Proceedings of the Prague Stringology Conference 2019

Edited by Jan Holub and Jan Žďárek

August 2019

PSC

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

ISBN 978-80-01-06618-8

Preface

The proceedings in your hands contains a collection of papers presented in the Prague Stringology Conference 2019 (PSC 2019) held on August 26–28, 2019 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 eleven 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 "Pattern Matching on Weighted Strings" by Jakub Radoszewski.

The Prague Stringology Conference has a long tradition. PSC 2019 is the twenty-third PSC conference. In the years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conferences (PSC's) in 2001–2006, 2008–2018 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 2019 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 2019. 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 2019

Jan Holub and Gabriela Andrejková

Conference Organisation

Program Committee

Amihood Amir	(Bar-Ilan University, Israel)
Gabriela Andrejková, *Co-chair*	(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	(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)
Marie-France Sagot	(INRIA Rhône-Alpes, France)
William F. Smyth	(McMaster University, Canada, and Murdoch University, Australia)
Bruce W. Watson	(FASTAR Group/Stellenbosch University, South Africa)
Jan Žďárek	(Czech Technical University in Prague, Czech Republic)

Organising Committee

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

External Referees

Keisuke Goto	Neerja Mhaskar	Takuya Mieno
Arnaud Lefebvre		

Table of Contents

Pattern Matching on Weighted Strings
(*Abstract*)

Jakub Radoszewski[⋆]

Institute of Informatics, University of Warsaw, Warsaw, Poland
Banacha 2, 02-097 Warszawa, Poland
`jrad@mimuw.edu.pl`

Keywords: weighted string, weighted sequence, uncertain string, Position-Weight Matrix (PWM), pattern matching

A weighted string is a sequence of probability distributions over a given finite alphabet Σ. A weighted string represents many standard strings, each with the probability of occurrence equal to the product of probabilities of its letters at subsequent positions of the weighted string. Usually a threshold $1/z > 0$ is specified and one considers as matches only the strings for which the probability of occurrence is at least $1/z$.

Weighted strings, also known as position weight matrices or uncertain strings, arise naturally in many applications. In molecular biology, position weight matrices were introduced as an alternative to consensus sequences and may appear due to flexible sequence modeling, such as binding profiles of molecular sequences. Weighted strings are also present in mining applications due to imprecise data measurements or when observations are private and thus sequences of observations may have artificial uncertainty introduced deliberately. Weighted strings can also be viewed as a generalization of indeterminate strings (i.e., degenerate strings).

This talk will focus on the solutions to Weighted Pattern Matching problem, in which we are to find all positions of the given weighted text where a given string pattern occurs with probability above the threshold, and its indexing variant. We will also survey other algorithmic results on weighted strings, including a variant of Weighted Pattern Matching in which both the text and the pattern are weighted, practical approaches to Weighted Pattern Matching, and differences between the longest common subsequence and shortest common supersequence problems on weighted strings.

[⋆] The author was supported by the "Algorithms for text processing with errors and uncertainties" project carried out within the HOMING programme of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.

Computing Maximal Palindromes and Distinct Palindromes in a Trie

Mitsuru Funakoshi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda

Department of Informatics, Kyushu University, Japan
{mitsuru.funakoshi, yuto.nakashima, inenaga, bannai, takeda}@inf.kyushu-u.ac.jp

Abstract. It is known that all maximal palindromes of a given string T of length n can be computed in $O(n)$ time by Manacher's algorithm [J. ACM '75]. Also, all distinct palindromes in T can be computed in $O(n)$ time [Groult et al., Inf. Process. Lett. 2010]. In this paper, we consider the problem of computing maximal palindromes and distinct palindromes of a given trie $\mathcal{T}$ (i.e. rooted edge-labeled tree). A trie is a natural generalization of a string which can be seen as a single path tree. We propose algorithms to compute all maximal palindromes and all distinct palindromes in $\mathcal{T}$ in $O(N \log h)$ time and $O(N)$ space, where N is the number of edges in $\mathcal{T}$ and h is the height of $\mathcal{T}$. To our knowledge these are the first sub-quadratic time solutions to these problems.

Keywords: palindromes, string and tree algorithms, periodicity, suffix arrays

1 Introduction

Palindromes are strings that read the same forward and backward. Finding palindromic structures in a given string is a fundamental task in string processing, and thus it has extensively been studied (e.g., see [2,27,17,25,33,23,32,14] and references therein).

Consider a set $C = \{1, 1.5, 2, \ldots, n\}$ of $2n-1$ half-integer and integer positions in a string T of length n. The maximal palindrome for a position $c \in C$ in T is a non-extensible palindrome whose center lies on c. It is easy to store all maximal palindromes with $O(n)$ total space; e.g., simply store their lengths in an array of length $2n-1$ together with the input string T. If $P = T[i..j]$ is a maximal palindrome with center $c = \frac{i+j}{2}$, then clearly any substrings $P' = T[i+d..j-d]$ with $0 \leq d \leq \frac{j-i}{2}$ are also palindromes. Hence, by computing and storing all maximal palindromes in T, we can obtain a compact representation of all palindromes in T. Manacher [26] gave an elegant $O(n)$-time algorithm to compute all maximal palindromes in T. This algorithm works for a general alphabet. For the case where the input string is drawn from a constant size alphabet or an integer alphabet of size polynomial in n, there is an alternative suffix tree [38] based algorithm which takes $O(n)$ time [18]. In this method, the suffix tree of $T\#T^R\$$ is constructed, where T^R is the reversed string of T, and # and \$ are special characters not occurring in T. By enhancing the suffix tree with a lowest common ancestor (LCA) data structure [10], *outward* longest common extension (LCE) queries from a given $c \in C$ can be answered in $O(1)$ time after an $O(n)$-time preprocessing.

Another central question regarding substring palindromes is distinct palindromes. Droubay et al. [9] showed that any string of length n can contain at most $n+1$ distinct palindromes (including the empty string). Strings of length n that contain exactly $n+1$ distinct palindromes are called *rich* strings in the literature [16,7]. Groult et al. [17] proposed an $O(n)$-time algorithm for computing all distinct palindromes in a string of length n over a constant-size alphabet or an integer alphabet of size polynomial in n.

A *trie* is a rooted tree where each edge is labeled by a single character and the out-going edges of each node are labeled by mutually distinct characters. A trie is a natural extension to a string, and is a compact representation of a set of strings. There are a number of works for efficient algorithms on tries, such as indexing a (reversed) trie [5,24,35,21,29,11,31,20] for exact pattern matching, parameterized pattern matching on a trie [1,12], order preserving pattern matching on a trie [30], and finding all maximal repetitions (a.k.a. runs) in a trie [37].

In this paper, we tackle the problems of computing all maximal palindromes and all distinct palindromes in a given trie $\mathcal{T}$. Naïve methods for solving these problems would be to apply Manacher's algorithm [26] or Groult et al.'s algorithm [17] for each string in $\mathcal{T}$, but this requires $\Omega(N^2)$ time in the worst case since there exists a trie with N edges that can represent $\Theta(N)$ strings of length $\Theta(N)$ each. We also remark that a direct application of Manacher's algorithm to a trie does not seem to solve our problem efficiently, since the amortization argument in the case of a single string does not hold in our case of a trie. The aforementioned suffix tree approach [18] cannot be applied to our trie case either; while the number of suffixes in the reversed leaf-to-root direction of the trie $\mathcal{T}$ is N, the number of suffixes in the forward root-to-leaf direction can be $\Theta(N^2)$ in the worst case. Thus one cannot afford to construct the suffix tree that contains all suffixes of the forward paths of $\mathcal{T}$.

In this paper, we first show that the number of maximal palindromes in a trie $\mathcal{T}$ with N edges and L leaves is exactly $2N - L$ and that the number of distinct palindromes in $\mathcal{T}$ is at most $N + 1$. These generalize the known bounds for a single string. Then, we present two algorithms to compute all maximal palindromes both of which run in $O(N \log h)$ time and $O(N)$ space in the worst case, where h is the height of the trie $\mathcal{T}$. We then present how to compute all distinct palindromes in a given trie $\mathcal{T}$ in $O(N \log h)$ time with $O(N)$ space. The key tools we use are periodicities of suffix palindromes and string data structures that are built on the (reversed) trie. To the best of our knowledge, these are the first algorithms for finding maximal/distinct palindromes from a given trie in sub-quadratic time.

Related work

There are a few combinatorial results for palindromes in an *unrooted* edge-labeled tree. Brlek et al. [6] showed an $\Omega(M^{3/2})$ lower bound on the maximum number of distinct palindromes in an unrooted tree with M edges. Later Gawrychowski et al. [15] showed a matching upper bound $O(M^{3/2})$ on the maximum number of distinct palindromes in an unrooted tree with M edges. Note that these previous works consider an unrooted tree, and, to the best of our knowledge, palindromes of a trie (rooted edge-labeled tree) have previously not been studied. Concerning repetitive structures in tries, Sugahara et al. [37] proved that any trie with N edges can contain less than N maximal repetitions (or runs), and showed that all runs in a given trie can be found in $O(N(\log\log N)^2)$ time with $O(N)$ space. Our paper can be considered as computing palindromes, instead of runs, given the same input.

2 Preliminaries

2.1 String notation

Let Σ be the *alphabet*. An element of Σ^* is called a *string*. The length of a string T is denoted by $|T|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a string $T = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of T, respectively. For two strings X and Y, let $\mathsf{lcp}(X, Y)$ denote the length of the longest common prefix of X and Y.

For a string T and an integer $1 \leq i \leq |T|$, $T[i]$ denotes the ith character of T, and for two integers $1 \leq i \leq j \leq |T|$, $T[i..j]$ denotes the substring of T that begins at position i and ends at position j. For convenience, let $T[i..j] = \varepsilon$ when $i > j$. An integer $p \geq 1$ is said to be a *period* of a string T iff $T[i] = T[i+p]$ for all $1 \leq i \leq |T| - p$.

Let T^R denote the reversed string of T, i.e., $T^R = T[|T|] \cdots T[1]$. A string T is called a *palindrome* if $T = T^R$. We remark that the empty string ε is also considered to be a palindrome. For any non-empty substring palindrome $T[i..j]$ in T, $\frac{i+j}{2}$ is called its *center*. A non-empty substring palindrome $T[i..j]$ is said to be a *maximal palindrome* centered at $\frac{i+j}{2}$ in T if $T[i-1] \neq T[j+1]$, $i = 1$, or $j = |T|$. It is clear that for each center $c = 1, 1.5, \ldots, n-0.5, n$, we can identify the maximal palindrome $T[i..j]$ whose center is c (namely, $c = \frac{i+j}{2}$). Thus, there are exactly $2n - 1$ maximal palindromes in a string of length n. In particular, maximal palindromes $T[1..i]$ and $T[i..|T|]$ for $1 \leq i \leq n$ are respectively called a *prefix palindrome* and a *suffix palindrome* of T.

2.2 Tries and algorithmic tools

A *trie* $\mathcal{T} = (V, E)$ is a rooted tree where each edge in E is labeled by a single character from Σ and the out-going edges of a node are labeled by pairwise distinct characters. For any non-root node u in $\mathcal{T}$, let $\mathsf{parent}(u)$ denote the parent of u. For any node v in $\mathcal{T}$, let $\mathsf{children}(v)$ denote the set of children of v. For any node u and its arbitrary descendant v, we denote by $\mathsf{str}(u, v)$ the substring of $\mathcal{T}$ that begins at u and ends at v.

A trie can be seen as a representation of a set of strings which are root-to-leaf path labels. Note that for a trie with N edges, the total length of such strings can be quadratic in N. An example can be given by the set of strings $X = \{xc_1, xc_2, \cdots xc_N\}$ where $x \in \Sigma^{N-1}$ is an arbitrary string and $c_1, \ldots, c_N \in \Sigma$ are pairwise distinct characters. Here, the size of the trie is $\Theta(N)$, while the total length of strings is $\Theta(N^2)$. Also notice that the total number of distinct suffixes of strings in X is also $\Theta(N^2)$. However if we consider the strings in the reverse direction, i.e., consider edges of the trie to be directed toward the root, the number of distinct suffixes is linear in the size N of the trie. We call it a reversed trie.

Consider a trie with N edges such that the root has a single out-edge labeled with a special character \$ that does not appear elsewhere in the trie and is lexicographically the smallest. We consider the reversed trie of this trie. The *suffix array* of this reversed trie can be constructed in $O(N)$ time [36,11]. Also, the *longest common prefix array* (*LCP array*) for this suffix array can also be constructed in $O(N)$ time [22].

2.3 Computing palindromes in a string

Manacher [26] showed an elegant online algorithm which computes all maximal palindromes of a given string T of length n in $O(n)$ time. An alternative offline approach is

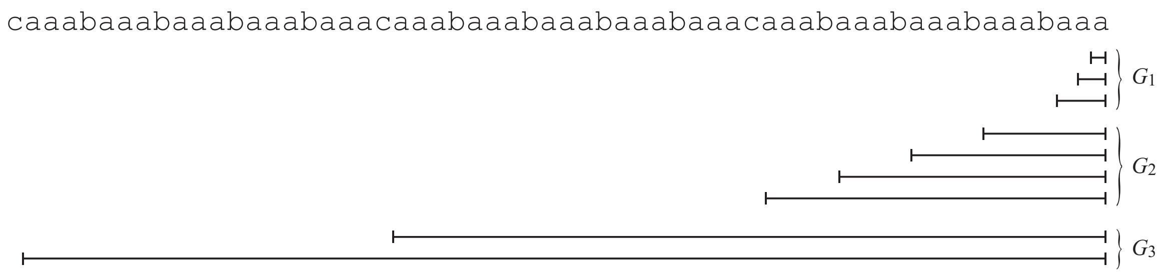

caaabaaabaaabaaabaaacaaabaaabaaabaaabaaacaaabaaabaaabaaabaaa
G_1
G_2
G_3

Figure 1. Examples of arithmetic progressions representing the suffix palindromes of a string. The first group G_1 is represented by $\langle 1,1,3\rangle$, the second group G_2 by $\langle 7,4,4\rangle$, and the third group G_3 by $\langle 39,20,2\rangle$.

to use outward LCE queries for $2n-1$ pairs of positions in T. Using the suffix tree [38] for string $T\$T^R\#$ enhanced with a lowest common ancestor data structure [19,34,3], where \$ and # are special characters which do not appear in T, each outward LCE query can be answered in $O(1)$ time. For any integer alphabet of size polynomial in n, preprocessing for this approach takes $O(n)$ time and space [10,18].

Let T be a string of length n. For each $1 \le i \le n$, let $\mathit{MaxPalEnd}_T(i)$ denote the set of maximal palindromes of T that end at position i. Let $\mathbf{S}_i = s_1, \ldots, s_g$ be the sequence of lengths of maximal palindromes in $\mathit{MaxPalEnd}_T(i)$ sorted in increasing order, where $g = |\mathit{MaxPalEnd}_T(i)|$. Let d_j be the progression difference for s_j, i.e., $d_j = s_j - s_{j-1}$ for $2 \le j \le g$. In particular, let $d_1 = s_1 - |\varepsilon| = s_1$. We use the following lemma which is based on periodic properties of maximal palindromes ending at the same position.

Lemma 1 (Lemma 2 of [13]).

(i) *For any* $1 \le j < g$, $d_{j+1} \ge d_j$.
(ii) *For any* $1 < j < g$, *if* $d_{j+1} \ne d_j$, *then* $d_{j+1} \ge d_j + d_{j-1}$.
(iii) $\mathbf{S}_i$ *can be represented by* $O(\log i)$ *arithmetic progressions, where each arithmetic progression is a tuple* $\langle s,d,t\rangle$ *representing the sequence* $s, s+d, \ldots, s+(t-1)d$ *with common difference* d.
(iv) *If* $t \ge 2$, *then the common difference* d *is a period of every maximal palindrome which ends at position* i *in* T *and whose length belongs to the arithmetic progression* $\langle s,d,t\rangle$.

Each arithmetic progression $\langle s,d,t\rangle$ is called a *group* of maximal palindromes. See also Figure 1 for a concrete example.

Since each arithmetic progression can be stored in $O(1)$ space, and since there are only $O(\log i)$ arithmetic progressions for each position i, we can represent all maximal palindromes ending at position i in $O(\log i)$ space.

For all $1 \le i \le n$ we can compute $\mathit{MaxPalEnd}_T(i)$ in total $O(n)$ time: After computing all maximal palindromes of T in $O(n)$ time, we can bucket sort all the maximal palindromes with their ending positions in $O(n)$ time.

Since suffix palindromes are also maximal palindromes, $\mathit{MaxPalEnd}_T(n)$ is the set of suffix palindromes of T, where $n = |T|$. Thus Lemma 1 holds for suffix palindromes in T. This particular case of Lemma 1 was shown in the literature [2,28].

Our algorithms will make a heavy use of periodicity of maximal/suffix palindromes of a string stated in Lemma 1.

3 Maximal/distinct palindromes in a trie

Consider a trie $\mathcal{T}$ with N edges. A substring palindrome $P = \mathsf{str}(u, v)$ in $\mathcal{T}$ can be represented by the pair $(|P|, v)$ of its length and the ending point v. Since the reversed path from v to u is unique and since P is a palindrome, one can retrieved P from $\mathcal{T}$ in $O(|P|)$ time from this pair $(|P|, v)$.

A substring palindrome $\mathsf{str}(u, v)$ is called a *maximal palindrome* in $\mathcal{T}$ if

(1) $\mathsf{str}(\mathsf{parent}(u), v')$ is not a palindrome with *any* child v' of v,
(2) u is the root, or
(3) v is a leaf.

Lemma 2. *There are exactly $2N - L$ maximal palindromes in any trie $\mathcal{T}$ with N edges and L leaves.*

Proof. Let r be the root of $\mathcal{T}$ and u any internal node of $\mathcal{T}$. Because the reversed path from u to r is unique, and because the out-going edges of u are labeled by pairwise distinct characters, there is a unique longest palindrome of even length (or length zero) that is centered at u. Since there are $N + 1$ nodes in $\mathcal{T}$, there are exactly $(N + 1) - L - 1 = N - L$ maximal palindromes of even length in $\mathcal{T}$.

Let $e = (u, v)$ be any edge in $\mathcal{T}$. From the same argument as above, there is a unique longest palindrome of odd length that is centered at e. Thus there are exactly N maximal palindromes of odd length in $\mathcal{T}$. □

For any trie $\mathcal{T}$, let $\mathbf{P}_{\mathcal{T}} \subset \Sigma^*$ be the set of all strings such that each $P \in \mathbf{P}_{\mathcal{T}}$ is a substring palindrome in $\mathcal{T}$. We call the elements of $\mathbf{P}$ as *distinct palindromes* in $\mathcal{T}$.

Lemma 3. *There are at most $N+1$ distinct palindromes in any trie $\mathcal{T}$ with N edges.*

Proof. We follow the proof from [9] which shows that the number of distinct palindromes in a string of length n is at most $n + 1$.

We consider a top-down traversal on $\mathcal{T}$. The proof works with any top-down traversal but for consistency with our algorithm to follow, let us consider a breadth first traversal. Let r be the root of $\mathcal{T}$ and let $\mathcal{T}_0$ be the trie consisting only of the root r. For each $1 \leq i \leq n$, let $e_i = (u_i, v_i)$ denote the ith visited edge in the traversal, and let $\mathcal{T}_i$ denote the subgraph of $\mathcal{T}_i$ consisting of the already visited edges when we have just arrived at e_i. Since we have just added e_i to $\mathcal{T}_{i-1}$, it suffices to consider only suffix palindromes of $\mathsf{str}(r, v_i)$ since any other palindromes in $\mathsf{str}(r, v_i)$ already appeared in $\mathcal{T}_{i-1}$. Moreover, only the longest suffix palindrome S_i of $\mathsf{str}(r, v_i)$ can be a new palindrome in $\mathcal{T}_i$ which does not exist in $\mathcal{T}_{i-1}$, since any shorter suffix palindrome S' is a suffix of S_i and hence is a prefix of S_i, which appears in $\mathcal{T}_{i-1}$. Thus there can be at most $N + 1$ distinct palindromes in $\mathcal{T}$ (including the empty string). □

See Figure 2 for examples of maximal palindromes and distinct palindromes in a trie.

In the next sections, we will present our algorithms to compute maximal/distinct palindromes from a given trie.

4 Computing maximal palindromes in a trie

In this section, we present two algorithms that compute all maximal palindromes in a given trie.

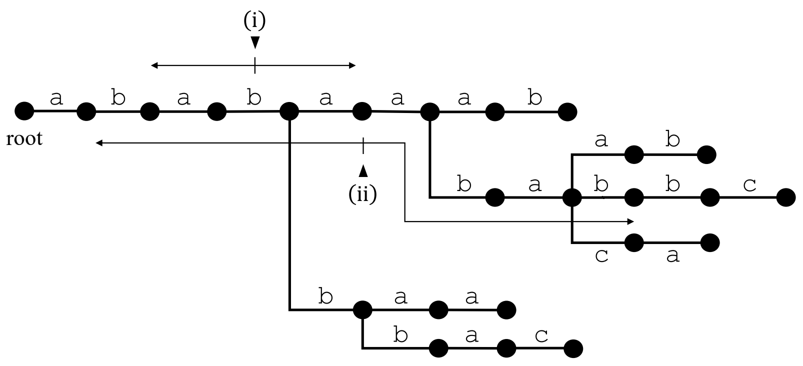


Figure 2. The maximal palindrome centered at (i) is aba and the maximal palindrome centered at (ii) is babaabab. The set of distinct palindromes in this trie is $\{\varepsilon$, a, b, c, aa, bb, aaa, aba, aca, bab, bbb, abba, baab, aabaa, ababa, abbba, baaab, abaaba, baabaab, babaabab$\}$.

4.1 $O(N \log h)$-time $O(h)$-space algorithm

In this section, we present an algorithm that compute all maximal palindromes in a given trie $\mathcal{T}$ in $O(N \log h)$ time and $O(h)$ working space, where N is the number of edges in $\mathcal{T}$ and $h \leq N$ is the height of $\mathcal{T}$.

The basic strategy of our algorithm is as follows. We perform a depth-first traversal on $\mathcal{T}$. Let r be the root of $\mathcal{T}$. We use Lemma 1 in our algorithm. When visiting a node u during the depth-first traversal on trie $\mathcal{T}$, we maintain the arithmetic progressions for the maximal palindromes in the path string $\mathsf{str}(r, u)$. In each node x in the path from r to u, the arithmetic progressions representing the maximal palindromes ending at x are sorted in the increasing order of the lengths of the corresponding maximal palindromes. Since $\mathsf{str}(r, u)$ is a single string, and since $|\mathsf{str}(r, u)|$ is bounded by the height h of $\mathcal{T}$, we can store all these arithmetic progressions in $O(h)$ total space during the traversal. Suppose that u has two or more children, and let v, v' be two distinct children of u. Notice that some of the maximal palindromes ending at u could be extended by the edge label from u to v. Furthermore, since the edge label between u and v differs from the edge label between u and v', those palindromes that are not extended with v could still be extended with v'. This in turn means that when we backtrack to u after visiting v, then we can use the maximal palindromes in the path string $\mathsf{str}(r, v)$ that ends at the parent u of v, for finding the palindromes ending at another child v'. In the sequel, we will describe how to efficiently maintain these maximal palindromes during the traversal.

Suppose that now we are to process non-leaf node u in the traversal. For each $1 \leq i \leq |\mathsf{children}(u)|$, let v_i be the ith visited child of u in the tree traversal, and let a_i be the label of the edge (u, v_i). The task here is to check if the suffix palindromes ending at u extends with a_i. We will process the groups of suffix palindromes ending at u in increasing order of their lengths. Let $\langle s, d, t\rangle$ be the arithmetic progression representing a given group of suffix palindromes ending at u, where s is the length of the shortest suffix palindrome in the group, d is a common period of the suffix palindromes and t is the number of suffix palindromes in this group. The cases where $t = 1$ and $t = 2$ are trivial, so we consider the case where $t \geq 3$. Let P be any suffix palindrome in the group that is not the longest one (i.e, $s \leq |P| \leq s + (t-2)d$). Due to the periodicity (Claim (iv) of Lemma 1), every P is immediately preceded by a unique string $P[1..d]$ of length d. Let $b = P[d]$ and c be the character that immediately precedes the longest suffix palindrome in the group. There are four cases to consider:

1. $a_i = b$ and $a_i = c$ (namely $a_i = b = c$): In this case, all the suffix palindromes in the group extend with a_i and become suffix palindromes of $\mathsf{str}(r, v_i)$. We update $s \leftarrow s + 2$. The values of d and t stay unchanged.
2. $a_i = b$ and $a_i \neq c$. In this case, all the suffix palindromes but the longest one in the group extend with a_i and become suffix palindromes of $\mathsf{str}(r, v_i)$. We update $s \leftarrow s + 2$ and $t \leftarrow t - 1$. The value of d stays unchanged.
3. $a_i \neq b$ and $a_i = c$. In this case, only the longest suffix palindromes in the group extends with a_i and becomes a suffix palindrome of $\mathsf{str}(r, v_i)$. We first update $s \leftarrow s + (t - 1)d + 2$ and then $t \leftarrow 1$. The new value of d is easily calculated from the length of the longest suffix palindrome in the previous group (recall the definition of d just above Lemma 1).
4. $a_i \neq b$ and $a_i \neq c$. In this case, none of the members in the group extends with a_i. Then, we do nothing.

In each of the above cases, we store all these extended palindromes in v_i as the set of maximal palindromes ending at v_i in $\mathsf{str}(r, v_i)$, and exclude all these extended palindromes from the set of maximal palindromes ending at u.

See Figure 1 for concrete examples of the above cases. Let a_i be the next character that is appended to the string in Figure 1. Case 1 occurs to group G_3 when $a_i = \mathtt{c}$. Case 2 occurs to group G_1 when $a_i = \mathtt{a}$, and to group G_2 when $a_i = \mathtt{b}$. Case 3 occurs to group G_1 when $a_i = \mathtt{b}$, and to group G_2 when $a_i = \mathtt{c}$. Case 4 occurs to all the groups when $a_i = \mathtt{d}$.

Suppose that we have finished traversing the subtree rooted at u, namely, we have performed the above procedures for all characters a_i with $1 \leq i \leq |\mathsf{children}(k)|$. Then, we output, as the maximal palindromes ending at u, all suffix palindromes of u that did not extend with any a_i. Also, each time we reach a leaf in the traversal, we simply output all suffix palindromes ending at the leaf as the maximal palindromes ending at the leaf.

In each of the above four cases, we can check if the palindromes in a given group extends with a_i by at most two character comparisons. Since there are $O(\log h)$ arithmetic progressions representing the suffix palindromes ending at node u, for each child v_i of u, it takes $O(\log h)$ time to compute the suffix palindromes ending at v_i. The total cost to output the maximal palindromes is less than $2N$ (Lemma 2).

There is one more issue remaining. When only one or two members from a group extend with a_i, then we may need to merge these suffix palindromes into a single arithmetic progression with the suffix palindromes from the previous group. However, this can easily be done in a total of $O(\log h)$ time per node v_i, since the suffix palindromes ending at u was given as $O(\log h)$ arithmetic progressions (groups). See Figure 1 for a concrete example of this merging process. When $a_i = \mathtt{c}$, $\mathtt{c}$ is a suffix palindrome and forms a single arithmetic progression $\langle 1, 0, 1 \rangle$. All the palindromes in G_1 are not extended. The longest suffix palindrome in group G_2 is extended to $\mathtt{caaabaaabaaabaaabaac}$ forming an arithmetic progression $\langle 21, 20, 1 \rangle$, where $20 = |\mathtt{caaabaaabaaabaaabaac}| - |\mathtt{c}|$, but all the other suffix palindromes in group G_2 are not extended. Finally all the suffix palindromes in group G_3 are extended and are represented by an arithmetic progression $\langle 41, 20, 2 \rangle$. Since the three suffix palindromes of lengths 21, 41, and 61 share the common difference 20, the two arithmetic progressions are merged into a single arithmetic progression $\langle 21, 20, 3 \rangle$.

We have shown the following:

Theorem 1. *We can compute all maximal palindromes in a given trie $\mathcal{T}$ in $O(N \log h)$ time and $O(h)$ working space, where N and h respectively denote the number of edges in $\mathcal{T}$ and the height of $\mathcal{T}$.*

Remark 1. Note that for a balanced trie with $h = \Theta(\log_\sigma N)$, our algorithm runs in $O(N \log \log_\sigma N)$ time with $O(\log_\sigma N)$ working space. In the worst case where $h = \Theta(N)$, our algorithm still runs in $O(N \log N)$ time with $O(N)$ space.

4.2 Alternative algorithm based on Manacher's algorithm

In this subsection, we present an alternative algorithm for computing all maximal palindromes in a given trie $\mathcal{T}$ that is based on Manacher's algorithm [26] that is originally designed for computing maximal palindromes in a single string.

For ease of explanation, we consider the path-contracted trie $\mathcal{T}'$ that can be obtained by contracting every unary path of the original trie $\mathcal{T}$ into a single edge that is labeled by a non-empty string. Let r denote the root of $\mathcal{T}'$. Throughout this subsection, for any node u in $\mathcal{T}'$, $\mathsf{parent}(u)$ and $\mathsf{children}(u)$ respectively denote the parent of u and the set of children of u in the path-contracted trie $\mathcal{T}'$.

The basic strategy of our alternative algorithm is as follows. We perform a depth first traversal on $\mathcal{T}'$, where only the root, branching internal nodes, and leaves are explicitly visited. Let u be any branching node visited in the traversal. As was done in the algorithm of Section 4.1, for each branching node v in the path from the root r to u, we maintain the arithmetic progressions representing the suffix palindromes ending at v, which will be used when the traversal traces back to these branching nodes.

Now we are processing node u to extend the suffix palindromes. For this sake, we use the idea of Manacher's algorithm [26]. Let Σ_u be the set of the first characters of the out-edges of u in $\mathcal{T}'$. For each $a \in \Sigma_u$, $e_a = (u, v_a)$ denote the out-edge of u in $\mathcal{T}'$ whose label begins with a. For each $a \in \Sigma_u$ (in any order), we search for the groups of the suffix palindromes of $\mathsf{str}(r, u)$ that are immediately preceded by a, since these will be the only groups that will extend with the edge e_a. Let $\mathbf{P}_a$ be the set of suffix palindromes extended with a (which are represented by $O(\log h)$ arithmetic progressions). For each $1 \leq i \leq |\mathbf{P}_a|$, let P_i denote the ith longest suffix palindrome in $\mathbf{P}_a$. While we move forward on the edge e_a, we keep two invariants ℓ and f such that P_ℓ denotes the longest suffix palindrome whose extension ends with the currently processed character on e_a, and P_f denotes the suffix palindrome whose extension is to be determined by symmetry of P_ℓ. We process the suffix palindromes in $\mathbf{P}_a$ in decreasing order of their lengths, by picking up their lengths from the arithmetic progressions. Namely, we initially set $\ell \leftarrow 1$ and $f \leftarrow 2$ and increase the values of ℓ and f accordingly while reading the characters on the edge e_a. In any following step $\ell \leq f$ will hold.

When $\ell = 1$, as a initial step, we extend the left arm of P_ℓ on the reversed path and the right arm of P_ℓ on the path from u to v_a with naïve character comparisons. Now suppose we are processing P_ℓ. Let $s = |P_\ell|$, c be the center of P_ℓ in the path string from the root, and τ be the length of the extension of P_ℓ, namely, P_ℓ has been extended to a maximal palindrome of length $s + 2\tau$ for center c. This means that the maximal palindromes for any centers less than c in the path from the root to u have already been computed. Then we process P_f. Let $s' = |P_f|$ and c' be the center for P_f. There are three possible cases:

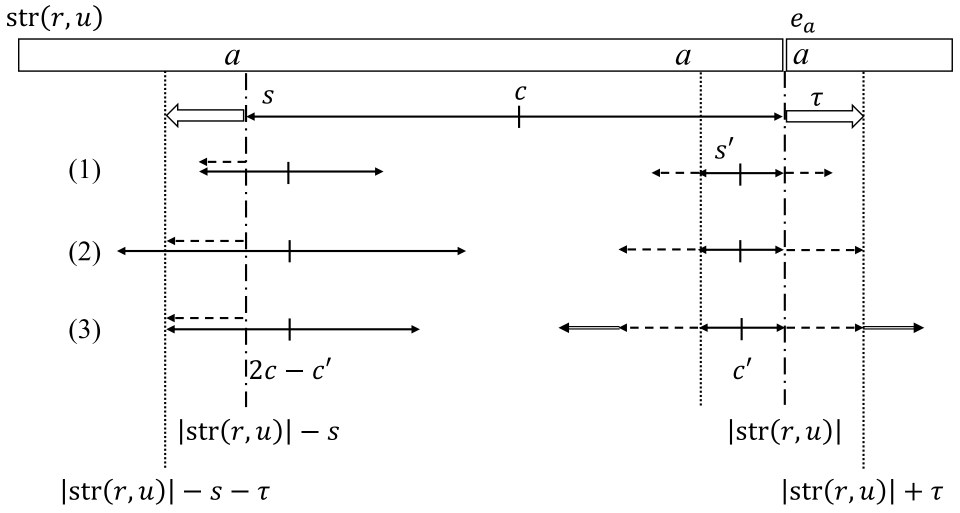


Figure 3. Illustration for our alternative algorithm that computes maximal palindromes in a given trie, that is based on Manacher's algorithm.

(1) The depth of the left-end of the maximal palindrome for center $2c - c'$ in the path from the root is lager than $|\mathsf{str}(r, u)| - s - \tau$.
(2) The depth of the left-end of the maximal palindrome for center $2c - c'$ in the path from the root is less than $|\mathsf{str}(r, u)| - s - \tau$.
(3) The depth of the left-end of the maximal palindrome for center $2c - c'$ is equal to $|\mathsf{str}(r, u)| - s - \tau$.

See Figure 3 for illustration of the above three cases.

In Case(1), by symmetry P_f is extended exactly to the same length as the maximal palindrome for center $2c - c'$. We keep $\ell = 1$ and update $f \leftarrow f + 1$. In Case (2), P_f is extended exactly to length $s' + 2\tau$, because of the mismatching characters $\mathsf{str}(r, u)[|\mathsf{str}(r, u)| - s - \tau]$ and $\mathsf{str}(u, v_a)[\tau + 1]$. We keep $\ell = 1$ and update $f \leftarrow f + 1$. In Case (3), P_f is extended at least to length $s' + 2\tau$. Now we update $\ell \leftarrow f$ and then $f \leftarrow f + 1$. To check if this palindrome is further extended, we perform naïve character comparisons until we find the final value of the extension.

We perform the above procedure until we read all characters on the edge e_a, or we finish extending all palindromes from $\mathbf{P}_a$. This gives us the maximal palindromes whose centers are in the path spelling out $\mathsf{str}(r, u)$. Then we store all these extended maximal palindromes at v_a as $O(\log h)$ arithmetic progressions, and exclude all these maximal palindromes from the set of maximal palindromes ending at u. This ensures that, as in the previous subsection, the number of maximal palindromes stored at the nodes in the current path string is bounded by the height h of the original trie. Note that all maximal palindromes whose centers are on e_a need to be additionally computed. This can be done in linear time in the length of the label of e_a, by running Manacher's algorithm on this edge label.

Suppose that we have performed the above procedures for all out-edges of u in $\mathcal{T}'$. Then, we output, as the maximal palindromes ending at u, all suffix palindromes of u that did not extend with any out-edges. Also, each time we reach a leaf in the traversal, we simply output all suffix palindromes ending at the leaf as the maximal palindromes ending at the leaf.

Let us analyze the complexities of this method. Consider each branching node u in $\mathcal{T}'$. For each $a \in \Sigma_a$, we can find the arithmetic progressions representing $\mathbf{P}_a$ in

$O(\log h)$ time as in the previous subsection. Each character in edge e_a is involved in exactly one character comparison. To perform each character comparison on the trie in $O(1)$ time, we preprocess the original trie $\mathcal{T}$ with N edges in $O(N)$ time and space so that *level ancestor queries* on the trie can be answered in $O(1)$ time each [4]. Hence, if N' is the number of edges in the path-contracted trie $\mathcal{T}'$, then our algorithm of this section runs in $O(N' \log h + N)$ time and $O(N)$ space.

Theorem 2. *We can compute all maximal palindromes in a given trie $\mathcal{T}$ in $O(N' \log h + N)$ time and $O(N)$ working space, where N and h respectively denote the number of edges in $\mathcal{T}$ and the height of $\mathcal{T}$, and N' denotes the number of edges in the path-contracted trie $\mathcal{T}'$.*

Remark 2. Note that $N' \leq N$ always holds, and therefore the algorithm of Theorem 2 is at least as fast as the algorithm of Theorem 1. Moreover, in case where $N' = O(N/\log h)$ (which happens when the average length of the unary paths in $\mathcal{T}$ is $\Omega(\log h)$), then the algorithm of Theorem 2 runs in $O(N)$ time.

5 Computing distinct palindromes in a trie

In this section we present our algorithm that computes all distinct palindromes in a given trie.

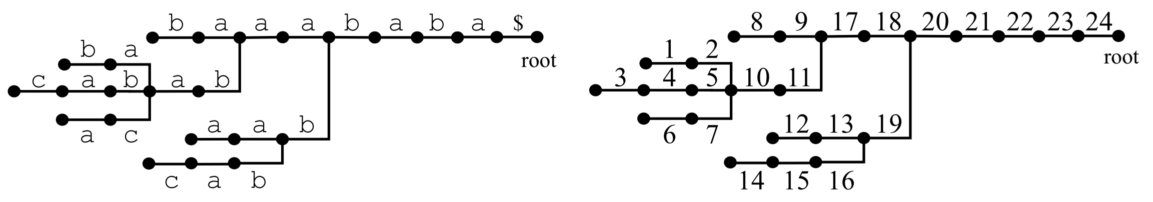
Our algorithm is based on Groult et al.'s [17] that finds distinct palindromes in a single string. Recall the proof of Lemma 3 in Section 3. There we showed that for each node u in a trie $\mathcal{T}$, only the longest suffix palindrome of $\mathsf{str}(r, u)$ can be accounted for as a distinct palindrome, where r is the root of $\mathcal{T}$. Let N and h be the number of edges in $\mathcal{T}$ and the height of $\mathcal{T}$. In this section, we assume that the root has a single out-edge labeled with a special character \$ that does not appear elsewhere in the trie and is lexicographically the smallest.

Lemma 4. *For each node u in a given trie $\mathcal{T}$, we can compute the longest suffix palindrome of $\mathsf{str}(r, u)$ in a total of $O(N' \log h + N)$ time with $O(N)$ working space, where N' denotes the number of edges in the path-contracted trie $\mathcal{T}'$.*

Proof. Clear from our algorithm to compute maximal palindromes in $\mathcal{T}$ which was presented in Section 4. ☐

Now, we consider the *reversed* trie $\mathcal{T}^R$. For any reversed path from u to u' in $\mathcal{T}^R$ in the leaf-to-root direction, let $(u, u') = \mathsf{str}(u', u)^R$. Observe that a suffix of $\mathsf{str}(r, u)$ is a prefix of $\mathsf{rev\_str}(u, r)$. Therefore, a suffix palindrome of $\mathsf{str}(r, u)$ that ends at node u in $\mathcal{T}$ is a prefix palindrome of $\mathsf{rev\_str}(u, r)$ that begins at node u in the reversed trie $\mathcal{T}^R$. For each $1 \leq j \leq N$, let e_j denote the $(N-j+1)$th visited edge in a breadth-first traversal on the original trie $\mathcal{T}$. The *id* of edge e_j is j. See Figure 4 for examples of a reversed trie and the associated integers to its edges.

For each edge id j, let $e_j = (v_j, u_j)$ be the corresponding reversed edge. Let *LPrePal* be an array of length N such that for each $1 \leq j \leq N$ $\mathit{LPrePal}[j]$ stores the length of the longest prefix palindrome in the reversed path string beginning with e_j (namely $\mathsf{rev\_str}(v_j, r)$). Also, let *LFF* be an array of length N called the *longest following factor array*, such that for each $1 \leq i \leq N$ $\mathit{LFF}[j]$ stores the length of the longest prefix of $\mathsf{rev\_str}(v_j, r)$ that occurs as a prefix of $\mathsf{rev\_str}(v_k, r)$ with $k > j$. See Figure 4 for examples of *LPrePal* and *LFF* arrays.



j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
$SA[j]$	24	23	9	2	17	12	21	10	18	4	13	15	6	22	8	1	11	20	5	19	16	7	3	14
$LCP[j]$	-	0	1	2	4	3	1	3	3	5	2	3	1	0	2	3	5	2	4	1	2	0	4	3
$LFF[j]$	0	0	2	4	1	3	1	3	3	5	3	2	1	0	3	5	2	2	4	1	2	3	4	0
$LPrePal[j]$	1	1	3	5	2	2	3	6	5	10	4	5	3	1	5	4	4	3	8	2	3	1	1	1

Figure 4. Upper left: An example of a reversed trie. Upper right: The edge id's based on a breadth-first traversal. Lower: *SA*, *LCP*, *LFF* and *LPrePal* arrays built on the reversed trie shown above.

We design an algorithm that reports a shallowest occurrence of each distinct palindrome in the (reversed) trie. If there are multiple occurrences of the same palindrome beginning at nodes on the same depth, then we report the occurrence that begins with the edge with the largest id. Now we can see that for each j, the occurrence of the longest prefix palindrome of $\mathsf{rev\_str}(v_j, r)$ should be reported iff $LFF[j] < LPrePal[j]$. Hence, we can report all distinct palindromes in the trie in $O(N)$ time by simply scanning the two arrays *LFF* and *LPrePal* from left to right. The *LFF* array can be computed in $O(N)$ time from the *LCP* array for the trie, by using the same technique for the longest previous factor array (LPF array) for a single string [8]. Together with Theorem 2, we obtain the following:

Theorem 3. *We can compute all distinct palindromes in a given trie $\mathcal{T}$ in $O(N' \log h + N)$ time and $O(N)$ working space, where N and h respectively denote the number of edges in $\mathcal{T}$ and the height of $\mathcal{T}$, and N' denotes the number of edges in the path-contracted trie $\mathcal{T}'$.*

Remark 3. The suffix array of the reversed trie with N edges can be constructed in $O(N)$ time and space if the edge labels are drawn from a constant-size alphabet or an integer alphabet of polynomial size in N [36]. In the case of a general ordered alphabet of size σ, the suffix array of the reversed trie can be constructed in $O(N \log \sigma)$ time and space [5]. The other arrays can be constructed in $O(N)$ time after the suffix array has been built. In summary, our algorithm runs in $O(N' \log h + N \log \sigma)$ time and $O(N \log \sigma)$ working space in the case of a general ordered alphabet.

Acknowledgements

This work was supported by JSPS Grant Numbers JP18K18002 (YN), JP17H01697 (SI), JP16H02783 (HB), and JP18H04098 (MT).

References

1. A. Amir and G. Navarro: *Parameterized matching on non-linear structures.* Inf. Process. Lett., 109(15) 2009, pp. 864–867.

2. A. Apostolico, D. Breslauer, and Z. Galil: *Parallel detection of all palindromes in a string.* Theoretical Computer Science, 141 1995, pp. 163–173.
3. M. A. Bender and M. Farach-Colton: *The LCA problem revisited*, in LATIN 2000, 2000, pp. 88–94.
4. M. A. Bender and M. Farach-Colton: *The level ancestor problem simplified.* Theor. Comput. Sci., 321(1) 2004, pp. 5–12.
5. D. Breslauer: *The suffix tree of a tree and minimizing sequential transducers.* Theoretical Computer Science, 191(1–2) January 1998, pp. 131–144.
6. S. Brlek, N. Lafrenière, and X. Provençal: *Palindromic complexity of trees*, in Proc. DLT 2015, 2015, pp. 155–166.
7. M. Bucci, A. D. Luca, A. Glen, and L. Q. Zamboni: *A new characteristic property of rich words.* Theor. Comput. Sci., 410(30-32) 2009, pp. 2860–2863.
8. M. Crochemore and L. Ilie: *Computing longest previous factor in linear time and applications.* Inf. Process. Lett., 106(2) 2008, pp. 75–80.
9. X. Droubay, J. Justin, and G. Pirillo: *Episturmian words and some constructions of de Luca and Rauzy.* Theor. Comput. Sci., 255(1-2) 2001, pp. 539–553.
10. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan: *On the sorting-complexity of suffix tree construction.* J. ACM, 47(6) 2000, pp. 987–1011.
11. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan: *Compressing and indexing labeled trees, with applications.* J. ACM, 57(1) 2009.
12. N. Fujisato, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda: *The parameterized position heap of a trie*, in CIAC 2019, 2019, pp. 237–248.
13. M. Funakoshi, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda: *Longest substring palindrome after edit*, in CPM 2018, 2018, pp. 12:1–12:14.
14. P. Gawrychowski, T. I, S. Inenaga, D. Köppl, and F. Manea: *Tighter bounds and optimal algorithms for all maximal α-gapped repeats and palindromes - finding all maximal α-gapped repeats and palindromes in optimal worst case time on integer alphabets.* Theory Comput. Syst., 62(1) 2018, pp. 162–191.
15. P. Gawrychowski, T. Kociumaka, W. Rytter, and T. Walen: *Tight bound for the number of distinct palindromes in a tree*, in Proc. SPIRE 2015, 2015, pp. 270–276.
16. A. Glen, J. Justin, S. Widmer, and L. Q. Zamboni: *Palindromic richness.* Eur. J. Comb., 30(2) 2009, pp. 510–531.
17. R. Groult, É. Prieur, and G. Richomme: *Counting distinct palindromes in a word in linear time.* Inf. Process. Lett., 110(20) 2010, pp. 908–912.
18. D. Gusfield: *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.
19. D. Harel and R. E. Tarjan: *Fast algorithms for finding nearest common ancestors.* SIAM J. Comput., 13(2) 1984, pp. 338–355.
20. S. Inenaga: *Suffix trees, DAWGs and CDAWGs for forward and backward tries.* CoRR, abs/1904.04513 2019.
21. S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa: *Construction of the CDAWG for a trie*, in Proc. PSC 2001, 2001, pp. 37–48.
22. D. Kimura and H. Kashima: *A linear time subpath kernel for trees*, in IEICE Technical Report, vol. IBISML2011-85, 2011, pp. 291–298.
23. R. Kolpakov and G. Kucherov: *Searching for gapped palindromes.* Theor. Comput. Sci., 410(51) 2009, pp. 5365–5373.
24. S. R. Kosaraju: *Efficient tree pattern matching (preliminary version)*, in Proc. FOCS 1989, 1989, pp. 178–183.
25. D. Kosolobov, M. Rubinchik, and A. M. Shur: *Finding distinct subpalindromes online*, in PSC 2013, 2013, pp. 63–69.
26. G. Manacher: *A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string.* Journal of the ACM, 22 1975, pp. 346–351.
27. W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto: *Efficient algorithms to compute compressed longest common substrings and compressed palindromes.* Theor. Comput. Sci., 410(8–10) 2009, pp. 900–913.
28. W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto: *Efficient algorithms to compute compressed longest common substrings and compressed palindromes.* Theor. Comput. Sci., 410(8-10) 2009, pp. 900–913.

29. M. Mohri, P. J. Moreno, and E. Weinstein: *General suffix automaton construction algorithm and space bounds.* Theor. Comput. Sci., 410(37) 2009, pp. 3553–3562.
30. T. Nakamura, S. Inenaga, H. Bannai, and M. Takeda: *Order preserving pattern matching on trees and dags*, in Proc. SPIRE 2017, 2017, pp. 271–277.
31. Y. Nakashima, T. I, S. Inenaga, H. Bannai, and M. Takeda: *Constructing LZ78 tries and position heaps in linear time for large alphabets.* Inf. Process. Lett., 115(9) 2015, pp. 655–659.
32. S. Narisada, Diptarama, K. Narisawa, S. Inenaga, and A. Shinohara: *Computing longest single-arm-gapped palindromes in a string*, in SOFSEM 2017, 2017, pp. 375–386.
33. A. H. L. Porto and V. C. Barbosa: *Finding approximate palindromes in strings.* Pattern Recognition, 35 2002, pp. 2581–2591.
34. B. Schieber and U. Vishkin: *On finding lowest common ancestors: Simplification and parallelization.* SIAM J. Comput., 17(6) 1988, pp. 1253–1262.
35. T. Shibuya: *Constructing the suffix tree of a tree with a large alphabet.* IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, E86-A(5) 2003, pp. 1061–1066.
36. T. Shibuya: *Constructing the suffix tree of a tree with a large alphabet.* IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, E86-A(5) 2003, pp. 1061–1066.
37. R. Sugahara, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda: *Computing runs on a trie*, in Proc. CPM 2019, 2019, pp. 23:1–23:11.
38. P. Weiner: *Linear pattern matching algorithms*, in 14th Annual Symposium on Switching and Automata Theory, 1973, pp. 1–11.

Algorithms to Compute the Lyndon Array Revisited

Frantisek Franek and Michael Liut

Department of Computing and Software
McMaster University, Hamilton, Canada
{franek/liutm}@mcmaster.ca

Abstract. The motivation for having an efficient algorithm for identifying all maximal Lyndon substrings of a string comes from the work of Bannai et al. on the Runs Conjecture. In 2015, they resolved the conjecture by considering Lyndon roots of runs and they also presented a unique linear algorithm for computing all runs. The uniqueness of the algorithm lies in the fact that it relies on the knowledge of all maximal Lyndon substrings, while all other linear algorithms for runs rely on Lempel-Ziv factorization of the input string. A Lyndon array is a data structure that encodes the information of all maximal Lyndon substrings of a string. In a 2016 Prague Stringology Conference paper, Franek et al. discussed various known algorithms for computing the Lyndon array. In 2015, in his Masters' thesis, Baier designed a linear algorithm for suffix sorting. Inspired by Phase II of Baier's algorithm, in a 2017 Prague Stringology Conference paper, Franek et al. discussed the linear co-equivalency of sorting suffixes and sorting maximal Lyndon substrings. As noticed by C. Diegelmann, the first phase of Baier's algorithm identifies and sorts all maximal Lyndon substrings of the input string. Based on Phase I of Baier's algorithm, in a 2018 Prague Stringology Conference paper, Franek et. al presented an elementary (in the sense of not relying on a pre-processed global data structure) linear algorithm for identifying and sorting all maximal Lyndon substrings. This paper revisits the subject of algorithms for the Lyndon array and closes off the series of our Prague Stringology Conference contributions on the topic – it provides a simple overview of all currently known algorithms including the new development since 2016. In particular, it presents a detailed analysis of a new algorithm *TRLA*, and comparative measurements of three of the algorithms.

Keywords: string, suffix, suffix array, Lyndon array, Lyndon string, maximal Lyndon substring

1 Introduction

There are at least two reasons for having an efficient algorithm for identifying all maximal Lyndon substrings in a string: firstly, Bannai et al. introduced in 2015 (arXiv, [3]), and published in 2017, [4], a linear algorithm to compute all runs in a string that relies on knowing all maximal Lyndon substrings of the string, and secondly, in 2017, Franek et al. in [13] showed a linear co-equivalence of sorting suffixes and sorting maximal Lyndon substrings, based on a novel suffix sorting algorithm introduced by Baier in 2015 (Master's thesis, [1]), and published in 2016, [2].

The most significant feature of the runs algorithm presented in [4] is that it relies on knowing all maximal Lyndon substrings of the input string for some order of the alphabet and for the inverse of that order, while all other linear algorithms for runs rely on Lempel-Ziv factorization of the input string. Thus, computing runs became yet another application of Lyndon words. It also raised the issue which approach may be more efficient: to compute the Lempel-Ziv factorization or to compute all maximal Lyndon substrings. Interestingly, Kosolobov argues that computing Lempel-Ziv factorization may be harder than computing all runs, however his archive paper [17] was not followed up with. There are several efficient linear algorithms for Lempel-Ziv factorization; for example see [5,7] and the references therein.

Baier introduced in [1], and published in [2], a new algorithm for suffix sorting. Though Lyndon strings are never mentioned there, it was noticed by Cristoph Diegelmann in a personal communication, [8], that Phase I of Baier's suffix sort identifies and sorts all maximal Lyndon substrings.

The maximal Lyndon substrings of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ can be best encoded in the so-called ***Lyndon array*** introduced in [14]: an integer array $\mathcal{L}[1..n]$ so that for any $i \in 1..n$, $\mathcal{L}[i] =$ *the length of the maximal Lyndon substring starting at position i.*

Our research group has presented in the Prague Stringology Conference a series of three papers on the topic of maximal Lyndon substrings:

(1) In 2016, [14] presented an overview of then-current algorithms for computing the Lyndon array.
(2) In 2017, [13] presented the linear co-equivalency of sorting suffixes and sorting maximal Lyndon substrings.
(3) In 2018, [12] presented an elementary[1] linear algorithm to identify and sort all maximal Lyndon substrings, inspired by Phase I of Baier's algorithm.

This paper completes the series and briefly recapitulates the algorithms presented in [14] and then presents the development since 2016 not described in [14], in particular a novel algorithm *TRLA* for computing the Lyndon array based on τ-reduction, and empirical comparisons of three of the algorithms: *IDLA*, *BSLA*, and *TRLA*.

The structure of the paper is as follows. In Section 2, the basic notations and notions are presented. Section 3 contains a brief recapitulation of *IDLA*, the *Iterated Duval algorithm for Lyndon array.* Section 4 contains a brief recapitulation of *RDLA*, the *Recursive Duval algorithm for Lyndon array.* Section 5 contains a brief recapitulation of *SSLA*, the *algorithmic scheme of suffix sorting followed by Next Smaller Value.* Section 6 introduces *BWLA*, the 2018 *algorithmic scheme for computing the Lyndon array via inversion of Burrows-Wheeler transform.* Section 7 contains a brief recapitulation of *RGLA*, the *ranges based algorithm for Lyndon array.* Section 8 contains a brief recapitulation of *BSLA*, the *Baier's sort Phase I inspired algorithm.* Section 9 contains a detailed description and analysis of *TRLA*, the recursive algorithm based on τ-reduction. In Section 10, the empirical measurements of the performance of *IDLA*, *BSLA*, and *TRLA* are presented on various datasets with random strings of various lengths and over various alphabets. The results are presented in a graphical form. In Section 11, the conclusion of the research is presented, and the necessary future work described.

2 Basic notation and terminology

For two integers $i \leq j$, the ***range*** $\boldsymbol{i..j} = \{k \text{ } integer : i \leq k \leq j\}$. An ***alphabet*** is a finite or infinite set of ***symbols*** (equivalently called letters). We assume

[1] not needing a pre-processed global structure such as suffix array

that a sentinel symbol \$ is not in the alphabet and is always assumed to be lexicographically the smallest. A ***string*** over an alphabet $\mathcal{A}$ is a finite sequence of symbols from $\mathcal{A}$. A ***\$-terminated string*** over $\mathcal{A}$ is a string over $\mathcal{A}$ terminated by \$. We use the array notation indexing from 1 for strings, thus $\boldsymbol{x}[1..n]$ indicates a string of length n, the first symbol is the symbol with index 1, i.e. $\boldsymbol{x}[1]$, the second symbol is the symbol with index 2, i.e. $\boldsymbol{x}[2]$, etc. Thus, $\boldsymbol{x}[1..n] = \boldsymbol{x}[1]\boldsymbol{x}[2]\cdots\boldsymbol{x}[n]$. For a \$-terminated string $\boldsymbol{x}$ of length n, $\boldsymbol{x}[n+1] = \$$. The ***alphabet of string*** $\boldsymbol{x}$, denoted as $\mathcal{A}_{\boldsymbol{x}}$, is the set of all distinct alphabet symbols occurring in $\boldsymbol{x}$. By a ***constant alphabet*** we mean a fixed finite alphabet. A string $\boldsymbol{x}$ is over an ***integer alphabet*** if $\mathcal{A}_{\boldsymbol{x}} \subseteq \{0, 1, \ldots, |\boldsymbol{x}|\}$. Thus, the class of ***strings over integer alphabets*** $= \{\boldsymbol{x} \mid \boldsymbol{x}\ \textit{is a string over}\ \{0, 1, \ldots, |\boldsymbol{x}|\}\}$. A string $\boldsymbol{x}$ over an integer alphabet is ***tight*** if $\mathcal{A}_{\boldsymbol{x}} = \{0, 1, \ldots, k\}$ for some $k \leq |\boldsymbol{x}|$. Thus, for instance $\boldsymbol{x} = 010$ is tight as $\mathcal{A}_{\boldsymbol{x}} = \{0, 1\}$, while $\boldsymbol{y} = 020$ is not as $\mathcal{A}_{\boldsymbol{y}} = \{0, 2\}$ – i.e. 1 is missing from $\mathcal{A}_{\boldsymbol{y}}$.

We use a bold font to denote strings, thus $\boldsymbol{x}$ denotes a string, while x denotes some other mathematical entity such as an integer. The ***empty string*** is denoted by $\boldsymbol{\varepsilon}$ and has length 0. The ***length*** or ***size*** of string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is n. The length of a string $\boldsymbol{x}$ is denoted by $|\boldsymbol{x}|$. For two strings $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and $\boldsymbol{y} = \boldsymbol{y}[1..m]$, the ***concatenation xy*** is a string $\boldsymbol{u}$ where $\boldsymbol{u}[i] = \begin{cases} \boldsymbol{x}[i]\ \textit{for}\ i \leq n, \\ \boldsymbol{y}[i-n]\ \textit{for}\ n < i \leq n+m. \end{cases}$
If $\boldsymbol{x} = \boldsymbol{uvw}$, then $\boldsymbol{u}$ is a ***prefix***, $\boldsymbol{v}$ a ***substring***, and $\boldsymbol{w}$ a ***suffix*** of $\boldsymbol{x}$. If $\boldsymbol{u}$ (respectively $\boldsymbol{v}$, $\boldsymbol{w}$) is empty, then it is called a ***trivial prefix*** (respectivly ***trivial substring***, ***trivial suffix***), if $|\boldsymbol{u}| < |\boldsymbol{x}|$ (respectively $|\boldsymbol{v}| < |\boldsymbol{x}|$, $|\boldsymbol{w}| < |\boldsymbol{x}|$) then it is called a ***proper prefix*** (respectively ***proper substring***, ***proper suffix***). If $\boldsymbol{x} = \boldsymbol{uv}$, then $\boldsymbol{vu}$ is called a ***rotation*** or a ***conjugate*** of $\boldsymbol{x}$; if either $\boldsymbol{u} = \boldsymbol{\varepsilon}$ or $\boldsymbol{v} = \boldsymbol{\varepsilon}$, then the rotation is called ***trivial***. A non-empty string $\boldsymbol{x}$ is ***primitive*** if there is no string $\boldsymbol{y}$ and no integer $k \geq 2$ so that $\boldsymbol{x} = \boldsymbol{y}^k = \underbrace{\boldsymbol{yy}\cdots\boldsymbol{y}}_{k\ times}$.

A non-empty string $\boldsymbol{x}$ has a non-trivial border $\boldsymbol{u}$ if $\boldsymbol{u}$ is both a non-trivial proper prefix and a non-trivial proper suffix of $\boldsymbol{x}$. Thus, both $\boldsymbol{\varepsilon}$ and $\boldsymbol{x}$ are trivial borders of $\boldsymbol{x}$. A string without a non-trivial border is called ***unbordered***.

Let $\prec$ be a total order of an alphabet $\mathcal{A}$. The order is extended to all finite strings over the alphabet $\mathcal{A}$: for $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and $\boldsymbol{y} = \boldsymbol{y}[1..n]$, $\boldsymbol{x} \prec \boldsymbol{y}$ if either $\boldsymbol{x}$ is a proper prefix of $\boldsymbol{y}$, or there is a $j \leq \min\{n, m\}$ so that $\boldsymbol{x}[1] = \boldsymbol{y}[1]$, ..., $\boldsymbol{x}[j-1] = \boldsymbol{y}[j-1]$ and $\boldsymbol{x}[j] \prec \boldsymbol{y}[j]$. This total order induced by the order of the alphabet is called the ***lexicographic*** order of all non-empty strings over $\mathcal{A}$. We write $\boldsymbol{x} \preceq \boldsymbol{y}$ if either $\boldsymbol{x} \prec \boldsymbol{y}$ or $\boldsymbol{x} = \boldsymbol{y}$. A string $\boldsymbol{x}$ over $\mathcal{A}$ is ***Lyndon*** for a given order $\prec$ of $\mathcal{A}$ if $\boldsymbol{x}$ is strictly lexicographically smaller than any non-trivial rotation of $\boldsymbol{x}$. A substring $\boldsymbol{x}[i..j]$ of $\boldsymbol{x}[1..n]$, $1 \leq i \leq j \leq n$ is a ***maximal Lyndon substring of x*** if it is Lyndon and either $j = n$ or for any $k > j$, $\boldsymbol{x}[i..k]$ is not Lyndon. The ***Lyndon array*** of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is an integer array $\mathcal{L}[1..n]$ so that $\mathcal{L}[i] = j$ where $j \leq n-i$ is a maximal integer such that $\boldsymbol{x}[i..i+j-1]$ is Lyndon. Alternatively, we can define it as an integer array $\mathcal{L}'[1..n]$ so that $\mathcal{L}'[i] = j$ when $\boldsymbol{x}[i..j]$ is a maximal Lyndon substring. The relationship between those two definitions is straightforward: $\mathcal{L}'[i] = \mathcal{L}[i]+i-1$, or $\mathcal{L}[i] = \mathcal{L}'[i]-i+1$.

3 Iterated Duval algorithm - *IDLA*

This algorithm was presented in [14]. The algorithm is based on Duval's work on Lyndon factorization, [9]. The procedure `maxLyn(`$\boldsymbol{x}$`)` returns the length of the maximal Lyndon prefix of the string $\boldsymbol{x}$. In Duval's factorization algorithm, `maxLyn` is then applied to the position immediately after the maximal Lyndon prefix. Here, we apply `maxLyn` iteratively to every suffix of $\boldsymbol{x}$. Why and how `maxLyn` works, and its complexity can be found in [14], including the pseudo-code. The `C++` implementation can be found in the file `lynarr.hpp`, [6]. Note that *IDLA* is referred to as *Iterated MaxLyn* in [14]. The worst-case complexity of $IDLA(\boldsymbol{x})$ is $\mathcal{O}(|\boldsymbol{x}|^2)$. The algorithm works in-place, so the storage requirements are just the storage for the string and the storage for the Lyndon array. The alphabet of the input string need not be sorted, but must be ordered. The alphabet is *sorted* if the alphabet is in the form of an ordered list, so that for each letter you can access in constant time the immediately preceding letter and the immediately succeeding letter. The alphabet is *ordered* if there is a partial order on the alphabet so that comparison of any two letters can be computed in constant time.

4 Recursive Duval algorithm - *RDLA*

This algorithm was presented in [14]. The algorithm is also based on Duval's algorithm for Lyndon factorization which is applied recursively: if $\boldsymbol{x}[1..i_1]\boldsymbol{x}[i_1{+}1..i_2]\cdots\boldsymbol{x}[i_k{+}1..n]$ is a Lyndon factorization of $\boldsymbol{x}$, the algorithm is recursively applied to $\boldsymbol{x}[2..i_1]$, to $\boldsymbol{x}[i_1{+}2..i_2]$, ..., to $\boldsymbol{x}[i_k{+}2..n]$, and so on. The correctness of the algorithm follows from the correctness of Duval's algorithm. The alphabet of the input string need not be sorted, but must be ordered. The algorithm works in the worst-case complexity of $\mathcal{O}(|\boldsymbol{x}|^2)$, and in the special case of the binary alphabet of $\boldsymbol{x}$, it is $\mathcal{O}(|\boldsymbol{x}|\ log(|\boldsymbol{x}|))$, see [14]. Storage requirements are the same as for *IDLA*, plus the additional storage for the stack controlling the recursion.

5 Algorithmic scheme based on suffix sorting - *SSLA*

This scheme was presented in [14]. It is based on Lemma 1 which follows from Hohlweg and Reutenauer's work [14,15]. The lemma characterizes maximal Lyndon substrings in terms of the relationships of the suffixes.

Lemma 1. *Consider a string $\boldsymbol{x}[1..n]$ over an alphabet ordered by $\prec$. The substring $\boldsymbol{x}[i..j]$ is Lyndon if $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[k..n]$ for any $i < k \leq j$, and is maximal Lyndon if it is Lyndon and either $j = n$ or $\boldsymbol{x}[j{+}1..n] \prec \boldsymbol{x}[i..n]$.*

Therefore, the Lyndon array of $\boldsymbol{x}$ is the *NSV* (Next Smaller Value) array of the inverse suffix array. The scheme is as follows: sort the suffixes, from the resulting suffix array compute the inverse suffix array, and then apply *NSV* to the inverse suffix array. Computing the inverse suffix array and applying *NSV* are "naturally" linear and computing the suffix array can be implemented to be linear, see [14,20] and the references therein. The time and space characteristics of the whole scheme are dominated by the time and space characteristics of the first step – the computation of the suffix array. For linear suffix sorting, the input strings must be over constant or integer alphabets.

6 Algorithmic scheme based on Burrows-Wheeler transform – *BWLA*

This scheme was not presented in [14] as it was introduced in 2018, see [18]. The algorithm is linear and computes the Lyndon array from a given Burrows-Wheeler transform of the input string. Since the Burrows-Wheeler transform is computed in linear time from the suffix array, it is yet another scheme of how to obtain the Lyndon array via suffix sorting: compute the suffix array, from the suffix array compute the Burrows-Wheeler transform, and then compute the Lyndon array during the inversion of the Burrows-Wheeler transform. As for *SSLA*, the execution and space characteristics of the scheme are dominated by the computation of the suffix array.

7 An algorithm based on ranges – *RGLA*

This algorithm was discussed in [14] where it was referred to as *NSV**.
In case of a constant alphabet, ranges can be compared in constant time if the Parikh vector for each range is pre-computed, which can be done in linear time. An increasing range is a maximal substring $\boldsymbol{x}[i..j]$ so that $\boldsymbol{x}[k] \preceq \boldsymbol{x}[\ell]$ for every $i \leq k < \ell \leq j$, while a decreasing range is a maximal substring $\boldsymbol{x}[i..j]$ so that $\boldsymbol{x}[k] \succ \boldsymbol{x}[\ell]$ for every $i \leq k < \ell \leq j$. The algorithm emulates the classic stack implementation of *NSV*. The time and space complexity of the algorithm was not given in [14], but the time complexity is at worst $\mathcal{O}(n^2)$, though it was indicated in the paper it could possibly be $\mathcal{O}(n\ log(n))$, where n is the length of the input string. An informal analysis of the correctness of the algorithm was provided.

8 Baier's suffix sort Phase I inspired algorithm – *BSLA*

Introduced in 2018, this algorithm was not discussed in [14], however, it was presented in [12]. There it was referred to as *Baier's sort* and the code was available as `bls`, see [6]. For the interested reader, a simpler and more elegant description and analysis of the correctness of the algorithm can be found in [11]. Our `C++` implementation can be found at [6] as `BSLA`, though based on the ideas of Phase I of Baier's suffix sort, our implementation necessarily differs from Baier's.

The input strings for *BSLA* are tight strings over integer alphabets. Note that this requirement does not significantly detract from the applicability of the algorithm as any string over an integer alphabet can easily be transformed in $\mathcal{O}(|\boldsymbol{x}|)$ time to a tight string so that the original string and the transformed string have the same Lyndon array. Thus, computing the Lyndon array for the transformed tight string also gives the Lyndon array for the original string.

The algorithm is based on a refinement of a list of groups of indices of the input string $\boldsymbol{x}$. The refinement is driven by a group that is already complete and the refinement process makes the immediately preceding group complete, too. In turn, this newly completed group is used as the driver of the next round of the refinement. In this fashion, the refinement proceeds from right to left until all the groups in the list are complete. The initial list of groups consists of the groups of indices with the same alphabet symbol; it will be shown that the group for the largest alphabet symbol is complete, so it is a proper start for the refinement process.

Each group is assigned a specific substring of the input string referred to as the ***context*** of the group; it has the property that for every i in the group, the group's context occurs at the position i. Throughout the process, the list of the groups is maintained in an increasing lexicographic order by their contexts. Moreover, at every stage, the contexts of all the groups are Lyndon substrings of $\boldsymbol{x}$ with the additional property that the contexts of complete groups are maximal occurrences in $\boldsymbol{x}$. Hence, when the refinement is complete, the contexts of all the groups in the list represent all maximal Lyndon substrings of $\boldsymbol{x}$.

The process of refinement is rather technical and we refer the interested reader to the original presentation in [12], or a better presentation in [11]. The complexity of the algorithm is linear in the length of the input string. The space requirements are relatively high; our `C++` implementation, see [6], uses $12n$ integers of working memory. We refer to the algorithm as *elementary*, as no global data structure needs to be pre-processed, as is the case for *SSLA* and *BWLA*.

9 τ-reduction algorithm – *TRLA*

Introduced in 2017, this algorithm was not presented in [14]. The first idea of the algorithm was proposed in Paracha's 2017 Ph.D. thesis [21]. It follows Farach's approach used in his remarkable linear algorithm for suffix tree construction [10], and reproduced very successfully in all linear algorithms for suffix sorting, see for instance [19,20] and the references therein. The scheme for computing the Lyndon array works as follows:

(1) reduce the input string $\boldsymbol{x}$ to $\boldsymbol{y}$,
(2) by recursion compute the Lyndon array of $\boldsymbol{y}$,
(3) from the Lyndon array of $\boldsymbol{y}$ compute the Lyndon array of $\boldsymbol{x}$.

The input strings are \$-terminated strings over integer alphabets. The reduction computed in (1) is important. All linear algorithms for suffix array computations use the proximity property of suffixes: comparing $\boldsymbol{x}[i..n]$ and $\boldsymbol{x}[j..n]$ can be done by comparing $\boldsymbol{x}[i]$ and $\boldsymbol{x}[j]$, and if they are the same, comparing $\boldsymbol{x}[i+1..n]$ with $\boldsymbol{x}[j+1..n]$. For instance, in the first linear algorithm for suffix array by Kärkkäinen and Sanders, [16], obtaining the sorted suffixes for positions $i \equiv 0$ $(mod\ 3)$ and $i \equiv 1$ $(mod\ 3)$ via the recursive call is sufficient to determine the order of suffixes for $i \equiv 2$ $(mod\ 3)$ positions, and then to merge both lists together. However, there is no such proximity property for maximal Lyndon substrings, so the reduction itself must have a property that helps determine some of the values of the Lyndon array of $\boldsymbol{x}$ from the Lyndon array of $\boldsymbol{y}$ and compute the rest. We present such a reduction that we call τ-reduction, and it may be of some general interest as it preserves order of some suffixes and hence, by Lemma 1, some maximal Lyndon substrings.

The algorithm computes $\boldsymbol{y}$ as a τ-reduction of $\boldsymbol{x}$ in step (1) in linear time and in step (3) it expands the Lyndon array of the reduced string computed by step (2) to an incomplete Lyndon array of the original string also in linear time. However, it computes the missing values of the incomplete Lyndon array in $\Theta(n\ log(n))$ time resulting in the overall worst-case complexity of $\Theta(n\ log(n))$. If the missing values of the incomplete Lyndon array of $\boldsymbol{x}$ were computed in linear time, the overall algorithm would be linear as well. Since for τ-reduction, the size of $\tau(\boldsymbol{x})$ is at most $\frac{2}{3}|\boldsymbol{x}|$, we eventually obtain, through the recursion of step (2) applied to $\tau(\boldsymbol{x})$, a partially filled Lyndon array of the input string; the array is about $\frac{1}{2}$ to $\frac{2}{3}$ full and for every position

i with an unknown value, the values at positions $i-1$ and $i+1$ are known and $\boldsymbol{x}[i-1] \preceq \boldsymbol{x}[i]$. In particular, the value at position 1 and position n are both known. So, a lot of information is provided by the recursive step. For instance, given the string 00011001, via the recursive call we would identify the maximal Lyndon substrings that are underlined in $\underline{00\underline{01\underline{1}}\,\underline{00\underline{1}}}$ and would need to compute the missing maximal Lyndon substrings that are underlined in $0\underline{00\underline{1}1}0\underline{01}$. It is possible that in the future we may come up with a linear procedure to compute the missing values making the whole algorithm linear. We describe the τ-reduction in several steps: first the τ-pairing, then choosing the τ-alphabet, and finally the computation of the τ-reduction of $\boldsymbol{x}$.

9.1 $\boldsymbol{\tau}$-pairing

Consider a \$-terminated string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ whose alphabet $\mathcal{A}_{\boldsymbol{x}}$ is ordered by $\prec$ with $\boldsymbol{x}[n+1] = \$$ and $\$ \prec a$ for any $a \in \mathcal{A}_{\boldsymbol{x}}$. A ***τ-pair*** consists of a pair of adjacent positions from the range $1..n+1$. The τ-pairs are computed by induction:

- the initial τ-pair is $(1,2)$;
- if $(i-1,i)$ is the last τ-pair computed, then:

```
if $i = n-1$ then
   the next $\tau$-pair is set to $(n, n+1)$
   stop
elseif $i \geq n$ then
   stop
elseif $\boldsymbol{x}[i-1] \succ \boldsymbol{x}[i]$ and $\boldsymbol{x}[i] \preceq \boldsymbol{x}[i+1]$ then
   the next $\tau$-pair is set to $(i, i+1)$
else
   the next $\tau$-pair is set to $(i+1, i+2)$
```

Every position of the input string that occurs in some τ-pair as the first element is labeled ***black***, all others are labeled ***white***. Note that most of the τ-pairs do not overlap; if two τ-pairs overlap, they overlap in a position i such that $1 < i < n$ and $\boldsymbol{x}[i-1] \succ \boldsymbol{x}[i]$ and $\boldsymbol{x}[i] \preceq \boldsymbol{x}[i+1]$. Moreover, a τ-pair can be involved in at most one overlap; for illustration see Fig. 1, for formal proof see Lemma 2.

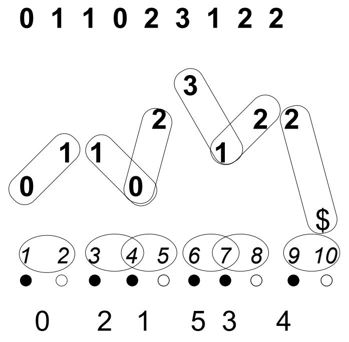


Figure 1. Illustration of τ-reduction of a string **011023122**
The rounded rectangles indicate symbol τ-pairs, the ovals indicate the τ-pairs below are the colour labels of positions, at the bottom is the τ-reduction

Lemma 2. *Let* $(i_1, i_1{+}1)\cdots(i_k, i_k{+}1)$ *be the* τ*-pairs of a strings* $\boldsymbol{x} = \boldsymbol{x}[1..n]$*. Then for any* $j, \ell \in 1..k$

(1) *if* $|(i_j, i_j{+}1) \cap (i_\ell, i_\ell{+}1)| = 1$*, then for any* $m \neq j, \ell$, $|(i_j, i_j{+}1) \cap (i_m, i_m{+}1)| = 0$*,*
(2) $|(i_j, i_j{+}1) \cap (i_\ell, i_\ell{+}1)| \leq 1$*.*

Proof. For the proof, please see the online report [11], Observation 3 and Lemma 3. □

9.2 τ-reduction

For each τ-pair $(i, i{+}1)$, we consider the pair of alphabet symbols $(\boldsymbol{x}[i], \boldsymbol{x}[i{+}1])$. We call them ***symbol τ-pairs***. They are in a total order $\lhd$ induced by $\prec$: $(\boldsymbol{x}[i_j], \boldsymbol{x}[i_j{+}1]) \lhd (\boldsymbol{x}[i_\ell], \boldsymbol{x}[i_\ell{+}1])$ if either $\boldsymbol{x}[i_j] \prec \boldsymbol{x}[i_\ell]$, or $\boldsymbol{x}[i_j] = \boldsymbol{x}[i_\ell]$ and $\boldsymbol{x}[i_j{+}1] \prec \boldsymbol{x}[i_\ell{+}1]$. They are sorted using radix sort, with keys of size 2, and assigned letters from a chosen τ-alphabet that is a subset of $\{0, 1, \ldots, |\tau(\boldsymbol{x})|\}$ so that the assignment preserves the order. Because the input string was over an integer alphabet, the radix sort is linear.

In the example, Fig. 1, the τ-pairs are $(1,2)(3,4)(4,5)(6,7)(7,8)(9,10)$ and so the symbol τ-pairs are $(0,1)(1,0)(0,2)(3,1)(1,2)(2,\$)$. The sorted symbol τ-pairs are $(0,1)(0,2)(1,0)(1,2)(2,\$)(3,2)$. Thus we chose as our τ-alphabet $\{0,1,2,3,4,5\}$ and so the symbol τ-pairs are assigned these letters: $(0,1) \to 0$, $(0,2) \to 1$, $(1,0) \to 2$, $(1,2) \to 3$, $(2,\$) \to 4$ and $(3,1) \to 5$. Note that the assignments respect the order $\lhd$ of the symbols τ-pairs, and the natural order $<$ of $\{0,1,2,3,4,5\}$.

The τ-letters are substituted for the symbol τ-pairs and the resulting string is terminated with \$. This string is called the ***τ-reduction*** of $\boldsymbol{x}$ and denoted $\boldsymbol{\tau(x)}$, and it is a \$-terminated string over an integer alphabet. For our running example from Fig. 1, $\tau(\boldsymbol{x}) = 021534$. The next lemma justifies calling the above transformation a reduction.

Lemma 3. *For any string* $\boldsymbol{x}$*,* $\frac{1}{2}|\boldsymbol{x}| \leq |\tau(\boldsymbol{x})| \leq \frac{2}{3}|\boldsymbol{x}|$*.*

Proof. One extreme case is when all the τ-pairs do not overlap at all, then $|\tau(\boldsymbol{x})| = \frac{1}{2}|\boldsymbol{x}|$. The other extreme case is when all the τ-pairs overlap, then $|\tau(\boldsymbol{x})| = \frac{2}{3}|\boldsymbol{x}|$. Any other case must be in between.

□

Let $\mathcal{B}(\boldsymbol{x})$ denote the set of all black positions of $\boldsymbol{x}$. For any $i \in 1..|\tau(\boldsymbol{x})|$, $b(i) = j$ where j is a black position in $\boldsymbol{x}$ of the τ-pair corresponding to the new symbol in $\tau(\boldsymbol{x})$ at position i, while $t(j)$ assigns each black position of $\boldsymbol{x}$ the position in $\tau(\boldsymbol{x})$ where the corresponding new symbol is, i.e. $b(t(j)) = j$ and $t(b(i)) = i$. Thus,

$$1..|\tau(\boldsymbol{x})| \overset{\mathrm{b}}{\underset{\mathrm{t}}{\rightleftarrows}} \mathcal{B}(\boldsymbol{x})$$

In addition, we define p as the mapping of the τ-pairs to the τ-alphabet.

In our running example from Fig. 1, $t(1) = 1$, $t(3) = 2$, $t(4) = 3$, $t(6) = 4$, $t(7) = 5$, and $t(9) = 6$, while $b(1) = 1$, $b(2) = 3$, $b(3) = 4$, $b(4) = 6$, $b(5) = 7$, and $b(6) = 9$. For the letter mapping, we get $p(1,2) = 0$, $p(3,4) = 2$, $p(4,5) = 1$, $p(6,7) = 5$, $p(7,8) = 3$, and $p(9,10) = 4$.

9.3 Properties preserved by τ-reduction

The most important property of τ-reduction is the preservation of maximal Lyndon substrings of $\boldsymbol{x}$ that start at black positions. By that we mean the fact there is a closed formula that gives for every maximal Lyndon substring of $\tau(\boldsymbol{x})$ a corresponding maximal Lyndon substring of $\boldsymbol{x}$. Moreover, the formula for any black position can be computed in constant time. It is simpler to present the following results using $\mathcal{L}'$, the alternative form of Lyndon array, the one where the end positions of maximal Lyndon substrings are stored rather than their lengths. More formally:

Theorem 4. *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$, let $\mathcal{L}'_{\tau(\boldsymbol{x})}[1..m]$ be the Lyndon array of $\tau(\boldsymbol{x})$, and let $\mathcal{L}'_{\boldsymbol{x}}[1..n]$ be the Lyndon array of $\boldsymbol{x}$.*
Then for any black $i \in 1..n$, $\mathcal{L}'_{\boldsymbol{x}}[i] = \begin{cases} b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]\big) & \text{if } \boldsymbol{x}[b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]\big)+1] \preceq \boldsymbol{x}[i] \\ b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]\big)+1 & \text{otherwise.} \end{cases}$

The proof of the theorem requires a series of lemmas that are presented below. First we show that τ-reduction preserves relationships of certain suffixes of $\boldsymbol{x}$.

Lemma 5. *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and let $\tau(\boldsymbol{x}) = \tau(\boldsymbol{x})[1..m]$. Let $1 \le i, j \le n$. If i and j are both black positions, then $\boldsymbol{x}[i..n] \prec \boldsymbol{x}[j..n]$ implies $\tau(\boldsymbol{x})[t(i)..m] \prec \tau(\boldsymbol{x})[t(j)..m]$.*

Proof. For the proof, please see the online report [11], Lemma 6. □

Lemma 6 shows that τ-reduction preserves the Lyndon property of certain Lyndon substrings.

Lemma 6. *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and let $\tau(\boldsymbol{x}) = \tau(\boldsymbol{x})[1..m]$. Let $1 \le i < j \le n$. Let $\boldsymbol{x}[i..j]$ be a Lyndon susbtsring of $\boldsymbol{x}$, and let i be a black position.*
Then $\begin{cases} \tau(\boldsymbol{x})[t(i)..t(j)] \text{ is Lyndon} & \text{if } j \text{ is black} \\ \tau(\boldsymbol{x})[t(i)..t(j{-}1)] \text{ is Lyndon} & \text{if } j \text{ is white.} \end{cases}$

Proof. For the proof, please see the online report [11], Lemma 7. □

Now we can show that τ-reduction preserves some maximal Lyndon substrings.

Lemma 7. *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ and let $\tau(\boldsymbol{x}) = \tau(\boldsymbol{x})[1..m]$. Let $1 \le i < j \le n$. Let $\boldsymbol{x}[i..j]$ be a maximal Lyndon substring, and let i be a black position.*
Then $\begin{cases} \tau(\boldsymbol{x})[t(i)..t(j)] \text{ is a maximal Lyndon substring} & \text{if } j \text{ is black} \\ \tau(\boldsymbol{x})[t(i)..t(j{-}1)] \text{ is a maximal Lyndon substring} & \text{if } j \text{ is white.} \end{cases}$

Proof. For the proof, please see the online report [11], Lemma 8. □

Now we are ready to tackle the proof Theorem 4 as we promised; please, see the online report [11] for the proof.

```
for $i \leftarrow 1$ to $n$
  if $i=1$ or $\big(\boldsymbol{x}[i{-}1] \succ \boldsymbol{x}[i]$ and $\boldsymbol{x}[i] \preceq \boldsymbol{x}[i{+}1]\big)$ then
    if $\boldsymbol{x}\big[b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]\big){+}1\big] \preceq \boldsymbol{x}[i]$ then
      $\mathcal{L}'_{\boldsymbol{x}}[i] \leftarrow b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]\big)$
    else
      $\mathcal{L}'_{\boldsymbol{x}}[i] \leftarrow b\big(\mathcal{L}'_{\tau(\boldsymbol{x})}[t(i)]\big){+}1$
  else
    $\mathcal{L}'_{\boldsymbol{x}}[i] \leftarrow nil$
```

Figure 2. Computing partial Lyndon array of the input string

9.4 Computing $\mathcal{L}'_{\boldsymbol{x}}$ from $\mathcal{L}'_{\tau(\boldsymbol{x})}$.

Theorem 4 indicates how to compute the partial $\mathcal{L}'_{\boldsymbol{x}}$ from $\mathcal{L}'_{\tau(\boldsymbol{x})}$. The procedure is given in Fig. 2.

How to compute the missing values? The partial array is processed from right to left. When a missing value at position i is encountered (note that it is recognized by $\mathcal{L}'_{\boldsymbol{x}}[i] = nil$), the Lyndon array $\mathcal{L}'_{\boldsymbol{x}}[i{+}1..n]$ is completely filled and also $\mathcal{L}'_{\boldsymbol{x}}[i-1]$ is known. Note that $\mathcal{L}'_{\boldsymbol{x}}[i{+}1]$ is the ending position of the maximal Lyndon substring starting at the position $i{+}1$. If $\boldsymbol{x}[i] \succ \boldsymbol{x}[i{+}1]$, then the maximal Lyndon substring from position $i{+}1$ cannot be extended to the left, and hence the maximal Lyndon substring at the position i has length 1 and so ends in i. Otherwise, $\boldsymbol{x}\big[i..\mathcal{L}'_{\boldsymbol{x}}[i{+}1]\big]$ is Lyndon, and we have to test if we can extend the maximal Lyndon substring right after, and so on. But of course, this is all happening inside the maximal Lyndon substring starting at $i-1$ and ending at $\mathcal{L}'_{\boldsymbol{x}}[i-1]$ due to Monge property[2] of the maximal Lyndon substrings.

This is the **while** loop in the procedure given in Fig. 3 that gives it the $\mathcal{O}(n\ log(n))$ complexity as we will show later. At the first, it may seem that it might actually give it $\mathcal{O}(n^2)$ complexity, but the “doubling of size” trims it effectively down to $\mathcal{O}(n\ log(n))$; see Section 9.5.

```
$\mathcal{L}'_{\boldsymbol{x}}[n] \leftarrow n$
for $i \leftarrow n{-}1$ downto 2
  if $\mathcal{L}'[i] = nil$ then
    if $\boldsymbol{x}[i] \succ \boldsymbol{x}[i{+}1]$ then
      $\mathcal{L}'[i] \leftarrow i$
    else
      if $\mathcal{L}'[i{-}1] = i{-}1$ then
        $stop \leftarrow n$
      else
        $stop \leftarrow \mathcal{L}'[i{-}1]$
      $\mathcal{L}'[i] \leftarrow \mathcal{L}'[i{+}1]$
      while $\mathcal{L}'[i] < stop$ do
        if $\boldsymbol{x}[i..\mathcal{L}'[i]] \prec \boldsymbol{x}[\mathcal{L}'[i]{+}1..\mathcal{L}'[\mathcal{L}'[i]{+}1]]$ then
          $\mathcal{L}'[i] \leftarrow \mathcal{L}'[\mathcal{L}'[i]{+}1]$
        else
          break
```

Figure 3. Computing missing values of the Lyndon array of the input string

Consider our running example from Fig. 1. Since $\tau(\boldsymbol{x}) = 021534$, we have $\mathcal{L}'_{\tau(\boldsymbol{x})}[1..6] = 6, 2, 6, 4, 6, 6$ giving $\mathcal{L}'_{\boldsymbol{x}}[1..9] = 9, \bullet, 3, 9, \bullet, 6, 9, \bullet, 9$. Computing $\mathcal{L}'_{\boldsymbol{x}}[8]$

[2] two maximal Lyndon susbtrings are either disjoint or one completely includes the other

is easy as $\boldsymbol{x}[8] = \boldsymbol{x}[9]$ and so $\mathcal{L}'_{\boldsymbol{x}}[8] = 8$. $\mathcal{L}'_{\boldsymbol{x}}[5]$ is more complicated: we can extend the maximal Lyndon substring from $\mathcal{L}'_{\boldsymbol{x}}[6]$ to the left to 23, but no more, so $\mathcal{L}'_{\boldsymbol{x}}[5] = 6$. Computing $\mathcal{L}'_{\boldsymbol{x}}[2]$ is again easy as $\boldsymbol{x}[2] = \boldsymbol{x}[3]$ and so $\mathcal{L}'_{\boldsymbol{x}}[2] = 2$. Thus $\mathcal{L}'_{\boldsymbol{x}}[1..9] = 9, 2, 3, 9, 6, 6, 9, 8, 9$.

9.5 The complexity of *TRLA*

The complexity of *TRLA* is $\Theta(n\ log(n))$ for a string of length n. The detailed analysis can be found in the online report [11], Section 3.5. The analysis involves two steps: the first step is showing that the complexity is $\mathcal{O}(n\ log(n))$, and the second steps gives a *scheme (F)* of generating binary strings that force $\Theta(n\ log(n))$ execution.

The space complexity of our `C++` implementation is bounded by $9n$ integers. This upper bound is derived from the fact that a `Tau` object (see `Tau.hpp`, [6]) requires $3n$ integers of space for a string of length n. So the first call to *TRLA* requires $3n$, the next recursive call requires at most $3\frac{2}{3}n$, the next recursive call requires at most $3(\frac{2}{3})^2n$, $\ldots$, thus, $3n + 3\frac{2}{3}n + 3(\frac{2}{3})^2n + 3(\frac{2}{3})^3n + \ldots\ = 3n(1 + \frac{2}{3} + (\frac{2}{3})^2 + (\frac{2}{3})^3 + (\frac{2}{3})^4 + \ldots) = 3n\frac{1}{1-\frac{2}{3}} = 9n$.

10 Measurements

All the measurements were performed on the ***moore*** server of McMaster University's Department of Computing and Software; Memory: 32GB (DDR4 @ 2400 MHz), CPU: 8 of the Intel Xeon E5-2687W v4 @ 3.00GHz, OS: Linux version 2.6.18-419.el5 (gcc version 4.1.2) (Red Hat 4.1.2-55), further, all the programs were compiled without any additional level of optimization[3]. The CPU time was measured for each of the programs in seconds with a precision of 3 decimal places (i.e. milliseconds). Since the execution time was negligible for short strings, the processing of the same string was repeated several times (the repeat factor varied from 10^6, for strings of length 10, to 1, for strings of length 10^6), resulting in a higher precision (of up to 7 decimal places). Thus, for graphing, the logarithmic scale was used for both, the x-axis representing the length of the strings, and the y-axis representing the time.

There were 4 categories of datasets: random tight binary strings over the alphabet $\{0, 1\}$, random tight 4-ary strings (kind of random DNA) over the alphabet $\{0, 1, 2, 3\}$, random tight 26-ary strings (kind of random English) over the alphabet $\{0, 1, \ldots, 25\}$, and random tight strings over integer alphabets. Each of the dataset contained 500 randomly generated strings of the same length. For each category, there were datasets for length 10, 50, 10^2, $5{\cdot}10^2$, $\ldots$, 10^5, $5{\cdot}10^5$, and 10^6. The average time for each dataset was computed and used in the following graphs.

As the graphs clearly indicate, the performance of the three algorithms is virtually indistinguishable. We expected *IDLA* and *TRLA* to exhibit linear behaviour on random strings as such strings tend to have almost all maximal Lyndon substrings short with respect the length of the strings. However, we did not expect the results to be so close.

We also tested all three algorithms on datasates containing a single string $01234\ldots n$ referred to as an *extreme_idla* string, which, of course makes *IDLA* exhibit its quadratic complexity, and indeed the results show it; see Fig. 8. The *extreme_trla* strings were generated according to the scheme (F) used in the analysis

[3] i.e. neither `-O1`, nor `-O2`, nor `-O3` flag were specified for the compilation

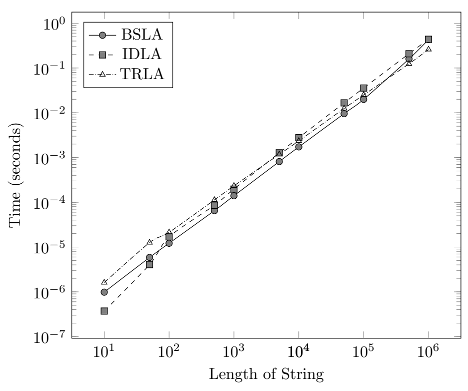


Figure 4. Binary strings

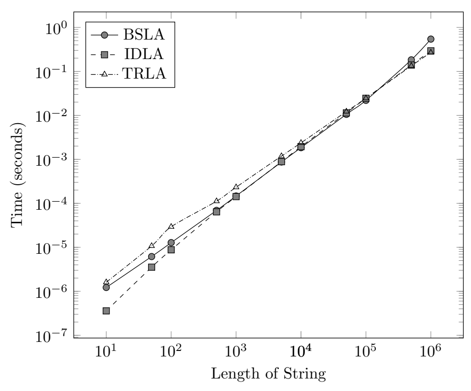


Figure 5. 4-ary strings

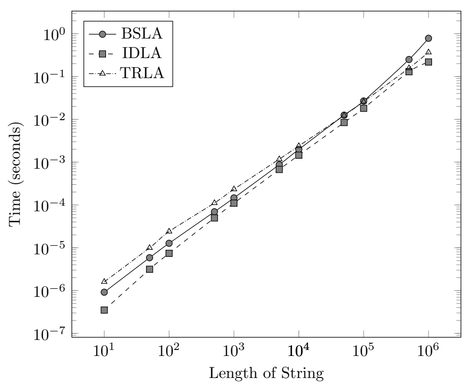


Figure 6. 26-ary strings

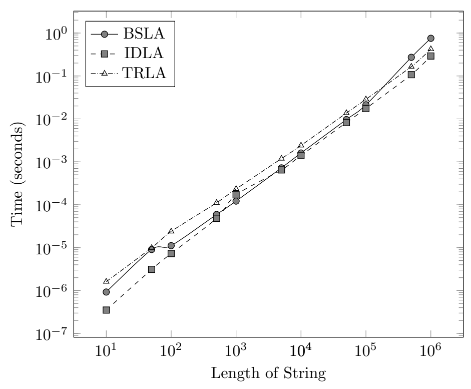


Figure 7. Strings over integer alphabets

of the complexity of *TRLA* in section 9.5. These strings force worst-case execution for *TRLA*. However, even $log(10^6)$ is too small to really highlight the difference, so the results were again very close, see Fig. 9.

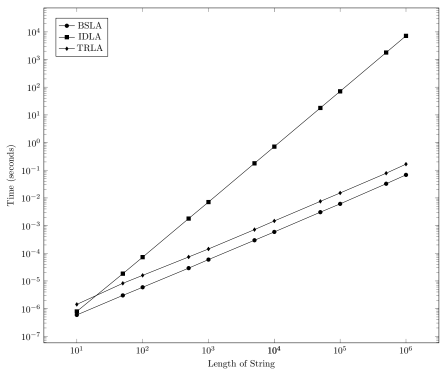


Figure 8. extreme_idla strings

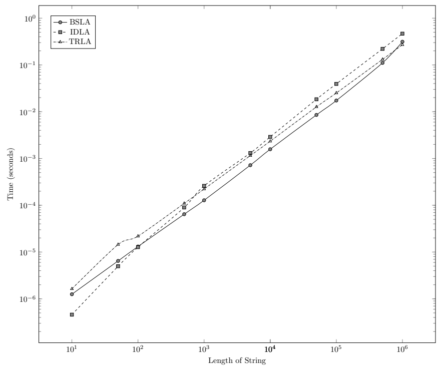


Figure 9. extreme_trla strings

11 Conclusion and Future Work

We presented an overview of current algorithms for computing maximal Lyndon substrings, including new development since the publication of [14]:

- the algorithmic scheme based on the computation of the inverse Burrows-Wheeler transform, *BWLA*, [18];

- the linear algorithm inspired by Phase I of Baier's algorithm, *BSLA*, [11,12]; and
- the novel algorithm based on τ-reduction, *TRLA*.

Then performance of three of the presented algorithms, *IDLA*, *BSLA*, and *TRLA* was compared on various datasets of random strings. The algorithm *TRLA* is mostly of theoretical interest since it has the worst-case complexity $\Theta(n\log(n))$ for strings of length n. Interestingly, on random strings it slightly outperformed *BSLA*, which is linear. Additional effort will go into improving *TRLA*'s complexity in the computation of the missing values. It is imperative that the three algorithms be compared to some efficient *SSLA* or *BWLA* implementation.

References

1. U. Baier: *Linear-time suffix sorting — a new approach for suffix array construction.* M.Sc. Thesis, University of Ulm, Ulm, Germany, 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. 1–12.
3. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta: *The "Runs" Theorem.* Available at `https://arxiv.org/abs/1406.0263`, 2015.
4. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta: *The "Runs" Theorem.* SIAM J. COMPUT., 46 2017, pp. 1501–1514.
5. G. Chen, S. Puglisi, and W. Smyth: *Lempel-Ziv factorization using less time & space.* Mathematics in Computer Science, 1(4) 2013, pp. 605–623.
6. *C++ code for IDLA, TRLA and BSLA algorithms*: Available at `http://www.cas.mcmaster.ca/~franek/research.html`.
7. M. Crochemore, L. Ilie, and W. Smyth: *A simple algorithm for computing the Lempel-Ziv factorization*, in Proc. 18th Data Compression Conference, 2008, pp. 482–488.
8. C. Digelmann: *Personal communication*, 2016.
9. J.-P. Duval: *Factorizing words over an ordered alphabet.* J. Algorithms, 4(4) 1983, pp. 363–381.
10. M. Farach: *Optimal suffix tree construction with large alphabets*, in Proc. 38th IEEE Symp. Foundations of Computer Science, IEEE, October 1997, pp. 137–143.
11. F. Franek and M. Liut: *Computing maximal Lyndon substrings of a string, AdvOL Report 2019/2, McMaster University.* Available at `http://optlab.mcmaster.ca//component/option,com_docman/task,cat_view/gid,77/Itemid,92.`
12. F. Franek, M. Liut, and W. Smyth: *On Baier's sort of maximal Lyndon substrings*, in Proceedings of Prague Stringology Conference 2018, 2018, pp. 63–78.
13. F. Franek, A. Paracha, and W. Smyth: *The linear equivalence of the suffix array and the partially sorted Lyndon array*, in Proc. Prague Stringology Conference, 2017, pp. 77–84.
14. F. Franek, A. Sohidull Islam, M. Sohel Rahman, and W. Smyth: *Algorithms to compute the Lyndon array*, in Proceedings of Prague Stringology Conference 2016, 2016, pp. 172–184.
15. C. Hohlweg and C. Reutenauer: *Lyndon words, permutations and trees.* Theoretical Computer Science, 307(1) 2003, pp. 173–178.
16. J. Kärkkäinen and P. Sanders: *Simple linear work suffix array construction*, in Proceedings of the 30th international conference on Automata, languages and programming, ICALP'03, Berlin, Heidelberg, 2003, Springer–Verlag, pp. 943–955.
17. D. Kosolobov: *Lempel-Ziv factorization may be harder than computing all runs.* Avaiable at `https://arxiv.org/abs/1409.5641`, 2014.
18. F. Louza, W. Smyth, G. Manzini, and G. Telles: *Lyndon array construction during Burrows–Wheeler inversion.* Journal of Discrete Algorithms, 50 2018, pp. 2–9.
19. G. Nong: *Practical linear-time O(1)-workspace suffix sorting for constant alphabets.* ACM Trans. Inf. Syst., 31(3) 2013, pp. 1–15.
20. G. Nong, S. Zhang, and W. H. Chan: *Linear suffix array construction by almost pure induced-sorting*, in 2009 Data Compression Conference, 2009, pp. 193–202.
21. A. Paracha: *Lyndon factors and periodicities in strings.* Ph.D. Thesis, McMaster University, Hamilton, Ontario, Canada, 2017.

k-Abelian Pattern Matching: Revisited, Corrected, and Extended

Golnaz Badkobeh[1], Hideo Bannai[2⋆], Maxime Crochemore[3], Tomohiro I[4⋆⋆], Shunsuke Inenaga[2⋆⋆⋆], and Shiho Sugimoto[5]

[1] Department of Computing, Goldsmiths University of London, London, UK
`g.badkobeh@gold.ac.uk`

[2] Department of Informatics, Kyushu University, Fukuoka, Japan
`{bannai,inenaga}@inf.kyushu-u.ac.jp`

[3] Department of Informatics, King's College London, London, UK
`maxime.crochemore@kcl.ac.uk`

[4] Department of Artificial Intelligence, Kyushu Institute of Technology, Iizuka, Japan
`tomohiro@ai.kyutech.ac.jp`

[5] Security Research Laboratories, NEC, Kawasaki, Japan
`s-sugimoto@ik.jp.nec.com`

Abstract. Two strings of equal length are called k-Abelian equivalent, if they share the same multi-set of factors of length at most k. Ehlers et al. [JDA, 2015] considered the *k-Abelian pattern matching problem*, where the task is to find all factors in a text T that are k-Abelian equivalent to a pattern P. They claimed a number of algorithmic results for the off-line and on-line versions of the k-Abelian pattern matching problem. In this paper, we first argue that some of the claimed results by Ehlers et al. [JDA, 2015] contain major errors, and then we present a new algorithm that correctly solves the offline version of the problem within the same bounds claimed by Ehlers et al., in $O(n + m)$ time and $O(m)$ space, where $n = |T|$ and $m = |P|$. We also show how to correct errors in their online algorithm, and errors in their real-time algorithms for a slightly different problem called the *extended* k-Abelian pattern matching problem.

Keywords: Abelian equivalence, pattern matching, suffix trees, suffix arrays

1 Introduction

Two strings X and Y of equal length are said to be *Abelian equivalent* if the numbers of occurrences of each letter are equal in X and Y. For instance, strings `ababaac` and `caaabba` are Abelian equivalent. Since the seminal paper by Erdős [14] published in 1961, the study of Abelian equivalence on strings has attracted much attention, both in word combinatorics and string algorithmics. One good example is the Abelian version of pattern matching problem, where the task is to locate all factors of a given text T that are Abelian equivalent to a given pattern P (reporting version), or to test whether there is such a factor in T (existence version). This problem is called the *jumbled pattern matching problem*, and a number of algorithms have been proposed for this problem; see the subsection for related work below.

k-Abelian equivalence is a natural generalization of Abelian equivalence: For a positive integer k, two strings X and Y of equal length are said to be k-Abelian equivalent if the numbers of occurrences of each string of length *at most* k are equal in X and Y. The notion of k-Abelian equivalence of strings was first introduced

⋆ Supported by JSPS KAKENHI Grant Number JP16H02783
⋆⋆ Supported by JSPS KAKENHI Grant Number JP19K20213
⋆⋆⋆ Supported by JSPS KAKENHI Grant Number JP17H01697

by Huova et al. [20], and then has extensively been studied in the context of word combinatorics such as k-Abelian repetitions [20,19,21], k-Abelian periodicities [23], k-Abelian equivalence classes [22,10,25], just to mention a few.

The first (and only, to our knowledge) algorithmic results concerning k-Abelian equivalence of strings were given by Ehlers et al. [13] for the *k-Abelian pattern matching problem*. Here the task is, given a text T and pattern P, to locate all factors of T that are k-Abelian equivalent to P. Ehlers et al. [13] considered the offline and online versions of the k-Abelian pattern matching problem, and claimed a number of results with different bounds.

In this paper, we first argue that some of those claimed results by Ehlers et al. [13] contain major errors. These errors are due to the abuse of the van Emde Boas data structure [7] that uses space linear in the size of the *universe* of the integers, no matter how many elements are stored in the data structure. There are also other major issues such as carelessness on the size of the integer alphabet from which the text is drawn, unknown construction time of the weighted ancestor data structure on suffix trees [17], and so on. We then present a new algorithm that actually solves the offline version of the problem within the same bounds claimed by Ehlers et al., namely in $O(n+m)$ time and $O(m)$ space, where $n = |T|$ and $m = |P|$.

Ehlers et al. [13] also considered a slightly different variant of the k-Abelian pattern matching problem called the *extended k-Abelian pattern matching problem*. Here, two strings X and Y of equal length are said to be extended k-Abelian equivalent if the numbers of occurrences of each string of length *exactly* k are equal in X and Y. We point out the major errors in their solutions to extended k-Abelian pattern matching, and show how to obtain alternative solutions.

Related work

For jumbled pattern matching (i.e. 1-Abelian pattern matching), there is a simple algorithm that compares the number of occurrences of all letters $a \in \Sigma$ of the pattern P and a sliding window of length m over the text T. In case P is over an integer alphabet of size linear in m, shifting the window takes $O(1)$ time per text letter and hence this algorithm runs in $O(n+m)$ time and $O(m)$ working space, where n and m are the lengths of T and P, respectively. Butman et al. [9] considered how to solve this problem on run-length encoded strings. When n' and m' are respectively the sizes of the run-length encoded text and pattern, then their algorithm runs in $O(n'+m')$ time with $O(m')$ working space, given that the pattern is over an integer alphabet of size linear in m'. The essentially same algorithm was later rediscovered by Sugimoto et al. [35] and was used as a sub-routine in their algorithms to compute Abelian regularities from run-length encoded strings.

The *indexing* version of the jumbled pattern matching is more challenging and has attracted much attention. Amir et al. [1] proposed an indexing structure of $O(n^{1+\epsilon})$ space that can be constructed in $O(n^{1+\epsilon}\log\sigma)$ time and can decide whether there is an occurrence of the pattern in $O(m^{\frac{1}{\epsilon}}+\log\sigma)$ time, where σ is the alphabet size and $0 < \epsilon < 1$ is any constant. Their algorithm works for any alphabets. For any constant-size alphabets, Kociumaka et al. [29] proposed an $O(n^2/L)$-space data structure which can be constructed in $O(n^2(\log\log n)^2)/\log n)$ time and can report the left-most occurrence in $O(L^{2\sigma-1})$ time, where n is the length of a given text t and L is a trade-off parameter ranging from 1 to n. For alphabets of size $\sigma = \omega(1)$, Amir et al. [2] showed that jumbled indexing requires $\Omega(n^{2-\lambda})$ preprocessing time or $\Omega(n^{1-\delta})$ query time for every $\lambda, \delta > 0$, under the famous 3SUM-hardness assumption.

There have been several indexing structures for *binary* jumbled pattern matching (BJPM). Cicalese et al. [12] gave an index for BJPM that uses $O(n)$ space, can be constructed in $O(n^2)$ time and can decide the existence of occurrences in $O(1)$ time. Improved $O(n^2/\log n)$-time construction of the BJPM indexing was independently proposed by Burcsi et al. [8] and by Moosa and Rahman [31]. Later, construction time was further improved to $O(n^2/(\log n)^2)$ by Moosa and Rahman [32]. Hermelin et al. [18] showed how to reduce the BJPM problem to the all-pairs shortest paths problem and presented an $O(n^2/2^{\Omega(\log n/\log\log n)^{0.5}})$ preprocessing scheme for the BJPM problem. Chan and Lewenstein [11] presented a breakthrough solution for the BJPM problem that takes $O(n^{1.864})$ time for preprocessing and requires $O(1)$ time for queries. Their solution can also be extended to larger constant-size alphabets, with strongly sub-quadratic preprocessing time and strongly sub-linear query time.

2 Preliminaries

Let Σ be an ordered alphabet of size σ. An element of Σ^* is called a *string*. Let ε denote the empty string of length 0. For a non-negative integer k, let Σ^k denote the set of strings of length k. For a string $u = xyz$, x, y, and z are called a *prefix*, *factor*, and *suffix* of u, respectively. For a string u of length n, let $u[i]$ denote the ith letter in u for $1 \le i \le n$, and $u[i..j]$ denote the factor of u that begins at position i and ends at position j for $1 \le i \le j \le n$. For a non-negative integer k, a factor of length k in a string u is called a *k-gram* in u. For a positive integer n, let $[1..n]$ denote the set of n positive integers from 1 to n.

For any string $u \in \Sigma^*$ and letter $a \in \Sigma$, $|u|_a$ denotes the number of occurrences of a in u. Two strings u and v are said to be *Abelian equivalent* if $|u|_a = |v|_a$ for all letters $a \in \Sigma$. To simplify the argument, let us identify each letter $a \in \Sigma$ with its lexicographical rank in Σ.

Now, we extend the aforementioned notion from occurrences of letters to those of strings. Namely, for a string t, let $|u|_t$ denote the number of occurrences of t in u.

Definition 1 (k-Abelian equivalence). *For a positive integer k, two strings u and v of equal length n are said to be* k-Abelian equivalent *if either*

(1) $u = v$ or
(2) all the following conditions hold:
  (a) $|u|, |v| \ge k$;
  (b) $|u|_t = |v|_t$ for all strings $t \in \Sigma^k$;
  (c) $u[1..k-1] = v[1..k-1]$;
  (d) $u[n-k+2..n] = v[n-k+2..n]$.

According to [24], the last condition (2)-(d) for having the same suffix of length $k-1$ can actually be dropped.

We denote $u \equiv_k v$ when u and v are k-Abelian equivalent. It is known that $u \equiv_k v$ iff $|u|_s = |v|_s$ for every string s of length at most k.

Example 1. Let $x = abaababbaab$ and $y = abbaabaabab$. For $k = 3$, x and y are k-Abelian equivalent, since they satisfy $|x| = |y| = 11 \ge 3$, $|x|_t = |y|_t$ for all strings $t \in \Sigma^3$ i.e. $|x|_{aaa} = |y|_{aaa} = 0, |x|_{aab} = |y|_{aab} = 2, |x|_{aba} = |y|_{aba} = 2, |x|_{abb} = |y|_{abb} = 1, |x|_{baa} = |y|_{baa} = 2, |x|_{bab} = |y|_{bab} = 1, |x|_{bba} = |y|_{bba} = 1, |x|_{bbb} = |y|_{bbb} = 0$, and their prefixes of length $k-1 = 2$ are equal i.e. $x[1..2] = y[1..2] = ab$.

In this paper, we consider the following problem.

Problem 1. Given a text T and a pattern P over an alphabet Σ and a positive integer k, locate all factors of T that are k-Abelian equivalent to P.

For simplicity, suppose that a string u terminates with a special letter that does not appear elsewhere in u. The *suffix tree* of string u of length n is a rooted edge-labeled tree such that (1) each internal node is branching, (2) each edge is labeled with a non-empty substring of u, (3) the labels of out-going edges of each node are mutually distinct, and (4) there is a one-to-one correspondence between suffixes of u and the leaves of the tree. The *locus* of a substring x of u in the suffix tree of u is the ending position of the path that spells out x from the root. When there is a node such that the path from the root to this node spells out x, then the locus of x is on that node. When there is no such node, then the locus of x is on an edge.

The *suffix array* of string u of length n, denoted SA_u, is an array of length n such that, for $1 \leq i \leq n$, $\mathsf{SA}_u[i] = j$ iff $u[j..n]$ is the ith lexicographically smallest suffix of u. SA_u can be seen as an array of the leaves of the suffix tree for u where the out-going edges are sorted in lexicographical order. The *LCP* array of string u, denoted LCP_u, is an array of length n such that $\mathsf{LCP}_u[1] = -1$ and, for $2 \leq r \leq n$, $\mathsf{LCP}_u[r]$ stores the length of the longest common prefix of the suffixes stored at positions $r-1$ and r in the suffix array i.e., $u[\mathsf{SA}_u[r-1]..n]$ and $u[\mathsf{SA}_u[r]..n]$.

3 Online and offline k-Abelian pattern matching

In this section, we point out some errors in the claims from the previous work of Ehlers et al. [13]. They considered Problem 1 in two settings, the *offline version* where the whole text and pattern are given together as input, and the *online version* where the pattern is given first to preprocess and the text letters are given in an online manner, one by one from left to right. In the online version, each time a new text letter arrives, a new k-Abelian equivalent occurrence of the pattern in the text must be reported (if it exists).

3.1 Offline k-Abelian pattern matching problem

In this subsection, we consider the offline version of Problem 1.

Errors in the previous work. Ehlers et al. [13] stated the following claim.

Claim (Remark 2 of [13] in conjunction with Theorem 2 of [13]). The offline version of Problem 1 can be solved in $O(n+m)$ time and $O(m)$ space for integer alphabet $\Sigma = [1..n]$[1].

Below, we show that Ehlers et al.'s approach [13] does not fulfill the above claim and uses more space than $O(m)$. To see why, let us briefly describe their approach from Theorem 2 of their paper [13]. For a string $u \in \Sigma^n$, consider the k-encoded string $\#(u,k)$ of length $n-k+1$ such that for each i ($1 \leq i \leq n-k+1$), $\#(u,k)[i]$ stores the lexicographical rank of the k-gram $u[i..i+k-1]$ in the set $\{u[i..i+k-1] \mid 1 \leq i \leq n-k+1\}$ of all k-grams in T. Given a text T and pattern P, they consider a concatenated string $w = T\$P$ where \$ is a special letter not appearing in T or P,

[1] This assumption of the integer alphabet is given in Section 2 (Preliminaries) in [13].

and compute $\#(w, k)$. Let $T' = w[1..n-k+1]$ and $P' = w[n+1..n+m-k+2] = P[1..m-k+1]$. Then, a key observation is that for each i $(1 \leq i \leq n-m+1)$, $T[i..i+m-1]$ and P are k-Abelian equivalent iff $T'[i..i+m-k]$ and P' are Abelian equivalent, and $T[i..i+k-2] = P[1..k-1]$. To compute $\#(w, k)$ and to test whether $T'[i..i+m-k]$ and P' are Abelian equivalent or not, they build the suffix array SA_w for the concatenated string w together with the LCP array LCP_w enhanced with a constant-time range minimum query data structure [5].

While the aforementioned approach works in $O(n+m)$ time for the integer alphabet $\Sigma = [1..n]$, it also requires $O(n+m)$ space. As an attempt to reduce the space requirement to $O(m)$, they chose the following approach: For ease of explanation, suppose that n is divisible by m. For each $0 \leq t \leq \frac{n}{m} - 2$, they pick the factor $T[tm{+}1..(t{+}2)m]$ of length $2m$, and built the suffix array of $w_t = T[tm{+}1..(t{+}2)m]\$P$ and apply the above method to w_t, namely construct the suffix array SA_{w_t} and LCP array LCP_{w_t} for each t.

However, this approach indeed takes $O(n+m)$ time and $O(n)$ space for *each* t. This is because, regardless of its length, any factor of the text T over the integer alphabet $[1..n]$ can contain letters (i.e. integers) up to n. In other words, any factor of such text T is still a string over the integer alphabet $[1..n]$. The above argument implies that the *universe* of the letters in $T[tm{+}1..(t{+}2)m]$ is $[1..n]$ in the worst case. Recall that *any* existing linear-time suffix array construction algorithms for the integer alphabet $[1..n]$ use bucket sort [28,27,26,33,3], and that any suffix array construction with comparison-based sorting must take $\Omega(n \log n)$ time for any ordered alphabet of size $O(n)$ [15]. In general, bucket sort for a set of s integers over the integer universe $[1..u]$ requires $O(s+u)$ time and $O(u)$ space, since it uses an integer array of length u. Therefore, Ehlers et al.'s method (Remark 2 of [13]) must use $O(n+m)$ time and $O(n)$ space for *each* t. Moreover, this leads to $O(n(n+m)/m)$ total time for all t's, which is super-linear in reasonably common cases where $m = o(n)$.

New offline algorithm. Now we present a new algorithm for the offline version of Problem 1 that indeed uses only $O(m)$ space. To achieve this goal, we introduce a reasonable assumption that the pattern P of length m is over an integer alphabet of size $[1..cm]$ with any positive constant c such that cm is a positive integer. Then we show the following:

Theorem 1. *Let P be a pattern of length m over an integer alphabet $[1..cm]$ with any positive constant c, and T be a text of length n over an arbitrary integer alphabet. Then, for a given integer $k > 0$, we can solve Problem 1 in $O(n+m)$ time using $O(m)$ space.*

Proof. Our proposed algorithm uses suffix trees. Namely, for each t $(0 \leq t \leq \frac{n}{m} - 2)$, we construct the suffix tree of $w_t = T[tm+1..(t+2)m]\$P$. For each occurrence of a k-gram in P, we construct a bucket that is associated to the locus of the k-gram in the suffix tree. The locus is an implicit or explicit node of string depth k. If a k-gram occurs z times in P, then there will be z buckets in its corresponding locus. Initially, all the buckets are empty.

Now, we check whether each factor of T of length m fulfill all the buckets. For each $x = 0, \ldots, t-1$ in increasing order, we map the factor $T[tm+1+x..tm+k+x]$ the (implicit or explicit) node of string depth k representing $T[tm{+}1{+}x..tm{+}k{+}x]$, and if there is a bucket there, we fulfill it with position $tm{+}1{+}x$. This can easily be done in $O(1)$ time per x after an $O(m)$-time preprocessing – for every leaf in the suffix tree, we can compute its ancestor of string depth k in $O(m)$ total time with a standard tree traversal. We keep track of a sliding window of length m over $T[tm+1..(t+2)m]$, and the positions in the buckets are removed as soon as they are out of the window. This can easily be done by implementing the set of buckets in each node by a queue. Each time all the buckets are fulfilled, then we additionally check if the $(k-1)$-gram of the text beginning with the current smallest position j in the buckets satisfies Condition (2)-(c) of Definition 1 with $P[1..k-1]$. This additional step can easily be done in $O(1)$ time by marking the locus of the suffix tree representing $P[1..k-1]$. If the condition is satisfied, then we output the beginning position j of the text factor that is k-Abelian equivalent to P. Then, we delete the position j from the corresponding bucket, and proceed to the next position by increasing x.

What remains is how to reduce the alphabet size of T. For this sake we replace *any* letter in T that exceeds cm with $cm+1$, where cm is the largest letter appearing in P. The resulting new text $\hat{T}$ is now a string of length n over the integer alphabet $[1..cm+1]$. For each t, the suffix tree of $\hat{T}[tm+1..(t+2)m]\$P$ can be constructed in $O(m)$ time and space, by the suffix tree construction algorithm for integer alphabets [15], or via any linear-time suffix array construction algorithm for integer alphabets and the LCP array. Note that all k-Abelian equivalent occurrences of P in the text are preserved in the new text $\hat{T}$. Thus, our algorithm runs in $O(n+m)$ time with $O(m)$ space. □

3.2 Online k-Abelian pattern matching problem

In this subsection, we consider the online version of Problem 1. In this variant of the problem, the authors assume that $\Sigma = [1..\sigma]$ with $\sigma \in O(m)$ [13][2].

The key idea of their algorithm is to use the following list L: Let $D_{k-1}(P)$ and $D_k(P)$ be the set of $(k-1)$-grams and k-grams that occur in P, respectively. Let f_1 be an array of length $|D_{k-1}(P)|$ such that $f_1[i]$ stores an occurrence of the lexicographically ith $(k-1)$-gram in $D_{k-1}(P)$. Similarly, let f_2 be an array of length $|D_k(P)|$ such that $f_2[j]$ stores an occurrence of the lexicographically jth k-gram in $D_k(P)$. Now the list L is defined as follows.

$$L = \{(i,a,j) \mid 1 \le i \le |D_{k-1}(P)|, 1 \le j \le |D_k(P)|, a \in \Sigma, f_1[i]a = f_2[j]\}.$$

While the original online algorithm by Ehlers et al. uses the suffix array and lcp array for P to implement L, in our explanation we use the suffix tree for P since it seems more intuitive and easier to follow[3]. Also, recall our offline algorithm of Theorem 1 as the method to follow can be seen as its online version. Let $\mathsf{STree}(P)$ denote the suffix tree for P.

Now one can regard L as the set of the edges of $\mathsf{STree}(P)$ that connect the (implicit or explicit) nodes of string depth $k-1$ to the nodes of string depth k. Now the basic strategy is the following. Let T' be the current text, a the next letter to be appended to T', and $T = T'a$. Suppose that we know the locus in $\mathsf{STree}(P)$ that represents the

[2] Ehlers et al. deal with the offline version of the problem in Section 3 and the online version in Section 4 in their paper [13]. While they write "As before, we assume that $\Sigma = [1..\sigma]$ with $\sigma \in O(m)$." in the beginning of Section 4, we cannot find such assumption in Section 3 or earlier in their paper.

[3] This variant of algorithm with suffix trees is also used by Ehlers et al. for online extended k-Abelian pattern matching (Section 5 of [13]).

suffix of T' of length $(k-1)$, which is the rightmost $(k-1)$-gram in T' (if it exists in the tree). Then, the task is to quickly find the out-going edge labeled a from this locus, since there we can find the locus of the suffix of T of length k in $\mathsf{STree}(P)$. This is a classical problem of implementing the set of out-going edges of a node of labeled trees, and a number of data structures can be used for this purpose. Ehlers et al. stated the following claim:

Claim (The 4th bound of Theorem 4 of [13]). Given a static pattern $P \in \Sigma^m$ over the integer alphabet $[1..\sigma]$, and a positive integer k, the online version of Problem 1 can be solved in $O(m)$ preprocessing time, $O(m)$ working space, and $O(\log\log\sigma)$ time per text letter.

The idea of the above claim is to use the van Emde Boas data structure [7]. However, it is well known that for an integer universe $U = [1..u]$ of size u, the van Emde Boas data structure of a set $S \subseteq U$ requires $\Theta(u)$ space regardless of the cardinality of S. In the above context, $u = \sigma$ since the universe here is the integer alphabet $[1..\sigma]$. This implies that this approach by Ehlers et al.'s must use $O(\sigma m)$ space, since there can be $O(m)$ nodes of string depth $k-1$ in the suffix tree. Thus the above claim does not hold.

Indeed, Ehlers et al. also proposed a simple array-based implementation of the branching edges (the 1st variant of Theorem 4 of [13]). When one can afford to using $O(\sigma m)$ space, then this simple array-based approach is faster since each edge can be accessed in $O(1)$ time. By the way, the 1st variant of Theorem 4 of [13] states that their preprocessing requires only $O(m)$ time. This is not the case, since this variant must use $O(\sigma m)$ time to construct all the arrays.

4 Extended k-Abelian pattern matching

Ehlers et al. also considered a slightly different notion of k-Abelian equivalence, called *extended k-Abelian equivalence.*

Definition 2 (extended k-Abelian equivalence). *For a positive integer k, two strings u and v of equal length said to be* extended k-Abelian equivalent *if their multi-sets of factors of length k coincide, i.e., both of the last two conditions (2)-(c) and (2)-(d) of having the same prefixes and suffixes are dropped from Definition 1.*

Example 2. Let $x = abaababbaab$ and $y = baabaabbaba$. For $k = 3$, x and y are not k-Abelian equivalent but extended k-Abelian equivalent, since they satisfy $|x| = |y| = 11 \geq 3$, $|x|_t = |y|_t$ for all strings $t \in \Sigma^3$ i.e. $|x|_{aaa} = |y|_{aaa} = 0, |x|_{aab} = |y|_{aab} = 2, |x|_{aba} = |y|_{aba} = 2, |x|_{abb} = |y|_{abb} = 1, |x|_{baa} = |y|_{baa} = 2, |x|_{bab} = |y|_{bab} = 1, |x|_{bba} = |y|_{bba} = 1, |x|_{bbb} = |y|_{bbb} = 0$, but their prefixes of length $k-1 = 2$ are not equal i.e. $x[1..2] = ab \neq ba = y[1..2]$.

Problem 2. Given a text T and a pattern P over an alphabet Σ and a positive integer k, locate all factors of T that are extended k-Abelian equivalent to P.

4.1 Errors in the previous work

They considered the online version of Problem 2 and claimed the following *real-time* bounds. Here, an online algorithm is called real-time if an $O(1)$ worst-case time is guaranteed per text symbol.

Claim (Theorem 6 of [13]). Given a static pattern $P \in \Sigma^m$ over the integer alphabet $[1..\sigma]$, and a positive integer k, the online version of Problem 2 can be solved in:

- $O(m \log k)$ preprocessing time, $O(\sigma m)$ working space, and $O(1)$ worst-case time per text letter;
- $O(m(\log\log m + \log k))$ preprocessing time, $O(m)$ working space, and $O(1)$ worst-case time per text letter;
- $O(m \log k)$ expected preprocessing time, $O(m)$ working space, and $O(1)$ worst-case time per text letter;
- $O(m \log k)$ preprocessing time, $O(m)$ working space, and $O(\log\log \sigma)$ worst-case time per text letter.

We note that the 4th variant is based on the same flawed argument with the van Emde Boas data structure as in Section 3.2, and thus it indeed requires $O(\sigma m)$ space. Hence the 1st variant is better, and we will ignore the 4th variant in the sequel. Also, since the 1st variant uses $O(\sigma m)$ space for preprocessing, there should be an additive σm term in the preprocessing time.

The $m \log k$ term that are common in the preprocessing time of the above claim comes from the next statement from [13]:

Claim (Lemma 5 of [13]). One can preprocess pattern P of length m in $O(m \log k)$ time and linear space such that, for each i and j with $j - i \leq k$, one can return in constant time the (explicit or implicit) node of $\mathsf{STree}(P)$ that corresponds to the factor $P[i..j]$.

Their claim relies on the result of Gawrychowski et al. [17] for the *constant-time weighted ancestor queries on suffix trees.*

A weighted tree is a rooted tree where an integer weight is assigned to each node, so that the weight of any node is strictly greater than the weight of its parent. A weighted ancestor query is, given a node V and an integer g, find the highest ancestor of V that has a weight at least g. In the context of suffix trees, the weight of a node is its string depth. Namely, Ehlers et al.'s approach [13] is to apply Gawrychowski et al.'s algorithm to the *truncated suffix tree* for P that consists only of the paths of string depths at most k. However, Gawrychowski et al.'s paper [17] does *not* consider construction time of their constant-time weighted ancestor data structure on suffix trees. Even on the (non-truncated) suffix tree for a string of length m, it seems rather challenging to construct the constant-time weighted ancestor data structure in $O(m \log m)$ time[4]. Hence, it is not known whether there exists an algorithm that satisfies the above claim (Lemma 5 of [13]), nor whether there exist algorithms that satisfy the other claim (Theorem 6 of [13]).

4.2 New real-time algorithms for extended *k*-Abelian matching

Here we propose some solutions for the online (and real-time) version of the extended k-Abelian pattern matching problem.

The basic framework of the approach by Ehlers et al. [13] is to compute the *k-gram matching statistics* for T against P, defined as follows: The k-gram matching statistics of T against P is the sequence of $|T| - k + 1$ integers $\ell_1, \ldots, \ell_{|T|}$ such that for each $1 \leq j \leq |T| - k + 1$ each ℓ_j is the length of the longest prefix of the k-gram

[4] One of the authors from [17] wondered that $O(m \log^3 m)$ or $O(m \log^4 m)$ construction time might be plausible [16], but any non-trivial construction algorithm is not known to date.

$T[j..j+k-1]$ that occurs in P. A k-gram $T[h..h+m-1]$ occurring at position h in T is extended k-Abelian equivalent to P iff $\ell_h = \ell_{h+1} = \cdots = \ell_{h+m-1} = k$. For each text position $1 \leq j \leq |T| - k + 1$, ℓ_j can be computed in $O(1)$ *amortized* time per text letter, using a similar technique to Ukkonen's online suffix tree construction [36], where traversals of "virtual" suffix links of implicit nodes are simulated by the suffix links of their explicit parents. The number of nodes that are visited in the simulation of each suffix link in $\mathsf{STree}(P)$ (and hence for each text letter) can be amortized constant [36], and this is basically what Theorem 5 of their paper [13] for non real-time solutions achieves.

To de-amortize the cost, Ehlers et al. [13] considered to use a weighted ancestor data structure instead of virtual suffix links. The idea is that one can find the locus pointed by a (virtual) suffix link with a weighted ancestor query from a corresponding leaf in the truncated suffix tree for P.

Instead of using the data structure by Gawrychowski et al. [17] whose construction time is unknown, one could use *level ancestor queries* on a limited class of weighted trees where the weight of each node is its node depth (not string depth). It is known that one can preprocess a tree with m nodes in $O(m)$ time so that level ancestor queries can be answered in $O(1)$ worst-case time [6]. We build this data structure on the full suffix tree $\mathsf{STree}(P)$. We also preprocess $\mathsf{STree}(P)$ such that for each $1 \leq i \leq m-k+1$ the leaf representing the suffix $P[i..m]$ has a pointer to its (implicit or explicit) ancestor of string depth k. These pointers can easily be precomputed in $O(m)$ time by a standard tree traversal, as was done in Section 3.1. Now suppose that we have just computed ℓ_j for $T[j..j+k-1]$ against P, and let i be one of the positions in P such that $P[i..i+\ell_j] = T[j..j+\ell_j]$. Then, for the weight $\ell_j - 1$ that represents the string depth we wish to jump up for the next text position $j+1$ in the text, we first take the pointer from the next leaf for $P[i+1..m]$, and from this leaf we binary search the nearest ancestor with weight $\ell_j - 1$ by level ancestor queries. Recall that this simulates the (virtual) suffix link traversal. If we can traverse with letter $T[j+\ell_j+1]$ from this locus of weight (i.e. string depth) $\ell_j - 1$, we are done for position $j+1$. Otherwise, we move to the next leaf for $P[i+2..m]$ and perform binary search for weight $\ell_j - 2$, and so forth. Since we need level ancestor queries only from nodes of string depth at most k (and hence node depth at most k), we can binary search the weighted ancestor with $O(\log k)$ level ancestor queries, in $O(\log k)$ worst-case time for each text letter.

Alternatively, we can use a weighted ancestor data structure that is designed for arbitrary weighted trees (i.e., not specialized for suffix trees). Kopelowitz and Lewenstein [30] showed that weighted ancestor queries on a weighted tree with m nodes can be reduced to a constant number of predecessor queries on a collection of predecessor data structures that maintain a total of $O(m)$ elements, where the number of elements in each predecessor data structure is bounded by the height of the tree. Insertions of new nodes can also be supported by a constant number of updates (insertions/deletions) in the collection of predecessor data structures. Therefore, if there is a dynamic predecessor data structure for a set of m integers over the universe $[1..u]$, that allows for queries/updates in $\mathrm{pred}(m,u)$ time and $O(m)$ space, then weighted ancestor queries on a weighted tree with m nodes with weights from $[1..u]$ can be answered in $O(\mathrm{pred}(m,u))$ time with $O(m)$ space (see Theorem 7.1 of [30]). We can plug-in the following linear-space dynamic predecessor data structures to the above result.

Lemma 1 ([4]). *There is a dynamic predecessor data structure for a set of up to m integers from the universe $[1..u]$ that uses $O(m)$ space and supports updates and queries in $O\Big(\min\Big\{\frac{(\log\log m)(\log\log u)}{\log\log\log u}, \sqrt{\frac{\log m}{\log\log m}}\Big\}\Big)$ worst-case time.*

Lemma 2 (y-fast trie [37] in conjunction with cuckoo hashing [34]). *There is a dynamic predecessor data structure for a set of up to m integers from the universe $[1..u]$ that uses $O(m)$ space and supports updates in $O(\log\log u)$ expected amortized time and predecessor queries in $O(\log\log u)$ worst-case time.*

In the current context we have $u = m$, since any node in $\mathsf{STree}(P)$ has string depth at most m. Note that $\min\Big\{\frac{(\log\log m)^2}{\log\log\log m}, \sqrt{\frac{\log m}{\log\log m}}\Big\} = \frac{(\log\log m)^2}{\log\log\log m}$.

Plugging these bounds where appropriate, we obtain the following:

Theorem 2 (Near real-time extended k-Abelian pattern matching). *Given a static pattern P of length m over the integer alphabet $[1..\sigma]$, and a positive integer k, the online version of Problem 2 can be solved in:*

- *$O(m\sigma)$ preprocessing time, $O(m\sigma)$ working space, and $O(\log k)$ worst case per text letter;*
- *$O(m\log\log m)$ preprocessing time, $O(m)$ working space, and $O(\log k)$ worst case per text letter;*
- *$O\Big(m\frac{(\log\log m)^2}{\log\log\log m}\Big)$ preprocessing time, $O(m)$ working space, and $O\Big(\frac{(\log\log m)^2}{\log\log\log m}\Big)$ worst-case time per text letter;*
- *$O(m\log\log m)$ expected preprocessing time, $O(m)$ working space, and $O(\log\log m)$ worst-case time per text letter.*

5 Conclusions and future work

In this paper, we pointed out some errors in the previous work by Ehlers et al. [13], provided a rigorous analysis on the complexities of some of the proposed algorithms by Ehlers et al. [13], and presented correct and alternative algorithms. For the offline k-Abelian pattern matching problem, we described that the algorithm by Ehlers et al. [13] indeed uses $O(n+m)$ space, and proposed a new offline algorithm which woks within $O(m)$ space. For the online k-Abelian pattern matching problem, we pointed out the abuse of the van Emde Boas data structure in Ehlers et al.'s algorithm and explained that this approach indeed uses $O(\sigma m)$ space. Finally, we pointed out that all the bounds claimed in [13] for the real-time extended k-Abelian pattern matching seem difficult to achieve, as these are heavily dependent on Gawrychowski et al.'s structure [17] of whose construction time is unknown. We proposed new alternative real-time algorithms for extended k-Abelian pattern matching with other data structures.

An interesting future work is to consider an efficient *indexing structure* for (extended) k-Abelian pattern matching. In a restricted case where both the alphabet size σ and k are fixed, then we can simply transform each k-gram in a given text T into a meta-letter, and transform T to a meta-string of length roughly kn. Since we have assumed that σ and k are fixed, $kn = O(n)$ and this meta-string is a string over an alphabet of size σ^k which can be seen as a constant as well. Thus we can use Amir et al.'s jumbled matching index [1] that works for any alphabet, or Kociumaka et al.'s jumbled matching index [29] that works for any constant-size alphabet. It would be interesting to develop an indexing structure that is specially designed for (extended) k-Abelian pattern matching with better complexities.

References

1. A. Amir, A. Butman, and E. Porat: *On the relationship between histogram indexing and block-mass indexing.* Philos. Trans. A. Math. Phys. Eng. Sci., 372(2016) 2014, p. 20130132.
2. A. Amir, T. M. Chan, M. Lewenstein, and N. Lewenstein: *On hardness of jumbled indexing*, in Proc. ICALP 2014, 2014, pp. 114–125.
3. U. Baier: *Linear-time suffix sorting - A new approach for suffix array construction*, in Proc. CPM 2016, 2016, pp. 23:1–23:12.
4. P. Beame and F. E. Fich: *Optimal bounds for the predecessor problem and related problems.* J. Comput. Syst. Sci., 65(1) 2002, pp. 38–72.
5. M. A. Bender and M. Farach-Colton: *The LCA problem revisited*, in Proc. LATIN 2000, 2000, pp. 88–94.
6. M. A. Bender and M. Farach-Colton: *The level ancestor problem simplified.* Theor. Comput. Sci., 321(1) 2004, pp. 5–12.
7. P. v. E. Boas, R. Kaas, and E. Zijlstra: *Design and implementation of an efficient priority queue.* Mathematical Systems Theory, 10 1977, pp. 99–127.
8. P. Burcsi, F. Cicalese, G. Fici, and Z. Lipták: *Algorithms for jumbled pattern matching in strings.* Int. J. Found. Comput. Sci., 23(2) 2012, pp. 357–374.
9. A. Butman, R. Eres, and G. M. Landau: *Scaled and permuted string matching.* Inf. Process. Lett., 92(6) 2004, pp. 293–297.
10. J. Cassaigne, J. Karhumäki, S. Puzynina, and M. A. Whiteland: *k-Abelian equivalence and rationality.* Fundam. Inform., 154(1-4) 2017, pp. 65–94.
11. T. M. Chan and M. Lewenstein: *Clustered integer 3SUM via additive combinatorics*, in Proc. STOC 2015, 2015, pp. 31–40.
12. F. Cicalese, G. Fici, and Z. Lipták: *Searching for jumbled patterns in strings*, in Proc. PSC 2009, 2009, pp. 105–117.
13. T. Ehlers, F. Manea, R. Mercas, and D. Nowotka: *k-Abelian pattern matching.* J. Discrete Algorithms, 34 2015, pp. 37–48.
14. P. Erdös: *Some unsolved problems.* Hungarian Academy of Sciences Mat. Kutató Intézet Közl, 6 1961, pp. 221–254.
15. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan: *On the sorting-complexity of suffix tree construction.* J. ACM, 47(6) 2000, pp. 987–1011.
16. P. Gawrychowski: *Personal communication*, October 2017.
17. P. Gawrychowski, M. Lewenstein, and P. K. Nicholson: *Weighted ancestors in suffix trees*, in Proc. ESA 2014, 2014, pp. 455–466.
18. D. Hermelin, G. M. Landau, Y. Rabinovich, and O. Weimann: *Binary jumbled pattern matching via all-pairs shortest paths.* CoRR, abs/1401.2065 2014.
19. M. Huova and J. Karhumäki: *On the unavoidability of k-abelian squares in pure morphic words.* Journal of Integer Sequences, 16 2013, p. article 13.2.9.
20. M. Huova, J. Karhumäki, A. Saarela, and K. Saari: *Local squares, periodicity and finite automata*, in Rainbow of Computer Science - Dedicated to Hermann Maurer on the Occasion of His 70th Birthday, 2011, pp. 90–101.
21. M. Huova and A. Saarela: *Strongly k-Abelian repetitions*, in Proc. WORDS 2013, 2013, pp. 161–168.
22. J. Karhumäki, S. Puzynina, M. Rao, and M. A. Whiteland: *On cardinalities of k-abelian equivalence classes.* Theor. Comput. Sci., 658 2017, pp. 190–204.
23. J. Karhumäki, S. Puzynina, and A. Saarela: *Fine and Wilf's theorem for k-Abelian periods.* Int. J. Found. Comput. Sci., 24(7) 2013, pp. 1135–1152.
24. J. Karhumäki, A. Saarela, and L. Q. Zamboni: *On a generalization of Abelian equivalence and complexity of infinite words.* J. Comb. Theory, Ser. A, 120(8) 2013, pp. 2189–2206.
25. J. Karhumäki and M. A. Whiteland: *Regularity of k-Abelian equivalence classes of fixed cardinality*, in Adventures Between Lower Bounds and Higher Altitudes - Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday, 2018, pp. 49–62.
26. J. Kärkkäinen, P. Sanders, and S. Burkhardt: *Linear work suffix array construction.* J. ACM, 53(6) 2006, pp. 918–936.
27. D. K. Kim, J. S. Sim, H. Park, and K. Park: *Constructing suffix arrays in linear time.* J. Discrete Algorithms, 3(2-4) 2005, pp. 126–142.

28. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays.* J. Discrete Algorithms, 3(2-4) 2005, pp. 143–156.
29. T. Kociumaka, J. Radoszewski, and W. Rytter: *Efficient indexes for jumbled pattern matching with constant-sized alphabet.* Algorithmica, 77(4) 2017, pp. 1194–1215.
30. T. Kopelowitz and M. Lewenstein: *Dynamic weighted ancestors*, in Proc. SODA 2007, 2007, pp. 565–574.
31. T. M. Moosa and M. S. Rahman: *Indexing permutations for binary strings.* Inf. Process. Lett., 110(18-19) 2010, pp. 795–798.
32. T. M. Moosa and M. S. Rahman: *Sub-quadratic time and linear space data structures for permutation matching in binary strings.* J. Discrete Algorithms, 10 2012, pp. 5–9.
33. G. Nong, S. Zhang, and W. H. Chan: *Two efficient algorithms for linear time suffix array construction.* IEEE Trans. Computers, 60(10) 2011, pp. 1471–1484.
34. R. Pagh and F. F. Rodler: *Cuckoo hashing.* J. Algorithms, 51(2) 2004, pp. 122–144.
35. S. Sugimoto, N. Noda, S. Inenaga, H. Bannai, and M. Takeda: *Computing Abelian string regularities based on RLE*, in Proc. IWOCA 2017, 2017, pp. 420–431.
36. E. Ukkonen: *On-line construction of suffix trees.* Algorithmica, 14(3) 1995, pp. 249–260.
37. D. E. Willard: *Log-logarithmic worst-case range queries are possible in space $\Theta(N)$.* Inf. Process. Lett., 17(2) 1983, pp. 81–84.

Online Parameterized Dictionary Matching with One Gap

Avivit Levy and B. Riva Shalom

Department of Software Engineering
Shenkar College, Ramat Gan, Israel
avivitlevy@shenkar.ac.il, rivash@shenkar.ac.il

Abstract. We study the online Parameterized Dictionary Matching with One Gap problem (PDMOG) 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 of the patterns from D that parameterized match to suffixes of the text that has arrived so far, before the next character arrives. Two equal-length strings are a parameterized match if there exists a bijection on the alphabets, such that one string matches the other under the bijection. The gap symbols are associated with bounds determining the possible lengths of matching strings. Online Dictionary Matching with One Gap (DMOG) captures the difficulty in a bottleneck procedure for cyber-security, as many digital signatures of viruses manifest themselves as patterns with a single gap. Parameterized match captures possible encryption of the patterns. We also define and study the *strict* PDMOG problem, in which sub-patterns of the same dictionary pattern should be parameterized matched via the same bijection. This captures situations where sub-patterns of a dictionary pattern are encoded simultaneously.

Keywords: pattern matching, dictionary matching, online dictionary matching with gaps, parameterized matching

1 Introduction

Cyber security is a critical modern concern. Network intrusion detection systems (NIDS) perform protocol analysis, content searching and content matching, in order to detect harmful software. Such malware may appear on several packets, hence the need for *gapped matching* [25].

A *gapped pattern* P is one of the form $lp\ \{\alpha,\beta\}\ rp$, where each sub-pattern lp, rp 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 [8]. Consider for example the SNORT software, which is a free open source network intrusion detection system and intrusion prevention system. Analyzing the set of gapped patterns considered by the SNORT software rules shows that 77% of the patterns have at most one gap, and more than 44% of the patterns containing gaps have only one gap [7].

For this reason, Amir et al. [9,8] defined the Dictionary Matching with One Gap problem (DMOG) as follows. Preprocess a dictionary D of total size $\mathfrak{D}$ over alphabet Σ consisting of d gapped patterns each containing a single gap, so that given a query text T of length n over alphabet Σ, we can output all locations ℓ in T, where any gapped pattern ends. Note that, $\mathfrak{D}$ is the sum of lengths of all patterns in the dictionary, *not including* the gaps sizes. For example, let D be $\{P_1 = a\ b\ a\ \{2,4\}\ d\ d,$

$P_2 = a\ b\ \{2,4\}\ c\ d$, $P_3 = b\ a\ \{2,4\}\ c\}$. Then, the text $T = c\ \ d\ \ a\ \ b\ \ a\ \ b\ \ e\ \ b\ \ c\ \ d\ \ a\ \ c$ has occurrences of P_2 ending at location 10 with gap length 4 and gap length 2, and of P_3 ending at location 9 with gap length 3.

We study an extension to the dictionary matching with one gap (DMOG) problem suggested by Shalom [31], where every pattern in the dictionary has a single gap, in which the gapped malware is encrypted in order to evade virus scanners. We consider the situation where units of plain text are replaced with ciphertext according to a fixed system, i.e., a parameterized mapping is used as a strategy of encryption. Parameterized matching is a well-known problem in computer science [12]. Two equal-length strings are a parameterized match if there exists a bijection on their alphabet symbols under which one string matches the other.

The Parameterized Matching problem (PM) is formally defined as follows. Given a Text T of length n and a pattern P of length m, both over alphabet $\Sigma' \cup \Sigma$, s.t. $\Sigma' \cap \Sigma = \emptyset$, output all locations ℓ in T, where there exists a bijection $f : \Sigma \to \Sigma$ and the following hold: (1) $\forall P[i] \in \Sigma'$, $P[i] = T[\ell + i - 1]$, and (2) $\forall P[i] \in \Sigma$, $f(P[i]) = T[\ell + i - 1]$. For example, let $\Sigma' = \{a, b\}, \Sigma = \{x, y, z\}$ for text $T = x\ x\ y\ b\ z\ y\ y\ x\ b\ z\ x$ and pattern $P = z\ z\ x\ b$ there are two p-matches ending at locations $\{4, 9\}$. The former implies mapping function $f(z) = x, f(x) = y$, while the latter implies mapping function $f(z) = y, f(x) = x$. Throughout the paper we denote a parameterized match by *p-match*.

We thus study the Parameterized Dictionary Matching with One Gap (PDMOG) problem formally defined below. However, unlike Shalom [31], who studied the offline scenario where all the text is given in advance, we focus on the online setting.

Definition 1. The Parameterized Dictionary Matching with One Gap problem (PDMOG)*:*

Preprocess: *A dictionary D consisting of d gapped patterns $\{P_i\}$ over alphabet $\Sigma' \cup \Sigma$, s.t. $\Sigma' \cap \Sigma = \emptyset$, where every P_i is of the form $lp_i\{\alpha_i, \beta_i\}rp_i$ and α_i, β_i, are P_i's gap boundaries.*

Query: *A text T of length n over alphabet $\Sigma' \cup \Sigma$, $\Sigma' \cap \Sigma = \emptyset$*

Output: *All locations ℓ in T, where a p-match of gapped pattern $P_i \in D$* ends*, i.e., there exist bijections $f_1, f_2 : \Sigma \to \Sigma$ and the following hold for some P_i and a gap length $g \in [\alpha_i, \beta_i]$:*
(1) $\forall lp_i[j] \in \Sigma'$, $lp_i[j] = T[\ell - |rp_i| - g - |lp_i| + j]$.
(2) $\forall lp_i[j] \in \Sigma$, $f_1(lp_i[j]) = T[\ell - |rp_i| - g - |lp_i| + j]$.
(3) $\forall rp_i[j] \in \Sigma'$, $rp_i[j] = T[\ell - |rp_i| + j]$.
(4) $\forall rp_i[j] \in \Sigma$, $f_2(rp_i[j]) = T[\ell - |rp_i| + j]$.

Note, that the gapped pattern parts lp_i, rp_i need to be p-matched separately, hence, each can be matched using a different matching function. For example, let $\Sigma' = \{a, b\}, \Sigma = \{q, u, v, w, z\}$ for text $T = a\ u\ v\ b\ u\ b\ a\ z\ w\ w\ z$ and $D = \{P_1 = z\ x\ b\ z\{2, 4\}u\ u\ q$, $P_2 = u\ b\ q\{1, 4\}a\ u\ v\}$. We have two p-matches ending at locations $\{11, 9\}$. The first p-matches P_1 using matching function $f(z) = u, f(x) = v$ for lp_1, a gap of length 3 and a matching function $f(u) = w, f(q) = z$ for rp_1. The second p-matches P_2 using $f(u) = v, f(q) = u$ for matching lp_2, a single character gap and $f(u) = z, f(v) = w$ for matching rp_2. For simplicity, we assume that $\Sigma' = \emptyset$. Our solutions can be easily adapted to $\Sigma' \neq \emptyset$ case.

The Strict PDMOG Problem. In some situations it is more reasonable that the encodings of both sub-patterns of the same dictionary pattern are done simultaneously and, therefore, equal. Hence, we suggest another formalization of the PDMOG definition, which we call *strict* PDMOG, that enforces the requirement of both left and right sub-patterns have the same parameterized matching function. The strict PDMOG formal definition is identical to Definition 1, with the additional requirement that $f_1 = f_2$, implying that both sub-patterns of a gapped pattern are parameterized matched via the same bijection function.

In the above example, the parameterized occurrence of P_1 in the text T ending at location 11 is a strict parameterized occurrence, since the bijection $f(z) = u, f(x) = v$ for lp_1, and the bijection $f(u) = w, f(q) = z$ for rp_1 do not contain collisions, i.e., matching of the same character to different characters. In contrary, the p-matching of P_2 ending at location 9 using a matching function $f(u) = v, f(q) = u$ for lp_2 and a mapping function $f(u) = z, f(v) = w$ for rp_2 contains a collision matching the character u to two different characters in lp_2 and rp_2, and therefore, is not a strict parameterized occurrence.

Throughout the paper we use the following notations. Let $D = \{P_1, \ldots, P_d\}$, where every P_i is a gapped pattern of the form $lp_i\{\alpha_i, \beta_i\}rp_i$. We denote $\beta^* = \max_i \beta_i$, $\alpha^* = \min_i \alpha_i$. If D has uniform gap boundaries $\{\alpha, \beta\}$, then $\forall 1 \leq i \leq d$, $\alpha_i = \alpha$, $\beta_i = \beta$.

1.1 Related Previous Work and the Current Work

Dictionary Matching with Gaps. Dictionary matching has been amply researched (see e.g. [1,2,3,4,6,15]). The problem definition varies and many parameters affect the complexity when patterns are gapped. [30] [14] and [13] solve the problem, yet their solutions include a factor of *socc* – the total number of occurrences of the sub-patterns in the text, which can be very large. Others [26,33] solve the problem of matching a set of patterns with variable length of don't cares, yet, they report only a leftmost occurrence of a pattern if there exists one. In [21] an online algorithm for the problem is given, however, at most one occurrence for each pattern at each text position is reported.

The first results on the DMOG problem are due to [9], which solved the offline DMOG problem for a single set of gap boundaries reporting all appearances of all gapped patterns. They suggest an algorithm using range queries and an additional algorithm using a look-up table. The solution is generalized to variable-length gaps dictionaries achieving linear space in [22]. Finally, the online DMOG problem is considered in [8,7] and a connection to the 3SUM conjecture is shown. The *conditional lower bound* (CLB) provides insight for the inherent difficulty in DMOG, and reveals that the CLB from the 3SUM conjecture can be phrased in terms of a new parameter of the problem – $\delta(G_D)$, where G_D is a graph representing the input dictionary. $\delta(G_D)$ turns out to be a small constant in some input instances considered by NIDS. In fact, $\delta(G_D)$ is not greater than 5 in the graph created using SNORT software rules [7]. This leads to designing algorithms whose runtime can be expressed in terms of $\delta(G_D)$, and can therefore be helpful in such practical settings. Online Dictionary Recognition with One Gap (DROG), where each gapped pattern is reported at most once during the entire online text scan, is considered in [10].

Parameterized Matching. The problem was initially defined as a tool for software maintenance, motivated by the observation that programmers introduce duplicate code into large software systems when they add new features or fix bugs, thus slightly modify the duplicated sections [12]. The problem has many application in various fields, such as Image processing, where parameterized matching can help searching an icon on the screen, or improving ergonomy of databases of URLS [28] and extensive research followed (see [27,28]). Among the extensions are: suggesting a parameterized version of KMP [5], studies of maximal p-matches over a threshold length and a p-suffix tree [11,12], parameterized fixed and dynamic dictionary problems presented [23] and improved by [20], efficient parameterized text indexing [18], p-suffix arrays [17], parameterized LCS [24], and many more.

Parameterized Dictionary Matching with One Gap (PDMOG). Shalom [31] first formalized the extension of the dictionary matching with one gap (DMOG) problem to parameterized dictionary matching with one gap (PDMOG), in which the gapped malware is encrypted in order to evade virus scanners. [31] study the *offline* scenario where all the text is given in advance, and give two solutions. The first solves offline PDMOG for dictionaries with non-uniform gap boundaries with $O(n(\beta^* - \alpha^*)\log^2 d + occ)$ query time, where n is the size of the text, d is the number of gapped patterns in the dictionary, $\beta^* - \alpha^*$ is the maximal gap size and occ is the number of the gapped patterns reported as output. The second offline PDMOG solution is for dictionaries with uniform gap boundaries with $O(n(\beta - \alpha) + occ)$ query time, where n is the size of the text, $\beta - \alpha$ is the gap size and occ is the output size.

This Paper Contributions. In this paper, we focus on the *online* setting of PDMOG and *strict* PDMOG, where the text arrives a character at a time, and the requirement is to report all gapped patterns that parameterize-match to suffixes of the text that has arrived so far, before the next character arrives. This is the more realistic situation in NIDS applications. The main contributions of this paper are:

- Formalizing the online PDMOG and strict PDMOG problems, which are natural extensions to the online DMOG problem.

- Obtaining algorithms for online PDMOG that are fast for some practical inputs. A basic property of suffixes of any dictionary pattern is that they form a chain where each is a proper suffix of the other. This property, that was crucial in the online DMOG solutions [8,7], no longer holds for parameterized suffixes. Nevertheless, we show that it is possible to by-pass this difficulty.

- Obtaining algorithms for online strict PDMOG that are fast for some practical inputs where dictionary sub-patterns contain the same alphabet symbols. Enforcing the requirement that both left and right sub-patterns of a dictionary pattern are p-matched using the same parameterized matching function necessitates representation and maintenance of these functions.

Paper Organization. The paper is organized as follows. Section 2 describes our results for the online PDMOG problem. Section 3 details the algorithms for solving online strict PDMOG problem. Section 4 concludes the paper and poses some open questions.

2 Solving Online PDMOG

In this section we describe the online PDMOG solution. We detail the changes and modifications that should be made to the basic scheme of [8,7] in order to adapt it to our problem.

The Bipartite Graph G_D. We use [8,7] dictionary representation as a graph $G_D = (V, E)$: sub-patterns are represented by vertices and there is an edge $(u, v) \in E$ if and only if there is a pattern $P_i \in D$, where lp_i is associated with node u and rp_i is associated with v. The graph $G_D = (V, E)$ is converted to a bipartite graph by creating two copies of V called L (left vertices) and R (right vertices) as follows. 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.

P-Matches Detection. Parameterized matching does not require exact matches between the characters, but rather to capture the characters order. Therefore, Baker [12] defined a *p-string* over a string $S = s_1, s_2 \cdots$ using the *prev* function, where $prev(s_i) = s_i$ in case $s_i \in \Sigma'$, but for $s_i \in \Sigma$, $prev(s_i) = 0$ if s_i is the leftmost position in S of s_i, and $prev(s_i) = i - k$ if k is the previous position to the left at which s_i occurs. For example, let $\Sigma' = \{a, b\}$, $\Sigma = \{u, v\}$ and $S = a\ b\ u\ v\ a\ b\ u\ v\ u$, then the p-string of S is $prev(S) = a\ b\ 0\ 0\ a\ b\ 4\ 4\ 2$.

Lemma 2. [12] *Strings S_1, S_2 have $prev(S_1) = prev(S_2)$ if and only if they are p-matched.*

[23] construct a modified Aho-Corasick automaton (AC) [1] suitable for p-strings similarly to the original AC construction, with modifications to goto and fail links adapting it to work with p-strings. Their automaton is constructed in time $O(\mathfrak{D} \log |\Sigma|)$, takes $O(m \log m) = O(\mathfrak{D} \log \mathfrak{D})$ bits, where m is the number of automaton states, and reports all p-matches of dictionary D patterns in text T in $O(|T| \log |\Sigma| + occ)$ time, where occ is the number of reported occurrences. If only the longest pattern located for each text location is reported, the query is answered in $O(|T| \log |\Sigma|)$.[1]

In the preprocessing, we calculate in linear time the p-string, $prev(lp_i)$, $prev(rp_j)$ for every lp_i, rp_j of some $P_i, P_j \in D$, and construct a parameterized AC automaton upon them, denoted by pAC. Using a standard binary encoding technique each character costs $O(\log |\Sigma|)$ worst-case time. However, for simplicity of exposition, $|\Sigma|$ is assumed to be constant, whenever this assumption is not critical, i.e, if $|\Sigma|$ only appears as a logarithmic factor due to this binary encoding. Note that, the *prev* function does not preserve the suffix relation of the strings it is applied to. Consider sub-patterns x, y, where x is a suffix of y, then, $prev(x)$ is not necessarily a suffix of $prev(y)$. For example, consider $lp_i = uuua$ and its suffix uua. It holds that $prev(lp_i) = 011a$, yet $prev(uua) = 01a$, which is not a suffix of $011a$. Nevertheless, p-suffixes of all dictionary sub-patterns can be traced using fail links of pAC automaton.

The pAC automaton consists of states representing p-strings of prefixes of dictionary sub-patterns. A state representing *prev* function of a sub-pattern lp_i or rp_j is called an accepting state. An arriving character may correspond to several arriving

[1] [19] suggest a space efficient data structure for the parameterized dictionary matching, improving the pAC automaton of [23] by using sparsification technique. Due to our cyber security motivation, we prefer the data structure of [23] for its faster query time.

parameterized sub-patterns, since *prev* function of a sub-pattern could be a proper p-suffix of *prev* function of another sub-pattern. We therefore, phrase complexities in terms of $plsc$ – the maximum number of sub-patterns that their prev functions are p-suffixes of each other, implying vertices in the bipartite graph that arrive due to a character arrival. A similar (probably smaller) lsc factor was used in [8,7] DMOG solutions, and even for simplified DMOG relaxations [21]. While lsc (and $plsc$) could theoretically be as large as d, in many practical situations it is very small. A graph created using SNORT software rules has lsc not greater than 5 [7]. In natural languages dictionaries such as the English dictionary lsc is also a small constant. While it is possible to find a suffix chain of English words with length 7, it is difficult (if possible) to find chains of greater length.

At each time unit, at most $plsc$ vertices are handled, as follows. A vertex $u \in L$, representing a longest sub-pattern associated with the current accepting pAC automaton state, is handled. Other (not necessarily proper) p-suffixes of that sub-pattern are also handled. The preprocessing enabling this procedure uses a p-graph structure.

The P-Graph Structure. We construct a graph pG among the sub-patterns associated with vertices of G_D, where an edge $(u', u) \in E(pG)$ if and only if the sub-pattern associated with u' is a p-suffix of the sub-pattern associated with u. An additional *end* vertex corresponding to the empty string is added to the graph pG, since it is a p-suffix of every sub-pattern. The graph pG can be constructed in linear time while constructing the pAC automaton of D. The bipartite graph G_D vertices arriving due to a text character arrival correspond to vertices on a BFS scan of pG from a vertex u associated with the pAC accepting state (one of the longest sub-patterns having the same p-suffix recognized by this state), creating a BFS-tree rooted at u, not including the *end* leaves of the BFS-tree.

Text Scan. This phase online detects *p-matching* sub-patterns in the text, while saving in adequate data structures occurrences of sub-patterns represented by $u \in L$ nodes that where located during the proper bounds of time units ago (which are calculated differently for the uniform/non-uniform gap bounds cases). When a sub-pattern represented by a $v \in R$ node is p-matched, parameterized occurrences of all gapped patterns P_i where rp_i is represented by $v \in R$ and lp_i is represented by $u \in L$ saved in the data structures, are reported. During text scan phase, $prev(T)$ is calculated online using a $|\Sigma|$-sized array preserving for each $\sigma \in \Sigma$ its last occurrence. Scanning $prev(T)$ using pAC enables finding all sub-patterns p-matching $T[1..\ell]$ *ending* at ℓ. Note, that even for non-fixed alphabets, calculating $prev(T)$ requires $O(|T|)$ time by using perfect hash tables for latest occurrence position of a character in T. Due to synchronization reasons described in [8], removal from the data structures of vertices u that become non-relevant is delayed by $M - 1$ time units, where M is the length of the longest sub-pattern corresponding to a vertex in R.

2.1 PDMOG via Graph Orientations

Graph Orientation. As in [8,7], the graph G_D is preprocessed using linear time greedy algorithm suggested by Chiba and Nishizeki [16] to obtain a $\delta(G_D)$-orientation of the graph G_D, where every vertex has out-degree at most $\delta(G_D) \geq 1$. Orientation is viewed as assigning "responsibility" for data transfers occurring on an edge to one of its endpoints, depending on the direction of the edge in the orientation. If an edge $e = (u, v)$ is oriented from u to v, 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 a 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. Hence, 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.

The construction and use of the data structures in the algorithms is done as in [8,7], except for the use of the auxiliary pG for recognizing the actual arriving vertices when an accepting state of pAC is reached. This gives Theorem 3.

Theorem 3. *1. The online PDMOG problem with uniformly bounded gap borders can be solved in: $O(\mathfrak{D} \log |\Sigma|)$ preprocessing time, $O(\delta(G_D) \cdot plsc + pocc)$ time per text character, where pocc is the number of parameterized patterns reported due to character arrival, and $O(\mathfrak{D} + plsc \cdot (\beta + M))$ space.*

2. The online PDMOG problem with non-uniformly bounded gap borders can be solved in: $O(\mathfrak{D} \log |\Sigma|)$ preprocessing time, $\tilde{O}(plsc \cdot \delta(G_D) + pocc)$ time per text character, where pocc is the number of parameterized patterns reported due to character arrival, and $O(\mathfrak{D} \log |\Sigma| + plsc \cdot \delta(G_D)(\beta^* - \alpha^* + M) + plsc \cdot \alpha^*)$ space.

2.2 PDMOG via Threshold Orientations

Subsection 2.1 focuses on orientations whose out-degree is bounded by $\delta(G_D)$. Thus, when $\delta(G_D) = \sqrt{d}$ the PDMOG algorithms basically take $O(plsc \cdot \sqrt{d})$ time. In the non-uniform case the degeneracy can be much larger, since the same sub-patterns can represent different gapped patterns if they have different gaps boundaries, thus two vertices can be connected by more than one edge. Moreover, the *plsc* factor maybe larger than the *lsc* factor used in DMOG solutions. Therefore, in this subsection we reduce the factor of $plsc \cdot \delta(G_D)$ to $\sqrt{plsc \cdot d}$, by using a different graph orientation method, referred to as a *threshold* orientation.

Definition 4. *A vertex in G_D is* heavy *if it has more than $\sqrt{d/plsc}$ neighbors, and* light *otherwise.*

Two key properties are used: light vertices have at most $\sqrt{d/plsc}$ neighbors, and the number of heavy vertices is less than $\sqrt{plsc \cdot d}$. We orient all edges that touch a light vertex to leave that vertex, breaking ties arbitrarily if both vertices are light. Thus, every edge e connecting a light vertex with a light/heavy vertex, the light vertex is the responsible-neighbor, and the heavy vertex, if exists in e, is the assigned-neighbor. We handle differently edges with at most one heavy vertex as an endpoint and edges connecting two heavy vertices.

Edges Connecting at Most One Heavy Vertex. Data structures used for dealing with edges where at most one of its endpoints is heavy when considering uniformly bounded gaps, are as in Subsection 2.1 in uniformly bounded gaps (ordered reporting lists $\mathcal{L}_v$ for each $v \in R$, ordered lists τ_u of the time stamps for each $u \in L$ and the list $\mathcal{L}_\beta$ of the last $\beta + M$ vertices $u \in L$). Data structures used for dealing with edges where at most one of their endpoints is heavy when considering non-uniformly bounded gaps, are as in Subsection 2.1 in non-uniformly bounded gaps (Range query data structures S_v for each $v \in R$, ordered lists τ_u of the time stamps for each $u \in L$ and the list $\mathcal{L}_{\beta^*}$ of the last $\beta + M$ vertices $u \in L$).

Edges Connecting two Heavy vertices. The set of heavy vertices is less than $\sqrt{plsc \cdot d}$, and so even if the number of vertices from L arriving at the same time can be as large as *plsc* and the set of their neighbors can be very large, the number of vertices in R is still less than $\sqrt{plsc \cdot d}$. Thus, using a batched scan on all of R keeps the time cost low, after some preprocessing. In addition, at each time unit, we handle only a single $u \in L$ currently arriving, one representing a longest sub-pattern found by the *pAC* automaton at that time, which is a sub-pattern associated with the current accepting state. Other sub-patterns, which are (not necessarily proper) p-suffixes of that sub-pattern are handled implicitly, without increasing the time complexity unless they are reported. The preprocessing that enables this procedure uses a p-heavy-graph structure *phG* (replacing the tree-structure T used in [8,7]).

The p-heavy-Graph Structure. A graph *phG* among the sub-patterns associated with *heavy* vertices from L is constructed, where an edge $(u', u) \in E(phG)$ if and only if the sub-pattern associated with u' is a p-suffix of the sub-pattern associated with u. An additional *end* vertex corresponding to the empty string is added to the graph *phG*, since it is a p-suffix of every sub-pattern. The graph *phG* can be constructed in linear time during the construction of *pG*. The bipartite graph G_D *heavy* vertices arriving due to a text character arrival correspond to vertices on a BFS scan of *phG* from some vertex u, creating an $O(plsc)$-size BFS-tree rooted at u, not including the *end* leaves of the BFS-tree.

The construction and use of the data structures in the algorithms is done as in [8,7] except for the replacement of T by *phG* and the use of the auxiliary *pG* for recognizing the actual arriving vertices when an accepting state of *pAC* is reached. This gives Theorem 5.

Theorem 5. *1. The online PDMOG problem with uniform gap borders can be solved in $O(\mathfrak{D} \log |\Sigma|)$ preprocessing time, $O(plsc + \sqrt{plsc \cdot d} + pocc)$ time per text character, and $O(\mathfrak{D} + plsc(\beta + M))$ space.*

2. The online PDMOG problem with non-uniform gap borders can be solved in $\tilde{O}(\mathfrak{D} + d(\beta^* - \alpha^*))$ preprocessing time, $\tilde{O}(plsc + \sqrt{plsc \cdot d}(\beta^* - \alpha^* + M) + pocc)$ time per text character, and $O(\mathfrak{D} \log |\Sigma| + d(\beta^* - \alpha^*) + \sqrt{plsc \cdot d}(\beta^* + M))$ space.

3 Solving Online Strict PDMOG

In this section we study online strict PDMOG problem, requiring both sub-patterns of a gapped pattern to be p-matched via the same function. We solve the problem for dictionaries where every sub-pattern contains all characters of Σ. The basic algorithmic scheme is as Section 2, however, there are now additional issues to handle due to the new requirement.

The Matching Permutation. The *pAC* automaton reports an occurrence of parameterized match of sub-patterns in the text, yet it does not report the matching function used. We define the *matching permutation*, $\pi_{u,t}$, via which the current p-matching of sub-pattern represented by node u to the suffix ending at time t in the text occurred. $\pi_{u,t}$ can be represented as a $|\Sigma|$-length array, where entry i contains σ' if $f(\sigma_i) = \sigma'$ for the current p-match. Hence, we can save at every step the last M symbols of the text, where M is the length of the longest sub-pattern in the dictionary. $\pi_{lp_i,t}$ of a sub-pattern lp_i can be calculated in $|\Sigma_{lp_i}|$ time, where Σ_{lp_i} contains all distinct symbols that appear in lp_i.

3.1 Uniformly Bounded Gaps

The Permutation Tree. Matching permutations are saved in a *permutation tree* data structure. The permutation tree T_u maintains a set S of permutations of Σ used to p-match occurrences of sub-pattern associated with node u. A permutation tree can be basically maintained as a y-fast trie [32] with additional linked lists in its leaves.

The data structures used in this case are:

1. For each $u \in L$ we save **a y-fast trie** T_u containing all matching permutations $\pi_{u,t}$ via which the sub-pattern represented by u was p-matched to the text ending at location t, at least α and at most β time units ago. For every node l in T_u, representing the matching permutation π_l, we save an ordered list τ_{u,π_l} of time stamps of the occurrences of u p-matching the text via π_l.
2. For each vertex $v \in R$, we save **a y-fast trie** T_v containing all matching permutations $\pi_{u,t}$ via which the sub-pattern represented by u was matched to the text ending at location t, at least α and at most β time units ago, where u is **a responsible-neighbours of** v. For every node l' in T_v, representing a permutation $\pi_{l'}$, we save an ordered list $\mathcal{L}_{v,\pi'_l}$ of links to the nodes representing permutation $\pi_{l'}$ in trees T_u, where u is a responsible neighbour of v.
3. The **list** $\mathcal{L}_\beta$ of delayed vertices $u \in L$ for at least α time units before they are considered. To each node u in $\mathcal{L}_\beta$ we attach the matching permutation $\pi_{u,t}$ via which the sub-pattern represented by u was p-matched to the text ending at time stamp t.

The details of the construction and use of the data structures in the algorithms are as follows.

When the *pAC* automaton reaches state s in time t, the data structures of the vertices are updated accordingly, as follows.
For every vertex v associated with a sub-pattern that its *prev* function is a p-suffix of the *prev* function of the sub-pattern represented by state s,

```
1. If the arrived vertex is $v \in R$ that was p-matched to the text via $\pi_{v,t}$,
   (a) Search the matching permutation $\pi_{v,t}$ in $T_v$.
   (b) if $\pi_{v,t}$ appears in $T_v$ at node $l'$,
       Let $l^* = \mathcal{L}_{v,\pi_{l'}}.first$ (a link to a node $l \in T_u$, where $\pi_l = \pi_{l'}$).
       i. Let $t' = \tau_{u,\pi_l}.first$
       ii. while $t' \geq t - m_v - \beta - 1$
           A. Report edge $(u, v)$ with matching permutation $\pi_{v,t}$, where the nodes $u$,
              $v$ appearances are at locations $t'$, $t$.
           B. If all appearances of the gapped pattern associated with edge $(u, v)$ are
              required, continue the scan of $t'$ elements of $\tau_{u,\pi_l}$ while $t' \geq t-m_v-\beta-1$.
           C. Let $l^*$ be the next link in the list $\mathcal{L}_{v,\pi_{l'}}$.
   (c) For every vertex $u$ which $v$ is its responsible-neighbour.
       i. If $T_u$ contains a node $l$ where $\pi_l = \pi_{v,t}$, let $t' = \tau_{u,\pi_l}.first$
```

   ii. If $t - m_v - \beta - 1 \leq t'$, report edge (u, v) with matching permutation $\pi_{v,t}$ where the nodes u, v appearances are at locations t', t.
   iii. If all appearances of the gapped pattern associated with edge (u, v) are required, continue the scan of the elements of τ_{u,π_l} while $t - m_v - \beta - 1 \leq t'$ where t' is the next element in τ_{u,π_l}.
2. If the arrived vertex is $u \in L$, $(u, \pi_{u,t})$ is inserted into $\mathcal{L}_\beta$.

In addition, the data structures are updated by perviously arrived nodes $u \in L$ saved in $\mathcal{L}_\beta$ that have become relevant.
For vertices $u \in L$, where $(u, \pi_{u,t-\alpha-1})$ was inserted into $\mathcal{L}_\beta$, $\alpha + 1$ time units before time t,

1. Search for node l, representing the matching permutation $\pi_{u,t-\alpha-1}$ in T_u.
2. If $\pi_{u,t-\alpha-1}$ is not saved in T_u, insert a new node l representing $\pi_{u,t-\alpha-1}$ to T_u.
3. For every $v \in R$ that is an assigned neighbour of u,
   (a) Search for node l', representing the matching permutation $\pi_{u,t-\alpha-1}$ in T_v,
   (b) If $\pi_{u,t-\alpha-1}$ is not saved in T_v, insert a new node l' representing $\pi_{u,t-\alpha-1}$ to T_v.
   (c) Add to the beginning of $\mathcal{L}_{v,\pi_{l'}}$ a link to the node l in T_u, where $\pi_l = \pi_{l'}$.
   (d) If τ_{u,π_l} is not empty, remove the previous link to node l of T_u, from $\mathcal{L}_{v,\pi_{l'}}$.
4. The time stamp $t - \alpha - 1$ is added to the beginning of τ_{u,π_l} saved for the node l in T_u.

For vertices $u \in L$,where $(u, \pi_{u,t-\beta-M-1})$ was inserted into $\mathcal{L}_\beta$, exactly $\beta + M + 1$ time units before time t,

1. $(u, \pi_{u,t-\beta-M-1})$ is removed from $\mathcal{L}_\beta$.
2. Search T_u for node l representing the matching permutation $\pi_{u,t-\beta-M-1}$.
3. The time stamp $t - \beta - M - 1$ is removed from the end of the listτ_{u,π_l}, attached to node l.
4. If τ_{u,π_l} becomes empty,
   (a) The node l is deleted from T_u.
   (b) For every T_v, where v is an assigned neighbour of u, the link to node $l \in T_u$ (for $\pi_l = \pi_{u,t-\beta-M-1}$), is removed from the end of $\mathcal{L}_{v,\pi_{l'}}$ where $l' \in T_v$ and $\pi_{l'} = \pi_{u,t-\beta-M-1}$.

This gives Theorem 6.

Theorem 6. *The online strict PDMOG problem with uniformly bounded gap borders can be solved in: $O(\mathfrak{D} \log |\Sigma|)$ preprocessing time, $O(plsc \cdot \delta(G_D) \log(|\Sigma| \log |\Sigma|) + pocc)$ time per text character, and $O(\mathfrak{D} \log \mathfrak{D} + \delta(G_D) \cdot plsc \cdot |\Sigma|(\beta - \alpha + M) + plsc \cdot \alpha)$ space.*

Proof. In preprocessing, the *pAC* automaton is built in time $O(\mathfrak{D} \log |\Sigma|)$. The query algorithm scans the text *prev* function using *pAC* in $O(|T| \log |\Sigma|)$. At time/location t, the vertex representing the sub-pattern *prev* function is recognized by *pAC*, and all its possible $O(plsc)$ vertices representing p-suffixes sub-patterns in the dictionary. Every such vertex requires $O(\delta(G_D))$ operations on y-fast trie (search - when $\pi_{v,t}$ of $v \in R$ is searched in T_v as well as in all the T_u of its assigned neighbours $u \in L$, insert - when $\pi_{u,t}$ of $u \in L$ is inserted into T_u as well as a link to the node representing $\pi_{u,t}$ inserted to all the T_v of its assigned neighbours $v \in R$, delete - when $\pi_{u,t}$ of $u \in L$ is deleted from all the T_v of its assigned neighbours $v \in R$, and from T_u when the p-matches of u become irrelevant.) Search, insert and delete operations applied to a y-fast trie, require $O(\log(|\Sigma|\log|\Sigma|))$ time each, as the number of possible matching permutations is $O(|\Sigma|^{|\Sigma|})$ and each operation on a y-fast trie costs $O(\log\log U)$, where U is the maximum value of an element [32].

Reporting the output: A vertex $v \in R$ that was p-matched via permutation $\pi_{v,t}$ scans the list $\mathcal{L}_{v,\pi_{l'}}$ of node $l' \in T_v$ representing permutation $\pi_{v,t}$, where each element in the list is a link to a node $l \in T_u$, representing permutation $\pi_{v,t}$ for some responsible neighbour u of v. If τ_{u,π_l} of node l is scanned, each time stamp represents additional parameterized occurrence of u. $\mathcal{L}_{v,\pi_{l'}}$ scan terminates when a link to τ_{u,π_l} is reached, where the gap between the newest time stamp of τ_{u,π_l} and $t - m_v + 1$ (the current v occurrence beginning) is larger than β, as the rest of the elements of $\mathcal{L}_{v,\pi_{l'}}$ are older. Hence, each considered element in $\mathcal{L}_{v,\pi_{l'}}$, except the last one, is a parameterized occurrence. In addition, each scanned element in the τ_{u,π_l} lists of nodes in T_u trees, where u is an assigned-neighbor of v and $\pi_l = \pi_{v,t}$, is reported.

Regarding space: The pAC automaton requires $O(\mathfrak{D}\log\mathfrak{D})$ space. Each T_u maintains distinct permutations of all p-matches of u in T at the last $\beta - \alpha + M$ locations. Since at every time there are at most $plsc$ p-matches of sub-patterns simultaneously, all T_u for $u \in L$ have at most $plsc \cdot (\beta - \alpha + M)$ nodes and the same total time stamps number. For each T_u node, the saved matching permutations represented by it require $|\Sigma|$ space. Hence, the total T_u trees size is $O(plsc \cdot |\Sigma| \cdot (\beta - \alpha + M))$. The space of all permutation trees T_v, $v \in R$, is $O(\delta(G_D) \cdot plsc \cdot |\Sigma|(\beta - \alpha + M))$, since a single $l \in T_u$ can be linked to $\delta(G_D)$ leaves of T_v, where u is a responsible-neighbor of v. Additional $O(plsc \cdot \alpha)$ is required for the $u \in L$ vertices maintained by $\mathcal{L}_\beta$ for α time units until they are considered as arrived.

3.2 Non-Uniformly Bounded Gaps

Non-uniformly bounded gapped patterns yield a multi-graph, where each edge $e = (u, v)$ has its own boundaries $\{\alpha_e, \beta_e\}$, as (u, v) with boundaries $[3, 5]$ is a distinct edge from (u, v) with boundaries $[4, 10]$. A framework similar to Subsection 3.1 is used, yet using permutation trees is not efficient as the information saved at the leaves of the trees should be checked and not necessarily be reported, due to the different boundaries of the edges. Fortunately, we can overcome this by exploiting an alphabetical ordering of the permutations which maps each permutation π into a unique number $num(\pi)$ in $O(|\Sigma|)$ time. Thus, a fully dynamic data structure S_v supporting 6-sided 3-dimensional orthogonal range reporting queries, is used (instead of the 4-sided 2-dimensional S_v in Subsection 2.1) for saving occurrences of responsible neighbour of v. Similar structures for 2-dimensional orthogonal range reporting queries are used for the time stamps of the occurrences of $u \in L$ nodes.

The data structures used in this case are:

1. For each vertex $v \in R$, **a data structure** S_v maintaining points from $\mathbb{R}^3$, representing all time intervals in which an occurrence of v implies a gapped pattern occurrence due to a previous occurrence of u, a responsible-neighbour of v, if the matching permutations of u and v are the same.
2. For each $u \in L$ we save **a data structure** S_u maintaining points from $\mathbb{R}^2$ representing time stamps t of the occurrences of u with the number of the matching permutation $\pi_{u,t}$.
3. The **list** $\mathcal{L}_{\beta^*}$ of the last $\beta^* + M$ vertices $u \in L$ with their matching permutation. They are delayed for at least α^* time units before they are considered.

The details of the construction and use of the data structures in the algorithms are as follows.

When the *pAC* automaton reaches state s in time t, the data structures of the vertices are updated accordingly, as follows.
For every vertex v associated with a sub-pattern that its *prev* function is a p-suffix of the *prev* function of the sub-pattern represented by state s,

1. If the arrived vertex is $v \in R$, that was p-matched to the text via $\pi_{v,t}$,
   (a) A range query of $[0, t - m_v + 1] \times [t - m_v + 1, \infty] \times [num(\pi_{v,t}), num(\pi_{v,t})]$ is performed over S_v.
   (b) The edges representing the range output are reported.
   (c) For every vertex u which v is its responsible-neighbour, such that $e = (u, v)$,
      i. A range query of $[t - m_v - \beta_e, t - m_v - \alpha_e] \times [num(\pi_{v,t}), num(\pi_{v,t})]$ is performed over S_u.
      ii. The edges representing the range output are reported.
2. If the arrived vertex is $u \in L$, $(u, \pi_{u,t})$ is inserted into $\mathcal{L}_{\beta^*}$.

In addition, the *active window* is maintained by updating $\mathcal{L}_{\beta^*}$ and acknowledging arrived nodes $u \in L$ that have become relevant.
For vertices $u \in L$ where $(u, \pi_{u,t-\alpha^*-1})$ was inserted into $\mathcal{L}_{\beta^*}$, exactly $\alpha^* + 1$ time units before time t,

1. The point $(t - \alpha^* - 1, num(\pi_{u,t-\alpha^*-1})$ is inserted to S_u.
2. For every $v \in R$ that is an assigned neighbour of u, such that $e = (u, v)$, the point $(t - \alpha^* + \alpha_e, t - \alpha^* + \beta_e, num(\pi_{u,t-\alpha^*-1}))$ is inserted to S_v.

For vertices $u \in L$, where $(u, \pi_{u,t-\beta^*-M-1})$ was inserted into $\mathcal{L}_{\beta^*}$, exactly $\beta^* + M + 1$ time units before time t,

1. $(u, \pi_{u,t-\beta-M-1})$ is removed from the end of $\mathcal{L}_{\beta^*}$.
2. The point $(t - \beta - M - 1, num(\pi_{u,t-\beta^*-M-1}))$ is removed from S_u.
3. For every v that is an assigned neighbour of u, such that $e = (u, v)$, the point $(t - \beta^* - M + \alpha_e, t - \beta^* - M + \beta_e, num(\pi_{u,t-\beta-M-1}))$ is removed from S_v.

This gives Theorem 7.

Theorem 7. *The online strict PDMOG problem with non-uniformly bounded gap borders can be solved in:* $\tilde{O}(\mathfrak{D})$ *preprocessing time,* $\tilde{O}(plsc \cdot \delta(G_D) + pocc)$ *time per text character, and* $\tilde{O}(\mathfrak{D} + plsc \cdot \delta(G_D)(\beta^* - \alpha^* + M) + plsc \cdot \alpha^*)$ *space.*

Proof. Preprocessing is similar to the uniformly bounded gaps case, detailed in 3.1. The query algorithm scans the text *prev* function using *pAC*. At each time/location t, the vertex representing the sub-pattern *prev* function recognized by *pAC* at t and all its possible $O(plsc)$ p-suffixes, are considered and their data structures are updates or queried. To implement the data structures, we use Mortensen's data structure [29] supporting the set of $|S|$ points from $\mathbb{R}^d$ with $O(|S| \log^{d-2+\epsilon} |S|)$ words of space, insertion and deletion time of $O(\log^{d-2+\epsilon} |S|)$ and $O((\frac{\log |S|}{\log \log |S|})^{d-1} + op)$ time for range reporting queries on S, where op is the output size.

When a vertex $u \in L$ arrives, i.e., it was p-matched via the matching permutation $\pi_{u,t}$ in time t, the point $(t, num(p_{u,t}))$ is inserted to S_u in $O(\log^{\epsilon} |S_u|)$ time. In addition, for each assigned neighbour v of u, where $e = (u, v)$, the point $(t + \alpha_e + m_v, t + \beta_e + m_v, num(\pi_{u,t}))$, where m_v is the length of the sub-pattern represented by node v, is inserted into S_v. This insertion requires $O(\log^{1+\epsilon}|S_v|)$ time, yielding the time requires for the $u \in L$ nodes is $O(plsc\cdot(\log^{\epsilon} plsc(\beta^*-\alpha^*+M)+\delta(G_D)\log^{1+\epsilon} plsc(\beta^*-\alpha^*+M)))$.

When a vertex $v \in R$ arrives at time t, a range query $[0,t] \times [t,\infty] \times [num(\pi_{v,t}), num(\pi_{v,t})]$ over S_v returns the points that have (x,y,p) coordinates in the given range, thus a parameterized appearance. The range query is applied to S_v containing at most $plsc(\beta^*-\alpha^*+M)$ points, thus requires $O(\frac{\log^2(plsc\cdot(\beta^*-\alpha^*+M))}{\log\log^2(plsc(\beta^*-\alpha^*+M))}+pocc)$ time. Additional time is required for considering all the u assigned-neighbors of v and applying range query $[t-m_v-\beta_e, t-m_v-\alpha_e] \times [num(\pi_{v,t}), num(\pi_{v,t})]$ on the S_u structures in order to report all occurrences sharing the same matching permutations within the gaps boundaries. The total number of time stamps saved in all T_u trees is $O(plsc(\beta^*-\alpha^*+M))$, thus, the total time required for the $v \in R$ nodes is $O(plsc \cdot (\delta(G_D) \cdot \frac{\log plsc(\beta^*-\alpha^*+M)}{\log\log plsc(\beta^*-\alpha^*+M)} + \frac{\log^2 plsc(\beta^*-\alpha^*+M)}{\log\log^2 plsc(\beta^*-\alpha^*+M)} + pocc)$.

Regarding space: The pAC requires $O(\mathfrak{D}\log\mathfrak{D})$ space. Each S_u maintains all appearances u in the text, at the last $\beta^*-\alpha^*+M$ locations in the text. Hence, all S_u for $u \in L$ have at most $plsc(\beta^*-\alpha^*+M)$ points, thus the space required for them is $O(plsc(\beta^*-\alpha^*+M)\log^{\epsilon} plsc(\beta^*-\alpha^*+M))$. S_v contains points only from its responsible neighbor, thus, each of the $plsc$ vertices that were located at each of the last $\beta^*-\alpha^*+M$ locations in the text can be inserted to $\delta(G_D)$ structures of S_v, yielding the space of all the S_v lists is $O(plsc\cdot\delta(G_D)(\beta^*-\alpha^*+M)\log^{1+\epsilon} plsc\cdot\delta(G_D)(\beta^*-\alpha^*+M))$. The additional space usage is required for the $O(plsc)$ vertices maintained for $O(\alpha^*)$ time units by $\mathcal{L}_{\beta^*}$ until they can be considered as arrived.

4 Conclusion and Open Problems

We presented the online PDMOG and strict PDMOG problems and described efficient algorithms for their solution in some practical inputs. As demonstrated in this paper, online PDMOG and strict PDMOG pose additional challenges and difficulties to overcome while designing algorithms for their solutions. It is an open question whether there exist better solutions or efficient solutions for other practical inputs. Additional open research direction is to consider other types of encryption instead of parameterized mapping.

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 and G. Călinescu: *Alphabet-independent and scaled dictionary matching.* J. Algorithms, 36(1) 2000, pp. 34–62.
3. A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park: *Dynamic dictionary matching.* J. Comput. Syst. Sci., 49(2) 1994, pp. 208–222.
4. 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.
5. A. Amir, M. Farach, and S. Muthukrishnan: *Alphabet dependence in parameterized matching.* Inf. Process. Lett., 49(3) 1994, pp. 111–115.
6. 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.
7. A. Amir, T. Kopelowitz, A. Levy, S. Pettie, E. Porat, and B. R. Shalom: *Mind the gap! online dictionary matching with one gap.* Algorithmica, 2019.

8. 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, Sydney, Australia, December 12-14, 2016, pp. 12:1–12:12.
9. A. Amir, A. Levy, E. Porat, and B. R. Shalom: *Dictionary matching with a few gaps.* Theor. Comput. Sci., 589 2015, pp. 34–46.
10. A. Amir, A. Levy, E. Porat, and B. R. Shalom: *Online recognition of dictionary with one gap*, in Proceedings of the Prague Stringology Conference (PSC), Prague, Czech Republic, August 28-30, 2017, pp. 3–17.
11. B. S. Baker: *Parameterized duplication in strings: Algorithms and an application to software maintenance.* SIAM J. Comput., 26(5) 1997, pp. 1343–1362.
12. B. S. Baker: *A theory of parameterized pattern matching: algorithms and applications*, in Proceedings of the 25th Annual ACM Symposium on Theory of Computing, San Diego, CA, USA, May 16-18, 1993, pp. 71–80.
13. 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.
14. P. Bille and M. Thorup: *Regular expression matching with multi-strings and intervals*, in Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, Austin, Texas, USA, January 17-19, 2010, pp. 1297–1308.
15. G. S. Brodal and L. Gasieniec: *Approximate dictionary queries*, in 7th Annual Symposium on Combinatorial Pattern Matching CPM, Laguna Beach, California, USA, June 10-12, 1996, pp. 65–74.
16. N. Chiba and T. Nishizeki: *Arboricity and subgraph listing algorithms.* SIAM Journal on Computing (SICOMP), 14(1) 1985, pp. 210–223.
17. S. Deguchi, F. Higashijima, H. Bannai, S. Inenaga, and M. Takeda: *Parameterized suffix arrays for binary strings*, in Proceedings of the Prague Stringology Conference (PSC), Prague, Czech Republic, September 1-3, 2008, pp. 84–94.
18. P. Ferragina and R. Grossi: *The string b-tree: A new data structure for string search in external memory and its applications.* J. ACM, 46(2) 1999, pp. 236–280.
19. A. Ganguly, W. Hon, K. Sadakane, R. Shah, S. V. Thankachan, and Y. Yang: *Space-efficient dictionaries for parameterized and order-preserving pattern matching*, in 27th Annual Symposium on Combinatorial Pattern Matching CPM, Tel Aviv, Israel, June 27-29, 2016, pp. 2:1–2:12.
20. A. Ganguly, W. Hon, and R. Shah: *A framework for dynamic parameterized dictionary matching*, in 15th Scandinavian Symposium and Workshops on Algorithm Theory SWAT, Reykjavik, Iceland, June 22-24, 2016, pp. 10:1–10:14.
21. T. Haapasalo, P. Silvasti, S. Sippu, and E. Soisalon-Soininen: *Online dictionary matching with variable-length gaps*, in 10th International Symposium on Experimental Algorithms SEA, Kolimpari, Chania, Crete, Greece, May 5-7, 2011, pp. 76–87.
22. W. Hon, T. W. Lam, R. Shah, S. V. Thankachan, H. Ting, and Y. Yang: *Dictionary matching with a bounded gap in pattern or in text.* Algorithmica, 80(2) 2018, pp. 698–713.
23. R. M. Idury and A. A. Schäffer: *Multiple matching of parameterized patterns.* Theor. Comput. Sci., 154(2) 1996, pp. 203–224.
24. O. Keller, T. Kopelowitz, and M. Lewenstein: *On the longest common parameterized subsequence.* Theor. Comput. Sci., 410(51) 2009, pp. 5347–5353.
25. M. Krishnamurthy, E. S. Seagren, R. Alder, A. W. Bayles, J. Burke, S. Carter, and E. Faskha: *How to cheat at securing linux*, Syngress Publishing, Inc., Elsevier, Inc., 2008.
26. G. Kucherov and M. Rusinowitch: *Matching a set of strings with variable length don't cares.* Theor. Comput. Sci., 178(12) 1997, pp. 129–154.
27. M. Lewenstein: *Parameterized pattern matching.* Encyclopedia of Algorithms, 2016, pp. 1525–1530.
28. J. Mendivelso and Y. Pinzón: *Parameterized matching: Solutions and extensions*, in Proceedings of the Prague Stringology Conference (PSC), Prague, Czech Republic, August 24-26, 2015, pp. 118–131.
29. C. W. Mortensen: *Fully dynamic orthogonal range reporting on RAM.* SIAM J. Comput., 35(6) 2006, pp. 1494–1525.

30. M. S. Rahman, C. S. Iliopoulos, I. Lee, M. Mohamed, and W. F. Smyth: *Finding patterns with variable length gaps or don't cares*, in 12th Annual International Conference on Computing and Combinatorics COCOON, Taipei, Taiwan, August 15-18, 2006, pp. 146–155.
31. B. R. Shalom: *Parameterized dictionary matching with one gap*, in Proceedings of the Prague Stringology Conference (PSC), Prague, Czech Republic, 2018, pp. 103–116.
32. D. E. Willard: *Log-logarithmic worst-case range queries are possible in space $\theta(n)$*. Information Processing Letters, 17(2) 1983, pp. 81–84.
33. 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.

An Improvement of the Franek-Jennings-Smyth Pattern Matching Algorithm

Satoshi Kobayashi, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara

Graduate School of Information Sciences, Tohoku University
6-6-05 Aramaki Aza Aoba, Aoba-ku, Sendai, Japan
satoshi_kobayashi@shino.ecei.tohoku.ac.jp
{diptarama, ryoshinaka, ayumis}@tohoku.ac.jp

Abstract. In this paper, we propose a new pattern matching algorithm based on the Franek-Jennings-Smyth (FJS) algorithm. The FJS algorithm is a hybrid of the Knuth-Morris-Pratt (KMP) and the Sunday algorithms. The worst case time complexity of the KMP algorithm is linear time and the Sunday algorithm is quadratic time. However, the Sunday algorithm is faster than the KMP algorithm in practice. Inheriting the virtues of those algorithms, the FJS algorithm runs in linear time in the worst case and fast in practice. We improve the FJS algorithm by further taking an idea inspired by the Quite-Naive algorithm by Cantone and Faro. The experimental results show that our algorithm is faster than the FJS algorithm in general except when a pattern is extremely short.

Keywords: exact pattern matching, Knuth-Morris-Pratt algorithm, Sunday algorithm

1 Introduction

The pattern matching problem is a very fundamental problem in string processing. Given a text T of length n and a pattern P of length m, the pattern matching problem is a task to find all occurrences of P in T. A naive solution of this problem is to compare P with all the substrings of T of length m, which takes $O(mn)$ time. One approach to perform pattern matching more efficiently is to avoid comparing position pairs of the pattern and the text as much as possible based on the property of the pattern obtained by preprocessing it. Among the previously proposed pattern matching algorithms in this approach, the Knuth-Morris-Pratt (KMP) algorithm [9] is well known, which exploits the periodicity of prefixes of P to perform pattern matching in $O(n)$ time with $O(m)$ preprocessing time. On the other hand, the Boyer-Moore (BM) algorithm [2] is famous as an algorithm that can perform pattern matching fast in practice while it has $O(nm)$ worst case time complexity. The BM algorithm uses occurrence positions of each symbol in P and periodicity of suffixes of P rather than prefixes.

Many BM-type algorithms have been proposed. Typical BM-type algorithms are the Horspool algorithm [8] and the Sunday algorithm [10]. The Horspool algorithm is a simpler version of the BM algorithm. The Sunday algorithm is a slight improvement of the Horspool algorithm and known to be practically fast. While the BM algorithm checks characters from right to left of the pattern, the Sunday algorithm can check the characters of the pattern in any order. Franek et al. [7] proposed a hybrid algorithm of the KMP and the Sunday algorithms called the Franek-Jennings-Smyth (FJS) algorithm. Inheriting the virtues of those algorithms, the FJS algorithm runs in linear time in the worst case and fast in practice. A comprehensive survey of the exact online string matching problem is written by Faro and Lecroq [5].

In this paper, we propose a new pattern matching algorithm based on the FJS algorithm. The FJS algorithm uses the shift functions of the KMP and Sunday algorithms. Our algorithm additionally uses a new shift function inspired by the Quite-Naive algorithm by Cantone and Faro [3]. The time complexity of the preprocessing phase of our algorithm is $O(m + \sigma)$ and the search phase runs in $O(n)$ time, where σ denotes the alphabet size. Our algorithm is faster than the FJS algorithm in general except when a pattern is extremely short. It is not faster than the state-of-the-art in general, but it is effective when a pattern frequently appears in a text.

This paper is organized as follows. Section 2 briefly reviews the KMP, Sunday, and FJS algorithms, which are the basis of the proposed algorithm. Section 3 proposes our algorithm. Section 4 shows experimental results comparing the proposed algorithm with several other algorithms using artificial and real data.

2 Preliminaries

2.1 Notation

Let Σ be a set of characters called an *alphabet* and $\sigma = |\Sigma|$ be the size of the alphabet. Σ^* denotes the set of all strings over Σ. The length of a string $w \in \Sigma^*$ is denoted by $|w|$. The *empty string*, denoted by ε, is the string of length zero. The i-th character of w is denoted by $w[i]$ for each $1 \leq i \leq |w|$. The substring of w starting at i and ending at j is denoted by $w[i : j]$ for $1 \leq i \leq j \leq |w|$. For convenience, let $w[i : j] = \varepsilon$ if $i > j$. A string $x = w[1 : i]$ is called a *prefix* of w and a string $z = w[i : |w|]$ is called a *suffix* of w. In particular, a prefix x (resp. suffix z) of w is a *proper prefix* (resp. *proper suffix*) of w when $x \neq w$ (resp. $z \neq w$). A string v is a *border* of w if v is both a prefix and a suffix of w. Note that the empty string is a border of any string. Moreover, it is a *proper border* of w if $v \neq w$. The length of the longest proper border of $w[1 : i]$ for $1 \leq i \leq |w|$ is given by

$$Bord_w[i] = \max\{\, j \mid w[1 : j] = w[i - j + 1 : i] \text{ and } 0 \leq j < i \,\}\,.$$

The exact pattern matching problem is defined as follows:

Input: A text $T \in \Sigma^*$ of length n and a pattern $P \in \Sigma^*$ of length m,
Output: All positions i such that $T[i : i + m - 1] = P$ for $1 \leq i \leq n - m + 1$.

We will use a text $T \in \Sigma^*$ of length n and a pattern $P \in \Sigma^*$ of length m throughout the paper.

Let us consider comparing $T[i : i + m - 1]$ and $P[1 : m]$. The naive method compares characters of the two strings from left to right. When a character mismatch occurs, the pattern is shifted to the right by one character. That is, we compare $T[i + 1 : i + m]$ and $P[1 : m]$. This naive method takes $O(nm)$ time for matching. There are a number of ideas to shift the pattern more so that searching T for P can be performed more quickly, using shift functions obtained by preprocessing the pattern.

2.2 Knuth-Morris-Pratt algorithm

The Knuth-Morris-Pratt (KMP) algorithm [9] is a pattern matching algorithm that has linear worst case time complexity. When the KMP algorithm has confirmed that

Algorithm 1: Calculating *KMP_Shift*

```
Function PreKMPShift($P$)
    $m \leftarrow |P|$;
    $i \leftarrow 1$; $j \leftarrow 0$;
    $KMP\_Shift[1] = 1$;
    while $i \leq m$ do
        while $j > 0$ and $P[i] \neq P[j]$ do
            $j \leftarrow j - KMP\_Shift[j]$;
        $i \leftarrow i + 1$; $j \leftarrow j + 1$;
        if $i \leq m$ and $P[i] = P[j]$ then
            $KMP\_Shift[i] \leftarrow i - (j - KMP\_Shift[j])$
        else
            $KMP\_Shift[i] \leftarrow i - j$
    return $KMP\_Shift$
```

$T[i : i+j-2] = P[1 : j-1]$ and $T[i+j-1] \neq P[j]$ for some $j \leq m$, it shifts the pattern so that a suffix of $T[i : i+j-2]$ matches a prefix of P and we do not have to re-scan any part of $T[i : i+j-2]$ again. Thus, the pattern can be shifted by $j-k-1$ for $k = Bord_P[j-1]$. However, if $P[k+1] = P[j]$, the same mismatch occurs again after the shift. In order to avoid this kind of mismatch, we use $Strong\_Bord[1 : m+1]$ given by

$$Strong\_Bord_P[j] = \begin{cases} -1 & \text{if } j = 1, \\ Strong\_Bord_P[k] & \text{if } j \leq m \text{ and } P[k+1] = P[j], \\ k & \text{otherwise,} \end{cases}$$

where $k = Bord_P[j-1]$. The amount $KMP\_Shift[j]$ of the shift is given by

$$KMP\_Shift[j] = j - Strong\_Bord_P[j] - 1.$$

Fact 1 *If $P[1 : j-1] = T[i : i+j-2]$ and $P[j] \neq T[i+j-1]$, then $P[1 : j-k_j-1] = T[i+k_j : i+j-2]$ holds for $k_j = KMP\_Shift[j]$. Moreover, there is no positive integer $k < KMP\_Shift[j]$ such that $P = T[i+k : i+k+m-1]$.*

Note that if the algorithm has confirmed $T[i : i+m-1] = P$, the shift is given by $KMP\_Shift[m+1]$ after reporting the occurrence of the pattern. Algorithm 1 shows pseudocode to compute the array *KMP_Shift*. Clearly, it runs in $O(m)$ time. By using *KMP_Shift*, the KMP algorithm finds all occurrences of P in T in $O(n)$ time.

2.3 Sunday algorithm

In the Sunday algorithm [10], the amount of the shift when a mismatch occurs between P and $T[i : i+m-1]$ depends on the character $T[i+m] \in \Sigma$. It shifts the pattern so that the character $T[i+m]$ will match the rightmost occurrence of the same character in P. If $T[i+m]$ does not occur in P, by skipping the position $i+m$ of T, we check whether $T[i+m+1 : i+2m] = P$. The preprocessing phase of the Sunday algorithm calculates an array *Sunday_Shift* of size σ given by

$$Sunday\_Shift[c] = m + 1 - \max(\{\, i \mid P[i] = c \,\} \cup \{0\})$$

for $c \in \Sigma$.

Algorithm 2: Calculating *Sunday_Shift*

```
Function PreSundayShift($P, \Sigma$)
    $m \leftarrow |P|$;
    for $c$ in $\Sigma$ do
        $Sunday\_Shift[c] \leftarrow m + 1$;
    for $i \leftarrow 1$ to $m$ do
        $Sunday\_Shift[P[i]] \leftarrow m - i + 1$;
    return $Sunday\_Shift$;
```

Fact 2 *For any i, then there is no positive integer $k < Sunday\_Shift[T[i+m]]$ such that $P = T[i+k : i+k+m-1]$.*

In principle, the order of comparison of characters in P and $T[i : i+m-1]$ is arbitrary. Wherever a mismatch is found, the Sunday algorithm shifts the pattern by $Sunday\_Shift[T[i+m]]$. Algorithm 2 shows pseudocode to calculate the array *Sunday_Shift*, which runs in $O(m+\sigma)$ time. Although the searching time is $O(nm)$ in the worst case, it is practically very fast for large alphabets.

2.4 Franek-Jennings-Smyth algorithm

The Franek-Jennings-Smyth (FJS) algorithm [7] (Algorithm 3) is a hybrid of the KMP and Sunday algorithms. It computes both arrays *KMP_Shift* and *Sunday_Shift* in the preprocessing phase. In the matching phase, it shifts a pattern using the better array depending on the situation where a mismatch has been found. We call the shift based on *KMP_Shift* and *Sunday_Shift* the KMP-shift and the Sunday-shift, respectively. Practically using the Sunday-shift is very effective, but it does not guarantee a linear time execution if we consistently use the Sunday-shift when we have found some *partial match* between the pattern and a text substring, i.e., when it has been confirmed that some nonempty prefixes of the pattern and the text substring match. When a partial match has been found, the FJS algorithm uses the KMP-shift so that it runs in linear time.

When comparing the pattern P with a text substring $T[i : i+m-1]$, the FJS algorithm checks whether the last characters of the pattern and the substring match. If $P[m] \neq T[i+m-1]$, it performs the Sunday-shift repeatedly until we find a position in T that has the character $P[m]$. We call this procedure (**while** loop at Line 8 in Algorithm 3) the *Sunday-phase*. Once we find a position j such that if $P[m] = T[j+m-1]$, we go into the *KMP-phase*, where we use the KMP-shift. As long as the algorithm sees a partial match between the pattern and a text substring, it behaves just like the KMP algorithm. Otherwise, it goes back to the Sunday-phase.

The preprocessing time and the searching time of the algorithm are $O(m+\sigma)$ and $O(n)$, respectively.

Example 1. The arrays *KMP_Shift* and *Sunday_Shift* for $P =$ `abaaca` are calculated as follows:

j	1	2	3	4	5	6	7
$P[j]$	`a`	`b`	`a`	`a`	`c`	`a`	
$KMP\_Shift[j]$	1	1	3	2	3	6	5

c	`a`	`b`	`c`
$Sunday\_Shift[c]$	1	5	2

Algorithm 3: The FJS algorithm

```
Function FJS($P, T, \Sigma$)
    $KMP\_Shift \leftarrow$ PreKMPShift($P$);
    $Sunday\_Shift \leftarrow$ PreSundayShift($P, \Sigma$);
    $n \leftarrow |T|$; $m \leftarrow |P|$;
    $i \leftarrow 1$; $j \leftarrow 1$; $ip \leftarrow m$;
    while $ip \leq n$ do
        if $j \leq 1$ then                                  // No partial match
            while $P[m] \neq T[ip]$ do
                $ip \leftarrow ip + Sunday\_Shift[T[ip + 1]]$;
                if $ip > n$ then
                    halt;
            $j \leftarrow 1$; $i \leftarrow ip - m + 1$;
            while $j < m$ and $P[j] = T[i]$ do
                $i \leftarrow i + 1$; $j \leftarrow j + 1$;
            if $j = m$ then
                $i \leftarrow i + 1$; $j \leftarrow j + 1$;
                output $i - m$;
            $j \leftarrow j - KMP\_Shift[j]$;
        else                                                // Partial match is found
            while $j \leq m$ and $P[j] = T[i]$ do
                $i \leftarrow i + 1$; $j \leftarrow j + 1$;
            if $j = m + 1$ then
                output $i - m$;
            $j \leftarrow j - KMP\_Shift[j]$;
        $ip \leftarrow i + m - j$;
```

Figure 1 illustrates an example run of the FJS algorithm for finding P in $T =$ `abababcababbbca`.

Attempt 1 The first attempt compares the character at the end of P with the character at the corresponding position in T. Since $P[6] \neq T[6]$, the pattern is shifted by $Sunday\_Shift[T[7]] = Sunday\_Shift[\texttt{c}] = 2$

Attempt 2 Since $P[6] = T[8]$, we compare the characters from left to right. The first mismatch occurs at $P[4] \neq T[6]$, so the pattern is shifted by $KMP\_Shift[4] = 2$.

Attempt 3 Because there is a partial match $P[1:1] = T[5:5]$ due to *KMP_Shift* of Attempt 2, the character comparison is started from $P[2]$. A mismatch $P[3] \neq T[7]$ occurs, so the pattern is shifted by $KMP\_Shift[3] = 3$.

Attempt 4 Since there is no partial match, we compare the last character of the pattern $P[6]$ with $T[13]$. Since $P[6] \neq T[13]$, the pattern is shifted by $Sunday\_Shift[T[14]] = Sunday\_Shift[\texttt{c}] = 2$.

Attempt 5 Since $P[6] = T[15]$, we compare the characters of the pattern from left to right. A mismatch $P[3] \neq T[12]$ occurs and the search ends.

3 Proposed algorithm

In this section, we propose to improve the FJS algorithm by introducing another shift function. To make the shift amount bigger on Line 18 of Algorithm 3, we employ and

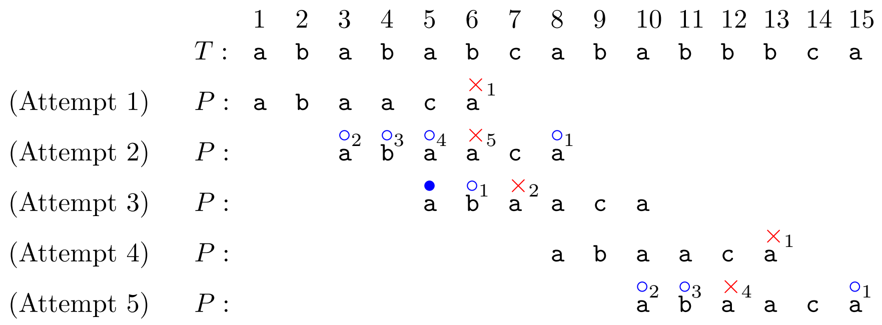

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
T : a b a b a b c a b a b b b c a
(Attempt 1) P : a b a a c a
(Attempt 2) P : a b a a c a
(Attempt 3) P : a b a a c a
(Attempt 4) P : a b a a c a
(Attempt 5) P : a b a a c a

Figure 1. An example run of the FJS algorithm for a pattern P = abaaca and a text T = abababcabababbbca. For each alignment of the pattern, ○ and × indicate a match and a mismatch between the text and the pattern, respectively. The character with ● is known to match the character at the corresponding position in the text without comparison. Subscript numbers show the order of character comparisons in each attempt.

modify the shift function used in Cantone and Faro's Quite-Naive algorithm [3]. Their algorithm compares the pattern P and a text substring $T[i : i + m - 1]$ from right to left. When a mismatch has been found in the middle after at least it has been confirmed that $P[m] = T[i + m - 1]$,[1] the pattern is shifted by δ so that $P[m-\delta] = T[i+m-1]$ hold if $P[m]$ occurs in $P[1 : m-1]$. The shift amount is given by $\delta = \min(\{ j \mid P[m - j] = P[m],\ 1 \le j < m \} \cup \{m\})$, which is the distance between the rightmost and the second rightmost occurrences of the rightmost character. If $P[m]$ does not occur in $P[1 : m-1]$, we shift $\delta = m$ to skip the position $T[i+m-1]$.

We generalize their idea to make the shift amount bigger by focusing on characters as well as the last one in P. Define an array d of size m by

$$d[i] = \min(\{ j \mid P[i - j] = P[i],\ 1 \le j < i \} \cup \{i\})$$

for $1 \le i \le m$. Obviously $d[m]$ is the shift amount used in the Quite-Naive algorithm. We take the maximum value md and the position mdp that gives md by

$$md = \max_{1 \le j \le m} d[j],$$
$$mdp = \arg\max_{1 \le j \le m} d[j].$$

If more than one position gives the maximum value md, we simply take the largest index as mdp.

Fact 3 *If $P[mdp] = T[i + mdp - 1]$, then there is no positive integer $k < md$ such that $P = T[i + k : i + k + m - 1]$.*

While the FJS algorithm uses the rightmost character of P for triggering the Sunday-shift, we use mdp in the Sunday-phase. That is, when comparing P and $T[i : i + m - 1]$, we first check whether $P[mdp] = T[i + mdp - 1]$ holds. If $P[mdp] \neq T[i + mdp - 1]$, we perform the Sunday-shift like the FJS algorithm. This is not an improvement from the FJS algorithm, since, as we have explained in Section 2.3, the shift amount is always determined by the character $T[i + m]$. On the other hand, when we have $P[mdp] = T[i + mdp - 1]$, we compare the corresponding characters of

[1] The Quite-Naive algorithm uses a different shift amount in other situations, which we refrain from explaining in this paper.

Algorithm 4: Calculating *md* and *mdp*

```
1  Function GetMdp($P, \Sigma$)
2      $md \leftarrow 0, mdp \leftarrow 1$;
3      for $c$ in $\Sigma$ do
4          $prevpos[c] \leftarrow 0$;
5      for $j \leftarrow 1$ to $|P|$ do
6          if $md \leq j - prevpos[P[j]]$ then
7              $md \leftarrow j - prevpos[P[j]]$;
8              $mdp \leftarrow j$;
9          $prevpos[P[j]] \leftarrow j$;
10     return $mdp$, $md$;
```

P and $T[i : i+m-1]$ from left to right. When a mismatch is found in the middle, we may perform the KMP-shift, just like the FJS algorithm does. Yet, it is also possible to shift the pattern by md by taking the advantage of the knowledge $P[mdp] = T[i+mdp-1]$, due to Fact 3. We take the bigger one between md and $KMP\_Shift[j]$, provided that the choice guarantees the theoretical linear time performance of the algorithm. Recall that when the KMP algorithm finds that $P[1 : j-1] = T[i : i+j-2]$ and $P[j] \neq T[i+j-1]$, it resumes comparison from checking the match between $T[i+j-1]$ and $P[j - KMP\_Shift[j]]$ if $KMP\_Shift[j] < j$, and $T[i+j]$ and $P[1]$ if $KMP\_Shift[j] = j$. On the other hand, if we shift the pattern by md, we simply start matching $T[i+md]$ and $P[1]$. Therefore, we should use $KMP\_Shift[j]$ rather than md when either $KMP\_Shift[j] < j$ and $md < j-1$ or $KMP\_Shift[j] = j > md$. Summarizing the discussion, we obtain the following shift function:

$$MAX\_Shift[j] = \begin{cases} md & \text{if } md \geq \max\{KMP\_Shift[j],\ j-1\}, \\ KMP\_Shift[j] & \text{otherwise,} \end{cases}$$

where $1 \leq j \leq m+1$. When the shift amount is $MAX\_Shift[j] = md$, we have no partial match between P and the substring of T at the alignment obtained by the shift and thus go to the Sunday-phase. If $MAX\_Shift[j] = KMP\_Shift[j]$, our algorithm behaves just like the FJS algorithm.

We replace *KMP_Shift* on Line 18 of Algorithm 3 by our shift function *MAX_Shift*. On the other hand, the other occurrence of *KMP_Shift* on Line 24 of Algorithm 3 cannot be replaced by *MAX_Shift*, where $P[mdp] = T[i+mdp-1]$ is not guaranteed. Algorithms 4 and 5 calculate the values mdp, md and the shift function *MAX_Shift* in $O(m+|\Sigma|)$ and $O(m)$ time, respectively. Our proposed searching algorithm is shown as Algorithm 6, where differences from Algorithm 3 are highlighted. The correctness of our algorithm is supported by Facts 1, 2 and 3. Clearly it runs in linear in $O(n)$ time apart from the preprocessing phase.

Example 2. We use the same pattern and text as Example 1 for an example run of our algorithm. The arrays *KMP_Shift*, *MAX_Shift* and *Sunday_Shift* for $P =$ `abaaca` are calculated as follows. We have $md = 5$ and $mdp = 5$ for $d[5] = \max_{1 \leq j \leq 6} d[j] = 5$.

Algorithm 5: Calculating *MAX_Shift*

```
Function PreMaxShift($P$, $KMP\_Shift$, $md$)
    for $j \leftarrow 1$ to $|P|+1$ do
        if $md \geq j-1$ and $md \geq KMP\_Shift[j]$ then
            $MAX\_Shift[j] \leftarrow md$;
        else
            $MAX\_Shift[j] \leftarrow KMP\_Shift[j]$;
    return $MAX\_Shift$;
```

Algorithm 6: The proposed algorithm

```
Function ImprovedFJS($P, T, \Sigma$)
    $KMP\_Shift \leftarrow$ PreKMPShift($P$);
    $mdp, md \leftarrow$ GetMdp($P, \Sigma$);
    $MAX\_Shift \leftarrow$ PreMaxShift($P, KMP\_Shift, md$);
    $Sunday\_Shift \leftarrow$ PreSundayShift($P, \Sigma$);
    $n \leftarrow |T|$; $m \leftarrow |P|$;
    $i \leftarrow 1$; $j \leftarrow 1$; $ip \leftarrow m$;
    while $ip \leq n$ do
        if $j \leq 1$ then                                  // No partial match
            while $P[mdp] \neq T[ip-(m-mdp)]$ do
                $ip \leftarrow ip + Sunday\_Shift[T[ip+1]]$;
                if $ip > n$ then
                    halt;
            $j \leftarrow 1$; $i \leftarrow ip - m + 1$;
            while $j \leq m$ and $P[j] = T[i]$ do
                $i \leftarrow i+1$; $j \leftarrow j+1$;
            if $j = m+1$ then
                output $i - m$;
            $j \leftarrow j - MAX\_Shift[j]$;
        else                                                // Partial match is found
            while $j \leq m$ and $P[j] = T[i]$ do
                $i \leftarrow i+1$; $j \leftarrow j+1$;
            if $j = m+1$ then
                output $i - m$;
            $j \leftarrow j - KMP\_Shift[j]$;
        $ip \leftarrow i + m - j$;
```

j	1	2	3	4	**5**	6	7
$P[j]$	a	b	a	a	c	a	
$d[j]$	1	2	2	1	**5**	2	
$KMP\_Shift[j]$	1	1	3	2	3	6	5
$MAX\_Shift[j]$	5	5	5	5	5	6	5

c	a	b	c
$Sunday\_Shift[c]$	1	5	2

Figure 2 shows an example run of our algorithm finding P in $T =$ ababababcababbbca.

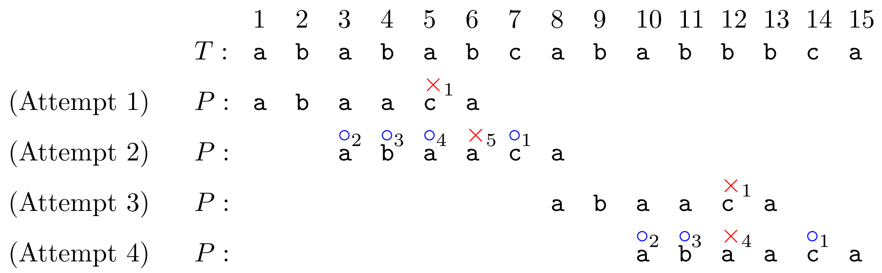

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
T : a b a b a b c a b a b b b c a
(Attempt 1) P : a b a a c a
(Attempt 2) P : a b a a c a
(Attempt 3) P : a b a a c a
(Attempt 4) P : a b a a c a

Figure 2. Running our algorithm for $P =$ `abaaca` and $T =$ `abababcabababbca`

Table 1. Algorithms used for the experiment

Name	Description
KMP	Knuth-Morris-Pratt [9]
QS	Quick-Search [10]
FJS	Franek-Jennings-Smyth [7]
LWFR2	Linear-Weak-Factor-Recognition implemented with a 2-chained-loop [4]
Ours	Proposed algorithm

Attempt 1 First we compare $P[mdp]$ with the corresponding text character $T[mdp]$. Since $P[5] \neq T[5]$, we shift the pattern by $\mathit{Sunday\_Shift}[T[7]] = \mathit{Sunday\_Shift}[\texttt{c}] = 2$.

Attempt 2 Again, $P[mdp]$ is compared with the character at the corresponding position of the text. Because $P[5] = T[7]$, the letters of the pattern are compared from left to right. Finding a mismatch $P[4] \neq T[6]$, the pattern is shifted by $\mathit{MAX\_Shift}[4] = md = 5$.

Attempt 3 Since we are aware of no partial match at this alignment, we compare $P[mdp] = P[5]$ with $T[12]$. By $P[5] \neq T[12]$, the pattern is shifted by $\mathit{Sunday\_Shift}[T[14]] = \mathit{Sunday\_Shift}[\texttt{c}] = 2$.

Attempt 4 We compare $P[mdp] = P[5]$ and $T[14]$. For $P[5] = T[14]$, we compare the characters of the pattern from left to right and find a mismatch $P[3] \neq T[12]$. The shift amount lets the pattern go beyond the end of the text and thus the algorithm halts.

4 Experiments

In this section, we compare the execution times of the proposed algorithm with several other algorithms using various texts and patterns. Table 1 shows the algorithms we used. The experiments were performed on AOBA [11], an integrated online platform evaluating string processing algorithms, with a PC with Xeon E3-1220 V2, 8 GB RAM, Ubuntu 16.04 and Docker 18.09.0. The evaluations are executed on a Docker container, and resources provided by the sandbox are limited to 1 CPU and 1 GB RAM. We used the implementations in SMART [6] for all algorithms except for our and the FJS algorithms, with modification to conform to the AOBA execution format. The implementation of the FJS algorithm we used is the original [7].[2] All implementations are in the C language, compiled using GCC 5.4.0 with the optimization option `-O3`. We used the best performance result among three trials for each experiment.

[2] The SMART implementation of the FJS algorithm is in error.

4.1 Random strings

We prepared a random text of length $n = 1000000$ and random patterns of length $m = 2, 4, 8, 16, 32, 64, 128, 256$ and 512 over alphabets of size $\sigma = 2, 4, 8, 16, 32, 64$ and 95. We measured the total time of 100 runs. Table 2 compares the performance of our algorithm with the ones in Table 1. Our algorithm runs faster than the FJS algorithm in most cases due to the bigger shift on Line 19 of Algorithm 6 than that on Line 18 of Algorithm 3. The difference becomes quite significant when the alphabet is rather small, since both algorithms easily exit the Sunday-phase and execute those lines more often. Exceptional cases, where ours is a little slower than the FJS algorithm, are when the pattern is extremely short. Recall that after the FJS algorithm confirms $P[m] = T[i+m-1]$ in the Sunday-phase, it does not compare $P[m]$ and $T[i+m-1]$ any more in the succeeding KMP-phase, while our algorithm may compare the same positions twice on Lines 10 and 15 in Algorithm 6. The redundancy will be relatively large when the pattern is extremely short. On the other hand, LWFR2 is the fastest for almost all random data.

4.2 Artificial strings

We also experimented with the following strings.

1. Fibonacci strings are generated by the following recurrence:

$$Fib_1 = \texttt{b},\ Fib_2 = \texttt{a} \text{ and } Fib_n = Fib_{n-1} \cdot Fib_{n-2} \text{ for } n > 2.$$

   The text is fixed to $T = Fib_{32}$ of length $n = 2178309$, and the patterns are randomly extracted from T of length $m = 2, 4, 8, 16, 32, 64, 128$ and 256. Fibonacci strings are known to be highly repetitive. We measured the total time after 100 executions.
2. Texts with frequent pattern occurrences are generated by intentionally embedding a lot of patterns. We embedded 32768 occurrences of a pattern of length $m = 2, 4, 8, 16, 32$ and 64 into a text of length $n = 4000000$ over an alphabet of size $\sigma = 8$. More specifically, we first randomly generate a pattern and a provisional text, which may contain the pattern. Then we randomly change characters of the text until the pattern does not occur in the text. Finally we embed the pattern at random positions without overlapping. We measured the total time after 25 executions.

Our proposed algorithm is the fastest for the Fibonacci string (Table 3). Since patterns appear so frequently in a Fibonacci string, algorithms that perform verification after hash-filtering, such as LWFR, could be slow. Ours performed the best for texts with frequent occurrences of long patterns (Table 4). Therefore, our algorithm seem to work efficiently when patterns frequently appear in the text.

4.3 Practical strings

We also made similar measurements on practical data. The total time of 25 runs was measured. We used the following data as texts.

1. Genome sequence: the genome sequence of E. coli of length $n = 4641652$ with $\sigma = 4$, from NCBI[3].

[3] `https://www.ncbi.nlm.nih.gov/genome/167?genome_assembly_id=161521`

Table 2. The execution time of the algorithms in Table 1 using the random strings

$\sigma = 2$

m	2	4	8	16	32	64	128	256	512
KMP	815.84	816.46	813.49	814.84	817.05	815.29	816.96	817.65	816.14
QS	669.15	709.66	735.67	761.96	775.29	763.89	774.25	729.14	754.32
LWFR2	629.74	**549.76**	**342.04**	**238.98**	**188.72**	**169.70**	**158.24**	**151.89**	**151.89**
FJS	**608.69**	629.12	693.13	757.15	759.19	757.78	763.55	737.30	755.06
Ours	689.81	603.26	520.00	467.80	444.09	408.40	386.28	361.21	344.48

$\sigma = 4$

m	2	4	8	16	32	64	128	256	512
KMP	725.49	746.48	750.65	746.55	742.36	748.35	746.22	751.83	746.07
QS	544.44	441.19	376.97	343.46	350.00	341.49	353.08	360.16	344.43
LWFR2	**329.47**	**273.56**	**239.52**	**193.17**	**162.70**	**153.32**	**148.54**	**146.78**	**144.95**
FJS	532.69	477.87	445.55	418.53	425.92	417.98	431.15	439.43	420.17
Ours	547.44	408.54	329.00	275.47	255.39	241.96	233.55	227.61	220.60

$\sigma = 8$

m	2	4	8	16	32	64	128	256	512
KMP	591.71	589.70	588.05	588.54	585.03	589.40	589.90	590.50	588.08
QS	406.26	318.84	256.1	224.52	217.62	217.83	217.91	219.07	219.32
LWFR2	**292.49**	**213.36**	**189.96**	**181.20**	**166.41**	**151.78**	**148.68**	**146.11**	**146.44**
FJS	393.45	322.11	269.27	240.90	236.04	235.12	234.38	237.45	236.64
Ours	395.28	300.18	243.03	211.71	196.09	190.36	184.00	184.13	181.70

$\sigma = 16$

m	2	4	8	16	32	64	128	256	512
KMP	499.51	493.94	495.27	490.56	492.42	490.36	491.20	493.83	492.84
QS	345.50	269.98	219.15	190.88	178.53	174.37	173.40	174.93	174.68
LWFR2	**260.03**	**193.25**	**172.27**	**164.07**	**160.09**	**153.35**	**146.72**	**144.97**	**144.88**
FJS	327.49	263.10	217.15	192.90	182.34	178.80	177.77	178.92	178.82
Ours	327.14	256.38	211.15	185.85	174.03	168.53	165.59	165.38	163.69

$\sigma = 32$

m	2	4	8	16	32	64	128	256	512
KMP	440.08	436.27	435.02	433.58	436.51	434.87	433.98	435.30	435.02
QS	319.45	249.25	202.80	176.55	164.43	158.70	157.47	157.51	157.20
LWFR2	**247.72**	**182.07**	**164.72**	**155.89**	**153.35**	**152.02**	**148.13**	**145.45**	**145.00**
FJS	299.48	238.50	198.54	175.64	165.54	162.61	161.32	160.53	161.08
Ours	299.54	236.41	196.95	173.16	162.98	156.66	154.97	154.47	155.43

$\sigma = 64$

m	2	4	8	16	32	64	128	256	512
KMP	405.14	403.63	405.03	403.81	404.46	402.91	402.73	403.97	404.15
QS	307.45	239.90	197.00	171.20	158.06	153.09	149.10	148.35	148.46
LWFR2	**243.00**	**177.85**	**158.95**	**151.39**	**148.76**	**147.38**	**147.05**	**146.53**	**145.62**
FJS	282.46	229.53	190.89	169.34	157.98	155.70	153.75	153.41	151.86
Ours	284.43	226.99	188.94	167.47	157.18	152.22	149.44	148.78	149.27

$\sigma = 95$

m	2	4	8	16	32	64	128	256	512
KMP	394.58	394.42	392.13	394.58	393.01	392.00	394.32	395.56	393.35
QS	303.62	237.59	194.73	169.84	155.15	151.36	147.39	145.69	**145.23**
LWFR2	**241.08**	**176.17**	**157.90**	**150.18**	**147.23**	**146.94**	**145.41**	**145.59**	146.12
FJS	278.63	224.24	188.01	166.81	155.56	153.84	151.51	150.28	149.10
Ours	280.46	224.78	187.13	167.33	155.23	149.86	148.36	147.49	146.93

Table 3. The execution time of the algorithms in Table 1 using the Fibonacci strings

m	2	4	8	16	32	64
KMP	806.02	782.25	771.26	783.94	733.25	656.00
QS	809.51	824.72	835.41	954.10	1119.16	1280.31
LWFR2	842.78	968.64	847.70	774.92	676.61	648.86
FJS	**616.63**	582.88	505.26	449.33	425.88	427.02
Ours	821.58	**581.54**	**472.51**	**405.69**	**377.28**	**372.04**

Table 4. The execution time of the algorithms in Table 1 using the texts with 32768 occurrences of a pattern

m	2	4	8	16	32	64
KMP	578.23	609.99	605.00	591.71	554.09	476.73
QS	397.98	340.59	285.62	259.85	267.48	284.01
LWFR2	**271.54**	**242.86**	**236.93**	**236.70**	244.92	291.22
FJS	383.25	340.95	291.77	261.48	251.58	234.06
Ours	380.58	325.98	272.63	240.43	**233.28**	**229.03**

Table 5. The execution time of the algorithms in Table 1 using the genome sequences

m	2	4	8	16	32	64
KMP	841.13	860.16	836.45	869.12	869.73	847.45
QS	630.21	501.58	417.28	398.56	387.16	399.61
LWFR2	**373.87**	**318.24**	**257.32**	**214.21**	**178.39**	**166.65**
FJS	604.83	541.89	488.88	480.37	465.15	478.47
Ours	619.65	462.60	359.56	310.54	285.58	272.14

Table 6. The execution time of the algorithms in Table 1 using the English texts

m	2	4	8	16	32	64
KMP	528.59	516.48	534.79	514.70	515.38	533.08
QS	377.53	288.36	237.96	199.98	184.02	172.70
LWFR2	**270.67**	**197.57**	**178.91**	**169.13**	**163.86**	**157.35**
FJS	351.93	286.69	229.52	202.25	184.61	174.37
Ours	359.03	292.18	221.54	190.36	175.27	166.16

2. English text: the King James version of the Bible of length $n = 4017009$ with $\sigma = 62$, from the Large Canterbury Corpus[4] [1]. We removed the line break from the text.

In both cases, the patterns are randomly extracted from the text of length $m = 2, 4, 8, 16, 32, 64, 128$ and 256.

Table 5 shows that our algorithm is faster than the FJS algorithm for genome sequences. For English texts, Table 6 shows that our and the FJS algorithm have almost the same speed, probably because the Sunday-shift is dominant. For both texts, LWFR2 is the fastest for all pattern lengths.

[4] `http://corpus.canterbury.ac.nz/`

5 Conclusion

We propose a new algorithm modifying the FJS algorithm. Our experimental results show that it runs faster than the FJS algorithm in general except when a pattern is extremely short. Moreover, our algorithm outperformed representative existing algorithms for data where patterns appear frequently in text.

Acknowledgment

This work is partially supported by JSPS KAKENHI Grant Numbers JP19K20208 and JP15H05706.

References

1. R. Arnold and T. Bell: *A corpus for the evaluation of lossless compression algorithms*, in Proceedings DCC '97. Data Compression Conference, 1997, pp. 201–210.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) Oct. 1977, pp. 762–772.
3. D. Cantone and S. Faro: *Searching for a substring with constant extra-space complexity*, in Proc. of Third International Conference on Fun with algorithms, 2004, pp. 118–131.
4. D. Cantone, S. Faro, and A. Pavone: *Linear and Efficient String Matching Algorithms Based on Weak Factor Recognition.* Journal of Experimental Algorithmics, 24(1) feb 2019, pp. 1–20.
5. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) 2013, pp. 13:1–13:42.
6. S. Faro, T. Lecroq, S. Borzì, S. D. Mauro, and A. Maggio: *The string matching algorithms research tool*, in Proceedings of the Prague Stringology Conference 2016, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2016, pp. 99–113.
7. F. Franek, C. G. Jennings, and W. Smyth: *A simple fast hybrid pattern-matching algorithm.* Journal of Discrete Algorithms, 5(4) dec 2007, pp. 682–695.
8. R. N. Horspool: *Practical fast searching in strings.* Software: Practice and Experience, 10(6) jun 1980, pp. 501–506.
9. D. E. Knuth, J. H. Morris Jr., and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal on Computing, 6(2) 1977, pp. 323–350.
10. D. M. Sunday: *A very fast substring search algorithm.* Communications of the ACM, 33(8) aug 1990, pp. 132–142.
11. R. Wakimoto, S. Kobayashi, Y. Igarashi, J. Davaajav, D. Hendrian, R. Yoshinaka, and A. Shinohara: *AOBA: An online benchmark tool for algorithms in stringology*, 2019, http://aoba.iss.is.tohoku.ac.jp/.

Lexicalized Syntactic Analysis by Restarting Automata

František Mráz[1], Friedrich Otto[1], Dana Pardubská[2]*, and Martin Plátek[1]**

[1] Charles University, Department of Computer Science
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
martin.platek@mff.cuni.cz, frantisek.mraz@mff.cuni.cz, otto@ktiml.mff.cuni.cz

[2] Comenius University in Bratislava, Department of Computer Science
Mlynská Dolina, 84248 Bratislava, Slovakia
pardubska@dcs.fmph.uniba.sk

Abstract. We study *h-lexicalized two-way restarting automata* that can rewrite at most i times per cycle for some $i \geq 1$ ($\mathsf{hRLWW}(i)$-automata). This model is considered useful for the study of lexical (syntactic) disambiguation, which is a concept from linguistics. It is based on certain reduction patterns. We study lexical disambiguation through the formal notion of *h-lexicalized syntactic analysis* (hLSA). The hLSA is composed of a *basic language* and the corresponding *h-proper language*, which is obtained from the basic language by mapping all basic symbols to input symbols. We stress the sensitivity of hLSA by $\mathsf{hRLWW}(i)$-automata to the size of their windows, the number of possible rewrites per cycle, and the degree of (non-)monotonicity. We introduce the concepts of *contextually transparent languages* (CTL) and *contextually transparent lexicalized analyses* based on very special reduction patterns, and we present two-dimensional hierarchies of their subclasses based on the size of windows and on the degree of synchronization. The bottoms of these hierarchies correspond to the context-free languages. CTL creates a proper subclass of context-sensitive languages with syntactically natural properties.

1 Introduction

This paper is a continuation of conference paper [11]. The motivation for this paper is to study lexical disambiguation, which is a basic concept of linguistic schools working with lexicalized syntax. Let us note that traditional dependency syntaxes are lexicalized in our sense.

In a lexicalized syntactical analysis of a sentence, at first all input words are replaced by disambiguated word forms, i.e., original words are enhanced with, e.g., morphological and syntactic categories, like the input word 'means' can be extended with a tag that it is a verb or a different tag that it is a noun which is further refined with another tag distinguishing whether it plays the role of subject or object. After such disambiguation, the lexicalized syntactic analysis checks whether the tagged word forms constitute a (grammatically) correct sentence which is correctly tagged.

A model of the restarting automaton that formalizes lexicalized syntactic disambiguation in a similar way as categorial grammars (see, e.g., [1]) – the *h-lexicalized restarting automaton* (hRLWW) – was introduced in [9]. This model is obtained from the two-way restarting automaton of [8] by adding a letter-to-letter morphism h that assigns an input symbol to each working symbol. This morphism models the (pure

* The research is partially supported by VEGA 2/0165/16.

** The research was partially supported by the grant of the Czech Science Foundation No. 19-05704S by the authors stay at Institute of Computer Science, Czech Academy of Sciences.

non-syntactic) lexical disambiguation. Now the *basic language* $L_C(M)$ of an hRLWW-automaton M consists of all words over the working alphabet of M that are accepted by M, and the *h-proper language* $L_{hP}(M)$ of M is obtained from $L_C(M)$ through the morphism h.

The set of pairs $\{ (h(w), w) \mid w \in L_C(M) \}$, denoted as $L_A(M)$, is called the h-lexicalized syntactic (sentence) analysis (hLSA) by M. Thus, in this setting the auxiliary symbols themselves play the role of the tagged items. That is, each auxiliary symbol b can be seen as a pair consisting of an input symbol $h(b)$ and some additional syntactico-semantic information (tags or categories).

In contrast to the original hRLWW-automaton that uses exactly one length-reducing rewrite in a cycle, here we study *h-lexicalized restarting automata* that allow up to $i \geq 1$ length-reducing rewrites in a cycle (hRLWW(i)). Our first goal is to show that these models are suited for a transparent and sufficiently flexible modeling of the lexical analysis by analysis by reduction (compare to [6]).

Analysis by reduction is traditionally used to analyze sentences of natural languages with a higher degree of word-order freedom like, e.g., Czech, Latin, or German (see, e.g., [6]). Usually, a human reader is supposed to understand the meaning of a given sentence before he starts to analyze it; h-lexicalized syntactic analysis (hLSA) based on the h-lexicalized analysis by reduction (AR_h) simulates such a behavior by analyzing sentences, where morphological and syntactical tags have been added to the word forms and punctuation marks (see, e.g., [6]). An important property of analysis by reduction is the so-called correctness preserving property. Using hRLWW(i)-automata the linguistic correctness preserving property is simulated by the formal notion of *basic correctness preserving property.*

We stress here newly the constraint of *strong cyclic form.* It preserves the essential part of the power of hRLWW(i)-automata, and, additionally, it allows to extend the complexity results obtained for classes of infinite languages and hLSAs also to classes of finite languages and hLSAs. This is quite useful for the classification and learning of individual phenomena in computational and corpus linguistics, where all the (syntactic) observations are of a finite nature. It is also useful for the formulation of techniques for the localization of syntactic errors (grammar-checking).

Finally, we introduce the concepts of *contextually transparent languages* (CTL) and *contextually transparent lexicalized analyses* (CTLA), which create a formal basis for an environment for lexical analysis and grammar-checking of natural languages. We establish a two level and two-dimensional essential refinement of the Chomsky hierarchy in the area of CTL. We transfer this refinement also to the area of CTLA.

In this paper we do not pay attention to input languages, which are the languages usually studied in the automata theory, as they are not suitable for the modeling of lexicalized sentence disambiguation (see [11]).

2 Definitions

By $\subseteq$ and $\subset$ we denote the subset and the proper subset relation, respectively. Throughout the paper, λ will denote the empty word.

We start with the definition of the two-way restarting automaton as an extension to the original definition from [8]. In contrast to [10], we do not consider general h-lexicalized two-way restarting list automata which can rewrite arbitrary many times during each cycle. Instead, we introduce two-way restarting automata which can rewrite at most i times per cycle for an integer $i \geq 1$.

Definition 1. *Let i be a positive integer. A* two-way restarting automaton, *an* $\mathsf{RLWW}(i)$*-automaton for short, is a machine with a flexible tape and a finite-state control. It is defined through a 9-tuple $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, i, \delta)$, where Q is a finite set of states, Σ is a finite input alphabet, and $\Gamma (\supseteq \Sigma)$ is a finite working alphabet. The symbols from $\Gamma \smallsetminus \Sigma$ are called* auxiliary symbols. *Further, the symbols $¢, \$ \notin \Gamma$, called* sentinels, *are the markers for the left and the right border of the workspace, respectively, $q_0 \in Q$ is the initial state, $k \geq 1$ is the size of the* read/write window, *$i \geq 1$ is the number of allowed rewrites in a cycle (see below), and*

$$\delta : Q \times \mathcal{PC}^{\leq k} \to \mathcal{P}(\,(Q \times (\{\mathsf{MVR}, \mathsf{MVL}\} \cup \{\,\mathsf{SL}(v) \mid v \in \mathcal{PC}^{\leq k-1}\,\}))$$
$$\cup \{\mathsf{Restart}, \mathsf{Accept}, \mathsf{Reject}\})$$

is the transition relation. *Here $\mathcal{P}(S)$ denotes the powerset of a set S and*

$$\mathcal{PC}^{\leq k} = (\{¢\} \cdot \Gamma^{k-1}) \cup \Gamma^{k} \cup (\Gamma^{\leq k-1} \cdot \{\$\}) \cup (\{¢\} \cdot \Gamma^{\leq k-2} \cdot \{\$\})$$

is the set of possible contents *of the read/write window of M.*

Being in a state $q \in Q$ and seeing a word $u \in \mathcal{PC}^{\leq k}$ in its window, the automaton can perform six different types of transition steps (or instructions):

1. *A* move-right step $(q, u) \longrightarrow (q', \mathsf{MVR})$ *assumes that $(q', \mathsf{MVR}) \in \delta(q, u)$, where $q' \in Q$ and $u \notin \{\lambda, ¢\} \cdot \Gamma^{\leq k-1} \cdot \{\$\}$. This move-right step causes M to shift the window one position to the right and to enter state q'.*
2. *A* move-left step $(q, u) \longrightarrow (q', \mathsf{MVL})$ *assumes that $(q', \mathsf{MVL}) \in \delta(q, u)$, where $q' \in Q$ and $u \notin \{¢\} \cdot \Gamma^{*} \cdot \{\lambda, \$\}$. It causes M to shift the window one position to the left and to enter state q'.*
3. *An* SL-step $(q, u) \longrightarrow (q', \mathsf{SL}(v))$ *assumes that $(q', \mathsf{SL}(v)) \in \delta(q, u)$, where $q' \in Q$, $v \in \mathcal{PC}^{\leq k-1}$, v is shorter than u, and v contains all the sentinels that occur in u (if any). It causes M to replace u by v, to enter state q', and to shift the window by $|u| - |v|$ items to the left – but at most to the left sentinel $¢$ (that is, the contents of the window is 'completed' from the left, and so the distance to the left sentinel decreases, if the window was not already at $¢$).*
4. *A* restart step $(q, u) \longrightarrow \mathsf{Restart}$ *assumes that* $\mathsf{Restart} \in \delta(q, u)$. *It causes M to place its window at the left end of its tape, so that the first symbol it sees is the left sentinel $¢$, and to reenter the initial state q_0.*
5. *An* accept step $(q, u) \longrightarrow \mathsf{Accept}$ *assumes that* $\mathsf{Accept} \in \delta(q, u)$. *It causes M to halt and accept.*
6. *A* reject step $(q, u) \longrightarrow \mathsf{Reject}$ *assumes that* $\mathsf{Reject} \in \delta(q, u)$. *It causes M to halt and reject.*

A *configuration* of an $\mathsf{RLWW}(i)$-automaton M is a word $\alpha q \beta$, where $q \in Q$, and either $\alpha = \lambda$ and $\beta \in \{¢\} \cdot \Gamma^{*} \cdot \{\$\}$ or $\alpha \in \{¢\} \cdot \Gamma^{*}$ and $\beta \in \Gamma^{*} \cdot \{\$\}$; here q represents the current state, $\alpha\beta$ is the current contents of the tape, and it is understood that the read/write window contains the first k symbols of β or all of β if $|\beta| < k$. A *restarting configuration* is of the form $q_0¢w\$$, where $w \in \Gamma^{*}$; if $w \in \Sigma^{*}$, then $q_0¢w\$$ is an *initial configuration.* We see that any initial configuration is also a restarting configuration, and that any restart transfers M into a restarting configuration.

In general, an $\mathsf{RLWW}(i)$-automaton M is *nondeterministic*, that is, there can be two or more steps (instructions) with the same left-hand side (q, u), and thus, there can be more than one computation that start from a given restarting configuration. If this is not the case, the automaton is *deterministic.*

A *computation* of M is a sequence $C = C_0, C_1, \ldots, C_j$ of configurations of M, where C_0 is an initial or a restarting configuration and $C_{\ell+1}$ is obtained from C_ℓ by a step of M, for all $0 \leq \ell < j$. In the following we only consider computations of $\mathsf{RLWW}(i)$-automata which are finite and end either by an accept or by a reject step.

Cycles and tails: Any finite computation of an $\mathsf{RLWW}(i)$-automaton M consists of certain phases. A phase, called a *cycle*, starts in a restarting configuration, the window moves along the tape performing non-restarting steps until a restart step is performed and thus a new restarting configuration is reached. If no further restart step is performed, any finite computation necessarily finishes in a halting configuration – such a phase is called a *tail*. It is required that in each cycle an $\mathsf{RLWW}(i)$-automaton executes at least one, but at most i SL-steps. Moreover, it must not execute any SL-step in a tail.

This induces the following relation of *cycle-rewriting* by M: $u \Rightarrow^c_M v$ iff there is a cycle that begins with the restarting configuration $q_0¢u\$$ and ends with the restarting configuration $q_0¢v\$$. The relation $\Rightarrow^{c^*}_M$ is the reflexive and transitive closure of $\Rightarrow^c_M$. We stress that the cycle-rewriting is a very important feature of an $\mathsf{RLWW}(i)$-automaton. As each SL-step is strictly length-reducing, we see that $u \Rightarrow^c_M v$ implies that $|u| > |v|$. Accordingly, $u \Rightarrow^c_M v$ is also called a *reduction* by M.

An *input word* $w \in \Sigma^*$ *is accepted by* M, if there is a computation which starts with the initial configuration $q_0¢w\$$ and ends by executing an accept step. By $L(M)$ we denote the language consisting of all input words accepted by M; we say that M *recognizes (or accepts) the input language* $L(M)$.

A *basic (or characteristic) word* $w \in \Gamma^*$ *is accepted by* M if there is a computation which starts with the restarting configuration $q_0¢w\$$ and ends by executing an accept step. By $L_C(M)$ we denote the set of all words from Γ^* that are accepted by M; we say that M *recognizes (or accepts) the basic (or characteristic*[1]*) language* L_C.

Finally, we come to the definition of the h-lexicalized $\mathsf{RLWW}(i)$-automaton.

Definition 2. *Let i be a positive integer. An* h-lexicalized $\mathsf{RLWW}(i)$-automaton, *or an* $\mathsf{hRLWW}(i)$-automaton, *is a pair* $\widehat{M} = (M, h)$, *where* $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, i, \delta)$ *is an* $\mathsf{RLWW}(i)$*-automaton and* $h : \Gamma \to \Sigma$ *is a letter-to-letter morphism satisfying* $h(a) = a$ *for all input letters* $a \in \Sigma$. *The* input language $L(\widehat{M})$ *of* $\widehat{M}$ *is simply the language* $L(M)$ *and the* basic language $L_C(\widehat{M})$ *of* $\widehat{M}$ *is the language* $L_C(M)$. *Further, we take* $L_{hP}(\widehat{M}) = h(L_C(M))$, *and we say that* $\widehat{M}$ recognizes (or accepts) the h-proper language $L_{hP}(\widehat{M})$.

Finally, the set $L_A(\widehat{M}) = \{ (h(w), w) \mid w \in L_C(M) \}$ *is called the* h-lexicalized syntactic analysis *(shortly hLSA) by* $\widehat{M}$.

We say that, for $x \in \Sigma^*$, $L_A(\widehat{M}, x) = \{ (x, y) \mid y \in L_C(M), h(y) = x \}$ *is the* h-syntactic analysis (lexicalized syntactic (sentence) analysis) *for* x *by* $\widehat{M}$. *We see that* $L_A(\widehat{M}, x)$ *is non-empty only for* x *from* $L_{hP}(\widehat{M})$.

Evidently, for an $\mathsf{hRLWW}(i)$-automaton $\widehat{M}$, $L(\widehat{M}) \subseteq L_{hP}(\widehat{M}) = h(L_C(\widehat{M}))$. Let us note that h-syntactic analysis formalizes the linguistic notion of *lexical sentence disambiguation.* Each auxiliary symbol $x \in \Gamma \smallsetminus \Sigma$ of a word from $L_C(\widehat{M})$ can be

[1] Basic languages were also called characteristic languages in [9] and several other papers, therefore, here we preserve the original notation with the subscript C.

considered as a disambiguated form of the input symbol $h(x)$. The following fact ensures the transparency for computations of $\mathsf{hRLWW}(i)$-automata.

Definition 3. (Basic Correctness Preserving Property)
Let M be an $\mathsf{hRLWW}(i)$-automaton. If $u \Rightarrow_M^{c^*} v$ and $u \in L_{\mathrm{C}}(M)$ induce that $v \in L_{\mathrm{C}}(M)$, and therewith $h(v) \in L_{\mathrm{hP}}(M)$ and $(h(v), v) \in L_{\mathrm{A}}(M)$, then we say that M is basically correctness preserving.

Fact 4. *Let M be a deterministic $\mathsf{hRLWW}(i)$-automaton. Then M is basically correctness preserving.*

Notations. For brevity, the prefix det- will be used to denote the property of being deterministic. For any class A of automata, $\mathcal{L}(\mathsf{A})$ will denote the class of input languages that are recognized by automata from A, $\mathcal{L}_{\mathrm{C}}(\mathsf{A})$ will denote the class of basic languages that are recognized by automata from A, $\mathcal{L}_{\mathrm{hP}}(\mathsf{A})$ will denote the class of h-proper languages that are recognized by automata from A, and $\mathcal{L}_{\mathrm{A}}(\mathsf{A})$ will denote the class of hLSA (h-lexicalized syntactic analyses) that are defined by automata from A.

For a natural number $k \geq 1$, $\mathcal{L}(k\text{-}\mathsf{A})$, $\mathcal{L}_{\mathrm{C}}(k\text{-}\mathsf{A})$, $\mathcal{L}_{\mathrm{hP}}(k\text{-}\mathsf{A})$, $\mathcal{L}_{\mathrm{A}}(k\text{-}\mathsf{A})$ will denote the classes of input, basic, h-proper languages, and hLSAs, respectively, that are recognized by those automata from A that use a read/write window of size at most k.

2.1 Further Refinements, and Constraints on hRLWW(i)-Automata

Here we introduce some constrained types of rewrite steps which are motivated by different types of linguistic reductions.

A *delete-left step* $(q, u) \to (q', \mathsf{DL}(v))$ is a special type of an SL-step $(q', \mathsf{SL}(v)) \in \delta(q, u)$, where v is a proper (scattered) subsequence of u, containing all the sentinels from u (if any). It causes M to replace u by v (by deleting excessive symbols), to enter state q', and to shift the window by $|u| - |v|$ symbols to the left, but at most to the left sentinel ¢.

A *contextual-left step* $(q, u) \to (q', \mathsf{CL}(v))$ is a special type of DL-step $(q', \mathsf{DL}(v)) \in \delta(q, u)$, where $u = v_1u_1v_2u_2v_3$, $u_1, u_2 \in \Gamma^*$, $|u_1u_2| \geq 1$, and $v = v_1v_2v_3$, such that v contains all the sentinels from u (if any). It causes M to replace u by v (by deleting the factors u_1 and u_2 of u), to enter state q', and to shift the window by $|u| - |v|$ symbols to the left, but at most to the left sentinel ¢.

An $\mathsf{RLWW}(i)$-automaton is called an $\mathsf{RLW}(i)$-automaton if its working alphabet coincides with its input alphabet, that is, no auxiliary symbols are available for this automaton. Note that in this situation, each restarting configuration is necessarily an initial configuration. Within the denotation for types of automata, R denotes the use of moves to the right, L denotes the use of moves to the left, WW denotes the use of both input and working alphabets, and a single W denotes the use of an input alphabet only (that is, the working alphabet coincides with the input alphabet).

Evidently, we need not distinguish between $\mathsf{hRLW}(i)$-automata and $\mathsf{RLW}(i)$-automata, since for $\mathsf{RLW}(i)$-automata the only possible morphism h is the identity.

Fact 5. (Equalities of Languages for $\mathsf{hRLW}(i)$-automata.)
For any $\mathsf{RLW}(i)$-automaton M, $L(M) = L_{\mathrm{C}}(M) = L_{\mathrm{hP}}(M)$.

An $\mathsf{RLW}(i)$-automaton is called an $\mathsf{RLWD}(i)$-*automaton* if all its rewrite steps are DL-steps, and it is an $\mathsf{RLWC}(i)$-automaton if all its rewrite steps are CL-steps. Further, an $\mathsf{RLWW}(i)$-automaton is called an $\mathsf{RLWWC}(i)$-*automaton* if all its rewrite steps are CL-steps. Similarly, an $\mathsf{RLWW}(i)$-automaton is called an $\mathsf{RLWWD}(i)$-*automaton* if all its rewrite steps are DL-steps. Observe that when concentrating on input languages, DL- and CL-steps ensure that no auxiliary symbols can ever occur on the tape; if, however, we are interested in basic or h-proper languages, then auxiliary symbols can play an important role even though a given $\mathsf{RLWW}(i)$-automaton uses only DL- or CL-steps. Therefore, we distinguish between $\mathsf{RLWWC}(i)$- and $\mathsf{RLWC}(i)$-automata, and between $\mathsf{RLWWD}(i)$- and $\mathsf{RLWD}(i)$-automata.

In the following we will use the corresponding notation also for subclasses of $\mathsf{RLWW}(i)$- and $\mathsf{hRLWW}(i)$-automata. Additionally, prefix k- for a type X of $\mathsf{RLWW}(i)$-automata and an integer $k \geq 1$ will denote the subclass of X of automata of window size at most k. For example, 3-det-$\mathsf{hRLWC}(i)$ denotes the class of deterministic h-lexicalized $\mathsf{RLWC}(i)$-automata with window size at most 3.

We recall the notion of *monotonicity* (see e.g. [11]) as an important constraint for computations of $\mathsf{RLWW}(i)$-automata. Let M be an $\mathsf{RLWW}(i)$-automaton, and let $C = C_k, C_{k+1}, \ldots, C_j$ be a sequence of configurations of M, where $C_{\ell+1}$ is obtained by a single transition step from C_ℓ, $k \leq \ell < j$. We say that C is a *sub-computation* of M. If $C_\ell = ¢\alpha q\beta\$$, then $|\beta\$|$ is the *right distance* of C_ℓ, which is denoted by $D_{\mathrm{r}}(C_\ell)$. We say that a subsequence $(C_{\ell_1}, C_{\ell_2}, \ldots, C_{\ell_n})$ of C is *monotone* if $D_{\mathrm{r}}(C_{\ell_1}) \geq D_{\mathrm{r}}(C_{\ell_2}) \geq \cdots \geq D_{\mathrm{r}}(C_{\ell_n})$. A computation of M is called *monotone* if the corresponding subsequence of rewrite configurations is monotone. Here a configuration is called a *rewrite configuration* if in this configuration an SL-step is being applied. Finally, M itself is called *monotone* if each of its computations is monotone. We use the prefix mon- to denote monotone types of $\mathsf{hRLWW}(i)$-automata. This notion of monotonicity has already been considered in various papers (see [4]) similarly as the following generalization of it.

A det-mon-$\mathsf{hRLWW}(i)$-automaton can be used to model bottom-up, correctness preserving, context-free parsers. In order to model also bottom-up, correctness preserving, mildly context-sensitive parsers, a notion of j-monotonicity for restarting automata is used here; j-monotonicity was introduced in [8]. For an integer $j \geq 1$, an $\mathsf{hRLWW}(i)$-automaton is called j-monotone if, for each of its computations, the corresponding sequence of rewriting configurations can be partitioned into at most j (possibly noncontinuous) subsequences such that each of these subsequences is monotone. We use the prefix $\mathsf{mon}(j)$- to denote j-monotone types of $\mathsf{hRLWW}(i)$-automata.

A restriction of the form of restarting automata called *strong cyclic form* (see [3]) can also be transferred to $\mathsf{hRLWW}(i)$-automata. An hRLWW M is said to be in *strong cyclic form* if $|uv| \leq k$ for each halting configuration $¢uqv\$$ of M, where k is the size of the read/write window of M. Thus, before M can halt, it must erase sufficiently many letters from its tape. The prefix scf- will be used to denote restarting automata that are in strong cyclic form. The concept of strong cyclic form is useful for techniques of grammar-checking (localization of syntactic errors) by $\mathsf{hRLWW}(i)$-automata.

Lemma 6. *Let $i \geq 1$, and let M be an $\mathsf{RLWW}(i)$-automaton. Then there exists a scf-$\mathsf{RLWW}(i)$-automaton M_{scf} such that $L_{\mathrm{C}}(M) = L_{\mathrm{C}}(M_{\mathrm{scf}})$ and, for all words u, v, $u \Rightarrow^{c^*}_M v$ implies $u \Rightarrow^{c^*}_{M_{\mathrm{scf}}} v$. Moreover, all reductions of M_{scf} that are not possible for M are in contextual form. If M is deterministic and/or j-monotone for some $j \geq 1$, then M_{scf} is deterministic and/or j-monotone as well.*

Proof. Let $M = (Q, \Sigma, \Gamma, ¢, \$, q_0, k, i, \delta)$ be an $\mathsf{RLWW}(i)$-automaton. It is easy to see that the language L_{a} of words from Γ^* accepted by M in tail computations is a regular sublanguage of $L_{\mathrm{C}}(M)$. Therefore, there exists a deterministic finite automaton A_{a} such that $L(A_{\mathrm{a}}) = \{\, w \in L_{\mathrm{a}} \mid |w| > k \,\}$. Similarly, the language L_{r} of words rejected by M in tail computations is regular and there exists a deterministic finite automaton A_{r} such that $L(A_{\mathrm{r}}) = \{\, w \in L_{\mathrm{r}} \mid |w| > k \,\}$. Assume that the automata A_{a} and A_{r} have n_{a} and n_{r} states, respectively.

Now we can transform M into an $\mathsf{scf\text{-}RLWW}(i)$-automaton $M_{\mathrm{scf}} = (Q_{\mathrm{scf}}, \Sigma, \Gamma, ¢, \$, q_0, k_{\mathrm{scf}}, i, \delta_{\mathrm{scf}})$ of window size $k_{\mathrm{scf}} = \max\{k, n_{\mathrm{a}} + 1, n_{\mathrm{r}} + 1\}$. The transition relation δ_{scf} contains all transitions of M with the following exception. All accepting steps of M are replaced by MVL-steps into a new state $q_{\mathrm{l,a}}$. As M cannot rewrite in tail computations, when M_{scf} enters the state $q_{\mathrm{l,a}}$, the contents of its tape has not been changed since the last restart. In this case, M_{scf} moves to the left sentinel and starts to simulate A_{a}. During this simulation:

– either M_{scf} detects that the current tape contents is of length at most k_{scf} and it accepts,
– or M_{scf} detects that the current tape contents w is of length greater than k_{scf}; in this case, while moving to the right, it continues to simulate A_{a} until the right sentinel \$ appears in the window. From the pumping lemma for regular languages we know that if $w \in L_{\mathrm{a}}$, then there exists a factorization $w = w'xyz$ such that $|y| > 0$, $|y|+|z| \leq n_{\mathrm{a}}$, $w'xz \in L_{\mathrm{a}}$, and the word $xyz\$$ of length k_{scf} is the contents of the read/write window of M_{scf}. Accordingly, M_{scf} deletes the factor y and restarts. Even if there exist several such factorisations of w, for constructing M_{scf} we select one such factor for any contents of the read/write window $xyz\$$.

Finally, we must ensure that M_{scf} does not halt and reject for any word of length greater than k_{scf}. We can do that by adding new steps to the transition relation δ_{scf}. If $\delta(q, u)$ contains Reject for some state $q \in Q$ and some contents u of the read/write window, then we replace this reject step by a MVL-step into a new state $q_{\mathrm{l,r}}$, in which the automaton will move its window to the leftmost position, and then it starts to move to the right while simulating A_{r}. Similarly as above for A_{a}, during the simulation of A_{r}, the automaton either rejects if the current contents of the tape is not longer than k_{scf}, or it shortens the tape by applying the pumping lemma for L_{r}. Such a simulation of A_{r} is possible, as when M enters a configuration with state q and u in its read/write window, then it can halt, and hence, the tape contents has not been rewritten since the last restart (as M cannot rewrite in tail computations).

From the construction above we immediately see that M_{scf} is in strong cyclic form and that $L_{\mathrm{C}}(M_{\mathrm{scf}}) = L_{\mathrm{C}}(M)$. Moreover, if M is deterministic, then M_{scf} is deterministic, too. Additionally, if M is j-monotone, then M_{scf} is j-monotone, too, as the property of j-monotonicity is not disturbed by the delete operations at the very right end of the tape that are executed at the end of a computation. Moreover, all added reductions are in contextual form. □

3 On the Power and Sensitivity of Lexicalized Constructions

First we introduce the constraint of synchronization. We say that an $\mathsf{hRLWW}(i)$-automaton is *synchronized* if its degree of monotonicity is not higher than the number i of allowed rewrites per cycle. We denote the constraint of synchronization by the prefix syn-. In this paper we stress the relations of the degree of synchronization to the levels of the Chomsky hierarchy and we show that it can also be used for building a hierarchy within finite languages.

In this section we will study lexicalized constructions of scf-hRLWW(i)-automata. By lexicalized constructions we mean the basic and h-proper languages and hLSAs. We will see that with respect to lexicalized constructions, scf-hRLWW(i)-automata (and their variants) are sensitive to several types of constraints, as, e.g., the window size, the number of rewrites per cycle, and the degree of synchronization. Through these constraints we essentially refine the Chomsky hierarchy, and we will do so in two phases. In phase one we refine the context-sensitive languages by degrees of synchronization. Then, by using the window size, we refine the individual areas of lexicalized constructions that are given by the individual degrees of synchronization.

In order to present our results, we still have to introduce some additional notions. For any type X of RLWW(i)-automaton and any integer $j \geq 0$, we use fin(j)-X to denote the subclass of X-automata that perform at most j reductions in any accepting computation. Thus, for such an automaton, each accepting computation consists of up to j cycles only and a tail. Finally, by fin-X, we denote those X-automata that are of type fin(j)-X for some $j \geq 0$.

3.1 Small Finite Separating Witness Languages

This subsection represents the technical core of this section. It establishes the sensitivity of basic and h-proper languages of scf-hRLWW(i)-automata to the size of their windows, to the number of deletions by a reduction, and to the degree of monotonicity. This is achieved by constructions of small finite languages. In this way it is shown that the sensitivity relies on small syntactic observations.

Proposition 7. *Let $k \geq 2$, let a be a letter, and let $L_1(k) = \{a^k\}$. Then the following statements hold for $L_1(k)$:*

(a) $L_1(k) \in \mathcal{L}_{\mathrm{C}}(k\text{-scf-fin(0)-det-mon-RLWC})$.
(b) $L_1(k) \notin \mathcal{L}_{\mathrm{C}}((k-1)\text{-scf-hRLWW}) \cup \mathcal{L}_{\mathrm{hP}}((k-1)\text{-scf-hRLWW})$.

This proposition shows that for the separation of language classes based on the size of the read/write window it suffices to consider witness languages of cardinality one.

Proof. (a) Let $M_1(k)$ be the deterministic RLWC-automaton with window size k that proceeds as follows given a word $w = a^n$ as input:

1. If $n < k$, then $M_1(k)$ rejects in a tail computation.
2. If $n = k$, then $M_1(k)$ accepts in a tail computation after moving its window to the right sentinel.
3. If $n = i \cdot k$ for some $i \geq 2$, then $M_1(k)$ deletes the last occurrence of the letter a and restarts.
4. If $n = i \cdot k + j$ for some $i \geq 1$ and some $j \in \{1, 2, \ldots, k-1\}$, then $M_1(k)$ deletes the suffix a^k and restarts.

It is now easily seen that $L(M_1(k)) = L_{\mathrm{C}}(M_1(k)) = L_1(k)$, that $M_1(k)$ is in strong cyclic form, and that it is monotone (of degree 1). Further, as each accepting computation of $M_1(k)$ consists of just a tail, $M_1(k)$ is a fin(0)-RLWC-automaton.

(b) For any $(k-1)$-scf-hRLWW-automaton M, the language $L_{\mathrm{C}}(M)$ (and therewith the language $L_{\mathrm{hP}}(M)$) is either empty or it contains at least one word of length at most $k-1$. As $L_1(k)$ only contains a word of length $k > k-1$, we see that $L_1(k)$ is neither the basic language nor the h-proper language of any $(k-1)$-scf-hRLWW-automaton. □

Proposition 8. *Let $k, j \geq 1$, let a be a letter, and let $L_2(j,k) = \{a^{k\cdot(j+1)}, a^k\}$. Then the following statements hold for $L_2(j,k)$:*

(a) $L_2(j,k) \in \mathcal{L}_{\mathrm{C}}(k\text{-scf-fin(1)-det-mon-RLWC}(j))$.
(b) $L_2(j,k) \notin \mathcal{L}_{\mathrm{C}}(k\text{-scf-hRLWW}(j')) \cup \mathcal{L}_{\mathrm{hP}}(k\text{-scf-hRLWW}(j'))$ *for any* $j' < j$.

This proposition shows that for the separation of language classes based on the number of rewrites that can be executed during a cycle it suffices to consider witness languages of cardinality two.

Proof. (a) Let $M_2(j,k)$ be the deterministic RLWC-automaton with window size k that proceeds as follows given the word a^n as input:

1. If $n < k$, then $M_2(j,k)$ rejects in a tail computation.
2. If $n = k$, then $M_2(j,k)$ accepts in a tail computation.
3. If $n = i \cdot k$ for some $2 \leq i \leq j$, then $M_2(j,k)$ rewrites the word a^n into the empty word and restarts. Each of the rewrite steps deletes the suffix a^k of the current tape contents.
4. If $n = (j+1)\cdot k$, then $M_2(j,k)$ rewrites the word a^n into the word a^k and restarts. Each of the rewrite steps deletes the suffix a^k of the current tape contents.
5. If $n = i \cdot k$ for some $i \geq j+2$, then $M_2(j,k)$ simply deletes the last occurrence of the letter a and restarts.
6. If $n = i \cdot k + \ell$ for some $i \geq 1$ and $\ell \in \{1, 2, \ldots, k-1\}$, then $M_2(j,k)$ simply deletes the suffix a^k and restarts.

It follows that $L(M_2(j,k)) = L_2(j,k)$, that $M_2(j,k)$ is in strong cyclic form, and monotone (of degree 1). Further, each accepting computation of $M_2(j,k)$ consists of at most a single cycle and a tail, that is, $M_2(j,k)$ is a fin(1)-RLWC-automaton.

(b) Assume that M is a k-scf-hRLWW(j')-automaton such that $L_{\mathrm{C}}(M) = L_2(j,k)$, where $j' < j$. As $a^{k\cdot(j+1)} \in L_2(j,k)$ and $|a^{k\cdot(j+1)}| = k\cdot(j+1) > k$, and as M is in strong cyclic form, each accepting computation of M on input $a^{k\cdot(j+1)}$ begins with a cycle. As a^k is the only other word in $L_2(j,k)$, this cycle must rewrite $a^{k\cdot(j+1)}$ into the word a^k, for which $j\cdot k$ letters must be deleted. However, as M has window size k and can only execute $j' < j$ many rewrites per cycle, we see that it can delete at most $j'\cdot k < j\cdot k$ many letters in a single cycle. This implies that $L_{\mathrm{C}}(M) \neq L_2(j,k)$. Finally, as each word $w \in L_{\mathrm{hP}}(M)$ corresponds to a word of the same length from $L_{\mathrm{C}}(M)$, the argument above also shows that $L_{\mathrm{hP}}(M) \neq L_2(j,k)$. □

Proposition 9. *Let $k, j \geq 2$, $\Sigma = \{a,b,c\}$, and let $u_i = a^k$ for even i and $u_i = b^k$ for odd i. Finally, let*

$$L_3(j,k) = \{\, (u_i u_{i+1} \cdots u_j c^{k\cdot j^2})^j \mid i = 1, 2, \ldots, j \,\} \cup \{\, c^{k\cdot j\cdot r} \mid 0 \leq r \leq j^2 \,\}.$$

Then the following statements hold for $L_3(j,k)$:

(a) $L_3(j,k) \in \mathcal{L}_{\mathrm{C}}(k\text{-scf-fin}(j+j^2)\text{-det-mon}(j)\text{-RLWC}(j))$.

(b) $L_3(j,k) \notin \mathcal{L}_{\mathrm{C}}(k\text{-}\mathsf{scf}\text{-}\mathsf{mon}(j')\text{-}\mathsf{hRLWW}(j)) \cup \mathcal{L}_{\mathrm{hP}}(k\text{-}\mathsf{scf}\text{-}\mathsf{mon}(j')\text{-}\mathsf{hRLWW}(j))$ *for any* $j' < j$.

(c) $L_3(j,k) \notin \mathcal{L}_{\mathrm{C}}(k\text{-}\mathsf{scf}\text{-}\mathsf{hRLWW}(j')) \cup \mathcal{L}_{\mathrm{hP}}(k\text{-}\mathsf{scf}\text{-}\mathsf{hRLWW}(j')))$ *for any* $j' < j$.

This proposition shows that for the separation of language classes based on the degree of monotonicity it suffices to consider finite witness languages.

Proof. To simplify the notation we introduce, for each $i = 1, 2, \ldots, j$, the word $w_i = (u_i u_{i+1} \cdots u_j c^{k \cdot j^2})^j$ and the word $w_{j+1} = c^{k \cdot j^2 \cdot j}$.

(a) Let $M_3(j,k)$ be the deterministic RLWC-automaton that proceeds as follows given a word w as input. If $|w| \leq k$, then $M_3(j,k)$ performs a tail computation in which it accepts if $w = \lambda$ and rejects otherwise. If $|w| > k$, then $M_3(j,k)$ performs a cycle with the following rewrites completed by a restart step:

1. If $w = w_i$ for some $i \in \{1, 2, \ldots, j\}$, then $M_3(j,k)$ executes j rewrite steps that each delete a factor u_i. In this way w_i is rewritten into w_{i+1}.
2. If $w = c^{k \cdot j \cdot r}$ for some $r \in \{1, 2, \ldots, j^2\}$, then $M_3(j,k)$ deletes the suffix $c^{k \cdot j}$ by executing j rewrite steps that each delete the suffix c^k.
3. When w is not of any of the forms considered above, then $M_3(j,k)$ executes a rewrite step which either
   – deletes the last symbol of w, if $|w| = k \cdot \ell$, for some $\ell > 1$, or
   – deletes a k-letter suffix of w, if the length of w is not divisible by k.

Now it is easily seen that $L(M_3(j,k)) = L_3(j,k)$, that $M_3(j,k)$ is in strong cyclic form, and that each of its accepting computations consists of at most $j + j^2$ cycles and a tail. It remains to show that $M_3(j,k)$ is j-monotone.

If the input word w does not belong to the language $L_3(j,k)$, then in each cycle just a suffix is deleted, that is, the resulting computation is monotone. If $w = c^{k \cdot j \cdot r}$ for some $r \in \{1, 2, \ldots, j^2\}$, then the suffix $c^{k \cdot j}$ is deleted by j rewrite steps that all have right distance $k + 1$. Thus, the resulting accepting computation is monotone.

Hence, it remains to consider the first j cycles for the input $w_1 = (u_1 u_2 \cdots u_j c^{k \cdot j^2})^j$. In the first cycle the j factors u_1 are deleted, in the second cycle the j factors u_2 are deleted, and so forth. Now the leftmost rewrites in all these cycles yield a monotone sequence, the leftmost but one rewrites in all these cycles yield another monotone sequence, and so forth. Thus, we see that $M_3(j,k)$ is indeed j-monotone.

(b) Let M be a k-scf-$\mathsf{hRLWW}(j)$-automaton such that $L_{\mathrm{C}}(M) = L_3(j,k)$. We claim that M is not j'-monotone for any $j' < j$. Assume to the contrary that M is j'-monotone for some $j' < j$. As $w_1 \in L_3(j,k)$, M has an accepting computation for w_1 that is j'-monotone. In addition, as M is in strong cyclic form, this accepting computation must rewrite w_1 into a word of length at most k before it accepts. From the definition of $L_3(j,k)$ we see that this computation must start with the following sequence of cycles, as in each cycle M can delete at most $k \cdot j$ letters:

$$w_1 \Rightarrow^c_M w_2 \Rightarrow^c_M \cdots \Rightarrow^c_M w_j \Rightarrow^c_M w_{j+1}.$$

Thus, the rewrite (delete) steps that are executed in this sequence can be displayed as follows:

cycle no.	1-st rewrite	2-nd rewrite	$\cdots$	$(j-1)$-st rewrite	j-th rewrite
1	u_1	u_1	$\cdots$	u_1	u_1
2	u_2	u_2	$\cdots$	u_2	u_2
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$j-1$	u_{j-1}	u_{j-1}	$\cdots$	u_{j-1}	u_{j-1}
j	u_j	u_j	$\cdots$	u_j	u_j

The next table shows the corresponding right distances, where we disregard the right sentinel:

cycle no.	1-st rewrite	2-nd rewrite	$\cdots$	j-th rewrite
1	$j \cdot k \cdot (j^2+j)$	$(j-1) \cdot k \cdot (j^2+j)$	$\cdots$	$1 \cdot k \cdot (j^2+j)$
2	$j \cdot k \cdot (j^2+j-1)$	$(j-1) \cdot k \cdot (j^2+j-1)$	$\cdots$	$1 \cdot k \cdot (j^2+j-1)$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$j-1$	$j \cdot k \cdot (j^2+2)$	$(j-1) \cdot k \cdot (j^2+2)$	$\cdots$	$1 \cdot k \cdot (j^2+2)$
j	$j \cdot k \cdot (j^2+1)$	$(j-1) \cdot k \cdot (j^2+1)$	$\cdots$	$1 \cdot k \cdot (j^2+1)$

Now it can be checked easily that, for each $i \in \{1, 2, \ldots, j-1\}$, the right distance of the i-th rewrite in cycle j, which is $d_1 = (j+1-i) \cdot k \cdot (j^2+1)$, is larger than the right distance of the $(i+1)$-st rewrite in cycle 1, which is $d_2 = (j-i) \cdot k \cdot (j^2+j)$, since

$$d_1 = (j+1-i) \cdot k \cdot (j^2+1) = (j-i) \cdot k \cdot (j^2+1) + k \cdot (j^2+1),$$

while

$$d_2 = (j-i) \cdot k \cdot (j^2+j) = (j-i) \cdot k \cdot (j^2+1) + (j-i) \cdot k \cdot (j-1).$$

Hence,

$$\begin{aligned}\tfrac{1}{k} \cdot (d_1 - d_2) &= j^2 + 1 - (j-i) \cdot (j-1) = j^2 + 1 - (j^2 - j - i \cdot j + i) \\ &= 1 + j + i \cdot j - i = 1 + j \cdot (1+i) - i > 0.\end{aligned}$$

Hence, in order to arrange the j^2-many rewrite steps in the above computation into monotone subsequences, we need at least j such subsequences. This implies that the above computation is not j'-monotone for any $j' < j$. Thus, $L_3(j,k)$ is not the basic language of a k-scf-mon(j')-hRLWW(j)-automaton for any $j' < j$. The same argument also shows that $L_3(j,k)$ is not the h-proper language of such an automaton.

(c) Let M be a k-scf-hRLWW(j')-automaton for some $j' < j$. We will first show that $L_C(M)$ cannot be the language $L_3(j,k)$. Assume to the contrary that $L_C(M) = L_3(j,k)$. As $w_1 \in L_3(j,k)$, M has an accepting computation for w_1. In addition, as M is in strong cyclic form, this accepting computation must reduce w_1 into a word $v \in L_3(j,k)$. But this is impossible with window size k and less than j rewrites in a cycle. The same argument also shows that $L_3(j,k)$ is not the h-proper language of such an automaton. □

3.2 Sensitivity of scf-hRLWW(i)-Automata

Now we focus on results that are related to the sensitivity of scf-hRLWW(i)-automata. In particular, we show the sensitivity of these automata to the size of the window, to the number of rewrites in a cycle, and to the degree of monotonicity.

Corollary 10. *For all $j, k \geq 1$, the following hold:*

(1) $\mathcal{L}_{\mathrm{C}}(k\text{-}\mathsf{scf\text{-}fin}(1)\text{-}\mathsf{det\text{-}syn\text{-}RLWC}(j+1)) \smallsetminus \mathcal{L}_{\mathrm{hP}}(k\text{-}\mathsf{scf\text{-}hRLWW}(j)) \neq \emptyset$.
(2) $\mathcal{L}_{\mathrm{C}}((k+1)\text{-}\mathsf{scf\text{-}fin}(0)\text{-}\mathsf{det\text{-}syn\text{-}RLWC}(j)) \smallsetminus \mathcal{L}_{\mathrm{hP}}(k\text{-}\mathsf{scf\text{-}hRLWW}(j)) \neq \emptyset$.

Proof. The statement in (1) follows from Proposition 8, where $L_2(j,k)$ was shown to belong to $\mathcal{L}_{\mathrm{C}}(k\text{-}\mathsf{scf\text{-}fin(1)\text{-}det\text{-}mon\text{-}RLWC}(j))$, but not to $\mathcal{L}_{\mathrm{C}}(k\text{-}\mathsf{scf\text{-}hRLWW}(j')) \cup \mathcal{L}_{\mathrm{hP}}(k\text{-}\mathsf{scf\text{-}hRLWW}(j')))$ for any $j' < j$. Just observe that the automaton $M_2(j,k)$ for the language $L_2(j,k)$ is synchronized and that each of its accepting computations consists of at most one cycle and a tail.

The statement in (2) follows from Proposition 7, where $L_1(k) = \{a^k\}$ was shown to belong to $\mathcal{L}_{\mathrm{C}}(k\text{-}\mathsf{scf\text{-}fin(0)\text{-}det\text{-}mon\text{-}RLWC})$ and not to $\mathcal{L}_{\mathrm{C}}((k-1)\text{-}\mathsf{scf\text{-}hRLWW}) \cup \mathcal{L}_{\mathrm{hP}}((k-1)\text{-}\mathsf{scf\text{-}hRLWW})$. Recall that the automaton $M_1(k)$ for $L_1(k)$ is synchronized and that its only accepting computation consists just of a tail computation. □

Next we show a similar hierarchy with respect to the degree of monotonicity which is related to the number of rewrites in a cycle.

Corollary 11. *For all $j, k \geq 1$, the following hold:*

$$\mathcal{L}_{\mathrm{C}}(k\text{-}\mathsf{scf\text{-}fin\text{-}det\text{-}syn\text{-}RLWC}(j+1)) \smallsetminus \mathcal{L}_{\mathrm{hP}}(k\text{-}\mathsf{scf\text{-}mon}(j)\text{-}\mathsf{hRLWW}(j+1)) \neq \emptyset.$$

Proof. This result follows from Proposition 9. □

3.3 On Characterizations of Context-Free Constructions

In what follows we use LRR to denote the class of left-to-right regular languages. Several characterizations of LRR in terms of restarting automata can be found in [7].

Theorem 12. *Let $X \in \{\mathsf{hRLWW}(1), \mathsf{hRLWWD}(1), \mathsf{hRLWWC}(1)\}$. Then*
$\mathsf{LRR} = \mathcal{L}_{\mathrm{C}}(\mathsf{scf\text{-}det\text{-}syn}\text{-}X)$ *and* $\mathsf{CFL} = \mathcal{L}_{\mathrm{hP}}(\mathsf{scf\text{-}det\text{-}syn}\text{-}X)$.

Proof. An $\mathsf{hRLWW}(1)$-automaton is synchronized if and only if it is monotone. It is known that the basic languages of monotone $\mathsf{hRLWW}(1)$-automata are context-free [10]. As the class of context-free languages is closed under the application of morphisms, it follows that $\mathcal{L}_{\mathrm{hP}}(\mathsf{syn\text{-}hRLWW(1)})$ only contains context-free languages.

On the other hand, it is shown in [10] that the class CFL coincides with the class of h-proper languages of $\mathsf{det\text{-}mon\text{-}hRLWWC}(1)$-automata. Now we can use Lemma 6 to complete the proof. □

Remark. This theorem presents the robustness of the constraint of synchronization for several subclasses of $\mathsf{hRLWW}(1)$-automata with respect to basic and h-proper languages. It enhances the results about the robustness of context-free and LRR-languages.

Notation. By CFLA we denote the class $\mathcal{L}_{\mathrm{A}}(\mathsf{scf\text{-}det\text{-}syn\text{-}hRLWW}(1))$. With this notion we enhance the concept of context-freeness from formal languages to lexicalized syntactic analyses.

Corollary 13. *For all $X \in \{\mathsf{hRLWW}(1), \mathsf{hRLWWD}(1), \mathsf{hRLWWC}(1)\}$,*

$$\mathsf{CFLA} = \mathcal{L}_{\mathrm{A}}(\mathsf{scf\text{-}det\text{-}syn}\text{-}X).$$

Proof. Let us first recall the definition of $\mathcal{L}_\mathrm{A}(X)$. For a class of restarting automata X, a set of pairs L belongs to the class $\mathcal{L}_\mathrm{A}(X)$ if there are a restarting automaton $M \in X$ and a letter-to-letter homomorphism h such that $L = \{\, (h(w), w) \mid w \in L_\mathrm{C}(M) \,\}$. The proof then follows from the Theorem 12. $\square$

Remark. This corollary presents the robustness of the constraint of synchronization for several subclasses of $\mathsf{hRLWW}(1)$-automata with respect to the lexicalized syntactic analysis.

4 On Contextually Transparent Constructions

In this section we introduce and study classes of contextually transparent (lexicalized language) constructions (CTC) which are composed from infinitely many subclasses given by degrees of synchronization.

Notations. For $i \geq 1$, we denote by $\mathsf{CTL}(i)$ the class $\mathcal{L}_\mathrm{hP}(\mathsf{scf\text{-}det\text{-}syn\text{-}hRLWWC}(i))$ and by $\mathsf{CTLA}(i)$ the class $\mathcal{L}_\mathrm{A}(\mathsf{scf\text{-}det\text{-}syn\text{-}hRLWWC}(i))$. Taking the union over all positive integers we obtain the classes $\mathsf{CTL} = \bigcup_{i \geq 1} \mathsf{CTL}(i)$ and $\mathsf{CTLA} = \bigcup_{i \geq 1} \mathsf{CTLA}(i)$. We say that CTL is the set of *contextually transparent languages* and that CTLA is the set of *contextually transparent lexicalized analyses*.

Corollary 14. *For all $i \geq 1$, we have the following relations:*

(1)	$\mathsf{CFL} = \mathsf{CTL}(1)$,	$\mathsf{CFLA} = \mathsf{CTLA}(1)$,
(2)	$\mathsf{CTL}(i) \subset \mathsf{CTL}(i+1) \subset \mathsf{CTL}$,	$\mathsf{CTLA}(i) \subset \mathsf{CTLA}(i+1) \subset \mathsf{CTLA}$,
(3)	$\mathsf{CTL} \subset \mathsf{CSL}$.	

Proof. We just give outlines of the proofs. With the definitions in mind, claim (1) follows from Theorem 12 and Corollary 13, and claim (2) follows from Corollary 11.

Finally, the separation in (3) can be shown by using the context-sensitive language $L_e = \{\, a^{2^n} | n \geq 1 \,\}$. It is easily seen that the languages in CTL have the constant growth property, which is defined as follows (cf. [5]). Let X be an alphabet, let $L \subseteq X^*$. The language L is said to have the *constant growth property* if there are a constant $c_0 > 0$ and a finite set of positive integers C such that, for all $w \in L$ with $|w| > c_0$, there is a $w' \in L$ with $|w| = |w'| + c$ for some $c \in C$. Obviously, the language L_e does not have the constant growth property, and hence, it does not belong to the class CTL. $\square$

The sensitivity of $\mathsf{hRLWWC}(i)$-automata to the size of their windows can be utilized to essentially refine the hierarchies of CTLA. These refined hierarchies yield a fine classification of syntactic phenomena in lexicalized syntaxes of natural languages.

Recall that the prefix k- indicates the window size. So, k-$\mathsf{CTL}(i)$ is the class $\mathcal{L}_\mathrm{hP}(k\text{-}\mathsf{scf\text{-}det\text{-}syn\text{-}hRLWWC}(i))$; analogously for k-$\mathsf{CTLA}(i)$, k-CTLA, and k-CTL. We say that k-$\mathsf{CTL}(i)$ is the set of *k-transparent context-sensitive languages* of degree i, k-$\mathsf{CTLA}(i)$ is the set of *k-transparent context-sensitive lexicalized (sentence) analyses* of degree i, k-CTL is the set of *k-contextually transparent languages*, and k-CTLA is the set of *k-contextually transparent lexicalized analyses.* The next corollary easily follows from Corollary 10.

Corollary 15. *For all $i, k \geq 1$, the following relations hold:*

(1)	k-$\mathsf{CTL}(i) \subset k$-$\mathsf{CTL}(i+1) \subset k$-$\mathsf{CTL}$,	k-$\mathsf{CTLA}(i) \subset k$-$\mathsf{CTLA}(i+1) \subset k$-$\mathsf{CTLA}$,
(2)	k-$\mathsf{CTL}(i) \subset (k+1)$-$\mathsf{CTL}(i)$,	k-$\mathsf{CTLA}(i) \subset (k+1)$-$\mathsf{CTLA}(i)$.

For $i, k \geq 1$, we denote the class $\mathcal{L}_{\mathrm{hP}}(k\text{-scf-fin-det-syn-hRLWWC}(i))$ by k-$\mathsf{CTL}(i)\mathsf{FIN}$ and the class $\mathcal{L}_{\mathrm{A}}(k\text{-scf-fin-det-syn-hRLWWC}(i))$ by k-$\mathsf{CTLA}(i)\mathsf{FIN}$. Further, by k-CTLFIN we denote the union $\bigcup_{i\geq 1} k\text{-}\mathsf{CTL}(i)\mathsf{FIN}$ and by k-$\mathsf{CTLAFIN}$ we denote the union $\bigcup_{i\geq 1} k\text{-}\mathsf{CTLA}(i)\mathsf{FIN}$.

Corollary 16. *For all $i, k \geq 1$, the following relations hold:*

(1) $k\text{-}\mathsf{CTL}(i)\mathsf{FIN} \subset k\text{-}\mathsf{CTL}(i+1)\mathsf{FIN} \subset k\text{-}\mathsf{CTLFIN}$,
$k\text{-}\mathsf{CTLA}(i)\mathsf{FIN} \subset k\text{-}\mathsf{CTLA}(i+1)\mathsf{FIN} \subset k\text{-}\mathsf{CTLAFIN}$,
(2) $k\text{-}\mathsf{CTL}(i)\mathsf{FIN} \subset (k+1)\text{-}\mathsf{CTL}(i)\mathsf{FIN}$ *and* $k\text{-}\mathsf{CTLA}(i)\mathsf{FIN} \subset (k+1)\text{-}\mathsf{CTLA}(i)\mathsf{FIN}$.

Proof. These results also follow from Corollary 10. □

5 Conclusion

The $\mathsf{hRLWW}(i)$-automata satisfy the reduction correctness preserving property with respect to their basic and h-proper languages, and consequently also with respect to their lexicalized syntactic analysis and analysis by reduction. The basic correctness preserving property enforces the sensitivity to the degree of synchronization, number of rewrites in a cycle, and to the size of the window.

Thanks to the long time study of the PDT (Prague Dependency Treebank) we believe that class 12-$\mathsf{CTLA}(2)$ defined above is strong enough to model the lexicalized surface syntax of Czech, that is, to model the lexicalized sentence analysis based on PDT.

Our long term goal is to propose and support a formal (and possibly also software) environment for a further study and development of Functional Generative Description (FGD) of Czech (see [6]). We believe that the lexicalized syntactic analysis of full (four level) FGD can be described by tools very close to 24-$\mathsf{CTLA}(4)$.

We stress that our current efforts cover an important gap in theoretical tools supporting computational and corpus linguistics. Chomsky's and other types of phrase-structure grammars and the corresponding types of automata do not support lexical disambiguation, as these grammars work with categories bound to individual constituents with respect to constituent syntactic analysis. They do not support syntactic analysis with any kind of correctness preserving property, they do not support any type of sensitivity to the size of individual grammar (automata) rules (see several normal forms for context-free grammars, like Chomsky normal form [2]), and, finally, they do not support any kind of classification of finite syntactic constructions of (natural) languages.

On the other hand, in traditional and corpus linguistics, only finite language phenomena can be directly observed. Now the basic and h-proper languages of $\mathsf{hRLWWC}(i)$-automata in strong cyclic form with constraints on the window size allow common classifications of finite phenomena as well as classifications of their infinite relaxations. All these classifications are based on the reduction correctness preserving property and the strong cyclic form. Let us recall that for restarting and list automata the monotonicity means a synonymy for context-freeness. Here we are able to distinguish degrees of non-monotonicity of finite languages (syntactic phenomena), too.

Finally, note that many practical problems in computational and corpus linguistic become decidable when we only consider languages parametrized by the size of the windows, or even easier when they are parametrized by a finite number of reductions.

References

1. Y. Bar-Hillel: *A quasi-arithmetical notation for syntactic description.* Language, 29(1) 1953, pp. 47–58.
2. J. E. Hopcroft and J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1980.
3. P. Jančar, F. Mráz, M. Plátek, and J. Vogel: *Restarting automata*, in FCT'95, H. Reichel, ed., vol. 965 of LNCS, Dresden, Germany, August 1995, Springer, pp. 283–292.
4. T. Jurdziński, F. Mráz, F. Otto, and M. Plátek: *Degrees of non-monotonicity for restarting automata.* Theoretical Computer Science, 369(1–3) 2006, pp. 1–34.
5. L. Kallmeyer: *Parsing Beyond Context-Free Grammars*, Cognitive Technologies, Springer, 2010.
6. M. Lopatková, M. Plátek, and P. Sgall: *Towards a formal model for functional generative description: Analysis by reduction and restarting automata.* The Prague Bulletin of Mathematical Linguistics, 87 2007, pp. 7–26.
7. F. Mráz, F. Otto, and M. Plátek: *Characterizations of LRR-languages by correctness-preserving computations*, in NCMA 2018, Proc., R. Freund, M. Hospodár, G. Jirásková, and G. Pighizzini, eds., Vienna, 2018, Österreichische Computer Gesellschaft, pp. 149–164.
8. M. Plátek: *Two-way restarting automata and j-monotonicity*, in SOFSEM 2001: Theory and Practice of Informatics, 28th Conference on Current Trends in Theory and Practice of Informatics, L. Pacholski and P. Ružička, eds., vol. 2234 of LNCS, Berlin, 2001, Springer, pp. 316–325.
9. M. Plátek and F. Otto: *On h-lexicalized restarting automata*, in AFL 2017, Proc., E. Csuhaj-Varjú, P. Dömösi, and G. Vaszil, eds., vol. 252 of EPTCS, 2017, pp. 219–233.
10. M. Plátek, F. Otto, and F. Mráz: *On h-lexicalized automata and h-syntactic analysis*, in ITAT 2017, Proc., J. Hlaváčová, ed., vol. 1885 of CEUR Workshop Proceedings, 2017, pp. 40–47.
11. M. Plátek, D. Pardubská, and F. Mráz: *On (in)sensitivity by two-way restarting automata*, in ITAT 2018, Proc., S. Krajči, ed., vol. 2203 of CEUR Workshop Proceedings, 2018, pp. 10–17.

A Fast SIMD-Based Chunking Algorithm

Yehonatan Dude, Michael Hirsch, and Yair Toaff

Toga Networks, a Huawei Company
4 Ha'Harash St, Hod Hasharon, Israel
{johnny.david, michael.hirsch, yair.toaff}@toganetworks.com

Abstract. Deduplication is a special case of data compression where repeated chunks of data are stored only once. The input data is divided into chunks using a chunking algorithm and a cryptographically strong hash is calculated on each chunk and used as its unique identifier for further searching and duplicate elimination. As the input stream is processed, a chunk boundary is declared at a byte address in the input stream if some weak hash of a fixed number of preceding bytes (the "hash window") satisfies some criterion. Commonly, a rolling hash like Karp-Rabin [6] or some cyclic polynomial [7] is used for the weak hash since these cheaply support moving the hash window forward one byte in the input stream.
This work presents a way to calculate n weak rolling hashes at a time using single instruction multiple data (SIMD) instructions available on today's processors. Furthermore, it shows how to calculate chunk boundaries cheaply using other instructions also available on these processors. Empirical results show that the proposed algorithm is four times as fast as previous algorithms, and that these optimizations save up to 25% of the computation required for deduplication.

Keywords: chunking algorithm, deduplication, rolling hash

1 Introduction

In todays world, growing quantities of data need to be stored and/or transported. These enormous quantities of data present major costs and complexity challenges with respect to storage space and network bandwidth. Often, the data contains duplicates of data that is available elsewhere in some broader system. Data deduplication is a technique for reducing the data volume by eliminating this repeated data, reducing the storage and transportation requirements.

In the deduplication process of an input data stream, the input stream is divided into chunks which are compressed and stored uniquely in storage. The chunks may be entire files or objects, blocks of a block device, or content-aware variable length chunks. In deduplication, only one unique chunk of a data stream is actually retained while redundant data chunks (which are identical to an already retained data chunk) are replaced with pointers to their respective retained data chunks.

Figure 1 shows an example of an input data stream with two versions of a text file. A small insert or delete in the content of the file shifts the remainder of the file. In the case of file deduplication, this results in the need to store the entire file. In the case of block deduplication, all the blocks including the change and after it need to be retained. By contrast, in the case of content aware deduplication, only the changed chunk needs to be retained.

Content-aware deduplication usually provides the best results with respect to data reduction. The problem with content-aware variable length deduplication is that it is computationally expensive and adds latency to the process. This paper focuses on

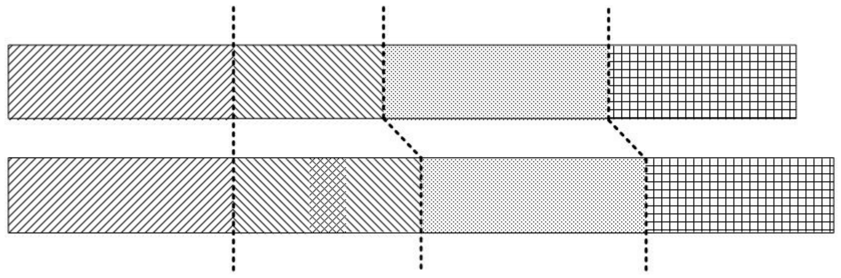


Figure 1. In this example we can see two nearly identical data streams. The second contains a small portion of inserted data. By splitting the stream into chunks using a content aware chunking algorithm, only the chunk containing the inserted data will be retained. The other chunks will be replaced by pointers to the chunks of the first stream.

one of the steps in content-aware variable length deduplication, showing how to chunk data approximately 4 times faster than accepted practice.

The process of content aware deduplication is composed of four CPU intensive parts: chunking, hashing, searching, and compressing. First, the input data stream is divided into chunks using a content aware chunking algorithm. Then a probabilistically unique hash value is generated for each chunk using a hash algorithm. Usually, a cryptographical strength hash algorithm is used for this purpose, for example SHA-2, mainly for its probabilistically unique properties and optimizations that CPU vendors have implemented. Some kind of index data structure maps existing hashes to their locations. This data structure is then searched using this hash value. If the hash is found, the chunk is not retained but rather replaced by a pointer to the existing chunk. If the hash is not found, then it is retained: the chunk is compressed to further reduce storage efficiency, stored, and the hash and its location added to the index data structure.

In this paper we present a chunking algorithm that exploits single instruction multiple data (SIMD) technology in order to process vectors of bytes rather than single bytes, and other instructions to cheaply calculate the criteria for chunk boundaries.

The methods discussed here complements previous work [2,3,4,5] in the field by one of the authors.

This paper is organized as follows. In Section 2, we survey previous algorithms. In Section 3, we present our rolling hash function and an algorithm to calculate it efficiently. In Section 4, we compare its performance with previous algorithms. Conclusions are in Section 5.

2 Related Work

Throughout the paper we will make use of the following notation and terminology. A string x of length $|x|$ is represented as a finite array $x_1x_2\cdots x_n$ of characters from a finite alphabet Σ of size σ. We refer to the i-th element in x as x_i and use the notation $x_i\cdots x_j$ to denote the subsequence of x from the element at position i to the element at position j including both, where $1 \leq i \leq j \leq |x|$.

"Chunking" is a way to split a data stream on context sensitive boundaries. The main goal of effectively chunking the data stream is to ensure that the chunk boundaries are affected as little as possible by changes to the chunks' data contents. A chunk boundary is designated when a (weak) hash of some n consecutive bytes complies with one or more predefined chunking criteria. In this way, only the last n consecutive bytes affect the boundary, not changes within the chunk.

Calculating even a weak hash at every byte offset in a data stream is expensive. In general, in the industry, rolling hash techniques are used for chunking data streams. A rolling hash is a hash that can be calculated either as a function of a sequence of n bytes $x_i \cdots x_{i+n-1}$ or as a function of rolling hash of $x_{i-1} \cdots x_{i+n-2}$, and the values x_{i-1} and x_{i+n-1}. The latter is usually significantly cheaper to compute. The term "rolling" is used to illustrate how the hash "rolls" from one byte window to the next by removing the effect of the byte leaving the sequence and adding the effect of the byte entering the sequence. As before, the calculated hash value is checked for compliance with some predefined one or more chunking criteria and in case the compliance is identified, the end of the respective rolling sequence is designated as a chunk boundary.

```
Input: A string $x = x_1 x_2 \cdots x_l$
Result: Chunk Boundary Positions[]
for $i \leftarrow w$ to $l$ do
    $h \leftarrow$ rolling hash of $x_{i-w+1} \cdots x_i$;
    if criterion(h) then
        Append $i$ to Chunk Boundary Positions;
    end
end
```

Algorithm 1: Content Aware Chunking

Algorithm 1 provides a brief implementation of a chunking algorithm using a rolling hash [8], such as cyclic polynomial [7] or Karp-Rabin [6]. As an example, the criterion function here could be true if $h = 0 \bmod N$ and false otherwise, where N is the average chunk size. An alternative criterion function could be accomplished by comparing a bit masked version of the hash to a number, which will give 2^n average chunk size for mask with n bits.

The problem is that chunking in algorithm 1 is CPU-intensive because it iterates over all the bytes. Also, at each byte, there is a certain amount of computation that depends on results computed in the previous iteration. This forces the iteration to rely on sequential execution of the algorithm. In fact, it is so expensive that it takes significantly more time to chunk the data than to hash it using a cryptographic strength hash algorithm (see Section 4.4). On paper, cryptographic strength hashes do more work, but they were designed for parallel execution and modern CPUs have further optimized hardware for them.

This problem exists in many of the proposed algorithms and optimizations, including ones exploiting SIMD, SSE, and other modern CPU architectures.

3 A Fast Chunking Algorithm

This section is divided into two parts. The first is the description of the rolling hash function, and the second is how to calculate it efficiently.

3.1 The Rolling Hash and Boundary Criteria

The hash function we are about to define is designed as a chunking algorithm. Let x be the input data string over an alphabet Σ of size σ. Let k be the size of a vector we use for calculations, l be the number of bits in each element, such that $\sigma \leq 2^l$, and

$w = kl$ be the window size of the rolling hash. We will use $\oplus$ to denote bitwise xor, $|$ to denote bitwise or, & for bitwise and, $\ll$ for bitwise shift left, and $\gg$ for bitwise shift right.

The rotate n bits left of a l bits charecter c function could be written this way:

$$rol(c, n) = \big((c \ll n) \mid (c \gg (l - n))\big) \,\&\, (2^l - 1)$$

We define an intermediate hash h_i at a position $i > w$ in the string x as following:

$$h_i = \bigoplus_{j=1}^{l} rol(x_{i-k(l-j)}, l - j)$$

In the algorithm we discuss here, calculating the criterion function of a vector of intermediate hashes is a two stage process. First, we calculate an intermediate criterion vector of the intermediate hashes. Then the final criterion holds only when a sequence of k intermediate criteria hold. The probability of a boundary to be declared is then the probability of k intermediate criteria holding.

Some criterion function c operates on one intermediate hash value. We define a intermediate criterion result c_i at position i in string x as following:

$$c_i = \begin{cases} 0 & \text{if } c(h_i) \text{ holds} \\ 1 & \text{otherwise} \end{cases}$$

The final criterion C_i for a chunk boundary at position i is a combination of all k preceding intermediate criteria $c_{i-k+1}, c_{i-k+2}, \ldots, c_i$.

$$C_i = \begin{cases} \text{true} & \text{if } \left(\sum_{j=i-k+1}^{i} c_i\right) = 0 \\ \text{false} & \text{otherwise} \end{cases}$$

If C_i is true, a boundary is designated at position i.

Let's show how C_i is calculated with an example. Suppose we have $l = 4$ bits, and a vector of size $k = 4$, and window size of $w = 16$, and let's use the criterion function $c = (b >= h)$, and hence $c_i = (b < h_i)$ (using 0 as "false"). The following four conditions show how to calculate $c_{13} \cdots c_{16}$:

$$\begin{aligned} b < h_{13} &= rol(x_1, 3) + rol(x_5, 2) + rol(x_9, 1) + rol(x_{13}, 0) \\ b < h_{14} &= rol(x_2, 3) + rol(x_6, 2) + rol(x_{10}, 1) + rol(x_{14}, 0) \\ b < h_{15} &= rol(x_3, 3) + rol(x_7, 2) + rol(x_{11}, 1) + rol(x_{15}, 0) \\ b < h_{16} &= rol(x_4, 3) + rol(x_8, 2) + rol(x_{12}, 1) + rol(x_{16}, 0) \end{aligned}$$

Since k is 4, if all 4 $c_{13} \cdots c_{16}$ evaluate to 0, then C_{16} holds, designating a boundary at position 16.

There are a couple of "tricks" here. Firstly, we use a single SIMD instruction to convert a vector of hash values to bits in a computer word that represent whether or not they meet the criterion function (for example _mm256_cmpgt_epi8[1]). Secondly, we use instructions that count sequences of zero bits (for example _tzcnt_u32[1]). In this way, we can count the number of consecutively satisfied conditions in $O(1)$ computer instructions. This combination makes this algorithm efficient.

Below, we discuss how to calculate k intermediate hashes at a time.

3.2 Boundary Criteria Algorithm

Our algorithm is divided into three parts, the first initializes the intermediate hashes, the second checks if there is a boundary at any position among the last intermediate hashes calculated, and third calculates the next intermediate hashes by inserting and omitting new and old vectors (the rolling part). The algorithm operates as following:

```
Input: A string $x_1 x_2 \cdots x_n$
Result: Chunk Boundaries (Positions)

// Part 1 - Initialize hashes
$H \leftarrow 0, 0, \ldots, 0$;
for $i \leftarrow 0$ to $l - 1$ do
    $j \leftarrow ik$;
    $V \leftarrow x_j x_{j+1} \cdots x_{j+k-1}$;
    $H \leftarrow \mathrm{rol}(H, 1) \oplus V$;
end
$i \leftarrow lk$;
$p \leftarrow 0$;
// Iterate over the string
while $i + k \leq n$ do
    // Part 2 - Check for boundaries
    $B \leftarrow b, b, \ldots, b$;
    $C \leftarrow H \leq B$;
    $c \leftarrow \mathrm{mask}(C)$;
    $s \leftarrow p + \mathrm{count\_leading\_zeros}(c)$;
    if $s \geq k$ then
        mark a boundary at $i - p$;
    end
    $p \leftarrow \mathrm{count\_trailing\_zeros}(c)$;
    // Part 3 - Calculate the next $k$ hashes
    $N \leftarrow x_i x_{i+1} \cdots x_{i+k-1}$;
    $O \leftarrow x_{i-kl} x_{i-kl+1} \cdots x_{i-kl+k-1}$;
    $H \leftarrow \mathrm{rol}(H, 1) \oplus N \oplus O$;
    $i \leftarrow i + k$;
end
```

Algorithm 2: Generate Chunk Boundaries on an input String

The first part calculates the rolling hash value at the start.

The second part checks for boundaries. In the implemented version of the algorithm we use 0 to denote passing of the criterion c_i in order to count the leading and trailing zeros of the resulting bitmask register. By counting two integers, k bits of length, representing sequential $2k$ intermediate criteria results, we ensure finding any k sequences of pass results. The second integer is saved to be used in the next iteration after we calculate the next k criteria.

And in the third part, we move forward with rolling calculation of the intermediate hashes of the next vector (of k bytes).

4 Results

This section focuses mainly on empirical results we obtained from benchmark tests executed with an AVX version of the code. Our main goal was to produce an algorithm comparable to that of Karp-Rabin in terms of its quality and usefulness for chunking for deduplication, namely keeping the distribution of chunk size similar, but speeding up the calculations.

All tests were performed using AVX instructions which are present in Intel CPUs that are commonly found in today's data center servers. If we had used the newer AVX3 instructions and vector length, we would expect to get a further 3 to 4 factor performance increase.

In this section, we present results from several aspects. The first is a table showing the amortized cost of the different instructions when calculating Karp-Rabin and SIMD chunking. Secondly, we show a simple speed comparison. Thirdly, we show that the chunk sizes resulting from both algorithms on the same data sets are similar. Lastly, we show the effect of this speedup on the overall system performance.

4.1 Workloads Comparison

Command	Cyclic Polynomial	SIMD Optimized
Shift	4	2/16
And	1	0
Or	2	1/16
Xor	2	2/16
Other	2	4/16
Total	11	9/16

Table 1. Number of CPU operations per byte comparison

Different arithmetic operations require different amounts of time. Division for example takes more time than xor. However, for similar operations, the execution time of regular arithmetic operations and that of SIMD arithmetic operations are similar.

In table 1, we can see a comparison between Cyclic Polynomial and SIMD Optimized Chunking written using AVX instructions with a vector of 16 bytes, and a window size of 64 bytes. Theoretically, the rolling part should be 20 times faster.

4.2 Chunk Size Distribution Results

As we noted before, one of the goals was to keep a similar chunk size distribution, because it implicitly affects many aspects of the deduplication system.

For this test, we used a generated corpus of data that emulates the data we expect to work with in a real storage system. We processed the data both with Karp-Rabin and SIMD Optimized Chunking, and counted the number of chunks in each range of sizes, consisting of 32 buckets of size ranges, each containing about 447 discrete sizes.

In figure 2, we see visually that the chunk size distribution is very similar. In fact, they differ by less than 2%. The end result, that is the quality of the deduplication, is unchanged.

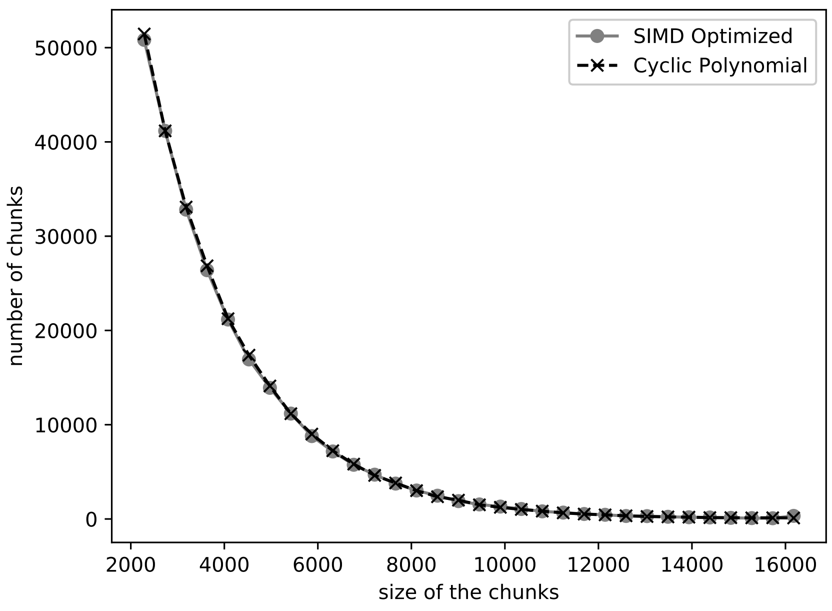


Figure 2. Chunk Size Distribution Comparison

Algorithm	Random Data	Mixed Data
Karp-Rabin	975 MB/s	927 MB/s
Cyclic Polynomial	1675 MB/s	1676 MB/s
SIMD optimized	6715 MB/s	7136 MB/s

Table 2. Benchmark of the chunking algorithms

4.3 Chunking Performance Results

In table 2, we see the performance results from executing the chunking algorithms alone. Karp-Rabin, Cyclic Polynomial, or SIMD Optimized Chunking are compared. Each algorithm was run on two sets of data: random data, and the same corpus data used previously to test the chunk size distribution.

4.4 Deduplication Performance Results

We also ran an emulation of an entire deduplication system, including chunking, hashing, and compressing data. The process was limited to a single thread on a single CPU core.

Chunking Algorithm	MB/s	CPU Usage			
		LZ-4	Sha-1	Other	Chunking
Karp-Rabin	262	63.9%	4.1%	3.8%	28.1%
SIMD Optimized	345	84.5%	5.4%	5.1%	4.7%

Table 3. Benchmark of deduplication

In table 3, we see that the throughput of the deduplication system increased by 32% from 262MB/s to 345MB/s. We also show how the CPU usage is split between the most CPU intensive parts.

5 Conclusions

This paper has shown a new way to calculate a rolling hash. It shows a way to "roll" the hash an entire vector at a time. The process of rolling a whole vector at a time maps cleanly onto SIMD instructions available in today's CPUs, making for fast implementations. In this way, only a fraction of the time is needed to calculate the hashes compared to previous methods. In a further step, this paper also shows how to evaluate a criterion function a whole vector at a time, and in so doing, yet more saving is made to the overall process.

References

1. *Intel intrinsics guide*: `http://software.intel.com/sites/landingpage/IntrinsicsGuide`.
2. L. Aronovich, R. Asher, D. Harnik, M. Hirsch, S. T. Klein, and Y. Toaff: *Similarity based deduplication with small data chunks*, in Proceedings of the Prague Stringology Conference 2012, J. Holub and J. Žd̆árek, eds., Czech Technical University in Prague, Czech Republic, 2012, pp. 3–17.
3. M. Hirsch, A. Ish-Shalom, and S. T. Klein: *Optimal partitioning of data chunks in deduplication systems*, in Proceedings of the Prague Stringology Conference 2013, J. Holub and J. Žd̆árek, eds., Czech Technical University in Prague, Czech Republic, 2013, pp. 128–141.
4. M. Hirsch, S. T. Klein, D. Shapira, and Y. Toaff: *Controlling the chunk-size in deduplication systems*, in Proceedings of the Prague Stringology Conference 2015, J. Holub and J. Žd̆árek, eds., Czech Technical University in Prague, Czech Republic, 2015, pp. 78–89.
5. M. Hirsch, S. T. Klein, and Y. Toaff: *Improving deduplication techniques by accelerating remainder calculations*, in Proceedings of the Prague Stringology Conference 2011, J. Holub and J. Žd̆árek, eds., Czech Technical University in Prague, Czech Republic, 2011, pp. 173–183.
6. R. M. Karp and M. O. Rabin: *Efficient randomized pattern-matching algorithms*, 1987.
7. D. Lemire and O. Kaser: *Recursive hashing and one-pass, one-hash n-gram count estimation.* CoRR, abs/0705.4676 2007.
8. M. O. Rabin: *Fingerprint by random polynomials*, 1981.

Bidirectional Adaptive Compression

Aharon Fruchtman[1], Shmuel T. Klein[2], and Dana Shapira[1]

[1] Dept. of Computer Science, Ariel University, Ariel 40700, Israel
aralef@gmail.com, shapird@g.ariel.ac.il

[2] Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
tomi@cs.biu.ac.il

Abstract. A new dynamic Huffman encoding has been proposed in earlier work, which instead of basing itself on the information gathered from the already processed portion of the file, as traditional adaptive codings do, uses rather the information that is still to come. The current work extends this idea to *bidirectional* adaptive compression, taking both past and future into account, and not only performs at least as good as static Huffman, but also provably improves on the future-only based variant. We give both theoretical and empirical results that support the enhancement of the new compression algorithm.

1 Introduction

Data compression techniques are often classified according to various criteria. One of the popular partitions is into static and adaptive variants. Alternatively, they can be arranged by whether being statistical or dictionary based methods. We focus in this paper on adaptive statistical compression such as Huffman [7] and arithmetic coding.

Research in data compression has evolved in several directions over the years, e.g., *compressed pattern matching* in texts [1,19], in images [11,14] and in structured files [12,3], *compact data structures* [8,18,15,2] and *cryptosystems* [16]. The current paper is yet another research regarding the core of encoding such as [5,23,7,4,20,17,13].

The traditional approach to adaptive coding updates the model dynamically according to what has already been seen in the file processed so far. The distribution of the following item to be encoded at some current location in the file is determined according to the distribution of the elements that have occurred up to that point. A different adaptive approach is suggested in [9], assuming that the exact statistics of the number of occurrences of each element in the entire file are known. For example, these statistics could have been collected in a first, preprocessing, pass over the file. In this approach, the dynamic model adapts itself while processing the file by using its knowledge of what is still to come, i.e., it looks into the *future*, rather than what is done by the traditional dynamic methods, which base their current model on what has already been seen in the *past*.

This "looking into the future" paradigm has also been suggested in [10] for performing stream compressed matching in LZSS [20], where instead of letting the Ziv-Lempel type (offset, length) pairs point backward to reoccurring strings, the locations of these pointers were moved and their direction was reversed to point forward.

Classical dynamic compression algorithms focus only on the item that is currently processed and increments its frequency, which may consequently imply a shorter corresponding codeword in future usage. However, as a consequence these savings may come at the price of having certain other codewords lengthened. The forward approach improves upon this "egoistic" behavior by a more social approach of the forward looking variant, where the frequency of the currently processed element is *decreased*, even at the price that the corresponding codeword can become longer. However, this operation may shorten the overall codeword length of those elements that are still present in the tree, yielding a better space savings.

2 A new hybrid adaptive coding method

The performance of both the classical static Huffman coding and its dynamic variants suffers from the fact that they use some information about the distribution of the characters to be encoded which is not necessarily needed. Moreover, this additional information has a negative impact on the compression ratio that may be achieved. More precisely, static Huffman coding uses throughout the frequencies of the characters in the entire text, yet the occurrences of these characters are not necessarily spread uniformly. In an extreme case in which a certain character x is clustered locally and does not appear elsewhere, the appearance of a leaf assigned to x in the Huffman tree is a burden, reducing the compression gain for the parts of the text outside of these clusters.

As another example, consider Huffman's dynamic variant, as proposed by Vitter [22], and an input text T consisting first of a long string of characters from $\{a, b, \dots, z\}$, followed by a long string of numbers from $\{0, 1, \dots, 9\}$. While for the beginning of the file, the Huffman tree will only have 27 leaves (including one for the not yet transmitted elements), the Huffman tree for the second part will have 37 leaves, including the newly encountered digits. However, the fact that non-numeric characters do not appear in the second part is not taken advantage of, which could have reduced the average codeword length.

The forward looking dynamic Huffman coding [9] starts with the full frequencies, as for the static algorithm, and then decreases them gradually after the occurrence of each of the characters. The extreme case of the input file T will then imply a symmetric behavior, in which the encoding of the first part of the file will be based on a full Huffman tree with 37 leaves and therefore be wasteful, and only in the second part will the tree be reduced to 11 nodes for the set of digits alone.

To remedy this potential source of inefficiency, we propose the following hybrid method taking advantage of both forward and backward looking dynamic Huffman coding and thereby correcting some of their drawbacks.

The idea is to start with the same tree as the traditional dynamic Huffman encoding. That is, at the beginning, the Huffman tree contains only a special leaf, symbolizing the set of characters that has not yet occurred in the text read so far. This special character is often labelled NYT for *Not Yet Transmitted*. The corresponding codeword is the empty string. The forward looking dynamic variant requires the entire distribution of the alphabet to be known at the beginning of the process, whereas the classical backward looking dynamic method of Vitter transmits the alphabet incrementally, each character immediately after its first occurrence. For Vitter's method, there is no need to transmit the character frequencies, which are updated on the encoder and decoder sides in synchronization.

For the hybrid method, we again assume that the whole distribution of the characters in the entire input file is known to the encoder, as for static encoding, but that this distribution will not be transferred to the decoder in advance as is done in the forward looking dynamic method. We now suggest transmitting a newly encountered character y immediately at its first occurrence, for example by issuing the

Huffman codeword for NYT, followed by some encoding of the new character y, e.g., in ASCII. The innovation is that at this point, we also transmit the frequency of y in the remaining part of the file.

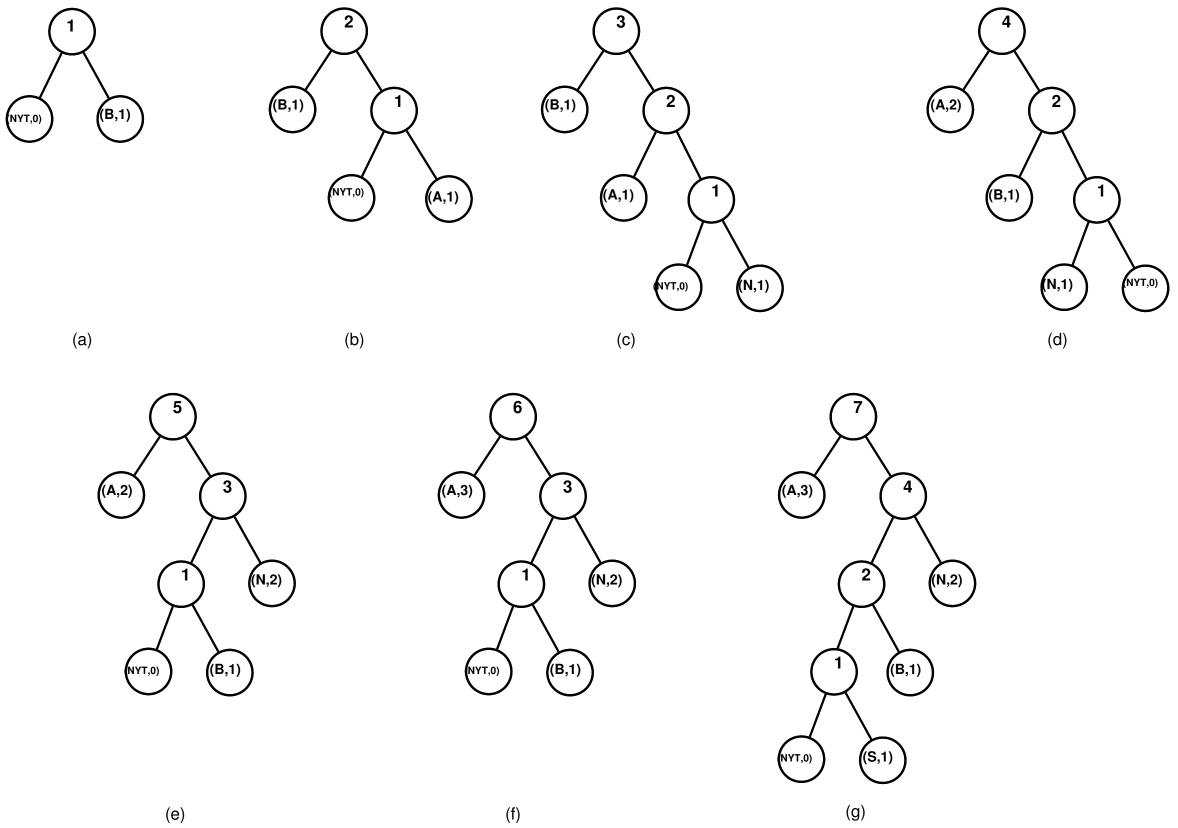


Figure 1. Illustration of the standard VITTER adaptive Huffman algorithm for T = BANANAS.

The expected savings are of course not in the transmitted statistics of the characters, since instead of sending them as a bulk in a header at the beginning, they are spread within the encoded file. The real advantage lies in the fact that on the one hand, at each point, the current Huffman tree reflects only the set of characters in the prefix of the file so far, as in the backward looking variant, but that the frequencies are according to the suffix of the file, as in the forward looking variant, which is better from the compression point of view. Moreover, characters are altogether eliminated from the tree after their last occurrence, which in extreme cases, when the suffix only includes a small subset of the original alphabet, may yield significant savings.

3 Bidirectional Adaptive Coding

The bidirectional adaptive compression can be adapted to any statistical dynamic compression such as dynamic Huffman, adaptive arithmetic coding and Prediction by Partial Matching (PPM) [4]. The generic encoding algorithm, named HYBRID-ENCODE, is given in Algorithm 1. Decoding is done symmetrically. In a preprocessing stage, the text T is scanned in order to gather the underlying statistics storing the frequency freq(σ_i) for each character σ_i in the alpahbet Σ. The tree is initialized by NYT having as frequency the size of the alphabet, since this is the number of times it will be used. The text $T = x_i \cdots x_n$ is rescanned. In case a character x_i is encountered for the first time, the codeword for NYT is transmitted followed by the ASCII encoding of x_i, and some encoding of the frequency $\mathsf{freq}(x_i)$. One of the possible choices for the encoding method of these frequencies could be Elias's C_δ code [5], a universal encoding method of the integers ≥ 1 using about $\log x + \log\log x$ bits to encode the value x. Otherwise, x_i is already in the model and encoded accordingly, its frequency is decremented by 1 and the model is updated.

Algorithm 1: HYBRID-ENCODE

```
HYBRID-ENCODE($T = x_1 \cdots x_n$ )
1  preprocess $T$ to get freq($\sigma_i$),   $\forall \sigma_i \in \Sigma$
2  initialize the model with a single element, for NYT, with freq(NYT)) $\leftarrow |\Sigma|$
3  encode freq(NYT)
4  for $i \leftarrow 1$ to $n$ do
5      if $x_i$ has already appeared earlier then
6          encode $x_i$ according to current model
7          freq($x_i$) $\leftarrow$ freq($x_i$) $- 1$
8      else // first occurrence
9          encode NYT according to the current model
10         freq(NYT) $\leftarrow$ freq(NYT) $- 1$
11         output ASCII($x_i$)
12         encode freq($x_i$)
13         Update the model with $x_i$, freq($x_i$) and freq(NYT)
```

Consider as example the text T = `BANANAS` over the alphabet {`A`, `B`, `N`, `S`} with corresponding weights $\{3, 1, 2, 1\}$. Figure 1 shows the way the Huffman tree alters for the standard adaptive Huffman coding of [22] while T is processed. The tree is initialized with the NYT node with 0 weight. When `B` is encountered, ASCII(`B`) is output and the letter `B` is inserted as a new leaf into the Huffman tree with weight 1 resulting in the tree presented in Figure 1(a). Note that for this standard algorithm, there is no need to transmit the frequency of `B`, which will be learned by the decoder incrementally while processing the remaining part of the file. When `A` is processed, the NYT with codeword 0 is output, followed by ASCII(`A`), and a new leaf for `A` is inserted resulting in Figure 1(b). The following character `N` is dealt with similarly, the codeword of NYT is now 10, followed by ASCII(`N`) (Figure 1(c)). To process the second `A`, and then the second `N`, the corresponding codewords 10 and 110 are output, their weights are incremented, and the tree is updated to Figure 1(d) and then to Figure 1(e). The codeword for the last `A` is 0, and while the structure of the tree remains the same, `A`'s weight is incremented to 3 (Figure 1(f)). For the last character `S`, NYT is encoded by 100 and followed by ASCII(`S`), and the process stops (Figure 1(g)).

The FORWARD algorithm basically works in the opposite way, starting with the final tree of the dynamic variant, and ending with the empty tree. The main difference is that a special node for NYT and the transmission of the frequencies is not needed, as the entire alphabet and its statistics are assumed known to the decoder. Our running example for FORWARD-HUFFMAN is presented in Figure 2.

The initial tree, shown in Figure 2(a), contains statistics for the entire alphabet, exactly as for the static version of Huffman coding. When the only appearance of `B` is processed, it is encoded by 101, and then `B` is removed from the tree, resulting in

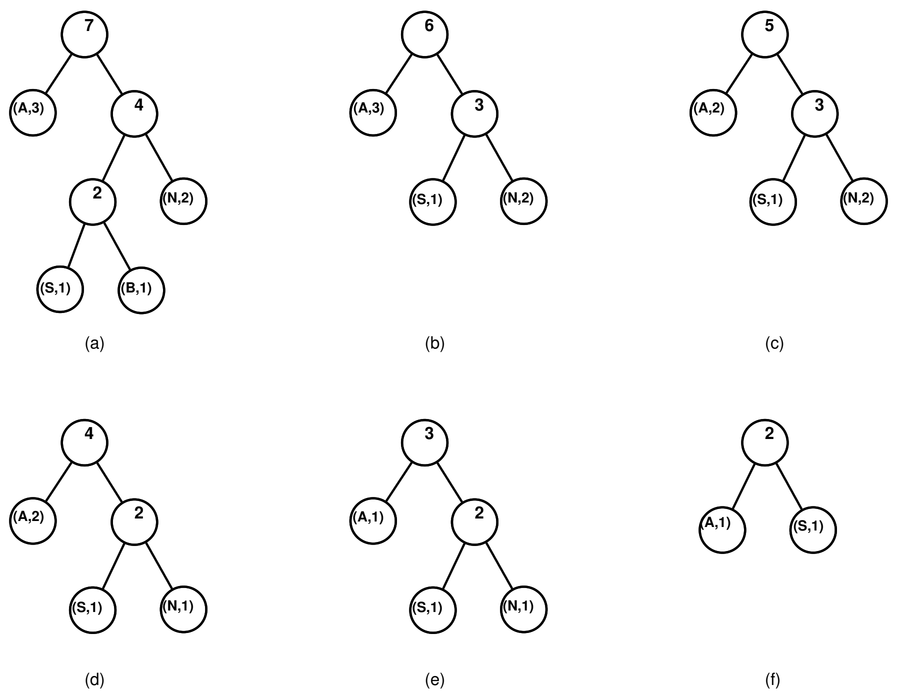


Figure 2. Illustration of FORWARD adaptive Huffman algorithm for $T =$ `BANANAS`.

the tree of Figure 2(b). The following characters `A`, `N`, and `A` generate 0, 11, and 0 as output, each followed by a decrement of their correnponding frequencies, with no other changes, yielding the trees Figure 2(c), Figure 2(d) and Figure 2(e), respectively. When the last `N` is processed, 11 is output again, and `N` is removed from the tree, resulting in the tree of Figure 2(f). When the last `A` is processed, 0 is output, and `A` is removed from the tree, resulting in a tree with a single node corresponding to `S`, thus no further bits need to be transferred to the decoder, who already realizes that the remaining suffix of the file may only contain a single character, which must be `S`, and which repeats freq(`S`) times.

The proposed hybrid algorithm starts with a tree containing the NYT node, but unlike for the standard dynamic Huffman algorithm, its frequency is initialized by $|\Sigma|$, which is 4 in our case. Each time a new character is encountered, the procedure uses its knowledge of the "future" and updates the exact frequency, which, subsequently, alters the model. When `B` is processed, it does not need to be inserted into the tree, as it only occurs once, and ASCII(`B`) is output, along with a single bit 1, which is the C_δ encoding of its frequency 1, resulting in the tree of Figure 3(b). When the following character `A` is encountered, having frequency 3, there is no need to encode NYT, which is still the only leaf in the tree, so only ASCII(`A`) is output, followed by $C_\delta(3) = 0101$, and `A` is inserted into the tree with frequency 2 (Figure 3(c)). To process `N`, NYT is now encoded by 0, followed by ASCII(`N`) (Figure 3(d)), followed by $C_\delta(2) = 0100$. `A` is then encoded by 0 (Figure 3(e)), `N` is encoded by 11 and removed from the tree (Figure 3(f)), and `A` is encoded as 0, and the tree is updated to contain again only NYT (Figure 3(g)). For `S`, which occurs only once in T, only its ASCII code is output, followed by its frequency $C_\delta(1) = 1$ (and not preceded by NYT).

Figure 4 summarizes this example in a comparative chart, showing the binary output sequences produced by the different approaches. For the standard VITTER

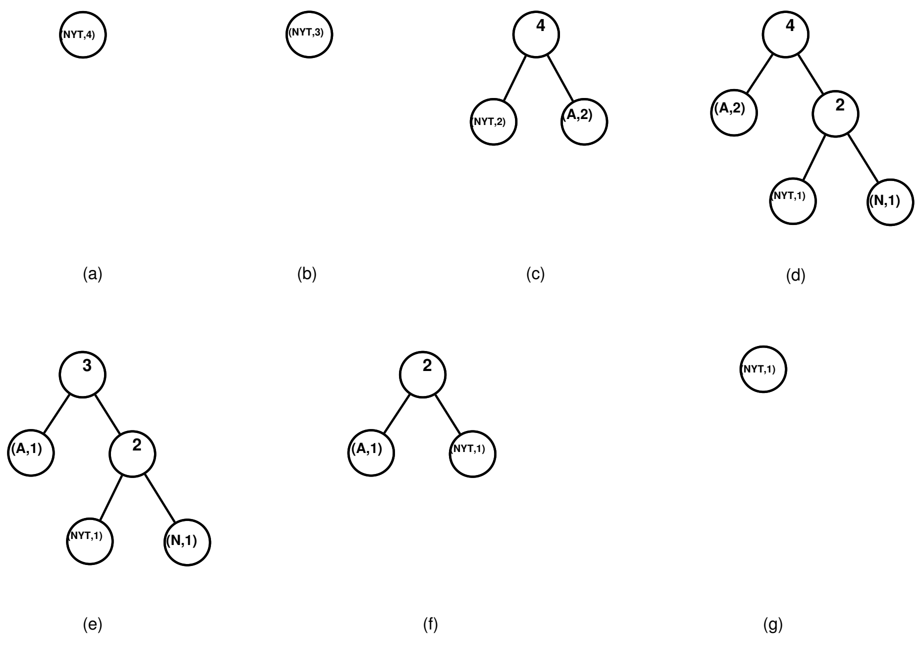


Figure 3. Illustration of HYBRID adaptive Huffman algorithm for T = `BANANAS`.

algorithm, the output consists of Huffman codewords, and if the special codeword for NYT has been emitted, it is followed by the ASCII representation of the newly encountered character. For FORWARD, there are only Huffman codewords in the output, since the alphabet is known in advance, so there is no need for NYT. Note that there is no encoding for the last letter `S`, because in this particular example, `S` appears only once, and after all other characters have been eliminated from the tree, so the corresponding codeword is the empty string. In the HYBRID technique, the output is a sequence of elements of four different kinds: Huffman codewords of elements of the alphabet, the Huffman codeword for NYT, newly encountered characters in ASCII, and frequencies in C_δ, all of which can be uniquely identified according to their position in the compressed text.

VITTER	0100 0010	0	0100 0001		10	0100 1110	10	110	0	100	0101 0011	
	`B`	NYT	`A`		NYT	`N`	`A`	`N`	`A`	NYT	`S`	
FORWARD	101		0			11	0	11	0			
	`B`		`A`			`N`	`A`	`N`	`A`		`S`	
HYBRID	0100 0010	1	0100 0001	0101	0	0100 1110	0100	0	11	0	0101 0011	1
	`B`	$C_\delta(1)$	`A`	$C_\delta(3)$	NYT	`N`	$C_\delta(2)$	`A`	`N`	`A`	`S`	$C_\delta(1)$

Figure 4. Comparative chart for the output of the three adaptive encoding procedures.

4 Analysis of the proposed algorithm

The contribution of the forward looking dynamic Huffman coding algorithm proposed in [9] is that it provably always achieves compression savings that are at least as good as those of the static Huffman algorithm, which is often claimed to yield "optimal" performance, and practically also improves upon the dynamic compression efficiency. The contribution of the current paper is yet a new twist on the implementation, with both theoretical and practical improvements over the forward algorithm.

Theorem: The expected performance of the HYBRID algorithm is at least as good as that of the FORWARD algorithm.

Proof: Note that the FORWARD and HYBRID algorithms only differ in their behavior at the beginning of the processing of a file, up to the moment where the entire alphabet to be used is already known. From that point on, the two algorithms are identical and generate exactly the same codewords. Define as i_0 the index of the character in the input file from which on the two models coincide, that is, i_0 is the index of the first occurrence of the last character of the alphabet that is encountered in a left to right scan of the input. Denote by f_j and h_j the lengths of the codewords generated for the j-th input character by the FORWARD and HYBRID algorithms, respectively. We have that

$$h_j = f_j \qquad \text{for} \quad j \geq i_0. \tag{1}$$

Define the sequence $k_1 = 1, k_2, \ldots, k_{|\Sigma|} = i_0$, as the indices of the first occurrences of all the characters of the alphabet Σ, in the order of their appearances. At any point $k < i_0$, the set of characters seen so far, denoted by Σ_k, is a proper subset of Σ. When processing the input characters with indices between k_r and k_{r+1}, for $1 \leq r < |\Sigma|$, the HYBRID algorithm bases its encoding on an alphabet of only r characters, just as VITTER's variant would do. The FORWARD algorithm, on the other hand, uses already the entire alphabet, and therefore works with another set of *probabilities* for the given subset Σ_{k_r}, in spite of using the same *frequencies*.

Denote by H^{k_r} the Huffman code based on the frequencies of the characters in Σ_{k_r} and by $H_\sigma^{k_r}$ the codeword length for a character $\sigma \in \Sigma_{k_r}$ in this Huffman code. At point k_r (and up to the characters indexed $k_{r+1} - 1$), the HYBRID algorithm uses the Huffman code H^{k_r}, but the FORWARD algorithm uses other probabilities, and therefore the corresponding codeword lengths $F_\sigma^{k_r}$ cannot be better, because of the optimality of Huffman's procedure, that is

$$\sum_{\sigma \in \Sigma_{k_r}} p_\sigma H_\sigma^{k_r} \leq \sum_{\sigma \in \Sigma_{k_r}} p_\sigma F_\sigma^{k_r}, \tag{2}$$

where p_σ is the probability of occurrence of the character σ at the given index k_r.

In other words, looking at the character indexed k_r, we encode it for HYBRID using a Huffman code which has been built for $P = \{p_\sigma \mid \sigma \in \Sigma_{k_r}\}$, which is optimal at that specific point. But for FORWARD, we base ourselves on other probabilities, so the resulting Huffman code is *not* necessarily optimal for P. Therefore, when averaging with the same set of probabilities, we get (2), that is, the former set of lengths yields an average which is not larger than that obtained by the latter set.

Summing over all r for the indices up to i_0 and adding eq. (1) for the larger indices, we conclude that the *expected* length of a codeword for HYBRID is $\leq$ than for FORWARD. ∎

We remark that this does not mean that necessarily $h_j \leq f_j$ for all $j < i_0$, and that the claim is only for their expected values. Indeed we found at least one example for which $h_j = f_j + 1$ for some j.

Another remark concerns the expected savings by using HYBRID instead of FORWARD. These will be rather modest for many "typical" large files. The reason is that there is a possible advantage of the former only before the entire alphabet has been discovered in a scan from the beginning. But this will often happen fairly soon in typical natural language texts.

Indeed, the distribution of the characters in standard English texts is well known and can be found in many books, see, e.g., [6]. Though the details vary from one source to another, it is well known that the letters J and Q, for example, are rare and appear with probability about 0.002. But that means that the expected number of characters to scan until the first appearance of one of these characters is about $1/0.002 = 500$, if we assume for simplicity that the text is generated by independently appearing characters. Therefore it will only rarely occur that after a few thousand letters, there is still a part of the alphabet that has not been seen.

We therefore see the main contribution of this paper as a theoretical one, showing that one can improve, even if only slightly, on a method which already seems better than one considered generally as optimal. Useful applications might be restricted to special files with sharply varying statistics, e.g., a file consisting first only of digits, then of characters, and similar settings.

5 Experimental Results

In order to compare the compression savings of the herein suggested HYBRID method relative to the three other algorithms, the dynamic VITTER and FORWARD Huffman encodings, as well as the STATIC Huffman variant, we have considered several datasets of different sizes and nature, and using different alphabets. *ebib* is the Bible (King James version) in English, in which the text has been stripped of all punctuation signs except blank; *ftxt* is the French version of the European Union's JOC corpus, a collection of pairs of questions and answers on various topics used in the ARCADE evaluation project [21]; *eng* is the concatenation of English text files, downloaded from the Pizza & Chili Corpus, selected from etext02 to etext05 collections of the Gutenberg Project, from which the headers related to the project were deleted so as to leave just the real text; *exe* is the executable file produced by compiling the source code we used for the hybrid Huffman algorithm; and *nt* is a dataset constructed from *ebib* in order to obtain a file composed of two parts having disjoint sets of alphabets — digits, followed by characters. The first part of the file takes the ten thousand first characters of the *ebib* file and writes them as decimal digits; the second part consists of the remaining characters of *ebib* in their original form. Table 1 summarizes the information regarding the used datasets.

In many encoding algorithms, the description of the underlying model on which the method relies on, is given in the header of the encoded file, and varies for each method. For static arithmetic coding the exact frequencies of the characters are not needed, and approximate probabilities suffice. On the other hand, the forward looking coding requires the exact frequencies of the items, while the traditional adaptive arithmetic method does not need any frequencies, since they are incrementally learned by both encoder and decoder. As the size of the prelude describing the model does not usually grow as a function of the size of the underlying file, it can be treated as a constant

File	full size (bytes)	$\lvert\Sigma\rvert$
ebib	3,711,020	53
exe	48,640	256
ftxt	7,648,930	132
eng	52,428,800	176
nt	3,726,683	63

Table 1. Information about the used datasets

sized header. Alternatively, we decided to compare the core of the encoding, neglecting the technical issue of transferring the model itself (which can be probably squeezed to be much shorter by a fine tuning of its encoding). The figures presented in Table 2 are the net sizes of the encodings, that is, the sizes of the entire codewords of the file without the description of the model, given in bytes. As can be seen from the results, the compression efficiencies of all methods are very close, which justifies the omission of the headers, as the description of the model is not negligible in the size of the difference.

File	Size of Encoded file (bytes)			
	Static	Adaptive	Forward	Hybrid
ebib	1,940,573	1,941,321	1,940,527	1,940,268
exe	31,296	31,851	31,132	28,930
ftxt	4,443,525	4,444,660	4,443,419	4,442,447
eng	29,914,197	29,915,562	29,914,021	29,912,644
nt	1,969,884	1,970,694	1,969,830	1,945,310

Table 2. Compression performance

Table 2 presents our experimental results on each file of our dataset in which the figures are given in bytes. The first column is the file's name. The second to fifth columns correspond to the size of the encoded files of static Huffman (Static), adaptive Huffman of Vitter (Adaptive), forward (Forward), and our proposed method (Hybrid), respectively. As theoretically expected, Hybrid consistently slightly improves Forward which itself improves both the static and dynamic versions, except for the file, *nt*, based on two different alphabets, for which the improvement is more significant.

References

1. A. Amir and G. Benson: *Efficient two-dimensional compressed matching*, in Proc. IEEE Data Compression Conference DCC–92, 1992, pp. 279–288.
2. G. Baruch, S. T. Klein, and D. Shapira: *A space efficient direct access data structure*. Journal of Discrete Algorithms, 43 2017, pp. 26–37.
3. G. Baruch, S. T. Klein, and D. Shapira: *Applying compression to hierarchical clustering*, in Similarity Search and Applications - 11th International Conference, SISAP 2018, Lima, Peru, October 7-9, 2018, Proceedings, 2018, pp. 151–162.
4. J. Cleary and I. Witten: *Data compression using adaptive coding and partial string matching*. IEEE Transactions on Communications, 32(4) 1984, pp. 396–402.
5. P. Elias: *Universal codeword sets and representations of the integers*. IEEE Trans. Information Theory, 21(2) 1975, pp. 194–203.

6. H. Heaps: *Information Retrieval, Computational and Theoretical Aspects*, Academic Press, 1978.
7. D. A. Huffman: *A method for the construction of minimum-redundancy codes.* Proceedings of the IRE, 40(9) 1952, pp. 1098–1101.
8. G. Jacobson: *Space efficient static trees and graphs*, in Proceedings of FOCS, 1989, pp. 549–554.
9. S. T. Klein, S. Saadia, and D. Shapira: *Forward looking Huffman coding*, in The 14th Computer Science Symposium in Russia, CSR, Novosibirsk, Russia, July 1-5, 2019.
10. S. T. Klein and D. Shapira: *A new compression method for compressed matching*, in Data Compression Conference, DCC 2000, Snowbird, Utah, USA, 2000, pp. 400–409.
11. S. T. Klein and D. Shapira: *Compressed pattern matching in* JPEG *images.* Int. J. Found. Comput. Sci., 17(6) 2006, pp. 1297–1306.
12. S. T. Klein and D. Shapira: *Compressed matching in dictionaries.* Algorithms, 4(1) 2011, pp. 61–74.
13. S. T. Klein and D. Shapira: *On improving Tunstall codes.* Inf. Process. Manage., 47(5) 2011, pp. 777–785.
14. S. T. Klein and D. Shapira: *Compressed matching for feature vectors.* Theor. Comput. Sci., 638 2016, pp. 52–62.
15. S. T. Klein and D. Shapira: *Random access to Fibonacci encoded files.* Discrete Applied Mathematics, 212 2016, pp. 115–128.
16. S. T. Klein and D. Shapira: *Integrated encryption in dynamic arithmetic compression*, in Language and Automata Theory and Applications - 11th International Conference, LATA 2017, Umeå, Sweden, March 6-9, 2017, Proceedings, 2017, pp. 143–154.
17. S. T. Klein and D. Shapira: *Context sensitive rewriting codes for flash memory.* Comput. J., 62(1) 2019, pp. 20–29.
18. G. Navarro: *Compact Data Structures: A Practical Approach*, Cambridge University Press, Cambridge, UK, 2016.
19. D. Shapira and A. H. Daptardar: *Adapting the knuth-morris-pratt algorithm for pattern matching in huffman encoded texts.* Inf. Process. Manage., 42(2) 2006, pp. 429–439.
20. J. A. Storer and T. G. Szymanski: *Data compression via textural substitution.* J. ACM, 29(4) 1982, pp. 928–951.
21. J. Véronis and P. Langlais: *Evaluation of parallel text alignment systems: The* ARCADE *project*, in Parallel Text Processing, J. Véronis, ed., Kluwer Academic Publishers, Dordrecht, Chapter 19, 2000, pp. 369–388.
22. J. S. Vitter: *Design and analysis of dynamic Huffman codes.* J. ACM, 34(4) 1987, pp. 825–845.
23. J. S. Vitter: *Algorithm 673: Dynamic Huffman coding.* ACM Trans. Math. Softw., 15(2) 1989, pp. 158–167.

Selective Dynamic Compression

Shmuel T. Klein[1], Elina Opalinsky[2], and Dana Shapira[2]

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

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

Abstract. Dynamic compression methods continuously update the model of the underlying text file to be compressed according to the already processed part of the file, assuming that such a model accurately predicts the distribution in the remaining part. Since this premise is not necessarily true, we suggest to update the model only selectively. We give empirical evidence that this hardly affects the compression efficiency, while it obviously may save processing time and allow the use of the compression scheme for cryptographic applications.

1 Introduction

Compression systems often comprise three major components: the model, the encoding process, and the corresponding inverse decoding process. The model is basically the definition of the set of symbols, called the alphabet, and their distribution. The alphabet defined in the model is not limited to characters only, and can also include words and phrases, and we shall use the term *alphabet* in this border sense. Generally, as the model becomes more and more accurate and adapted to the underlying text, the compression efficiency achieved by the encoding phase becomes better. However, the space overhead imposed by extending the model to be more precise, might diminish the savings achieved by the enhanced encoding, implying a crucial trade-off. Thus, determining the model has a crucial impact on the compression efficiency.

Text compression systems can be partitioned into static and dynamic compression techniques. Static variants determine the model in a preprocessing stage, and the model remains the same throughout the compression stage. The model can be constructed based on known distributions as well as on statistics gathered through a double pass over the file, the latter being suitable to off-line compressions. Dynamic variants, called also *adaptive compression*, save the additional pass, and usually consist of three main steps for each processed symbol:

1. reading the following symbol;
2. encoding the symbol according to the current model;
3. updating the model by incrementing the frequency of the currently read symbol.

The intuition of the dynamic variants is that as more information is collected about the characters in the text seen so far, the more accurate the probabilities become, and, thus, better approximation is achieved to the probabilities of the characters that are still to be seen. Indeed, all adaptive compression methods are based on the assumption that the distribution of the characters in the text starting from the current position onwards will be similar to the distribution in the part of the text preceding this current position. However, this is a statistic observation, which is not necessarily true, especially not for non-homogeneous texts. In these kind of texts it may happen that basing the prediction of the character probabilities on a random subset of the recently read characters may yield a compression performance that is not inferior, and sometimes even better, than using the entire history. Such an example is given in Section 3.2.

Evidence for basing the distribution of a certain alphabet in a given file on a subset of its characters, rather than on the entire file, can be found in the reported numbers for letter frequency analysis, which are often used in cryptography and linguistics. For example, Norvig [3] generated tables of counts for letters, words and letter sequences, on *Google books*, a collection of 23GB of the books that have been scanned by Google. The probabilities are given with an accuracy of at most 5 digits after the decimal point, corresponding to frequencies within a text of size 100,000, less than a fraction of 10^{-5} of the actual text. Even if the entire text has been used to generate the probabilities, the lower precision suffices to discriminate between them and therefore using a higher precision would have been a overkill.

Another example for using a restricted history, rather than the entire available one, can be found in the LZSS [4] compressor, which represents a given file T as a sequence of substring copies and single characters. The copies are described in the form of ordered pairs $(\mathit{off}, \mathit{len})$, meaning that the substring starting at the current position can be copied from *off* characters before the current position in the decompressed file, and the length of this substring is *len*. The variable *off* is often bounded by the size of a predefined *sliding window*, which limits the history in practice. GZIP uses a default window size of $32K$, and thus the *off* values can be written in 15 bits. The use of a limited history implemented as a sliding window is motivated by the saving incurred by the required bits to represent offsets within the restricted window. In this research we show that a limited history may have advantages beyond explicit storage savings.

The core motivation for basing the statistics of subsequent encoding only on a part of the previously seen symbols, rather than on the entire history, is time savings due to the processing of a smaller set. There is, however, a concern whether the restricted history may hurt the compression efficiency. A *Compression-Crypto System* based on arithmetic coding is introduced in [2], where the model gets updated according to a secret key shared only by encoder and decoder. This is yet another motivation for the encoding that is based on selectively updating the model, called *selective encoding* for short, however, here we extend the method to general adaptive compressors as well as for other purposes, other than encryption needs.

We focus on three different adaptive algorithms and examine the effect on time and compression performance in case the model is updated selectively. Our empirical results suggest that the loss in compression efficiency incurred by turning to a selective updating procedure as suggested, is hardly noticeable. Moreover, the encoding and decoding processing times can only be improved using the selective method, as expected, by saving the operations in case the model does not get updated, giving noticeable practical time savings.

Traditional dynamic Huffman codes turn the encoded file into an extremely vulnerable one in case of even a single bit error [1]. As a solution to this problem, blockwise dynamic Huffman variants are suggested, where the Huffman tree is periodically, rather than constantly, updated. Experiments show that the new scheme is more robust against single errors introduced in the encoded file. Here we suggest a "semi-blockwise" variant in which not all occurrences get updated.

The paper is constructed as follows. Section 2 presents the general method for selective encoding. Section 3 adapts the general selective algorithms to the three basic adaptive codings: dynamic Huffman, LZW and arithmetic coding. Section 4 presents empirical results and concludes.

2 General Method

Traditional dynamic algorithms update the frequencies adaptively after every character, according to the assumption that better compression can be achieved when all previous characters are taken into account. This seems to justify the slow processing time of some of the adaptive methods, such as *dynamic Huffman* coding. The selective method calls for omitting a subset of these updates in a predefined setting that is synchronized between the encoder and decoder. The selection may be done periodically, randomly, or by supplying the specific cases in which the model should get updated. We present here only the random version, with probability $\frac{1}{2}$ for the occurrences of 0 and 1 bits, which can be easily updated to the other selection variants.

Algorithm 1: SELECTIVE-ENCODE

```
SELECTIVE-ENCODE($T = x_1 \cdots x_n$ )
1 initialize the model
2 initialize a random bit generator
3 for $i \leftarrow 1$ to $n$ do
4     encode $x_i$ according to the current model
5     $bit \leftarrow random()$
6     if $bit = 1$ then
7         Update the model
```

The general random selective encoding and corresponding decoding algorithms are presented in Algorithm 1 and Algorithm 2, respectively. The selective encoding algorithm is applied on a given text $T = x_1 \cdots x_n$ where each x_i is a symbol of the alphabet Σ.

Algorithm 2: SELECTIVE-DECODE

```
SELECTIVE-DECODE($\mathcal{E}(\mathcal{T})$)
1 initialize the model
2 initialize a random bit generator identical to the one in SELECTIVE-ENCODE
3 for $i \leftarrow 1$ to $n$ do
4     decode $x_i$ according to the current model
5     $bit \leftarrow random()$
6     if $bit = 1$ then
7         Update the model
```

The coding method is chosen in advance and known to both the encoder and decoder. The model gets updated in case the bit returned by the random number generator is set to 1. We refer to the chosen encoding and corresponding decoding applications as a black box, and deal with specific encodings in the following section. The selective decoding algorithm is applied on a given compressed text denoted by

$\mathcal{E}(\mathcal{T})$, where $\mathcal{E}$ is the encoding function. The model and random bit generator are initialized identically in both procedures.

As mentioned above, instead of using a randomized algorithm, basing the selection on a random bit generator, one may decide in advance of setting a constant k, so that the model gets updated exactly every k symbols. The randomized version can be approximated by the constant updates with $k = 2$.

3 Specific Variants

The general selective approach may require adaption when applied to a specific method. Here we examine our proposed selective algorithms on three main adaptive compression techniques: arithmetic coding [7], dynamic Huffman coding [5] and LZW [6]. The following describes the adaptation suggested for each variant.

3.1 Arithmetic Coding

One of the most effective compression schemes, in theory as well as in practice, is arithmetic coding, for which the compression efficiency approaches the underlying text's entropy. The arithmetic compressor is initialized with the interval $[low, high) = [0, 1)$, which is narrowed for each processed character of the input file, according to the character's probability.

More formally, given a text $T = x_1 \cdots x_n$ over an alphabet Σ of size σ, the current interval $[low, high)$ is partitioned into σ subintervals, each corresponding to one of the symbols $\sigma_i \in \Sigma$, where the size of the subinterval assigned to σ_i is proportional to its probability p_i. After iterating on all symbols of T, the compressed text is represented by a single number within the final interval that corresponds to T.

For example, if $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$, initialized with uniform probabilities, one may start with (fictitious) frequencies 1 for each of the three characters. The partition is $\{[0, \frac{1}{3}), [\frac{1}{3}, \frac{2}{3}), [\frac{2}{3}, 1)\}$, and the intervals may be assigned lexicographically. If the text to be compressed is BCB, then the initial interval $[0, 1)$ is narrowed to $I_1 = [\frac{1}{3}, \frac{2}{3})$ after having read the first B, and the probabilities are updated to $P = \{\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\}$; the interval then narrows to $I_2 = [\frac{7}{12}, \frac{2}{3})$ after the processing of C and the probabilities are updated to $P = \{\frac{1}{5}, \frac{2}{5}, \frac{2}{5}\}$; and finally, after reading the second B, the interval narrows to $I_3 = [\frac{3}{5}, \frac{19}{30})$. Any real number within I_3 can be chosen to represent the compressed file, e.g., 0.625 whose binary representation 0.101 is the shortest.

However, when the model gets updated selectively, for example for every second character ($k = 2$ in our notation), the initial interval $[0, 1)$ gets narrowed in the first step, as in the traditional arithmetic coding, to $S_1 = [\frac{1}{3}, \frac{2}{3})$ after reading the first character B, and P becomes $P = \{\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\}$; processing the character C does not cause an update of the model, yet the interval narrows to $S_2 = [\frac{7}{12}, \frac{2}{3})$; but then, processing the following B narrows the interval to $S_3 = [\frac{29}{48}, \frac{31}{48})$ and updates the probabilities to $P = \{\frac{1}{5}, \frac{2}{5}, \frac{2}{5}\}$. The same binary number 0.101 can represent this interval. The compressed file will thus consist of a three bits in both cases.

Let $P = (p_1, \ldots, p_\sigma)$ be the probability distribution of all the characters of Σ in the entire text, and let P' be the probability distribution of the characters within the subtext that has been selected by the model, consisting by those corresponding to the 1-bits chosen by the random number generator. Because of the random selection and if the subset is large enough, the distribution remains almost the same, and therefore

$H(P) \simeq H(P')$, where $H(P) = -\Sigma_{i=1}^{\sigma} p_i \log p_i$ is the entropy of the distribution P. Since we know that the size of the encoded text is $n \cdot H(P)$ in the first case and $n \cdot H(P')$ in the second, we conclude that there is practically no loss in compression efficiency.

In the cryptographic application of [2], the bit sequence generated by the random bit generator is kept as a secret key K shared only by encoder and decoder. Using different keys yields completely different output files, and there seems to be no easy way to decipher the message without guessing K, yet the sizes of the compressed files were practically unchanged for different keys, as long as their 1-bit density was kept at $\frac{1}{2}$.

3.2 Dynamic Huffman Coding

We assume, for simplicity, that all dynamic Huffman variants initialize the tree with all the characters of the alphabet. We suggest two approaches for applying the proposed selective method in case of dynamic Huffman coding. Both algorithms still update the model periodically or via a random bit generator. The difference between them relates to the way the model gets updated.

The first approach, analogous to the one used for arithmetic coding, updates the dynamic Huffman tree by advancing the frequency of the current character only, ignoring the previous characters that have been skipped, and using Vitter's Algorithm in order to fix the Huffman tree in case the sibling property is violated. The first variant is denoted by Huf-subset.

The second approach, denoted by Huf-full, performs the updates according to the changes in the frequencies of *all* the characters seen since the last update. Practically, the updates in this case are spaced out and only done at the end of a *block* of several characters, therefore, immediately after the processing of a set bit in Algorithm 1, the updated Huffman tree reflects all symbols processed so far. As Vitter's Algorithm is based on updating the Huffman tree when only a single character frequency has been incremented by 1, we generated a static Huffman tree from scratch to implement the second suggestion.

Obviously, a bad selection may in the worst case jeopardize any compression savings, for example, when processing fixed length lines of 80 characters and choosing $k = 80$. The subset of Σ would then consist of the newline character only! On the other hand, the selective Huffman algorithm can produce a compressed file that is not only marginally smaller than the file constructed by traditional dynamic Huffman, but may even reach only about $\frac{3}{4}$ of its size, as shown in the following example.

Let $T = \mathtt{B}\{\mathtt{CCBB}\}^t$ for some positive integer t. For both variants, the Huffman tree is initialized with $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ as shown in Figure 1(a). Consider first the traditional dynamic algorithm as proposed in [5]. The first B is encoded as two bits, 10, and B is exchanged with A, as shown in Figure 1(b). When processing C of the first quadruple CCBB, only the second C causes a change in the structure of the Huffman tree, but this happens *after* the two Cs have already been encoded by 11, using 2 bits each. The Huffman tree after reading the prefix BCC of T, is depicted in Figure 1(c). When the following two Bs of the first quadruple CCBB are processed, again the positions of the B and C nodes are swapped only *after* the frequency of B exceeds that of C, so each of the Bs is also encoded by two bits (11). This alternating structure between two different Huffman trees proceeds until the end of the file, thus every character of T uses 2 bits, for a total of $2|T| = 8t + 2$ bits.

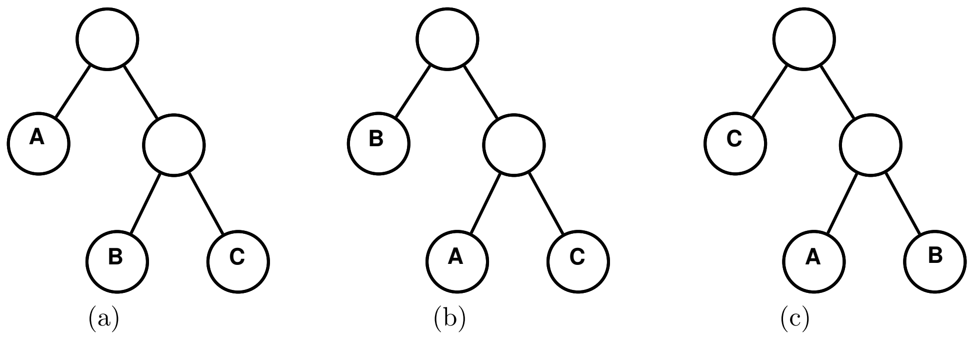


Figure 1. Example for which selective Huffman coding produces a file $\frac{3}{4}$ of the size of that constructed by standard Huffman.

However, when selective Huffman coding is used with $k = 2$, the first `B` is again encoded as two bits, 10, and `B` is exchanged with `A`, as shown in Figure 1(b). When processing the following `C`, it does not change the model, and is encoded with 2 bits as 11. The following `C` does cause an update of the frequencies, but does not change the structure of the Huffman tree. When the following two `B`s of the first quadruple `CCBB` are processed, they are both encoded as 0, using a single bit, and only the second `B` causes an update of the frequencies in the tree, but no change in its structure. This behavior proceeds until the end of the file, where the structure of the tree remains the same, and only the frequencies of the `B`s and `C`s get updated, encoding all `B`s as 0, and all `C`s by 11. Thus every quadruple `CCBB` uses 6 bits, for a total of $6t + 2$ bits, which is about $\frac{3}{4}$ of the regular encoding.

A random selection of the characters that are chosen for updating the model may be used to turn a dynamic Huffman encoder into a cryptosystem. Although in the long run, the distribution of the characters in the selected subset will be so close to their distribution in the entire text that there will be no noticeable loss in the compression, the small details of when exactly to increment which frequency will have a cumulative impact, producing ultimately completely different output sequences. Nevertheless, the sizes of these output files will be very close. As example, we compressed a French text file of 7.3MB with five independently generated random keys, and got compression ratios 0.57801, 0.57812, 0.57803, 0.578058 and 0.578061.

3.3 LZW

We turn to LZW [6], which is a member of the family of dictionary methods introduced by Ziv and Lempel [8,9], unlike the statistical compressions dealt with in the previous sections. The dictionary is initialized by the single characters of the alphabet, and then is updated dynamically by adding newly encountered substrings that have not been seen previously in the parsing of the underlying text. The text is thereby partitioned into a sequence of longest possible phrases that already occur in the dictionary, and the encoded file is a list of indices, each pointing to the entry of the corresponding phrase. LZW starts with a dictionary of size 512 which is filled to half its capacity by the alphabet of ASCII symbols, and each entry index is encoded by 9 bits. Once the dictionary has filled up after adjoining 256 new phrases, its size is doubled to 1024 entries, and all dictionary entries are encoded from this point on using 10-bit pointers.

In general, after processing $2^9, 2^{10}, \ldots, 2^i, \ldots$ more elements of the input file, the size of the dictionary is doubled to $2^{11}, 2^{12}, \ldots, 2^{i+2}, \ldots$ entries, up to a predetermined maximal size, which is 2^{16} in our implementation. We consider two variants:

1. restarting the dictionary from scratch each time the dictionary reaches its maximum size, denoted here by LZW-restart; or
2. considering the dictionary as static once it gets full, and not adjoining any more strings, denoted by LZW-static.

The dictionary built by LZW maintains a *prefix property*, that is, for each substring in the dictionary, all its prefixes also are included in the dictionary. In both cases of the selective variant, the dictionary gets filled up at a slower pace than for the traditional approach. As not all elements of the dictionary are necessarily used later, there are situations where the selective variant may improve the space savings of the traditional one, as can be seen in our experimental results presented in the following section.

Similarly to what could be done for Huffman coding, if the subset of newly encountered substrings is chosen randomly according to a secret key K, it will be hard to decode the file for whoever has to guess the 1-bits in K. Even though the compressed file consists of a sequence of integers and those are easy to decipher, it is their interpretation as representing certain substrings that is only known to the encoder and decoder. As before, different keys produce completely different dictionaries and thus different output strings, but there is hardly any change in the size of these files. For the same example input as in the previous section, the obtained compression ratios for 5 different keys were 0.3988, 0.3978, 0.3994, 0.3987 and 0.3981.

4 Experimental Results and conclusion

We examined the traditional and selective methods comparing the compression efficiency as well as the encoding and decoding processing times using the three basic adaptive compression algorithms mentioned above. We considered the 50MB file *English*, downloaded from the Pizza&Chili Corpus, which is the concatenation of English text files selected from etext02 to etext05 collections of the Gutenberg Project, from which the headers related to the project were deleted so as to leave just the real text. All experiments were conducted on a machine running 64 bit Windows 10 with an Intel Core i5-8250 @ 1.60GHz processor, 6144K L3 cache, and 8GB of main memory.

Our first experiment compares the compression efficiency of the traditional vs. selective variants of the various adaptive compression methods: Huf-subset, Huf-full, LZW-restart, LZW-static and arithmetic codings. The results are given in Figure 2 depicting the compression ratio (defined as the size of the compressed over the size of the original file) for $k = 1, \ldots, 8$ and $k = 16, 32$. The traditional dynamic algorithms correspond to $k = 1$, where the model gets updated after every character.

As can be seen, LZW-restart is less effective as the blocks become larger. However, when the dictionary stays constant once it gets filled up, the compression improves for larger blocks, until a point (not depicted in the graph), where the compression gain declines. According to our results, the compression efficiency becomes worse than for the traditional LZW for $k = 256$. This phenomenon can be explained by the fact that the static variant needs only a single learning period, while the other method undergoes, after each restart, a new learning phase during which the compression is less effective. In any case, the difference between selective and traditional techniques is less than 3%.

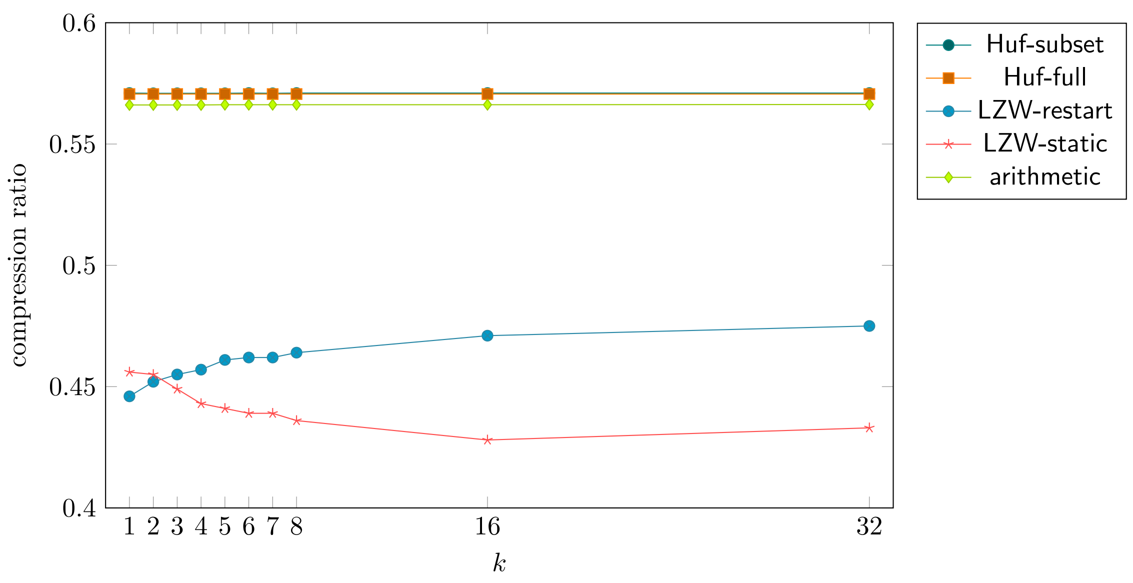


Figure 2. Compression Efficiency as a function of k.

The performance of the blockwise dynamic Huffman encoding, Huf-full, is practically identical to that of the partial update variant, Huf-subset, and their plots are overlapping in Figure 2. As for arithmetic coding, our results coincide with the theory mentioned above, and selective arithmetic compression remains very similar to the traditional one, as the probabilities are quite the same.

Figure 3 shows the processing times for both compression and decompression, for the standard and selective methods. For each of the test files, the encoding and decoding times were averaged over 10 runs. The displayed times are averages, given in seconds. Obviously, the compression and decompression times improve as the spacing becomes larger, because a smaller number of model updates is required.

	Compression ratio		Encoding time		Decoding time	
	Trad	Rand	Trad	Rand	Trad	Rand
Huf-subset	0.571	0.571	4447	2789	2388	1190
LZW-restart	0.452	0.446	10.667	9.781	5.321	4.348
LZW-static	0.456	0.454	8.796	8.771	3.096	3.097
arithmetic	0.5661	0.5661	41.61	27.11	46.65	32.39

Table 1. Results for Random as a function of k.

Table 1 presents the results for a random selection in which the choice to update the model is determined by a random bit generator (columns Rand). The columns headed Trad bring the results of the traditional, non-selective, approach. A random selection is similar to a fixed length selection model with $k = 2$, but has the advantage of almost surely avoiding worst case scenarios, like, e.g., a file for which even indexed characters belong to a different and disjoint alphabet than the odd indexed characters. As can be seen, the compression efficiencies are about the same, and that of the selective choice may sometimes be better. The processing times are consistently better, suggesting the usefulness of a selective encoder.

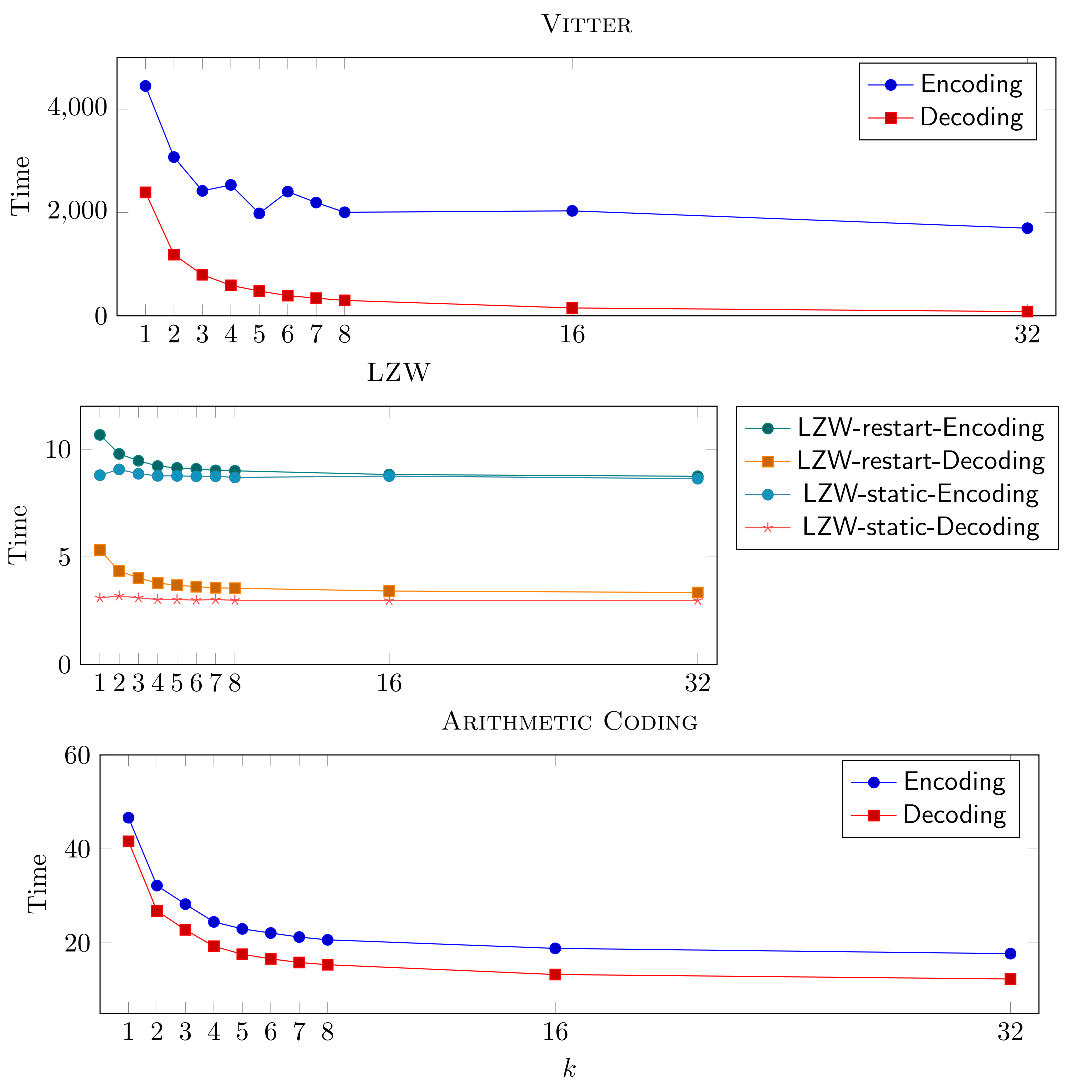


Figure 3. Processing Times as a function of k.

References

1. S. T. Klein, E. Opalinsky, and D. Shapira: *Synchronizing dynamic Huffman codes*, in Proceedings of the Prague Stringology Conference 2018, Czech Technical University in Prague, Czech Republic, 2018, pp. 27–37.
2. S. T. Klein and D. Shapira: *Integrated encryption in dynamic arithmetic compression*, in Language and Automata Theory and Applications - 11th International Conference, LATA 2017, Umeå, Sweden, March 6-9, 2017, Proceedings, 2017, pp. 143–154.
3. P. Norvig: *English letter frequency counts: Mayzner revisited or etaoin srhldcu*, 2018, `http://norvig.com/mayzner.html`.
4. J. A. Storer and T. G. Szymanski: *Data compression via textual substitution.* J. ACM, 29(4) 1982, pp. 928–951.
5. J. Vitter: *Design and analysis of dynamic Huffman codes.* J. ACM, 34(4) 1987, pp. 825–845.
6. T. Welch: *A technique for high-performance data compression.* IEEE Computer, 17(6) 1984, pp. 8–19.
7. I. Witten, R. Neal, and J. Cleary: *Arithmetic coding for data compression.* Commun. ACM, 30(6) 1987, pp. 520–540.
8. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Trans. Information Theory, 23(3) 1977, pp. 337–343.
9. J. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Trans. Information Theory, 24(5) 1978, pp. 530–536.

Optimal Time and Space Construction of Suffix Arrays and LCP Arrays for Integer Alphabets[⋆]

Keisuke Goto

Fujitsu Laboratories Ltd., Kawasaki, Japan, goto.keisuke@fujitsu.com

Abstract. Suffix arrays and LCP arrays are one of the most fundamental data structures widely used for various kinds of string processing. We consider two problems for a read-only string of length N over an integer alphabet $[1, \ldots, \sigma]$ for $1 \leq \sigma \leq N$, the string contains σ distinct characters, the construction of the suffix array, and a simultaneous construction of both the suffix array and LCP array. For the word RAM model, we propose algorithms to solve both of the problems in $O(N)$ time by using $O(1)$ extra words, which are optimal in time and space. Extra words means the required space except for the space of the input string and output suffix array and LCP array. Our contribution improves the previous most efficient algorithms, $O(N)$ time using $\sigma + O(1)$ extra words by [Nong, TOIS 2013] and $O(N \log N)$ time using $O(1)$ extra words by [Franceschini and Muthukrishnan, ICALP 2007], for constructing suffix arrays, and it improves the previous most efficient solution that runs in $O(N)$ time using $\sigma + O(1)$ extra words for constructing both suffix arrays and LCP arrays through a combination of [Nong, TOIS 2013] and [Manzini, SWAT 2004].

Keywords: suffix array, longest common prefix array, in-place algorithm

1 Introduction

Suffix arrays [21] are data structures that store all suffix positions of a given string sorted in lexicographical order according to their corresponding suffixes. They were proposed as a space efficient alternative to suffix trees, which are one of the most fundamental and powerful tools used for various kinds of string processing. LCP arrays [21] are auxiliary data structures that store the lengths of the longest common prefixes between adjacent suffixes stored in suffix arrays. Suffix arrays with LCP arrays are sometimes called *enhanced suffix arrays* [1], and they can simulate various operations of suffix trees. Suffix arrays or enhanced suffix arrays can be used for efficiently solving problems in various research areas, such as pattern matching [21,23], genome analysis [1,19], text compression [3,5,11], and data mining [9,12]. In these applications, one of the main computational bottlenecks is the time and space needed to construct suffix arrays and LCP arrays.

In this paper, we consider two problems that are for a given read-only string: constructing suffix arrays and constructing both suffix arrays and LCP arrays. For both problems, we propose optimal time and space algorithms. We assume that an input string of length N is read only, consists of an integer alphabet $[1, \ldots, \sigma]$ for $1 \leq \sigma \leq N$, and contains σ distinct characters [1]. We assume that the word RAM model with a word size of $w = \lceil \log N \rceil$ bits and that basic arithmetic and bit operations on constant number of words take constant time. We say that an algorithm runs *in-place*

[⋆] The full paper is available at https://arxiv.org/abs/1703.01009.

[1] As we will describe later, this is a slightly stronger assumption than commonly used in previous research.

and runs in optimal space if the algorithm requires constant extra words, which is the space except for the input string, output suffix array, and LCP array.

Suffix array construction. The first linear time (optimal time) algorithms for constructing suffix arrays for a string over an integer alphabet $[1, \ldots, \sigma]$ for $1 \leq \sigma \leq N$ were proposed at the same time by several authors [14, 16, 18], and they require at least N extra words. Nong [24] proposed a linear time and space efficient algorithm that requires $\sigma + O(1)$ extra words, but it still requires about N extra words in the worst case since σ can be N. An in-place algorithm that runs in $O(N \log N)$ time was proposed by Franceschini and Muthukrishnan [10] [2]. It has been an open problem whether there exists a suffix array construction algorithm that runs in linear time and in-place.

We propose an in-place linear time algorithm for a string over an integer alphabet $[1, \ldots, \sigma]$ that consists of σ distinct characters. Our algorithm is based on the induced sorting framework [10, 18, 24, 25], which splits all suffixes into L- and S-suffixes, sorts either of which first, and then sorts the other. The induced sorting framework uses two arrays: (1) a bit array of N bits to store each type of suffix, and (2) an integer array of σ words, for each character t, to store a pointer to the next insertion position of a suffix starting with t in the suffix array, so this framework requires $\sigma + N/\log N + O(1)$ extra words in a naive way. Our algorithm runs in almost the same way as the previous ones [10,18,24,25], but it stores these two arrays in the space of the output suffix array. Therefore, our algorithm runs in linear time and in-place. As a minor contribution, we also propose a simple space saving technique for the induced sorting framework. The framework has to store the beginning and ending positions of sub-arrays in recursive steps, which requires $O(\log N)$ words in total in a naive way. Franceschini and Muthukrishnan [10] proposed a method for storing them in-place and obtained each value in $O(\log N)$ time. We propose a simpler one for storing them in-place and obtain each value in $O(1)$ time (see the full paper for details).

Our assumption is slightly stronger than those of previous research in that all characters of an alphabet must appear in the input string [3]. However, if an input string can be writable not read-only, our algorithm still runs in linear time and in-place also for the same problem setting to previous research which an input string is over an integer $[1, \ldots, \sigma]$ and some characters may not appear in the string. Because we can transform the string to a string over an integer alphabet $[1, \ldots, \sigma']$ that consists of $\sigma' \leq \sigma$ distinct characters by the counting sort [17] that uses the space of output suffix arrays before our algorithm runs.

Recent and independent works for suffix array construction. Recently and independently, some in-place suffix array construction algorithms were proposed.

Li et al. [20] proposed an in-place linear time algorithm for a read-only string over an integer alphabet $[1, \ldots, O(N)]$ whose assumption is more general, and the result is stronger than ours. Though Li et al.'s algorithm is also based on the induced sorting framework, the details are different from ours. Both their and our algorithm look simple according to the framework, but this simplicity comes from using complex data structures and algorithms as tools. Li et al.'s algorithm uses two complex tools, in-place stable merge sort algorithm [4] and succinct data structures supporting select queries in constant time [13] which is used for storing pointers in compressed space.

[2] They assume that an input string is over a general alphabet, i.e., only comparison of any two characters is allowed, which can be done in $O(1)$ time. Their time and space complexities are optimal for general alphabets but not for integer alphabets.

[3] The same problem setting also appeared in [2].

On the other hand, our algorithm uses only the former one and store pointers in a normal array. Using complex tools tend to increase the runtime in practice, e.g., according to [7], select query times on a succinct data structure are several time slower than normal array accesses. In this perspective, our algorithm is simpler than theirs, and our work may contribute to develop practically faster in-place linear time suffix array construction algorithms in future.

Prezza [26] studied a similar problem that, for a writable input string, sorts suffixes of a size-b subset instead of all suffixes and constructs a *sparse* suffix array and *sparse* LCP array. His algorithm is based on a longest common extension data structure that, for two given positions i and j, efficiently computes the length of the longest common prefix between two suffixes starting at i and j, and it runs in $O(N+b\log^2 N)$ expected time and in-place.

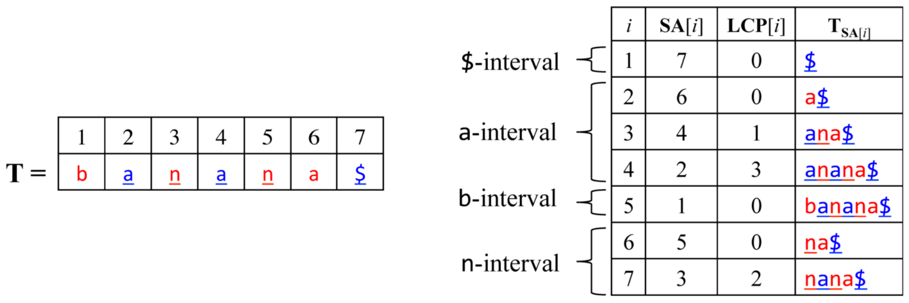
Suffix array and LCP array construction. Most previous research focused on a setting that computes LCP arrays from a given string and its suffix array. Kasai et al. [15] proposed the first linear time (optimal time) algorithm that computes the inverse suffix array and then uses it and computes the LCP array. Since it stores the inverse suffix array in extra space, it requires $N + O(1)$ extra words. Manzini [22] proposed a more space efficient linear time algorithm. The algorithm constructs the ψ array, which is similar to the inverse suffix array, in the output space of the LCP array and then rewrites it to the LCP array. The rewriting process runs in-place, but constructing the ψ array requires $\sigma+O(1)$ extra words, so the algorithm runs in linear time using $\sigma + O(1)$ words in total. Suffix arrays and LCP arrays can be computed in the same time and space by computing the suffix array with Nong's algorithm [24] and then by computing the LCP array with Manzini's algorithm [22]. The problem for constant alphabets with $\sigma \in O(1)$ has been studied in [6, 8], and the algorithms in [6, 8] are very competitive in practice for constant alphabets.

Our proposed linear time in-place algorithm constructs the suffix array and LCP array on the basis of a simple but non-trivial strategy. First, we construct the ψ array by using the space of the suffix array and LCP array and store it in the space of the LCP array. Then, we construct the suffix array in-place by using our linear time in-place algorithm, and we rewrite the ψ array to the LCP array as in Manzini's algorithm. Thus, we finally obtain both the suffix array and LCP array in linear time and in-place.

Organization. This paper is organized as follows. In Section 2, we introduce notations and definitions. In Section 3, we explain the induced sorting framework on which our algorithms are based. In Section 4 and Section 5, we propose optimal time and space algorithms for constructing suffix arrays and both suffix arrays and LCP arrays, respectively.

2 Preliminaries

Let Σ be an integer alphabet, the elements of which are $[1, \ldots, \sigma]$ for an integer $\sigma \geq 1$. An element of Σ^* is called a *string*. The length of a string $\mathbf{T}$ is denoted by $|\mathbf{T}|$. The empty string ϵ is a string of length 0. For a string $\mathbf{T} = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of $\mathbf{T}$, respectively. For a string $\mathbf{T}$ of length N, the i-th character of $\mathbf{T}$ is denoted by $\mathbf{T}[i]$ for $1 \leq i \leq N$, and the substring of $\mathbf{T}$ that begins at position i and that ends at position j is denoted by $\mathbf{T}[i \ldots j]$ for $1 \leq i \leq j \leq N$. For convenience, we assume that $\mathbf{T}[N] = \$$, where \$ is a special



	1	2	3	4	5	6	7
T =	b	a	n	a	n	a	$

	i	**SA**[i]	**LCP**[i]	$\mathbf{T}_{\mathbf{SA}[i]}$
$-interval	1	7	0	$
a-interval	2	6	0	a$
	3	4	1	ana$
	4	2	3	anana$
b-interval	5	1	0	banana$
n-interval	6	5	0	na$
	7	3	2	nana$

Figure 1. Suffix array and LCP array of $\mathbf{T} = \texttt{banana\$}$. $\mathbf{T}[i]$ is colored red if $\mathbf{T}_i$ is an L-suffix, and blue otherwise. Moreover, $\mathbf{T}[i]$ is underlined if $\mathbf{T}_i$ is an LML- or LMS-suffix.

character lexicographically smaller than any characters in the string $\mathbf{T}[1 \dots N-1]$. In this paper, we also assume that $\sigma \leq N$ and that $\mathbf{T}$ contains σ distinct characters.

A suffix starting at a position i and its first character are denoted by $\mathbf{T}_i$ and t_i, respectively, and the position i is also called a *pointer* to $\mathbf{T}_i$. If no confusion occurs, we sometimes use $\mathbf{T}_i$ as a pointer i. For $1 \leq i \leq N$, $\mathbf{T}_i$ is called a *small type suffix* (S-suffix) if $i = N$ or $\mathbf{T}_i$ is lexicographically smaller than $\mathbf{T}_{i+1}$, and it is called a *large type suffix* (L-suffix) otherwise. An S-suffix/L-suffix $\mathbf{T}_i$ is also called a *leftmost-S-suffix/leftmost-L-suffix* (LMS-suffix/LML-suffix) if $i > 1$ and $\mathbf{T}_{i-1}$ is an L-suffix/S-suffix. For $i < N$, $\mathbf{T}'_i$ denotes the substring $\mathbf{T}[i \dots j]$, where $j > i$ is the leftmost position such that $\mathbf{T}_j$ is an LMS-suffix. Each type of $\mathbf{T}'_i$ is equal to that of $\mathbf{T}_i$, and $\mathbf{T}'_i$ is referred to as an L-, S-, LML-, or LMS-substring according to its type. The important property is that, for $1 \leq i < N$, $\mathbf{T}_i$ is an L-suffix if $t_i > t_{i+1}$, or $t_i = t_{i+1}$ and $\mathbf{T}_{i+1}$ is an L-suffix, and $\mathbf{T}_i$ is an S-suffix otherwise. From this property, each type of suffix can be obtained in $O(N)$ time with a right-to-left scan on $\mathbf{T}$ by comparing the first characters of adjacent suffixes. $\mathit{suf}(\mathit{all})$ denotes the set of all suffixes of $\mathbf{T}$, and also $\mathit{suf}(L)$, $\mathit{suf}(S)$, $\mathit{suf}(\mathit{LML})$, and $\mathit{suf}(\mathit{LMS})$ denote the set of all L-, S-, LML-, and LMS-suffixes of $\mathbf{T}$, respectively. The size of a set M is denoted by N_M. Note that either $N_{\mathit{suf}(L)}$ or $N_{\mathit{suf}(S)}$ must be less than or equal to $N/2$ because all of the suffixes belong to either one. Moreover, $N_{\mathit{suf}(\mathit{LMS})}$ must be less than or equal to the smallest of $N_{\mathit{suf}(L)}$ and $N_{\mathit{suf}(S)}$. For a subset M of $\mathit{suf}(\mathit{all})$ and a suffix $\mathbf{T}_j$ of M, the rank of $\mathbf{T}_j$ is denoted by $\mathit{rank}_M(j)$, namely, $\mathbf{T}_j$ is the $\mathit{rank}_M(j)$-th smallest suffix of M. When the context is clear, we denote $\mathit{rank}_{\mathit{suf}(\mathit{all})}$ as rank.

For a subset M of $\mathit{suf}(\mathit{all})$, the suffix array $\mathbf{SA}_M$ of length N_M is an integer array that stores all pointers of M such that corresponding suffixes are lexicographically sorted. More precisely, for all suffixes $\mathbf{T}_i$ of M, $\mathbf{SA}_M[\mathit{rank}_M(i)] = i$. When the context is clear, we denote $\mathbf{SA}_{\mathit{suf}(\mathit{all})}$ as $\mathbf{SA}$. For each character t, the maximum interval in which the first characters of suffixes are equal to t in $\mathbf{SA}$ is called the *t-interval*. Because L- and S-suffixes are respectively larger and smaller than their succeeding suffix, for any character t, L-suffixes that start with t are always located before S-suffixes starting with t in $\mathbf{SA}$.

The LCP array is an auxiliary array of $\mathbf{SA}$ such that $\mathbf{LCP}[i]$ contains the length of the longest common prefix of $\mathbf{T}_{\mathbf{SA}[i]}$ and $\mathbf{T}_{\mathbf{SA}[i-1]}$ for $1 < i \leq N$, and $\mathbf{LCP}[1] = 0$. See Figure 1.

3 Induced Sorting Framework

Our algorithm is based on the induced sorting framework [10,18,24,25], so, in this section, we explain the algorithm in [25] as an example of the framework. This algorithm runs in $O(N)$ time using $\sigma + N/\log N + O(1)$ extra words [4].

The key point of the framework is to sort a subset of suffixes once and then sort another subset of suffixes from the sorted subset. From this perspective, we say that the sorting of latter suffixes is induced from the former suffixes. Let $\mathbf{T}^0$ be $\mathbf{T}$ and let $\mathbf{T}^{i+1}$ be a string such that $|\mathbf{T}^{i+1}|$ is the number of LMS-substrings of $\mathbf{T}^i$ and $\mathbf{T}^{i+1}[j] = k$, where the j-th LMS-substring from the left of $\mathbf{T}^i$ is the k-th lexicographically smallest LMS-substring of $\mathbf{T}^i$. There are two important properties; the first is that $|\mathbf{T}^{i+1}| \leq \lfloor |\mathbf{T}^i|/2 \rfloor$ since the number of LMS-substrings in $\mathbf{T}^i$ is at most $\lfloor |\mathbf{T}^i|/2 \rfloor$, and the second is that the rank of the j-th LMS-suffix from the left of $\mathbf{T}^i$ within all LMS-suffixes in $\mathbf{T}^i$ corresponds to the rank of the j-th suffix from the left of $\mathbf{T}^{i+1}$ within all suffixes in $\mathbf{T}^{i+1}$. The algorithm recursively computes the suffix array $\mathbf{SA}^i$ of the string $\mathbf{T}^i$ at each recursive step i, namely, the algorithm sorts suffixes the number of which is smaller in more inner recursive steps. Note that $\mathbf{T}^i$ has the same property of $\mathbf{T}$ such that $\mathbf{T}^i$ consists of an integer alphabet of $[1, \ldots \sigma']$ for $1 \leq \sigma' \leq |\mathbf{T}^i|$ and $\mathbf{T}^i$ contains σ' distinct characters.

Below is an overview of the algorithm for computing the $\mathbf{SA}^i$ of $\mathbf{T}^i$ at a recursive step i. Note that all suffixes and substrings that appear in the overview indicate those of $\mathbf{T}^i$.

1. Sort all LMS-substrings.
2. Sort all LMS-suffixes from sorted LMS-substrings.
3. Sort all suffixes from sorted LMS-suffixes.
   (a) Perform preprocessing for Step 3b.
   (b) Sort all L-suffixes from sorted LMS-suffixes.
   (c) Sort all S-suffixes from sorted L-suffixes.

The essence of the algorithm is the part in which all suffixes are sorted from the sorted LMS-suffixes in Step 3. Step 1 runs in almost the same way as Step 3. Step 2 creates $\mathbf{T}^{i+1}$ and computes its suffix array recursively. Therefore, we herein explain only Step 3.

We consider only the case of computing the $\mathbf{SA}^0 = \mathbf{SA}$ of $\mathbf{T}^0 = \mathbf{T}$ at the recursive step 0 since we can also compute the $\mathbf{SA}^i$ of $\mathbf{T}^i$ similarly at each recursive step i. The algorithm requires three arrays, $\mathbf{A}$, $\mathbf{LE}/\mathbf{RE}$ [5], and **type**. $\mathbf{A}$ is an integer array of length N to be $\mathbf{SA}$ at the end of the algorithm. At the beginning of the algorithm, we assume that each $\mathbf{A}[i]$ is initialized to *empty* in linear time, where empty is a special symbol that is used so that any element storing this symbol stores no meaningful value[6]. **type** is a binary array of length N, which indicates the type of $\mathbf{T}_j$ such that **type**$[j] = L$ if $\mathbf{T}_j$ is an L-suffix, and **type**$[j] = S$ otherwise. The **type** can

[4] The space for storing beginning and ending positions of sub-arrays in recursive steps is not accounted for.

[5] The notation was borrowed from $\mathbf{LF}/\mathbf{RF}$ used in [20], which is the abbreviation of leftmost/rightmost free. Although the definition is the same as the *bkt* array commonly used in previous research [6, 24, 25], the name $\mathbf{LF}/\mathbf{RF}$ is more specific. In our paper, we frequently use empty as the special symbol, so we prefer to use the notation $\mathbf{LE}/\mathbf{RE}$, which is the abbreviation for leftmost/rightmost empty.

[6] Practically, the special symbol is represented as an integer $N + 1$ indicating a position out of $\mathbf{A}$ so that we can distinguish the special symbol from pointers of $\mathbf{A}$.

be computed in $O(N)$ time with a right-to-left scan on $\mathbf{T}$ by comparing the first characters of the current suffix and its succeeding suffix. $\mathbf{LE}/\mathbf{RE}$ is an integer array of length σ such that $\mathbf{LE}[t]/\mathbf{RE}[t]$ indicates the next insertion position of a suffix starting with a character t in $\mathbf{A}$. $\mathbf{LE}[t]/\mathbf{RE}[t]$ is initially set to the head/tail of the t-interval of $\mathbf{SA}$, and it is managed in order to indicate the leftmost/rightmost empty position of the t-interval at any step of the algorithm. $\mathbf{LE}$ can be initialized in $O(N)$ time as follows. Let C_t be the number of suffixes starting with t. First, $\mathbf{LE}[t] = C_t$ is computed for all characters t by counting t_i with a single scan on $\mathbf{T}$ by using $\mathbf{LE}[t_i]$ as a counter, and last, $\mathbf{LE}[t] = 1 + \sum_{t'<t} C_{t'}$ is computed by accumulating $\mathbf{LE}[t] = C_t$ lexicographically. Similarly, $\mathbf{RE}$ can also be computed in $O(N)$ time.

We assume that $\mathbf{LE}$ is initialized at the beginning of Step 3b and that $\mathbf{RE}$ is also at the beginning of Step 3a and Step 3c. During the steps, types of suffixes are obtained by **type**.

Step 3a: As the result of Step 2, we have $\mathbf{SA}_{suf(LMS)} = \mathbf{A}[1 \dots N_{suf(LMS)}]$, and $\mathbf{A}[N_{suf(LMS)}+1 \dots N]$ is filled with empty. With a right-to-left scan on $\mathbf{SA}_{suf(LMS)}$, we move each $\mathbf{SA}_{suf(LMS)}[i] = \mathbf{T}_j$ into $\mathbf{A}[\mathbf{RE}[t_j]]$, which is the rightmost empty position of the t_j-interval, and decrease $\mathbf{RE}[t_j]$ by one to indicate the new rightmost empty position of the t_j-interval.

Step 3b: With a left-to-right scan on $\mathbf{A}$, we read all L- and LMS-suffixes $\mathbf{A}[i] = \mathbf{T}_j$ in lexicographic order, if $\mathbf{T}_{j-1}$ is an L-suffix, store $\mathbf{T}_{j-1}$ in $\mathbf{A}[\mathbf{LE}[t_{j-1}]]$, and increase $\mathbf{LE}[t_{j-1}]$ by one.

Step 3c: This step runs almost the same way as Step 3b. With a right-to-left scan on $\mathbf{A}$, we read all L- and S-suffixes $\mathbf{A}[i] = \mathbf{T}_j$ in reverse lexicographic order, if $\mathbf{T}_{j-1}$ is an S-suffix, store $\mathbf{T}_{j-1}$ in $\mathbf{A}[\mathbf{RE}[t_{j-1}]]$ and decrease $\mathbf{RE}[t_{j-1}]$ by one.

Steps 3a, 3b, and 3c run in $O(N)$ time because each step scans $\mathbf{A}$ only one time, and any of the operations take constant time per access. From Lemma 1, all induced L- and S-suffixes $\mathbf{T}_{j-1}$ are stored in $\mathbf{A}[rank(j-1)]$, so $\mathbf{A} = \mathbf{SA}$ is obtained at the end of Step 3. Roughly speaking, the correctness of Lemma 1 comes from the invariant that all suffixes stored in $\mathbf{A}$ are always sorted during the steps. When reading $\mathbf{A}[i] = \mathbf{T}_j$, the L-suffix $\mathbf{T}_{j-1}$ must be larger than any suffix $\mathbf{T}_{k-1}$ already stored in t_{j-1}-interval since $\mathbf{T}_k$ must appear at $\mathbf{A}[i']$ for $i' < i$, and it holds that $\mathbf{T}_k < \mathbf{T}_j$ from the invariant. Moreover, we do not miss any L-suffixes since we always store an induced L-suffix from a suffix stored in $\mathbf{A}[i]$ in a more rightward position $\mathbf{A}[i']$ for $i' > i$.

Lemma 1 ([25]). *When an L-suffix/S-suffix* $\mathbf{T}_{j-1}$ *is being induced in Step 3b/Step 3c,* $\mathbf{LE}[t_{j-1}]/\mathbf{RE}[t_{j-1}]$ *indicates* $rank(j-1)$.

Since $|\mathbf{T}^{i+1}| \leq \lfloor |\mathbf{T}^i|/2 \rfloor$ and the algorithm runs in $O(|\mathbf{T}^i|)$ time for all $\lceil \log N \rceil$ recursive steps i, the algorithm runs in $O(N)$ time in total. Moreover, the algorithm requires $\sigma + N/\log N + O(1)$ extra words, the first and second factors are for $\mathbf{LE}/\mathbf{RE}$ and **type**, respectively.

4 Optimal Time and Space Construction of Suffix Arrays

We propose a novel algorithm for constructing suffix arrays on the basis of the induced sorting framework. The space bottleneck of the previous algorithm [25] is the space of $\mathbf{LE}/\mathbf{RE}$ and **type**. Our algorithm embeds both arrays in the space of $\mathbf{A}$, and runs in $O(N)$ time and in-place, namely, in optimal time and space.

As seen in Section 3, the essence of the induced sorting framework is the part in which L-suffixes are sorted. Therefore, we focus on how to sort the L-suffixes from sorted LMS-suffixes in $O(N)$ time and in-place. We can also sort S-suffixes in the same way (see Appendix A.2) and also LMS-substrings. Thus, we have the following theorem.

Theorem 2. *Given a read-only string* $\mathbf{T}$ *of length* N*, which consists of integers* $[1, \ldots, \sigma]$ *for* $1 \leq \sigma \leq N$ *and contains* σ *distinct characters, there is an algorithm for computing the* $\mathbf{SA}$ *of* $\mathbf{T}$ *in* $O(N)$ *time and in-place.*

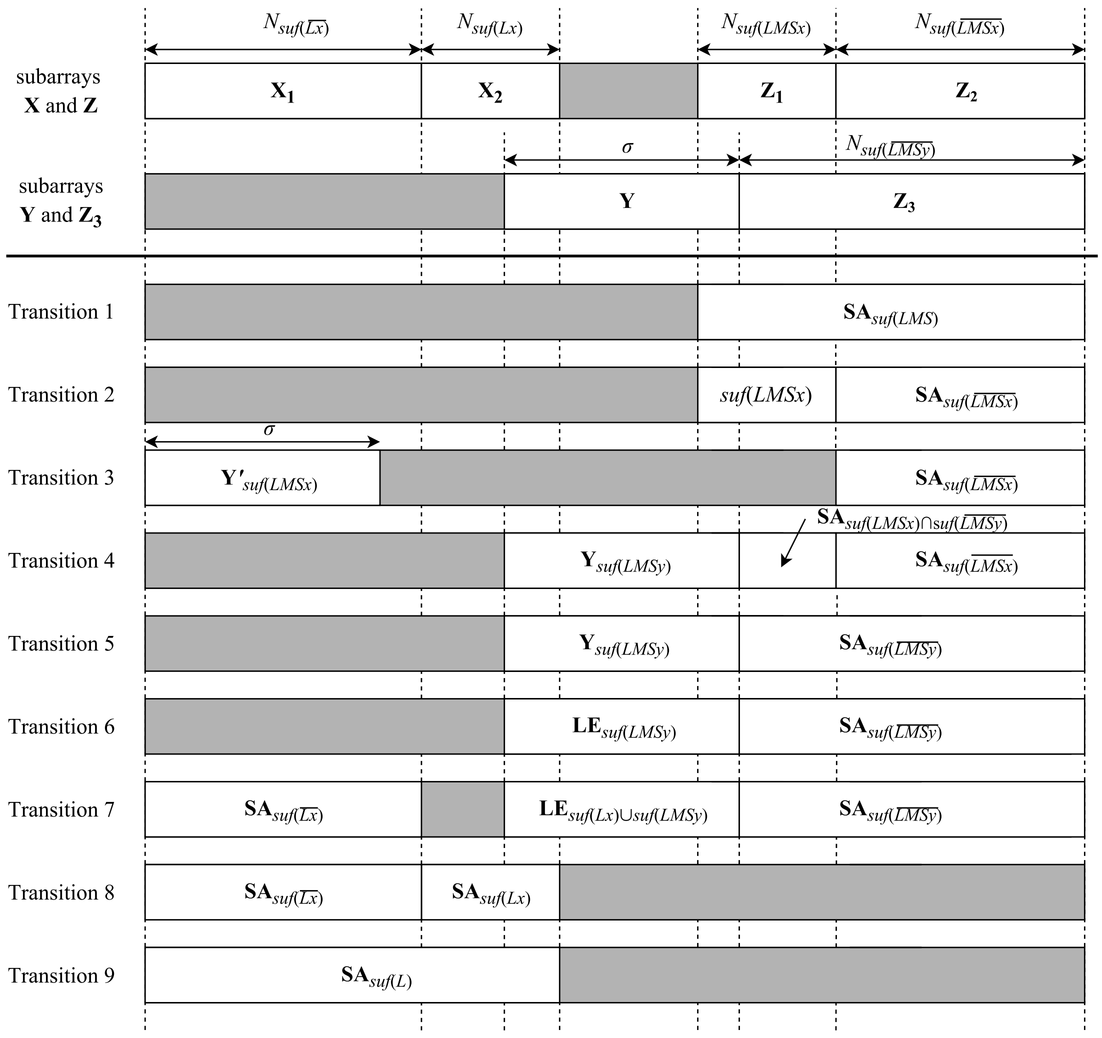
Our main idea for reducing the space is to store sorted L- and LMS-suffixes in three internal sub-arrays in $\mathbf{A}$. We refer to these arrays as $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$. The length of $\mathbf{Y}$ is σ, and for each character t, $\mathbf{Y}[t]$ stores either $\mathbf{LE}[t]$, the largest L-suffix starting with t, or the smallest LMS-suffix starting with t. $\mathbf{X}$ and $\mathbf{Z}$ store all L- and LMS-suffixes other than the ones stored in $\mathbf{Y}$, respectively.

We embed $\mathbf{LE}$ in $\mathbf{Y}$. Intuitively, this idea does not work because the total size of $\mathbf{X}$ and $\mathbf{LE}$ may exceed N, and if so, $\mathbf{X}$ overlaps with $\mathbf{LE}$ in $\mathbf{A}$, and elements of $\mathbf{LE}$ required in the future may be overwritten by induced L-suffixes. Moreover, we may not be able to even store $\mathbf{SA}_{suf(LMS)}$ and $\mathbf{LE}$ in $\mathbf{A}$ at the same time before sorting L-suffixes because their total size can also be greater than N. We avoid this problem by overwriting $\mathbf{LE}[t]$ only when it is no longer used in the future, namely, when all L-suffixes starting with t have been induced or there is no L-suffix starting with t. We detect such timing by causing a conflict between induced L-suffixes. Let CL_t be the number of L-suffixes starting with t. We try to store all L-suffixes in $\mathbf{X}$, whose space is limited that can store only $CL_t - 1$ L-suffixes starting with t for each character t. More precisely, for a character t, the beginning and ending position of t-interval overlaps with the ending position of the preceding t'-interval for $t' < t$ and the beginning position of the succeeding t''-interval for $t'' > t$, respectively. Therefore, conflict must occur between the largest (the last induced) L-suffix starting with a character t and the smallest (the first induced) L-suffix starting with $t'' > t$. We can detect the timing on the basis of conflicts, and we find that all L-suffixes starting with a smaller character t have been induced and that $\mathbf{LE}[t]$ is no longer needed in the future.

We do not need **type** anymore for detecting the type of suffixes being induced. We read L- and LMS-suffixes $\mathbf{T}_i$ stored either in $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$ in lexicographic order. If $\mathbf{T}_i$ is read from $\mathbf{X}$ or $\mathbf{Z}$, we know the type of $\mathbf{T}_i$, so the type of $\mathbf{T}_{i-1}$ is easily obtained. Otherwise, we do not know the type of $\mathbf{T}_i$ in $\mathbf{Y}$, so the type of $\mathbf{T}_{i-1}$ is non-trivial. An important observation is that, for a suffix $\mathbf{T}_i$ in $\mathbf{Y}$, the preceding character t_{i-1} must be different from t_i since $\mathbf{T}_i$ is the largest L-suffix starting with t_i or the smallest LMS-suffix starting with t_i. Therefore, the type of $\mathbf{T}_{i-1}$ can be determined without **type** by comparing the characters t_{i-1} and t_i.

We use **type** of σ bits rather than N bits for distinguishing the L- and LMS-suffixes and the elements of $\mathbf{LE}$, which are stored in $\mathbf{Y}$. Although **type** require σ bits if it is stored naively, we can embed it in the space of $\mathbf{Y}$. Details on the in-place implementation of **type** will be given in Section 4.2.

The sub-arrays $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$ and their more internal sub-arrays are located in $\mathbf{A}$ as shown in Figure 2. The figure also shows all of the steps of our algorithm. We partition the suffixes in certain subsets, which allows us to run each transition above in $O(N)$ time and in-place. Let $suf(LMSx)$ be the set of the LMS-suffixes that are the smallest among all LMS-suffixes starting with the same character, and let $suf(\overline{LMSx})$ be the set of all other LMS-suffixes. Let $suf(LMSy)$ be a subset of $suf(LMSx)$ that contains all $\mathbf{T}_i \in suf(LMSx)$ such that there is no L-suffix starting with t_i, and let $suf(\overline{LMSy})$ be the set of all other LMS-suffixes. We have $suf(LMSy) \subseteq$



subarrays X and Z
$N_{suf(\overline{Lx})}$
$N_{suf(Lx)}$
$N_{suf(LMSx)}$
$N_{suf(\overline{LMSx})}$
X1
X2
Z1
Z2
subarrays Y and Z3
σ
$N_{suf(\overline{LMSy})}$
Y
Z3
Transition 1
$SA_{suf(LMS)}$
Transition 2
suf(LMSx)
$SA_{suf(\overline{LMSx})}$
Transition 3
$Y'_{suf(LMSx)}$
$SA_{suf(\overline{LMSx})}$
$SA_{suf(LMSx)\cap suf(\overline{LMSy})}$
Transition 4
$Y_{suf(LMSy)}$
$SA_{suf(\overline{LMSx})}$
Transition 5
$Y_{suf(LMSy)}$
$SA_{suf(\overline{LMSy})}$
Transition 6
$LE_{suf(LMSy)}$
$SA_{suf(\overline{LMSy})}$
Transition 7
$SA_{suf(\overline{Lx})}$
$LE_{suf(Lx)\cup suf(LMSy)}$
$SA_{suf(\overline{LMSy})}$
Transition 8
$SA_{suf(\overline{Lx})}$
$SA_{suf(Lx)}$
Transition 9
$SA_{suf(L)}$

Figure 2. Inside transition of **A** while computing $\mathbf{SA}_{suf(L)}$ from $\mathbf{SA}_{suf(LMS)}$. Space colored with gray indicates empty space.

$suf(LMSx)$, $|suf(LMSx)| \le \sigma$, and $suf(\overline{LMSx}) \subseteq suf(\overline{LMSy})$. Let $suf(Lx)$ be the set of all L-suffixes that are the largest among all L-suffixes starting with the same character, and let $suf(\overline{Lx})$ be the set of all other L-suffixes. Intuitively, $suf(Lx)$ and $suf(LMSx)$ consist of L- and LMS-suffixes that are closest to the border of L- and S-suffixes in each t-interval in $\mathbf{SA}_{suf(all)}$, respectively. Moreover, $suf(Lx) \cup suf(LMSy)$ is made by selecting one suffix for each interval from the set $suf(Lx) \cup suf(LMSx)$, where we give an L-suffix priority over an LMS-suffix if both exists. Thus, we have $|suf(Lx) \cup suf(LMSy)| \le \sigma$.

We store various types of elements in $\mathbf{Y}$. To reduce ambiguity, $\mathbf{Y}_M$ denotes $\mathbf{Y}$ *overwritten* by a set of suffixes M whose first characters are distinct (we consider that the initial $\mathbf{Y}$ is filled with empty). More precisely, for a character t, $\mathbf{Y}_M[t] = \mathbf{T}_i$ if $\mathbf{T}_i$ starting with t exists in M, and $\mathbf{Y}_M[t] = \mathbf{Y}[t]$ otherwise. For example, $\mathbf{Y}_{suf(LMSy)}[t] = \mathbf{T}_i$ if $\mathbf{T}_i \in suf(LMSy)$ starting with t exists, and $\mathbf{Y}_{suf(LMSy)}[t]$ is empty otherwise. Moreover, $\mathbf{LE}_{suf(LMSy)}[t] = \mathbf{T}_i$ if $\mathbf{T}_i \in suf(LMSy)$ starting with t exists, and $\mathbf{LE}_{suf(LMSy)}[t] = \mathbf{LE}[t]$ otherwise.

4.1 Sort all L-suffixes

We compute $\mathbf{SA}_{suf(L)}$ in the head of $\mathbf{A}$ from $\mathbf{SA}_{suf(LMS)}$ stored in the head of $\mathbf{A}$, which is given by the result of sorting the LMS-suffixes. The internal transitions of $\mathbf{A}$ in the algorithm are shown in Figure 2, and each transition runs in $O(N)$ time and in-place. In Transitions 1-6, we compute $\mathbf{LE}$ and move LMS-suffixes in $\mathbf{Y}$ and $\mathbf{Z}$. Transition 7 induces all L-suffixes from LMS-suffixes stored in $\mathbf{Y}$ and $\mathbf{Z}$ and stores them in $\mathbf{X}$ and $\mathbf{Y}$. The concept of this transition is almost the same as Step 3b in Section 3. In Transitions 8-9, we merge the L-suffixes of $\mathbf{X}$ and $\mathbf{Y}$ and obtain $\mathbf{SA}_{suf(L)}$. The former part Transitions 1-5 is not so hard, so we omitted (we left the details in Appendix A.1), and we only describe the latter part Transitions 6-9 which is the most technical part of our algorithm. We assume that we have a bit array **type** of σ bits without extra space, and details on its in-place implementation are given in Section 4.2.

As the result of Transitions 1-5, we have $\mathbf{Y}_{suf(LMSy)}$ for which $\mathbf{Y}[t] = \mathbf{T}_i$ if $\mathbf{T}_i \in suf(LMSy)$ starting with t exists, and $\mathbf{Y}[t]$ is empty otherwise, and we also have **type** for which $\mathbf{type}[t] = 1$ if an L-suffix starting with t exists, and $\mathbf{type}[t] = 0$ otherwise.

Transition 6: We transform $\mathbf{Y}_{suf(LMSy)}$ into $\mathbf{LE}_{suf(LMSy)}$. With a right-to-left scan on $\mathbf{T}$, we compute CL_t for each character t for which $\mathbf{type}[t] = 1$, namely for which $CL_t > 0$, and store it in $\mathbf{Y}[t]$ by using $\mathbf{Y}[t]$ as a counter. Note that we never overwrite a suffix of $suf(LMSy)$ stored in $\mathbf{Y}[t]$ since $\mathbf{type}[t] = 0$ and there is no L-suffix starting with t. With a left-to-right scan on $\mathbf{Y}$, we compute the prefix sum $\mathbf{Y}[t] = \mathbf{LE}[t] = 1 + \sum_{t'<t} \max(0, CL_{t'} - 1)$ for each t for which $\mathbf{type}[t] = 1$. Finally, we have $\mathbf{LE}_{suf(LMSy)}$ in $\mathbf{Y}$ that $\mathbf{Y}[t] = \mathbf{LE}[t]$ if $\mathbf{type}[t] = 1$, and $\mathbf{Y}[t]$ is $\mathbf{T}_i \in suf(LMSy)$ or empty otherwise.

Transition 7: We compute $\mathbf{SA}_{suf(\overline{Lx})}$ in $\mathbf{X}_1$ and $\mathbf{LE}_{suf(Lx) \cup suf(LMSy)}$ in $\mathbf{Y}$. This transition consists of the following three parts whose concept is almost same as Step 3b in Section 3. Part 1 reads all L- and LMS-suffixes $\mathbf{T}_i$ lexicographically from $\mathbf{X}_1$, $\mathbf{Y}$, and $\mathbf{Z}_3$, Part 2 judges whether $\mathbf{T}_{i-1}$ is an L-suffix or not, and Part 3 stores $\mathbf{T}_{i-1}$ if it is an L-suffix. During the transition, we use **type** to determine whether $\mathbf{Y}[t]$ is a suffix (including both L- and LMS-suffixes) or an element of $\mathbf{LE}$. The invariant is that $\mathbf{type}[t] = 1$ if $\mathbf{Y}[t]$ is an element of $\mathbf{LE}$, and $\mathbf{type}[t] = 0$ otherwise, that is, $\mathbf{Y}[t]$ is empty or a suffix of $suf(Lx) \cup suf(LMSy)$. As the result of the previous transition, **type** has already satisfied the invariant.

We explain Part 1 last because it depends on Part 3.

Part 2: Judge whether $\mathbf{T}_{i-1}$ is an L-suffix or not. As already explained, the type of $\mathbf{T}_{i-1}$ can be obtained by comparing the first character t_{i-1} and its succeeding characters t_i.

Part 3: Store $\mathbf{T}_{i-1}$ if it is an L-suffix. We try to store an L-suffix $\mathbf{T}_{i-1}$ in $\mathbf{X}_1[\mathbf{LE}[t_{i-1}]]$, which is the next insertion position of a suffix starting with t_{i-1}. If $\mathbf{X}_1[\mathbf{LE}[t_{i-1}]]$ is empty, we simply store $\mathbf{T}_{i-1}$ there and increase $\mathbf{LE}[t_{i-1}]$ by one to update the next insertion position. Otherwise, $\mathbf{X}_1[\mathbf{LE}[t_{i-1}]]$ has already stored a suffix $\mathbf{T}_j$. As already explained, a conflict must occur between the largest (the last induced) L-suffix starting with a character t and the smallest (the first induced) L-suffix starting with $t' > t$, and all L-suffixes starting with the smaller character t have already induced. Therefore, we compare the first characters t_{i-1} and t_j, store the smaller one in $\mathbf{LE}[\min(t_{i-1}, t_j)]$, store the larger one in $\mathbf{X}[\mathbf{LE}[t_{i-1}]]$, and update $\mathbf{type}[\min(t_{i-1}, t_j)] = 0$. Moreover, we increase $\mathbf{LE}[t_{i-1}]$ by one if $t_{i-1} > t_j$.

Part 1: Read all L- and LMS-suffixes $\mathbf{T}_i$ lexicographically. Recall that the arrays $\mathbf{X}_1$, $\mathbf{Y}$, and $\mathbf{Z}_3$ store sorted suffixes. With a left-to-right scan on $\mathbf{X}_1$, $\mathbf{Y}$, and $\mathbf{Z}_3$, we scan $\mathbf{X}_1[i_X]$, $\mathbf{Y}[i_Y]$, and $\mathbf{Z}_3[i_Z]$ simultaneously in lexicographic order, where i_X, i_Y, and i_Z are the scanning positions of $\mathbf{X}_1$, $\mathbf{Y}$, and $\mathbf{Z}_3$, respectively. We recall the types of suffixes stored in $\mathbf{X}_1$, $\mathbf{Y}$, and $\mathbf{Z}_3$:

– $\mathbf{X}_1[i_X]$ is either empty or an L-suffix.
– $\mathbf{Y}[i_Y]$ is either empty, a suffix of $suf(Lx) \cup suf(LMSy)$, or $\mathbf{LE}[i_Y]$.
– $\mathbf{Z}_3[i_Z]$ is a suffix of $\in suf(\overline{LMSy})$.

Let t_X, t_Y, and t_Z be the first characters of suffixes stored in $\mathbf{X}_1[i_X]$, $\mathbf{Y}[i_Y]$, and $\mathbf{Z}_3[i_Z]$, respectively, where t_Y equals i_Y. Here, we assume that t_X, t_Y, and t_Z are $\sigma + 1 \notin \Sigma$ if each index i_X, i_Y, or i_Z indicates a position out of the corresponding array, that is, such t_X, t_Y, and t_Z must not be chosen. We also assume that t_X is $\sigma + 1$ if $\mathbf{X}_1[i_X]$ is empty.

We choose the smallest character t_i of t_X, t_Y, and t_Z. In case we need to break a tie, we give t_X priority over t_Y, and t_Y priority over t_Z. Note that $\mathbf{T}_i$ is the smallest suffix of the three candidates because, for suffixes $\mathbf{T}_{j_1}$, $\mathbf{T}_{j_2}$, and $\mathbf{T}_{j_3}$ of $suf(\overline{Lx})$, $suf(Lx) \cup suf(LMSy)$, and $suf(\overline{LMSy})$, respectively, we have $\mathbf{T}_{j_1} < \mathbf{T}_{j_2} < \mathbf{T}_{j_3}$ if they all start with the same character, and $\mathbf{T}_i$ is chosen in this order of priority. Next, we increase the scanning position by one. Thus, we can read all L- and LMS-suffixes lexicographically.

One concern is that we may choose $t_Y = i_Y$ for which either $\mathbf{Y}[i_Y]$ is empty or $\mathbf{Y}[i_Y] = \mathbf{LE}[i_Y]$. The former case implies that none of the L- or LMS-suffixes start with t_Y, so we increase i_Y by one and choose the smallest character from the three candidates again. In the latter case, let $\mathbf{T}_i$ be the largest L-suffix starting with t, it implies that $\mathbf{type}[i_Y]$ must be 1, a conflict with $\mathbf{T}_i$ has not occured yet, and $\mathbf{T}_i$ has already been read and is still stored in $\mathbf{X}_1$. So, in this case also, we increase i_Y by one and choose the smallest character from the three candidates again. $\mathbf{T}_i$ stored in $\mathbf{X}_1$ will conflict with another L-suffix and be stored in $\mathbf{LE}[t_Y]$ in the future.

Transition 8: We compute $\mathbf{SA}_{suf(Lx)}$ in $\mathbf{X}_2$ and initialize the space except for $\mathbf{X}$ in $\mathbf{A}$ as empty. All L-suffixes of $suf(Lx)$ are stored in $\mathbf{Y}$, and we have $\mathbf{type}$ for which $\mathbf{type}[t] = 1$ if $\mathbf{Y}[t]$ stores an L-suffix of $suf(Lx)$, and $\mathbf{type}[t] = 0$ otherwise. With a left-to-right scan on $\mathbf{Y}$, we move all L-suffixes $\mathbf{T}_i$ for which $\mathbf{type}[t_i] = 1$ in back of $\mathbf{X}_1$ while preserving the order, and we obtain $\mathbf{SA}_{suf(Lx)}$ in $\mathbf{X}_2$. Finally, we fill $\mathbf{A}[N_{suf(L)} \dots N]$ with empty.

Transition 9: We compute $\mathbf{SA}_{suf(L)}$. By applying the in-place stable merge algorithm in Theorem 3 to $\mathbf{SA}_{suf(Lx)}$ and $\mathbf{SA}_{suf(\overline{Lx})}$ considering the first characters as keys, we compute $\mathbf{SA}_{suf(L)}$ in $O(N)$ time and in-place.

Theorem 3 ([4]). *For two sorted integer arrays* $\mathbf{A}_1 = \mathbf{A}[1 \dots N_1]$ *and* $\mathbf{A}_2 = \mathbf{A}[N_1 + 1 \dots N_1 + N_2]$ *that are stored in an array* $\mathbf{A}[1 \dots N_1 + N_2]$*, there is an in-place linear time (*$O(N_1 + N_2)$ *time) algorithm that can stably merge* $\mathbf{A}_1$ *and* $\mathbf{A}_2$ *in* $\mathbf{A}$*.*

Remark: For ease of explanation, we use the complex stable merge algorithm in Transition 5 and 9 for sorting L-suffixes. We can optimize the algorithm so that the algorithm does not use the merge algorithm for sorting L-suffixes and use only two times for sorting S-suffixes. Since $\mathbf{SA}_{suf(\overline{LMSy})}$ is read only sequentially, we can simulate the sequential scan of $\mathbf{SA}_{suf(\overline{LMSy})}$ by scanning $\mathbf{SA}_{suf(LMSx) \cap suf(\overline{LMSy})}$ and $\mathbf{SA}_{suf(\overline{LMSx})}$ sequentially. Moreover, Transition 9 is equal to Transition 2 in sorting S-suffixes (see Appendix A.2), so we can skip Transition 9 and avoid to use the stable merge algorithm.

4.2 In-place implementation of type

We store suffixes and elements of $\mathbf{LE}$ in $\mathbf{Y}$ in a compact representation so that whose most significant bits (MSBs) are vacant, and embed **type** in the MSBs of $\mathbf{Y}$. Since each original value can be obtained from the simple compact representation in $O(1)$ time, it does not cause any problems for all transitions shown in Figure 2.

$\mathbf{LE}$ is a non-decreasing sequence, so we remember the leftmost m-interval that includes the position $2^{\lceil \log N \rceil - 1}$ in $\mathbf{X}_1$ whose MSB is 1, and also remember the MSB of $\mathbf{LE}[m]$ as *msb*. In Transition 7, *msb* is initially 0 but finally becomes 1. All elements of $\mathbf{LE}[t]$ are stored in $\mathbf{Y}$ in the compact representation by clearing the MSBs to 0. The original value of each $\mathbf{LE}[t]$ can be obtained in $O(1)$ time as follows;

- Set the MSB to 0 for $t < m$.
- Set the MSB to *msb* for $t = m$.
- Set the MSB to 1 for $t > m$.

A suffix $\mathbf{T}_i$ is stored as $\lfloor i/2 \rfloor$ so that the MSB is vacant. We use two important properties to obtain original values that, for a suffix $\mathbf{T}_i$ stored in $\mathbf{Y}[t]$, (1) the first character of $\mathbf{T}_i$ must be t, and (2) the preceding character t_{i-1} does not equal t (since $\mathbf{T}_i$ is the largest L-suffix starting with t or the smallest LMS-suffix starting with t). We can obtain an original suffix $\mathbf{T}_i$ from its compact representation $\mathbf{Y}[t] = j$. The candidate of i is $2j$ or $2j+1$. If $t_{2j} \neq t_{2j+1}$, we choose one that equals t with Property 1. Otherwise, we choose $2j$ with Property 2.

Thus, we can store all elements of $\mathbf{LE}$ and suffixes in $\mathbf{Y}$ in a compact representation whose MSBs are vacant and store **type** in-place in the MSBs of $\mathbf{Y}$.

5 Optimal Time and Space Construction of Suffix Arrays and LCP Arrays

We propose an algorithm for computing the suffix array and LCP array of a given read-only string $\mathbf{T}$ in $O(N)$ time and in-place. We revisit Manzini's algorithm [22], which constructs an LCP array $\mathbf{LCP}$ from a given string $\mathbf{T}$ and a suffix array $\mathbf{SA}$ in $O(N)$ time by using $\sigma + O(1)$ extra words. The algorithm uses a ψ array $\mathbf{\Psi}$ which is also called the *rank next array*, where $\mathbf{\Psi}[\mathit{rank}(i)] = \mathit{rank}(i+1)$ for $1 \leq i < N$. The algorithm consists of two parts. The first part computes $\mathbf{\Psi}$ in $O(N)$ time by using $\sigma + O(1)$ extra words. The second part converts $\mathbf{\Psi}$ into $\mathbf{LCP}$ in $O(N)$ time and in-place. Therefore, $\mathbf{LCP}$ can be computed in $O(N)$ time and in-place if $\mathbf{\Psi}$ can be computed in $O(N)$ time and in-place.

Let $\mathbf{A}$ and $\mathbf{B}$ be integer arrays of length N to be $\mathbf{SA}$ and $\mathbf{LCP}$ at the end of the algorithm, respectively. Our algorithm computes $\mathbf{B} = \mathbf{\Psi}$ with both arrays $\mathbf{A}$ and $\mathbf{B}$ and $O(1)$ extra words. After that, it computes $\mathbf{A} = \mathbf{SA}$ in-place as described in Section 4 and converts $\mathbf{B} = \mathbf{\Psi}$ into $\mathbf{B} = \mathbf{LCP}$ in-place as in Manzini's way. For computing $\mathbf{\Psi}$, we use the inverse suffix array $\mathbf{ISA}$ such that $\mathbf{ISA}[\mathbf{SA}[i]] = i$, which is also called the *rank array* since $\mathbf{ISA}[i] = \mathit{rank}(i)$. The algorithm runs in the following steps.

1. Compute $\mathbf{B} = \mathbf{SA}$.
2. Compute $\mathbf{A} = \mathbf{ISA}$ from $\mathbf{SA}$.
3. Compute $\mathbf{B} = \mathbf{\Psi}$, that is, drop $\mathbf{SA}$. With a left-to-right scan on $\mathbf{ISA}$, set $\mathbf{B}[\mathbf{ISA}[i]] = \mathbf{ISA}[i+1]$ if $\mathbf{ISA}[i] < N$.

4. Compute $\mathbf{A} = \mathbf{SA}$ as described in Section 4.
5. Convert $\mathbf{B} = \mathbf{\Psi}$ into $\mathbf{B} = \mathbf{LCP}$ as in Manzini's way.

All of the steps run in $O(N)$ time and in-place. Thus, we have the following theorem.

Theorem 4. *Given a read-only string* $\mathbf{T}$ *of length* N*, which consists of integers* $[1, \ldots, \sigma]$ *for* $1 \leq \sigma \leq N$ *and contains* σ *distinct characters, there is an algorithm for computing both* $\mathbf{SA}$ *and* $\mathbf{LCP}$ *of* $\mathbf{T}$ *in* $O(N)$ *time and in-place.*

Acknowledgement

We wish to thank Takashi Kato, Shunsuke Inenaga, Hideo Bannai, Dominik Köppl, and anonymous reviewers for their many valuable suggestions on improving the quality of this paper, and especially, we also wish to thank an anonymous reviewer for giving us a simpler algorithm for computing LCP arrays as described in Section 5.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch: *The enhanced suffix array and its applications to genome analysis*, in Algorithms in Bioinformatics, Second International Workshop, WABI 2002, Rome, Italy, September 17-21, 2002, Proceedings, 2002, pp. 449–463.
2. J. Barbay, F. Claude, T. Gagie, G. Navarro, and Y. Nekrich: *Efficient fully-compressed sequence representations.* Algorithmica, 69(1) 2014, pp. 232–268.
3. M. Burrows and D. J. Wheeler: *A block-sorting lossless data compression algorithm*, tech. rep., 1994.
4. J. Chen: *Optimizing stable in-place merging.* Theor. Comput. Sci., 302(1-3) 2003, pp. 191–210.
5. M. Crochemore and L. Ilie: *Computing longest previous factor in linear time and applications.* Inf. Process. Lett., 106(2) 2008, pp. 75–80.
6. F. A. da Louza, S. Gog, and G. P. Telles: *Optimal suffix sorting and LCP array construction for constant alphabets.* Inf. Process. Lett., 118 2017, pp. 30–34.
7. O. Delpratt, N. Rahman, and R. Raman: *Engineering the LOUDS succinct tree representation*, in Experimental Algorithms, 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006, Proceedings, 2006, pp. 134–145.
8. J. Fischer: *Inducing the lcp-array*, in Algorithms and Data Structures - 12th International Symposium, WADS 2011, New York, NY, USA, August 15-17, 2011. Proceedings, 2011, pp. 374–385.
9. J. Fischer, V. Heun, and S. Kramer: *Fast frequent string mining using suffix arrays*, in Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005), 27-30 November 2005, Houston, Texas, USA, 2005, pp. 609–612.
10. G. Franceschini and S. Muthukrishnan: *In-place suffix sorting*, in Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings, 2007, pp. 533–545.
11. K. Goto and H. Bannai: *Space efficient linear time Lempel-Ziv factorization for small alphabets*, in Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26-28 March, 2014, 2014, pp. 163–172.
12. K. Goto, H. Bannai, S. Inenaga, and M. Takeda: *Fast q-gram mining on SLP compressed strings.* J. Discrete Algorithms, 18 2013, pp. 89–99.
13. G. Jacobson: *Space-efficient static trees and graphs*, in 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989, 1989, pp. 549–554.
14. J. Kärkkäinen, P. Sanders, and S. Burkhardt: *Linear work suffix array construction.* J. ACM, 53(6) 2006, pp. 918–936.

15. 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, 2001, pp. 181–192.
16. D. K. Kim, J. S. Sim, H. Park, and K. Park: *Constructing suffix arrays in linear time.* J. Discrete Algorithms, 3(2-4) 2005, pp. 126–142.
17. D. E. Knuth: *The art of computer programming, Volume III, 2nd Edition, Sorting and Searching*, Addison-Wesley, 1998.
18. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays.* J. Discrete Algorithms, 3(2-4) 2005, pp. 143–156.
19. F. Li and G. D. Stormo: *Selection of optimal DNA oligos for gene expression arrays.* Bioinformatics, 17(11) 2001, pp. 1067–1076.
20. Z. Li, J. Li, and H. Huo: *Optimal in-place suffix sorting*, in String Processing and Information Retrieval - 25th International Symposium, SPIRE 2018, Lima, Peru, October 9-11, 2018, Proceedings, 2018, pp. 268–284.
21. U. Manber and E. W. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM J. Comput., 22(5) 1993, pp. 935–948.
22. G. Manzini: *Two space saving tricks for linear time LCP array computation*, in Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings, 2004, pp. 372–383.
23. G. Navarro: *A guided tour to approximate string matching.* ACM Comput. Surv., 33(1) 2001, pp. 31–88.
24. G. Nong: *Practical linear-time O(1)-workspace suffix sorting for constant alphabets.* ACM Trans. Inf. Syst., 31(3) 2013, p. 15.
25. G. Nong, S. Zhang, and W. H. Chan: *Two efficient algorithms for linear time suffix array construction.* IEEE Trans. Computers, 60(10) 2011, pp. 1471–1484.
26. N. Prezza: *In-place sparse suffix sorting*, in Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018, 2018, pp. 1496–1508.

A Appendix

A.1 Sort all L-suffixes: Former Transitions

We describe Transitions 1-5, which are omitted in Section 4.1.

Transition 1: We shift $\mathbf{SA}_{suf(LMS)}$ stored in the head of $\mathbf{A}$ into $\mathbf{Z}$.

Transition 2: We store $suf(LMSx)$ in $\mathbf{Z}_1$ and compute $\mathbf{SA}_{suf(\overline{LMSx})}$ in $\mathbf{Z}_2$. Note that the suffixes in $\mathbf{Z}_2$ are sorted but may not be in $\mathbf{Z}_1$. Let j be the insertion position in $\mathbf{Z}_2$ for $\mathbf{SA}_{suf(\overline{LMSx})}$, which is initially set to $N_{suf(LMS)}$, namely, the end of $\mathbf{Z}$. With a right-to-left scan on $\mathbf{Z} = \mathbf{SA}_{suf(LMS)}$, we swap $\mathbf{SA}_{suf(LMS)}[i] = \mathbf{T}_k$ with $\mathbf{Z}[j]$ and decrease j by one if $\mathbf{T}_k \in suf(\overline{LMSx})$ and do nothing otherwise. Whether or not $\mathbf{T}_k$ belongs to $suf(\overline{LMSx})$ can be judged in $O(1)$ time by comparing the first characters because the first characters $t_{\mathbf{SA}_{suf(LMS)}[i]}$ and $t_{\mathbf{SA}_{suf(LMS)}[i-1]}$ are the same if and only if $\mathbf{T}_k \in suf(\overline{LMSx})$. Since we shift the suffixes of $suf(\overline{LMSx})$ to the end of $\mathbf{Z}$ while preserving the order of the shifted suffixes, we obtain $suf(LMSx)$ in $\mathbf{Z}_1$ (which may not be sorted) and $\mathbf{SA}_{suf(\overline{LMSx})}$ in $\mathbf{Z}_2$.

Unfortunately, we cannot compute $\mathbf{Y}_{suf(LMSy)}$ at this point directly because we currently do not know the size of $N_{suf(\overline{LMSy})}$ determining the starting position of $\mathbf{Y}$ within $\mathbf{A}$. We obtain this information in Transition 4. To start with, we consider a temporary array $\mathbf{Y}' = \mathbf{A}[1 \dots \sigma]$ and compute $\mathbf{Y}'_{suf(LMSx)}$.

Transition 3: We compute $\mathbf{Y}'_{suf(LMSx)}$. With a right-to-left scan on $\mathbf{Z}_1$, we try to move $\mathbf{Z}_1[i] = \mathbf{T}_{j_1}$ into $\mathbf{Y}'[t_{j_1}]$. However, $\mathbf{Y}'[t_{j_1}]$ may contain an LMS-suffix $\mathbf{T}_{j_2}$ because $\mathbf{Y}'$ may overlap with $\mathbf{Z}_1$. We simply move $\mathbf{T}_{j_1}$ into $\mathbf{Y}'[t_{j_1}]$ if $\mathbf{Y}'[t_{j_1}]$ is empty and do nothing if $\mathbf{Y}'[t_{j_1}]$ is $\mathbf{T}_{j_1}$ because then $\mathbf{Z}_1[i]$ and $\mathbf{Y}'[t_{j_1}]$ are the same entry in $\mathbf{A}$. Otherwise, $\mathbf{Y}'[t_{j_1}]$ contains a suffix $\mathbf{T}_{j_2}$ such that $\mathbf{T}_{j_2} \neq \mathbf{T}_{j_1}$. In this case, we move $\mathbf{T}_{j_1}$ into $\mathbf{Y}'[t_{j_1}]$ and then try to move $\mathbf{T}_{j_2}$ into $\mathbf{Y}'[t_{j_2}]$. We repeat this procedure until we move $\mathbf{T}_{j_k}$ to $\mathbf{Y}'[t_{j_k}]$, which is empty, or encounter $\mathbf{Y}'[t_{j_k}] = \mathbf{T}_{j_k}$. Because $N_{suf(LMSx)} \leq \sigma$ and the first characters of $suf(LMSx)$ are all different, the number of insertions is $O(\sigma)$, and this transition can be done in $O(\sigma)$ time. Finally, we have $\mathbf{Y}'_{suf(LMSx)}$ such that $\mathbf{Y}'[t_i] = \mathbf{T}_i$ if $\mathbf{T}_i \in suf(LMSx)$ or $\mathbf{Y}'[t_i]$ is empty otherwise.

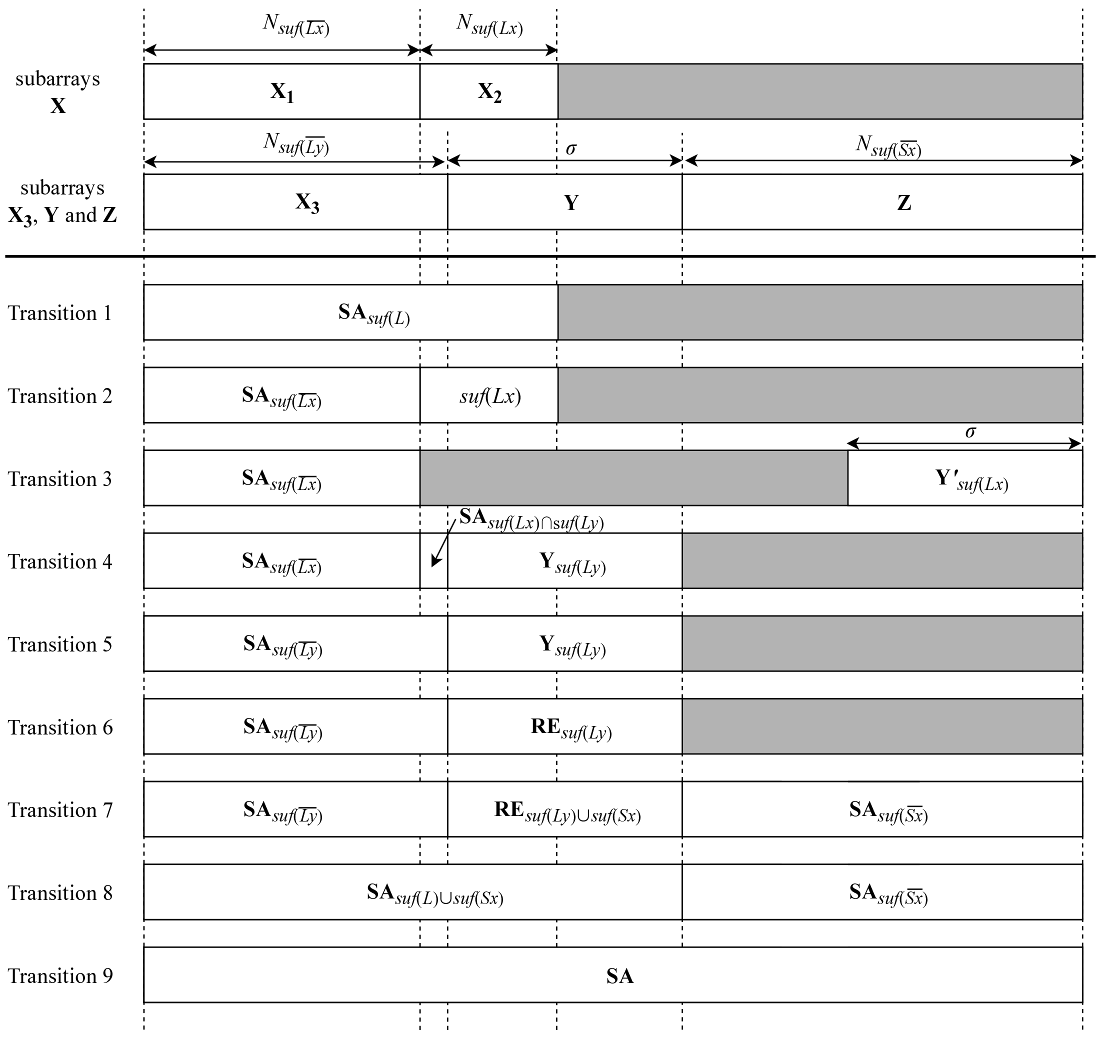
Transition 4: We compute $\mathbf{Y}_{suf(LMSy)}$ and $\mathbf{SA}_{suf(LMSx)\cap suf(\overline{LMSy})}$. The set $suf(LMSx) \cap suf(\overline{LMSy})$ consists of each $suf(LMSx)$ suffix for which there is an L-suffix starting with the same character, and $suf(LMSy)$ is other $suf(LMSx)$. We compute $\mathbf{type}[t] = 1$ if there is an L-suffix starting with t, and $\mathbf{type}[t] = 0$ otherwise. We initialize $\mathbf{type}$ with 0. With a right-to-left scan on $\mathbf{T}$, we set $\mathbf{type}[t] = 1$ for an L-suffix starting with t. Now we know that a suffix stored in $\mathbf{Y}'_{suf(LMSx)}[t]$ with $\mathbf{type}[t] = 1$ belongs to $suf(\overline{LMSy}) \cap suf(LMSx)$. With a right-to-left scan on $\mathbf{Y}'_{suf(LMSx)}$, we move such suffixes in front of $\mathbf{Z}_2 = \mathbf{SA}_{suf(\overline{LMSx})}$ while preserving the order; then, we have $\mathbf{Z}_1 = \mathbf{SA}_{suf(LMSx)\cap suf(\overline{LMSy})}$. We just move $\mathbf{Y}'$ in front of $\mathbf{Z}_1$, and we have $\mathbf{Y}_{suf(LMSy)}$.

Transition 5: We compute $\mathbf{SA}_{suf(\overline{LMSy})}$. Because a suffix $\mathbf{T}_i$ of $suf(\overline{LMSy}) \cap suf(LMSx)$ is smaller than all suffixes of $suf(\overline{LMSx})$ starting with the same character t_i, $\mathbf{SA}_{suf(\overline{LMSy})}$ can be obtained by stably merging the last two arrays $\mathbf{SA}_{suf(\overline{LMSy})\cap suf(LMSx)}$ and $\mathbf{SA}_{suf(\overline{LMSx})}$ with respect to the first characters as keys. The merged array contains *all* $suf(\overline{LMSy})$ suffixes since $(suf(\overline{LMSy}) \cap suf(LMSx)) \cup suf(\overline{LMSx}) = suf(\overline{LMSy})$. By applying Theorem 3 to $\mathbf{SA}_{suf(\overline{LMSy})\cap suf(LMSx)}$ and $\mathbf{SA}_{suf(\overline{LMSx})}$, we compute $\mathbf{SA}_{suf(\overline{LMSy})}$ in $O(N)$ time and in-place.

A.2 Sort all S-suffixes

We can sort all S-suffixes in almost the same way as sorting L-suffixes but compute $\mathbf{SA}$ instead of $\mathbf{SA}_{suf(S)}$. The same can be said by switching the roles of $\mathbf{LE}$, L- and LMS-suffixes with $\mathbf{RE}$, S-, and L-suffixes, respectively. Let $suf(Sx)$ be the smallest suffixes starting with each character t, let $suf(\overline{Sx})$ be the set of the other S-suffixes, let $suf(Ly)$ be the set of the largest L-suffixes starting with each character t such that no S-suffix starts with t, and let $suf(\overline{Ly})$ be the set of the other L-suffixes. We compute $\mathbf{SA}_{suf(\overline{Ly})}$, $\mathbf{RE}_{suf(Ly)\cup suf(Sx)}$, and $\mathbf{SA}_{suf(\overline{Sx})}$ from $\mathbf{SA}_{suf(L)}$ in a similar way as Transitions 1-7 in Section 4.1. Note that $\mathbf{RE}_{suf(Ly)\cup suf(Sx)}$ equals $\mathbf{SA}_{suf(Ly)\cup suf(Sx)}$ from the definition. We compute $\mathbf{SA}$ in $O(N)$ time and in-place by considering the first characters as keys, by applying Theorem 3 to $\mathbf{SA}_{suf(\overline{Ly})}$ and $\mathbf{SA}_{suf(Ly)\cup suf(Sx)}$, and then by applying the result and $\mathbf{SA}_{suf(\overline{Sx})}$.

Thus, all S-suffixes can be sorted in $O(N)$ time and in-place as in Section 4.1. See Figure 3.



$N_{suf(\overline{Lx})}$
$N_{suf(Lx)}$
subarrays X
$\mathbf{X_1}$
$\mathbf{X_2}$
$N_{suf(\overline{Ly})}$
σ
$N_{suf(\overline{Sx})}$
subarrays $\mathbf{X_3}$, $\mathbf{Y}$ and $\mathbf{Z}$
$\mathbf{X_3}$
$\mathbf{Y}$
$\mathbf{Z}$
Transition 1
$\mathbf{SA}_{suf(L)}$
Transition 2
$\mathbf{SA}_{suf(\overline{Lx})}$
$suf(Lx)$
Transition 3
$\mathbf{SA}_{suf(\overline{Lx})}$
σ
$\mathbf{Y'}_{suf(Lx)}$
$\mathbf{SA}_{suf(Lx)\cap suf(Ly)}$
Transition 4
$\mathbf{SA}_{suf(\overline{Lx})}$
$\mathbf{Y}_{suf(Ly)}$
Transition 5
$\mathbf{SA}_{suf(\overline{Ly})}$
$\mathbf{Y}_{suf(Ly)}$
Transition 6
$\mathbf{SA}_{suf(\overline{Ly})}$
$\mathbf{RE}_{suf(Ly)}$
Transition 7
$\mathbf{SA}_{suf(\overline{Ly})}$
$\mathbf{RE}_{suf(Ly)\cup suf(Sx)}$
$\mathbf{SA}_{suf(\overline{Sx})}$
Transition 8
$\mathbf{SA}_{suf(L)\cup suf(Sx)}$
$\mathbf{SA}_{suf(\overline{Sx})}$
Transition 9
$\mathbf{SA}$

Figure 3. Inside transition of **A** while computing **SA** from $\mathbf{SA}_{suf(L)}$. Space colored with gray indicates empty space.

Translating Between Wavelet Tree and Wavelet Matrix Construction

Patrick Dinklage

TU Dortmund University
Chair of Algorithm Engineering (LS11)
Otto-Hahn-Straße 14
44227 Dortmund
Germany
`patrick.dinklage@tu-dortmund.de`

Abstract. The wavelet tree (Grossi et al. [SODA, 2003]) and wavelet matrix (Claude et al. [Inf. Syst., 2015]) are compact data structures with many applications such as text indexing or computational geometry. By continuing the recent research of Fischer et al. [ALENEX, 2018], we explore the similarities and differences of these heavily related data structures with focus on their construction. We develop a data structure to modify construction algorithms for either the wavelet tree or matrix to construct instead the other. This modification is *efficient*, in that it does not worsen the asymptotic time and space requirements of any known wavelet tree or wavelet matrix construction algorithm.

Keywords: text indexing, wavelet tree, wavelet matrix

1 Introduction

The wavelet tree [5] is a data structure with numerous applications in text indexing, data compression, computational geometry (as an alternative to fractional cascading) and other areas [3, 8, 10]. Common queries that the wavelet tree can answer efficiently are *rank* and *select* for any symbol that occurs in the underlying text, as well as *access* queries to restore said text. The wavelet matrix [2] is a related data structure with the same asymptotic running times for these queries However, they are faster in practice, because they require less subqueries on bit vectors to be answered.

Both data structures are based on storing $n\lceil\log\sigma\rceil$ bits for the text of length n over an alphabet of size σ and answer access, rank and select queries in asymptotic time $\mathcal{O}(\log\sigma)$. Since they can also be used for accessing individual characters in time $\mathcal{O}(\log\sigma)$, they can both be seen as different encodings of the text. They differ (a) in the order these bits are stored, and (b) in the auxiliary data required to answer the queries. However, there are many similarities between these two data structures and it is natural to ask how far these similarities go. In this work, we focus on the construction process of the data structures.

Related work. Fischer et al. [4, Sect. 5.2] recently showed that there is a data structure to efficiently transform any construction algorithm for the wavelet tree to construct instead the wavelet matrix without worsening the asymptotic construction times. This makes it possible to apply techniques used by (parallel) wavelet tree construction algorithms, which make use of the tree structure, to the wavelet matrix, which discards the tree structure. Their data structure occupies $\mathcal{O}(n + \sigma\log n)$ bits of space and can be constructed in time $\mathcal{O}(n + \sigma)$ using $o(n + \sigma)$ bits of memory.

Our contributions. Fischer et al. left open whether there is a data structure for the inverse direction, i.e., whether there is an efficient way to construct the wavelet tree using a construction algorithm for the wavelet matrix. In order to learn more about the similarities and differences between the two, we propose a first solution to this problem and give the corresponding data structure of the same asymptotic space requirements as that in [4]. It can be constructed easily in time $\mathcal{O}(\sigma)$ from the text's histogram and its principle works for both directions. However, there is a slight limitation that gives us some insight on the different information contained in the wavelet tree and matrix.

2 Preliminaries

Let $T \in \Sigma^n$ be a text over an alphabet Σ. For some integer $i < n$, let $T[i]$ be the i-th symbol of T. We use zero-based indexing, so that $T[0]$ is the first symbol of T and $T[n-1]$ is the last.

Computational model. We use the *word RAM* model, where we assume that we can perform arithmetic operations on words of bit width $\mathcal{O}(\log n)$ in constant time.

Histogram. The *histogram* $H : c \mapsto \mathrm{occ}_T(c)$ of T maps each symbol $c \in \Sigma$ to its number $\mathrm{occ}_T(c)$ of occurrences in T. The set of those σ symbols with $\mathrm{occ}_T(c) > 0$ are the *effective alphabet* of T. We represent it as the interval $\Sigma' = [0, \sigma)$, so that the lexicographically smallest symbol is represented by 0 and the largest symbol by $\sigma - 1$. Let $\mathrm{eff}_T(c) \in \Sigma'$ be the rank of c in the effective alphabet. In the *effective transformation* T' of T, we set $T'[i] := \mathrm{eff}_T(T[i])$ for each $i < n$. As an example, consider the text and alphabet in Figure 1. The effective transformation of the text is $T' = 6\,0\,5\,1\,2\,1\,4\,4\,3\,1\,1$.

C array. For every $x \in \Sigma'$, the C *array* contains the accumulated number of occurrences of symbols in T' that are lexicographically smaller than x. Formally, it is $C[x] := \sum_{k=0}^{x-1} \mathrm{occ}_{T'}(k)$. We furthermore define $C[\sigma] := n$.

Bit vectors. A *bit vector* is a text over the binary alphabet $\mathbb{B} = \{0, 1\}$. Let $B = \mathbb{B}^n$ be a bit vector of length n. For every position $i < n$, the function $\mathrm{rank}_1(B, i)$ returns the number of 1-bits in B from its beginning up to (including) position i. For a $k > 0$, the function $\mathrm{select}_1(B, k)$ returns the position of the k-th 1-bit in B. The functions rank_0 and select_0 are defined analogously for 0-bits. There is a data structure that can answer rank and select queries for a fixed B and any i or k, respectively, in time $\mathcal{O}(1)$, requires $o(n)$ bits of memory and can be constructed in time $\mathcal{O}(n)$ [6].

Bit reversal. Let $B \in \mathbb{B}^*$ be a bit vector and let $(B)_{\mathbb{N}} \in \mathbb{N}$ denote the integer that B is the binary representation of. For $k > 0$ and an integer $i < 2^k$, we call $(i)_{\mathbb{B},k} \in \mathbb{B}^k$ the k-bit binary representation of i. Let B^R denote the reversal of B. We define the *k-bit reversal* $\mathrm{bitrev}_k(i) := (((i)_{\mathbb{B},k})^R)_{\mathbb{N}}$ as the integer represented by the reversal of i's k-bit binary representation. For a fixed k, the *bit-reversal permutation* maps each integer $i < 2^k$ to its k-bit reversal. To give examples, Table 1 shows the bit-reversal permutations for $k = 2$ and $k = 3$.

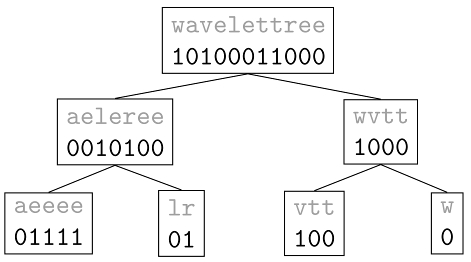
i	$(i)_{\mathbb{B},2}$	$((i)_{\mathbb{B},2})^R$	$\text{bitrev}_2(i)$
0	00	00	0
1	01	10	2
2	10	01	1
3	11	11	3

(a) Bit-reversal permutation for $k = 2$.

i	$(i)_{\mathbb{B},3}$	$((i)_{\mathbb{B},3})^R$	$\text{bitrev}_3(i)$
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

(b) Bit-reversal permutation for $k = 3$.

Table 1: Breakdowns of the bit-reversal permutations for $k = 2$ (left) and $k = 3$ (right). The first column contains the integers $i < 2^k$, the second shows their k-bit binary representations, the third shows the reversals and the final column contains the k-bit reversal of i.



$\Sigma = \{\texttt{a}, \texttt{e}, \texttt{l}, \texttt{r}, \texttt{t}, \texttt{v}, \texttt{w}\}$
$\Sigma' = [0, 7)$

c	$\text{eff}_T(c)$		
a	0	=	000_b
e	1	=	001_b
l	2	=	010_b
r	3	=	011_b
t	4	=	100_b
v	5	=	101_b
w	6	=	110_b

Figure 1: The wavelet tree (left), alphabet, effective alphabet and binary representations of symbols (right) for $T = \texttt{wavelettree}$. The texts above the node bit vectors are shown only for comprehensibility; they are not a part of the node labels and are not stored.

2.1 The Wavelet Tree

The *wavelet tree* [5] is a binary tree of height $\lceil \log \sigma \rceil$ where each node v represents an interval $[a, b] \subseteq \Sigma'$ of the effective alphabet and is labeled by a bit vector $B_v \in \mathbb{B}^+$. B_v contains one bit for each text position i, in text order, where $T'[i] \in [a, b]$: a 0-bit if $T'[i] \leq \lfloor \frac{a+b}{2} \rfloor$, i.e., if the symbol $T'[i]$ lies in the left half of the represented interval, or a 1-bit otherwise.

The root node represents the entire effective alphabet Σ' and thus its bit vector has length n. A node v has two children iff $a < b$. We apply the described structure recursively for the left child to represent the interval $[a, \lfloor \frac{a+b}{2} \rfloor]$ (the *left half*) and the right child to represent $[\lfloor \frac{a+b}{2} \rfloor + 1, b]$ (the *right half*). Following that, the tree's leaves are those nodes that represent an interval of size one, i.e., precisely one symbol from the input alphabet ($a = b$). Since the bit vector of a leaf contains only zero-bits, we need not store level $\lceil \log \sigma \rceil + 1$ of the wavelet tree, because it would consist of leaves only. Figure 1 shows an example of a wavelet tree.

The size of any node in the wavelet tree, i.e., the length of its bit vector label, can be precomputed using the C array:

Observation 1 *Let $[a, b] \subseteq \Sigma'$ be the alphabet interval represented by a wavelet tree node v. The length of the bit vector B_v that labels v is $|B_v| = C[b + 1] - C[a]$.*

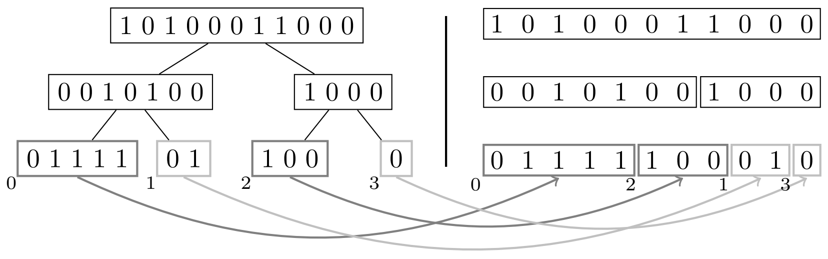


Figure 2: Comparison of the node ordering in the wavelet tree (left) and the wavelet matrix (right). Due to the nature of the bit reversal permutation, the ordering on the first two levels remains the same in the wavelet matrix. On the third level, we observe how nodes 0 and 2 (left children of their respective parents) go to the left part of the corresponding wavelet matrix bit vector and nodes 1 and 3 (right children of their respective parents) go to the right.

For storing the wavelet tree, we consider the *pointerless* representation (also known as the *levelwise* representation), where we concatenate the bit vectors on each level and enhance them by constant-time rank/select support. This is is enough information to be able to navigate in the tree [10]. The concatenation of bit vectors on any level has a length of at most n bits, so that the wavelet tree's bit vectors consume at most $n\lceil\log\sigma\rceil$ bits in total.

2.2 The Wavelet Matrix

The *wavelet matrix* [2] can be thought of as an alternative representation of the pointerless wavelet tree. In the wavelet tree, in order to retrieve the bit vector B^T_ℓ for level ℓ, we concatenate the bit vectors of the single nodes on that level from *left to right*. In the wavelet matrix, the nodes are concatenated in a different order to obtain bit vector B^M_ℓ: all left children of their respective parents are moved to the left and all right children are moved to the right. Like in the pointerless wavelet tree, we concatenate the bit vectors of all nodes on every level. Figure 2 shows an example. The re-ordering of nodes corresponds to the bit-reversal permutation of the node ranks on the respective level [4].

A practical consequence of the different ordering is that navigation in the wavelet matrix becomes easier than in the pointerless wavelet tree. In the tree, we need to keep track of the current node's interval — its left and right boundary — within the respective level's bit vector while navigating. This can be done using two rank queries on the respective bit vector when navigating from a node to either child. In the matrix, the simpler structure makes it feasible to precompute the left boundary for the right children on each level, all of which have been concatenated in the right part of the level's bit vector. This boundary is often referred to as value z in literature, as it corresponds to the number of zero bits in the bit vector. We can store z for all levels using negligible $\mathcal{O}(\log\sigma\log n)$ bits and use it to save one rank query on each level while navigating.

One could precompute the same information for the wavelet tree. However, this would require us to store the left boundary of every node, resulting in $\mathcal{O}(\sigma\log n)$ bits as there are $\mathcal{O}(\sigma)$ nodes. For this reason, the wavelet matrix can be considered more relevant for practical applications where the alphabet is large.

3 Wavelet Tree and Wavelet Matrix Construction

We continue the research of Fischer et al. [4] and are interested in how a construction algorithm for the wavelet tree or matrix can be modified efficiently to construct the other. We consider such a modification *efficient* if the asymptotic time and space boundaries of the modified construction algorithm are not worsened. Fischer et al. show that there is a data structure that can be used to efficiently transform any construction algorithm for the wavelet tree to construct instead the wavelet matrix. We propose a data structure for the inverse direction, transforming a wavelet matrix construction algorithm to one for the wavelet tree, with the same asymptotic space requirements.

Formally, let us consider the situation where, during the construction of the wavelet tree, the i-th bit is set in bit vector B^T_ℓ of level ℓ of (assuming, without loss of generality, the pointerless representation). Fischer et al. [4] present a data structure to efficiently compute a function $f : (\ell, i) \mapsto (\ell, j)$ so that j is the corresponding position for the bit to be set in bit vector B^M_ℓ of the wavelet matrix. That is, by modifying the wavelet tree constructor to set the bit at position $f(\ell, i)$ instead of i on level ℓ, it instead constructs the wavelet matrix. Because f can be computed in constant time, there is no asymptotic overhead. For input length n and alphabet size σ, their data structure occupies $n + \sigma + (\sigma + 2)\lceil \log n \rceil$ bits of space and can be constructed in time $\mathcal{O}(n + \sigma)$ using $o(n + \sigma)$ bits of memory, not worsening the asymptotic construction time and space requirements for any known wavelet tree constructor.

In the following, we first observe various properties of the wavelet tree that lead to a similar result for f as that of [4]. Based on these observations, we develop a novel data structure for the inverse f^{-1}, which maps (ℓ, j) back to (ℓ, i) with the same asymptotic time and space boundaries as for f.

3.1 Locating Nodes and Bit Offsets

As previously noted, the wavelet matrix can also be represented as a tree by re-ordering the nodes of the wavelet tree on each level according to the bit-reversal permutation. Even though there are no practical advantages of storing the wavelet matrix as a tree, the notion will help us develop our data structures.

The simple nature of the re-ordering makes it easy to translate a node *ID* (the node's rank in a breadth-first traversal of the tree) between the two data structures. Based on this, we employ the following strategy to find data structures for functions f and f^{-1}: given the level and position of the bit to be written, we attempt to find

(1) the ID of the node that the bit belongs to, and
(2) the position of the node's first bit in its level's bit vector.

Once this information is available, f and f^{-1} are easy to compute in constant time.

	a	e	l	r	t	v	w	$\top$	
c	0	1	2	3	4	5	6	7	
$\mathrm{occ}_T(c)$	1	4	1	1	2	1	1	0	
$C[c]$	0	1	5	6	7	9	10	11	11

Figure 3: The histogram and the C array for $T =$ `wavelettree`. We added the artificial symbol $\top$ so $\sigma = 8$ is a power of two. The new symbol never occurs in T and is lexicographically larger than the other symbols.

Bottom level node sizes. Observation 1 shows the relation between the C array and the sizes of the wavelet tree's nodes. This relation is especially interesting regarding the *virtual* bottom-most level $h = \lceil \log \sigma \rceil$ of a full binary wavelet tree. We call this level virtual, because all bits on it would be zero and there is no need to actually store it. On this level, each node corresponds to a single symbol from the effective alphabet. Let node v_c on level h correspond to symbol $c \in \Sigma'$. We have $|B_{v_c}| = C[c+1] - C[c] = \mathrm{occ}_{T_{\mathrm{eff}}}(c)$, i.e., the size of v_c matches the number of occurrences of c.

This property is only valid if the wavelet tree is a *full* binary tree: if it was not, there would be leaves on level $h-1$ and not all nodes on level h would exist. Without loss of generality, let us assume from now on that $\sigma = 2^h$ for some integral $h > 0$, i.e., that the alphabet size is a power of two. Then, the wavelet tree is a full binary tree. In case σ is not a power of two, we introduce artificial symbols that never occur in the input and are lexicographically *larger* than all symbols of Σ'. This way, the empty nodes for these symbols are moved to the far right of the wavelet tree and can be ignored in the following.

Locating in the wavelet tree. We consider the situation where a wavelet tree constructor sets the i-th of bit vector B^T_ℓ. Let $v(\ell, i)$ be the rank of the wavelet tree node on level ℓ to which the i-th bit belongs. We represent $v(\ell, i)$ relative to the number of the first node on level ℓ, i.e., $v(\ell, 0) = 0$ and $v(\ell, n-1) = 2^\ell - 1$. This representation requires ℓ bits, because there are precisely $2^\ell - 1$ nodes on level ℓ. Furthermore, let $p(\ell, v)$ be the position of the first bit in B^T_ℓ that belongs to node v and let $\delta_v(\ell, i) := i - p(\ell, v(\ell, i))$ be the distance of i from that position.

We take a closer look at v and p on the virtual level h and observe that

$$v(h, i) = \min\{x \mid C[x] > i\} - 1.$$

This is because each node on this level corresponds to precisely one symbol from the input alphabet and the C array encodes, for every c, the number of symbols in the input that are lexicographically smaller than c. This corresponds to the accumulated sizes of the node's left siblings. An example of this relation can be seen comparing Figure 3 and Figure 4b (in row $\ell = 3$). The node that i belongs to on level h is left of the first node whose accumulated size — its entry in the C array — exceeds i. We can immediately conclude that the first bit that belongs to node v is located at position

$$p(h, v) = C[v].$$

How do v and p on level h relate to those on the other levels $\ell < h$ that we are actually interested in? To answer this, we make use of the fact that our wavelet tree is a full binary tree: the size of a node equals the sum of its children's sizes, because the children partition the alphabet interval of their parent. As a consequence, the *accumulated* size of any node is retained in its right child, as can be seen in Figure 4b. Since the C array encodes the accumulated sizes of the nodes on level h, it also implicitly encodes the accumulated sizes of all nodes on levels $\ell < h$. Following this notion, we can conclude the following relations:

$$v(\ell, i) = \left\lfloor \frac{\min\{x \mid C[x] > i\} - 1}{2^{h-\ell}} \right\rfloor$$

and

$$p(\ell, v) = C[v \cdot 2^{h-\ell}]. \tag{1}$$

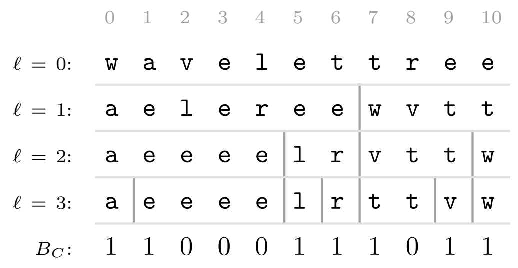
If the C array is stored in ascending order, the minimum query required to find v can be answered in time $\mathcal{O}(\log \sigma)$ using binary search. However, we seek a computation in constant time. We construct a bit vector B_C of length n and set $B_C[k] := 1$ if $C[c] = k - 1$ for some c and $B_C[k] := 0$ otherwise and prepare it for constant-time rank queries. This can be done in time $\mathcal{O}(n)$ and requires $n + o(n)$ bits of additional space. B_C marks the node boundaries on level h of the wavelet tree, see Figure 4a for an example. We can now compute

$$v(\ell, i) = \left\lfloor \frac{\mathrm{rank}_1(B_C, i) - 1}{2^{h-\ell}} \right\rfloor \tag{2}$$

in constant time.

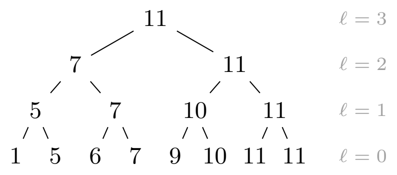
We now know that the i-th bit in B_ℓ^T corresponds to the (δ_v)-th bit in the v-th node on level ℓ in the wavelet tree. We can compute v, p and δ_v in constant time using the C array and rank-enhanced bit vector B_C, which together occupy $\sigma\lceil \log n \rceil + n(1 + o(1))$ bits of space. Asymptotically, this space boundary matches that of the data structure presented by Fischer et al. [4].

Example 1. Figure 4, in combination with Figure 3, shows an example of the data structure for $T = \texttt{wavelettree}$. Assume that we are interested in locating the node for bit $i = 9$ on level $\ell = 2$. With Equation 2, we get $v(2, 9) = \left\lfloor \frac{\mathrm{rank}_1(B_C, 9) - 1}{2^{3-2}} \right\rfloor = \left\lfloor \frac{5}{2} \right\rfloor = 2$.



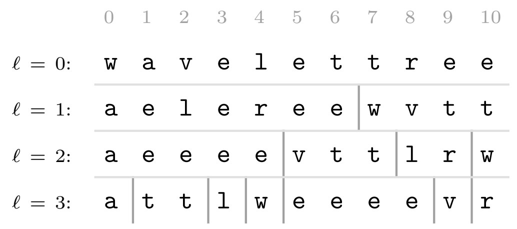
	0	1	2	3	4	5	6	7	8	9	10
$\ell = 0$:	w	a	v	e	l	e	t	t	r	e	e
$\ell = 1$:	a	e	l	e	r	e	e	w	v	t	t
$\ell = 2$:	a	e	e	e	e	l	r	v	t	t	w
$\ell = 3$:	a	e	e	e	e	l	r	t	t	v	w
B_C:	1	1	0	0	0	1	1	1	0	1	1

(a) The text re-ordering on each level and the bit vector B_C. The vertical lines mark the boundaries of the wavelet tree's nodes.



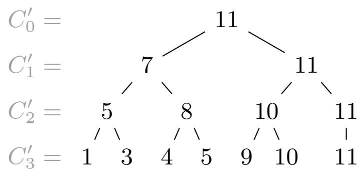
(b) The accumulated sizes of each of the wavelet tree's nodes. Note that the rightmost node on the bottom level corresponds to our artificial symbol $\top$ from Figure 3.

Figure 4: Display of the wavelet tree's text re-ordering on each level, including the virtual level $h = 3$, the bit vector B_C and the accumulated node sizes for our running example text $T = \texttt{wavelettree}$.



	0	1	2	3	4	5	6	7	8	9	10
$\ell = 0$:	w	a	v	e	l	e	t	t	r	e	e
$\ell = 1$:	a	e	l	e	r	e	e	w	v	t	t
$\ell = 2$:	a	e	e	e	e	v	t	t	l	r	w
$\ell = 3$:	a	t	t	l	w	e	e	e	e	v	r

(a) The text re-ordering on each level of the wavelet matrix. The vertical lines mark the boundaries of the *nodes* of the wavelet matrix.



(b) The accumulated sizes of each of the *nodes* of the wavelet matrix. Note that C'_3, the bottom level, is not actually needed and depicted only for the sake of completeness.

Figure 5: Display of the wavelet matrix's text re-ordering on each level for running example text $T =$ `wavelettree`.

This means that the bit belongs to the third node on level 2 (because we start counting at zero). Furthermore, with Equation 1, we get $p(2,2) = C[2 \cdot 2^{3-2}] = C[4] = 7$. This means that the third node on level 2 starts at position 7. Finally, it is $\delta_v(2,9) = 9 - p(2,2) = 9 - 7 = 2$, so bit 9 on level 2 ultimately corresponds to the third bit of the third node on that level.

Locating in the wavelet matrix. The question is how a similar locating can be done for the wavelet matrix. As previously mentioned, the bit vector B^M_ℓ of the wavelet matrix is the concatenation of the wavelet tree's node bit vectors on level ℓ in bit-reverse order.

We consider the situation where a wavelet matrix constructor sets the j-th bit of bit vector B^M_ℓ and are interested in the node to which this bit belongs. Analogously to v, p and δ_v, we define $u(\ell,j)$, $q(\ell,u)$ and $\delta_u(\ell,j) := j - q(\ell, u(\ell,i))$ as the node into which the written bit belongs, the position of the node's first bit in B^M_ℓ and the distance of j from the node's first bit, respectively.

Due to the re-ordering of the nodes, the correspondences between their accumulated sizes and the C array, which we observed for the wavelet tree, are no longer valid for the wavelet matrix. As a consequence, we need to find a different way to compute u and q.

The following observation is useful to find u: in both the wavelet tree and the wavelet matrix [2, Prop. 1], all occurrences of a symbol $c \in \Sigma'$ belong to the same node on any level. Therefore, in order to find the node to which any occurrence of c belongs on virtual level h, it suffices to know to which node the *first* occurrence of c belongs. This first occurrence of c on level h is always located at position $C[c]$. As seen previously, once the node for level h is known, it is easy to narrow it down to any level $\ell < h$. Of course, we then have the node in the wavelet *tree*, but in the wavelet matrix, the nodes are simply permuted in bit-reverse order. Let c be the symbol from which we computed the bit that we are setting in B^M_ℓ. If c is known, we can express

$$u(\ell,j,c) = \mathrm{bitrev}_\ell(v(\ell, C[c])). \tag{3}$$

The consequences of having to know c are discussed later.

It remains to compute q. As stated above, the C array cannot be used directly to compute the accumulated node sizes for the wavelet matrix, because nodes are permuted. However, the node sizes themselves remain the same and thus, with awareness of the bit-reversal ordering of nodes on every level, it is easy to precompute the accumulated node sizes for all nodes of the wavelet matrix using the C array in time $\mathcal{O}(\sigma)$. Since we are dealing with a full binary tree of height $h = \log \sigma$, the accumulated wavelet matrix node sizes can be stored in an array C' of length $2^h - 1 = \sigma - 1$ (since σ is a power of two), occupying $(\sigma - 1)\lceil \log n \rceil$ bits of space. Figure 5b shows an example. We imagine C' to be a set of arrays C'_ℓ for each level ℓ, so that the first entry of C'_ℓ contains the size of the first node on level ℓ. Then, q can be found as follows:

$$q(\ell, u) = \begin{cases} 0 & \text{if } u = 0. \\ C'_\ell[u-1] & \text{if } u > 0. \end{cases} \tag{4}$$

We then know that the j-th bit in B^M_ℓ of the wavelet matrix corresponds to the δ_u-th bit in the u-th node's bit vector on level ℓ. We can compute u, q and δ_u in constant time using the arrays C and C' and rank-enhanced bit vector B_C, which, in total, occupy $(2\sigma - 1)\lceil \log n \rceil + n(1 + o(1))$ bits of space.

Example 2. Figure 5, in combination with Figure 4 and Figure 3, shows an example for the data structure for T = `wavelettree`. Assume that we are interested in locating the node for bit $j = 9$ on level $\ell = 2$ of the wavelet matrix. The symbol for which the bit is written is $c = \texttt{r}$ (see Figure 5a). With Equation 3, we get $u(2, 9, \texttt{r}) = \text{bitrev}_3(v(2, C[\texttt{r}])) = \text{bitrev}_3(v(2, 6)) = \text{bitrev}_2(1) = 2$. This means that the bit belongs to the third node on level 2. Furthermore, with Equation 4, we get $q(2, 2) = C'_2[2 - 1] = 8$. This means that the third node on level 2 starts at position 8. Finally, it is $\delta_u(2, 9) = 9 - 8 = 1$, so bit 9 on level 2 ultimately corresponds to the second bit of the third node on that level.

3.2 Translating Between Wavelet Tree and Wavelet Matrix Construction

Using the locating data structures described above, we can express functions f and f^{-1} as follows:

$$f(\ell, i) = q(\ell, \text{bitrev}_\ell(v(\ell, i))) + \delta_v(\ell, i),$$
$$f^{-1}(\ell, j, c) = p(\ell, \text{bitrev}_\ell(u(\ell, j, c))) + \delta_u(\ell, j, c).$$

Both f and f^{-1} can be computed in constant time using the arrays C, C' and rank-enhanced bit vector B_C. These occupy $\sigma\lceil \log n \rceil + (2\sigma - 1)\lceil \log n \rceil + n(1 + o(1))$ bits of space can be constructed in time $\mathcal{O}(\sigma + n)$.

Limitations. We impose the restriction that for f^{-1}, the symbol c, for which a bit is being set in B^M_ℓ, has to be known when setting the bit. Even though this bit must ultimately have been computed from c, there are construction algorithms for the wavelet tree that redistribute the bits of c before constructing the bit vectors [1, 7, 9, 11]. Due to the existence of our function f alone, such techniques may as well be used for the construction of the wavelet matrix. In this case, c is *not* known when setting the bit in question and f^{-1} cannot be used.

More generally, in the wavelet tree, c is always implicitly given by the tree structure itself and implicitly used by f by jumping to the virtual bottom level to the leaf that would represent c via the C array. The wavelet matrix discards the tree structure and the information is lost, so that we need to receive it from the constructor in order to compute f^{-1}.

4 Conclusions

We solved an open theoretical problem concerning the construction of wavelet trees and wavelet matrices. We described a data structure that can be used to extend a construction algorithm for the wavelet matrix to construct instead the wavelet tree with constant time overhead. This data structure can be constructed in time $\mathcal{O}(\sigma+n)$ time and it requires $\mathcal{O}(\sigma \log n + n)$ bits of memory, matching the asymptotic time and space requirements of the data structure described by Fischer et al. [4] for the inverse direction, transforming wavelet tree construction into wavelet matrix construction.

However, because the wavelet matrix discards the wavelet tree's binary tree structure, we require some additional information from the constructor for our computations. This limitation makes our data structure unsuitable for the class of wavelet matrix constructors that do not keep the entire binary representation of the input symbols when computing the bit vectors. To that end, it is still open whether there is a data structure for our translation function with the same (or lower) asymptotic time and space requirements that does not require any information other than the position of the written bit in the wavelet matrix.

Acknowledgements

We would like to thank Johannes Fischer and Florian Kurpicz from the TU Dortmund University's Chair of Algorithm Engineering for the motivation of this work and the supportive discussions related to the topic.

References

1. M. A. Babenko, P. Gawrychowski, T. Kociumaka, and T. A. Starikovskaya: *Wavelet trees meet suffix trees*, in 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM, 2015, pp. 572–591.
2. F. Claude, G. Navarro, and A. O. Pereira: *The wavelet matrix: An efficient wavelet tree for large alphabets.* Inf. Syst., 47 2015, pp. 15–32.
3. P. Ferragina, R. Giancarlo, and G. Manzini: *The myriad virtues of wavelet trees.* Inform. and Comput., 207(8) 2009, pp. 849–866.
4. J. Fischer, F. Kurpicz, and M. Löbel: *Simple, fast and lightweight parallel wavelet tree construction*, in 20th Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM, 2018, pp. 9–20.
5. R. Grossi, A. Gupta, and J. S. Vitter: *High-order entropy-compressed text indexes*, in 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM, 2003, pp. 841–850.
6. G. Jacobson: *Space-efficient static trees and graphs*, in 30th Symposium on Foundations of Computer Science (FOCS), IEEE, 1989, pp. 549–554.
7. Y. Kaneta: *Fast wavelet tree construction in practice*, in 25th International Symposium on String Processing and Information Retrieval (SPIRE), Springer, 2018, pp. 218–232.
8. V. Mäkinen and G. Navarro: *Position-restricted substring searching*, in 7th Latin American Theoretical Informatics Symposium (LATIN), vol. 3887 of Lecture Notes in Computer Science, Springer, 2006, pp. 703–714.
9. J. I. Munro, Y. Nekrich, and J. S. Vitter: *Fast construction of wavelet trees.* Theor. Comput. Sci., 638 2016, pp. 91–97.
10. G. Navarro: *Wavelet trees for all.* J. Discrete Algorithms, 25 2014, pp. 2–20.
11. G. Tischler: *On wavelet tree construction*, in 22nd Annual Symposium on Combinatorial Pattern Matching (CPM), Springer, 2011, pp. 208–218.

Author Index

Proceedings of the Prague Stringology Conference 2019
Edited by Jan Holub and Jan Žďárek
Published by: Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science
Prague Stringology Club
Thákurova 9, Praha 6, 160 00, Czech Republic.
First edition.

ISBN 978-80-01-06618-8

URL: `http://www.stringology.org/`
E-mail: `psc@stringology.org` Phone: +420-2-2435-9811

Printed by powerprint s.r.o.
Brandejsovo nám. 1219/1, Praha 6 Suchdol, 165 00, Czech Republic