IEEE Press, 2009, pp. 193–202.
18. S. J. Puglisi, W. F. Smyth, and A. H. Turpin: *A taxonomy of suffix array construction algorithms.* ACM Comput. Surv., 39(2) 2007.

The Linear Equivalence of the Suffix Array and the Partially Sorted Lyndon Array

Frantisek Franek[1], Asma Paracha[1], and William F. Smyth[1,2]

[1] Department of Computing & Software
McMaster University, Hamilton, Canada
{franek/paracham/smyth}@mcmaster.ca
[2] School of Engineering & Information Technology
Murdoch University, Perth, Australia

Abstract. In 2015 Uwe Baier presented a linear-time algorithm that directly sorts the suffixes of a string, the first such algorithm that is not recursive. In fact, his approach implicitly gives quite a bit more: it includes a linear-time elementary algorithm for computing what turns out to be a partially sorted version of the Lyndon array, and then shows how this can be used to sort the suffixes. At the same time, it is known that the Lyndon array can be computed in linear time from the suffix array. This paper extends these aspects of Baier's work to establish the *linear equivalence* of certain orderings of maximal Lyndon substrings and of fully sorted suffixes. By this terminology we mean that each data structure can be transformed into the other by a simple linear-time computation.

Keywords: string; Lyndon string; suffix; suffix array; Lyndon array; sorted Lyndon array; partially sorted Lyndon array

1 Introduction

In [1,2] Baier described a linear-time algorithm to sort the suffixes of a string, the first such algorithm that is not recursive. In fact, his approach implicitly gives much more: it includes a linear-time elementary algorithm for computing a partially sorted version of the Lyndon array, and it shows how this partial sort can be used to yield a complete sort of the suffixes. (Baier does not in his paper or his thesis make explicit reference to Lyndon substrings or to the Lyndon array.) On the other hand, it is known that the regular (unsorted) Lyndon array can be computed in linear time from the suffix array [6,5]. Thus there is some sort of *linear equivalence* between certain orderings of the Lyndon array and the suffix array, a relationship that we make precise in this paper.

Baier's algorithm works in two phases: in the first phase the suffixes of the input string are distributed into "groups" (that actually correspond to a partial sort of entries in the Lyndon array); then in the second phase the suffix array of the input string is computed from the groups. This paper deals mainly with the second phase: we show that the groups of suffixes output by Baier's Phase 1 are in fact an arrangement of the maximal Lyndon substrings of the input string, and further that this arrangement leads naturally, in linear time, to the suffix array. We also show how to go in the reverse direction; that is, how to compute the groups from the suffix array.

In the next section we introduce the ideas and notation that we use — most importantly, precise definitions for various notions of "groups of suffixes". In Section 3 two main theorems are presented, showing the linear equivalence of a *partially sorted Lyndon array* and the *suffix array* of a string.

Frantisek Franek, Asma Paracha, William F. Smyth: The Linear Equivalence of the Suffix Array and the Partially Sorted Lyndon Array, pp. 77–84.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

2 Preliminaries

In the literature, *string* and *word* are used interchangeably. But *word* may be used to refer to a substring, as in: `let` k `be the length of the longest Lyndon word in` $\boldsymbol{x}$ `starting at position` i. The terms *subword*, *substring*, and *factor* are also often used interchangeably. To prevent confusion, we will strictly use *string* and *substring*.

We use the array notation for strings indexing from 1; that is, $\boldsymbol{x} = \boldsymbol{x}[1..n]$ indicates that string $\boldsymbol{x}$ consists of n alphabet symbols and thus has length n. Strings are given in bold to distinguish them from other entities such as numbers and alphabet symbols: for example, $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and $\boldsymbol{x}[i] = a$. The *length* of a string $\boldsymbol{x}$ is denoted by $|\boldsymbol{x}|$. An *empty string* of length 0 is denoted by ε. The notation $\boldsymbol{x}[i..j)$ is used as an abbreviation for $\boldsymbol{x}[i..j-1]$.

The symbol $\boldsymbol{xy}$ denotes the *concatenation* of $\boldsymbol{x}$, $\boldsymbol{y}$; in particular, $\boldsymbol{x}[1..n] = \boldsymbol{x}[1]\boldsymbol{x}[2]\cdots\boldsymbol{x}[n]$. If $\boldsymbol{x} = \boldsymbol{uvw}$, $\boldsymbol{u}$ is called a *prefix* of $\boldsymbol{x}$, $\boldsymbol{v}$ a *substring* of $\boldsymbol{x}$, $\boldsymbol{w}$ a *suffix* of $\boldsymbol{x}$. A prefix (substring, suffix) is *trivial* if it is empty, *proper* if not equal to $\boldsymbol{x}$.

The symbol $\mathcal{A}(\boldsymbol{x})$ denotes the *alphabet* of the string $\boldsymbol{x}$; that is, the set of all distinct symbols occurring in $\boldsymbol{x}$. A string $\boldsymbol{x}$ is said to be *over* an alphabet $\mathcal{B}$, denoted by $\boldsymbol{x} \in \mathcal{B}^*$, if $\mathcal{A}(\boldsymbol{x}) \subseteq \mathcal{B}$. If $\prec$ is a total order of $\mathcal{B}$, it can be naturally extended to a total *lexicographic* order of $\mathcal{B}^*$ (*lexorder* for short) by a simple rule: $\boldsymbol{x} \prec \boldsymbol{y}$ if either $\boldsymbol{x}$ is a proper prefix of $\boldsymbol{y}$, or $\boldsymbol{x}[1..j) = \boldsymbol{y}[1..j)$ and $\boldsymbol{x}[j] \prec \boldsymbol{y}[j]$ for some j, $1 < j \leq \min\{|\boldsymbol{x}|, |\boldsymbol{y}|\}$.

If a string $\boldsymbol{x} = \boldsymbol{uv}$, then $\boldsymbol{vu}$ is called a *rotation* of $\boldsymbol{x}$. The rotation is *trivial* if either $\boldsymbol{u}$ or $\boldsymbol{v}$ is empty. A string $\boldsymbol{x}$ is *Lyndon* [3] if $\boldsymbol{x} \prec \boldsymbol{y}$ for any non-trivial rotation of $\boldsymbol{x}$, where as above we suppose a total order $\prec$ on $\mathcal{A}(\boldsymbol{x})$. Clearly any string of length 1 is Lyndon, thus called a *trivial* Lyndon string.

Observation 1 ([4,8]) *For any $\boldsymbol{x} = \boldsymbol{x}[1..n]$, $n > 1$, the following are equivalent:*

1. *$\boldsymbol{x}$ is a non-trivial Lyndon string;*
2. *$\boldsymbol{x}[1..n] \prec \boldsymbol{x}[k..n]$ for any $1 < k \leq n$;*
3. *$\boldsymbol{x}[1..k) \prec \boldsymbol{x}[k..n]$ for any $1 < k \leq n$;*
4. *there is $1 < k \leq n$ so that $\boldsymbol{x}[1..k) \prec \boldsymbol{x}[k..n]$, both $\boldsymbol{x}[1..k)$ and $\boldsymbol{x}[k..n]$ are Lyndon.*

Item 4 of this observation is the basis for the definition of the *standard factorization* of a Lyndon string $\boldsymbol{x}$, given by $\boldsymbol{x}[1..k)\boldsymbol{x}[k..n]$ where k is the smallest integer such that $\boldsymbol{x}[1..k)$ and $\boldsymbol{x}[k..n]$ are both Lyndon.

A string is *primitive* if it is not a concatenation of two or more copies of a smaller string. A *border* of a string $\boldsymbol{x}$ is a prefix that is also a suffix; a border is *trivial* if it is empty, *proper* if it is not $\boldsymbol{x}$ itself. If $\boldsymbol{x}$ has only trivial or improper borders, it is said to be *unbordered*.

Observation 2 *For any string $\boldsymbol{x}$: $\boldsymbol{x}$ is Lyndon $\not\Rightarrow$ unbordered $\not\Rightarrow$ primitive.*

Suppose that a substring $\boldsymbol{u}$ of $\boldsymbol{x}$ is Lyndon. Then $\boldsymbol{u}$ is said to be *maximal Lyndon in $\boldsymbol{x}$* if it is not a proper prefix of any Lyndon substring of $\boldsymbol{x}$. Occasionally, we may abbreviate *maximal Lyndon* as *maxLyn*.

Theorem 1 ([6], Hohlweg and Reutenauer) *Any substring $\boldsymbol{x}[i..k]$ of string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is maximal Lyndon if and only if $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[j..n]$ for any j satisfying $i < j \leq k$, and either $k = n$ or $\boldsymbol{x}[k+1..n] \prec \boldsymbol{x}[i..n]$.*

The Lyndon array was introduced in [5] — it is closely related to the *Lyndon tree* of [6]:

Definition 1 *For a given string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, the* Lyndon array *of $\boldsymbol{x}$ is an integer array $L[1..n]$ such that $L[i] = j$ if and only if j is the length of the maximal Lyndon substring at i.*

We now introduce the *Lyndon grouping array*, the *partially sorted Lyndon array*, and the *sorted Lyndon array*. All three are two-dimensional arrays $\mathcal{L}[1..2][1..n]$, but for brevity we use $\mathcal{L}_1[i]$ instead of $\mathcal{L}[1][i]$, $\mathcal{L}_2[i]$ instead of $\mathcal{L}[2][i]$.

Definition 2 *(See Figure 1.) Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ be a string of length n. The* Lyndon grouping array *of $\boldsymbol{x}$ is a two-dimensional integer array $\mathcal{L}[1..2][1..n]$ such that*

1. *$\mathcal{L}_1[1..n]$ is a permutation of $1..n$;*
2. *if $\mathcal{L}_2[i] > 0$, then the maximal Lyndon substring starting at $\mathcal{L}_1[i]$ has length $\mathcal{L}_2[i]$;*
3. *if $\mathcal{L}_2[i] = 0$, then the maximal Lyndon substring starting at $\mathcal{L}_1[i]$ has length $\mathcal{L}_2[j]$ where j is the greatest integer less than i such that $\mathcal{L}_2[j] > 0$.*

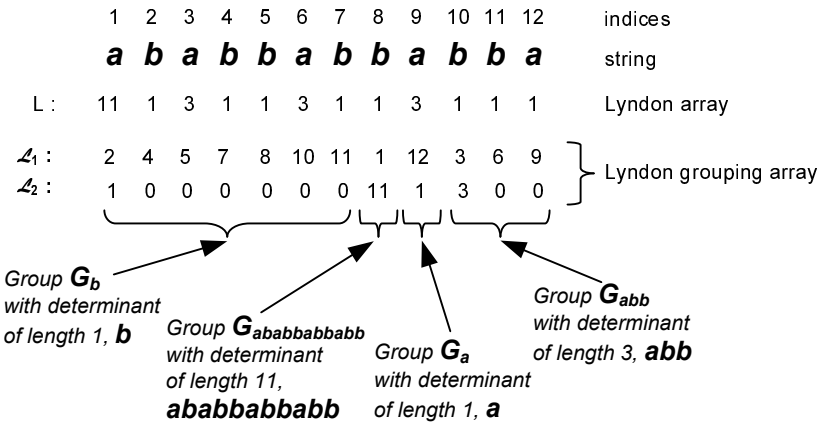


Figure 1. A Lyndon grouping array for ***ababbabbabba***

Thus, a Lyndon grouping array just partitions the positions of a string into groups determined by identical maxLyn substrings: all indices in the same group are starting positions of the same maxLyn substring. Baier calls this substring the *context* of the group [1,2] — here we will use the term *determinant*; we denote a group with a determinant $\boldsymbol{u}$ as $G_{\boldsymbol{u}}$. Note that the Lyndon grouping array is not unique; that is, for given $\boldsymbol{x}$ there may exist several such arrays with different orderings.

Lemma 1 *Let $\mathcal{L}[1..2][1..n]$ be a Lyndon grouping array of $\boldsymbol{x} = \boldsymbol{x}[1..n]$. Then the Lyndon array L of $\boldsymbol{x}$ can be computed from $L[1..2]$ in $\emptyset(n)$ steps.*

Proof. Replacing zeros in $\mathcal{L}_2$ with the value at the start of each group yields $L[\mathcal{L}_1[i]] = \mathcal{L}_2[i]$ for all $i \in 1..n$:

for $i = 1$ **to** n **do**
  if $\mathcal{L}_2[i] \neq 0$ **then** $m \leftarrow \mathcal{L}_2[i]$
  $L[\mathcal{L}_1[i]] \leftarrow m$

□

Note that the Lyndon array may provide weaker information than a Lyndon grouping array, as the Lyndon array can be computed from a Lyndon grouping array in linear time, but we do not know at this point how to compute in linear time a Lyndon grouping array from the Lyndon array.

Definition 3 *(See Figure 2.) A* partially sorted Lyndon array *of* $\boldsymbol{x}$ *is a Lyndon grouping array whose groups are sorted in ascending lexorder; that is,*

4. *For* $i < j$ *such that* $\mathcal{L}_2[i] > 0$, $\mathcal{L}_2[j] > 0$, $\boldsymbol{x}\big[\mathcal{L}_1[i]..\mathcal{L}_1[i]+\mathcal{L}_2[i]-1\big] \prec \boldsymbol{x}\big[\mathcal{L}_1[j]..\mathcal{L}_1[j]+\mathcal{L}_2[j]-1\big]$.

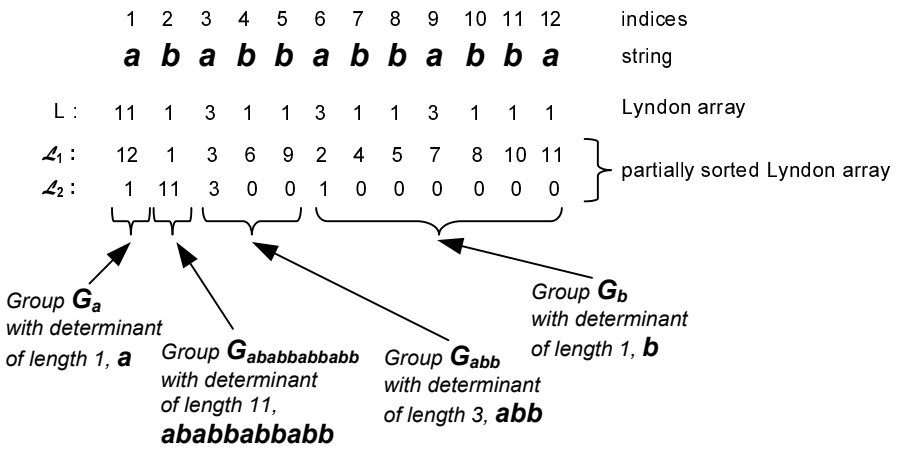


Figure 2. A partially sorted Lyndon array for ***ababbabbabba***

In Figure 2 the determinants a, $ababbabbabb$, abb, and b of the groups are sorted in ascending lexorder. However, the indices within the groups need not be in any particular order, though in our example they happen to fall in ascending order of position. Like the Lyndon grouping array, a partially sorted Lyndon array may not be unique.

Definition 4 *(See Figure 3.) A* sorted Lyndon array *of* $\boldsymbol{x}$ *is a partially sorted Lyndon array whose indices are ordered within each group in the* perfect order *according to the lexorder of the corresponding suffixes; that is,*

5. *If* $\mathcal{L}_1[i]$ *and* $\mathcal{L}_1[j]$ *belong to the same group,* $i < j \iff \boldsymbol{x}\big[\mathcal{L}_1[i]..n\big] \prec \boldsymbol{x}\big[\mathcal{L}_1[j]..n\big]$.

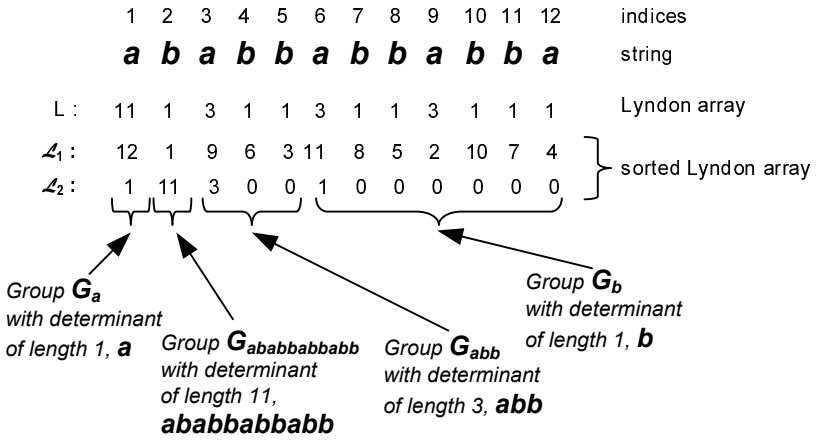


Figure 3. The sorted Lyndon array of ***ababbabbabba***

Definition 5 *(See Figure 4.)*

(a) Given $\boldsymbol{x} = \boldsymbol{x}[1..n]$*, the integer array* $SA[1..n]$ *is the* suffix array *of* $\boldsymbol{x}$ *iff the entries of* SA *form a permutation of* $1..n$ *and for every* $1 \leq i < n$, $\boldsymbol{x}\big[SA[i]..n\big] \prec \boldsymbol{x}\big[SA[i{+}1]..n\big]$.

(b) The lcp array *associated with* SA *is an integer array* $lcp[1..n]$ *in which* $lcp[i]$ *is the size of the longest common prefix of* $\boldsymbol{x}\big[SA[i]..n\big]$ *and* $\boldsymbol{x}\big[SA[i{-}1]..n\big]$ *for any* $1 < i \leq n$.

(c) The inverse suffix array $ISA[1..n]$ *is an integer array such that* $SA[i] = j$ *iff* $ISA[j] = i$.

Note that if $\mathcal{L}[1..2][1..n]$ is a sorted Lyndon array of $\boldsymbol{x}$, then in fact $\mathcal{L}_1[1..n]$ is the suffix array of $\boldsymbol{x}$. Thus, a sorted Lyndon array is unique, unlike a Lyndon grouping array and a partially sorted Lyndon array. Therefore we speak of **the** sorted Lyndon array of $\boldsymbol{x}$.

	1	2	3	4	5	6	7	8	9	10	11	12	
	a	**b**	**a**	**b**	**b**	**a**	**b**	**b**	**a**	**b**	**b**	**a**	string
SA :	12	1	9	6	3	11	8	5	2	10	7	4	suffix array
lcp :	-	1	2	4	6	0	2	5	8	1	3	6	lcp array
ISA :	2	9	5	12	8	4	11	7	3	10	6	1	inverse suffix array

Figure 4. suffix array, inverse suffix array, and lcp array of ***ababbabbabba***

3 Main Results

In this section we present the two main results tying together partially sorted Lyndon arrays and the suffix array of a string.

Theorem 2 *Let* $SA[1..n]$ *be the suffix array of a string* $\boldsymbol{x} = \boldsymbol{x}[1..n]$*. The sorted Lyndon array of* $\boldsymbol{x}$ *can be computed from* $\boldsymbol{x}$ *and* SA *in* $\mathcal{O}(n)$ *steps.*

Proof. As just observed, the top array $\mathcal{L}_1[1..n]$ is exactly the suffix array of $\boldsymbol{x}$. Thus we need only compute $\mathcal{L}_2[1..n]$. First we compute the inverse suffix array ISA from SA in $\mathcal{O}(n)$ steps. Then, as noted in [6] and explained in [5], we compute the Lyndon array $L[1..n]$ of $\boldsymbol{x}$ from ISA, also in $\emptyset(n)$ steps, using the next smaller value (NSV) algorithm. Thus we set $\mathcal{L}_2[i] = L\big[\mathcal{L}_1[i]\big]$ for every i.

To complete the calculation, we need only set the $\mathcal{L}_2$ values to zero except for the first entry in each group. For that we can use the $\mathcal{O}(n)$-time algorithm of Kasai *et al.* [7] to compute the lcp array. Then, for every i, if $lcp(\mathcal{L}_1[i], \mathcal{L}_1[i+1]) \geq \mathcal{L}_2[i]$ and $\mathcal{L}_2[i-1] = \mathcal{L}_2[i]$, we change the value of $\mathcal{L}_2[i]$ to 0. □

The reversed calculation is in essence Baier's Phase 2 algorithm. However, we will describe a different algorithm based on the same ideas. Though it is more complex to implement and requires more working memory than Baier's, it has the potential to be faster. This statement has not been verified by empirical testing, it is just based on the analysis of the implementation of the two algorithms. The actual testing will require to excise the second step from Uwe Baier's implementation.

The several following definitions are introduced only for use in the proof of Lemma 2. Thus they are not presented formally. We give them here because they are too complex to be included in the proof itself.

The *delta operator* is defined as follows: for $i \in G_{\boldsymbol{u}}$, $\Delta(i) = i+|\boldsymbol{u}|$. If $\Delta(i) \leq n$, then consider $\boldsymbol{v}$, the maxLyn substring at the position $\Delta(i)$. If $\boldsymbol{u}$ were lexicographically smaller than $\boldsymbol{v}$, then $\boldsymbol{uv}$ would be Lyndon, contradicting the maximality of $\boldsymbol{u}$. Thus, $\boldsymbol{v} \preceq \boldsymbol{u}$. It follows that for $i \in G_{\boldsymbol{u}}$ $\begin{cases} \Delta(i) = n+1, \text{ or} \\ \Delta(i) \in G_{\boldsymbol{v}} \text{ for some maxLyn } \boldsymbol{v} \prec \boldsymbol{u}, \text{ or} \\ \Delta(i) \in G_{\boldsymbol{u}}. \end{cases}$
The groups form a partition of the set of indices. Through the delta operator we define the Δ-refinement of this partition: let $\boldsymbol{u}$, $\boldsymbol{v}$ be maxLyn substrings of $\boldsymbol{x}$ so that $\boldsymbol{v} \preceq \boldsymbol{u}$, then we define the *subgroup* $G^{\boldsymbol{v}}_{\boldsymbol{u}} = \{i \in G_{\boldsymbol{u}} : \Delta(i) \in G_{\boldsymbol{v}}\}$, while we define the *subgroup* $G^{\$}_{\boldsymbol{u}} = \{i \in G_{\boldsymbol{u}} : \Delta(i) = n+1\}$.

It follows that each group $G_{\boldsymbol{u}}$ is a disjoint union of non-empty subgroups $G^{\boldsymbol{v}}_{\boldsymbol{u}}$ for all maxLyn $\boldsymbol{v} \preceq \boldsymbol{u}$ and possibly $G^{\$}_{\boldsymbol{u}}$. If $i \in G^{\boldsymbol{v}_1}_{\boldsymbol{u}}$ and $j \in G^{\boldsymbol{v}_2}_{\boldsymbol{u}}$, and $\boldsymbol{v}_1 \prec \boldsymbol{v}_2 \preceq \boldsymbol{u}$, then $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[j..n]$, as $\boldsymbol{x}[i..n] = \boldsymbol{u}\boldsymbol{v}_1\boldsymbol{w}_1$ for some $\boldsymbol{w}_1$, and $\boldsymbol{x}[j..n] = \boldsymbol{u}\boldsymbol{v}_2\boldsymbol{w}_2$ for some $\boldsymbol{w}_2$. Since $|G^{\$}_{\boldsymbol{u}}| \leq 1$, if $i \in G^{\$}_{\boldsymbol{u}}$ and $i \neq j \in G_{\boldsymbol{u}}$, then $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[j..n]$, as $\boldsymbol{x}[i..n] = \boldsymbol{u}$ and $\boldsymbol{x}[j..n] = \boldsymbol{u}\boldsymbol{w}$ for some $\boldsymbol{w}$. Thus, if we separately perfectly order the subgroup $G^{\boldsymbol{v}}_{\boldsymbol{u}}$ for each maxLyn $\boldsymbol{v} \prec \boldsymbol{u}$, then the group $G_{\boldsymbol{u}}$ will be perfectly ordered, as an important property of each subgroup $G^{\boldsymbol{v}}_{\boldsymbol{u}}$, $\boldsymbol{v} \prec \boldsymbol{u}$, is the fact that a perfect order of the group $G_{\boldsymbol{v}}$ induces a perfect order on $G^{\boldsymbol{v}}_{\boldsymbol{u}}$: we simply let i precede j only if $\Delta(i)$ precedes $\Delta(j)$. Similarly, a perfect order of $G^{1}_{\boldsymbol{u}}$, which is defined as the disjoint union of all subgroups of $G_{\boldsymbol{u}}$ except $G^{\boldsymbol{u}}_{\boldsymbol{u}}$, induces a perfect order on $G^{\boldsymbol{u}}_{\boldsymbol{u}}$.

For example, consider $\boldsymbol{x} = abb\,abb\,aa\,abb\,abb\,abb$ with $G_{abb} = \{1, 4, 9, 12, 15\}$. $G^{\$}_{abb} = \{15\}$, $G^{aaabbabbabb}_{abb} = \{4\}$, $G^{abb}_{abb} = \{1, 9, 12\}$ and $G_{abb} = G^{\$}_{abb} \cup G^{aaabbabbabb}_{abb} \cup G^{abb}_{abb}$. A perfect order of $G^{\$}_{abb}$ is 15, a perfect order of $G^{aaabbabbabb}_{abb}$ is 4. The elements of $G^{\$}_{abb}$ will be listed first, the elements of $G^{aaabbabbabb}_{abb}$. The perfect order of $G^{\$}_{abb} \cup G^{aaabbabbabb}_{abb} = \{15, 4\}$ determines the order of $G^{abb}_{abb} = \{1, 9, 12\}$. Now, $\Delta(1) = 1+3 = 4$, $\Delta(9) = 9+3 = 12$, and $\Delta(12) = 12+3 = 15$. Thus, 12 goes before 1, and then goes 9, i.e. the perfect order of G_{abb} is 15, 4, 12, 1, 9.

Lemma 2 *Let $\mathcal{L}[1..2][1..n]$ be a partially sorted Lyndon array of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$. Then in $\mathcal{O}(n)$ steps we can order the items in the groups to obtain the sorted Lyndon array.*

We only present here a sketch of the proof, as a complete proof would require an analysis of the code of the algorithm and so would exceed the scope of this contribution. However, for the interested reader, a `C++` implementation is available at `http://www.cas.ca/~franek/research.html/ub.cpp` for viewing, analysis, and testing.

Proof. We can achieve the desired ordering of $\mathcal{L}[1..2][1..n]$ by computing the suffix array SA of $\boldsymbol{x}$ and copying it into $\mathcal{L}_1[1..n]$.

First we compute triples $(I[i], G[i], SG[i])$ for $i \in 1..n$, where $I[i] = \mathcal{L}_1[i]$, $G[i]$ represent group (we are using integers $1..n$ to represent groups, and using $\Delta(i)$ we compute the subgroups (we are using integers $0..n$ to represents the subgroups). This can be achieved in two traversals.

Then we use a radix sort to sort the triples to be ascending in G and within each group to be ascending in SG. This can be achieved in six traversals.

In two traversals we can compute the inverse Δ relation, i.e. $i \in \Delta^{-1}(j)$ iff $\Delta(i) = j$.

Then we traverse the inverse Δ relation Δ^{-1} and record the indices as we encounter them. As explained in the text before this lemma, the perfect order of the previous groups induces a perfect order on the current group via the Δ operator. □

Theorem 3 *Let $\mathcal{L}[1..2][1..n]$ be a partially sorted Lyndon array of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$. The suffix array $SA[1..n]$ of $\boldsymbol{x}$ can be computed from $\boldsymbol{x}$ and $\mathcal{L}$ in $\mathcal{O}(n)$ steps.*

Proof. Using Lemma 2, we can compute the sorted Lyndon array $\mathcal{L}[1..2][1..n]$ of $\boldsymbol{x}$ by perfectly ordering $\mathcal{L}$. As previously noted, $\mathcal{L}[1][1..n]$ is then the suffix array of $\boldsymbol{x}$. □

4 Conclusion

Three arrays — *Lyndon grouping array*, *partially sorted Lyndon array*, *sorted Lyndon array* — have been introduced to formalize the notion of what is meant by *sorting the maximal Lyndon substrings*. The mutual relationship of these arrays has been examined and we have shown in what way the sorting of all maximal Lyndon substrings and sorting of suffixes of a string relate to each other.

Uwe Baier observed in [1,2], that his algorithm was slower than the state-of-the-art suffix sorting algorithms. He ascribed that to the early stages of the existence of his non-recursive approach and conjectured that with time, the approach would become more refined and thus faster. In essence, Phase 1 in Uwe Baier's algorithm is a direct construction of a partially sorted Lyndon array, which in Phase 2 is perfectly ordered to give the suffix array. The proof of Theorem 2 actually shows how much extra work is needed to get from the suffix array to a sorted Lyndon array. Thus, it seems to us that computing a partially sorted Lyndon array is essentially a harder task than "plain sorting" of the suffixes. So, maybe, no algorithm for computing a partially sorted Lyndon array can be as fast as sorting of suffixes, which in no way detracts from Uwe Baier's discovery of the deep connection hitherto unnoticed between the order of maximal Lyndon substrings and the order suffixes of a string.

In the diagram in Fig. 5, the arrow represent "simple linear computation". The diagram summarizes the relationships among the various arrays we were investigating. The two arrows with ? represent open questions: *Can a Lyndon array be used in a simple linear computation to compute a Lyndon grouping array?* and *Can a Lyndon grouping array be used in a simple linear computation to compute a sorted Lyndon array?* Note that Phase 1 of Uwe Baier's algorithm basically says **Yes** to both these questions. However it is not using any Lyndon array or Lyndon grouping array, it just computes it directly from the string. Maybe, having a Lyndon array or Lyndon grouping array can simplify the computation. From our point of view, having been interested in computation of Lyndon arrays, answer to the first question is much more interesting.

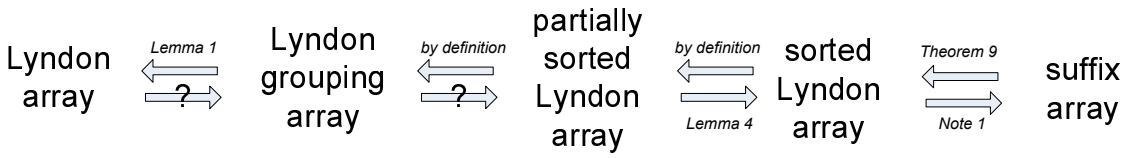


Figure 5.

Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada Discovery Grants (Franek, Smyth). The authors wish to thank Christoph Diegelmann who in reaction to questions posed in [5] brought the work of Uwe Baier to their attention and indicated that Phase 1 of Baier's algorithm in fact computes the Lyndon array of the input string.

References

1. U. Baier: *Linear-time suffix sorting — a new approach for suffix array construction.* M.Sc. Thesis, University of Ulm, 2015.
2. U. Baier: *Linear-time suffix sorting — a new approach for suffix array construction*, in 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016), R. Grossi and M. Lewenstein, eds., vol. 54 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2016, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 23:1–23:12.
3. K. T. Chen, R. H. Fox, and R. C. Lyndon: *Free differential calculus. iv. the quotient groups of the lower central series.* Annals of Mathematics, 68(1) 1958, pp. 81–95.
4. J.-P. Duval: *Factorizing words over an ordered alphabet.* J. Algorithms, 4(4) 1983, pp. 363–381.
5. F. Franek, A. S. Islam, M. S. Rahman, and W. Smyth: *Algorithms to compute the Lyndon array*, in Proceedings of Prague Stringology Conference 2016, PSC'16, 2016, pp. 172–184.
6. C. Hohlweg and C. Reutenauer: *Lyndon words, permutations and trees.* Theoretical Computer Science, 307(1) 2003, pp. 173 – 178, {WORDS}.
7. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in Combinatorial Pattern Matching: 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1–4, 2001 Proceedings, A. Amir, ed., Berlin, Heidelberg, 2001, Springer Berlin Heidelberg, pp. 181–192.
8. B. Smyth: *Computing Patterns in Strings*, Pearson Addison-Wesley, 2003.

Faster Batched Range Minimum Queries

Szymon Grabowski and Tomasz Kowalski

† Lodz University of Technology, Institute of Applied Computer Science,
Al. Politechniki 11, 90–924 Łódź, Poland,
{sgrabow|tkowals}@kis.p.lodz.pl

Abstract. Range Minimum Query (RMQ) is an important building brick of many compressed data structures and string matching algorithms. Although this problem is essentially solved in theory, with sophisticated data structures allowing for constant time queries, there are scenarios in which the number of queries, q, is rather small and given beforehand, which encourages to use a simpler approach. A recent work by Alzamel et al. starts with contracting the input array to a much shorter one, with its size proportional to q. In this work, we build upon their solution, speeding up handling small batches of queries by a factor of 3.8–7.8 (the gap grows with q). The key idea that helped us achieve this advantage is adapting the well-known Sparse Table technique to work on blocks, with speculative block minima comparisons. We also propose an even much faster (but possibly using more space) variant without the array contraction.

Keywords: string algorithms, range minimum query, bulk queries

1 Introduction

The Range Minimum Query (RMQ) problem is to preprocess an array so that the position of the minimum element for an arbitrary input interval (specified by a pair of indices) can be acquired efficiently. More formally, for an array $A[1 \dots n]$ of objects from a totally ordered universe and two indices i and j such that $1 \leq i \leq j \leq n$, the range minimum query $\mathsf{RMQ}_\mathsf{A}(i, j)$ returns $\mathrm{argmin}_{i \leq k \leq j} A[k]$, which is the position of a minimum element in $A[i \dots j]$. One may alternatively require the position of the leftmost minimum element, i.e., resolve ties in favour of the leftmost such element, but this version of the problem is not widely accepted. In the following considerations we will assume that A contains integers.

This innocent-looking little problem has quite a rich and vivid history and perhaps even more important applications, in compressed data structures in general, and in text processing in particular. Solutions for RMQ which are efficient in both query time and preprocessing space and time are building blocks in such succinct data structures as, e.g., suffix trees, two-dimensional grids or ordinal trees. They have applications in string mining, document retrieval, bioinformatics, Lempel-Ziv parsing, etc. For references to these applications, see [5,4].

The RMQ problem history is related to the LCA (lowest common ancestor) problem defined for ordinal trees: given nodes u and v, return $LCA(u, v)$, which is the lowest node being an ancestor of both u and v. Actually, the RMQ problem is linearly equivalent to the LCA problem [7,3], by which we mean that both problems can be transformed into each other in time linearly proportional to the size of the input. It is relatively easy to notice that if the depths of all nodes of tree T visited during an Euler tour over the tree are written to array A, then finding the LCA of nodes u and v is equivalent to finding the minimum in the range of A spanned between the first visits to u and v during the Euler tour (cf. [3, Observation 4]). Harel and Tarjan [10]

Szymon Grabowski, Tomasz Kowalski: Faster Batched Range Minimum Queries, pp. 85–95.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

were the first to give $O(n)$-time tree preprocessing allowing to answer LCA queries in constant time. The preprocessing required $O(n)$ words of space. A significantly simpler algorithm was proposed by Bender and Farach [3], with the same time and space complexities. Further efforts were focused on reducing the space of the LCA/RMQ solution, e.g. Sadakane [11] showed that LCAs on a tree of n nodes can be handled in constant time using only $2n + o(n)$ bits. A crowning achievement in this area was the algorithm of Fischer and Heun [5], who showed that RMQs on A can be transformed into LCA queries on the succinct tree, and this leads to an RMQ solution that also uses $2n + o(n)$ bits and (interestingly) does not access A at query time.

The Fischer and Heun solution, although allowing for constant time RMQ queries, is not so efficient in practice: handling one query takes several microseconds (see [4]). Some ingenious algorithmic engineering techniques, by Grossi and Ottaviano [9] and by Ferrada and Navarro [4], were proposed to reduce this time, but even the faster of these two [4] achieves about 2μs per query[1].

Very recently, Alzamel et al. [2] (implicitly) posed an interesting question: why should we use any of these sophisticated data structures for RMQ when the number of queries is relatively small and building the index (even in linear time, but with a large constant) and answering then the queries (even in constant time each, but again with a large constant) may not amortize? A separate, but also important point is that if we can replace a heavy tool with a simpler substitute (even if of limited applicability), new ideas may percolate from academia to software industry. Of course, if the queries $[\ell_i, r_i]$ are given one by one, we cannot answer them faster than in the trivial $O(r_i - \ell_i + 1) = O(n)$ time for each, but the problem becomes interesting if they are known beforehand. The scenario is thus offline (we can also speak about *batched queries* or *bulk queries*). Batched range minima (and batched LCA queries) have applications in string mining [6], text indexing and various non-standard pattern matching problems, for details see [2, Section 5].

As the ideas from Alzamel et al. [2] are a starting point for our solution and we directly compete with them, we dedicate the next section to presenting them.

We use a standard notation in the paper. All logarithms are of base 2. If not stated otherwise, the space usage is expressed in words.

2 The Alzamel et al. algorithm

Following [1] (see the proof of Lemma 2), the Alzamel et al. approach starts from contracting the array A into $O(q)$ entries. The key observation is that if no query starts or ends with an index i and $i + 1$, then, if $A[i] \neq A[i + 1]$, $\max(A[i], A[i + 1])$ will not be the answer to any of the queries from the batch. This can be generalized into continuous regions of A. Alzamel et al. mark the elements of A which are either a left or a right endpoint of any query and create a new array A_Q: for each marked position in A its original value is copied into A_Q, while each maximal block in A that does not contain a marked position is replaced by a single entry, its minimum. The relative order of the elements copied from A is preserved in A_Q, that is, in A_Q the marked elements are interweaved with representatives of non-marked regions between them. As each of q queries is a pair of endpoints, A_Q contains up to $4q + 1$ elements (repeating endpoint positions imply a smaller size of A_Q, but for relative small batches

[1] On an Intel Xeon 2.4 GHz, running on one core (H. Ferrada, personal comm.).

of random queries this effect is rather negligible). In an auxiliary array the function mapping from the indices of A_Q into the original positions in A is also kept.

For the contracted data, three procedures are proposed. Two of them, one offline and one online, are based on existing RMQ/LCA algorithms with linear preprocessing costs and constant time queries. Their practical performance is not competitive though. The more interesting variant, ST-RMQ$_{\mathsf{CON}}$, achieves $O(n+q\log q)$ time[2]. The required space (for all variants), on top of the input array A and the list of queries Q, is claimed to be $O(q)$, but a more careful look into the algorithm (and the published code) reveals that in the implementation of the contracting step the top bits of the entries of A are used for marking. There is nothing wrong in such a bit-stealing technique, from a practical point[3], but those top bits may not always be available and thus in theory the space should be expressed as $O(q)$ words plus $O(n)$ bits.

We come back to the ST-RMQ$_{\mathsf{CON}}$ algorithm. Bender and Farach [3] made a simple observation: as the minimum in a range R is the minimum over the minima of arbitrary ranges (or subsets) in R with the only requirement that the whole R is covered, for an array A of size n it is enough to precompute the minima for (only) $O(n\log n)$ ranges to handle any RMQ. More precisely, for each left endpoint $A[i]$ we compute the minima for all valid $A[i\ldots i+2^k-1]$ ($k=0,1,\ldots$) ranges, and then for any $A[i\ldots j]$ it is enough to compute the minimum of two already computed minima: for $A[i\ldots i+2^{k'}-1]$ and $A[j-2^{k'}+1\ldots j]$, where $k'=\lfloor\log(j-i)\rfloor$. Applying this technique for the contracted array would yield $O(q\log q)$ time and space for this step. Finally, all the queries can be answered with the described technique, in $O(q)$ time. In the cited work, however, the last two steps are performed together, with re-use of the array storing the minima. Due to this clever trick, the size of the helper array is only $O(q)$.

3 Our algorithms

3.1 Block-based Sparse Table with the input array contraction

On a high level, our first algorithm consists of the following four steps:

1. Sort the queries and remap them with respect to the contracted array's indices (to be obtained in step 2).
2. Contract A to obtain A_Q of size $O(q)$ (integers).
3. Divide A_Q into equal blocks of size k and for each block B_j (where $j=1,2,\ldots$) find and store the positions of $O(\log q)$ minima, where ith value ($i=1,2,\ldots$) is the minimum of $A_Q[1+(j-1)k\ldots(j-1)k+(2^{i-1}-k)k]$, i.e., the minimum over a span of 2^{i-1} blocks, where the leftmost block is B_j.
4. For each query $[\ell_i,r_i]$, find the minimum m'_i over the largest span of blocks fully included in the query and not containing the query endpoints. Then, read the minimum of the block to which ℓ_i belongs and the minimum of the block to which r_i belongs; only if any of them is less than m'_i, then scan (at most) $O(k)$ cells of A_Q to find the true minimum and return its position.

[2] Written consistently as $n+O(q\log q)$ in the cited work, to stress that the constant associated with scanning the original array A is low.

[3] One of the authors of the current work also practiced it in a variant of the SamSAMi full-text index [8, Section 2.3].

In the following paragraphs we are going to describe those steps in more detail, also pointing out the differences between our solution and Alzamel et al.'s one.

(1) Sorting/remapping queries. Each of the $2q$ query endpoints is represented as a pair of 32-bit integers: its value (position in A) and its index in the query list Q. The former 4-byte part is the key for the sort while the latter 4 bytes are satellite data. In the serial implementation, we use kxsort[4], an efficient MSD radix sort variant. In the parallel implementation, our choice was Multiway-Mergesort Exact variant implemented in GNU libstdc++ parallel mode library[5]. As a result, we obtain a sorted endpoint list $E[1 \dots 2q]$, where $E_i = (E_i^x, E_i^y)$ and $E_{i+1}^x \geq E_i^x$. Alzamel et al. do not sort the queries, which is however possible due to marking bits in A.

(2) Creating A_Q. Our contracted array A_Q contains the minima of all areas $A[E_i^x \dots E_{i+1}^x]$, in order of growing i. A_Q in our implementation contains thus (up to) $2q - 1$ entries, twice less than in Alzamel et al.'s solution. Like in the preceding solution, we also keep a helper array mapping from the indices of A_Q into the original positions in A.

(3) Sparse Table on blocks. Here we basically follow Alzamel et al. in their ST-RMQ$_{\mathsf{CON}}$ variant, with the only difference that we work on blocks rather than individual elements of A_Q. For this reason, this step takes $O(q + (q/k)\log(q/k)) = O(q(1 + \log(q/k)/k))$ time and $O((q/k)\log(q/k))$ space. The default value of k, used in the experiments, is 512.

(4) Answering queries. Clearly, the smaller of two accessed minima in the Sparse Table technique is the minimum over the largest span of blocks fully included in the query and not containing the query endpoints. To find the minimum over the whole query we perform *speculative reads* of the two minima of the extreme blocks of our query. Only if at least one of those values is smaller than the current minimum, we need to scan a block (or both blocks) in $O(k)$ time. This case is however rare for an appropriate value of k. This simple idea is crucial for the overall performance of our scheme. In the worst case, we spend $O(k)$ per query here, yet on average, assuming uniformly random queries over A, the time is $O((k/q)\times k + (1-k/q)\times 1) = O(1+k^2/q)$, which is $O(1)$ for $k = O(\sqrt{q})$.

Let us sum up the time (for a serial implementation) and space costs. A scan over array A is performed once, in $O(n)$ time. The radix sort applied to our data of $2q$ integers from $\{1, \dots, n\}$ takes (in theory) $O(q \max(\log n/\log q, 1))$ time. Alternatively, introsort from C++ standard library (i.e., the std::sort function) would yield $O(q \log q)$ time. To simplify notation, the $Sort(q)$ term will further be used to denote the time to sort the queries and we also introduce $q' = q/k$. A_Q is created in $O(q)$ time. Building the Sparse Table on blocks adds $O(q + q' \log q')$ time. Finally, answering queries requires $O(qk)$ time in the worst case and $O(q + k^2)$ time on average. In total, we have $O(n + Sort(q) + q' \log q' + qk)$ time in the worst case. The extra space is $O(q' \log q')$.

Let us now consider a generalization of the doubling technique in Sparse Table (a variant that we have not implemented). Instead of using powers of 2 in the formula $A_Q[1+(j-1)k \dots (j-1)k+(2^{i-1}-k)k]$, we use powers of an arbitrary integer $\ell \geq 2$ (in a real implementation it is convenient to assume that ℓ is a power of 2, e.g., $\ell = 16$). Then, the minimum over a range will be calculated as a minimum over ℓ precomputed values. Overall we obtain $O(n + Sort(q) + q' \log q'/\log \ell + q\ell + qk)$ worst-case time,

[4] `https://github.com/voutcn/kxsort`

[5] `https://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html`

which is minimized for $\ell = \max(\log q'/(k \log\log q'), 2)$. With k small enough to have $\ell = \log q'/(k \log\log q')$, we obtain $O(n + Sort(q) + q' \log q'/\log\log q' + qk)$ overall time and the required extra space is $O(q' \log q'/\log\log q')$.

If we focus on the average case, where the last additive term of the worst-case time turns into k^2/q, it is best to take $k = \sqrt{q}$, which implies $\ell = 2$. In other words, this idea has its niche only considering the worst-case time, where for a small enough k both the time and the space of the standard block-based Sparse Table solution are improved.

3.2 Block-based Sparse Table with no input array contraction

This algorithm greatly simplifies the one from the previous subsection: we do not contract the array A and thus also have no need to sort the queries. Basically, we reduce the previous variant to the last two stages. Naturally, this comes at a price: the extra space usage becomes $O((n/k)\log(n/k))$ (yet the optimal choice of k may be different, closer to $\sqrt{n}$). Experiments will show that such a simple idea offers very competitive RMQ times.

Let us focus on the space and time complexities for this variant, for both the worst and the average case. The analysis resembles the one from the previous subsection. We have two parameters, n and k, and two stages of the algorithm. The former stage takes $O(n + (n/k)\log(n/k))$ time, the latter takes $O(qk)$ time in the worst case and $O(q(1 + k^2/n))$ on average (which is $O(q)$ if $k = O(\sqrt{n})$). In total we have $O(n+(n/k)\log(n/k)+qk)$ time in the worst case and $O(n+(n/k)\log(n/k)+q)$ time on average, provided in the latter case that $k = O(\sqrt{n})$. The space is $O((n/k)\log(n/k))$. To minimize both the time and the space for the average case we set $k = \Theta(\sqrt{n})$. Then the average time becomes $O(n + \sqrt{n}\log\sqrt{n} + q) = O(n + q)$ and the space is $O(\sqrt{n}\log n)$.

3.3 Multi-level block-based Sparse Table

The variant from Subsection 3.2 can be generalized to multiple block levels. We start from the simplest case, replacing one level of blocks with two levels.

The idea is to compute minima for n/k_1 non-overlapping blocks of size k_1 and then apply the doubling technique from Sparse Table on larger blocks, of size k_2. We assume that k_1 divides k_2.

The first stage, finding the minima for blocks of size k_1, takes $O(n)$ time. The second stage, working on blocks of size k_2, takes $O(n/k_1 + (n/k_2)\log(n/k_2))$ time. The third stage answers the queries; if we are unlucky and one or two blocks of size k_2 have to be scanned, the procedure is sped up with aid of the precomputed minima for the blocks of size k_1. The query answering takes thus $O(q(k_2/k_1 + k_1))$ time in the worst case and $O(q)$ time on average if $(k_2/n) \times (k_2/k_1 + k_1) = O(1)$. The condition on the average case becomes clear when we notice that the probability of the unlucky case is $\Theta(k_2/n)$ and checking (up to) two blocks takes $O(k_2/k_1 + k_1)$ time. Fulfilling the given condition implies that $k_1 k_2 = O(n)$ and $k_2/k_1 = O(n/k_2)$.

Our goal is to find such k_1 and k_2 that the extra space is minimized but the average time of $O(n + q)$ preserved. To this end, we set $k_1 = \sqrt{n}/\log^{1/3} n$, $k_2 = \sqrt{n}\log^{2/3} n$, and for these values the average time becomes $O(n + n/k_1 + (n/k_2)\log(n/k_2) + q) = O(n + q)$. The space is $O(n/k_1 + (n/k_2)\log(n/k_2)) = O(\sqrt{n}\log^{1/3} n)$.

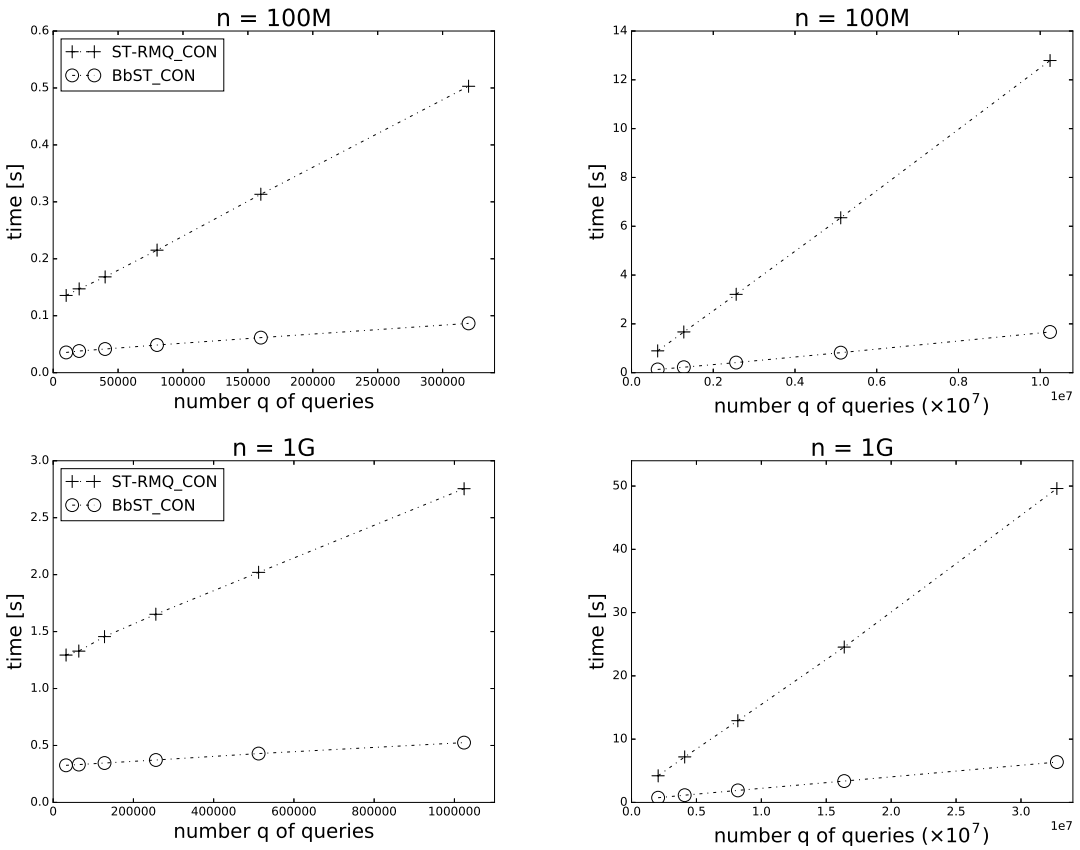


Figure 1. Running times for $\mathsf{ST\text{-}RMQ_{CON}}$ and $\mathsf{BbST_{CON}}$ with varying number of queries q, from $\sqrt{n}$ to $32\sqrt{n}$ (left figures) and from $64\sqrt{n}$ to $1024\sqrt{n}$ (right figures), where n is 100 million (top figures) or 1 billion (bottom figures)

Note that we preserved the average time of the variant from Subsection 3.2 and reduced the extra space by a factor of $\log^{2/3} n$. Note also that the space complexity cannot be reduced for any other pair of k_1 and k_2 such that $k_1 k_2 = O(n)$.

We can generalize the presented scheme to have $h \geq 2$ levels. To this end, we choose h parameters, $k_1 < \ldots < k_h$, such that each k_i divides k_{i+1}. The minima for non-overlapping blocks of size k_i, $1 \leq i < h$, are first computed, and then also the minima for blocks of size k_h, their doubles, quadruples, and so on. The $O(q)$ average time for query answering now requires that $(k_h/n)\times(k_h/k_{h-1}+k_{h-1}/k_{h-2}+\ldots+k_2/k_1+k_1) = O(1)$. We set $k_1 = \sqrt{n}/\log^{1/(h+1)} n$ and $k_i = \sqrt{n}\log^{(i-1)/(h-1)-1/(h+1)} n$ for all $2 \leq i \leq h$, which gives $k_h = \sqrt{n}\log^{h/(h+1)} n$. Let us suppose that $h = O(\log\log n)$. The aforementioned condition is fulfilled, the average time is $O(n+q)$, and the space is $O(n/k_1 + n/k_2 + \ldots + n/k_{h-1} + (n/k_h)\log(n/k_h)) = O(n/k_1 + (n/k_h)\log(n/k_h)n) = O(\sqrt{n}\log^{1/(h+1)} n)$. By setting $h = \log\log n - 1$ we obtain $O(\sqrt{n})$ words of space.

4 Experimental results

In the experiments, we followed the methodology from [2]. The array A stores a permutation of $\{1, \ldots, n\}$, obtained from the initially increasing sequence by swapping $n/2$ randomly selected pairs of elements. The queries are pairs of the form (ℓ_i, r_i), where ℓ_i and r_i are uniformly randomly drawn from $\{1, \ldots, n\}$ and if it happens that

q (in 1000s)	stage 1	stages 1–2	stages 1–3	stages 1–4
$n = 100,000,000$				
10	1.4	95.9	95.9	100.0
320	23.5	92.5	93.0	100.0
10240	65.8	88.3	89.1	100.0
$n = 1,000,000,000$				
32	0.4	99.6	99.6	100.0
1024	13.8	96.5	96.8	100.0
32768	59.0	87.9	88.6	100.0

Table 1. Cumulative percentages of the execution times for the successive stages of $\mathsf{BbST_{CON}}$ with the fastest serial sort (kxsort). The default value of k (512) was used. Each row stands for a different number of queries (given in thousands).

the former index is greater than the latter, they are swapped. The number of queries q varies from $\sqrt{n}$ to $1024\sqrt{n}$, doubling each time (in [2] they stop at $q = 128\sqrt{n}$).

Our first algorithm, $\mathsf{BbST_{CON}}$ (Block based Sparse Table with Contraction), was implemented in C++ and compiled with 32-bit gcc 6.3.0 with `-O3 -mavx -fopenmp` switches. Its source codes can be downloaded from `https://github.com/kowallus/BbST`. The experiments were conducted on a desktop PC equipped with a 4-core Intel i7 4790 3.6 GHz CPU and 32 GB of 1600 MHz DDR3 RAM (9-9-9-24), running Windows 10 Professional. All presented timings in all tests are medians of 7 runs, with cache flushes in between.

In the first experiment we compare $\mathsf{BbST_{CON}}$ with default settings ($k = 512$, kxsort in the first stage) against $\mathsf{ST\text{-}RMQ_{CON}}$ (Fig. 1). Two sizes of the input array A are used, 100 million and 1 billion. The left figures present the execution times for small values of q while the right ones correspond to bigger values of q. We can see that the relative advantage of $\mathsf{BbST_{CON}}$ over $\mathsf{ST\text{-}RMQ_{CON}}$ grows with the number of queries, which in part can be attributed to using a fixed value of k (the selection was leaned somewhat toward larger values of q). In any case, our algorithm is several times faster than its predecessor.

Table 1 contains some profiling data. Namely, cumulative percentages of the execution times for the four successive stages (cf. 3.1) of $\mathsf{BbST_{CON}}$ with default settings, are shown. Unsurprisingly, for a growing number of queries the relative impact of the sorting stage (labeled as stage 1) grows, otherwise the array contraction (stage 2) is dominating. The last two stages are always of minor importance in these tests.

In Fig. 2 we varied the block size k (the default sort, kxsort, was used). With a small number of queries the overall timings are less sensitive to the choice of k. It is interesting to note that optimal k can be found significantly below $\sqrt{n}$.

Different sorts, in a serial regime, were applied in the experiment shown in Fig. 3. Namely, we tried out C++'s qsort and std::sort, kxsort, __gnu_parallel::sort and Intel parallel stable sort (pss). The function qsort, as it is easy to guess, is based on quick sort. The other sort from the C++ standard library, std::sort, implements introsort, which is a hybrid of quick sort and heap sort. Its idea is to run quick sort and only if it gets into trouble on some pathological data (which is detected when the recursion stack exceeds some threshold), switch to heap sort. In this way, std::sort works in $O(n \log n)$ time in the worst case. The next contender, kxsort, is an efficient MSD radix sort. The last two sorters are parallel algorithms, but for this test they are run with a single thread. The gnu sort is a multiway mergesort (exact variant) from the

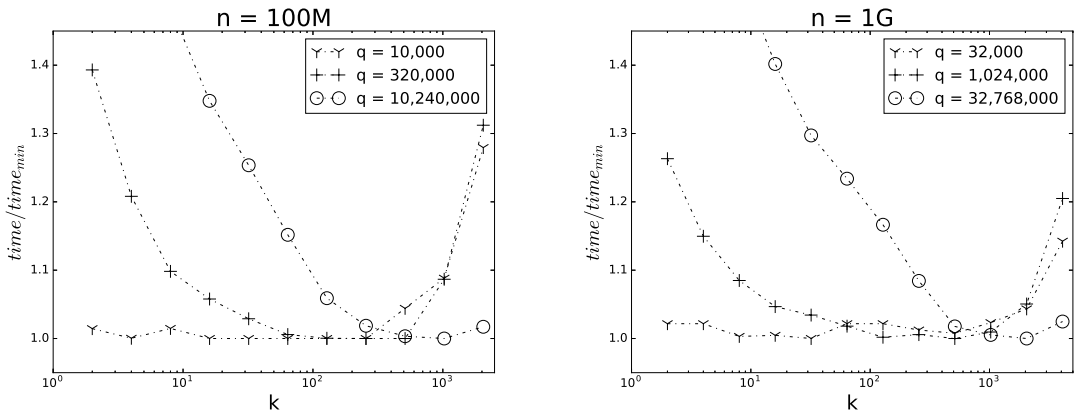


Figure 2. Ratio of running times to minimal running time for $\mathsf{BbST_{CON}}$ for several values of the block size k and varying the number of queries q, from $\sqrt{n}$ to $1024\sqrt{n}$, where n is 100 million (left figure) or 1 billion (right figure)

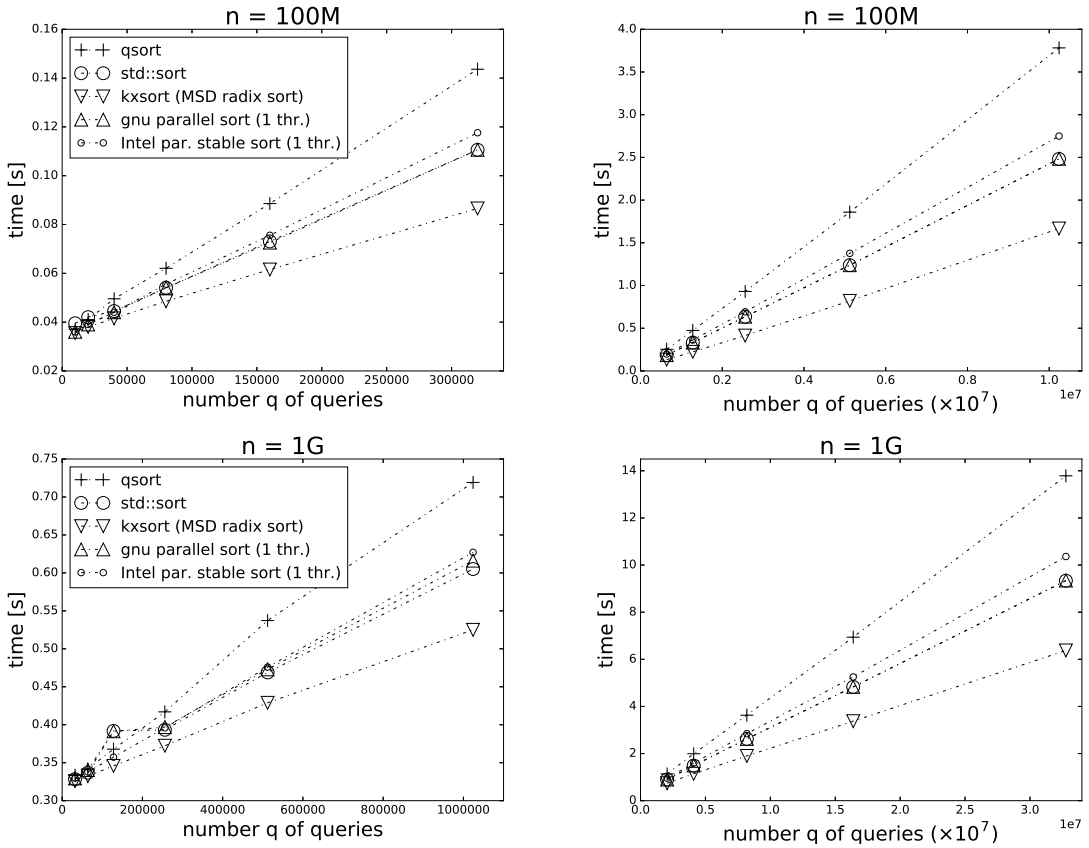


Figure 3. Impact of the sort algorithm on the running times of $\mathsf{BbST_{CON}}$. The number of queries q varies from $\sqrt{n}$ to $32\sqrt{n}$ (left figures) and from $64\sqrt{n}$ to $1024\sqrt{n}$ (right figures), where n is 100 million (top figures) or 1 billion (bottom figures)

GNU libstdc++ parallel mode library. Finally, Intel's pss is a parallel merge sort[6]. We use it in the OpenMP 3.0 version.

[6] `https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp`

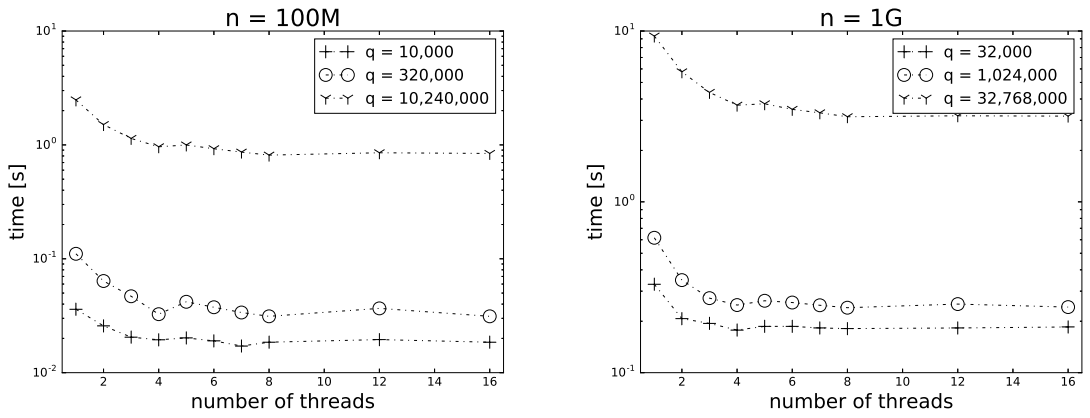


Figure 4. Impact of the number of threads in __gnu_parallel::sort and in creating A_Q (by independent scanning for minima in contiguous areas of A) on the overall performance of $\mathsf{BbST}_{\mathsf{CON}}$, for different number of queries q, where n is 100 million (left figure) or 1 billion (right figure). Note the logarithmic scale on the Y-axis.

For the last experiment with $\mathsf{BbST}_{\mathsf{CON}}$, we ran our algorithm in a parallel mode, varying the number of threads in $\{1, 2, \ldots, 8, 12, 16\}$ (Fig 4). For sorting the queries we took the faster parallel sort, __gnu_parallel::sort. The remaining stages also benefit from parallelism. The second stage computes in parallel the minima in contiguous areas of A and the third stage correspondingly handles blocks of A_Q. Finally, answering queries is handled in an embarassingly parallel manner. As expected, the performance improves up to 8 threads (as the test machine has 4 cores and 8 hardware threads), but the overall speedups compared to the serial variant are rather disappointing, around factor 2 or slightly more.

Finally, we ran a preliminary test of the algorithm from Subsection 3.2, BbST, using the parameters of $k = \{4096, 16384, 65536\}$ (Fig. 5). As expected, a smaller value of k fits better the smaller value of n and vice versa (but for small q and the larger n our timings were slightly unpredictable). Although we have not tried to fine tune the parameter k, we can easily see the potential of this algorithm. For example, with $k = 16384$ and the largest tested number of queries, BbST is 2.5 times faster than $\mathsf{BbST}_{\mathsf{CON}}$ for the smaller n and almost 6 times faster for the larger n. Changing k to 4096 in the former case increases the time ratio to well over 8-fold!

Table 2 presents the memory use (apart from input array A and the set of queries Q) for the two variants. BbST is insensitive here to q. The parameter k was set to 512 in the case of $\mathsf{BbST}_{\mathsf{CON}}$. As expected, the space for $\mathsf{BbST}_{\mathsf{CON}}$ grows linearly with q. BbST is more succinct for the tested number of queries ($q \geq \sqrt{n}$), even if for a very small q $\mathsf{BbST}_{\mathsf{CON}}$ would easily win in this respect.

5 Final remarks

We have proposed simple yet efficient algorithms for bulk range minimum queries. Experiments on random permutations of $\{1, \ldots, n\}$ and with ranges chosen uniformly random over the input sequence show that one of our solutions, $\mathsf{BbST}_{\mathsf{CON}}$, is from 3.8 to 7.8 times faster than its predecessor, $\mathsf{ST\text{-}RMQ}_{\mathsf{CON}}$ (the gap grows with increasing the number of queries). The key idea that helped us achieve this advantage is adapting the well-known Sparse Table technique to work on blocks, with speculative block minima comparisons.

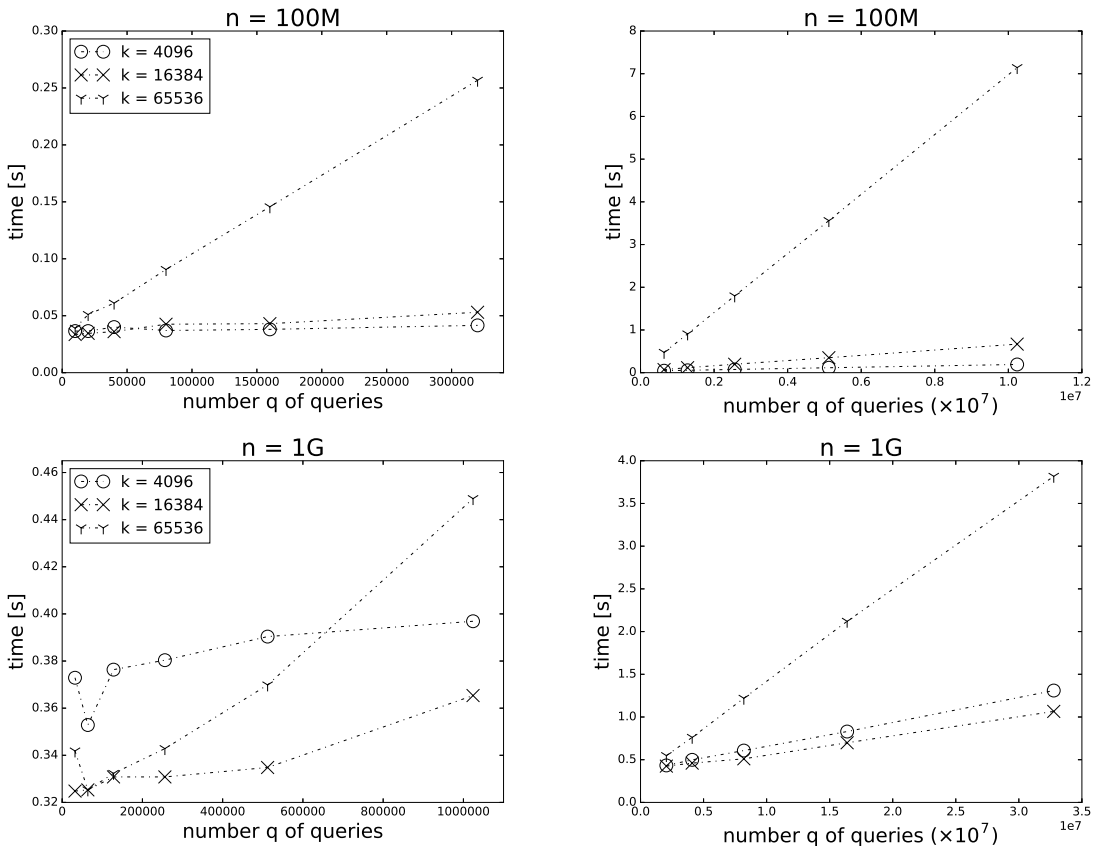


Figure 5. Running times for BbST for several values of the block size k and varying the number of queries q, from $\sqrt{n}$ to $32\sqrt{n}$ (left figures) and from $64\sqrt{n}$ to $1024\sqrt{n}$ (right figures), where n is 100 million (top figures) or 1 billion (bottom figures)

variant	extra space as % of the input	
with parameter	$n = 100,000,000$	$n = 1,000,000,000$
$\mathsf{BbST}_{\mathsf{CON}}$, $q \approx \sqrt{n}$	0.10	0.03
$\mathsf{BbST}_{\mathsf{CON}}$, $q \approx 32\sqrt{n}$	3.23	1.03
$\mathsf{BbST}_{\mathsf{CON}}$, $q \approx 1024\sqrt{n}$	103.68	33.20
BbST, $k = 2048$	1.56	1.86
BbST, $k = 4096$	0.73	0.88
BbST, $k = 8192$	0.34	0.42
BbST, $k = 16,384$	0.16	0.20
BbST, $k = 32,768$	0.07	0.09

Table 2. Memory use for the two variants, as the percentage of the space occupied by the input array A (which is $4n$ bytes). The parameter k was set to 512 for $\mathsf{BbST}_{\mathsf{CON}}$.

Not surprisingly, extra speedups can be obtained with parallelization, as shown by our preliminary experiments. This line of research, however, should be pursued further.

The variant BbST, although possibly not as compact as $\mathsf{BbST}_{\mathsf{CON}}$ (when the number of queries is very small), proves even much faster. We leave running more thorough experiments with this variant, including automated selection of parameter k, as a future work.

Acknowledgement

The work was supported by the Polish National Science Centre under the project DEC-2013/09/B/ST6/03117 (both authors).

References

1. P. Afshani and N. Sitchinava: *I/O-efficient range minima queries*, in SWAT, R. Ravi and I. L. Gørtz, eds., vol. 8503 of LNCS, Springer, 2014, pp. 1–12.
2. M. Alzamel, P. Charalampopoulos, C. S. Iliopoulos, and S. P. Pissis: *How to answer a small batch of RMQs or LCA queries in practice*. CoRR, abs/1705.04589 2017, accepted to IWOCA'17.
3. M. A. Bender and M. Farach-Colton: *The LCA problem revisited*, in LATIN, G. H. Gonnet, D. Panario, and A. Viola, eds., vol. 1776 of LNCS, Springer, 2000, pp. 88–94.
4. H. Ferrada and G. Navarro: *Improved range minimum queries*. Journal of Discrete Algorithms, 43 2017, pp. 72–80.
5. J. Fischer and V. Heun: *Space-efficient preprocessing schemes for range minimum queries on static arrays*. SIAM J. Comput., 40(2) 2011, pp. 465–492.
6. J. Fischer, V. Mäkinen, and N. Välimäki: *Space efficient string mining under frequency constraints*, in ICDM, IEEE Computer Society, 2008, pp. 193–202.
7. H. N. Gabow, J. L. Bentley, and R. E. Tarjan: *Scaling and related techniques for geometry problems*, in STOC, ACM, 1984, pp. 135–143.
8. S. Grabowski and M. Raniszewski: *Sampled suffix array with minimizers*. Softw., Pract. Exper., 2017, accepted.
9. R. Grossi and G. Ottaviano: *Design of practical succinct data structures for large data collections*, in SEA, V. Bonifaci, C. Demetrescu, and A. Marchetti-Spaccamela, eds., vol. 7933 of LNCS, Springer, 2013, pp. 5–17.
10. D. Harel and R. E. Tarjan: *Fast algorithms for finding nearest common ancestors*. SIAM J. Comput., 13(2) 1984, pp. 338–355.
11. K. Sadakane: *Compressed suffix trees with full functionality*. Theory Comput. Syst., 41(4) 2007, pp. 589–607.

A Lempel-Ziv-style Compression Method for Repetitive Texts

Markus Mauer, Timo Beller, and Enno Ohlebusch

Institute of Theoretical Computer Science
Ulm University
89069 Ulm, Germany
{Markus.Mauer,Timo.Beller,Enno.Ohlebusch}@uni-ulm.de

Abstract. In this paper, we present a compression algorithm that is based on finding repetitions in the file to be compressed. Our approach is a variant of longest-first-substitution compression that uses the suffix array and the LCP-array to find and encode long recurring substrings. We will show that our algorithm achieves very good compression ratios for repetitive texts.

Keywords: lossless data compression, longest-first-substitution compression, repetitive texts, suffix array

1 Introduction

The dictionary-based LZ-algorithms devised by Lempel and Ziv [14,21,22] are an important class of lossless compression algorithms. One can distinguish between *on-line* algorithms (in which the dictionary is dynamically built from the prefix of the text seen so far) and *off-line* algorithms (in which the dictionary is constructed from the whole text). The original LZ77-algorithm uses a window of size w and the dictionary consists of all substrings that start within the last w scanned positions of the text. In classical implementations, the LZ77-algorithm parses greedily, i.e., if $S[1..i-1]$ has already been scanned, then the next factor is the *longest* prefix of $S[i..n]$ that is in the dictionary and starts within $S[1..i-1]$. If the next factor has length ℓ and starts at position $j \leq i-1$ in S, then the LZ77-algorithm encodes the triple (d, ℓ, c), where $d = i - j < w$ is the offset and $c = S[i+\ell]$ is the character following the factor; it then continues parsing $S[i+\ell+1..n]$. If the window consists of all positions scanned so far (we will call this algorithm LZ77-compression without window), the offset d can be very large, so one should select the rightmost copy of a factor to keep d small (see [9] for an algorithm that does this with the help of the suffix tree of S). As pointed out in [16], the LZ77 compression algorithm without window that encodes the absolute position j at which the next factors starts (instead of the offset d), should be called LZ76 compression [14]. The greedy algorithm without window is optimal with respect to the number of factors, and it can be implemented in such a way that it uses only linear time and space [5] (the result of Crochemore and Ilie has been improved by many authors; see [10] and the references therein). If one encodes the factors by variable-length codes, then the greedy algorithm is in general not bit-optimal, i.e., it is not optimal in terms of the number of bits output by the compression algorithm; see [9]. Ferragina et al. [9, Theorem 5.4] use a linear-time algorithm for the single-source shortest path problem on a weighted DAG to obtain a bit-optimal algorithm for the LZ77-compression-scheme.

In this paper, we present an off-line compression algorithm that is different from the LZ-algorithms described above in that it does not try to parse the text in a

Markus Mauer, Timo Beller, Enno Ohlebusch: A Lempel-Ziv-style Compression Method for Repetitive Texts, pp. 96–107.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

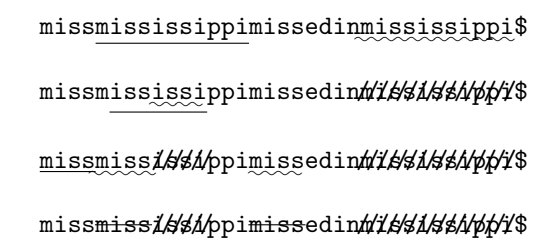


Figure 1. Consider the string $S =$ `missmississippimissedinmississippi$`. Our compression algorithm first detects the repeat mississippi and encodes the wavy underlined occurrence (repeat of type 2). Then, it detects the periodicity `ississi` with period-length 3 and encodes the wavy underlined occurrence of `issi` (repeat of type 1). Finally, it detects the three repetitions of `miss` and encodes the wavy underlined occurrences (this repeat gets the identifier 3). In the resulting string $S' =$ `miss##ppi#edin#$`, every occurrence of # stands for a factor and the vector $F = 3132$ contains the types (from left to right) of these factors. The factor of type 3 is $(1, 4)$, the factor of type 1 is $(3, 4)$, and the factor of type 2 is $(5, 11)$. That is, the list of factors (from left to right) is $[(1, 4), (3, 4), (5, 11)]$; see Section 4 for details.

left-to-right scan into phrases. By contrast, it identifies long repetitions in advance (prior to the compression) and then tries to greedily compress these repetitions (first the longest, then the second longest, etc.). This strategy is called longest-first-substitution. If the repetition is a periodicity (called repeat of type 1) or if it occurs only twice (called repeat of type 2), then it is stored in a list of factors and the type is stored in a vector F. However, if it occurs more than twice, then a factor is stored only for the second occurrence whereas the other occurrences are encoded by a unique identifier, which is stored in F. All occurrences of these repeats (except for the first occurrence) are replaced with a special symbol # in S, yielding a string S'. The three components (S', F, and the list of factors) are then compressed separately; see Fig. 1 for an example and Section 4 for details. Our software is available at `https://www.uni-ulm.de/in/theo/research/seqana/`

2 Related Work for DNA-sequences

When writing this paper, we were unaware of the work of Rivals et al. [20]. It turned out that they used the same basic idea as our algorithm, but they restrict their algorithm to DNA-sequences. Moreover, the details differ substantially. For example, they do not take periodicities (overlapping repeats) into account. Furthermore, they encode *one* sequence consisting of substrings, factors, and indices to the dictionary. By contrast, we separate the three types. This separation makes the three parts amenable to different compression techniques, i.e., one can apply every lossless data compression algorithm to S' and F (while the factors, which are pairs of position and length, are encoded separately; see Section 4 for details). More related work can be found in [2,15,19,7].

The problem of compressing a collection of genomes from individuals of the same species with respect to a reference genome has been extensively studied. The relative Lempel-Ziv (RLZ) algorithm devised by Kuruppu et al. [12,13] is a popular algorithm for this special case, especially when fast random access is required. The RLZ-algorithm was subsequently improved by Deorowicz and Grabowski [6], by Ferrada et al. [8], and by Cox et al. [4]. In contrast to these algorithms, our algorithm does not rely on a reference sequence: it can be applied to every (repetitive) text. On

the one hand, our algorithm provides better compression than these algorithms; on the other hand, our approach does not support random access.

3 Preliminaries

Let Σ be an ordered alphabet of size σ whose smallest element is the sentinel character \$. In the following, S is a string of length n on Σ having the sentinel character at the end (and nowhere else). For $1 \leq i \leq n$, $S[i]$ denotes the *character at position* i in S. For $i \leq j$, $S[i..j]$ denotes the *substring* of S starting with the character at position i and ending with the character at position j. Furthermore, S_i denotes the i-th suffix $S[i..n]$ of S. The *suffix array* SA of the string S is an array of integers in the range 1 to n specifying the lexicographic ordering of the n suffixes of S, that is, it satisfies $S_{\mathsf{SA}[1]} < S_{\mathsf{SA}[2]} < \cdots < S_{\mathsf{SA}[n]}$; see Fig. 2 for an example. We refer to the overview article [18] for suffix array construction algorithms (some of which have linear run-time).

The suffix array is closely related to the Burrows and Wheeler transform [3] $\mathsf{BWT}[1..n]$, which is defined by $\mathsf{BWT}[i] = S[\mathsf{SA}[i] - 1]$ for all i with $\mathsf{SA}[i] \neq 1$ and $\mathsf{BWT}[i] = \$$ otherwise; see Fig. 2.

The suffix array SA is often enhanced with the LCP-array containing the lengths of longest common prefixes between consecutive suffixes in SA; see Fig. 2. Formally, the LCP-array is an array so that $\mathsf{LCP}[1] = -1 = \mathsf{LCP}[n+1]$ and $\mathsf{LCP}[i] = |\mathsf{lcp}(S_{\mathsf{SA}[i-1]}, S_{\mathsf{SA}[i]})|$ for $2 \leq i \leq n$, where $\mathsf{lcp}(u, v)$ denotes the longest common prefix between two strings u and v. Kasai et al. [11] showed that the LCP-array can be computed in linear time from the suffix array and its inverse. Abouelhoda et al. [1] introduced the concept of lcp-intervals; see Fig. 2. An interval $[i..j]$, where $1 \leq i < j \leq n$, in the LCP-array is called an *lcp-interval of lcp-value* ℓ (denoted by ℓ-$[i..j]$) if

1. $\mathsf{LCP}[i] < \ell$,
2. $\mathsf{LCP}[k] \geq \ell$ for all k with $i+1 \leq k \leq j$,
3. $\mathsf{LCP}[k] = \ell$ for at least one k with $i+1 \leq k \leq j$,
4. $\mathsf{LCP}[j+1] < \ell$.

In Fig. 2, for example, the interval [9..14] is an lcp-interval of lcp-value 3.

Abouelhoda et al. [1] presented an algorithm that enumerates all lcp-intervals in a bottom-up fashion. Moreover, they showed that there is a one-to-one correspondence between the set of all lcp-intervals and the set of all internal nodes of the suffix tree of S (we assume a basic knowledge of suffix trees). Consequently, there are at most $n-1$ lcp-intervals for a string of length n.

If ℓ-$[i..j]$ is an lcp-interval, then the ℓ-length prefix ω of $S_{\mathsf{SA}[k]}$, where $i \leq k \leq j$, is a *repeat* because the number of occurrences of ω in S is $j - i + 1 \geq 2$. If $\{\mathsf{BWT}[k] \mid i \leq k \leq j\}$ is not a singleton set, then ω is a *maximal* repeat. In this case, the lcp-interval $[i..j]$ is also called *maximal*. For example, the lcp-interval 3-[9..14] in Fig. 2 is maximal.

If a non-empty string ω can be written as $\omega = u^k v$, where $k \geq 2$ and v is a proper prefix of u, then it is called a *periodicity* with *period-length* $|u|$.

4 The Compression Algorithm

In this section, we first describe the basic approach of our compression algorithm (implementation details will be discussed later). The key idea is to classify repeats

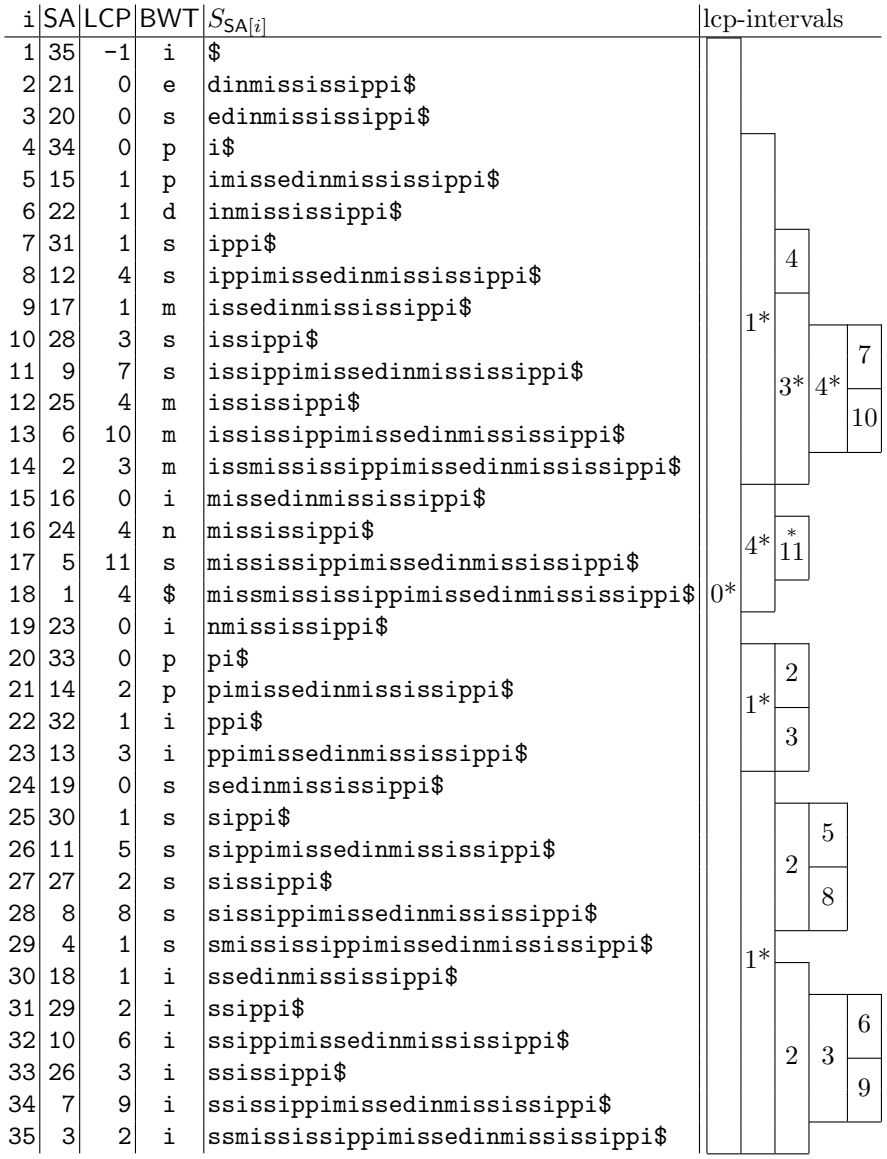


Figure 2. Suffix array, LCP-array, and the Burrows-Wheeler transform BWT of the string $S =$ `missmississippimissedinmississippi$`. If an lcp-interval is maximal, then its lcp-value is marked with an asterisk.

into different types, which are treated differently. These types are represent by the variable *id*. An overlapping repeat (a periodicity) is said to be of type 1 ($id = 1$). In this case, the part without the first period is encoded by a reference to the beginning of the repeat (and the period-length). If there are two non-overlapping occurrences of a repeat, then it is of type 2 ($id = 2$). In this case, the second occurrence is encoded as in LZ76 compression by a reference to the first occurrence. Finally, if there are more than two non-overlapping occurrences of a repeat, then each of the occurrences—except for the first one—is encoded by a unique identifier $id > 2$. It is important to note that for each such identifier, only one reference is stored.

4.1 The basic approach

Our basic compression algorithm works as follows:

1. Compute all maximal lcp-intervals of the LCP-array.[1] This can e.g. be done in linear time by a bottom-up traversal of the LCP-interval tree; see [17, Algorithm 5.15].
2. Store all maximal lcp-intervals with an lcp-value $\ell \geq \ell_{min}$ in a priority queue Q (the priority of an lcp-interval is its lcp-value ℓ; the higher the better), where ℓ_{min} is a threshold.[2]
3. Initialize a bit-vector B of size $n = |S|$ with zeros, initialize an empty list *list* and set $id \leftarrow 3$. In the following, a substring $S[k..m]$ of S is said to be marked if and only if $B[k..m]$ contains at least a one. An unmarked substring can be subject to compression, but a marked substring can not.
4. While Q is not empty, remove the lcp-interval ℓ-$[lb..rb]$ with the currently highest priority from Q and do:
   (a) Compute a subset *candidates* of $\{\mathsf{SA}[i] \mid lb \leq i \leq rb\}$ so that for each $k \in$ *candidates* the substring $S[k..k+\ell-1]$ is unmarked. This is the case if and only if $B[k] = 0$ and $B[k+\ell-1] = 0$.[3] During the computation, determine $occ_1 = \min\{\mathsf{SA}[i] \mid lb \leq i \leq rb\}$ and $occ_2 = \min(candidates \setminus \{occ_1\})$; note that occ_1 may or may not be a member of the set *candidates*.
   (b) Sort *candidates* and store the result in an array *sorted_candidates* of size $|candidates|$. Set $cur \leftarrow occ_1$ and $i \leftarrow 1$ and determine the subset *accepted* $\subseteq$ *candidates* as follows:
   while $i \leq |candidates|$ do
   – if $cur + \ell - 1 < sorted\_candidates[i]$, then add $sorted\_candidates[i]$ to *accepted* and set $cur \leftarrow sorted\_candidates[i]$; set $i \leftarrow i + 1$
   Note that occ_1 is not a member of the set *accepted*. As a result, for each pair $j, k \in$ *accepted* with $j < k$ we have $j + \ell - 1 < k$ (i.e., the corresponding substrings are non-overlapping). If the set *accepted* is non-empty, then $occ_1 + \ell - 1 < occ_3$ where $occ_3 = \min(accepted)$. Note that occ_3 may or may not be equal to occ_2.
   (c) If $occ_1 \in$ *candidates* and $t = occ_2 - occ_1 < \ell$, then $S[occ_1..occ_1 + \ell - 1]$ and $S[occ_2..occ_2+\ell-1]$ overlap and $occ_2 \notin$ *accepted*. In this case, add $(occ_1+t, 1, t, \ell)$ to *list* and set $B[occ_1 + t..occ_1 + t + \ell - 1] \leftarrow [1..1]$ unless $S[occ_2..occ_2 + \ell - 1]$ overlaps with $S[occ_3..occ_3 + \ell - 1]$ (i.e., $occ_3 \neq \perp$ and $occ_3 - occ_2 < \ell$).[4]
   (d) Let $size = |accepted|$ be the size of the set *accepted*. If $size > 0$ and $\ell \geq a/size + b$, where a and b are constants that will be explained in Section 4.2, then proceed with (4e); otherwise take the next interval from Q. In essence, the restriction on ℓ ensures that the compression of the factors (whose starting positions are in the set *accepted*) is worthwhile.
   (e) If $size = 1$, where $size = |accepted|$, then add $(occ_3, 2, occ_1, \ell)$ to *list* and set $B[occ_3..occ_3 + \ell - 1] \leftarrow [1..1]$.

[1] In a previous implementation we used all lcp-intervals, but this resulted in unacceptable run-times.
[2] In our implementation, ℓ_{min} equals the constant a, which will be explained in Section 4.2.
[3] If $B[k+1..k+\ell-2]$ would contain a one while $B[k] = 0$ and $B[k+\ell-1] = 0$, then a substring of $S[k+1..k+\ell-2]$ would have been subject to compression. Consequently, an lcp-interval of lcp-value $< \ell$ must have been chosen before the current lcp-interval of lcp-value ℓ. This, however, is impossible because lcp-intervals are chosen greedily (first the longest, then the second longest, etc.).
[4] $\perp$ denotes an undefined value.

(f) If $size > 1$, then add (occ_3, id, occ_1, ℓ) to *list* and set $B[occ_3..occ_3 + \ell - 1] \leftarrow [1..1]$. Furthermore, for each $k \in accepted \setminus \{occ_3\}$, add $(k, id, \perp, \ell)$ to *list* and set $B[k..k + \ell - 1] \leftarrow [1..1]$. Finally, increment *id* by one.

We note that (a part of) the first occurrence of a repeat is subject to compression only if it overlaps with the second occurrence; see Case (4c). Cases (4e) – (4f) deal with non-overlapping occurrences of the repeat under consideration. If there is just one unmarked occurrence apart from the first occurrence, then Case (4e) applies, whereas Cases (4f) applies if there is more than one unmarked occurrence apart from the first occurrence.

5. Let *sorted* be the list obtained by sorting the elements in *list* according to their first components (in increasing order).
6. Initialize an empty vector F, an empty list *factors*, an empty string S', and set $p \leftarrow 1$.
7. While *sorted* is not empty, remove its first element (k, id, occ, ℓ) and do:
   (a) If $id = 1$ or $id = 2$, then insert *id* at the back of vector F and insert (occ, ℓ) at the back of list *factors*.
   (b) If $id > 2$, then insert *id* at the back of vector F. Furthermore, if $occ \neq \perp$, insert (occ, ℓ) at the back of list *factors*.
   (c) Concatenate S' with $S[p..k-1]\#$ and set $p \leftarrow k$. (In essence, S' is obtained from S by replacing each factor $S[k..k + \ell - 1]$ with $\#$.)
8. Compress the list *factors*, the vector F, and the string S' separately.

As an example, consider $S =$ `missmississippimissedinmississippi$`. The corresponding suffix- and LCP-arrays are shown in Fig. 2. For $\ell_{min} = 4$, the priority queue looks as follows: $Q = [(11, [16..17]), (4, [11..13]), (4, [15..18])]$. In the first iteration of the while-loop (4), case (4e) applies for the lcp-interval $(11, [16..17])$, where $occ_1 = 5$ and $occ_3 = occ_2 = 24$. Thus, the quadruple $(24, 2, 5, 11)$ is added to *list* and all the bits in $B[24..34]$ are set to 1. In the second iteration of the while-loop (4), the sets $candidates = \{6, 9\}$ and $accepted = \emptyset$ are computed in steps (4a) and (4b), respectively. Furthermore, we have $occ_1 = 6$, $occ_2 = 9$, and $occ_3 = \perp$. It is not difficult to see that case (4c) applies with $t = 3$, so that the quadruple $(6 + 3, 1, 3, 4)$ is added to *list* and all the bits in $B[9..12]$ are set to 1. In the final iteration of the while-loop (4), we have $candidates = \{1, 5, 16\}$ and $accepted = \{5, 16\}$ as well as $occ_1 = 1$ and $occ_2 = occ_3 = 5$. Now case (4f) applies, so first $(5, 3, 1, 4)$ is added to *list* and all the bits in $B[5..8]$ are set to 1 and then $(16, 3, \perp, 4)$ is added to *list* and the bits in $B[16..19]$ are set to 1. It follows as a consequence that $sorted = [(5, 3, 1, 4), (9, 1, 3, 4), (16, 3, \perp, 4), (24, 2, 5, 11)]$. Furthermore, we have $factors = [(1, 4), (3, 4), (5, 11)]$, $F = 3132$, and $S' =$ `miss##ppi#edin#$`.

Let us analyse the worst-case time complexity of the compression algorithm. The first and the third step can be done in $O(n)$ time, while the second step requires $O(n \log n)$ time. As explained in Section 3, there are at most $n - 1$ lcp-intervals (note that there are strings, e.g. the string $S = a^{n-1}\$$, for which each of its lcp-intervals is maximal). It follows as a consequence that the while-loop in Case (4) has at most $n - 1$ iterations. Clearly, each iteration deals with an lcp-interval $[lb..rb]$ of size $rb - lb + 1 < n$. For each i with $lb \leq i \leq rb$, it can be tested in constant time whether SA$[i]$ belongs to the set *candidates* or not; see Case (4a). Moreover, the set *candidates* can be sorted in linear time in Case (4b) provided we use counting sort (in practice, however, a comparison based sorting algorithm will outperform counting sort). It is quite obvious that each of the Cases (4c) – (4f) takes at most $O(n)$ time. Consequently, the while-loop in Case (4) runs in $O(n^2)$ time. It is not difficult to see

that $O(n^2)$ is also an upper bound for each of the remaining steps. In summary, the compression algorithm has a worst-case time complexity of $O(n^2)$.

4.2 Implementations details

First of all, we will explain how a factor (a pair consisting of a position and a length) is encoded in our approach.

- Consider two consecutive factors (occ_1, ℓ_1) and (occ_2, ℓ_2) in the list *factors*. If $diff = (occ_2 - occ_1)$ satisfies $|diff| < 2^x$, where x is a fixed natural number, then we use a Rice code plus a sign bit to encode $diff$. Otherwise, the position occ_2 is encoded with $\lceil \log_2 n \rceil$ bits.
- If the length ℓ of a factor satisfies $\ell < 2^y$, where y is a fixed natural number, then we use a Rice code to encode it. Otherwise it is encoded with $\lceil \log_2 \ell_{max} \rceil$ bits, where ℓ_{max} is the maximum entry in the LCP-array.

Our software contains subroutines that compute the best values of x and y for the input file (prior to the compression of the factors).

Next, we will explain the constants a and b in step (4d) of the basic compression algorithm. To this end, let S_{bits} (S'_{bits}) be the average number of bits needed to encode one symbol in S (S') with a fixed compression algorithm X. In the following, we assume that S_{bits} and S'_{bits} are approximately the same and from now on we denote the average number of bits needed to encode one symbol by k. Similarly, let F_{bits} be the average number of bits needed to encode one symbol in F and let $Factor_{bits}$ denote the average number of bits needed to encode one factor. Recall that $size = |accepted|$ denotes the size of the set *accepted*. On the one hand, if ℓ is the length of the repeat to be compressed, then we would need approximately $(size+1) * \ell * k$ bits to encode all the occurrences with the compression algorithm X ($size+1$ many occurrences of the length ℓ repeat have to be taken into account). On the other hand, in our approach we would need

- $\ell * k$ bits to encode the first occurrence of the repeat plus $size * k$ bits to encode the extra # symbols with the compression algorithm X,
- $size * F_{bits}$ bits to encode the occurrences of the identifier (type) of the repeat in F, and
- $Factor_{bits}$ bits to encode the factor.

Our compression scheme is worthwhile whenever the following inequality holds:

$$\begin{aligned}
(size+1) * \ell * k &\geq \ell * k + size * k + size * F_{bits} + Factor_{bits} \\
\Leftrightarrow size * \ell * k &\geq size * k + size * F_{bits} + Factor_{bits} \\
\Leftrightarrow \ell &\geq 1 + \frac{F_{bits}}{k} + \frac{Factor_{bits}}{k * size} \\
\Leftrightarrow \ell &\geq \frac{a}{size} + b
\end{aligned}$$

where $a = \frac{Factor_{bits}}{k}$ and $b = 1 + \frac{F_{bits}}{k}$. In our implementation, we use the parameters $a = 30$ and $b = 80$ as default values because these values gave the best compression ratios in our experiments.

We would like to point out two more facts to the reader, which are important in practice:

- To limit the number of lcp-intervals, our algorithm uses only maximal lcp-intervals. However, if the string $S = a^{n-1}\$$ is input, then every lcp-interval is maximal and the run-time slows down significantly. Our algorithm deals with such cases at the very beginning (i.e., when all lcp-intervals are enumerated): it detects a periodicity and its period length (in case of $S = a^{n-1}\$$, it detects that a^{n-1} is a periodicity of period length 1) and does not add lcp-intervals that belong to the same periodicity to the queue Q.
- We use the special symbol # to denote the places of factors in S'. However, if S already contains #, then the decompression algorithm will not work properly. To avoid this, whenever # appears in S, a 0 is added to the type vector F at the appropriate place.

5 The Decompression Algorithm

The basic decompression algorithm decompresses the list *factors*, the vector F, and the string S' separately. It then restores the original string S from S' with the help of a variable *cur* (points to the current factor in *factors*), a variable *pos* (current position in S), and an array $table[1..max]$ (entries initialized with $\perp$), where max is the maximum number (identifier) in F, as follows. If the current symbol c in S' is not #, then it is simply copied, i.e., $S[pos] \leftarrow c$ and $pos \leftarrow pos + 1$. If $c = \#$, say c is the k-th occurrence of #, then the algorithm uses a case distinction on the type $F[k]$.

- If $F[k] = 0$, then $S[pos] \leftarrow \#$. Set $pos \leftarrow pos + 1$.
- If $F[k] = 1$, then $S[pos..pos + \ell - 1] \leftarrow S[pos - t..pos - t + \ell - 1]$, where $(t, \ell) \leftarrow factors[cur]$. Set $cur \leftarrow cur + 1$ and $pos \leftarrow pos - t + \ell$.
- If $F[k] = 2$, then $S[pos..pos + \ell - 1] \leftarrow S[occ..occ + \ell - 1]$, where $(occ, \ell) \leftarrow factors[cur]$. Set $cur \leftarrow cur + 1$ and $pos \leftarrow occ + \ell$.
- If $F[k] > 2$, then
  - if $table[k] = \perp$, then $table[k] \leftarrow factors[cur]$ and $cur \leftarrow cur + 1$

  set $S[pos..pos + \ell - 1] \leftarrow S[occ..occ + \ell - 1]$, where $(occ, \ell) \leftarrow table[k]$, and $pos \leftarrow occ + \ell$.

6 An Advanced Algorithm

In addition to the basic version of our algorithm (as described in the previous two sections), we implemented a second advanced version. The advanced version takes substrings of strings from the set $candidate \setminus accepted$ into account; see e.g. [19] for a similar approach. More importantly, the advanced version uses a different labeling scheme for the F vector that is obtained by replacing step (7) in the basic compression algorithm as follows:

Initialize the variable $newId$ with the value 3.
Let max be the maximum value of all identifiers in *sorted*.
Initialize an array *table* of size max.
While *sorted* is not empty, remove its first element (k, id, occ, ℓ) and do:

1. If $id = 1$ or $id = 2$, then insert id at the back of vector F and insert (occ, ℓ) at the back of list *factors*. If $id = 2$, then increment $newId$ by 1.

2. If $id > 2$ and $occ \neq \perp$, i.e., id occurs for the first time, then insert 2 at the back of vector F, insert (occ, ℓ) at the back of list *factors*, set $table[id] \leftarrow newId$, and increment $newId$ by one.
3. If $id > 2$ and $occ = \perp$, then insert $table[id]$ at the back of vector F.
4. Concatenate S' with $S[p..k-1]\#$ and set $p \leftarrow k$. (In essence, S' is obtained from S by replacing each factor $S[k..k+\ell-1]$ with $\#$.)

Thus, even if a repeat has several occurrences, each second occurrence is encoded by a 2 in F. Since this results in many occurrences of 2 in F, the compression ratio for F is better than before. Of course, the decompression algorithm must be able to cope with the new F vector. To this end, the following modification of the basic decompression algorithm is necessary:

Initialize the variable $newId$ with the value 3.
Initialize an array *table* of size $count + 2$, where *count* is the number occurrences of the value 2 in the new F vector.

- If $F[k] = 0$, then ... (the same as before).
- If $F[k] = 1$, then ... (the same as before).
- If $F[k] = 2$, then $S[pos..pos+\ell-1] \leftarrow S[occ..occ+\ell-1]$, where $(occ, \ell) \leftarrow factors[cur]$. Set $cur \leftarrow cur + 1$ and $pos \leftarrow occ + \ell$. Moreover, set $table[newId] \leftarrow factors[cur]$ and increment $newId$ by one.
- If $F[k] > 2$, then set $S[pos..pos+\ell-1] \leftarrow S[occ..occ+\ell-1]$, where $(occ, \ell) \leftarrow table[k]$, and $pos \leftarrow occ + \ell$.

7 Experimental Results

To test our compression method, we conducted several experiments using different state of the art compression methods. We compared the sizes of the compressed files as well as the compression and decompression times. As dataset we used four repetitive files from the Pizza & Chili corpus[5] and two from the `RLZAP` dataset.[6]

In our experiments, we used the lossless data compression methods `bzip2` Version 1.0.6, `gzip` Version 1.6, `xz`[7] Version 5.1.0alpha with the compression preset level -9 (the primary compression algorithm of `xz` is currently LZMA2), `zpaq`[8] Version 7.15 with the compression level -m5 (i.e. using a high order context mixing model), and `RLZAP`.[9] We compared these methods with both the basic and the advanced version of our compression method. Since `xz` and `zpaq` provide the best compression ratios, we used them to compress the three components S', F, and *factors*.

Table 1 shows the file sizes after compression. Both the basic and the advanced version of our method outperform the other methods in five of six cases. The poor compression ratios of gzip can be attributed to the fact that the files contain occurrences of repeats that are far apart (i.e., their distance is greater than the window size). A similar statement holds for bzip2 because it compresses blocks rather than the whole text (the default block size is 900k). In Table 2, we show exemplarily the sizes of the three components S', F, and *factors* for the file para before and after the

[5] `http://pizzachili.dcc.uchile.cl/index.html`
[6] `http://acube.di.unipi.it/rlzap-dataset/`
[7] `http://tukaani.org/xz/`
[8] `https://github.com/zpaq/zpaq`
[9] `https://github.com/farruggia/rlzap`

	world_leaders	**einstein.de**	**influenza**
Filesize	46.968	92.758	154.809
bzip2	3.261	4.010	10.197
gzip	8.288	28.797	10.637
xz	0.607	0.099	**2.068**
zpaq	0.519	0.130	2.639
basic+xz	0.552	0.096	2.203
basic+zpaq	0.476	0.540	10.051
advanced+xz	0.518	0.092	2.132
advanced+zpaq	**0.453**	**0.084**	2.491
RLZAP	-	-	-
	kernel	**e_coli**	**para**
Filesize	257.962	164.899	429.266
bzip2	56.074	44.465	112.236
gzip	69.396	46.136	116.073
xz	2.087	6.289	6.256
zpaq	3.652	30.386	87.787
basic+xz	2.037	6.158	5.709
basic+zpaq	4.088	14.458	17.821
advanced+xz	2.019	**6.073**	**5.567**
advanced+zpaq	**1.573**	8.600	8.925
RLZAP	-	22.556	11.634

Table 1. Sizes after compression in MB (10^6 bytes).

	S'		**F**		**factors**	
basic+xz	35.769	4.612	0.515	0.123	1.065	0.974
basic+zpaq	35.769	16.090	0.515	0.106	1.065	1.625
advanced+xz	34.126	4.507	0.592	0.011	1.137	1.049
advanced+zpaq	34.126	7.878	0.592	0.009	1.137	1.039

Table 2. Sizes of the different components in MB (10^6 bytes) for the file para, before and after the final compression step (8) of our algorithm. Note that in this case zpaq compresses S' much worse than xz. However, this varies from file to file.

final compression step (8). As already mentioned, our advanced method achieves a smaller size for S' by finding additional factors. While this gives a larger F vector as well as a larger *factors* list, the different naming scheme for the F vector results in better final compression ratios. Moreover, we would like to point out that S' is a lot smaller than the original string S.

The compression and decompression times are listed in Table 3. While bzip2 and gzip have the fastest compression times, their compression ratios are rather poor. Apart from these two, xz tends to give the best compression times, but our method is not far behind. Note that xz gives the best decompression times for all files, but our method is also very fast if xz is used in the final compression step. However, if we use zpaq as a final compression method, both compression and decompression times are significantly higher (but the combination of our algorithm with zpaq is always faster than zpaq itself).

All in all, the results show that our method can keep up with the state of the art compression algorithms both in terms of compression ratios and in terms of compression/decompression time. Furthermore, several improvements of our method seem possible. For example, the greedy strategy could be based on a sophisticated rating of factors (instead of the simple rating based on the lengths of factors) or there may

	world_leaders		**einstein.de**		**influenza**	
bzip2	0m02s	0.604s	0m09s	1.531s	**0m17s**	2.725s
gzip	**0m2s**	0.211s	**0m05s**	0.516s	0m22s	0.504s
xz	0m08s	**0.101s**	0m10s	**0.141s**	0m36s	**0.339s**
zpaq	1m34s	93.692s	2m34s	154.737s	6m23s	381.018s
basic+xz	0m30s	0.154s	0m35s	0.165s	2m22s	0.609s
basic+zpaq	0m37s	7.254s	0m36s	1.067s	2m52s	40.123s
advanced+xz	0m33s	0.206s	0m36s	0.288s	2m43s	0.764s
advanced+zpaq	0m38s	6.211s	0m36s	1.070s	3m15s	39.745s
	kernel		**e_coli**		**para**	
bzip2	0m18s	6.415s	**0m13s**	4.956s	**0m32s**	12.788s
gzip	**0m13s**	1.479s	1m36s	0.858s	4m01s	2.218s
xz	1m18s	**0.484s**	2m08s	**0.558s**	5m46s	**0.980s**
zpaq	6m03s	365.241s	6m53s	425.736s	18m33s	1140.174s
basic+xz	1m45s	0.625s	1m55s	0.996s	13m13s	1.681s
basic+zpaq	2m06s	25.458s	2m55s	95.598s	14m17s	102.614s
advanced+xz	1m43s	0.942s	1m54s	1.127s	14m56s	1.870s
advanced+zpaq	2m04s	24.564s	2m54s	93.652s	16m02s	98.342s

Table 3. Compression/decompression times for the files from our dataset. The compression times are given in minutes and seconds. For the decompression times, we use seconds.

be other ways of building the F vector. Finally, our method is quite flexible because it can be combined with other compression methods in the final compression step.

Acknowledgements: We thank the anonymous reviewers for their helpful comments.

References

1. M. Abouelhoda, S. Kurtz, and E. Ohlebusch: *Replacing suffix trees with enhanced suffix arrays*. Journal of Discrete Algorithms, 2(1) 2004, pp. 53–86.
2. A. Apostolico and S. Lonardi: *Compression of biological sequences by greedy off-line textual substitution*, in Proc. 10th Data Compression Conference, IEEE Computer Society, 2000, pp. 143–152.
3. M. Burrows and D. Wheeler: *A block-sorting lossless data compression algorithm*, Research Report 124, Digital Systems Research Center, 1994.
4. A. Cox, A. Farruggia, T. Gagie, S. Puglisi, and J. Sirén: *RLZAP: relative Lempel-Ziv with adaptive pointers*, in Proc. 23rd International Symposium on String Processing and Information Retrieval, vol. 9954 of Lecture Notes in Computer Science, Springer, 2016, pp. 1–14.
5. M. Crochemore and L. Ilie: *Computing longest previous factor in linear time and applications*. Information Processing Letters, 106(2) 2008, pp. 75–80.
6. S. Deorowicz and S. Grabowski: *Robust relative compression of genomes with random access*. Bioinformatics, 27(21) 2011, pp. 2979–2986.
7. P. Dinklage, J. Fischer, D. Köppl, M. Löbel, and K. Sadakane: *Compression with the tudocomp framework*. CoRR, abs/1702.07577 2017.
8. H. Ferrada, T. Gagie, S. Gog, and S. Puglisi: *Relative Lempel-Ziv with constant-time random access*, in Proc. 21st International Symposium on String Processing and Information Retrieval, vol. 8799 of Lecture Notes in Computer Science, Springer, 2014, pp. 13–17.
9. P. Ferragina, I. Nitto, and R. Venturini: *On the bit-complexity of Lempel-Ziv compression*. SIAM Journal on Computing, 42(4) 2013, pp. 1521–1541.
10. J. Kärkkäinen, D. Kempa, and S. Puglisi: *Linear time Lempel-Ziv factorization: Simple, fast, small*, in Proc. 24th Annual Symposium on Combinatorial Pattern Matching, vol. 7922 of Lecture Notes in Computer Science, Springer, 2013, pp. 189–200.

11. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *Linear-time longest-common-prefix computation in suffix arrays and its applications*, in Proc. 12th Annual Symposium on Combinatorial Pattern Matching, vol. 2089 of Lecture Notes in Computer Science, Springer, 2001, pp. 181–192.
12. S. Kuruppu, S. Puglisi, and J. Zobel: *Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval*, in Proc. 17th International Symposium on String Processing and Information Retrieval, vol. 6393 of Lecture Notes in Computer Science, Springer, 2010, pp. 201–206.
13. S. Kuruppu, S. Puglisi, and J. Zobel: *Optimized relative Lempel-Ziv compression of genomes*, in Proc 34th Australasian Computer Science Conference, Australian Computer Society, 2011, pp. 91–98.
14. A. Lempel and J. Ziv: *On the complexity of finite sequences.* IEEE Transactions on Information Theory, 21(1) 1976, pp. 75–81.
15. R. Nakamura, S. Inenaga, H. Bannai, T. Funamoto, M. Takeda, and A. Shinohara: *Linear-time text compression by longest-first substitution.* Algorithms, 2(4) 2009, pp. 1429–1448.
16. G. Navarro: *Compact Data Structures*, Cambridge University Press, Cambridge, 2016.
17. E. Ohlebusch: *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*, Oldenbusch Verlag, Bremen, 2013.
18. S. Puglisi, W. Smyth, and A. Turpin: *A taxonomy of suffix array construction algorithms.* ACM Computing Surveys, 39(2) 2007, p. Article 4.
19. S. Ristov and D. Korencic: *Using static suffix array in dynamic application: Case of text compression by longest first substitution.* Information Processing Letters, 115(2) 2015, pp. 175–181.
20. E. Rivals, J.-P. Delahaye, M. Dauchet, and O. Delgrange: *A guaranteed compression scheme for repetitive DNA sequences*, in Proc. 6th Data Compression Conference, IEEE Computer Society, 1996, p. 453.
21. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23(3) 1977, pp. 337–343.
22. J. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Transactions on Information Theory, 24(5) 1978, pp. 530–536.

On Reverse Engineering the Lyndon Tree

Yuto Nakashima[1], Takuya Takagi[2,3], Shunsuke Inenaga[1],
Hideo Bannai[1], and Masayuki Takeda[1]

[1] Department of Informatics, Kyushu University, Japan
{yuto.nakashima, inenaga, bannai, takeda}@inf.kyushu-u.ac.jp
[2] Graduate School of IST, Hokkaido University, Japan
tkg@ist.hokudai.ac.jp
[3] Japan Society for the Promotion of Science (JSPS), Japan

Abstract. We consider the problem of reverse engineering the Lyndon tree, i.e., given a full binary ordered tree T with n leaves as input, compute a string w of length n for which it's Lyndon tree is isomorphic to the input tree. Although the problem is easy and solvable in linear time when assuming a binary alphabet or when there is no limit on the alphabet size, how to efficiently find the smallest alphabet size for a solution string is not known. We show several new observations concerning this problem. Namely, we show that: 1) For any full binary ordered tree T, there exists a solution string w over an alphabet of size at most $h+1$, where h is the height of T. 2) For any positive n, there exists a full binary ordered tree T with n leaves, s.t. the smallest alphabet size of the solution string for T is $\lfloor \frac{n}{2} \rfloor + 1$.

1 Introduction

The problem of efficiently inferring a string from a given data structure that is defined based on the string has been considered in many contexts; for example, border array [9], directed acyclic word graph [2], suffix array [2], suffix tree [11,5,17], the run structure of a word [16], LCP array [12], etc. The motivation is to elucidate and better understand the combinatorial properties of the data structures in question, which could lead to better algorithms on constructing, representing, or using the structures.

In this paper, we consider the problem of inferring a string from its Lyndon tree [3]. A string is a *Lyndon word* [14] if it is lexicographically smaller than any of its proper suffixes. Given a Lyndon word w of length $n > 1$, (u, v) is the *standard factorization* [6,13] of w, if $w = uv$ and v is the longest proper suffix of w that is a Lyndon word, or equivalently, the lexicographically smallest proper suffix of w. It is well known that for the standard factorization (u, v) of any Lyndon word w, the factors u and v are also Lyndon words (e.g.[4,13]). The *Lyndon tree* of w is the full binary tree defined by recursive standard factorization of w; w is the root of the Lyndon tree of w, its left child is the root of the Lyndon tree of u, and its right child is the root of the Lyndon tree of v. Figure 1 shows an example of a Lyndon tree for the Lyndon word `aababaababb`. Lyndon trees have recently been shown to have connection with the structure of maximal repeats, or runs, contained in the string [1].

Releated Work

Franek et al. [8] considered the problem of reconstructing the string from its *Lyndon array*. The Lyndon array is an array of integers which contains the length of the longest Lyndon word that starts at each position. For example, for the string `aababaababb`, the Lyndon array is $(11, 2, 1, 2, 1, 6, 5, 1, 3, 1, 1)$. Since a node in the

Yuto Nakashima, Takuya Takagi, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda: On Reverse Engineering the Lyndon Tree, pp. 108–117.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

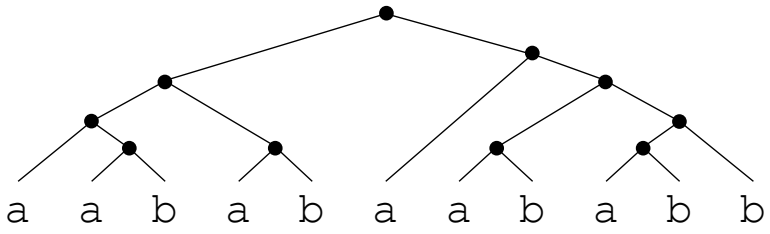


Figure 1. The Lyndon tree for the Lyndon word `aababaababb` with respect to order $\mathtt{a} \prec \mathtt{b}$.

Lyndon tree is a right child of its parent if and only if the node corresponds to the longest Lyndon word that starts at that position ([1] (Lemma 22)), the Lyndon array of a Lyndon word is simply a different representation of the Lyndon tree. It is straightforward to check whether such an array corresponds to a tree topology, and if so retrieve the tree. The remaining problem is to find a string whose Lyndon tree matches this tree structure.

2 Preliminaries

2.1 Strings

Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. The set of characters contained in a string w is denoted by $\Sigma(w)$. The length of a string w is denoted by $|w|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. Let Σ^+ be the set of non-empty strings, i.e., $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For a string $w = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of w, respectively. A prefix x is and a suffix z of w are respectively called a *proper prefix*, and *proper suffix* of w, if $x \neq w$ and $z \neq w$. The i-th character of a string w is denoted by $w[i]$, where $1 \leq i \leq |w|$. For a string w and two integers $1 \leq i \leq j \leq |w|$, let $w[i..j]$ denote the substring of w that begins at position i and ends at position j. For convenience, let $w[i..j] = \varepsilon$ when $i > j$.

2.2 Binary trees

A tree is said to be a binary tree if each internal node has at most two children. In this paper, we use full binary trees and complete binary trees as the representation of Lyndon trees. A binary tree is said to be a *full binary tree* if each internal node has exactly two children. We denote the set of full binary trees with n leaves by FBT_n. A binary tree is said to be a *complete binary tree* if the tree is a full binary tree and all the leaves have the same depth. We denote the set of complete binary tree with n leaves by CBT_n.

Parenthesis representation for full binary trees We will use a parenthesis representation for full binary trees, where a node is represented as a sequence of parenthesis representations of the subtrees rooted at each children, enclosed in parentheses. For example, the full binary tree (without each character at the leaves) shown in Figure 1 can be represented as follows.

`((((()(()()))(()()))(()((()())(((()())()))())))`

For brevity, we use the symbol $\Diamond$ instead of `()` which represents a leaf. We will also sometimes use variables to represent a full binary tree; if T is a tree represented by `(((())(((())())()))`, then the above example can also be represented as follows.

$$(((\Diamond(\Diamond\Diamond)(\Diamond\Diamond)(\Diamond T))$$

2.3 Properties on Lyndon words and Lyndon Trees

Lemma 1 (Proposition 1.3 of [7], [13]). *For any Lyndon words λ_1 and λ_2, $\lambda_1\lambda_2$ is a Lyndon word iff $\lambda_1 \prec \lambda_2$.*

It is easy to see that $\lambda_1 \prec \lambda_1\lambda_2 \prec \lambda_2$ also holds. By the definition of Lyndon trees, each node in a Lyndon tree represents a Lyndon word. For any node w in a Lyndon tree, let $str(w)$ be the Lyndon word which is represented by w. From Lemma 1 and the definition of Lyndon trees, for any internal node w in a Lyndon tree (let $u(v)$ be the left(right) child of w, respectively), $str(u) \prec str(w) \prec str(v)$ holds. Assume that λ is a Lyndon word which corresponds to a Lyndon tree. We can extend this lexicographic relation between $str(u)$ (or $str(w)$) and $str(v)$ to the lexicographic relation between suffixes of λ at each beginning position. In other word, the suffix of λ which begins at beginning position of $str(u)$ is lexicographically smaller than the suffix of λ which begins at beginning position of $str(v)$. For any nodes u, v s.t. v is not a sibling of u and $str(u)$ is followed by $str(v)$ in λ, $str(u) \succeq str(v)$ holds. This relation also can be extended to the relation between two suffixes. In this paper, we denote the Lyndon tree of a Lyndon word λ by $LTree(\lambda)$.

3 Reverse Engineering of the Lyndon Trees

Our reverse engineering problem on Lyndon trees is formalized as follows.

Problem 2. Given a full binary tree T with n leaves, compute a Lyndon word of length n s.t. its Lyndon tree is isomorphic to T.

For any ordered tree T_1 and T_2, we will write $T_1 \equiv_I T_2$ if T_1 and T_2 are isomorphic. Firstly, we summarize known results for our problem in Section 3.1. Secondly, we give an upper bound on the alphabet size of an output string in Section 3.2. Thirdly, we also give a lower bound in Section 3.3. Finally, we consider our problem restricted to complete binary trees in Section 3.4.

3.1 Algorithms on restricted alphabet

The following two Lemmas have been shown by Franek et al. [8] in a slightly different context.

Lemma 3 (algorithm on binary alphabet). *For any $T \in FBT_n$, we can compute a Lyndon word λ s.t. $|\Sigma(\lambda)| \leq 2$ and $LTree(\lambda) \equiv_I T$ in $O(n)$ time. (If no Lyndon word exists, return false.)*

Lemma 4 (algorithm on any alphabet). *For any $T \in FBT_n$, we can compute a Lyndon word λ s.t. $|\Sigma(\lambda)| = n$ and $LTree(\lambda) \equiv_I T$ in $O(n)$ time. (Lyndon word w always exists.)*

3.2 Upper bounds

We consider *Suffix Ordered Tree* of a full binary tree T, denoted by $SOT(T)$. We call the tree obtained by removing all leaves (as well as their incoming edges) from a full binary tree T the *internal tree* of T. $SOT(T)$ is a full binary tree which is isomorphic to the internal tree of T. For any node u in $SOT(T)$, let u' be the node in T which corresponds to u, and $pos(u)$ be the number i s.t. the leftmost leaf in subtree rooted at the right child of u' is the i-th leaf from left. For any node u in $SOT(T)$, u is labeled by $pos(u)$. Figure 2 shows an example of $SOT(T)$.

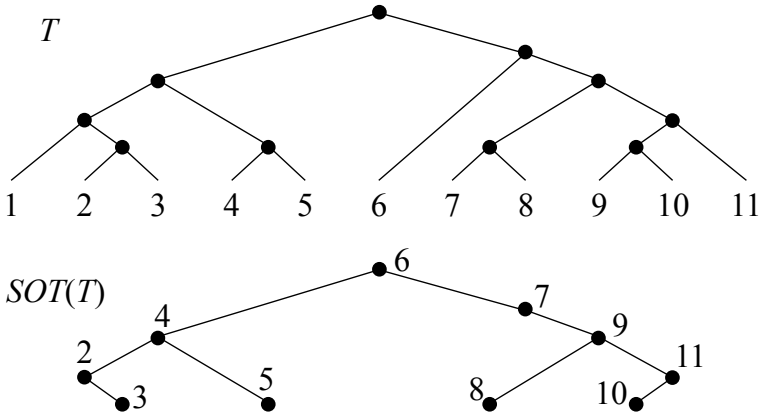


Figure 2. The suffix ordered tree for the full binary tree T.

This tree is known to be related to the *(inverse) suffix array* [15], as shown in the following Lemma. For any string w, we denote the suffix array (resp. inverse suffix array) by $SA(w)$ (resp. $ISA(w)$).

Lemma 5 ([10]). *For any Lyndon word λ, the internal tree of $LTree(\lambda)$ is isomorphic to the Cartesian tree of $ISA(\lambda)[2..|\lambda|]$.*

From Lemma 5, $SOT(T)$ represents necessary conditions w.r.t. lexicographic order of proper suffixes of a Lyndon word λ s.t. $LTree(\lambda) \equiv_I T$. For example, the suffix of λ at position 6 has to be the lexicographically smallest proper suffix of λ. Also, the suffix of λ at position 9 has to be lexicographically smaller than suffixes of λ at position 8, 11 and 10. By the definition of Lyndon words, the suffix at position 1 is always the smallest suffix of λ. Thus, for any Lyndon word λ, the suffix array of λ satisfies suffix orders represented by $SOT(T)$ iff $LTree(\lambda) \equiv_I T$. Then we can obtain the following upper bound of the alphabet size of λ.

Lemma 6 (upper bound). *For any $T \in FBT_n$ of height h, there exists a Lyndon word λ s.t. $|\Sigma(\lambda)| \leq h + 1$ and $LTree(\lambda) \equiv_I T$.*

Proof. We consider the string λ obtained by the following operations. For any node u, we assigned a character c which is lexicographically larger than the character assigned to the parent. If u is labeled by i, $\lambda[i] = c$. Let $\lambda[1]$ be the smallest character in λ s.t. the character does not occur at any other positions.

Then the suffix array of λ does not contradict to $SOT(T)$, and λ includes at most $h + 1$ distinct characters (since there exist h nodes on the longest path in $SOT(T)$). □

3.3 Lower bounds

Lemma 7 (lower bound for even length). *For any positive integer $k \geq 1$, there exists a full binary tree $T \in FBT_{2k}$ s.t.* $\min\{|\Sigma(\lambda)| : T \equiv_I LTree(\lambda)\} = k+1$.

Proof. We prove the lemma by induction on k. Although it suffices to show one tree for each k, we describe all the trees that we have discovered. For $k = 1$ consider the tree $(\Diamond\Diamond) \in FBT_2$, and for $k = 2$, consider the tree $((\Diamond\Diamond)(\Diamond\Diamond))$ or $((\Diamond(\Diamond\Diamond))\Diamond)$. It can be checked by exhaustive enumeration of strings λ of length $2k$ and $|\Sigma(\lambda)| \leq k+1$ that these trees satisfy the statement of the lemma. Next, assume that the lemma holds for all $k \leq i$ for some i and let $T \in FBT_{2i}$ be a full binary tree that satisfies the statement for $k = i$. Let $g(T)$ be the full binary tree $((\Diamond T)\Diamond)$ for any full binary tree T. We claim that the full binary tree $g(T) \in FBT_{2i+2}$ satisfies the statement.

Suppose there exists a Lyndon word $c_1\lambda c_2$ ($c_1, c_2 \in \Sigma$) of length $2i+2$ s.t. $g(T) \equiv_I LTree(c_1\lambda c_2)$. From the induction hypothesis, we have that $|\Sigma(\lambda)| = i+1$ and thus $|\Sigma(c_1\lambda c_2)| \geq i+1$. Due to the structure of $g(T)$, it must also be that $LTree(\lambda) \equiv_I T$. Since $g(T)$ is the Lyndon tree of $c_1\lambda c_2$, $c_1 \prec c_2 \preceq \lambda[1]$ holds. To prove $|\Sigma(c_1\lambda c_2)| \geq i+2$, assume to the contrary that $|\Sigma(c_1\lambda c_2)| \leq i+1$. Then, it must be that $c_1 = \lambda[1]$. However, this implies that $c_1 = c_2$ and contradicts that $c_1\lambda c_2$ is a Lyndon word. Thus, $|\Sigma(c_1\lambda c_2)| \geq i+2$. On the other hand, if we let λ be a string implied by the lemma for $k = i$, $c_1 \prec \lambda[1]$ a new character not in $\Sigma(\lambda)$, and $c_2 = \lambda[1]$, it is easy to see that $c_1\lambda c_2$ is a Lyndon word, $\Sigma(c_1\lambda c_2) = i+2$ and $g(T) \equiv_I LTree(c_1\lambda c2)$. Thus, the statement holds for $k = i+1$, proving the lemma. □

We define the set G of full binary trees described in Lemma 7 as follows : $G = \{g^k(T) \mid T \in \{((\Diamond\Diamond)(\Diamond\Diamond)), g((\Diamond\Diamond))\}, k \geq 1\}$.

Lemma 8 (lower bound for odd length). *For any positive integer $k \geq 1$, there exists a full binary tree $T \in FBT_{2k+1}$ s.t.* $\min\{|\Sigma(\lambda)| : T \equiv_I LTree(\lambda)\} = k+1$.

Proof. We prove the lemma by induction on k. Although it suffices to show one tree for each k, we describe all the trees that we have discovered. For $k = 1$ consider the tree $(\Diamond(\Diamond\Diamond))$ or $((\Diamond\Diamond)\Diamond)$ in FBT_3, for $k = 2$, consider one of the 8 trees in FBT_5, for $k = 3$, consider one of the 22 trees in FBT_7, and for $k = 4$, consider one of the 34 trees in FBT_9. It can be checked by exhaustive enumeration of strings λ of length $2k+1$ and $|\Sigma(\lambda)| \leq k+1$ that these trees satisfy the statement of the lemma. Next, assume that the lemma holds for all $k \leq i$ for some $i > 3$. We claim that all full binary trees in FBT_{2i+3} obtained by the construction described below, satisfy the lemma.

1. Let $T \in FBT_{2i+1}$ be a full binary tree that satisfies the statement for $k = i$. Consider the full binary tree $g(T) \in FBT_{2i+3}$ and suppose there exists a Lyndon word $c_1\lambda c_2$ s.t. $g(T) \equiv_I LTree(c_1\lambda c_2)$. Then, by similar arguments as in Lemma 7, we can see that $|\Sigma(c_1\lambda c_2)| \geq i+2$, while c_1, c_2 can be chosen so that $|\Sigma(c_1\lambda c_2)| = i+2$ and $g(T) \equiv_I LTree(c_1\lambda c_2)$.
2. Let $T \in FBT_{2i+2} \cap G$, and λ a Lyndon word s.t. $T \equiv_I LTree(\lambda)$. By Lemma 7, $|\Sigma(\lambda)| \geq i+2$. Consider the full binary tree $(\Diamond T) \in FBT_{2i+3}$ or $(T\Diamond) \in FBT_{2i+3}$. Suppose there exists a Lyndon word $c_1\lambda$ (resp. λc_2) s.t. $(\Diamond T) \equiv_I LTree(c_1\lambda)$ (resp. $(T\Diamond) \equiv_I LTree(\lambda c_2)$). If we let $c_1 = \lambda[1]$ (resp. $c_2 = \lambda[|\lambda|]$), it is easy to see that $c_1\lambda$ (resp. λc_2) is a Lyndon word and $(\Diamond T) \equiv_I LTree(c_1\lambda)$ (resp. $(T\Diamond) \equiv_I LTree(\lambda c_2)$). Then, $|\Sigma(c_1\lambda)| = |\Sigma(\lambda c_2)| = i+2$.

3. Let $T \in FBT_{2i} \cap G$, and λ a Lyndon word s.t. $T \equiv_I LTree(\lambda)$. By Lemma 7, $|\Sigma(\lambda)| \geq i+1$. Consider the full binary tree $T' = ((\Diamond\Diamond)(\Diamond T)) \in FBT_{2i+3}$ and suppose there exists a Lyndon word $c_1c_2c_3\lambda$ s.t. $T' \equiv_I LTree(c_1c_2c_3\lambda)$. From the structure of the tree, it must be that $c_1 \preceq c_3 \prec c_2$, $c_3 \preceq \lambda[1]$, and $c_1c_2 \preceq c_3\lambda[1]$. If $c_3 = \lambda[1]$, then $c_1 \prec \lambda[1]$ must hold. Thus, we have that either $c_1 \prec \lambda[1]$ or $c_3 \prec \lambda[1]$, i.e., at least one of c_1 or c_3 has to be a new smaller character not contained in $\Sigma(\lambda)$, and thus $|\Sigma(c_1c_2c_3\lambda)| \geq k+1$. On the other hand, if we choose $c_1 \prec c_3 = \lambda[1] \prec c_2 = \lambda[|\lambda|]$, $c_1c_2c_3\lambda$ is a Lyndon word s.t. $|\Sigma(c_1c_2c_3\lambda)| = k+1$ and $T' \equiv_I LTree(c_1c_2c_3\lambda)$.
4. Let $T \in FBT_{2i} \cap G$, and λ a Lyndon word s.t. $T \equiv_I LTree(\lambda)$. By Lemma 7, $|\Sigma(\lambda)| \geq i+1$. Consider the full binary tree $T' = (((\Diamond\Diamond)T)\Diamond) \in FBT_{2i+3}$, and suppose there exists a Lyndon word $c_1c_2\lambda c_3$ s.t. $T' \equiv_I LTree(c_1c_2\lambda c_3)$. From the structure of the tree, it must be that $c_1 \prec c_3 \preceq \lambda[1]$. Thus c_1 has to be a new smaller character not contained in $\Sigma(\lambda)$, and thus $|\Sigma(c_1c_2\lambda c_3)| \geq i+2$. On the other hand, if we choose $c_1 \prec c_3 = \lambda[1] \prec c_2 = \lambda[|\lambda|]$, $c_1c_2\lambda c_3$ is a Lyndon word s.t. $|\Sigma(c_1c_2\lambda c_3)| = i+2$ and $T' \equiv_I LTree(c_1c_2\lambda c_3)$.
5. Let $T \in FBT_{2i} \cap G$, and λ a Lyndon word s.t. $T \equiv_I LTree(\lambda)$. By Lemma 7, $|\Sigma(\lambda)| \geq i+1$. Consider the full binary tree $T' = ((T(\Diamond\Diamond))\Diamond) \in FBT_{2i+3}$, and suppose there exists a Lyndon word $\lambda c_1c_2c_3$ s.t. $T' \equiv_I LTree(\lambda c_1c_2c_3)$. Since $T = g(T'')$ for some $T'' \in G$, it follows from the arguments in Lemma 7 that the structure of T implies that $\lambda[1] \prec \lambda[|\lambda|] \preceq \lambda[2]$. Notice that since $\lambda[2..|\lambda|-1]$ is a Lyndon word, $\lambda[1]$ and $\lambda[|\lambda|]$ are the two smallest characters in $\Sigma(\lambda)$. From the structure of T', it must be that $\lambda[1] \prec c_3 \preceq c_1 \prec \lambda[|\lambda|]$, implying that $c_1, c_3 \notin \Sigma(\lambda)$ and $|\Sigma(\lambda c_1c_2c_3)| \geq i+2$. One the other hand, if we choose $\lambda[1] \prec c_1 = c_3 \prec \lambda[|\lambda|] = c_2c$, $\lambda c_1c_2c_3$ is a Lyndon word s.t. $|\Sigma(\lambda c_1c_2c_3)| = i+2$ and $T' \equiv_I LTree(\lambda c_1c_2c_3)$.
6. Let $T \in FBT_{2i-2} \cap G$, and λ a Lyndon word s.t. $T \equiv_I LTree(\lambda)$. By Lemma 7, $|\Sigma(\lambda)| \geq i$. Consider the full binary tree $T' = (((\Diamond(\Diamond T))(\Diamond\Diamond))\Diamond) \in FBT_{2i+3}$, and suppose there exists a Lyndon word $c_1c_2\lambda c_3c_4c_5$ s.t. $T \equiv_I LTree(c_1c_2\lambda c_3c_4c_5)$. From the structure of T', it must be that $c_1 \prec c_5 \preceq c_3 \preceq c_2 \preceq \lambda[1]$, $c_3 \prec c_4$, and $c_3c_4 \preceq c_2\lambda[1]$. If $c_3 = \lambda[1]$, this implies $c_2 = \lambda[1]$, but then $c_3c_4 \preceq c_2\lambda[1]$ cannot hold. Thus we have that $c_1 \prec c_3 \prec \lambda[1]$, and thus $|\Sigma(c_1c_2\lambda c_3c_4c_5)| \geq i+2$. On the other hand, if we choose $c_1 \prec c_5 = c_3 = c_2 \prec \lambda[1] = c_4$, $c_1c_2\lambda c_3c_4c_5$ is a Lyndon word s.t. $|\Sigma(c_1c_2\lambda c_3c_4c_5)| = i+2$ and $T' \equiv_I LTree(c_1c_2\lambda c_3c_4c_5)$.

□

We can obtain the following theorem by Lemma 7 and Lemma 8.

Theorem 9. *For any positive integer $n \geq 1$, there exists a full binary tree $T \in FBT_n$ s.t.* $\min\{|\Sigma(\lambda)| : T \equiv_I LTree(\lambda)\} = \lfloor \frac{n}{2} \rfloor + 1$.

Conjectures on lower bound We conjecture that any full binary tree which satisfies the above theorem is one of the trees described in Lemma 7 and Lemma 8. For any k, we have two types of trees which satisfies Lemma 7. In Lemma 8, for any k, there exist 12 new types of trees due to case 2-6. This implies that $\lfloor \frac{n}{2} \rfloor + 1$ is also an upper bound.

3.4 Problem on complete binary trees

Here, we restrict the input of our problem and consider a complete binary tree with 2^k leaves. We obtain the following theorem.

Theorem 10. *For any integer $k \geq 0$, there exists a Lyndon word λ s.t. $|\Sigma(\lambda)| \leq 4$ and $LTree(\lambda) \equiv_I CBT_{2^k}$.*

To prove this theorem, we consider a homomorphism f defined as follows.

$$f(\mathtt{a}) = \mathtt{ac}, f(\mathtt{b}) = \mathtt{ad}, f(\mathtt{c}) = \mathtt{bc}, f(\mathtt{d}) = \mathtt{bd}$$

We show an example in Figure 3. By the definition of f, we have the following lemma.

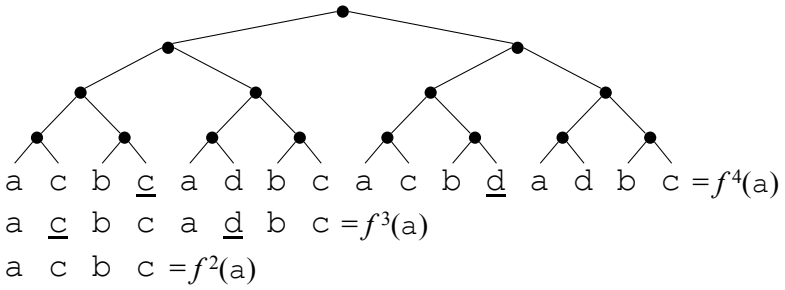


Figure 3. The Lyndon tree of $f^4(\mathtt{a})$.

Lemma 11. *For any $k \geq 3$, the following properties hold.*

1. $f^{k+1}(\mathtt{a}) = f^k(\mathtt{a}) \cdot f^k(\mathtt{a})[1..\frac{|f^k(\mathtt{a})|}{2} - 1] \cdot \mathtt{d} \cdot f^k(\mathtt{a})[\frac{|f^k(\mathtt{a})|}{2} + 1..|f^k(\mathtt{a})|]$.
2. *the length of longest common prefix of $f^k(\mathtt{a})$ and any of its proper suffix is less than $\frac{|f^k(\mathtt{a})|}{4}$.*

Proof.

1. We prove this by induction on k. For $k = 3$, $f^4(\mathtt{a}) = \mathtt{acbcadbcacbdadbc}$ satisfies the statement by $f^3(\mathtt{a}) = \mathtt{acbcadbc}$. We assume that

$$f^{k+1}(\mathtt{a}) = f^k(\mathtt{a}) \cdot f^k(\mathtt{a})[1..\frac{|f^k(\mathtt{a})|}{2} - 1] \cdot \mathtt{d} \cdot f^k(\mathtt{a})[\frac{|f^k(\mathtt{a})|}{2} + 1..|f^k(\mathtt{a})|]$$

holds for all $3 \leq k \leq i$ for some i. Since f derives two characters from each character,

$$f^{i+2}(\mathtt{a})[1..\frac{|f^{i+2}(\mathtt{a})|}{2}]$$

is derived from

$$f^{i+1}(\mathtt{a})[1..\frac{|f^{i+1}(\mathtt{a})|}{2}] = f^i(\mathtt{a}).$$

Thus

$$f^{i+2}(\mathtt{a})[1..\frac{|f^{i+2}(\mathtt{a})|}{2}] = f(f^i(\mathtt{a})) = f^{i+1}(\mathtt{a}).$$

Similarly,

$$\begin{aligned} f^{i+2}(\mathtt{a})[\frac{3}{4}|f^{i+2}(\mathtt{a})| + 1..|f^{i+2}(\mathtt{a})|] &= f(f^i(\mathtt{a})[\frac{|f^i(\mathtt{a})|}{2} + 1..|f^i(\mathtt{a})|]) \\ &= f^{i+1}(\mathtt{a})[\frac{|f^{i+1}(\mathtt{a})|}{2} + 1..|f^{i+1}(\mathtt{a})|]. \end{aligned}$$

Let

$$X = f^{i+1}(\mathtt{a})[1..\frac{|f^{i+1}(\mathtt{a})|}{4} - 1]$$
$$= f^{i+1}(\mathtt{a})[\frac{|f^{i+1}(\mathtt{a})|}{2} + 1..\frac{3}{4}|f^{i+1}(\mathtt{a})| - 1].$$

Then

$$f^{i+2}(\mathtt{a})[\frac{|f^{i+2}(\mathtt{a})|}{2} + 1..\frac{3}{4}|f^{i+2}(\mathtt{a})|] = f(X)\mathtt{bd}.$$

It is clear that the last character of $f^k(\mathtt{a})$ is $\mathtt{c}$ for any $k \geq 2$. Thus

$$f(X)\mathtt{b} = f^{i+1}(\mathtt{a})[1..\frac{|f^{i+1}(\mathtt{a})|}{2} - 1].$$

Therefore,

$$f^{i+2}(\mathtt{a}) = f^{i+1}(\mathtt{a}) \cdot f^{i+1}(\mathtt{a})[1..\frac{|f^{i+1}(\mathtt{a})|}{2} - 1] \cdot \mathtt{d} \cdot f^{i+1}(\mathtt{a})[\frac{|f^{i+1}(\mathtt{a})|}{2} + 1..|f^{i+1}(\mathtt{a})|],$$

and the statement holds for $i + 1$.

2. From the above arguments, the longest substring of $f^k(\mathtt{a})$ which is also a prefix is

$$f^k(\mathtt{a})[\frac{|f^k(\mathtt{a})|}{2} + 1..\frac{3}{4}|f^k(\mathtt{a})| - 1].$$

Thus the statement holds.

□

Lemma 12. *Let $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ be an ordered alphabet of size 4 s.t. $\mathtt{a} \prec \mathtt{b} \prec \mathtt{c} \prec \mathtt{d}$. For any integer $k \geq 0$, $LTree(f^k(\mathtt{a})) \equiv_I CBT_{2^k}$.*

Proof. We prove the lemma by induction on k. For $k = 1$ consider the string $f^1(\mathtt{a}) = \mathtt{ac}$, for $k = 2$, consider the string $f^2(\mathtt{a}) = \mathtt{acbc}$, and for $k = 3$, consider the string $f^3(\mathtt{a}) = \mathtt{acbcadbc}$. We can see the strings satisfy the statement of the lemma. Next, assume that the lemma holds for all $k \leq i$ for some i. We claim that $LTree(f^{i+1}(\mathtt{a})) \equiv_I CBT_{2^{i+1}}$ holds.

From Lemma 11,

$$f^{i+1}(\mathtt{a}) = f^i(\mathtt{a}) \cdot f^i(\mathtt{a})[1..\frac{|f^i(\mathtt{a})|}{2} - 1] \cdot \mathtt{d} \cdot f^i(\mathtt{a})[\frac{|f^i(\mathtt{a})|}{2} + 1..|f^i(\mathtt{a})|],$$

and thus

$$SA(f^{i+1}(\mathtt{a})[\frac{|f^{i+1}(\mathtt{a})|}{2} + 1..|f^{i+1}(\mathtt{a})|]) = SA(f^i(\mathtt{a})).$$

Thus

$$f^{i+1}(\mathtt{a})[\frac{|f^{i+1}(\mathtt{a})|}{2} + 1..|f^{i+1}(\mathtt{a})|]$$

is a Lyndon word and $LTree(f^{i+1}(\mathtt{a})[\frac{|f^{i+1}(\mathtt{a})|}{2}+1..|f^{i+1}(\mathtt{a})|]) \equiv_I LTree(f^i(\mathtt{a})) \equiv_I CBT_{2^i}$ holds. For any proper suffix w of $f^i(\mathtt{a})$, w is lexicographically larger than

$$f^{i+1}(\mathtt{a})[\frac{|f^{i+1}(\mathtt{a})|}{2} + 1..|f^{i+1}(\mathtt{a})|],$$

and $f^i(\mathtt{a})$ is lexicographically smaller than

$$f^{i+1}(\mathtt{a})[\frac{|f^{i+1}(\mathtt{a})|}{2}+1..|f^{i+1}(\mathtt{a})|].$$

From the induction hypothesis, $LTree(f^i(\mathtt{a})) \equiv_I LTree(f^{i+1}(\mathtt{a})[1..\frac{|f^{i+1}(\mathtt{a})|}{2}]) \equiv_I CBT_{2^i}$. Therefore, $LTree(f^{i+1}(\mathtt{a})) \equiv_I CBT_{2^{i+1}}$ and the statement holds for $k = i+1$. □

4 Conclusions and open question

We considered reverse engineering problems on Lyndon trees. We showed that: 1) For any full binary ordered tree T, there exists a solution string w over an alphabet of size at most $h+1$, where h is the height of T. 2) For any positive n, there exists a full binary ordered tree T with n leaves, s.t. the smallest alphabet size of the solution string for T is $\lfloor \frac{n}{2} \rfloor + 1$. We also conjectured that the trees described in Lemmas 7 and 8 are the only trees that satisfy the statements. We discovered the property on Lyndon trees which are isomorphic to complete binary trees.

Our remaining interest which is most important is an algorithm to reconstruct a Lyndon word s.t. the Lyndon tree is isomorphic to an input full binary tree and the alphabet size is smallest possible.

References

1. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta: *The "Runs" Theorem.* CoRR, abs/1406.0263 2014.
2. H. Bannai, S. Inenaga, A. Shinohara, and M. Takeda: *Inferring strings from graphs and arrays*, in Mathematical Foundations of Computer Science 2003, 28th International Symposium, MFCS 2003, Bratislava, Slovakia, August 25-29, 2003, Proceedings, 2003, pp. 208–217.
3. H. Barcelo: *On the action of the symmetric group on the free Lie algebra and the partition lattice.* Journal of Combinatorial Theory, Series A, 55(1) 1990, pp. 93–129.
4. F. Bassino, J. Clément, and C. Nicaud: *The standard factorization of Lyndon words: an average point of view.* Discrete Mathematics, 290(1) 2005, pp. 1–25.
5. B. Cazaux and E. Rivals: *Reverse engineering of compact suffix trees and links: A novel algorithm.* Journal of Discrete Algorithms, 28 2014, pp. 9 – 22.
6. K. T. Chen, R. H. Fox, and R. C. Lyndon: *Free differential calculus. iv. the quotient groups of the lower central series.* Annals of Mathematics, 68(1) 1958, pp. 81–95.
7. J. Duval: *Factorizing words over an ordered alphabet.* J. Algorithms, 4(4) 1983, pp. 363–381.
8. F. Franek, J. W. Daykin, J. Holub, A. S. M. S. Islam, and W. F. Smyth: *Reconstructing a string from its Lyndon arrays.* Theoretical Computer Science, 2017, p. in press.
9. F. Franek, S. Gao, W. Lu, P. Ryan, W. Smyth, Y. Sun, and L. Yang: *Verifying a border array in linear time.* J. Comb. math. Comb. Comput., 42 2002, pp. 223–236.
10. C. Hohlweg and C. Reutenauer: *Lyndon words, permutations and trees.* Theor. Comput. Sci., 307(1) 2003, pp. 173–178.
11. T. I, S. Inenaga, H. Bannai, and M. Takeda: *Inferring strings from suffix trees and links on a binary alphabet.* Discrete Applied Mathematics, 163, Part 3 2014, pp. 316 – 325, Stringology Algorithms.
12. J. Karkkainen, M. Piatkowski, and S. J. Puglisi: *String inference from longest-common-prefix array*, in Proc. ICALP, 2017, to appear.
13. M. Lothaire: *Combinatorics on Words*, Addison-Wesley, 1983.
14. R. C. Lyndon: *On Burnside's problem.* Transactions of the American Mathematical Society, 77 1954, pp. 202–215.
15. U. Manber and G. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM Journal on Computing, 22(5) 1993, pp. 935–948.

16. W. Matsubara, A. Ishino, and A. Shinohara: *Inferring strings from runs*, in Proceedings of the Prague Stringology Conference 2010, Prague, Czech Republic, August 30 - September 1, 2010, 2010, pp. 150–160.
17. T. A. Starikovskaya and H. W. Vildhøj: *A suffix tree or not a suffix tree?* J. Discrete Algorithms, 32 2015, pp. 14–23.

Trade-offs in Query and Target Indexing for the Selection of Candidates in Protein Homology Searches

Strahil Ristov[1⋆], Robert Vaser[2⋆⋆], and Mile Šikić[2,3]

[1] Ruđer Bošković Institute, Department of Electronics,
Bijenička 54, 10000 Zagreb, Croatia
[2] Faculty of Electrical Engineering and Computing,
Unska 3, 10000 Zagreb, Croatia
[3] Bioinformatics Institute,
Singapore 138671, Singapore
ristov@irb.hr, robert.vaser@fer.hr, mile.sikic@fer.hr

Abstract. We compare two recent similar and complementary indexing methods for fast seed discovery [10,12]. Both methods are based on the principle of counting matches on a diagonal with a goal to find the value and/or position of the best match between two sequences under Hamming distance on alphabet of k-mers, where k can equal 1. The matching k-mers in two sequences are found by scanning one sequence and using the index of the other. Indexing the shorter of the two sequences is easier to perform on-line; however, if the index is constructed off-line on the longer sequence, the number of comparison operation is potentially much smaller. We present the analysis of this effect for different real data sequence lengths in the context of protein search.

Keywords: protein sequence homology, string index, SWORD, Hamming distance vector, diagonal counting

1 Introduction

Finding the extent of the homology between two protein sequences is possibly the most important computational task in bioinformatics and biological sciences. The baseline results are obtained with the alignment algorithms such as Smith-Waterman (SW)[11] or Needleman-Wunsch [7]. Unfortunatelly, quadratic time complexity of the alignment algorithms renders them almost unusable when a large amount of sequences is compared. Therefore, different heuristic methods have been proposed throughout the years, from the early FASTA [8], BLAST [1] and BLAT [6] to the more recent ones such as Rapsearch2 [14] and DIAMOND [3]. They all share a common principle, i.e. decomposition of the alignment problem into seed and alignment phases. The former searches for seeds, i.e. subsequences of length k (k-*mers*) which are shared between a given pair of sequences, while the latter phase couples found seeds into local alignments based on different criteria. The seed phase is often implemented by creating an index containing all k-mers of the larger set of sequences, either stored on the hard drive or created on the fly. Only exception is the DIAMOND algorithm which uses double indexing, i.e. indexes are created for both query and target sequences and seeds are obtained by linearly traversing both indexes at the same time [3].

⋆ Supported in part by the Croatian Science Foundation under projects IP-11-2013-9623 and IP-11-2013-9070.
⋆⋆ Supported in part by the Croatian Science Foundation under project UIP-11-2013-7353.

Strahil Ristov, Robert Vaser, Mile Šikić: Trade-offs in Query and Target Indexing for the Selection of Candidates in Protein Homology Searches, pp. 118–125.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

A recently developed method SWORD [12] applies a different approach, i.e. in the search stage equal k-mers are found in order to obtain a measure of similarity for a pair of sequences which further enables running the optimal alignment algorithms on only a small subset containing the most similar sequences. The measure of similarity between two sequences involves counting the number of matches at the best overlap of the sequences. To facilitate this, SWORD employs an index that is built on the fly on the smaller sequence, or the set of sequences (the query set). In another recent paper, a similar and complementary method has been proposed for the efficient Hamming vector calculation [10]. The two methods are similar, as they are both based on finding the best diagonal, i.e. the position at which two sequences should overlap in order to have the highest number of matching symbols aligned, and complementary in the choice of which sequence is indexed. The index is constructed on query in [12], and on target in [10]. In this paper we investigate the potential of combining the SWORD method with the indexing of the target.

2 Counting hits on a diagonal in sequence pattern matching

The principle of counting matches on a diagonal (or *diagonal counting* for short) was used in computational biology applications as early as in 1983 [13]. The goal is to find the greatest number of matches, without insertion or deletions, between two sequences and the corresponding position(s), i.e. the relative position, one or more of them, between the two sequences where the number of matches is the largest. This information can further be used for the location of possible good alignments as in [13,8] or to score the whole sequences as in [12]. Besides in bioinformatics applications, the underlying principle of counting matches at a given offset from the beginning of the sequence has been used in general string matching problems such as k-mismatch problem in [2] and Hamming distace vector calculation in [10]. Incidentally, finding the number of matches on all diagonals amounts to finding the inverse of the Hamming distance vector for two sequences.

Diagonal counting consist of sliding one sequence over the other and at each position counting the number of aligned symbols that are equal. The alphabet of symbols can consist of single characters, or of *k-grams*, i.e. k consecutive characters. In the context of protein sequences k-grams are usually called k-mers.

The brute force procedure would be to slide one sequence over the other and iteratively test for matches. If m and n denote the lengths of the two sequences that are matched, the complexity of brute force approach is strictly quadratic $\Theta(mn)$. However, instead of comparing the symbols at all positions in two sequences, it is possible to use indexing of one sequence to reduce the number of comparisons, as there is no need to check the positions where no match exists. Indeed, the indexing has been combined with the diagonal counting from the earliest works in bioinformatics [13,4].

In the two recent articles by the authors the diagonal counting with indexes has been used to optimize search for protein homology [12], and for efficient calculation of Hamming distance vector for protein sequences [10]. A salient difference between methods described in [12] and [10] lies in the choice of the object of indexing. The method described in the first article uses index on shorter (query) sequence and constructs it on-line, while the method described in the second article employs the index on the longer (target) sequence which is constructed off-line. We shall describe both algorithms in some detail in the following section.

3 Query vs. target indexing

3.1 SWORD algorithm and query indexing

The SWORD [12] is an efficient algorithm for protein database search that aims to offer better speed vs. sensitivity ratio than BLAST, a gold standard in sequence homology search [1]. Same as BLAST, and as well as most other algorithms for finding the homology in biological sequences, SWORD uses the two stage approach: the first stage is reducing the whole database to a subset of probable candidates based on some heuristics, and the second stage is performing the full scale alignment using the standard SW [11] or, in some cases, the Needleman-Wunsch [7] method. However, while BLAST in the first phase searches for hits - short matching segments in sequences, and then expands these hits into local alignments, the SWORD method finds the best whole candidate sequences from the database of sequences, and performs the SW alignment on the whole sequences. The effect is that less work is invested in processing of all potential positions for a good local alignment at the expense of performing SW on longer inputs. The increased work on SW alignment is compensated by using fast parallel processing.

The first phase of determining the best candidate sequences is performed using the diagonal counting. For a query sequence and each protein sequence in the target database the algorithm calculates the highest value of any diagonal when matching the two sequences at every position. The fixed number of sequences with the highest score are then forwarded to the alignment phase, which is a fast parallel SW implementation based on SIMD (Single Instruction Multiple Data) instructions [12].

For the purpose of this work we shall consider only the first, heuristic, stage of SWORD method. The diagonal counting in SWORD is performed on the alphabet of k-mers, with the different values of k producing different speed vs. sensitivity ratio. Larger values lead to greater speed and lower sensitivity. To increase sensitivity, matches can include not only the identical k-mers but also those that are similar enough according to a given amino acid similarity matrix. In [12] k is proposed to be 3 with included similar k-mers for the best sensitivity, or 5, with exact matches only, for the greatest speed. The published version of the code can be found at https://github.com/rvaser/sword. A threshold for similarity T, when $k = 3$, is set to $T = 13$ using BLOSUM62 substitution matrix [5].

To perform fast counting of matches, SWORD uses index on the query sequence. The index is constructed as a perfect hash table that for each different k-mer returns the list of the corresponding positions in the sequence. In case of sensitive search, for each k-mer in the query, similar k-mers are generated and stored in the index.

The SWORD uses diagonal counting to optimize the choice of candidates in search for protein homology, and the value of the highest score has proven to be an adequate criterion for the choice of candidates. According to the results presented in [12], SWORD can be regarded as a viable alternative to BLAST, it is overall considerably faster while remaining comparatively sensitive to distant homologies. However, the actual times needed for the processing of complex inputs can be considerable. For instance, matching the complete E. Coli proteome to NCBI NR protein data base requires approximately 4 hours with SWORD (an order of magnitude less than with BLAST) which is a motive for further research on a possible speed up of the algorithm.

3.2 The potential advantage of target indexing

The indexing of target strings in the context of diagonal counting has been proposed in [10] with the application in the Hamming vector calculation where query is, as a rule, much shorter than the target. The complexity of the matching in the average case is then given with:

$$O(mn \sum_{a \in A} p_a^2) \tag{1}$$

where A is the alphabet, and p_a is the probability of a symbol $a \in A$ in the target sequence. In the case when all symbol probabilities are equal (1) reduces to:

$$O(mn/|A|) \tag{2}$$

The effectiveness of this approach is based on the sparse distribution of symbols in shorter sequences when the alphabet is large. Solely the symbols that are present in query are accessed in the index. As long as the length of query remains small compared to $|A|$, indexing the target can considerably reduce the total number of operations. Only when all symbols from the alphabet are present in the query, the number of index access operations is the same as when the index is constructed on the query. The downside of target indexing is that it has to be done off-line since target is, as a rule, much larger than query.

Let us consider an alphabet of k-mers formed from 20 different amino acids. At the first sight it would appear that with the sizes of the alphabet that equal 20^k there is a great probability that a significant number of symbols will not be present in the shorter query sequence. However, with protein sequences, and when using the SWORD method that generates similar k-grams at each position, the number of different symbols in the query quite rapidly reaches $|A|$.

3.3 Generalized procedure for counting matches on a diagonal

Generalized statement of the best diagonal problem is: Finding the position(s) of the highest scoring match between two sequences under Hamming distance metrics on the alphabets of $k-mers$. If $k = 1$, the problem reduces to a simple inverse Hamming distance vector calculation. In order to avoid quadratic matching of every position in both sequences, the simple solution is to index one sequence and scan the other. Since the mechanism is the same regardless of which sequence is indexed, we give a general pseudo code in Algorithm 1 where query and target can be freely interchanged. The notation used in Algorithm 1 is as follows: S_{idx} and S_{scan} denote the indexed and the scanned sequence, respectively; the respective lengths of the sequences are denoted with $|S_{idx}|$ and $|S_{scan}|$; $|A|$ is the size of alphabet A, where the symbols in the alphabet can be single characters or k-mers; $PositionList[a]$ stores positions of all instances of a in S_{idx}, for all symbols $a \in A$; $MatchVector$ stores the scores for $|S_{idx}| + |S_{scan}| - 1$ diagonals in an alignment array between the two sequences.

After the construction of $MatchVector$ it is easy to find, in one pass, the value and the position(s) of the best match(es). Index is an inverted file of indexed sequence and its construction is linear with $|S_{idx}|$. If the scanned sequence includes symbols that are not present in the indexed sequence, there will be unnecessary index accesses. In the next section we give experimental results on how long can a query protein sequence be to justify the target indexing approach.

Algorithm 1: *PositionList* and $MatchVector$ are initialized to zeros; $S_{idx}[i]$ and $S_{scan}[i]$ denote character at position i in S_{idx} and S_{scan}, respectively

```
1 int* PositionList[|A|];
2 int MatchVector[|S_idx| + |S_scan| − 1];
  /* pre-processing phase                                   */
3 for i = 0 to |S_idx|-1 do
4     add i to PositionList[S_idx[i]];
  /* Match vector computation                               */
5 for i = 0 to |S_scan|-1 do
6     for j in PositionList[S_scan[i]] do
7         MatchVector[j - i + |S_scan| − 1] ++;
```

4 Experimental results

Off-line indexing of a longer target sequence is justified when alphabet symbols are sparse in the query sequence. In such cases the reduced number of index access operations compensates for longer processing time needed for the index construction. The size of the fraction of the target alphabet that is present in the query depends on the size of the alphabet and the query length. Obviously, as the length of the query increases, more and more symbols are present. Let $pA_{t/q}$ denote the percentage of target alphabet symbols present in query sequence. In order to asses the possible speedup of SWORD algorithm that could be achieved with target indexing we have performed experimental analysis to obtain insight on what are the real values of $pA_{t/q}$, with different query lengths, on the alphabet of amino acids and real protein sequence data. We have employed the heuristic part of SWORD algorithm that scans the target and looks up the found k-mers in the index of query sequence. We counted the number of target k-mers that were found in the index and calculated the $pA_{t/q}$ percentage. The fraction of unsuccessful index accesses is the direct measure of the gain (i.e. reduction of the number of index accesses) that can be achieved using target indexing. In all our experiments all k-mers present in query were also present in the target, therefore with target indexing every index access would be successful.

The targets were different size subsets of the UniProt database [9], and the queries were random protein sequences of three different sizes, taken from the target. In Tables 1-6 we report the results for query lengths of 100, 500 and 1500 amino acids, and target data sets that are comprised of 10^4, 10^5 and $5{\times}10^5$ lines, where each target line is a different protein from UniProt database. The sizes of the targets are $3.7{\times}10^6$, $37.5{\times}10^6$, and $181{\times}10^6$ amino acids, respectively. Tables are organized according to the value of k, and present results for the exact k-mer matching as well as the matching with the expanded number of k-mers that include all similar k-mers within the given similarity threshold. The values in column *tested* represent the number of k-grams present in the target that are tested against the index of a query. These values are roughly equal to the sizes of the target sets. The values in column *matches* are the numbers of k-mers found in the query index. The values of $pA_{t/q}$, i.e. the percentage of tested k-mers that are actually matched are given under % sign.

Following the findings in the SWORD paper we have experimented with $k = 3$ and $k = 5$. The results for $k = 3$ are presented in Table 1 for the exact matching and in Table 2 for the expanded matching with similarity threshold used in SWORD algorithm. The results for exact matching with $k = 5$ are presented in Table 3. The

Table 1. k = 3, exact matching

	10^4 lines			10^5 lines			5×10^5 lines		
query length	tested	matches	%	tested	matches	%	tested	matches	%
100	3751688	95199	2.5	37557230	1008318	2.7	181015261	5013396	2.8
500	""	508787	13.6	""	5313058	14.1	""	26098594	14.4
1500	""	1272292	33.9	""	13257314	35.3	""	64529580	35.6

Table 2. k = 3, expanded matching with similar k-mers (BLOSUM62 similarity ≥ 13)

	10^4 lines			10^5 lines			5×10^5 lines		
query length	tested	matches	%	tested	matches	%	tested	matches	%
100	3751688	194925	5.2	37557230	1980520	5.3	181015261	9724152	5.4
500	""	895241	23.9	""	9044466	24.1	""	44277325	24.5
1500	""	3293569	87.8	""	37331149	99.4	""	159200790	87.9

Table 3. k = 5, exact matching

	10^4 lines			10^5 lines			5×10^5 lines		
query length	tested	matches	%	tested	matches	%	tested	matches	%
100	3731688	550	0.0	37357230	5259	0.0	180015261	27047	0.0
500	""	3113	0.1	""	24729	0.0	""	128832	0.1
1500	""	6883	0.2	""	59175	0.2	""	269872	0.1

Table 4. k = 5, expanded matching with similar k-mers (BLOSUM62 similarity ≥ 20)

	10^4 lines			10^5 lines			5×10^5 lines		
query length	tested	matches	%	tested	matches	%	tested	matches	%
100	3731688	7759	0.2	37357230	80798	0.2	180015261	394591	0.2
500	""	35800	1.0	""	357282	1.0	""	1789459	1.0
1500	""	181056	4.9	""	1788243	4.8	""	8616060	4.8

Table 5. k = 5, expanded matching with similar k-mers (BLOSUM62 similarity ≥ 22)

	10^4 lines			10^5 lines			5×10^5 lines		
query length	tested	matches	%	tested	matches	%	tested	matches	%
100	3731688	2516	0.1	37357230	24954	0.1	180015261	123087	0.1
500	""	9644	0.3	""	93423	0.25	""	469165	0.3
1500	""	54624	1.5	""	544963	1.5	""	2584125	1.4

Table 6. k = 5, expanded matching with similar k-mers (BLOSUM62 similarity ≥ 24)

	10^4 lines			10^5 lines			5×10^5 lines		
query length	tested	matches	%	tested	matches	%	tested	matches	%
100	3731688	951	0.0	37357230	9830	0.0	180015261	49270	0.0
500	""	4154	0.1	""	36207	0.1	""	189052	0.1
1500	""	20002	0.5	""	178632	0.5	""	854227	0.5

results in Tables 1-3 cover all indexing variants used in SWORD. Expanded matching with $k = 5$ would incur considerable processing overhead for on-line construction of query index. Especially so if the similarity threshold is set for higher sensitivity. However, if we accept the necessity of building the index off-line, we are not constrained to expanding only 3-mers. Off-line index can be constructed for any k and threshold that may result in good speed vs. sensitivity ratio. As an example, in Tables 4-6 we

have included results for the expanded matching with $k = 5$ and similarity thresholds of 20, 22, and 24, respectively.

The results show that with 3-mers the query alphabet comes close to saturation with the expanded matching when the length of the query is 1500 amino acids (Table 2). In such cases indexing of the target would not be economical. On the other hand, with the exact matching on 3-mers (Table 1), and both the exact and expanded matchings on 5-mers (Tables 3-6) the results are much more promising. Using target index in those cases can considerably reduce the number of index accesses. We find of particular interest the results presented in Table 4. Even with the expanded 5-mers, and a low threshold of similarity, indexing the target can reduce the number of index accesses by the factor of 20 with 1500 amino acids long query.

4.1 Discussion of the results

The heuristic phase of SWORD consumes approximately half of the total processing time. In this paper we present the preliminary findings on which we will base further investigation of possible speedup. To exploit the benefits of target indexing the query must be short. In the current version of SWORD algorithm multiple queries are combined in one query index. This could possibly be modified in a way to reduce the query alphabet saturation. Therefore, the potential for the improvement exists but it has to be investigated further.

The results obtained on 5-mers open the possibility of further investigation to establish the level of sensitivity with expanded k-mers with larger k and different thresholds. The speed of SWORD algorithm is equally the result of a careful implementation regarding the cache efficiency. Indexing expanded k-mers increases the index size by an order of magnitude i.e., 7 to 13 times in our experiments. This rises requirements on the design of the data structures for index storage and access. Obviously, cache friendly, succinct and localized data structures should be employed for storing the index.

5 Conclusions

We have compared two complementary methods of indexing for finding the best match between two sequences under Hamming distance: one where the index is constructed on a query sequence and scan is performed on a target, and one where the index is constructed on a target sequence and scan is performed on a query. Both approaches reduce the number of comparisons with respect to the brute force approach and produce the same final result. A query is, as a rule, much shorter than the target, and the advantages of query indexing are the possibility to perform it on-line and the low space requirements for the index. On the other hand, if the target is indexed off-line, the number of index accesses is restricted to the length of the query, which can significantly reduce the total number of matching operations. This effect is dependent on the length of the query and the percentage $pA_{t/q}$ of alphabet symbols represented in the query. We have performed the experiments to obtain the insight into actual values of the query length and $pA_{t/q}$ within the framework of the heuristic part of SWORD algorithm for protein search. Somewhat surprisingly, we have found out that in the core SWORD variant, when amino acid triplets are expanded with similar 3-mers, almost all of the alphabet is present in a query of length 1500. As a result, target indexing cannot be straightforwardly implemented. On the other hand, the

results on the longer k-mers show much more promise. It is apparent that using longer k-mers with larger similarity neighborhood can lead to strong reduction in the number of total matching operations. However, to explore this fact, further work is required in finding the appropriate cache efficient data structures for storage of the larger index, as well as the modification of SWORD algorithm in order to work with shorter queries.

The actual possible speedup of SWORD method will probably have more to do with cache efficiency related data handling than with string algorithms. Nevertheless, the underlying mechanism could very well be based on the findings presented in this investigation. To our knowledge, this is the first analysis of that kind and we hope it may be useful to designers of algorithms based on protein sequence indexing.

References

1. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman: *Basic local alignment search tool.* Journal of Molecular Biology, 215(3) 1990, pp. 403 – 410.
2. A. Amir, L. M., and P. E.: *Faster algorithms for string matching with k mismatches.* Journal of Algorithms, 50(2) 2004, pp. 257 – 275, Soda 2000 Special Issue.
3. B. Buchfink, C. Xie, and D. H. Huson: *Fast and sensitive protein alignment using DIAMOND.* Nature Methods, 12(1) 2015, pp. 59–60.
4. J. P. Dumas and J. Ninio: *Efficient algorithms for folding and comparing nucleic acid sequences.* Nucleic Acids Research, 10(1) 1982, pp. 197–206.
5. S. Henikoff and J. Henikoff: *Amino acid substitution matrices from protein blocks.* Proceedings of the National Academy of Sciences of the United States of America (PNAS), 89 1992, pp. 10915–10919.
6. W. J. Kent: *BLAT - The BLAST-Like Alignment Tool.* Genome Research, 12(1) 2002, pp. 656–664.
7. S. B. Needleman and C. D. Wunsch: *A general method applicable to the search for similarities in the amino acid sequence of two proteins.* Journal of Molecular Biology, 48(3) 1970, pp. 443 – 453.
8. W. R. Pearson and D. J. Lipman: *Improved tools for biological sequence comparison.* Proceedings of the National Academy of Sciences of the United States of America, 85(8) 1988, pp. 2444–2448.
9. The UniProt Consortium: *Uniprot: a hub for protein information.* Nucleic Acids Research, 43(D1) 2015, pp. D204–D212.
10. S. Ristov: *A fast and simple pattern matching with hamming distance on large alphabets.* Journal of Computational Biology, 23(11) 2016, pp. 874–876.
11. T. Smith and M. Waterman: *Identification of common molecular subsequences.* Journal of Molecular Biology, 147(1) 1981, pp. 195 – 197.
12. R. Vaser, D. Pavlovic, and M. Sikic: *SWORD - a highly efficient protein database search.* Bioinformatics, 32(17) 2016, pp. 680–684.
13. W. Wilbur and D. Lipman: *Rapid similarity searches of nucleic-acid and protein data banks.* Proceedings of the National Academy of Sciences of the United States of America - Biological Sciences, 80(3) 1983, pp. 726–730.
14. Y. Zhao, H. Tang, and Y. Ye: *RAPSearch2: a fast and memory-efficient protein similarity search tool for next-generation sequencing data.* Bioinformatics, 28(1) 2012, pp. 125–126.

Many-MADFAct: Concurrently Constructing MADFAs

Tobias Runge[1], Ina Schaefer[1], Loek Cleophas[2,3], and Bruce W. Watson[2,4]

[1] Software Engineering, TU Braunschweig, Germany
[2] Information Science, Stellenbosch University, South Africa
[3] Software Engineering Technology, Eindhoven University of Technology, The Netherlands
[4] Centre for Artificial Intelligence Research, CSIR, South Africa
{tobias.runge,i.schaefer}@tu-bs.de, {loek,bruce}@fastar.org

Abstract. Minimal acyclic deterministic finite automata (MADFAs) are used to represent dictionaries, i.e., finite sets of finite words, in, e.g., spell checkers and network security applications. Given the size of such dictionaries, which may contain millions of words, their efficient construction is a critical issue. Watson [31] published a classification of such algorithms in an algorithm taxonomy with correctness arguments. We report on a new algorithm which constructs MADFAs in parallel—each for a keyword set from a partition of the original keyword set—and afterwards merges and minimizes the resulting automata into a single MADFA; on our experience implementing the algorithms in a Java-based toolkit; and on empirical performance results obtained.

Keywords: minimal acyclic deterministic finite automata; dictionaries; algorithms; toolkits; benchmarking

1 Introduction

Minimal acyclic deterministic finite automata (MADFAs) are frequently used to represent dictionaries, i.e., finite sets of finite words, in, e.g., spell checkers, network security, packet filtering applications, and other tools. Given the size of such dictionaries, which may run into millions of words, their efficient construction is a critical issue [31]. Acyclic Deterministic Finite Automata (ADFAs) consist of a set of states, one of them being a start state, and one or more of them being final states; and labeled transitions between states. They are deterministic, i.e., each state has at most one out-transition for a specific label; and they are acyclic, i.e. as their name says, no transition cycles occur. An ADFA is a MADFA if no other ADFA with fewer states accepts the same set of words. This makes MADFAs an excellent data structure to store large finite word sets like dictionaries. As a result, quite some research has gone into MADFA construction algorithms (see Section 2). Yet, no coherent implementation covering all these algorithm variants exists.

In this paper, we make the following contributions: we provide a two-dimensional presentation of Watson's implicit taxonomy of sequential MADFA construction algorithms [31], as well as an implementation of the seven sequential MADFA algorithms from that taxonomy in a Java-based toolkit. Furthermore, we develop a new parallel approach to MADFA construction, offering a versatile family of algorithms for MADFA construction in contexts where concurrent processing is available or preferred. Using one of the existing, sequential algorithms, the new algorithm constructs MADFAs in parallel—one for each of a partition of the original keyword set—and afterwards merges and minimizes the resulting automata into a single MADFA. The merger process is newly implemented, but the minimization step is the same as that

Tobias Runge, Ina Schaefer, Loek Cleophas, Bruce W. Watson: Many-MADFAct: Concurrently Constructing MADFAs, pp. 126–142.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

in one of the existing, sequential MADFA algorithms. Finally, we provide the results of benchmarking the algorithms to evaluate their performance relative to each other.[1] The work reported here contrasts with related work as follows: typically, there are several implementations of each of the known algorithms (see the next section), but previously only one publicly available comprehensive toolkit (that by Jan Daciuk, available at `www.jandaciuk.pl/fsa.html`); our contribution is another such toolkit, built from a uniformly styled presentation of the algorithms [31]; previously, there have also been few comprehensive benchmarks where the algorithms are implemented in the same style and emperically compared against each other.

2 Related work and a short history

The following history is distilled from [31], which is until now the most comprehensive such collection of derivations of MADFA construction algorithms. Jan Daciuk maintains implementations of many MADFA construction algorithms and has authored what is arguably the most comprehensive work on optimization, minimization and implementation/engineering issues as related to automata [9] in addition to his extensive algorithmic work in this field (detailed below).

Before the 1990s, some MADFA construction algorithms may have been known and used in proprietary (commercial, trade-secret) software. The first efficient (linear time and space) algorithm was published by Dominique Revuz in the early 1990s [21,22]. Revuz's main algorithm uses an ordering of the words to quickly compress the endings of the words within the dictionary. Recent derivations by Johannes Bubenzer and Thomas Hanneforth have yielded efficient new algorithms bearing a resemblance to Revuz's [3]. These algorithm variants are essentially what appears as *Algorithm Trie* in Fig. 1; in that figure, *Algorithm General* is a generalized version of Revuz's algorithm, first presented in [31].

By the mid-1990s, several groups were working independently on *incremental* MADFA construction algorithms. In 1996–1997, Jan Daciuk derived several incremental algorithms as part of his PhD work [8]: one relying on the words being in lexicographic order. In 1996, Richard and Bruce Watson derived a generalized incremental algorithm, which included the possibility of incrementally removing words while maintaining minimality; owing to its commercial value, the algorithm was not published at that time. Collaboration between Daciuk, Watson & Watson led to [11]. More or less concurrently, Stoyan Mihov PhD work derived parts of the same algorithms [18], and further collaboration yielded [10] by Daciuk, Mihov, Watson & Watson. In the domain of pattern matching, Park *et al* derived a similar algorithm [19], while in program verification, Gerard Holzmann and Anuj Puri [14] discovered a restricted form of the algorithm, in which all words accepted by the automaton are the same length. In early 2000, Daciuk unearthed the derivational work of Sgarbas *et al* (an incremental algorithm [24]) and Marcin Ciura and Sebastian Deorowicz (lexicographic order algorithm, including some benchmarking [5]). Also in 2000, Revuz presented essentially the generalized algorithm [23]—though he also sketched word deletion algorithms similar to those previously derived by Watson & Watson. Jorge Graña *et al* subsequently summarized some of the current results and made improvements to several of the algorithms [13]. The generalized algorithm has also been extended by Rafael Carrasco and Mikel Forcada to handle *cyclic* automata [4]. In

[1] We are not focusing on absolute performance or on further tuning of the algorithms.

this paper, the generalized incremental algorithm is *Algorithm Incremental* in Fig. 1, while the sorted-input algorithm is *Algorithm Sorted* in the same figure. An alternative sorted-input algorithm (based on arbitrary sortings of decreasing lengths of the words) was developed in [31, Chapter 10], and appears in Fig. 1 as *Algorithm Depth Layered*.

In 1998, Watson derived a semi-incremental MADFA construction algorithm [29]. Such an algorithm does a form of pseudo (or *near*) minimization incrementally as words are added; after all words are added, a final 'cleanup' phase is required to reach true minimality. This is *Algorithm Semi-Incremental* in Fig. 1. In the same year, Watson used Brzozowski's minimization algorithm to give an elegant MADFA construction algorithm in [27,28] (which maps to *Algorithm Reverse* in Fig. 1). Concurrently, Watson derived a simple recursive algorithm in [30]; that algorithm does not appear separately in this paper's work, as it is a variant of *Algorithm Incremental*. Aside from [31], to which this paper relates, early taxonomies/classifications appeared [26].

3 Taxonomy and Algorithms

Algorithm taxonomies hierarchically structure algorithms stemming from a domain, in order to facilitate comparison and emphasize algorithms' similarities. The root of such a classification is formed by an abstract algorithm, and branches refine a parent algorithm into more concrete child algorithms. As such, by proving or at least considering the correctness of each such branch or refinement, one can prove or convince oneself of the correctness of each and every algorithm in the taxonomy.

Algorithm taxonomies have been in use since at least the 1970s; for example, Darlington [12] and Broy [1] classified different sorting algorithms, while Jonkers [15] classified garbage collection algorithms, and did so with an emphasis on correctness. Building on Jonkers' style, Marcelis considered attribute evaluation algorithms [16], while Watson presented taxonomies of string pattern matching and automata related algorithms [25]. Cleophas [6] similarly treated tree pattern matching and automata. Pieterse, going beyond just taxonomies, recently published a thesis on the use of topic maps for structuring algorithmic knowledge, including a topic map and taxonomy of transitive closure algorithms [20].

Algorithm taxonomies can also form the starting point for the development of implementations, as is done in the TABASCO method [7] where toolkit implementations of the taxonomised family are derived from the taxonomies. The taxonomies' structure guides that of a corresponding toolkit, including ensuring reuse of common algorithm parts and hence common implementation parts. The correctness arguments contained in the taxonomies provide confidence in the correctness of the implementations as well.

3.1 A two-dimensional taxonomy of MADFA construction

The seven known MADFA construction algorithms have been classified hierarchically in a taxonomy, to facilitate comparison, highlight similarities, and reason about correctness [31]. The root represents an abstract model of the algorithms, with methods *add_word* to add an individual word—resulting in a not necessarily minimal ADFA—and *cleanup* to ensure the final result is again a MADFA. The method *add_word* is called for every word in the set of input words, after which a call to *cleanup* minimizes

the ADFA to a corresponding MADFA. Some algorithms do not follow the separation between adding words and *cleanup*; these algorithms partially minimize the automaton during *add_word* and need just a single, final call to *cleanup*. In Figure 1, we show a taxonomy graph, conceptually representing the MADFA construction taxonomy that was left somewhat implicit in [31]. The algorithms are depicted as circles. They always have one link to an *add_word* method and one to a *cleanup* method. Both methods are depicted as rectangles. The connectors between the methods show the hierarchy. Algorithm-Skeleton has the two abstract methods *add_word*-Skeleton and *cleanup*-Skeleton; and both are refined by the specific algorithms.

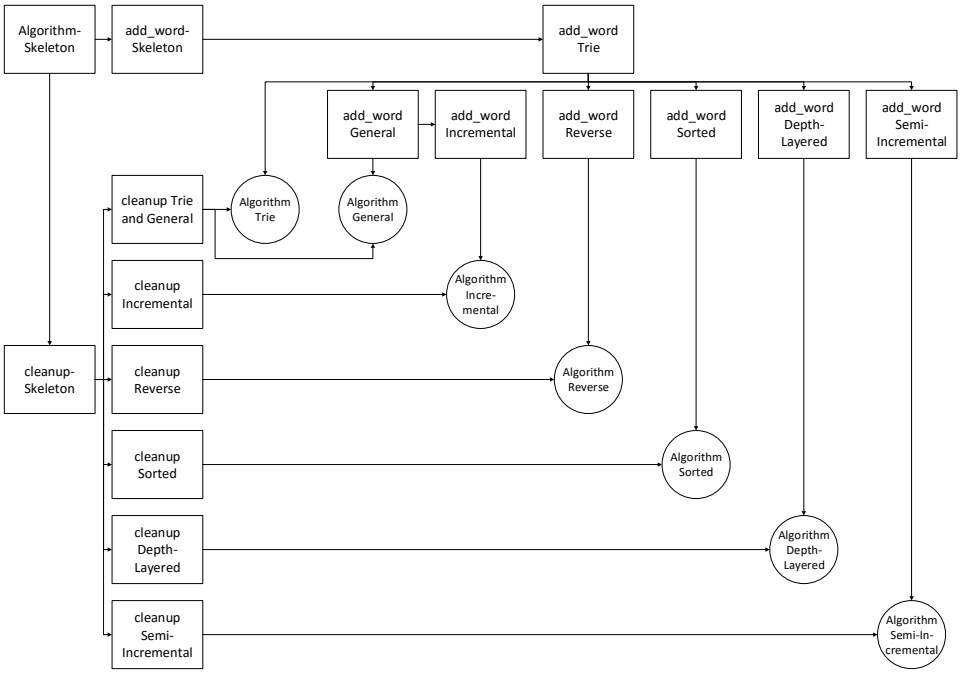


Figure 1. Taxonomy of the sequential MADFA construction algorithms

The method *add_word* is classified as follows. *add_word* of Algorithm *Trie* is the base for all the other algorithms. This method adds words by adding a path for every new word. The result is a trie. The Algorithm *General* extends the process such that the method is applicable for arbitrary ADFAs instead of just tries. The other four algorithms directly connected to Algorithm *Trie's add_word* method extend this *add_word* method with a minimization step. Finally, the Algorithm *Incremental* inherits from *General* directly and also adds a minimization step to this method. For the method *cleanup*, we can not find many commonalities between the different variants of the algorithms. All the considered algorithms have different *cleanup* methods, except for Algorithm *Trie* and *General*, which both use the same *cleanup* method. As a result, the method *add_word* is related in each algorithm and can be refined hierarchically, but the method *cleanup* differs between most algorithms.

As we will see in Section 3, a novel, parallel approach was developed which uses parallel threads to construct MADFAs and finally merge them into a single MADFA. This approach forms a whole new family of MADFA construction algorithms, as in each of the parallel MADFA construction threads, any of the algorithms from the above taxonomy can be used. The algorithm family corresponding to this parallel

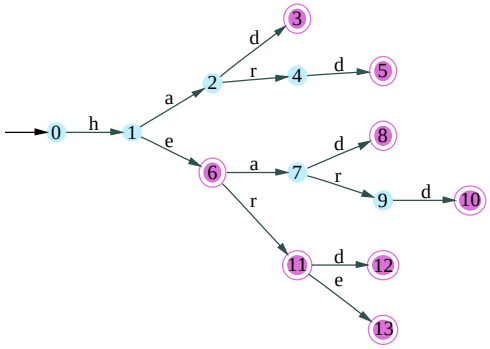
approach is not shown in Figure 1 because the parallelism is orthogonal to the algorithmic solution strategies of the algorithms shown in the figure. In the parallel algorithm, two or more threads call one of the preceding seven algorithms to construct a MADFA in parallel—each for a keyword set such that these sets form a partition of the original keyword set. The chosen algorithm may even be different per thread, since each such algorithm guarantees a MADFA to be constructed. After the construction of the separate MADFAs, a merge method merges these MADFAs into a single ADFA. This ADFA is then minimized with a call to the *cleanup* method of Algorithm Trie, as this *cleanup* method can be used for arbitrary ADFAs.

3.2 Algorithms

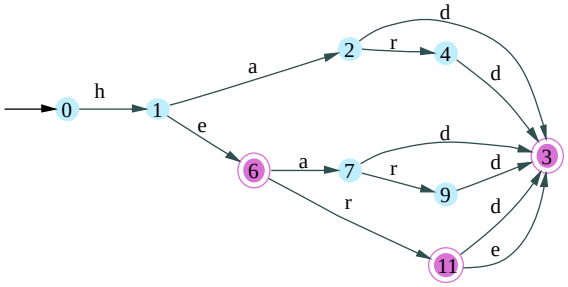
We briefly discuss four algorithms (Algorithms *Trie*, *Incremental*, *Reverse*, and *Semi-Incremental*) to give some insight into the behavior of MADFA construction algorithms. (More extensive examples of all these algorithms in action can be found throughout [31].) Algorithm *Trie* can be seen as the base algorithm for all the other ones. It realizes the abstract methods *add_word* and *cleanup*. At first, all words are added one by one by calling the method *add_word*. This is done by traversing the automaton according to the word under consideration, until no out-transition is found for a specific letter of the word; then the automaton is extended with new states and transitions so that it accepts the word. The result is a trie, which is then minimized to a MADFA with the help of *cleanup* [31]. This *cleanup* method is a Revuz-like algorithm [22]: it merges equivalent states of the automaton in order of decreasing height level. (A height level is a set of states which have the same length of their longest path to any final state.) The method starts with the leaves of the trie and ends at the root. Two states are equivalent if they have the same right language, i.e., they have the same set of words leading to final states. If that is the case, the states can be merged. During the merge, one state is deleted and its transitions are redirected to another state. If no more states are equivalent, no states can be merged, and therefore the automaton is minimal, i.e., in our context is a MADFA.

In Figure 2 we show an example of how this algorithm works. Firstly, all words are added. In this example the order of the words is lexicographic, i.e. had, hard, he, head, heard, her, herd, here. For every word, the automaton is traversed, and if necessary, new transitions and states are added. To give an example, 'head' is added after 'he' and before 'her' etc., i.e., state 6 is final and has no out-transitions before 'head' is added. During *add_word*, the automaton is traversed to state 6, following the letters 'h' and 'e'. Now, we are at a state with no out-transition for the next symbol, 'a'. We need to add a transition 'a' to a new state 7 and from there we add a transition 'd' to a new final state 8. Such a process is followed for every word. The result is the ADFA in Figure 2a. To minimize the automaton, Algorithm *Trie* computes height levels and merges equivalent states. In this example the first height level is the set of all states that have no out-transitions, i.e. their longest path to a final state is 0. The states $3, 5, 8, 10, 12, 13$ belong to this set. Every state is equivalent to the other states because every state is final and has no out-transition. That is why all states are merged into state 3 in Figure 2b. The next height level consists of all states with a longest path-length of one to a final state. This set includes $4, 9, 11$. Again, equivalent states are merged (not depicted), i.e. state 4 and 9 are merged, while 11 is not (both because it differs from 4 and 9 in out-transitions, and because it does so in its finality). Afterwards, the height level with a path-length of two is created

and equivalent states in it are merged, and so on, until the resulting automaton is a MADFA.



(a) Trie after adding words



(b) First minimization step

Figure 2. Example for Algorithm Trie

The other six MADFA construction algorithms use a similar method to add words but with some specific extensions. Algorithm *Incremental* for example minimizes the automaton directly after each word is added. States on the newly added path are compared with the other states of the automaton and equivalent states are merged. The comparison starts at the end of the path and ends at the start state. *cleanup* only is a skip statement since the automaton is minimized during *add_word* [8,17]. A characteristic of this algorithm is that sometimes states have to be cloned, so that the automaton stays correct.

Algorithm *Reverse* is different from the other ones, in that *add_word* adds words in reverse order compared to the *add_word* method of Algorithm *Trie*. The resulting ADFA is a trie for the reverse of the words; because of that, *cleanup* must reverse and determinize the whole ADFA to obtain a MADFA. (In essence, this is a specialization of Brzozowski's classical result for DFA minimization [2]).

Algorithm *Semi-Incremental* also uses the *add_word* method of Algorithm *Trie*, but it adds words in order of decreasing length; hence the final state added by calling *add_word* is never visited again and all successors of this state can already be considered for merging. These are all the states that are compared with other final states and their successors. This is done during the *add_word* method. The method *cleanup* visits the last non-considered successor states of the start state, which are all states that do not have a predecessor final state [31]. The number of states compared by the *cleanup* method depends on the input word set. In Figure 3 we give an example. We add the word 'herd' after 'heard' because 'herd' is shorter than 'heard'. The new states 6 and 7 are added, the second being final. The result is the upper automaton in Figure 3. *add_word* starts to merge afterwards. The new final state and its succes-

sors are compared against all other final states and their successors. In this example, state 5 and 7 are compared. They are equivalent and because of that they are merged into one state. The result is the second automaton in Figure 3. The next step of this algorithm is to add other words and merge final states until the entire input set has been processed.

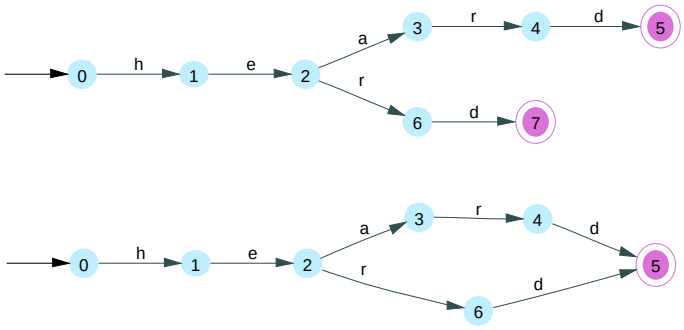


Figure 3. Example Algorithm Semi-Incremental: Adding word 'herd'

4 Parallel MADFA Construction

The seven MADFA construction algorithms [31] included in our taxonomy construct MADFAs sequentially. We present a novel approach to MADFA construction here, based on constructing multiple MADFAs in parallel, such that their keyword sets form a partition of the original keyword set; and then merging these MADFAs and ensuring the result is a single MADFA for the original keyword set. That is, we generate MADFAs in two or more threads and merge them afterwards; as this merger in general may provide an ADFA yet not a MADFA, it must be minimized again to obtain the final single MADFA for the original keyword set.

The new algorithm family forks threads which are then used to create multiple MADFAs, one per thread; and it joins them again once the threads are done. The construction of multiple MADFAs does not require much synchronization. Every call to a method is independent of other calls if both method calls operate on different automata. In our case, we opt to ensure that every state, across the MADFAs created in parallel, gets a unique id, so the access to the id counter is synchronized. The unique id facilitates the merge of automata because every state in the merged automaton can be attached to one input automaton, and the merged automaton does not include states with the same ids. The only point where we need to synchronize threads, therefore, is the creation of states. (This synchronisation has no substantial impact on the total running time, as the observed time for the complete parallel MADFA construction in our experiments was around one twentieth of the time taken for the final merger and minimization.)

We have implemented, two instantiations of the general approach described above. The first version creates two MADFAs in parallel, while the second approach creates four MADFAs. Conceptually, the two algorithms work as follows:

1. Split keyword set into 2 or 4 parts, respectively.
2. Create a thread for each of these parts, and use each such thread to create a MADFA for a particular part, using one of the seven sequential MADFA construction algorithms.

3. Merge the 2 or 4 MADFAs obtained into a single ADFA, using the classical product construction for the union of multiple automata.
4. For minimization, run the *cleanup* method of Algorithm *Trie* on the resulting ADFA, yielding a final MADFA for the original keyword set. We use this particular *cleanup* because it can be used for arbitrary ADFAs (whereas the other MADFA construction algorithms' *cleanup* methods cannot).

In our implementation, steps 3 and 4 are performed for 2 MADFAs at a time, and this process is then repeated once in the case of 4 threads/MADFAs; but in general, the merger could be performed in one go for all the MADFAs. The resulting MADFA is a MADFA for the original keyword set. The details of the above construction can be found in Subsection 5.2, where our Java implementation of the approach is discussed.

The product automaton of step 3 is generated recursively from the start state, following the outgoing transitions. We traverse every state of both automata and generate the combined state. We show an example of this product construction from two MADFAs. We want to merge the automata in Figures 4 and 5. Automaton 1 accepts the words 'he' and 'she'. Automaton 2 accepts 'his' and 'this'. Note that both automata are MADFAs. The product of both automata is shown in Figure 6. We start with the product of both start states, i.e. 0,4. From there, we reach the product state 1,5 with a transition 'h', as 0 has such a transition to 1 and 4 to 5. With a transition 'e', we reach the final state 2 in automaton 1. Automaton 2 has no transition 'e' from state 5 but a transition 'i' to state 6. In the merged automaton we get states 2,null and null,6 or short 2 and 6. state 6 in automaton 2 has a transition to 7, so state 7 is copied to the product automaton. The same goes for state 3 and 1 that are reached from state 0 in automaton 1 and state 8 and 5 that are reached from state 4 in automaton 2. The resulting automaton is an ADFA because state 2 and 7 can be merged to generate the minimal MADFA.

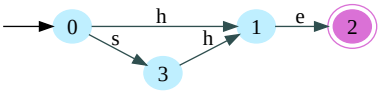


Figure 4. Automaton 1

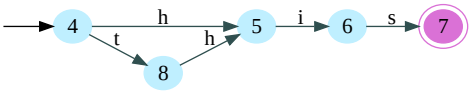


Figure 5. Automaton 2

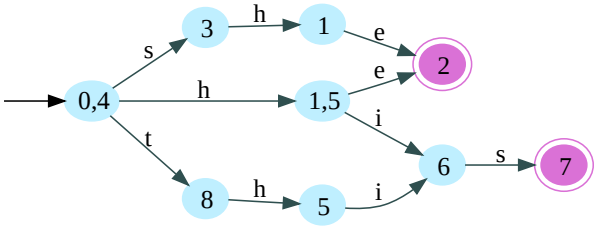


Figure 6. Merged automaton

The second variant of our algorithm generates four MADFAs in parallel. It also generates the product automaton during the merge step and minimize with a call to the *cleanup* method of Algorithm *Trie*. The difference is that this approach has two merge and minimization steps, re-using the merge of two MADFAs as mentioned above. First, two MADFAs are merged at a time and the intermediate ADFAs are minimized. The next step is to merge these two intermediate MADFAs again. The

resulting ADFA is minimized to the final MADFA. The first merge and minimization step is also done in parallel. We minimize the intermediate automata because the benchmark shows that this approach is faster than without a minimization step. This approach can be adapted to every number of threads that is a power of two; otherwise the merge scheduling has to be changed. Another possibility is to merge more than two automata at once, but this complicates the merge process.

5 Toolkit

Our MADFA construction toolkit[2], Many-MADFAct, implements a skeleton class which is shared by all the sequential MADFA construction algorithms; specific algorithm implementations directly or indirectly inherit from this base class and override the abstract methods *add_word* and *cleanup* as needed, as shown on the left of Figure 7. Some of the algorithms also need specific helper methods. Helper methods that are used for more than one algorithm are in a component *Util*. For the data representation we use an *automaton* class which includes *states* and *transitions*. The former have in- and out-transitions, and the latter are represented as triples of start state, label, end state, for efficient transition processing. The automaton contains states, of which one is the start state, and zero or more are final i.e. accepting states; and transitions that link states. To distinguish states, every state gets a unique id. The implementation was done in Java. After the data representation was chosen, the pseudo-code from the algorithms in the taxonomy was easily translated to Java.

The helper methods are combined in the component *Util*. This class is divided into three parts. Firstly, we use string manipulation methods, for example for returning the head or tail of a string, and for computing a left derivate or longest common prefix. The second part concerns the analysis of the automaton. It contains methods that creates state subsets of the automaton, like height levels or the state set corresponding to a path. The last part is the check for minimality. We compare states and decide whether they are equivalent or not.

5.1 Sequential Algorithm Implementation

As explained at the beginning of this section, the sequential MADFA construction algorithms are implemented as part of a hierarchy, derived using the TABASCO process mentioned in Section 3. The class diagram is shown on the left side of Figure 7. The root, *AlgorithmSkeleton*, is an abstract class that creates an empty automaton and calls method to generate a MADFA. It also declares the abstract methods *add_word* and *cleanup*. The general approach is to call the method *add_word* for every word and minimize the automaton with *cleanup* afterwards. This general approach is implemented in *createMadfa* using the template method design pattern. The specific algorithms inherit from this class and implement the abstract methods. They also import *Util*. If necessary, the algorithms declare private helper methods. Algorithm *Trie* only inherits from *AlgorithmSkeleton* directly. The other algorithms inherit from *Trie* and extend *add_word*. They call the super class's *add_word* and add specific operations at the end of the method. Method *cleanup* is always overridden, except in the case of Algorithm *General*. It uses the same *cleanup* as *Trie*. Algorithm *Incremental* is an exception: this algorithm inherits from Algorithm *General* because it has nearly

[2] https://github.com/TUBS-ISF/MADFAct

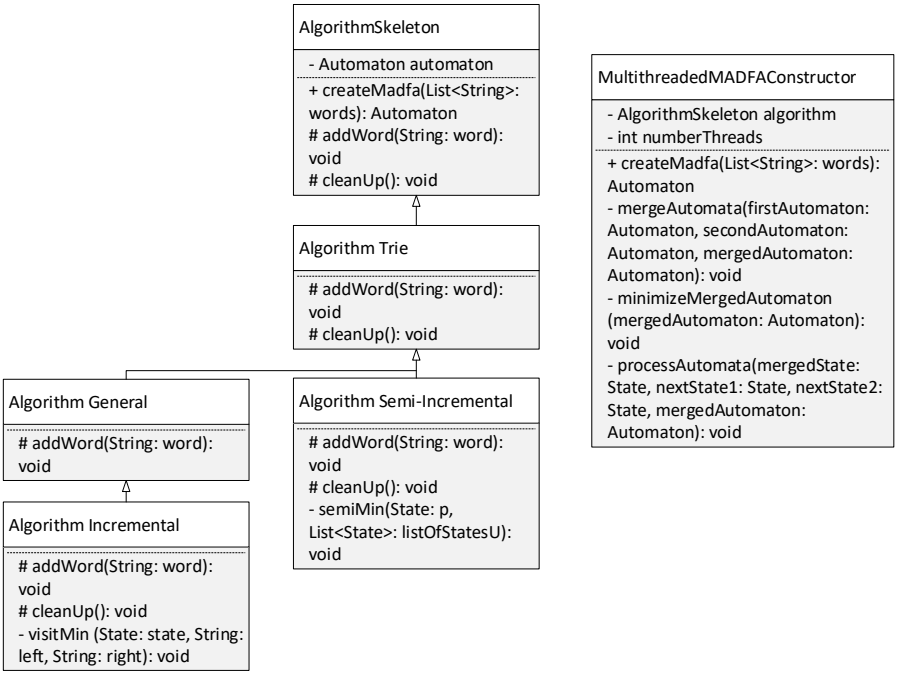


Figure 7. Class diagram of the toolkit

the same *add_word* method; i.e., *add_word* from *General* is called and extended. We implemented seven different sequential MADFA construction algorithms, of which this diagram shows three to illustrate the design without loosing clarity. The absent algorithms inherit from Algorithm *Trie* directly, just as Algorithm *Semi-Incremental* does.

5.2 Parallel Algorithm Implementation

The class *MultithreadedMAFDAConstructor* on the right of Figure 7 implements methods to create MADFAs in parallel and merge them afterwards. It contains a class variable that determines the construction algorithm used in each of the construction threads, e.g., Algorithm *Incremental*. Method *createMadfa* is the main method of this class. It creates MADFAs in parallel, and it calls the methods *mergeAutomata* and *minimizeMergedAutomaton* respectively to merge the resultant MADFAs into an ADFA, and to finally minimize this ADFA into a MADFA. Method *processAutomata* is a helper method of *mergeAutomata*. It creates the product automaton of multiple MADFAs by traversing the input automata recursively. The creation of MADFAs in parallel is done by forking and joining threads. Java is a multi-threaded programming language, so the implementation is straightforward; for every MADFA that should be constructed in parallel, we create a thread.

The procedure *createMadfa* is shown in Listing 1.1. Firstly, we divide the word list into the specific number of sub-lists (line 4). The next step is to create MADFA-threads and start them with a sub-list as input (line 7-12). They all execute the same algorithm and wait at the end. We implemented an algorithm to merge the intermediate MADFAs and minimize the result. *processAutomata*, the helper method for *mergeAutomata*, is shown in Listing 1.2. It traverses the input automata and merges them. It is a recursive method that gets a merged state and a state from each

input automaton as input. We check if the states are not null because it is possible that we process a merged state with one null state. If that is the case, we are at line 34 / 39, and we copy the outgoing transitions and successor states from this state. After that, we call the method for every successor again, i.e. the copied successor state is the new merged state. If both input states are not null, we search for outgoing transitions with the same labels (line 4-6). If we find a pair, we run the code in line 11-23. A new merged state is created if it does not already exist, and this is the merged state for the next call of this method. In the case of transitions that only appear in one input automaton, we do the same as we only have one input state. We copy the new transition and the successor state, cf. line 7-9 and line 26-31.

```
1  public Aut createMadfa(List<String> words) {
2    Aut mergedAut = new Aut();
3    List<Aut> intermediateAut = new ArrayList<>();
4    List<List<String>> subLists = chop(words, numberT);
5    //numberT is the number of threads
6
7    for (int i = 0; i < numberT; i++) {
8      List<String> subList = subLists.get(i);
9      MadfaThread thread = new MadfaThread(algorithm,
10       intermediateAut, subList);
11     thread.start();
12   }
13   Thread.join();
14
15   if (numberT == 2) {
16     //merge and minimize two automata
17   } else if (numberT == 4) {
18     //merge and minimize four automata
19   }
20   return mergedAut;
21 }
```

Listing 1.1. Code to start the parallel approach

6 Benchmarking

For benchmarking, we use the Java implementation of our toolkit and created MADFAs for different sets of input words. We use random English words[3] and subsequences from the ecoli genome[4]. We want to compare the runtimes of the seven algorithms. We also want to find out whether and how the lengths of the words impact performance. The results are presented below.

6.1 Setup

As input we decided for random English words to have a set with possibly many common prefixes and suffixes. We wanted to analyze how the algorithms behave if they can merge states during *cleanup*. For a totally different application setting, we also used the ecoli genome as input. Here, we cut substrings from the genome and use these as input. Most of the time such substrings have no common prefixes or suffixes because the probability to get a sequence of equal characters is low—especially for the natural language case. For example, the probability that two words share the same four characters as a prefix is lower than one percent. Therefore, the generated MADFAs consist of parallel state paths that do not have much in common.

[3] http://www-01.sil.org/linguistics/wordlists/english/
[4] http://www.dmi.unict.it/ faro/smart/download.php

```
1  private static void processAutomata(St mergedSt,
2    St nextSt1, St nextSt2) {
3    if (nextSt1 != null && nextSt2 != null) {
4      for (Tran trans1 : nextSt1.getOutgoingTran()) {
5        Tran trans2 = getEqualTran(nextSt2,
6                trans1.getLabel());
7        if (trans2 == null) {
8          St newMergedSt = copy(mergedSt, trans1);
9          processAut(newMergedSt, trans1.EndSt(), null);
10       } else {
11         String id = trans1.getEndSt().getId() + ";" +
12                   trans2.EndSt().getId();
13         St newMergedSt = getEqualSt(id, mergedAut);
14         if (newMergedSt == null) {
15           newMergedSt = new St(id);
16         }
17         if (mergedSt.getEqualTran(newMergedSt,
18                   trans1.getLabel()) == null) {
19           Tran newTran = new Tran(mergedSt,
20                       newMergedSt, trans1.getLabel());
21         }
22         processAut(newMergedSt, trans1.EndSt(),
23                   trans2.EndSt());
24       }
25     }
26     for (Tran trans2 : nextSt2.getOutgoingTran()) {
27       Tran trans1 = getEqualTran(nextSt1,
28               trans2.getLabel());
29       if (trans1 == null) {
30         St newMergedSt = copy(mergedSt, trans2);
31         processAut(newMergedSt, null, trans2.EndSt());
32       }
33     }
34   } else if (nextSt1 != null) {
35     for (Tran trans : nextSt1.getOutgoingTran()) {
36       St newMergedSt = copy(mergedSt, trans);
37       processAut(newMergedSt, trans.EndSt(), null);
38     }
39   } else if (nextSt2 != null) {
40     for (Tran trans : nextSt2.getOutgoingTran()) {
41       St newMergedSt = copy(mergedSt, trans);
42       processAut(newMergedSt, null, trans.EndSt());
43     }
44   }
45 }
```

Listing 1.2. Code to merge two automata

The setup for our benchmark is as follows. We select random sets of words and run every algorithm five times with each set. To deal with for example caching problems, we take the fastest run among these five as result. The sets form a sequence of increasing size and for every set size we generate 30 different sets, i.e. we add random words until the size of the set is reached. For example we build sets from size one to 2^{16} in the case of random English words. We always double the number of words from one set to the next. In the case of ecoli, we do two different runs. First, we construct sets which consist of strings of the same length. We vary the set size from 1 to 2^{10}, doubling the number in each iteration. For every set size, we insert strings of the same length, ranging from one to 2^6 (64) and again doubling in each iteration. The second benchmark run of ecoli is with substrings of varying length, called varying-length ecoli. We also construct sets from 1 to 2^{10}, but this time, we insert substrings of random length between 1 and 2^6.

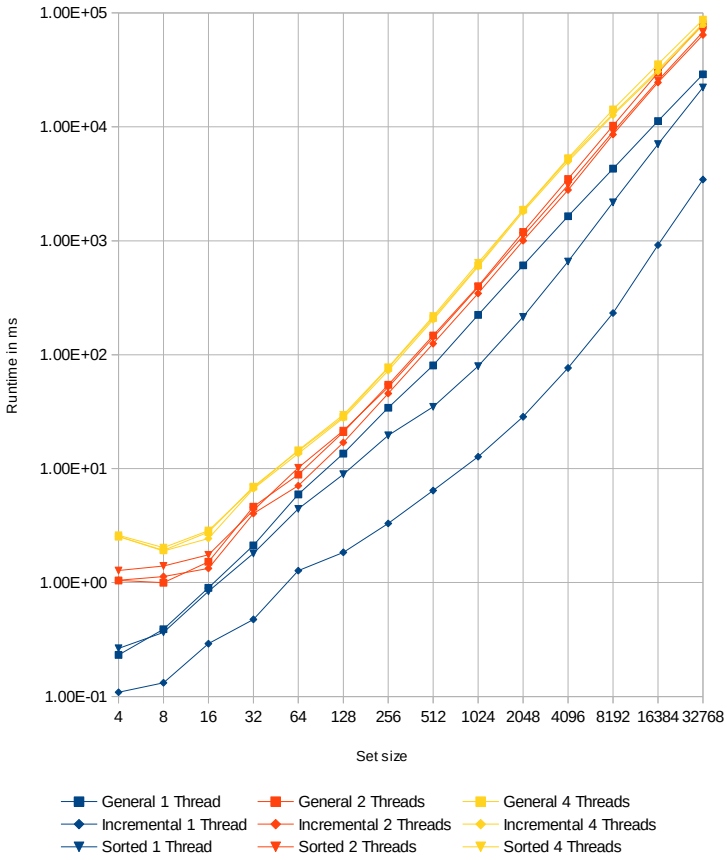


Figure 8. Benchmark result of fast algorithms with English words. The x-axis of the graph shows the set size of input words, the y-axis the runtime in ms.

6.2 Results

The benchmark results diverge between the seven algorithms. For example, with English words as input, Algorithm *Incremental* is the fastest. It needs ca. 15 seconds for 2^{16} words. The next faster algorithms are in this order: Sorted, Trie and General. Trie needs for example ca. 65 seconds for the same number of words. The other three algorithms, *Depth-Layered*, *Reverse* and *Semi-Incremental*, are much slower. They need up to two hours for this word set. We also tested the same sets with the new parallel implementation, using two and four threads. The fast algorithms *Incremental*, *Sorted*, *Trie* and *General* are not getting faster, they are even slower, cf. Figure 8. The scale for both axes is logarithmic. We start at set size four so that, in the case of four threads, every thread gets at least one input word. The graphs show that the runtime for each algorithm increases exponentially. Algorithm *Trie* is not shown in the figure because it behaves like Algorithm *General* if the MADFA is built from an empty automaton. The difference between Algorithm *Trie* and Algorithm *General* is that *General* looks for confluence states, i.e. states which need to be cloned before adding a new transition, and clones them [31]. If the automaton is built from scratch, it is constructed as a trie and no confluence states occur, so both algorithms execute identically. The runtimes for every algorithm for one thread are in every case shorter than the runtime for two or four threads.

The slow algorithms on the other hand get faster. For example we present in Figure 9 the algorithms *Depth-Layered*, *Semi-Incremental* and *Reverse*. For small

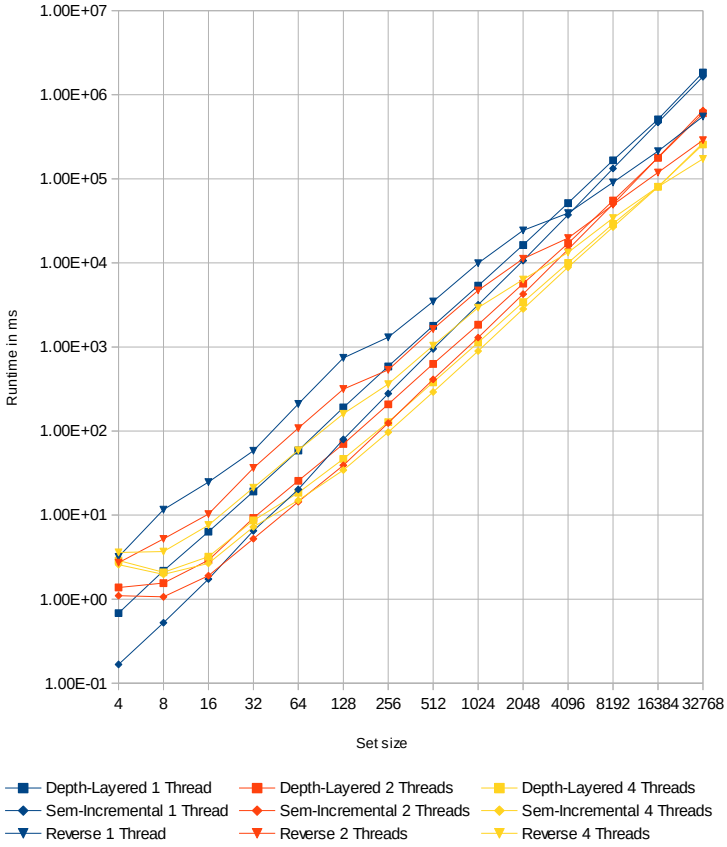


Figure 9. Benchmark result of slow algorithms with English words. The x-axis of the graph shows the set size of input words, the y-axis the runtime in ms.

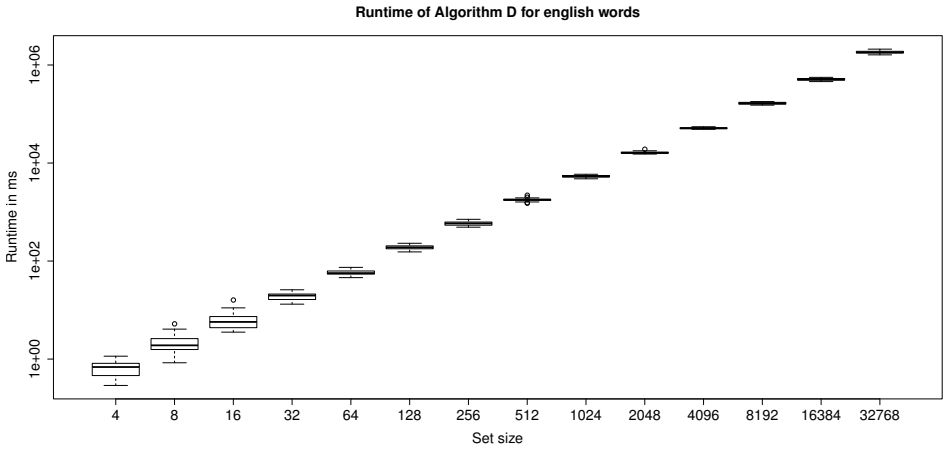


Figure 10. Benchmark result of algorithm Depth-Layered (one thread). The x-axis of the graph shows the set size of input words, the y-axis the runtime in ms.

sets, the execution of *Depth-Layered* and *Semi-Incremental* is the fastest, but the bigger the set is, the better the parallel implementation performs. For big sets, i.e., with 32768 words, the 4 thread implementation is in every case the fastest, followed by the 2 thread one. Here, we can save time by running the algorithm in parallel. A general observation for the three algorithms is, the more threads the faster the

runtime. However in the case of small sets, the runtime is slower because the parallel approach has some overhead for creating parallel automata and merging them. The approach does not pay off in such cases, due to the extra merge and minimization steps needed.

Figure 10 only shows the benchmark results for Algorithm *Depth-Layered*. Here, we do not compute the average of the 30 runs for every set size. The graph shows boxplots that include the 30 different runs. As we can see, the runtimes are similar and there are few spikes. The scale of the runtime is logarithmic, causing the boxplots to be very small. The other benchmark results are quite similar. We infer from the few spikes and the small box sizes that the use of mean values is ok, as the spread of values is limited.

The benchmarking using ecoli strings does not uncover new insights. The runtime increases exponentially for every algorithm and the ordering wrt. performance is the same, i.e. Algorithm *Incremental* is the fastest. If we want to compare the benchmark of ecoli with the benchmark of English words, we should not compare sets with the same number of words because the ecoli strings can be much longer. We decided to compare sets with the same summed word length. For example, we compared 256 equal length ecoli strings with fixed string length 64 with an English word set with 2048 words that has ca. the same summed word length. The running times of all seven algorithms are longer for ecoli than for English words. We get the same result, that the runtime is longer for longer string lengths, if we compare two different runs of ecoli with fixed string length. We take for example the results of set size 128 and length 64 and compare it with the results of set size 1024 and length 8. Both have 8192 characters in total and the algorithms are slower in case of the longer strings, i.e., the toolkit performs better for large sets with short words than for small sets with long words.

7 Conclusion and Future Work

In this project, we successfully implemented the algorithms presented by Watson [31]. First, we created a taxonomy graph by identifying the commonalities and differences between the algorithms. The next step was to create a toolkit based on this information, using the TABASCO process to do so. We also implemented a new algorithm family exploiting parallelism. The two algorithm variants from this family that we discussed and implemented create MADFAs in two or four threads and merge and minimize the resulting MADFAs into a single MADFA in the end. We benchmarked the toolkit using English words and ecoli substrings as input. The results show that the parallel approach improves the runtime of the slower algorithms.

For future work an implementation in C++ is planned to compare the implementations with respect to their runtime and their storage space consumption. We also want to improve the current parallel implementation. The current merge process creates an ADFA from two MADFAs. We minimize the ADFA afterwards to create the final MADFA. It should be possible to create a MADFA directly from two or more MADFAs, by adapting the merge process to minimize the product automaton during construction, possibly by reusing and generalizing ideas from Algorithm *Incremental*'s *add_word* method.

References

1. M. Broy: *Program construction by transformations: a family tree of sorting programs*, in Computer Program Synthesis Methodologies, A. W. Biermann and G. Guiho, eds., Reidel, Dordrecht, 1983, pp. 1–49.
2. J. A. Brzozowski: *Canonical regular expressions and minimal state graphs for definite events.* Mathematical theory of Automata, 12(6) 1962, pp. 529–561.
3. J. Bubenzer: *Construction of minimal ADFAs*, diplomarbeit, Universität Potsdam, Germany, 2011.
4. R. C. Carrasco and M. L. Forcada: *Incremental construction and maintenance of minimal finite-state automata.* Computational Linguistics, 28(2) June 2002, pp. 207–216.
5. M. Ciura and S. Deorowicz: *Experimental study of finite automata storing static lexicons*, tech. rep., Silesian Technical University, Poland, Nov. 1999.
6. L. Cleophas: *Tree Algorithms: Two Taxonomies and a Toolkit*, PhD thesis, Technische Universiteit Eindhoven, 2008.
7. L. Cleophas, B. W. Watson, D. G. Kourie, A. Boake, and S. Obiedkov: *TABASCO: using concept-based taxonomies in domain engineering.* South African Computer Journal, 2006, pp. 30–40.
8. J. Daciuk: *Incremental Construction of Finite-State Automata and Transducers, and their Use in the Natural Language Processing*, PhD thesis, Technical University of Gdańsk, Poland, 1998.
9. J. Daciuk: *Optimization of Automata*, Gdańsk University of Technology Publishing House, 2014.
10. J. Daciuk, S. Mihov, B. W. Watson, and R. E. Watson: *Incremental construction of minimal acyclic finite state automata.* Computational Linguistics, 26(1) Apr. 2000, pp. 3–16.
11. J. Daciuk, B. W. Watson, and R. E. Watson: *Incremental construction of minimal acyclic finite state automata and transducers*, in Proceedings of the International Workshop on Finite State Methods in Natural Language Processing, L. Karttunen and K. Oflazer, eds., Ankara, Turkey, June 1998, pp. 48–56.
12. J. Darlington: *A synthesis of several sorting algorithms.* Acta Informatica, 11(1) 1978, pp. 1–30.
13. J. Graña, F. M. Barcala, and M. A. Alonso: *Compilation methods of minimal acyclic finite-state automata for large dictionaries*, in Proceedings of the Sixth Conference on Implementations and Applications of Automata, B. W. Watson and D. Wood, eds., vol. 2494, Pretoria, South Africa, July 2002, Springer Verlag, pp. 116–129.
14. G. J. Holzmann and A. Puri: *A minimized automaton representation of reachable states.* Software Tools for Technology Transfer, 3 (1998)(1) 1998.
15. H. Jonkers: *Abstraction, specification and implementation techniques: with an application to garbage collection*, PhD thesis, Technische Hogeschool Eindhoven, 1982.
16. A. Marcelis: *On the classification of attribute evaluation algorithms.* Science of Computer Programming, 14(1) 1990, pp. 1–24.
17. S. Mihov: *Direct building of minimal automaton for given list.* Annuaire de l'Université de Sofia St. Kl. Ohridski, 91(1) 1998, pp. 212–225.
18. S. Mihov: *Direct building of minimal automaton for given list*, tech. rep., Bulgarian Academy of Science, 1999.
19. K.-H. Park, J.-I. Aoe, K. Morimoto, and M. Shishibori: *An algorithm for dynamic processing of DAWGs.* International Journal of Computational Mathematics, 54 1994, pp. 155–173.
20. V. Pieterse: *Topic Maps for Specifying Algorithm Taxonomies: A Case Study using Transitive Closure Algorithms*, PhD thesis, University of Pretoria, 2017.
21. D. Revuz: *Dictionnaires et lexiques: méthodes et algorithmes.*, PhD thesis, Institut Blaise Pascal, LITP 91.44, Paris, France, 1991.
22. D. Revuz: *Minimisation of acyclic deterministic automata in linear time.* Theoretical Computer Science, 92(1) 1992, pp. 181–189.
23. D. Revuz: *Dynamic acyclic minimal automaton*, in Proceedings of the Fifth Conference on Implementations and Applications of Automata, S. Yu and A. Păun, eds., vol. 2088, London, Canada, July 2000, Springer-Verlag, pp. 226–232.

24. K. Sgarbas, N. Fakotakis, and G. Kokkinakis: *Two algorithms for incremental construction of directed acyclic word graphs.* International Journal of Artificial Intelligence Tools, 4 1995, pp. 369–381.
25. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Technische Universiteit Eindhoven, 1995.
26. B. W. Watson: *A taxonomy of algorithms for constructing minimal acyclic deterministic finite automata.* South African Computer Journal, (27) 2001, pp. 12–17.
27. B. W. Watson: *Directly constructing minimal DFAs: Combining two algorithms by Brzozowski.* South African Computer Journal, 29 Dec. 2002, pp. 17–23.
28. B. W. Watson: *A fast and simple algorithm for constructing minimal acyclic deterministic finite automata.* Journal of Universal Computer Science, 8(2) 2002, pp. 363–367.
29. B. W. Watson: *A new algorithm for the construction of minimal acyclic DFAs.* Science of Computer Programming, 48(2–3) 2003, pp. 81–97.
30. B. W. Watson: *A new recursive incremental algorithm for building minimal acyclic deterministic finite automata*, in Grammars and Automata for String Processing: From Mathematics and Computer Science to Biology, and Back, C. Martin-Vide and V. Mitrana, eds., Taylor and Francis, 2003, pp. 189–202.
31. B. W. Watson: *Constructing Minimal Acyclic Deterministic Finite Automata*, PhD thesis, University of Pretoria, 2010.

A Family of Exact Pattern Matching Algorithms with Multiple Adjacent Search Windows

Igor O. Zavadskyi

Taras Shevchenko National University of Kyiv
Kyiv, Ukraine
2d Glushkova ave.
ihorza@gmail.com

Abstract. A new family of comparison-based exact pattern matching algorithms is presented. They utilize the multi-dimensional arrays in order to process more than one adjacent search window in each iteration of the search loop. This approach leads to a lower average computing time by the cost of space. However, the excessive space consumption can be avoided due to a special technique of replacing a multi-dimensional array with a series of one-dimensional arrays of pointers. The algorithms of this family perform well for short or middle-size patterns, when the shift of a search window by several lengths at once is quite probable. Our algorithms outperform all other known algorithms for some values of pattern length on English text, genomic sequence and a random text over an alphabet of size 8 or 32.

1 Introduction

Pattern matching is one of the most fundamental techniques in computer science. The most common pattern matching problem is formulated as finding all the exact occurrences of a given substring in a larger body of text. Through the entire presentation we use the following notation:

- $T[0\,..\,n-1]$ - input text
- $P[0\,..\,m-1]$ - pattern to be searched
- n - length of the input text
- m - length of the pattern
- Σ - alphabet of the input text and the pattern
- $|\Sigma|$ - size of the alphabet
- $|\Sigma_P|$ - number of different symbols in the pattern

It is worthwhile to compare the pattern matching algorithm efficiency on the $(|\Sigma|, m)$-plane. Our research concerns mostly its middle left area, where $m \leq 64$ and $4 \leq |\Sigma| \leq 32$. In this area the modifications of the Boyer-Moore algorithm (BM) [2], e.g. the Boyer-Moore-Horspool algorithm (BMH) [12], Sunday's "Quick Search" (QS) [17] or Tuned Boye-Moore (TBM) [14] were considered the best for a long time. However, a number of more efficient exact pattern matching algorithms were invented after 2000. According to experiment results on a random text presented in [9] the most successful algorithms are TVSBS [18] for $m = 2$, $8 \leq |\Sigma| \leq 32$, EBOM [6] for $4 \leq m \leq 16$, $8 \leq |\Sigma| \leq 32$ or $m = 4$, $\Sigma = 4$ and also HASHq [15], SBNDMq [4] and FSBNDM [6] in the right subarea of $m \leq 64$, $4 \leq \Sigma \leq 32$ rectangle. The Shift-And algorithm [20] proposed in 1992 remains optimal for $m = 2$, $\Sigma = 4$. The mentioned algorithms cover all three known approaches to pattern-matching: TVSBS and HASHq are comparison-based; EBOM is automata based, while Shift-And, SBNDMq and FSBNDM algorithms use the bit-parallel operations. We develop

Igor O. Zavadskyi: A Family of Exact Pattern Matching Algorithms with Multiple Adjacent Search Windows, pp. 143–157.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0

a new comparison-based algorithm family. Almost all comparison-based algorithms, including our new ones, exploit the idea of bad-character shift, which originates from the BM search. It is to shift the search window to align the character or characters around its end to their most right occurrence in the pattern. The BMH algorithm is based on this idea only. We develop a generalization of the BMH algorithm, which allows to perform several bad-character shifts in each iteration of the search loop.

Let us discuss the search loop of the BMH (Algorithm 1). The bad character shift is performed in the line 8 and its length is equal to $D[T[pos + m - 1]]$, where pos is the starting position of the search window and D is the shift array defined by

$D(c) = \min(\{1 \leq k < m | P[m - 1 - k] = c\} \cup \{m\})$.

If the ratio $|\Sigma_P|/|\Sigma|$ is small enough, the character $T[pos + m - 1]$ most likely does not occur in the pattern and the length of the shift is maximum, i.e. it stands m. These maximum length shifts are the main factor responsible for the efficiency of BMH in the left up area of the $(|\Sigma|, m)$-plane. And if the ratio $|\Sigma_P|/|\Sigma|$ is particularly small, one can assume that probably not only the character $T[pos + m - 1]$ does not belong to the pattern, but the characters $T[pos + 2m - 1]$, $T[pos + 3m - 1]$ etc. as well. This means that the search window can be shifted by several lengths at once, or, in other words, several adjacent search windows can be processed in the same iteration of the search loop. This is the main idea of the multiple adjacent window search algorithms (MAW), as well as of the Tuned Boyer-Moore algorithm.

Of course, at least k characters of the input text must be read and processed in each substring of the length km in order not to miss the possible pattern match. Thus, at least k readings of text characters should be done for each substring of the length km – just the same number as in k iterations of the single-window algorithm like BMH or QS. However, we can reduce the number of other operations. For this we use the k-dimensional array, unlike the TBM algorithm, where the search loop is unrolled. Such array occupies rather more memory than the shift table in a single-window search algorithm and its filling takes more preprocessing time. Nevertheless, these space overheads are not that big comparing to memory size of modern computers, while time overheads are more than covered in the main search loop.

Algorithm 1: The search phase of the Boyer-Moore-Horspool algorithm

```
1 pos ← 0 ;
2 while pos ≤ n − m do
3     j ← 0;
4     while T[pos + j] = P[j]  AND j < m do
5         j ← j + 1;
6     if j = m then
7         output pos;
8     pos ← pos + D[T[pos + m − 1]];
```

The idea of using two or more search windows is not new. Apart from the mentioned Tuned Boyer-Moore algorithm, it was implemented in Two- and Four-Sliding-Windows algorithms [13], variants of Backward-SNR-DAWG-Matching [7] for multiple windows [5]. Also, this idea was applied to different algorithms in [8]. However, search windows in these algorithms are not adjacent. Thus, they are well suited for parallel processing or multi-pattern search, but do not take the advantage of multi-dimensional search array. Though the idea of two-dimensional search array was also implemented in a number of algorithms, for instance, in the Berry-Ravindran algorithm [1], TVSBS and EBOM, in most of them it was proposed to use the consequent

characters of a text as indices. This significantly increases the probability of the maximum length shift if it is low for a single-character check but otherwise leads to superfluous density of the checks. In other words, if even single-character check causes the maximum shift with high probability, likely, there is no need to check two adjacent characters to shift the search window by m or $m+1$ positions. In this case it may be better to perform some special fast check of the characters $T[pos]$ and $T[pos+m]$ in the same iteration, which could shift the search window by $2m$ positions. These considerations lead to MAW2 algorithm. In general, we denote by MAWq the MAW algorithm based on the processing of q adjacent windows.

The attempt to speed up a search using two adjacent search windows was made in [19] (QLQS algorithm). Authors use two one-dimensional search tables ("forward" and "backward"). This allows to increase the average shift length against the single-character algorithms like QS, but, unlike the MAW2, does not guarantee that we make the maximum possible safe shift based on "couple of characters" heuristic. Our experiments show that the QLQS performs slower than the MAW2 (tables 4-6).

The two-dimensional search array was combined with the adjacent search windows in [3]. The so-called jumping-occurrence heuristic allows to perform the shift by checking two characters in an adjustable distance. When this distance is maximum, i.e. $m+1$, this solution called JOM becomes quite similar to MAW2 and even provide longer shifts in average due to forward character checks. However, this "forward" logic requires the pattern occurrence check in each iteration of JOM, while in the MAW2 this check is performed only under certain condition, which is satisfied infrequently. Besides that, accessing the array element in the MAW2 requires fewer additions. All this makes the MAW2 algorithm faster than JOM, as experiments show.

Nevertheless, the assumption that not only the character $T[pos]$ does not belong to the pattern, but the character $T[pos+m]$ as well, is quite strong. As experiments show, the MAW2 algorithm based on this assumption outperforms all the other only when the pattern is very short (3–6) and alphabet size is around 32 (table 6). If the pattern is longer or alphabet smaller, the adjacent characters check is efficient. In this case we could check the character bigrams in the right of the adjacent windows. For example, using the four-dimensional search table to check the characters $T[pos]$, $T[pos+1]$, $T[pos+m]$ and $T[pos+m+1]$, we obtain the MAW22 algorithm (2 adjacent search windows and 2 adjacent characters to check in each).

The similar combination of characters in shift heuristic is checked in the SBNDM algorithm with the "greedy" skip-loop (GSB) [16]. However, since it is based on one-dimensional search tables only, either the maximum or the average shift is shorter than in MAW22. Also, the operational complexity of the "skip" iteration of Greedy FSBNDM algorithm is higher (table 2). As a result, the GSB algorithm is essentially slower than the MAW22 on short alphabets ($|\Sigma| = 4$, table 3, or $|\Sigma| = 8$, table 5), where the MAW22 algorithm appears to be the most efficient.

However, for alphabets containing 25–30 or more characters, the 4-dimensional search table becomes too large and its processing slows down due to caching or other memory access issues. For this reason we investigate how to utilize the pointers in order to reduce the size of the search tables significantly, while increase the access time only a little. The resultant algorithms are called "MAW with pointers" (MAWP).

This paper is organized as follows. In section 2 we optimize the BMH search loop to construct an algorithm that checks single characters in two adjacent windows, it is MAW2. We discuss the types of search window shifts in MAW2 and compare the BMH and MAW2 complexity at operational level. In section 3 we generalize the

MAW2 algorithm with checking the bigrams of characters in the adjacent search windows. This is the MAW22 algorithm. In section 4 we discuss how to reduce the size of multi-dimensional search tables. In section 5 we describe the generalization of the MAW2 and MAW22 to the case of more than 2 adjacent search windows. Thus we obtain the MAWq and MAWq2 algorithm families. In section 6 the preprocessing stage of the MAWq and MAWq2 algorithms is discussed. In section 7 we present the experimental results and make the final conclusions in section 8.

2 The MAW2 algorithm with two adjacent search windows

In this section we discuss how to process two adjacent search windows of the length m, which could be considered as one window of double length $2m$. We try to reduce the total number of computing operations required to process the substring of the length $2m$. Let us examine the search loop of BMH (Algorithm 1). Two reads from the shift table D in the line 8 in two iterations can be replaced by one read from the two-dimensional shift table $M2_{|\Sigma|\times|\Sigma|}$ defined as follows: $M2[i][j]$ is the leftmost possible position of the first character of a pattern under the assumption that $T[m-1] = i$ and $T[2m-1] = j$. All shifts defined by the table $M2$ can be divided into 4 types shown in Figure 1.

- (a) Neither i nor j belongs to the pattern P. Then P can be safely shifted by $2m$ positions forward.
- (b) Character i doesn't belong to the pattern P, but j does. In this case P can be safely shifted by more than $m-1$ symbols but less than $2m$. Namely, the rightmost occurrence of j in P should be aligned with $T[2m-1]$.
- (c) Character i belongs to P and $P[m-1] \neq i$. Then P can be safely shifted forward by less than m symbols. Namely, the rightmost occurrence of i in P should be aligned with $T[m-1]$.
- (d) $P[m-1] = i$. Then the pattern can be matched at the current position. One should check if $T[0] \ldots T[m-2]$ coincides with the pattern before proceed forward.

The search loop of the MAW2 is shown in Algorithm 2. The text is assumed to be appended by the "stop" pattern. Note that the condition in line 4 is met only in the case (d), otherwise only lines 2, 3, 4 and 12 are executed.

Algorithm 2: The search phase of the Two Adjacent Windows algorithm MAW2

```
1  pos ← m − 1 ;
2  while pos < n do
3      r ← M2[T[pos]][T[pos + m]] ;
4      if r = 0 then
5          j ← 0 ;
6          while j < m − 1 AND T[pos − (m − 1) + j] = P[j] do
7              j ← j + 1 ;
8          if j = m − 1 then
9              output pos − (m − 1) ;
10         pos ← pos + D[T[pos]] ;
11     else
12         pos ← pos + r ;
```

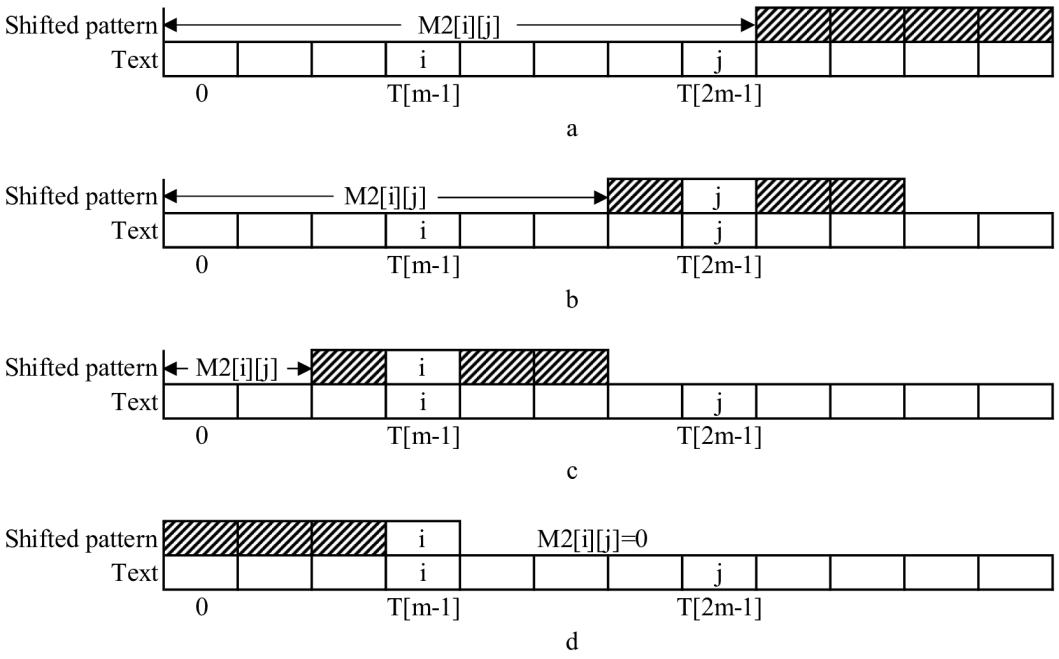


Figure 1. Pattern shifts in the MAW2 algorithm

Let us calculate the number of operations in the BMH and MAW2 algorithms required to shift the search window by $2m$ characters forward (the case of the maximum possible shift). This is the most probable case when the pattern length is small compared to alphabet size. In this case only lines 2, 3, 4, 6 and 8 in two iterations of the BMH algorithm and only lines 2, 3, 4 and 12 in some iteration of the MAW2 algorithm are executed. Note that getting an element of a one-dimensional array like $D[x]$ is equivalent to $*(D + x)$ in C notation, which requires one addition and two readings from memory, while getting an element of a two-dimensional $|\Sigma| \times |\Sigma|$ array like $M2[x][y]$ is equivalent to $*(M2 + |\Sigma| * x + y)$, which requires two additions, one multiplication and three readings from memory ($|\Sigma|$ is a constant).

The calculations are shown in Table 1. The number of operations in the MAW2 algorithm is more than twice less compared to BMH. The subtractions in expressions $m - 1$, $n - m$, $z - 1$ are not counted, since these values can be calculated in the preprocessing stage. Also, the comparison $j < m$ is not counted in the AND conjunction in BMH, since it is not actual in the case of the maximum shift. Let us note that multiplication is no longer time consuming on modern computers and exceeds the time of other operation by 20-30% at most.

One can observe that one iteration of the MAW2 search loop requires fewer operations even than one iteration of the BMH search loop in the case when the condition $r = 0$ is not met in the line 4 of Algorithm 2 (Figure 1a–c). Therefore, in the case shown in Figure 1c the MAW2 algorithm search loop still executes faster than the BMH search loop, while the equality $M2[i][j] = D[i]$ holds, i.e. the shift length in the MAW2 algorithm is just the same as in the BMH.

Of course, the advantage of the MAW2 search loop over the BMH search loop in the case (b) is lower than in the case (a) and in the case (c) is lower than in the case (b). While the ratio $|\Sigma_P|/|\Sigma|$ increases, the balance between the cases (a), (b) and (c) moves to (b) and (c) and then to (c) only. If $|\Sigma_P|/|\Sigma|$ is close to 1, the case (c)

Table 1. The operational complexity of the BMH and MAW2 algorithms

Operation	BMH	MAW2
Comparisons	6 = (lines 2, 4 and 6)×2	2 = lines 2 and 4
Assignments	4 = (lines 3 and 8)×2	2 = lines 3 and 12
Memory reads	28 = (2 in line 2; 1 in line 3; 5 in line 4; 1 in line 6; 5 in line 8)×2	10 = 2 in line 2; 6 in line 3; 2 in line 12
Additions	14 = (3 in line 4; 4 in line 8)×2	6 = 5 in line 3; 1 in line 12
Multiplications	–	1 in line 3
Total	52	21

occurs almost always and outperformance of the MAW2 search loop over the BMH search loop is small. The case shown in Figure 1d occurs with the probability $1/|\Sigma|$ regardless of $|\Sigma_P|$ value, for random text. This is when the internal loop of the MAW2 in the lines 6 and 7 of Algorithm 2 is executed and has the same number of iterations as the internal loop of BMH. However, each iteration of the MAW2 internal loop requires one operation more than that one of the BMH.

Thus, the search loop of the MAW2 algorithm is essentially faster than the search loop of the BMH algorithm when the following conditions are met: (1) the alphabet is large enough to make the case (d) not frequent; (2) the ratio $|\Sigma_P|/|\Sigma|$ is small enough to make the case (c) not frequent. In fact, any alphabet of size 8 and bigger could be considered as "large enough" to make the case (d) not frequent. The violation of condition (2) forces the MAW2 algorithm search loop to run only a bit faster than the BMH search loop. However, for the wide range of pattern length / alphabet size combinations the MAW2 outperforms the BMH essentially.

3 The bigram extension

As mentioned above, the MAW2 algorithm exploits the very classical approach consisting in checking single characters, i.e. characters that are far apart in the text. Although this idea could give some advantage when the ratio $|\Sigma_P|/|\Sigma|$ is small, a number of algorithms invented since 1990s show that checking two or more adjacent characters (q-grams) is more efficient in general case. Namely, such checks are performed in the EBOM, FSBNDM, Hashq and other algorithms, which are considered as the fastest ones in some areas of $(|\Sigma|, m)$-plane. Let us note that the MAW technique can be applied also to q-gram checks. In this case the fundamental assumption is that probably not only the pair of characters $(T[pos], T[pos+1])$ does not belong to the pattern, but the pair $(T[pos+m], T[pos+m+1])$ as well. Of course, this assumption is realistic for much wider range of values $(|\Sigma|, m)$ than that one for the pair of single characters. And it leads to the MAW22 algorithm (2 adjacent search

windows with 2 bigram checks in each). Its search loop is shown in Algorithm 3. It is assumed that the text is appended by the "stop" pattern.

Algorithm 3: The search phase of the MAW22 algorithm

```
1  pos ← m − 2;
2  while true do
3      r ← M22[T[pos]][T[pos + 1]][T[pos + m]][T[pos + m + 1]] ;
4      if r = 0 then
5          j ← 0;
6          while T[pos − (m − 2) + j] = P[j] AND j < m − 2 do
7              j ← j + 1;
8          if j = m − 2 then
9              output pos − (m − 2);
10             if pos ≥ n then
11                 break;
12             pos ← pos + D[T[pos]] ;
13     else
14         pos ← pos + r;
```

In Algorithm 3 the single bad character shift table is denoted by D, just as in the MAW2 or BHM algorithm, while the bigram shift table $M22$ is organized as follows. $M22[i][j][k][t]$ is the leftmost possible position of the first character of the pattern under the assumption that $T[m-2] = i$, $T[m-1] = j$, $T[2m-2] = k$, $T[2m-1] = t$.

Evidently, the MAW22 algorithm could be successful thanks to shifts that are essentially longer than m. Otherwise a search window shift is too short to compensate such expensive operation as accessing the 4-dimensional array element. As experiments show (Tables 3 and 5), the probability of "good" shifts is high enough to make the MAW22 algorithm the fastest one for some pattern lengths when $4 \leq |\Sigma| \leq 8$. Accordingly to [9], in this area of $(|\Sigma|, m)$-plane the best results belonged to the EBOM (foremost), Hashq and variations of the SBNDM algorithm. Among the SBNDM family the GSB algorithm is of the most interest, since it exploits the same idea of two bigram checks in the skip loop, as the MAW22 (Algorithm 5). The EBOM algorithm also contains the special fast skip loop shown in Algorithm 4 (we present the fast practical implementation of the EBOM taken from [10]). In two iterations of the EBOM skip loop or one iteration of the GSB skip loop the search window can be shifted by $2m - 2$ characters at most.

The number of basic operations in two iterations of the EBOM and one iteration of the MAW22 and GSB in the case of the maximum shift is shown in Table 2. Only the lines 2, 3, 4 and 14 of Algorithm 3 are executed. As is seen, the MAW22 algorithm performs only one operation less than EBOM and 4 operations less than GSB. However, the maximum shift in the MAW22 is 2 characters longer than the maximum shift in the GSB or double maximum shift in the EBOM. This is quite noticeable difference for short patterns. This maximum shift by $2m$ positions is achieved in the MAW22 when (1) neither the pair (i, j) nor the pair (k, t) belongs to the pattern, (2) t is not the first character of the pattern and (3) j and k are not the last and the

Table 2. The operational complexity of MAW22, GSB and EBOM algorithms

Operation	EBOM	GSB	MAW22
Comparisons	2=(1 in line 1)×2	2 (line 1)	1 in line 4
Assignments	2=(1 in line 2)×2	2 (lines 1 and 2)	1 in line 3
Memory reads	14 = (5 in line 1; 2 in line 2)×2	16 = 14 in line 1; 2 in line 2	14 = 11 in line 3; 1 in line 4; 2 in line 14
Additions	12 = (5 in line 1; 1 in line 2)×2	12 = 11 in line 1; 1 in line 2	12 = 11 in line 3; 1 in line 14
Multiplications	2=(1 in line 1)×2	–	3 in line 3
Bitwise shift	–	2 in line 1	–
Logical AND	–	1 in line 1	–
Total	32	35	31

first characters of the pattern respectively. If only conditions (1) and (3) are met, the length of the shift is equal to $2m - 1$.

Algorithm 4: The skip search loop of the EBOM algorithm

```
1 while FT[T[pos]][T[pos − 1]] = θ do
2     pos ← pos + m − 1;
```

Algorithm 5: The "greedy" skip loop of the GSB algorithm

```
1 while (D ← ((B[T[i + 1]] << 1)&B[T[i]])) = 0 AND
    ((B[T[i + m]] << 1)&B[T[i + m − 1]]) = 0 do
2     i ← i + 2m − 2;
```

In the case of a non-maximum shift the MAW22 algorithm most often gives the shift length at once, i.e. in the line 3 of Algorithm 3, which requires just the same number of operations as in the case of a maximum shift. At the same time any non-maximum shift in EBOM requires at least 2 extra readings from two-dimensional arrays as well as a non-maximum shift in GSB.

The computational experiments (Section 7) show that the MAW22 algorithm strongly outperforms the MAW2 for small alphabets ($|\Sigma| \leq 8$). Moreover, the MAW22 outperforms all the other known algorithms on short patterns in genomic sequences ($3 \leq m \leq 11$, $|\Sigma| = 4$) and on short and medium-size patterns ($3 \leq m \leq 72$) when $|\Sigma| = 8$. The MAW22 remains more efficient than the MAW2 on a random alphabet of size 16, although both are slightly inferior to other algorithms, such as EBOM, SDNDMq2 or Hash3. The MAW22 algorithm becomes too slow for larger alphabets, $|\Sigma| \geq 32$. This is not only due to superfluous character checks, but mostly due to enlarging the search table that may not fit into the cache memory.

4 Reducing the size of the search tables

The aforementioned cashing problems make the MAW22 algorithm impractical for search in natural language texts and other useful applications, where $|\Sigma|$ is greater than $25 - 30$. However, since the array $M22$ contains not more than $2m$ different values, likely it can be represented in a more compact form. Of course, we should invent such representation that does not reduce the access speed greatly. For this goal, we use 4 one-dimensional arrays $V_0, \ldots, V_3$. The array V_3 contains the shift lengths, while for $i < 3$ the array V_i contains the pointers to some elements of V_{i+1}. In C language these arrays can be declared as follows:

```
int ***V0[], **V1[], *V2[], V3[];
```

Also we need the pointers

```
int ***p1, **p2, *p3;
```

And the shift length r can be retrieved from the arrays as follows:

```
p1 = V0[T[pos]];
p2 = p1[T[pos + 1]];
p3 = p2[T[pos + m]];
r = p3[T[pos + m + 1]];
```

To explain how the arrays V_i are organized let us assume that the search window is aligned with the beginning of the text and $x_0 = T[m-2], x_1 = T[m-1], x_2 = T[2m-2], x_3 = T[2m-1]$. The maximum possible safe shift based on the knowledge of $x_0, \ldots, x_i$ we call the shift over the vector $(x_0, \ldots, x_i)$. The arrays V_i can be divided into chunks of $|\Sigma|$ elements each and each element of V_i contains the pointer to the beginning of some chunk of V_{i+1}. The j-th chunk of V_i is processed under the assumption that the shift over $(x_0, \ldots, x_{i-1})$ is equal to j. And the k-th element of each chunk of V_i corresponds to the shift over the vector $(x_0, \ldots, x_i)$ under the assumption that $x_i = k$. In other words, if the k-th element of the j-th chunk of the array V_i contains the pointer to the beginning of the t-th chunk of the array V_{i+1}, this means that the shift over the vector $(x_0, \ldots, x_i)$ is t under the assumption that $x_i = k$ and the shift over the vector $(x_0, \ldots, x_{i-1})$ is j.

The formal definitions of the arrays $V_0, \ldots, V_3$ are as follows.

$$
\begin{aligned}
V_0[i] &= |\Sigma| \cdot \min(\{0 \le t < m-1 |\ P[m-2-t] = i\} \cup \{m-1\}) \\
V_1[|\Sigma|i + j] &= |\Sigma| \cdot \min(\{0 \le t < m-2 |\ P[m-2-t] = i \text{ AND } P[m-1-t] = j\} \cup \\
&\qquad \{P[0] = j \Rightarrow m-1\} \cup \{m\}) \\
V_2[|\Sigma|i + j] &= |\Sigma| \cdot \min(\{i < m-1 \Rightarrow i\} \cup \{m \le t < 2m-1 |\ P[2m-2-t] = j\} \cup \\
&\qquad \{i = m-1 \text{ AND } P[m-1] = j \Rightarrow m-1\} \cup \{2m-1\}) \\
V_3[|\Sigma|i + j] &= \quad \min(\{i < m \Rightarrow i\} \cup \{P[0] = j \Rightarrow 2m-1\} \cup \{2m\} \cup \\
&\qquad \{m \le t < 2m-1 |\ P[2m-2-t] = i \text{ AND } P[2m-1-t] = j\})
\end{aligned}
$$

This principle is illustrated in Figure 2. The search tables for the pattern AGAT are shown, where Σ =AGCT and $A = 0, G = 1, C = 2, T = 3$. The pointer values are shown as the offsets from the beginning of the array V_i. The arrows show the pointer directions for the text characters x_0, x_1, x_2, x_3 =TAGG.

The search algorithms based on tables V_i we call the "Multiple adjacent windows with pointers" (MAWP). The search phases of MAWP2 and MAWP22 algorithms are just the same as in the MAW2 and MAW22 algorithms, except the computation of r, which is performed via 2 (MAWP2) or 4 (MAWP22) assignments, as shown above.

Let us calculate the size of the search tables for the MAWP2q algorithm. If the maximum shift over the vector $(x_0, \ldots, x_{i-1})$ is s, the table V_i contains $s+1$ chunks with $|\Sigma|$ elements in each. Therefore, the total size is $|\Sigma|(1+m+(m+1)+2m+\ldots+qm) = O(|\Sigma|mq^2)$, which is generally much less than $O(|\Sigma|^{2q})$ for the table $M2q$.

5 The multi-window extension

Let us consider the possibility of processing more than 2 adjacent search windows in one iteration of a search loop. The modification of the MAW2 algorithm is simple: the q-dimensional array Mq should be used instead of $M2$. It is defined as follows.

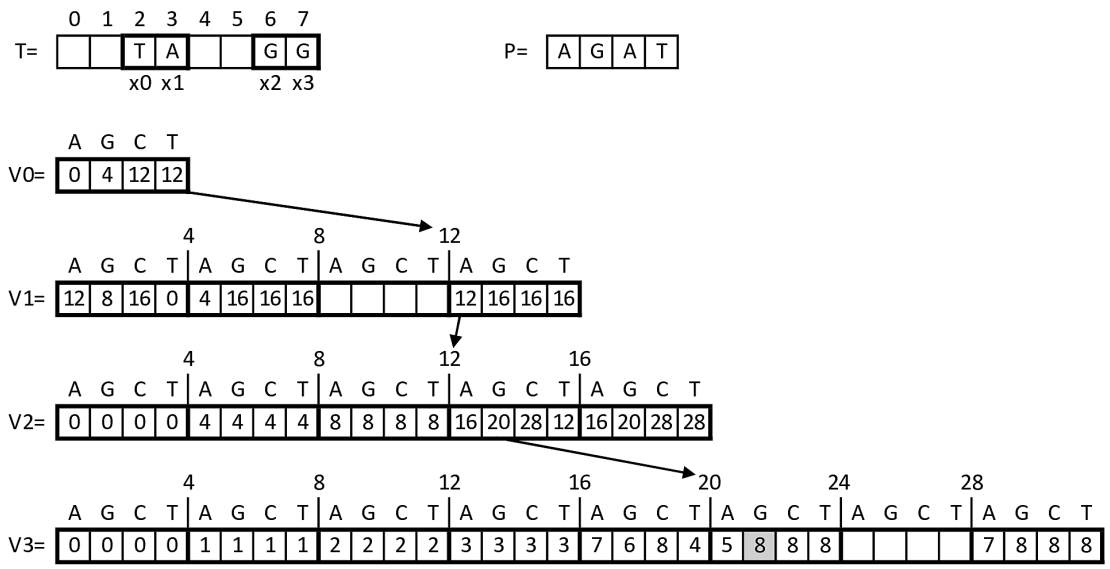


Figure 2. MAWP22 search tables structure

$Mq[i_1]\dots[i_q]$ is the leftmost possible position of a pattern under the assumption that $T[km-1]=i_k$, $k=1,\dots,q$. In Algorithm 2 only the line 3 should be changed in a following way:

$$r \leftarrow Mq[T[pos]][T[pos+m]]\dots[T[pos+qm]]. \tag{1}$$

Thus we obtain the Triple Adjacent Window (MAW3), Quadruple Adjacent Window (MAW4) and other Multiple Adjacent Windows algorithms. Using C notation assignment (1) can be rewritten as $r = *(Mq + b_{q-1}pos + \dots + b_1(pos+(q-1)m) + pos + qm)$, where $b_k = |\Sigma|^k$. The values $2m,\dots,qm$ can be pre-calculated to reduce the number of multiplications, while the values b_k are the constants.

Analogously, we can obtain the MAW32, MAW42 etc. algorithms using the $2q$-dimensional array instead of 4-dimensional in the line 2 of Algorithm 3.

However, every next dimension adds two additions, one multiplication and two memory reads in the case of MAWq and twice as large in the case of MAWq2. This overhead is covered by longer shifts until the value of $|\Sigma_P|/|\Sigma|$ is small enough. Nevertheless, any of MAWq or MAWq2 algorithms, $q \geq 3$, does not outperform the MAW2 or MAW22 respectively for any (Σ, m)-pair in the computational environment we used for the experiment. This is because (1) the probability of 3 or more adjacent maximum bad character or bad bigram shifts is not high enough and (2) filling and accessing the large search table may be time consuming.

6 Preprocessing

On the preprocessing stage of the MAWq algorithm filling the array Mq is the most time consuming operation. The following procedure completes this task.

Algorithm 6: Building the search table Mq

1 Assign the value qm to all elements of the array Mq;
2 **for** $t \leftarrow q$ downto 1 **do**
3 Replace the values $Mq[i_1]\dots[i_t]\dots[i_q]$, where $i_t \in P$, with $tm - r_t - 1$, where r_t is the position of the rightmost occurrence of i_t in P.

The first step takes $O(|\Sigma|^q)$ time, while each iteration of the loop requires $O(m|\Sigma|^{q-1})$ time. The overall time complexity of the preprocessing stage is $O(|\Sigma|^q + qm|\Sigma|^{q-1})$.

The preprocessing stage of the MAWq2 algorithm is more complicated, however, the main principle remains the same:

Algorithm 7: Building the search table $Mq2$

```
1 Assign the value qm to all elements of the array Mq2;
2 Replace the values Mq2[i_1]...[i_{2q-1}][P[0]], where i_1,...,i_{2q-1} are any
  characters, with qm − 1;
3 for t ← q downto 1 do
4     Replace the values Mq2[i_1]...[i_{2t-1}][i_{2t}]...[i_{2q}], where bigram (i_{2t-1}, i_{2t})
      belongs to the pattern, with tm − r_t − 1, where r_t is the rightmost position
      of this bigram in the pattern P (the position of i_{2t-1});
5     if t > 1 then
6         Replace the values Mq2[i_1]...[P[0]][P[m − 1]][i_{2t}]...[i_{2q}] with
          (t − 1)m − 1;
```

Using the special functions that copy memory blocks, like memcpy from memory.h C library, we have built the implementation [21] that is faster in times than the conventional method given above. The space complexity of Multi-window algorithms is, of course, strongly greater than that one of the BMH/TBM/QS. However, the array $M2$ occupies only 64Kb of memory even for a relatively large alphabet containing 256 symbols, which is absolutely admissible for present-day computers and programs. The size of the array $M22$ is equal 64Kb for $|\Sigma| = 16$, although it is 1Mb for $|\Sigma| = 32$, which may be too big to fit the search table into the cache memory and makes the preprocessing time significant (Table 7).

Also, the different methods of filling the MAWP search tables could be developed. We implement one of them in C language ([21]). It utilizes the modified BMH and BR search tables to obtain the bad character or bad bigram heuristic. Of course, the size of the BR shift table should be taken into account, which increases the space complexity to $O(|\Sigma|mq^2 + |\Sigma|^2)$. Nevertheless, it is significantly smaller than the size of the table $M2q$. And this makes the MAWP2q methods applicable to alphabets of size 128 and more, for example, to ASCII texts, without any transformation of their characters.

7 Experimental results

We implement the QLQS, GSB, JOM and different MAW/MAWP algorithms in C language and take the source code of a number of other known algorithms from the SMART tool [10]. We choose the algorithms that were considered the fastest ones at least for one $(|\Sigma|, m)$-combination, genome sequence or English text according to [9] or our own experiments. JOM, QS, TBM and BMH times are given for comparison with the MAW2. By $\text{JOM}_{\max}$ we denote the JOM algorithm, where the adjustable shift heuristic is replaced with its maximum possible value $m + 1$. On a random text such modification is more efficient than the original version. We use the Microsoft Visual Studio 2015 compiler with the Release configuration for Win32 platform to build the executables and run them on the Intel BYT-M Core2 2840 processor of 2.16 GHz, 1 MB of L2-cache, 4GB of RAM, Windows 10 OS. The texts over alphabets of size 8 and 32 contain 5 MB of randomly generated characters with the uniform distribution, while the 4.8MB English text (Bible, KJV version) and 4.4MB genome sequence

Table 3. Running times for a genome sequence $|\Sigma| = 4$

m	2	3	4	5	6	7	8	9	10	11	12	13	14	15
MAW22	15.43	**9.74**	**7.68**	**6.39**	**5.55**	**5.00**	**4.57**	**4.25**	**4.04**	**3.86**	**3.66**	3.55	3.48	3.34
MAW22P	17.33	11.09	8.64	7.22	6.28	5.67	5.19	4.83	4.59	4.38	4.17	4.04	3.96	3.81
MAW23	16.85	10.66	8.58	7.28	6.41	5.87	5.42	5.10	4.91	4.72	4.51	4.42	4.39	4.24
MAW2	22.67	16.39	13.91	12.46	11.55	11.15	10.82	10.52	10.54	10.43	10.09	10.15	10.04	10.18
QLQS	20.42	16.83	15.17	13.92	13.07	12.63	12.23	11.88	12.04	11.76	11.38	11.41	11.51	11.52
Hash3	–	31.12	14.62	9.97	7.69	6.35	5.46	4.84	4.38	3.99	3.70	**3.47**	3.28	3.11
EBOM	15.80	11.53	9.10	8.27	7.58	6.97	6.58	6.23	5.95	5.65	5.32	5.07	4.91	4.66
TVSBS	**14.28**	12.28	10.83	9.64	8.71	7.92	7.35	6.89	6.53	6.19	5.87	5.65	5.54	5.29
FSBNDM	23.36	15.04	11.76	9.63	8.14	7.21	6.44	5.81	5.36	4.95	4.58	4.30	4.05	3.81
SA	14.30	13.91	13.92	13.92	13.87	13.86	13.87	13.86	13.86	13.86	13.86	13.88	13.87	13.86
SBNDMq2	27.78	16.65	12.05	9.96	8.58	7.57	6.88	6.30	5.85	5.45	5.06	4.76	4.49	4.23
SBNDMq4	–	–	44.12	22.53	15.13	11.49	9.30	7.83	6.81	6.03	5.42	4.95	4.55	4.21
GSB	–	–	10.42	9.01	8.08	7.33	6.81	6.35	5.96	5.59	5.21	4.92	4.65	4.39
FSB31	34.54	18.85	12.80	9.83	8.02	6.80	6.01	5.38	4.91	4.55	4.23	3.97	3.78	3.57
FSB41	–	37.19	19.16	13.01	9.87	7.98	6.84	5.93	5.23	4.64	4.52	3.81	3.75	3.35
FSB51	–	–	19.31	12.97	9.87	8.13	6.69	5.78	5.20	4.65	4.44	3.83	3.73	3.24
BSDM4	–	–	27.34	13.43	9.73	7.43	6.45	5.28	4.95	3.98	**3.66**	3.60	**3.13**	**3.08**

Table 4. Running times for English text

m	4	5	6	8	10	12	14	16	18	20	22	24
MAW22P	4.70	3.82	3.23	2.58	**2.13**	**1.87**	**1.70**	**1.53**	**1.47**	**1.37**	**1.32**	1.33
MAW2	4.40	3.74	3.32	2.76	2.38	2.16	2.01	1.85	1.81	1.70	1.64	1.67
MAW3P	5.15	4.47	4.02	3.43	3.00	2.76	2.59	2.40	2.38	2.25	2.21	2.24
JOM	5.75	5.69	5.24	4.28	4.02	3.45	2.97	2.73	2.47	2.31	2.24	2.23
BMH	5.49	4.60	4.03	3.31	2.78	2.46	2.25	2.03	1.97	1.83	1.74	1.75
QS	5.09	4.29	3.71	3.10	2.63	2.33	2.12	1.91	1.85	1.72	1.65	1.64
QLQS	5.31	4.67	4.13	3.57	3.13	2.83	2.64	2.39	2.36	2.21	2.15	2.12
Hash3	10.92	7.38	5.59	3.85	2.96	2.43	2.08	1.83	1.65	1.50	1.39	**1.32**
EBOM	4.38	**3.55**	**3.05**	**2.57**	2.18	1.96	1.82	1.59	1.63	1.47	1.41	1.54
TVSBS	**4.27**	3.71	3.30	2.70	2.29	2.02	1.83	1.65	1.55	1.42	1.34	1.33
FSBNDM	5.31	4.38	3.80	3.07	2.55	2.25	2.05	1.78	1.76	1.58	1.48	1.57
SBNDMq4	38.46	19.44	13.06	8.01	5.81	4.61	3.84	3.28	2.92	2.60	2.36	2.22
GSB	5.79	4.57	3.87	3.17	2.70	2.43	2.24	2.02	1.99	1.85	1.78	1.85
FSB31	9.54	7.25	5.86	4.35	3.46	2.91	2.53	2.21	2.06	1.85	1.71	1.69

(E.Coli bacterium) we take from the SMART tool. The patterns are randomly taken from the text (all algorithms run on the same texts and patterns).

To increase the confidence of the results we measure the time of 1000 runs of each algorithm on the same text and 1000 different patterns. We repeat the 1000-run series 10 times generating a new random text for each series and calculate the standard deviation of the series time. For any algorithm it is less than 5% when $m = 2$ and less than 1% for longer patterns. The running time includes the preprocessing time. We use the fast memory fill functions to build the shift tables on the preprocessing stage of the MAW algorithms.

The results for different texts and pattern lengths are shown in Tables 3–6. They represent the average running time over all 10 000 runs in milliseconds.

As seen, the MAW22 algorithm performs the best for the alphabet of size 8. In this case it outperforms all other algorithms not only for short patterns, but also for mid-size up to $m = 72$. For longer patterns the Hash5 algorithm becomes superior. For the genomic sequences ($|\Sigma| = 4$), the area of MAW22 superiority is narrower:

Table 5. Experimental results on rand8 problem

m	2	3	4	8	16	24	32	40	48	56	64	72	80	88
MAW22	10.76	**7.07**	**5.08**	**2.80**	**1.73**	**1.40**	**1.23**	**1.15**	**1.08**	**1.05**	**1.03**	**0.99**	0.99	0.99
MAW22P	12.55	8.11	5.83	3.22	1.96	1.59	1.40	1.30	1.22	1.19	1.18	1.13	1.12	1.13
MAW23	17.78	11.61	8.29	4.67	2.95	2.43	2.17	2.06	1.98	1.95	1.96	1.87	1.88	2.00
BMH	14.45	10.42	7.93	5.12	3.96	3.58	3.44	3.49	3.51	3.57	3.75	3.57	3.55	3.52
QS	12.11	9.30	7.41	4.82	3.63	3.23	3.10	3.15	3.14	3.17	3.36	3.20	3.20	3.16
QLQS	12.86	10.04	8.40	6.32	5.41	4.98	4.82	4.92	4.90	4.94	5.26	5.00	4.99	4.93
MAW2	12.32	8.57	6.65	4.61	3.92	3.64	3.55	3.61	3.64	3.70	3.88	3.70	3.68	3.65
Hash3	–	30.29	13.96	5.08	2.56	1.90	1.59	1.45	1.34	1.29	1.25	1.18	1.16	1.14
Hash5	–	–	–	8.33	3.07	2.03	1.59	1.37	1.23	1.14	1.06	1.01	**0.96**	**0.93**
EBOM	14.05	7.91	5.48	3.15	2.21	1.92	1.71	1.57	1.44	1.34	1.26	1.17	1.10	1.07
TVSBS	**9.88**	8.07	6.47	4.11	2.55	1.97	1.67	1.51	1.39	1.31	1.27	1.19	1.16	1.14
FSBNDM	14.83	9.70	7.00	4.02	2.53	2.01	1.68	1.60	1.46	1.35	1.26	1.27	1.24	1.20
SBNDMq2	26.43	13.96	9.16	4.52	2.64	2.07	1.75	1.66	1.54	1.42	1.32	1.29	1.25	1.21
GSB	–	–	8.71	4.69	3.09	2.58	2.28	2.14	1.94	1.80	1.69	1.67	1.66	1.63
FSB31	34.46	17.66	11.33	5.11	2.64	1.90	1.55	1.48	1.39	1.32	1.25	1.28	1.26	1.22
FSB41	–	41.78	20.64	7.16	3.29	2.26	1.76	1.54	1.38	1.27	1.18	1.17	1.15	1.13
FSB51	–	–	20.59	7.16	3.29	2.26	1.76	1.54	1.38	1.28	1.19	1.17	1.16	1.13

Table 6. Experimental results on rand32 problem

m	2	3	4	5	6	7	8	9	10
BMH	10.65	7.07	5.29	4.37	3.74	3.29	2.96	2.70	2.49
QS	7.18	5.30	4.23	3.61	3.18	2.85	2.60	2.40	2.25
QLQS	7.76	5.66	4.49	3.85	3.42	3.09	2.86	2.67	2.52
TBM	8.01	5.26	4.07	3.40	2.96	2.57	2.37	2.13	2.02
MAW2	7.17	**4.83**	**3.67**	**3.08**	**2.68**	2.39	2.19	2.04	1.91
MAW2P	8.26	5.55	4.21	3.51	3.06	2.72	2.48	2.31	2.15
MAW22P	10.51	6.86	5.09	4.15	3.53	3.09	2.75	2.49	2.30
JOM	8.05	6.46	5.25	4.84	4.06	3.61	3.29	3.09	2.91
JOM_{max}	7.97	6.16	4.70	4.29	3.31	3.08	2.88	2.72	2.65
Hash3	–	27.14	13.22	8.95	6.81	5.52	4.69	4.07	3.60
EBOM	12.97	6.44	4.32	3.34	2.74	**2.38**	**2.09**	**1.89**	**1.73**
TVSBS	**6.44**	5.10	4.14	3.60	3.18	2.87	2.62	2.40	2.25
FSBNDM	9.83	6.35	4.73	3.86	3.28	2.87	2.56	2.35	2.14
GSB	–	–	7.54	5.83	4.80	4.12	3.63	3.26	2.98
FSB31	33.79	16.57	10.84	8.23	6.66	5.61	4.87	4.32	3.88

$3 \leq m \leq 11$, then the BSDM4 and Hash3 algorithms perform better. For larger alphabets the MAW22 algorithm becomes inefficient due to the enlarging search table. However, the MAW22P algorithm appears on the scene. It outperforms all the other when searching the patterns of the length $10 \leq m \leq 22$ in English text. We test the English text in ASCII encoding and assume $|\Sigma| = 128$. The MAW2 algorithm appears to be the fastest one for short patterns ($3 \leq m \leq 6$) and a random text over the alphabet of size 32. It also performs well on short patterns and larger alphabets, but becomes slightly inferior to some other algorithms, such as FJS [11].

Also, we measured the preprocessing and search time separately for 5MB random texts over the alphabets of different size and pattern lengths 10 and 20. Table 7 presents times of search in milliseconds and the ratio of preprocessing to search. As expected, the preprocessing time of the MAW22 grows rapidly depending on the alphabet size, making this algorithm impractical even for $|\Sigma| = 32$. At the same time, in all other algorithms the ratio of preprocessing to search remains reasonable even

Table 7. The balance between the preprocessing and the search

		MAW2		MAW2P		MAW22		MAW22P	
$\|\Sigma\|$	m	search	preprocessing	search	preprocessing	search	preprocessing	search	preprocessing
4	10	4.42	0.00013%	5.03	0.027%	1.90	0.044%	2.09	0.85%
4	20	3.69	0.00063%	4.25	0.065%	1.35	0.15%	1.49	2.35%
32	10	1.91	0.14%	2.14	0.31%	17.97	15.4%	2.27	4.1%
32	20	1.14	0.29%	1.27	0.45%	7.78	28.6%	1.12	6.4%
128	10	3.55	0.6%	3.94	0.3%	114.99	1151%	5.18	3.2%
128	20	2.94	2.2%	3.09	0.5%	61.7	2271.7%	3.38	5.4%

for $|\Sigma| = 128$. Note that in the MAW22 even the search phase itself is too expensive for $|\Sigma| \geq 32$ and this problem is also resolved in the MAW22P.

8 Conclusions

A new family of exact pattern matching algorithm is developed. They exploit the idea of processing more than one adjacent search window in each iteration of the search loop using the multi-dimensional search tables. This approach allows to decrease the search time by the cost of space. However, the space overhead is admissible even for large alphabets if we replace the multi-dimensional search tables with a series of one-dimensional tables linked by pointers. We carry out an experiment for English text, genome sequences and random texts over the alphabets of size 8 and 32. The performance of our algorithms was compared with other algorithms, which are known as the fastest ones for respective alphabet size / pattern length. In our computational environment the MAW22 algorithm outperforms all the other in searching genome sequences of length $3 \leq m \leq 11$ and random patterns of length $3 \leq m \leq 72$ for $|\Sigma| = 8$. The MAW22P algorithm demonstrates the best performance in searching the patterns of length $10 \leq m \leq 22$ in English text.

The multiple adjacent windows approach can be developed further. For example, it is worthwhile to investigate even more economic methods of packing the multi-dimensional shift information. This would make possible to construct practically efficient algorithms based on checking the q-grams ($q \geq 3$) in adjacent search windows.

References

1. T. Berry and S. Ravindran: *A fast string matching algorithm and experimental results*, in Proceedings of the Prague Stringology Club Workshop, Edited by Jan Holub and Milan Simanek, Eds. Czech Technical University in Prague, Czech Republic, 1999, pp. 16–28.
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: *Improved and self-tuned occurrence heuristics.* Journal of Discrete Algorithms, 28 2014, pp. 73–84.
4. B. Durian, J. Holub, H. Peltola, and J. Tarhio: *Tuning bndm with q-grams*, in Proceedings of the Workshop on Algorithm Engineering and Experiments, I. Finocchi and J. Hershberger, Eds. SIAM, New York, New York, USA, 2009, pp. 29–37.
5. S. Faro: *Evaluation and improvement of fast algorithms for exact matching on genome sequences*, in Algorithms for computational biology, Proceedings of Third International Conference, 2016, pp. 145–157.
6. S. Faro and T. Lecroq: *Efficient variants of the backward-oracle-matching algorithm*, in Proceedings of the Prague Stringology Conference, J. Holub and J. Zdarek, Eds. Czech Technical University in Prague, Czech Republic, 2008, pp. 146–160.

7. S. Faro and T. Lecroq: *A fast suffix automata based algorithm for exact online string matching*, in Moreira, N., Reis, R. (eds.) CIAA, vol. 7381, 2012, pp. 149–158.
8. S. Faro and T. Lecroq: *A multiple sliding windows approach to speed up string matching algorithms*, in 11-th International Symposium on Experimental Algorithms (SEA), 2012, pp. 172–183.
9. S. Faro and T. Lecroq: *The exact online string matching problem: a review of the most recent results.* ACM Computing Surveys (CSUR), 45(2) 2013, p. article 13.
10. S. Faro, T. Lecroq, S. Borzi, S. D. Mauro, and A. Maggio: *The string matching algorithms research tool*, in Proceedings of the Prague Stringology Conference, J. Holub and J. Zdarek, Eds. Czech Technical University in Prague, Czech Republic, 2016, pp. 99–111.
11. F. Franek, C. Jennings, and W. Smyth: *A simple fast hybrid pattern-matching algorithm.* J. Discret. Algorithms, 5(4) 2007, pp. 682–695.
12. N. R. Horspool: *Practical fast searching in strings.* Soft.Pract.Exp., 10(6) 1980, pp. 501–506.
13. A. Hudaib, R. Al-Khalid, A. Al-Anani, M. Itriq, and D. Suleiman: *Four sliding windows pattern matching algorithm (fsw).* Journal of Software Engineering and Applications, 8(8) 2015, pp. 154–165.
14. A. Hume and D. Sunday: *Fast string searching.* Software - Practice & Experience, 21(11) 1991, pp. 1221–1248.
15. T. Lecroq: *Fast exact string matching algorithms.* Inf. Process. Lett., 102(6) 2007, pp. 229–235.
16. H. Peltola and J. Tarhio: *String matching with lookahead.* Discrete Applied Mathematics, 163 2014, pp. 352–360.
17. D. M. Sunday: *A very fast substring search algorithm.* Comm. ACM, 33(8) 1990, pp. 132–142.
18. R. Thathoo, A. Virmani, S. S. Lakshmi, N. Balakrishnan, and K. Sekar: *Tvsbs: A fast exact pattern matching algorithm for biological sequences.* Indian Acad. Sci., Current Sci., 91(1) 2006, pp. 47–53.
19. B. W. Watson, D. G. Kourie, and L. G. Cleophas: *Quantum leap pattern matching*, in Proceedings of the Prague Stringology Conference, J. Holub and J. Zdarek, Eds. Czech Technical University in Prague, Czech Republic, 2015, pp. 104–117.
20. S. Wu and U. Manber: *Fast text searching allowing errors.* Commun. ACM, 35(10) 1992, pp. 83–91.
21. I. O. Zavadskyi: *The maw/mawp algorithms.* (https://github.com/zavadsky/stringology).

Author Index

Proceedings of the Prague Stringology Conference 2017
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 978-80-01-06193-0

URL: `http://www.stringology.org/`
E-mail: `psc@stringology.org` Phone: +420-2-2435-9811

Printed by Česká technika – Nakladatelství ČVUT
Zikova 4, Praha 6, 166 36, Czech Republic