Proceedings of the Prague Stringology Conference 2018

Edited by Jan Holub and Jan Žďárek

August 2018

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

ISBN 978-80-01-06484-9

Preface

The proceedings in your hands contains a collection of papers presented in the Prague Stringology Conference 2018 (PSC 2018) held on August 27–28, 2018 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 ten 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 "Discovery of regulatory motifs in DNA" by Esko Ukkonen.

The Prague Stringology Conference has a long tradition. PSC 2018 is the twenty-second PSC conference. In the years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conferences (PSC's) in 2001–2006, 2008–2017 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 2018 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 2018. 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 2018

Jan Holub and Tomi S. Klein

Conference Organisation

Program Committee

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

Organising Committee

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

External Referees

Keisuke Goto	Arnaud Lefebvre	Gabriele Fici
Diptarama Hendrian	Elise Prieur-Gaston	Donald Adjeroh

Table of Contents

Invited Talk

Contributed Talks

Discovery of Regulatory Motifs in DNA
(*Abstract*)

Esko Ukkonen

Department of Computer Science
University of Helsinki
P.O. Box 68, FI-00014, Finland
esko.ukkonen@cs.helsinki.fi

In biological sequence analysis, representation and discovery of various DNA motifs with a biological function is a central task. In particular, identification of regulatory elements such as the binding sites in DNA for the so-called transcription factors is an important step in the attempts to understand the regulation mechanisms of gene expression. Transcription factors are proteins that may bind to DNA, typically close to transcription start site of a gene. Such a binding may activate or inhibit the transcription machinery of the associated gene. As the regulated gene may again be a transcription factor, such pairwise regulatory relations between genes induce a genome-wide network model for gene regulation.

The possible binding sites of a transcription factor are short DNA segments. The DNA sequences of different sites are close variants of an underlying consensus sequence. For most transcription factors no biophysical model of this variation is currently known. Hence simplified formal representations of binding motifs have been used: a motif is represented as a set of weighted DNA sequences that may occur in the binding site, or a probabilistic Markov model of order 0 or 1 or higher is used. Here Markov model of order 0 is usually called a position weight matrix (PWM). Once we have available training DNA sequences that contain enriched amounts of instances of the motif, we may estimate the motif in our representation class that fits best the training data.

The talk surveys the representations and corresponding discovery algorithms for transcription factor binding motifs. We consider basic motifs for single factors (monomers) as well as composite motifs for pairs of factors (dimers) and for chains of factors. Such chains are models for regulatory modules built of clusters of several factors making together a regulatory complex. We will discuss both the EM algorithm based learning of motifs which is the dominating approach in practice as well as a novel seed-driven approach aiming at faster learning. The role of string algorithms in these methods will also be discussed.

Esko Ukkonen: Discovery of Regulatory Motifs in DNA, p. 1.
Proceedings of PSC 2018, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06484-9

Fibonacci Based Compressed Suffix Array

Ekaterina Benza[1], Shmuel T. Klein[2], and Dana Shapira[1]

[1] Dept. of Computer Science, Ariel University, Ariel 40700, Israel
benzakate@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. We propose Fibonacci based compressed suffix arrays, and show how repeated decompression can be avoided using our scheme. For a given file T of size n, the implementation requires $1.44\,n\,H_k + n + o(n)$ bits of space, where H_k is the k-th order empirical entropy of T, while retaining the searching functionalities. Empirical results support this theoretical bound improvement, and show that on most files, our implementation saves space as compared to previous suggestions.

1 Introduction

Given a text and some pattern we wish to locate in it, the *suffix array* of the text is a *self index*, meaning that the retrieval is done directly on the suffix array itself, without the use of the text. That is, the text is implicitly encoded, and the searching process decompresses only the necessary portion of the text. More formally, let T be a string of length $n-1$ over an alphabet Σ of size σ. A *suffix array* (SA) for $T\$$, $\$ \notin \Sigma$, is an array $SA[1:n]$ of the indices of the suffixes of $T\$$ which have been arranged in lexicographic order. By convention, \$ is lexicographically smaller than all other characters.

Suffix arrays have been introduced by Manber and Myers [17], and are more space efficient than *suffix trees* (compact tries), because suffix trees generally require additional space to store all the internal pointers in the tree. The *compressed suffix array* (CSA) introduced by Grossi and Vitter [9] is a text index that uses $2n\log\sigma$ bits in the worst case, and $O(m)$ processing time for searching a pattern of length m. Sadakane [19] extended the searching functionality to a self index, and proved that it uses search time $O(m\log n)$, and space $\epsilon^{-1}nH_0 + O(n\log\log\sigma)$ bits, where $0<\epsilon<1$ and $\sigma \leq \log^{O(1)} n$, H_0 being the 0-order empirical entropy of T.

Grossi et al. [8] present an implementation of compressed suffix arrays that achieves asymptotic entropy space as well as fast pattern matching. More precisely, the CSA uses $nH_k + O(\frac{n\log\log n}{\log_\sigma n})$ bits and $O(m\log\sigma + \text{polylog}(n))$ searching time, where H_k is the k-th order empirical entropy of T.

Ferragina and Manzini [6] introduce the *FM-index*: a text index based on the Burrows-Wheeler Transform [4], which supports efficient pattern matching using a *Backwards Search*. The FM-index uses at most $5nH_k+o(n)$ bits for small alphabet size σ, and $O(m+\log^{1+\epsilon} n)$ searching time. We refer the reader to the book of Navarro [18] for a comprehensive review on compact data structures in general and compressed suffix arrays in particular.

Huo et al. [10] construct a space efficient CSA; Huo et al. [11] extend their work for the reference genome sequence and propose approximate pattern matching on the compressed suffix array for short read alignment. Their implementation uses $2nH_k + n + o(n)$ bits of space, for $k \leq c\log_\sigma n - 1$ and any $c < 1$. They report

Ekaterina Benza, Shmuel T. Klein, Dana Shapira: Fibonacci Based Compressed Suffix Array, pp. 3–11.
Proceedings of PSC 2018, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06484-9

on extensive experiments to evaluate their CSA compression, construction time, and pattern matching processing time performance. The results suggest that their compression performance is better than that of the implementation of Sadakane [19] and the FM-index [6], except for evenly distributed data like that of DNA files.

In this paper we suggest the usage of Fibonacci Codes instead of Elias' C_γ code used in [10,11], and show how decompression can be avoided using our scheme. The implementation requires $1.44\,n\,H_k+n+o(n)$ bits of space, while retaining the searching functionalities. Empirical results support this theoretical bound improvement, and show that on most files, our implementation saves space as compared to the one of [10,11].

The paper is organized as follows. Section 2 recalls the details of CSA and Section 3 presents its Fibonacci coding based variant, including the analysis of its space requirements and the summation process applied on the compressed form. Empirical results are given in Section 4.

2 Compressed Suffix Array

A suffix array (SA) for $T\$$, where T is a string over Σ and $\$ \notin \Sigma$, is an array $SA[1:n]$ of the indices of the suffixes of $T\$$, stored in lexicographical order. That is, if $SA[i] = j$ then the suffix starting at the j-th position of T, $T[j:n]$, is the i-th item in the lexicographically sorted list of all n suffixes of $T\$$.

The numbers in a suffix array can be stored using $n \log n$ bits, as they are a permutation of the numbers $\{1, \ldots, n\}$, that require $\log n! = \Omega(n \log n)$ bits, at least. However, not all permutations correspond to actual suffix arrays, as there are only σ^n different texts of length n over Σ. Thus a better lower bound is, in fact, $n \log \sigma$ bits. Grossi and Vitter [9] improve the space requirements of a suffix array by decomposing it based on the *neighbor function* defined as follows.

$$\Phi[i] = j, \qquad \text{if } \; SA[j] = 1 + SA[i] \bmod n.$$

The inverse function $SA^{-1}[j]$ gives the position of $T[j:n]$ in the sorted list of the suffixes of T. The function Φ can also be rewritten as:

$$\Phi[i] = SA^{-1}[1 + SA[i] \bmod n]$$

If $SA[i]$ refers to the suffix $T[j:n]$, then $\Phi[j] = i'$ is the position where $SA[i'] = j+1$ refers to suffix $T[j+1:n]$. If $SA[i] = j$ then $SA[\Phi[i]] = j+1$, $SA[\Phi[\Phi[i]]] = j+2$, and generally, $SA[\Phi^{(k)}[i]] = j+k$.

It has been shown that the values of Φ at consecutive positions referring to suffixes that start with the same symbol must be increasing. This claim is explained as follows. Let i and $i+1$ be two adjacent indices in SA that correspond to suffixes that start by the same symbol. The index i cannot be 1, as the symbol at the first position corresponds to \$ that occurs only once. Let $j = SA[i]$ and $j' = SA[i+1]$. Following our assumption that they belong to suffixes starting with the same symbol, we get that $T[j] = T[j']$. Since $T[j:n] \prec T[j':n]$, it follows that $T[j+1,n] \prec T[j'+1,n]$, and $j'+1$ appears to the right of $j+1$ in SA. The position where $j'+1$ appears in SA is $SA^{-1}[j'+1] = SA^{-1}[SA[i+1]+1] = \Phi[i+1]$. Using the same argument, the position where $j+1$ appears in SA is $\Phi[i]$, thus, $\Phi[i] < \Phi[i+1]$.

As Φ is an increasing function for suffixes starting with the same symbol, Φ can be partitioned into σ increasing arrays $\Phi_a = [1 : n_a]$, for all $a \in \Sigma$, where n_a is the

number of occurrences of the character a in the text. As an example, consider the text $T =$ `mississippi`\$. The text itself, the suffix array SA, its inverse function SA^{-1}, and Φ are given in the first rows in Figure 1. The last row partitions the Φ row into subintervals, denoted by Φ_i, Φ_m, Φ_p and Φ_s, each referring to a different character of T. The first cell does always refer to the special character \$, denoted by $\Phi_\$$. To better understand this partition, we have preceded it with a row giving the first character of the corresponding suffix, that is, holding $T[SA[i]]$ at position i.

	1	2	3	4	5	6	7	8	9	10	11	12
T	**m**	**i**	**s**	**s**	**i**	**s**	**s**	**i**	**p**	**p**	**i**	\$
SA	12	11	8	5	2	1	10	9	7	4	6	3
SA^{-1}	6	5	12	10	4	11	9	3	8	7	2	1
Φ	6	1	8	11	12	5	2	7	3	4	9	10
$T[SA]$	\$	i	i	i	i	m	p	p	s	s	s	s
Φ_a	$\Phi_\$$	Φ_i				Φ_m	Φ_p		Φ_s			

Figure 1. CSA example for $T =$ `mississippi`\$

The implementation for CSA used in [10,11] applies differential encoding. Instead of Φ itself, the values $\Delta\Phi[i] = \Phi[i] - \Phi[i-1]$ in each block are encoded, except for the first entry, which is assumed to be 0, thus need not be encoded. These differences are then encoded using *Elias'* methods [5]. Elias considered mainly two *fixed* codeword sets, C_γ and C_δ, in what he calls *universal* codes, in which the integers are represented by binary sequences.

The Elias C_γ encoding of an integer x starts with a unary codeword of the number of bits in x followed by the standard binary codeword for x without its leading 1 bit. That is, $1 + \lfloor \log_2 x \rfloor$ is coded in unary, and $x - 2^{\lfloor \log_2 x \rfloor}$ is coded in binary for a total of $1 + 2\lfloor \log_2 x \rfloor$ bits. The Elias C_δ encoding uses the C_γ codeword for the number of bits in x, which requires $1 + 2\lfloor \log_2 \log_2 2x \rfloor$ bits, and again is followed by the binary codeword for x without the leading 1 bit, for a total of $1 + 2\lfloor \log_2 \log_2 2x \rfloor + \lfloor \log_2 x \rfloor$. A sample of Elias C_γ and C_δ codewords appears in Table 1 where blanks are inserted between the unary and the binary parts for clarity.

To provide faster access to the C_γ encoded sequence S of integers, which we denote as $C_\gamma(S)$, it is partitioned into so-called *super-blocks*, which in turn are sub-partitioned into *blocks*, and three auxiliary tables SB, B and SAM are defined. For given values of a and b, which are defined in the following paragraph, $\mathsf{SB}[0 : \frac{n}{a} - 1]$ stores the starting position of the encoding of each super-block in $C_\gamma(S)$, i.e., the total number of bits in super-blocks preceding the current super-block; $\mathsf{B}[0 : \frac{n}{b} - 1]$ stores the starting position in $C_\gamma(S)$ of the encoding of every block relative to the beginning of its corresponding super-block; and $\mathsf{SAM}[0 : \frac{n}{b} - 1]$ contains sampling values of Φ, so that the first value in each block is stored.

Each super-block refers to the encoding of $a = \log^3 n$ elements, and each block refers to the encoding of $b = \log^2 n$ elements. While the super-blocks store the absolute number of bits up to that position, the blocks record the relative position with respect to the beginning of the super-block. Figure 2 uses our running example illustrating the parsing of Φ into super-blocks of size $a = 8$ and into blocks of size $b = 4$. The differences are given in the row denoted by $\Delta\Phi$, and are encoded according to Elias'

C_γ. More precisely, the series $\Phi[ib+1]-\Phi[ib],\ldots,\Phi[(i+1)b-1]-\Phi[(i+1)b-2]$ is C_γ encoded, for all $0 \le i \le \frac{n}{b}$, and $\Phi[ib]=0$ is not encoded. In case the difference is negative, the value $\Phi[i]-\Phi[i-1]+n$ is used. The table also contains the encodings corresponding to the Fibonacci variants Fib_1 and Fib_2, presented in the next section, as well as the matching SB and B arrays.

	1 2 3	4	5 6 7	8	9 10 11 12
SAM	6		12		3
Φ	**6** 1 8	11	**12** 5 2	7	**3** 4 9 10
$\Delta\Phi$	0 8 7	3	0 6 10	5	0 1 5 1
$C_\gamma(\Delta\Phi)$	0001000	00111 011	00110 0001010	00101	1 00101 1
SB	0				32
B	0		15		0
$\mathsf{Fib}_1(\Delta\Phi)$	000011	01011 0011	10011 010011	00011	11 00011 11
SB	0				31
B	0		15		0
$\mathsf{Fib}_2(\Delta\Phi)$	100101	101001 1001	100001 1010001	10101	1 10101 1
SB	0				34
B	0		16		0

Figure 2. Super blocks and regular blocks parsing of Φ for $a=8$ and $b=4$ using C_γ, Fib_1 and Fib_2 codes.

The decoding function, denoted by $\mathcal{D}(\mathcal{E},s,\ell)$, is given the encoded array $\mathcal{E}$ to be decoded, the starting position s within $\mathcal{E}$, and the number of codewords ℓ to be decoded. The values of Φ are then computed using:

$$\Phi[i] = \mathsf{SAM}\left[\left\lfloor\tfrac{i}{b}\right\rfloor\right] + \mathcal{D}\left(\mathcal{E},\ \mathsf{SB}\left[\left\lfloor\tfrac{i}{a}\right\rfloor\right] + \mathsf{B}\left[\left\lfloor\tfrac{i}{b}\right\rfloor\right],\ i \bmod b\right). \tag{1}$$

To obtain $\Phi[i]$, SB and B are accessed to determine the corresponding bit position within $\mathcal{E}$. Starting at that position, $i \bmod b$ codewords are decoded and added to the sample values stored in SAM. As an example using Fib_2, $\Phi[11] = \mathsf{SAM}[11/4] + \mathcal{D}(\mathcal{E}, \mathsf{SB}[11/8]+\mathsf{B}[11/4], 11 \bmod 4) = \mathsf{SAM}[2]+\mathcal{D}(\mathcal{E},\mathsf{SB}[1]+\mathsf{B}[2],2) = 3+\mathcal{D}(\mathcal{E},34+0,2)$. Two consecutive values, 1 and 5, are decoded, and are added to 3, so that the final result 9 is returned.

3 Fibonacci Encodings

The lengths of the C_γ codewords grow logarithmically, which yields good asymptotic behavior. However, C_γ is then often efficient only for quite large alphabets, whereas the number of different elements in the CSA for natural language texts is usually small. The same is true for several other universal codeword sets such as ETDC [3] and (s,c)-dense codes [2]. This was also the motivation of using C_γ instead of the asymptotically better C_δ representation in the implementation of Huo et al. [10]. The

Fibonacci code is yet another universal variable length encoding of the integers, based on the sum of Fibonacci numbers rather than on the sum of powers of 2, as in the *standard* binary representation. More precisely, any number $x \geq 0$ can be uniquely represented by the string $b_r b_{r-1} \cdots b_2 b_1$, with $b_i \in \{0, 1\}$, such that $x = \sum_{i=1}^{r} b_i F_i$, where the Fibonacci numbers F_i are defined by:

$$F_i = F_{i-1} + F_{i-2} \qquad \text{for } i \geq 1,$$

and the boundary conditions

$$F_0 = 1 \qquad \text{and} \qquad F_{-1} = 0.$$

The uniqueness of the representation for every integer x is achieved by building the representation according to the following procedure: find the largest Fibonacci number F_r smaller than or equal to x, and repeat the process recursively with $x - F_r$. For example, $79 = 55 + 21 + 3 = F_9 + F_7 + F_3$, so its Fibonacci representation would be 101000100. As a result of this encoding scheme, there are never consecutive Fibonacci numbers in any of these sums, implying that in the corresponding binary representation, there are no adjacent 1s. It thus suffices to precede the Fibonacci based representation of any integer by a single 1-bit, which can act like a comma, to obtain a uniquely decipherable code.

The properties of Fibonacci codes have been exploited in several useful applications: robustness to errors [1], direct access [16], fast decoding and compressed search [13,15], compressed matching in dictionaries [14], faster modular exponentiation [12], etc. The present work is yet another application of this idea.

One variant of the Fibonacci code, denoted here by Fib_1, simply reverses the codewords in order to achieve an instantaneous code [7]. The adjacent 1s are then at the right instead of at the left end of each codeword, yielding the prefix code, a sample of which is presented in Table 1 in the column headed by Fib_1.

Another variant, denoted here by Fib_2, was introduced in [7], and found to be often preferable for the $\Delta\Phi$ encoding. The set of codewords Fib_2 is constructed from the set Fib_1 by omitting the rightmost 1-bit of every codeword and prefixing each codeword by 10; for example, 0100011 (for encoding the number 15 in Fib_1) is transformed into **10**010001 (for encoding the number 16 in Fib_2). As a result, every codeword now starts and ends with a 1-bit, so codeword boundaries may still be detected by the occurrence of the string 11. Since, as a result of this transformation, the shortest codeword 101 is of length three, one may add 1 as a single codeword of length 1, which explains the shift in the indices of corresponding codewords. Table 1 presents several codewords for Elias C_γ and C_δ, presented in the first two columns, followed by Fib_1 and Fib_2. For each presented value, the codewords of shortest length are emphasized, unless all are of the same length. Although most of the codewords of Fib_1 are the shortest, its disadvantage over the other codes is the encoding of the value 1 that uses two bits instead of a single one. This was found to be empirically crucial for our data sets, as the number 1 was the most common value to be encoded.

3.1 Space Analysis

Recall that H_k denotes the k-th order empirical entropy. Huo et al. [10] prove that the space used for the Elias C_γ based $\Delta\Phi$ encoding is $2nH_k + n + o(n)$ bits in the worst case for any $k \leq c \log_a n - 1$ and any constant $c < 1$. Navarro [18] shows that if $\Delta\Phi$ is encoded using C_δ, the space for CSA is $nH_k + n + O(n)$.

i	C_γ	C_δ	Fib_1	Fib_2
1	**1**	**1**	11	**1**
2	**01 0**	010 0	**011**	**101**
3	**01 1**	010 1	0011	1001
4	001 00	011 00	**1011**	10001
5	001 01	011 01	00011	10101
6	**001 10**	**011 10**	**10011**	100001
7	**001 11**	**011 11**	**01011**	101001
8	0001 000	00100 000	**000011**	**100101**
9	0001 001	00100 001	**100011**	1000001
10	0001 010	00100 010	**010011**	1010001
30	00001 1110	00101 1110	**10001011**	10 0000101
100	0000001 100100	00111 100100	**0010100001**	100100100001

Table 1. Several codewords of universal codes C_γ, C_δ, Fib_1 and Fib_2.

C_γ and C_δ require $2\lfloor \log x \rfloor + 1$ and $\lfloor \log x \rfloor + 1 + 2\lfloor \log(\lfloor \log x \rfloor + 1) \rfloor$ bits, respectively, to encode the number x. To evaluate the corresponding Fibonacci codeword lengths, let F_r be the largest Fibonacci number smaller than or equal to the given number x. Then r bits are necessary to encode x. A well known approximation to Fibonacci number F_r is $\frac{\phi^r}{\sqrt{5}}$, where $\phi = \frac{1+\sqrt{5}}{2} = 1.618$ is the golden ratio. From the fact that $F_{r-1} < x \leq F_r$ we may extract that r is of the order of

$$\log_\phi x = (\log_\phi 2)\ \log_2 x = 1.4404 \log x.$$

That is, the lengths of Fibonacci codewords are asymptoticly between those of C_γ and C_δ. However, in practice, Fibonacci codes may be preferable in case the numbers are not uniformly distributed, as in our application of compressed suffix arrays.

Emulating the space analysis given in [10] for the Elias C_γ encoded CSA, replacing the length estimates of $2 \log x$ for a value x by $1.44 \log x$, we get that at most $1.44\, n\, H_k(1 + o(1)) + O(n)$ bits are needed for the Fibonacci based representation of the CSA, for any $k \leq c \log_\sigma n - 1$, and any constant $c < 1$.

3.2 Compressed addition

According to equation (1), in order to obtain $\Phi[i]$, $i \bmod b$ codewords need to be decoded. The traditional approach is to decode each codeword and add the decoded values. One of the advantages of using a Fibonacci based representation of the integers is that it is possible to perform this addition directly on the compressed form of the CSA, without individually decoding each summand.

To add $i \bmod b$ Fibonacci encoded numbers, first strip the appended 1 for Fib_1 or the prepended 10 for Fib_2 (except for the first codeword 1, which is given a special treatment), and pad, if necessary, the shorter codewords with zeros at their right end so that all representations are of equal length ℓ. Considering this as an $(i \bmod b) \times \ell$ matrix, we record the number of 1-bits in each column into an array $C[1 : \ell]$. The sought result is obtained by summing $\sum_{j=1}^{\ell} C[j]F_j$ for Fib_1, or by summing $\sum_{j=1}^{\ell} C[j]F_j + i$ for Fib_2.

For example, assume that $(i \bmod b) = 5$ differences $\Delta\Phi[i]$, $2, 3, 5, 6$, and 4, should be added to obtain $2 + 3 + 5 + 6 + 4 = 20$. They are represented in Fib_1 as $\mathsf{011}$, $\mathsf{0011}$, $\mathsf{00011}$, $\mathsf{10011}$, and $\mathsf{1011}$, respectively.

The steps proposed are:

1. Strip the appended 1: resulting in 01-1, 001-1, 0001-1, 1001-1, and 101-1.
2. Pad the shorter codewords with 0s so that all of them are of length $\ell = 4$: 0100, 0010, 0001, 1001, and 1010.
3. Regard them as an $(i \mod b) \times \ell = 5 \times 4$ matrix:

$$\begin{pmatrix} 0\,1\,0\,0 \\ 0\,0\,1\,0 \\ 0\,0\,0\,1 \\ 1\,0\,0\,1 \\ 1\,0\,1\,0 \end{pmatrix}$$

4. Record the number of ones in each column in $C[1:4] = [2, 1, 2, 2]$.
5. $\sum_{j=1}^{\ell} C[j]F_j$ for i is $2 \cdot 1 + 1 \cdot 2 + 2 \cdot 3 + 2 \cdot 5 = 20$, as expected.

Encoding the same example, $2, 3, 5, 6, 4$, using Fib_2, attains 101, 1001, 10101, 100001, and 10001, respectively. Striping the prepended 10 and padding by 0s, we receive 1-000, 01-00, 101-0, 0001, and 001-0. Finally, putting them in a matrix:

$$\begin{pmatrix} 1\,0\,0\,0 \\ 0\,1\,0\,0 \\ 1\,0\,1\,0 \\ 0\,0\,0\,1 \\ 0\,0\,1\,0 \end{pmatrix}$$

$C[1:4]$ is then $[2, 1, 2, 1]$, and $\sum_{j=1}^{\ell} C[j]F_j + i$ is $2 \cdot 1 + 1 \cdot 2 + 2 \cdot 3 + 1 \cdot 5 + 5 = 20$, as expected.

The processing time is thus proportional to the size of the compressed file, which is, asymptotically and empirically on our test files, smaller than the corresponding C_γ encodings used in [10], and does not require decoding tables.

Similarly, the Elias C_γ code could be partially used directly in its compressed form, as the summation of integers represented by codewords of the same length can be evaluated by adding the binary parts, and copying the common unary part, or extending it by a single 1-bit if there has been a carry in the addition. However, handling codewords of different lengths is more involved.

4 Experimental Results

We considered the same test files as [10], taken from the Pizza & Chili Corpus[1] as well as from the Canterbury Corpus[2]. We used the implementation of [10][3] and adapted it to encode the $\Delta\Phi$ values with Fib_1 and Fib_2, instead of Elias C_γ. We also report the space usage of Elias C_δ, as it is asymptotically the best of these four universal codes. The space usage for the super-blocks and blocks in our implementation is about the same as for the implementation of [10]. Tables 2 and 3 report the sizes in MBs of the encodings of $\Delta\Phi$ using these universal codes, Table 2 corresponding to files taken

[1] http://pizzachili. dcc.uchile.cl
[2] http://corpus.canterbury.ac.nz
[3] https://github.com/Hongweihuo-Lab/CSA

from the Canterbury Corpus, and Table 3 referring to files of size 100MB each taken from the Pizza & Chili corpus.

As can be seen, Fib_2 based CSA encoding performs the best on most files. Surprisingly, C_γ gives the best results for DNA and E.COLI, which are two files in the test files of [10] for which FM-index and Sadakane's CSA implementation produce better results than C_γ. Huo et al. explain this performance by the frequency of small values (1 and 2) in $\Delta\Phi$, which tends to be lower in these files than in the others. The other file for which Fib_2 does not produce the most efficient CSA is PROTEINS, for which it is outperformed by Fib_1.

Name	size (MB)	C_γ	C_δ	Fib_1	Fib_2
E.COLI	4.42	**1.923**	2.158	2.033	2.025
BIBLE	3.859	1.342	1.378	1.557	**1.320**
WORLD192	2.36	0.776	0.772	0.923	**0.747**
NEWS	0.36	0.178	0.175	0.183	**0.169**
BOOK1	0.73	0.348	0.358	0.361	**0.341**
PAPER1	0.05	0.024	0.024	0.025	**0.023**
KENNEDY	0.98	3.360	3.155	3.640	**3.049**

Table 2. Canterbury Corpus CSA using C_γ, C_δ, Fib_1 and Fib_2.

Name	C_γ	C_δ	Fib_1	Fib_2
DNA	**40.32**	44.99	43.79	42.24
DBLP.XML	22.90	23.15	32.12	**22.51**
SOURCES	31.92	31.69	38.48	**30.72**
ENGLISH	37.53	38.23	42.40	**36.79**
PROTEINS	65.51	64.89	**62.21**	62.72

Table 3. Pizza & Chili Corpus CSA using C_γ, C_δ, Fib_1 and Fib_2.

5 Conclusion

Huo et al. [10] present experiments showing that their CSA implementation is empirically better than the FM-index and Sadakane's CSA implementations on most tested files. We suggest here a Fibonacci based CSA, which generally achieves even better compression performance on the same data-sets.

However, the power of Fibonacci encoding has still not been fully exploited, especially the fact that adjacent 1's indicate the codewords' boundaries for both Fib_1 and Fib_2. For instance, this feature can replace the usage of the array B needed to indicate the beginning of each block of codewords relative to the start position of the corresponding super-block, yielding additional savings. This trade-off of time versus space will be addressed in future work.

Acknowledgement: We would like to thank Hongwei Huo for sharing the implementation of [10].

References

1. A. Apostolico and A. S. Fraenkel: *Robust transmission of unbounded strings using Fibonacci representations.* IEEE Trans. Information Theory, 33(2) 1987, pp. 238–245.
2. N. R. Brisaboa, A. Fariña, G. Navarro, and M. F. Esteller: *(s, c)-dense coding: An optimized compression code for natural language text databases*, in String Processing and Information Retrieval, 10th International Symposium, SPIRE 2003, Manaus, Brazil, October 8-10, 2003, Proceedings, 2003, pp. 122–136.
3. N. R. Brisaboa, E. L. Iglesias, G. Navarro, and J. R. Paramá: *An efficient compression code for text databases*, in Advances in Information Retrieval, 25th European Conference on IR Research, ECIR 2003, Pisa, Italy, April 14-16, 2003, Proceedings, 2003, pp. 468–481.
4. M. Burrows and D. J. Wheeler: *A block sorting lossless data compression algorithm*, in Technical Report 124, Digital Equipment Corporation, 1994.
5. P. Elias: *Universal codeword sets and representations of the integers.* IEEE Trans. Information Theory, 21(2) 1975, pp. 194–203.
6. P. Ferragina and G. Manzini: *Indexing compressed text.* J. ACM, 52(4) 2005, pp. 552–581.
7. A. S. Fraenkel and S. T. Klein: *Robust universal complete codes for transmission and compression.* Discrete Applied Mathematics, 64(1) 1996, pp. 31–55.
8. R. Grossi, A. Gupta, and J. S. Vitter: *High-order entropy-compressed text indexes*, in Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '03, Philadelphia, PA, USA, 2003, Society for Industrial and Applied Mathematics, pp. 841–850.
9. R. Grossi and J. S. Vitter: *Compressed suffix arrays and suffix trees with applications to text indexing and string matching.* SIAM Journal on Computing, 35(2) 2005, pp. 378–407.
10. H. Huo, L. Chen, J. S. Vitter, and Y. Nekrich: *A practical implementation of compressed suffix arrays with applications to self-indexing*, in Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26-28 March, 2014, 2014, pp. 292–301.
11. H. Huo, Z. Sun, S. Li, J. S. Vitter, X. Wang, Q. Yu, and J. Huan: *CS2A: A compressed suffix array-based method for short read alignment*, in 2016 Data Compression Conference, DCC 2016, Snowbird, UT, USA, March 30 - April 1, 2016, 2016, pp. 271–278.
12. S. T. Klein: *Should one always use repeated squaring for modular exponentiation?* Inf. Process. Letters, 106(6) 2008, pp. 232–237.
13. S. T. Klein and M. K. Ben-Nissan: *On the usefulness of Fibonacci compression codes.* Comput. J., 53(6) 2010, pp. 701–716.
14. S. T. Klein and D. Shapira: *Compressed pattern matching in JPEG images.* Int. J. Found. Comput. Sci., 17(6) 2006, pp. 1297–1306.
15. S. T. Klein and D. Shapira: *Compressed matching for feature vectors.* Theor. Comput. Sci., 638 2016, pp. 52–62.
16. S. T. Klein and D. Shapira: *Random access to Fibonacci encoded files.* Discrete Applied Mathematics, 212 2016, pp. 115–128.
17. U. Manber and G. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM Journal on Computing, 22(5) 1993, pp. 935–948.
18. G. Navarro: *Compact Data Structures - A Practical Approach*, Cambridge University Press, 2016.
19. K. Sadakane: *New text indexing functionalities of the compressed suffix arrays.* J. Algorithms, 48(2) 2003, pp. 294–313.

$O(n \log n)$-time Text Compression by LZ-style Longest First Substitution

Akihiro Nishi, Yuto Nakashima, Shunsuke Inenaga,
Hideo Bannai, and Masayuki Takeda

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

Abstract. Mauer et al. [A Lempel-Ziv-style Compression Method for Repetitive Texts, PSC 2017] proposed a hybrid text compression method called *LZ-LFS* which has both features of Lempel-Ziv 77 factorization and longest first substitution. They showed that LZ-LFS can achieve better compression ratio for repetitive texts, compared to some state-of-the-art compression algorithms. The drawback of Mauer et al.'s method is that their LZ-LFS compression algorithm takes $O(n^2)$ time on an input string of length n. In this paper, we show a faster LZ-LFS compression algorithm that works in $O(n \log n)$ time. We also propose a simpler version of LZ-LFS that can be computed in $O(n)$ time.

1 Introduction

Text compression is a task to compute a small representation of an input text (or string). Given a vast amount of textual data that has been produced to date, text compression can play central roles in saving memory space and reducing data transmission costs.

Lempel-Ziv 77 (*LZ77*) [12] is a fundamental text compression method that is based on a greedy factorization of the input string. LZ77 factorizes a given string w of length n into a sequence of non-empty substrings $f_1, \ldots, f_k$ such that (1) $w = f_1 \cdots f_k$ and (2) each factor f_i is the longest prefix of $w[|f_1 \cdots f_{i-1}| + 1..n]$ that has an occurrence beginning at a position in range $[1..|f_1 \cdots f_{i-1}|]$ (this is a self-reference variant), or $f_i = c$ if it is the leftmost occurrence of the character c in w. Each factor f_i in the first case is encoded as a reference pointer to one of its previous occurrences in the string. LZ77 and its variants are basis of many text compression programmes, such as gzip.

In the last two decades, *grammar compression* has also gathered much attention. Grammar compression finds a small context-free grammar which generates only the input string. Since finding the smallest grammar representing a given string is NP-hard [9,8], various kinds of efficiently-computable greedy grammar compression algorithms have been proposed. The most well-known method called Re-pair [3] is based on a most frequent first substitution approach, such that most frequently occurring bigrams (substrings of length 2) are replaced with new non-terminal symbols recursively, until there are no bigrams with at least two non-overlapping occurrences. An alternative is a *longest first substitution* (*LFS*) approach, where longest substrings that have at least two non-overlapping occurrences are replaced with new non-terminal symbols recursively, until there are no substrings of length at least two with at least two non-overlapping occurrences.

Recently, Mauer et al. [5] proposed a hybrid text compression algorithm called *LZ-LFS*, which has both features of LZ77 and LFS. Namely, LZ-LFS finds a longest substring which occurs at least twice in the string, replaces its selected occurrences

Akihiro Nishi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda: $O(n \log n)$-time Text Compression by LZ-style Longest First Substitution, pp. 12–26.
Proceedings of PSC 2018, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06484-9

with a special symbol #, and encodes each of them as a reference to its leftmost occurrence. This is continued recursively, until there are no substrings of length at least two which occur at least twice in the string. The details on how the occurrences to replace are selected can be found in [5] as well as in a subsequent section in this paper. Mauer et al. showed that LZ-LFS can have good practical performance in compressing repetitive texts. Indeed, in their experiments, the compression ratio of LZ-LFS outperforms that of some state-of-the-art compression algorithms on data sets from widely-used corpora. The drawback, however, is that Mauer et al.'s compression algorithm for LZ-LFS takes $O(n^2)$ time for input strings of length n.

In this paper, we focus on a theoretical complexity for computing LZ-LFS, and propose a faster LZ-LFS algorithm which runs in $O(n \log n)$ time with $O(n)$ space. Our algorithm is based on Nakamura et al.'s algorithm for LFS-based grammar compression [7]. Although Nakamura et al.'s algorithm is quite involved, our algorithm for LZ-LFS is much less involved due to useful properties of LZ-LFS. We also show that a simplified version of LZ-LFS can be computed in $O(n)$ time and space with slight modifications to our algorithm.

2 Preliminaries

2.1 String notations

Let Σ be an alphabet. An element of Σ^* is called a *string*. Strings x, y, and z are said to be a *prefix*, *substring*, and *suffix* of string $w = xyz$, respectively.

The length of a string w is denoted by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. The i-th character of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of a string w that begins at position i and ends at position j is denoted by $w[i..j]$ for $1 \leq i \leq j \leq |w|$. For convenience, let $w[i..j] = \varepsilon$ for $j < i$, and $w[i..] = w[i..|w|]$ for $1 \leq i \leq |w|$.

An *occurrence* of a substring x of a string w is an interval $[i..i + |x| - 1]$ such that $w[i..i + |x| - 1] = x$. For simplicity, we will sometimes call the beginning position i of x as an occurrence of x in w. Let $\mathsf{Occ}_w(x)$ denote the set of the beginning positions of the occurrences of x in w. If x does not occur in w, then $\mathsf{Occ}_w(x) = \emptyset$.

If $|\mathsf{Occ}_w(x)| \geq 2$, then x is said to be a *repeat* of w. A repeat x of w is said to be a *longest repeat* (*LR*) of w if there are no repeats of w that are longer than x. We remark that there can exist more than one LR for w in general. A repeat y of w is said to be a *maximal repeat* of w if for any characters $a, b \in \Sigma$, $|\mathsf{Occ}_w(ay)| < |\mathsf{Occ}_w(y)|$ and $|\mathsf{Occ}_w(yb)| < |\mathsf{Occ}_w(y)|$. We also remark that any longest repeat of w is a maximal repeat of w.

Let $I = \{i_1, \ldots, i_k\} \subseteq \mathsf{Occ}_w(x)$ be a (sub)set of occurrences of a repeat x in w such that $k \geq 2$ and $i_1 < \cdots < i_k$. The occurrences in I are said to be *overlapping* if $i_1 + |x| - 1 \geq i_k$, and are said to be *non-overlapping* if $i_j + |x| - 1 < i_{j+1}$ for all $1 \leq j < k$.

2.2 Suffix trees

Assume that any string w terminates with a unique symbol \$ which does not occur elsewhere in w. The *suffix tree* of a string w, denoted $\mathsf{STree}(w)$, is a path-compressed trie such that each edge is labeled with a non-empty substring of a string of w, each internal node has at least two children, the labels of all out-going edges of each node

begin with mutually distinct characters, and each suffix of w is spelled out by a path starting from the root and ending at a leaf. Because we have assumed that w terminates with a unique symbol \$, there is a one-to-one correspondence between the suffixes of w and the leaves of $\mathsf{STree}(w)$. The *id* of a leaf of $\mathsf{STree}(w)$ is defined to be the beginning position of the suffix of w that it represents.

Each node of $\mathsf{STree}(w)$ is specifically called as an *explicit* node, and in contrast a locus on an edge is called as an *implicit* node. For ease of explanation, we will sometimes identify each node of $\mathsf{STree}(w)$ with the string obtained by concatenating the edge labels from the root to that node. In the sequel, the *string depth* of a node implies the length of the string that the node represents.

Each edge label x is represented by a pair (i, j) of positions in w such that $w[i..j] = x$, and in this way $\mathsf{STree}(w)$ can be represented with $O(n)$ space. Every explicit node v of $\mathsf{STree}(w)$ except for the root node has an auxiliary reversed edge called the *suffix link*, denoted $\mathsf{slink}(v)$, such that $\mathsf{slink}(v) = v'$ iff v' is a suffix of v and $|v'|+1 = |v|$. Notice that if v is a node of $\mathsf{STree}(w)$, then such node v' always exists in $\mathsf{STree}(w)$. $\mathsf{STree}(w)$ can be constructed in $O(n)$ time and space if a given string w of length n is drawn from an integer alphabet of size $n^{O(1)}$ [1], or in $O(n \log \sigma)$ time and $O(n)$ space if w is drawn from a general ordered alphabet and w contains σ distinct characters [11,6,10].

3 Text compression by LZ-style longest first substitution

Mauer et al. [5] proposed a text compression method which is a hybrid of the Lempel-Ziv 77 encoding (LZ) [12] and a grammar compression with longest first substitution (LFS) [7], which hereby is called *LZ-LFS*.

3.1 LZ-LFS

Here we describe how LZ-LFS compresses a given string w.

Let x be an LR of w, and let ℓ be the leftmost occurrence of x in w. Let $\mathsf{LGOcc}_w(x)$ denote the set of non-overlapping occurrences of x in w that are selected in a left-greedy manner (i.e., greedily from left to right). Notice that $\ell = \min(\mathsf{LGOcc}_w(x)) = \min(\mathsf{Occ}_w(x))$. An occurrence i of w is said to be of

- Type 1 if i is the second leftmost occurrence of x (i.e., $i = \min(\mathsf{Occ}_w(x) \setminus \{\ell\})$) and the occurrences ℓ and i overlap (i.e., $\ell + |x| - 1 \geq i$).

Let ℓ' be the Type 1 occurrence of x in w if it exists, and let

$$e = \begin{cases} \ell' + |x| - 1 & \text{if } \ell' \text{ exists,} \\ \ell + |x| - 1 & \text{otherwise.} \end{cases} \tag{1}$$

An occurrence i of x in w is said to be of

- Type 2 if i is the leftmost occurrence of x after e and there is no non-overlapping occurrence of x to the right of i (i.e., $\{i\} = \mathsf{LGOcc}_{w[e+1..]}(x)$).
- Type 3 if i is a left-greedily selected occurrence of x after e (i.e., $i \in \mathsf{LGOcc}_{w[e+1..]}(x)$) and there are at least two such occurrences of x (i.e., $|\mathsf{LGOcc}_{w[e+1..]}(x)| \geq 2$).
- Type 4 otherwise.

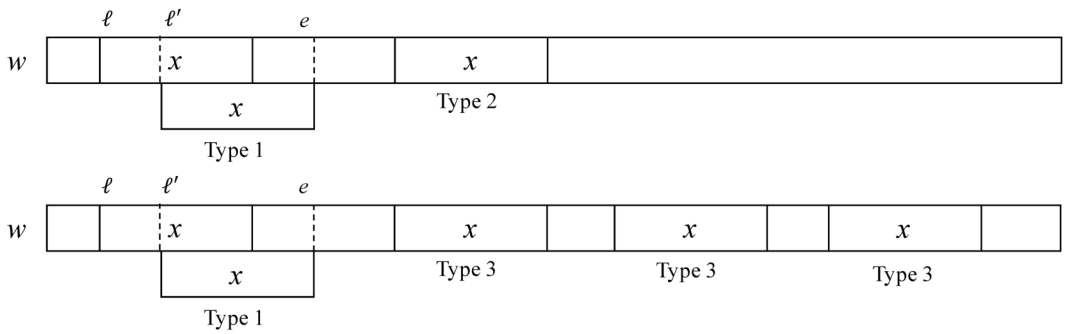


Figure 1. Upper: Type-2 occurrence when Type 1 occurrence exists. Lower: Type 3 occurrences when Type 1 occurrence exists.

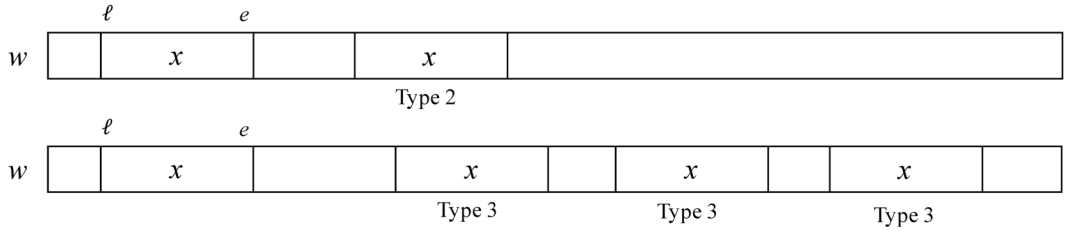


Figure 2. Upper: Type-2 occurrence when Type 1 occurrence does not exist. Lower: Type 3 occurrences when Type 1 occurrence does not exist.

Note that Type 2 and Type 3 occurrences of x cannot simultaneously exist. See Figures 1 and 2 for illustration.

LZ-LFS is a recursive greedy text compression method which works as follows: Given an input string w, LZ-LFS first finds an LR x of w and picks up its Type 1 occurrence (if it exists), and either its Type 2 occurrence or its Type 3 occurrences. Each of these selected occurrences of x is replaced with a special symbol # not appearing in w, together with a pointer to the leftmost occurrence ℓ of x which still remains in the modified string. The encoding of this pointer differs for each type of occurrences, see [5] for details. We remark that Type 4 occurrences are not selected for replacement and all the Type 4 occurrences but the leftmost occurrence of x disappear in the modified string. In the next step, LZ-LFS finds an LR of the modified string which does *not* include #, and performs the same procedure as long as there is a repeat in the modified string.

Let w_k denote the modified string in the kth step. Namely, $w_0 = w$ and w_k is the string after all the selected occurrences of an LR of w_{k-1} have been replaced with #. LZ-LFS terminates when it encounters the smallest m such that w_m does not contain repeats of length at least two which consists only of characters from the original string w (i.e., repeats without #'s).

LZ-LFS computes a list *Factors* as follows: Initially, *Factors* is an empty list. For each occurrence i of LR x that has been replaced with #, a pair $(\ell, |x|)$ of its leftmost occurrence ℓ and the length $|x|$ is added to *Factors* if it is of Type 2 or the first occurrence of Type 3. Otherwise (if it is of Type 1), then a pair $(i - \ell, |x|)$ is added to *Factors*. These pairs are arranged in *Factors* in increasing order of the corresponding occurrences in the input string.

LZ-LFS also computes an array F as follows: Suppose we have computed $w' = w_m$. For each $1 \leq h \leq |F|$, if the h-th # from the left in w' replaced a Type 1 occurrence

of an LR, then $F[h] = 1$. Similarly, if the h-th # from the left in w' replaced a Type 2 occurrence of an LR, then $F[h] = 2$. For Type 3 occurrences, $F[h] = 2 + j$ if the h-th # from the left in w' replaced the j-th LR that that has Type 3 occurrences. This array F can be computed e.g., by using an auxiliary array A of length n, where each entry is initialized to null. For each occurrence i of each LR x that has been replaced with #, the type of the occurrence (Type 1, 2, or 3) is stored at $A[i]$. After the final string $w' = w_m$ has been found, non-null values of A are extracted by a left-to-right scan, and are stored in F from left to right. A tuple $(w', \mathit{Factors}, F)$ is the output of the compression phase of LZ-LFS.

To see how LZ-LFS compresses a given string, let us consider a concrete example with string

$$w = w_1 = \texttt{abcabcaabcdabcacabc\$}.$$

There are two LRs `abca` and `cabc` in w, and suppose that `abca` has been selected to replace. Below, we highlight the occurrences of `abca` with underlines:

$$w_1 = \underline{\underline{\texttt{abca}}}\underline{\texttt{bca}}\texttt{abcd}\underline{\texttt{abca}}\texttt{cabc\$}.$$

The wavy-underlined occurrence of `abca` at position 4 is of Type 1 since it overlaps with the leftmost occurrence of `abca` which is doubly underlined. Then, pair $(3, 4)$ is added to *Factors*, where the first term 3 is the distance from the occurrence at position 4 to the leftmost occurrence at position 1, and the second term 4 is $|\texttt{abca}|$.

The singly underlined occurrence of `abca` at position 12 is of Type 2 since it does not overlap with the leftmost occurrence of `abca`, and there are no occurrences of `abca` to its right. Then, pair $(1, 4)$ is added to *Factors*, where 1 is the leftmost occurrence of `abca` and $4 = |\texttt{abca}|$.

These Type 1 and Type 2 occurrences of `abca` are replaced with with #, and the resulting string is

$$w_2 = \underline{\underline{\texttt{abc}}}\texttt{\#}\underline{\texttt{abc}}\texttt{d\#c}\underline{\texttt{abc}}\texttt{\$},$$

of which `abc` is an LR. Since neither the second occurrence nor the third one of `abc` overlaps with the leftmost occurrence of `abc`, both of these occurrences are of Type 3. Hence, pair $(1, 3)$ is added to *Factors*, where 1 is the leftmost occurrence of `abc` and $3 = |\texttt{abc}|$. Finally, we obtain

$$w_3 = \texttt{abc\#\#d\#c\#\$}.$$

Since w_3 has no repeats of length at least two which does not contain #'s, LZ-LFS terminates here. Together with this final string $w' = w_3$, LZ-LFS outputs $\mathit{Factors} = \langle(3, 4), (1, 3), (1, 4)\rangle$ and $F = [1, 3, 2, 3]$. Recall that the pairs in *Factors* are arranged in increasing order of the corresponding occurrences in the input string w.

Mauer et al. [5] showed how to decompress $(w', \mathit{Factors}, F)$ to get the original string w in $O(n)$ time. On the other hand, Mauer et al.'s LZ-LFS compression algorithm for computing $(w', \mathit{Factors}, F)$ from the input string w of length n uses $O(n^2)$ time and $O(n)$ space. Their algorithm is based on the suffix array and the LCP array of w [4].

In this paper, we propose a faster LZ-LFS compression algorithm for computing $(w', \mathit{Factors}, F)$ in $O(n \log n)$ time with $O(n)$ space, which is based on suffix trees and Nakamura et al.'s algorithm [7] for a grammar compression with LFS.

3.2 Differences between LZ-LFS and grammar compression with LFS

Here, we briefly describe main differences between LZ-LFS and grammar compression with LFS. In the sequel, grammar compression with LFS will simply be called LFS.

The biggest difference is that while the output of LFS is a context free grammar that generates only the input string w, that of LZ-LFS is not a grammar. Namely, in LFS each selected occurrence of the LR is replaced with a new non-terminal symbol, but in LZ-LFS each selected occurrence of the LR is represented as a pointer to the left-most occurrence of the LR in the current string w_k. This also implies that in LZ-LFS the left-most occurrence of the LR can remain in the string w_{k+1} for the next $(k+1)$-th step. On the other hand, in LFS no occurrences of the LR are left in the string for the next step.

Because of Type 1 occurrences, a repeat which only has overlapping occurrences in the current string w_k can become an LR in LZ-LFS. On the contrary, since LFS is a grammar-based compression, LFS always chooses a longest repeat which has non-overlapping occurrences.

The above differences also affect technical details of the algorithms. Nakamura et al.'s algorithm for LFS maintains an incomplete version of the sparse suffix tree [2] of the current string. On the other hand, our algorithm for LZ-LFS maintains the suffix tree of the current string w_k in each k-th step.

3.3 On parameters α and β

The algorithm of Mauer et al. [5] uses the suffix array and the LCP array [4] of the input string w, and finds an LR x_k for w_k at each k-th step using a maximal interval of the LCP array.

The suffix array SA for a string w of length n is a permutation of $[1..n]$ such that $\mathsf{SA}[j] = i$ iff $w[i..]$ is the lexicographically j-th suffix of w. The LCP array LCP for w is an array of length n such that $\mathsf{LCP}[1] = 0$ and $\mathsf{LCP}[i]$ stores the length of the longest common prefix of $w[\mathsf{SA}[i-1]..]$ and $w[\mathsf{SA}[i]..]$ for $2 \leq i \leq n$.

For a positive integer p, an interval $[i..j]$ of LCP array of w is called a *p-interval* if (1) $\mathsf{LCP}[i-1] < p$, (2) $\mathsf{LCP}[k] \geq p$ for all $i \leq k \leq j$, (3) $\mathsf{LCP}[k] = p$ for some $i \leq k \leq j$, and (4) $\mathsf{LCP}[j+1] < p$ or $j = n$. An interval $[i..j]$ of LCP array of w is called a *maximal interval* if it is a p-interval for some $p \geq 1$ and the longest common prefix of length p for all the corresponding suffixes $w[\mathsf{SA}[i]..], \ldots, w[\mathsf{SA}[j]..]$ is a maximal repeat of w. In each step of Mauer et al.'s method, the algorithm picks up a maximal interval as a candidate for an LR to replace.

Let $\mathsf{bit}(w')$, $\mathsf{bit}(F)$, and $\mathsf{bit}(\mathit{Factors})$ respectively denote the average number of bits to encode a single character from w', an element of F, and an element of *Factors* with a fixed encoding scheme. The original algorithm by Mauer et al. [5] uses two parameters α and β such that $\alpha = \frac{\mathsf{bit}(\mathit{Factors})}{\mathsf{bit}(w')}$ and $\beta = 1 + \frac{\mathsf{bit}(F)}{\mathsf{bit}(w')}$. In each k-th step, their algorithm performs replacement of an LR x_k of length len_k only if the following conditions holds:

$$\mathit{len}_k \geq \frac{\alpha}{s} + \beta, \tag{2}$$

where s denotes the number of Type 2 or Type 3 occurrences of the LR x_k in the current string w_k. However, since the values of α and β cannot be precomputed, in their implementation of LZ-LFS, they use ad-hoc pre-determined values for α and β. In particular, they set $\alpha = 30$ and $\beta = 80$ as default values in their experiments (see [5] for details).

However, we have found that there exist a series of strings for which Mauer et al.'s algorithm fails to recursively replace LRs for *any* pre-determined values for α and β.

Consider a series of strings

$$w = aXab_0aXab_1 \cdots aXab_s\$,$$

where $s \geq 1$, $a, b_1, \ldots, b_s \in \Sigma$, $a \neq b_i$ for any $0 \leq i \leq s$, $b_i \neq b_j$ for any $0 \leq i \neq j \leq s$, and $X \in (\Sigma \setminus \{a, b_0, \ldots, b_s, \$\})^+$. This string $w = w_1$ has a unique LR aXa. Hence we have $len_1 = r + 2$, where $r = |X|$. Since there are $s > 1$ non-overlapping occurrences of aXa which do not overlap with the left most occurrence of aXa in w, those occurrences are of Type 3. For this LR aXa to be replaced with $\#_1$, Inequality (2) or alternatively $r \geq \frac{\alpha}{s} + \beta - 2$ needs to hold. Now let us choose $1 \leq |X| = r < \beta - 1$ and $s \geq \alpha$. Then, since $\frac{\alpha}{s} \leq 1$, Inequality (2) never holds for such r. Hence, the original algorithm of Mauer et al. does not replace aXa and tries to find a next LR (which can be shorter than aXa). In this case, the second longest repeats are aX and Xa of length $r+1$ each. However, since neither is aX nor Xa a maximal repeat of w, it is not represented by a maximal interval of the LCP array. Hence, neither is aX nor Xa selected for replacement. Moreover, note that even X is not a maximal repeat of w, and that there are no repeats of length at least two consisting only of a and/or b_i $(0 \leq i \leq s)$. Therefore, Mauer et al.'s algorithm terminates at this point and does not compress this string $w = aXab_0aXab_1 \cdots aXab_s\$$ at all, even though it is highly repetitive and contains quite long repeats (e.g., for Mauer et al.'s default value $\beta = 80$, X can be as long as 78).

We also remark that one can easily construct instances where more candidates of LRs have to be skipped, by adding other strings in a similar way to X into the string, e.g., $aXab_0aXab_1 \cdots aXab_saYac_0aYac_1 \cdots aYac_s\$$, and so on.

Given the above observation, in our algorithm that follows, we will omit the condition of Inequality (2), and will replace Type 1, 2, 3 occurrences of *any* selected LR.

4 $O(n \log n)$-time algorithm for LZ-LFS

In this section, we show the following result:

Theorem 1. *Given a string w of length n, our algorithm for LZ-LFS works in $O(n \log n)$ time with $O(n)$ space.*

We begin with describing a sketch of our LZ-LFS algorithm. Let w be the input string of length n and let $w_1 = w$. As a preprocessing, we construct $\mathsf{STree}(w_1)$ in $O(n \log \sigma)$ time and $O(n)$ space [11,6,10], where $\sigma \leq n$ is the number of distinct characters that occur in w.

In the first step of the algorithm, we find an LR x_1 of w_1 with the aid of $\mathsf{STree}(w_1)$. Let w_k denote the string in the k-th step of the algorithm. For a technical reason, when computing w_{k+1} from w_k, we use a special symbol $\#_k$ that does not occur in w_k, and replace the selected occurrences of an LR x_k in w_k with $\#_k$. The reason will become clear later.

For each k-th step, we denote by len_k the length of an LR of w_{k-1}, namely, $len_k = |x_k|$. At the end of each k-th step, we update our tree so that it becomes identical to $\mathsf{STree}(w_{k+1})$, so that we can find an LR x_{k+1} for the next $(k+1)$-th step.

4.1 How to find an LR x_k using $\mathsf{STree}(w_k)$

Suppose that we maintain $\mathsf{STree}(w_k)$ in each k-th step. The two following lemmas are keys to our algorithm. There, each $\#_k$ used at each k-th step is regarded as a single character of length one, rather than a representation of the LR of length $len_k \geq 2$ that was replaced by $\#_k$.

Lemma 2. *For each k-th step, let v be any internal explicit node of $\mathsf{STree}(w_k)$ of string depth at least two. Then, the string represented by v does not contain $\#_j$ with any $1 \leq j < k$.*

Proof. Assume on the contrary that the string represented by v contains $\#_j$ for some $1 \leq j < k$. Since v is an internal explicit node of $\mathsf{STree}(w_k)$, v occurs at least twice in w_k. Since $|v| \geq 2$, we have that $len_k \geq |v| > len_j$. However, this contradicts the longest first strategy such that $len_j \geq len_k$ must hold. □

Lemma 3. *For each k-th step, any LR of w_k is represented by an internal node of $\mathsf{STree}(w_k)$.*

Proof. Suppose on the contrary that an LR x of w_k is represented by an implicit node of $\mathsf{STree}(w_k)$, and let (u, v) be the edge on which x is represented. Note that $|v| > |x|$. Since x is an LR, x must occur at least twice in w_k and hence v cannot be a leaf of $\mathsf{STree}(w_k)$. This implies that v is an internal branching node and hence v occurs at least twice in w_k. However, this contradicts that x is an LR of w_k. □

Based on Lemmas 2 and 3, we can find an LR at each step as follows. In each k-th step of our algorithm, we maintain an array B_k of length n such that $B_k[l]$ stores a list of all explicit internal nodes of string depth l that exist in $\mathsf{STree}(w_k)$. Hence, $B_k[len_k]$ will be the leftmost entry of B_k that stores a non-empty list of existing nodes. We do not store nodes of string depth one. Any node of string depth one represents either a single character from the original string w or $\#_j$ for some $1 \leq j < k$ which will never be replaced in the following steps. Therefore, $B_k[1]$ is always empty at every k-th step.

The initial array B_1 can easily be computed in $O(n)$ time by a standard traversal on $\mathsf{STree}(w_1) = \mathsf{STree}(w)$. We can also compute in $O(n)$ time the length len_1 of an LR for B_1 in a naïve manner. We then pick up the first element in the list stored at $B_1[len_1]$ as an LR x_1 of w_1 to be replaced with $\#_1$. After the replacement, we remove x_1 from the list, and proceed to the next step. In the next subsection, we will show how to efficiently update B_k to B_{k+1}.

The algorithm terminates when the string contains no repeats of length at least two. Let w_m denote this string, namely, the algorithm terminates at the m-th step. In this last m-th step, $\mathsf{STree}(w_m)$ consists only of the root, the leaves, and possibly internal explicit nodes of string depth one.

In the next subsection, we will show how to efficiently update $\mathsf{STree}(w_k)$ to $\mathsf{STree}(w_{k+1})$ and B_k to B_{k+1} in a total of $O(n)$ time for all $k = 1, \ldots, m-1$. We also remark that m cannot exceed $n/2$ since at least two positions are taken by the replacement of an LR at each step.

Now, let us focus on how our algorithm works at each k-th step. The next lemma shows how we can find the occurrences of an LR of each step efficiently.

Lemma 4. *Given a node of $\mathsf{STree}(w_k)$ which represents an LR x_k of w_k at each k-th step, we can compute Type 1, 2, 3 occurrences of x_k in w_k in a total of $O(n \log n)$ time and $O(n)$ space for all steps.*

Proof. It follows from Lemma 3 that all children of the node for x_k are leaves in $\mathsf{STree}(w_k)$. We sort all the leaves in increasing order of their id's (i.e., the beginning positions of the corresponding suffixes). If d_k is the number of the above-mentioned leaves, then this can be done in $O(d_k \log d_k)$ time and $O(d_k)$ space by a standard sorting algorithm. It is clear that we can compute Type 1, 2, and/or 3 occurrences of x_k in w_k from this sorted list, in $O(d_k)$ time.

Each occurrence i of x_k but the leftmost one either (a) is replaced with $\#_k$, or (b) overlaps with another occurrence of x_k that is replaced with $\#_k$. In case (a), it is guaranteed that there will be no LRs that begin at position i in the following steps, since LZ-LFS chooses repeats in a longest first manner. In case (b), there is another occurrence j of x_k that is replaced with $\#_k$ and $i \in [j+1..j+len_k-1]$. Since these positions in this range $[j+1..j+len_k-1]$ are already taken by the replacement of x_k with $\#_k$, there will be no LRs that begin at position i in the following steps. One delicacy is the leftmost occurrence ℓ of x_k, since the corresponding interval $[\ell..\ell+len_k-1]$ can contain up to len_k occurrences of x_k, and these positions may retain the original characters in the string w_{k+1} for the next $(k+1)$-th step. However, since at least one occurrence of x_k is always replaced, the cost of sorting the leaves whose id's are in range $[\ell..\ell+len_k-1]$ can be charged to an occurrence of x_k that is replaced with $\#_k$.

Overall, the time cost to sort all d_k children of x_k can be charged to the intervals of the occurrences of x_k in w_k that are replaced with $\#_k$'s. Therefore, the total time cost for sorting the corresponding leaves in all m steps is $O(\sum_{k=1}^{m}(d_k \log d_k)) = O(n \log n)$, where the equality comes from the fact that $\sum_{k=1}^{m} d_k = O(n)$ and $d_k \leq n$ for each k.

The space complexity is clearly $O(n)$. □

4.2 How to update $\mathsf{STree}(w_k)$ to $\mathsf{STree}(w_{k+1})$

In this subsection, we show how to update $\mathsf{STree}(w_k)$ to $\mathsf{STree}(w_{k+1})$.

Let i be any occurrence (Type 1, 2, or 3) of an LR x_k in w_k which will be replaced with $\#_k$ in the k-th step. Since $|x_k| = len_k \geq 2$, the replacement with $\#_k$ will always shrink the string length. However, it is too costly to relabel the integer pairs for the suffix tree edge labels with the positions in the shrunken string. To avoid this, we suppose that each selected occurrence of x_k is replaced with $\#_k\bullet^{len_k-1}$, where $\bullet$ is a special symbol that does not occur in the original string w. Namely, $\#_k$ is now at position i and positions $i+1,\ldots,i+len_k-1$ are padded with $\bullet$'s. This ensures that the length of w_k remains n for each k-th step, and makes it easy for us to design our LZ-LFS algorithm.

If an occurrence of x_k at position i is replaced with $\#_k$, then the positions in range $[i+1..i+len_k-1]$ are taken away from the string. This range $[i+1..i+len_k-1]$ is therefore not considered in the following steps, and is called a *dead zone*. Also, since any LRs in the following steps are of length at most len_k, it suffices for us only to take care of the substrings in range $[i-len_k,..i]$. This range is called as an *affected zone*. See Figure 3 for illustration of a dead zone and affected zone.

In our suffix tree update algorithm, we will remove the leaves for the suffixes that begin in the dead zones, and modify the leaves for the suffixes that begin in the affected zones.

Let q_k denote the number of selected occurrences (Type 1, 2, or 3) of x_k in w_k to be replaced with $\#_k$. We will replace the selected occurrences of x_k from left to right. For each $1 \leq h \leq q_k$, let i_h denote the h-th selected occurrence of x_k from the left,

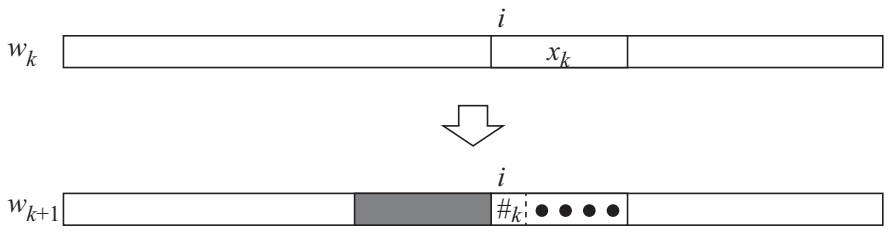


Figure 3. An occurrence of LR x_k at position i in the current string w_k is replaced with $\#_k$. In the next string w_{k+1}, the range padded with •'s is the dead zone and the gray range is the affected zone for this occurrence of x_k at position i.

and let w_k^h denote the string where the h occurrences $i_1, \ldots, i_h$ of x_k from the left are already replaced with $\#_k$'s. Namely, $w_k^0 = w_k$ and $w_k^{q_k} = w_{k+1}$.

Suppose that we have processed the $h-1$ occurrences of x_k from the left, and we are to process the h-th occurrence i_h of x_k. Namely, we have maintained $\mathsf{STree}(w_k^{h-1})$ and we are to update it to $\mathsf{STree}(w_k^h)$.

How to process the dead zones. First, we consider how to deal with the dead zone $[i_h + 1..i_h + len_k - 1]$ for this occurrence i_h of x_k in w_k^{h-1}. Since the positions in the dead zone will not exist in the modified string, and since no substrings beginning in this dead zone can be an LR in the following steps, we remove the leaves for the suffixes that begin at the positions in the dead zone $[i_h + 1..i_h + len_k - 1]$. In case $i_h + len_k - 1 > n$, which can happen only when $h = q_k$, then the dead zone for this occurrence is $[i_h + 1..n]$. In any case, we can easily remove those leaves in linear time in the number of the removed leaves.

How to process the affected zones. Next, we consider how to deal with the affected zone $[i_h - len_k..i_h]$ for this occurrence i_h of LR x_k in w_k^{h-1}. Let $y = w_k^{h-1}[i_h - len_k..i_h - 1]$, namely, y is the left context of length len_k from the occurrence of x_k at position i_h. Let y' be the longest non-empty suffix of y such that x_k down the locus of y' spans more than one edge in the tree. If such a node does not exist, then let $y' = \varepsilon$. For each suffix of y that is longer than y', x_k down its locus is represented on a single edge. Hence, it is "automatically" be replaced with $\#_k$ by replacing the occurrence of x_k at position i_h in the current string w_k^{h-1} with $\#_k \bullet^{len_k - 1}$. Therefore, no explicit maintenance on the tree topology is needed for these suffixes of y.

Now we consider the suffixes $y_j = y[j..len_k - 1]$ of y that are not longer than y', where $j = len_k - |y'| + 1, \ldots, len_k - 1$. Now x_k down the locus of each y_j spans more than one edge, and it will have to be replaced with a (single) special symbol $\#_k$. This introduces some changes in the tree topology. We note that the locus of $y_j x_k$ in the suffix tree before the update is on the edge that leads to the leaf with id $i_h - |y_j|$, since otherwise $y_j x_k$ must occur twice in the string, which contradicts our longest first strategy. Thus, we re-direct the edge that leads to the leaf with id $i_h - |y_j|$ from its original parent to the node that represents y_j (if it is an implicit node, then we create a new explicit node there). See Figure 4 for illustration.

The remaining problem is how to find the loci for the suffixes of y in the tree. We find them in decreasing order of their length. For the first suffix $y[1..len_k] = y$, we find the locus of y by simply traversing y from the root of the suffix tree. There are two cases to consider:

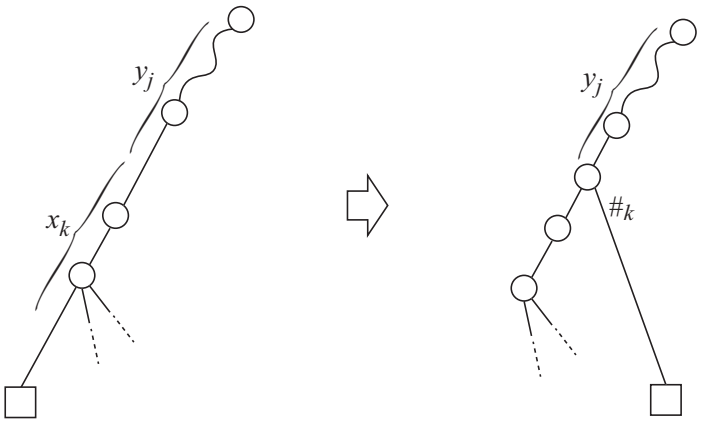


Figure 4. Illustration for a leaf edge redirection, where the circles represent internal explicit nodes and the square represents the leaf with id $i_h - |y_j|$. Since x_k down the locus of y_j spans more than one edge, the leaf edge is redirected from its original parent to y_j. This figure shows the case where a new internal node for y_j is created.

(A) If this locus for $y_1 = y$ is an explicit node in $\mathsf{STree}(w_k^{h-1})$, then by the property of the suffix tree, all suffixes of y are also represented by explicit nodes. Hence, we can find the loci for all the suffixes using a chain of suffix links from node y down to the root.

(B) If this locus for $y_1 = y$ is an implicit node in $\mathsf{STree}(w_k^{h-1})$, then we use the suffix link of the parent u_1 of y_1. Let $u'_2 = \mathsf{slink}(u_1)$. We go downward from u'_2 until finding the deepest node u_2 whose string depth is not greater than $|y_2| = len_k - 1$. If the string depth u_2 equals $|y_2|$ (i.e. $|u_2| = |y_2|$), then the locus of y_2 is on an explicit node. Hence, we can continue with y_3 as in Case (A) above. Otherwise (if $|u_2| < |y_2|$), then the locus of y_2 is on an out-going edge of u_2. We then continue with y_3 in the same way as for y_2.

Suppose we have processed all the q_k selected occurrences of x_k in w_k. The next lemma guarantees that re-direction of the leaf edges do not break the property of the suffix tree.

Lemma 5. *Let v be any non-root internal explicit node of the the tree obtained by updating $\mathsf{STree}(w_k^{h-1})$ as above. Then, the labels of the out-going edges of v begin with mutually distinct characters.*

Proof. Notice that in each k-th step, the label of any re-directed edge begins with $\#_k$. Since $\#_k \neq \#_j$ for any $1 \leq j < k$ and $\#_k$ does not occur in w_k, it suffices for us to show that there is at most one out-going edge of v whose label begins with $\#_k$.

If there are two out-going edges of v whose labels begin with $\#_k$, then there are at least two leaves whose path label begin with $v\#_k$. Thus $v\#_k$ occurs in w_k at least twice. Since v is not a root, $|v| \geq 1$. If x_k is the LR that was replaced by $\#_k$, then $|vx_k| > |x_k| = len_k$, which contradicts that x_k was an LR at the k-th step.

Thus, the labels of out-going edge of any node v begin with mutually distinct characters. □

The root of the resulting tree has a new child which represents $\#_k$, and the children of this new node are the leaves that correspond to the occurrences of the LR that have been replaced by $\#_k$.

Notice that the affected zone $[i_h - len_k..i_h - 1]$ for the occurrence i_h may overlap with the dead zone $[i_{h-1} + 1..i_{h-1} + len_k - 1]$ for the previous occurrence i_{h-1}. In this case, the affected zone for i_h is trimmed to $[i_{h-1} + len_k..i_h - 1]$ and we perform the same procedure as above for this trimmed affected zone.

Lemma 6. *Our algorithm updates* $\mathsf{STree}(w_k)$ *to* $\mathsf{STree}(w_{k+1})$ *for every* k*-th step in a total of* $O(n \log \sigma)$ *time with* $O(n)$ *space.*

Proof. First, let us confirm the correctness of our algorithm. It follows from Lemma 3 that in each k-th step the new internal explicit nodes that are created in this step can have string depth at most len_k. Therefore, in terms of updating $\mathsf{STree}(w_k)$ to $\mathsf{STree}(w_{k+1})$, it suffices for us to consider only the affected zone for each occurrence of LR x_k. Lemma 5 guarantees that the label of the out-going edges of the same node begin with mutually distinct characters. It is clear that the leaves for the suffixes which begin in the dead zones have to be removed, and only those leaves are removed. Thus, our algorithm correctly updates $\mathsf{STree}(w_k)$ to $\mathsf{STree}(w_{k+1})$.

Second, let us analyze the time complexity of our algorithm. For each occurrence i_h of x_k, finding the locus for the first suffix $y = w_k^{h-1}[i_h - len_k..i_h - 1]$ takes $O(len_k \log \sigma)$ time. Then, the worst case scenario is that Case (B) happens for all len_k suffixes of y. For each shorter suffix $y[i..len_k]$ with $i = 2, \ldots, len_k$, the above algorithm traverses at most $|u_j| - |u'_j| = |u_j| - |\mathsf{slink}(u_{j-1})| = |u_j| - |u_{j-1}| + 1$ edges. Hence, for all the shorter suffixes of y, the number of edges traversed is bounded by $\sum_{j=2}^{len_k}(|u_j| - |u_{j-1}| + 1) = |u_{len_k}| - |u_1| + len_k - 1 < 2len_k$. Hence, finding the locus for the shorter suffixes of y also takes $O(len_k \log \sigma)$ time. The len_k term in the $O(len_k \log \sigma)$ complexity can be charged to each selected occurrence of LR x_k, which is replaced with $\#_k \bullet^{len_k - 1}$. Therefore, the total time cost to update the suffix tree for all steps is $O(n \log \sigma)$. The space usage is clearly $O(n)$. □

4.3 How to update B_k to B_{k+1}

Suppose we have B_k in the k-th step, and we would like to update it to B_{k+1} for the next $(k+1)$-th step. Let u be an internal branching node of $\mathsf{STree}(w_{k-1})$ that is to be removed in $\mathsf{STree}(w_k)$. This can happen when u has only two children, one of which is a leaf to be removed from the current suffix tree. We then remove u from the list stored in $B_{k-1}[|u|]$, and connect its left and right neighbors in the list.

When we replace an LR x_k of w_k with $\#_k \bullet^{len_k - 1}$, an implicit node v of $\mathsf{STree}(w_k)$ may become branching due to the new symbol $\#_k$ and hence a new explicit internal node for v needs to be created to the suffix tree. In this case, we add this new node for v at the end of the list stored in $B_k[|v|]$. After these procedures are performed for all such nodes, we obtain B_{k+1} for the next $(k+1)$-th step.

Lemma 7. *At every* k*-th step, we can update* B_k *and maintain* len_k *in a total of* $O(n)$ *time and space.*

Proof. Initially, at most $n-1$ internal nodes are stored in B_1. Also, the total number of newly created nodes is bounded by the total size of the affected zones for the replaced occurrences of the LRs in all the steps, which can be charged to the positions that are taken by replacement of LRs for all the steps. As was shown in the previous subsection, once a position in the original string is taken by replacement of an LR, then this position will never be considered in the following steps. Thus, the total

number of newly created nodes is bounded by n. Clearly, computing the initial array B_1 from $\mathsf{STree}(w_1)$ takes $O(n)$ time, and deletion and insertion of a node on a list stored at an entry of B_k takes $O(1)$ time each (we use doubly linked lists here).

It follows from Lemma 3 and our suffix tree update algorithm that at each k-th step any newly created node has string depth at most len_k, and len_k is monotonically non-increasing as k grows. Hence, we can easily keep track of len_k for all steps in a total of $O(n)$ time.

The space usage is clearly $O(n)$. □

After computing w_m for the final m-th step, we replace every $\#_k$ in w_m with $\#$ for every k, and obtain the final string w' for LZ-LFS.

Summing up all the discussions above, we have proved our main result in Theorem 1.

5 $O(n)$-time algorithm for simplified LZ-LFS

In this section, we show that a simplified version of LZ-LFS can be computed in $O(n)$ time and space, by a slight modification to our $O(n \log n)$-time LZ-LFS algorithm from Section 4.

By a "simplified version" of LZ-LFS, we mean a variant of LZ-LFS where Type 3 non-overlapping occurrences of an LR of each step can be selected arbitrarily (namely, not necessarily in a left-greedy manner). More formally, in our simplified version of LZ-LFS, an occurrence i of x in w is said to be of Type 1/2 if the corresponding condition as in Section 3 holds, and

- Type 3 if i is an occurrence of x after e which is not of Type 2,

where e is as defined in Equation (1).

Notice that there can be multiple choices for non-overlapping Type 3 occurrences of LR x_k in w_k at each k-th step. Our algorithm takes a maximal set of non-overlapping Type 3 occurrences of x_k in w_k at each step, so that no Type 3 occurrences remain in the string. We remark that it is easy to compute a maximal set of size at least $\max\{\lceil |\mathsf{LGOcc}_{w_k[e+1..]}(x_k)|/2 \rceil, 2\}$, namely, this strategy allows us to select at least half the number of left-greedily selected Type 3 occurrences. Since this does not require to sort the occurrences of x_k, we can perform all the steps in a total of $O(n)$ time, as follows:

Theorem 8. *Given a string w of length n over an integer alphabet of size $n^{O(1)}$, our algorithm for a simplified version of LZ-LFS works in $O(n)$ time and space.*

Proof. As a preprocessing, we build $\mathsf{STree}(w)$ in $O(n)$ time and space [1].

We use essentially the same approach as in the previous section. Namely, we maintain the suffix tree for each step of our algorithm, and find Type 1, 2, and/or 3 occurrences of a selected LR using the suffix tree that we maintain.

Suppose that we are given a node v that represents an LR x_k in w_k at the k-th step. Since all children of v are leaves, we can easily compute the Type 1 occurrence of x_k (if it exists) by a simple scan over the children's leaf id's. After this, by another simple scan, we can also compute the Type 2 occurrence of x_k (if it exists). Then, we exclude the Type 1 and Type 2 occurrences, and any occurrences that overlap with the Type 1 and/or Type 2 occurrences, by removing the corresponding leaves which

are children of v. We then select a maximal set of non-overlapping Type 3 occurrences of x_k by picking up a child of v in an arbitrary order, and choosing it if it does not overlap with any already-selected occurrences.

Let d_k be the number of children of v. As in the standard LZ-LFS, each position of the original string can be involved in at most one event of the replacement of an LR. Hence, each step of the above algorithm takes $O(d_k)$ time, and thus the total time complexity for all the steps of this algorithm is $O(\sum_{k=1}^{m} d_k) = O(n)$, where m is the final step.

The space complexity is clearly $O(n)$. □

6 Conclusions and further work

LZ-LFS [5] is a new text compression method that has both features of Lempel-Ziv 77 [12] and grammar compression with longest first substitution [7].

In this paper, we proposed a suffix-tree based algorithm for LZ-LFS that runs in $O(n \log n)$ time and $O(n)$ space, where n denotes the length of the input string to compress. This improves on Mauer et al.'s suffix-array based algorithm that requires $O(n^2)$ time and $O(n)$ space. We also showed that a simplified version of LZ-LFS, where Type 3 occurrences may not be selected in a left-greedy manner, can be computed in $O(n)$ time and space with slight modifications to our LZ-LFS algorithm.

There are interesting open questions with LZ-LFS, including:

1. Does there exist a linear $O(n)$-time algorithm for (non-simplified) LZ-LFS? The difficulty here is to select Type 3 occurrences of each selected LR in a left-greedy manner. We remark that Nakamura et al.'s linear $O(n)$-time algorithm [7] for grammar compression with LFS does *not* always replace the left-greedy occurrences of each selected LR, either. Or, do there exist $\Omega(n \log n)$ lower bounds, probably by a reduction from sorting?
2. Does there exist a suffix-array based algorithm for LZ-LFS which works in time faster than $O(n^2)$? This kind of algorithm could be of practical significance.

References

1. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan: *On the sorting-complexity of suffix tree construction.* J. ACM, 47(6) 2000, pp. 987–1011.
2. J. Kärkkäinen and E. Ukkonen: *Sparse suffix trees*, in Proc. COCOON 1996, 1996, pp. 219–230.
3. N. J. Larsson and A. Moffat: *Offline dictionary-based compression*, in DCC 1999, 1999, pp. 296–305.
4. U. Manber and G. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM J. Computing, 22(5) 1993, pp. 935–948.
5. M. Mauer, T. Beller, and E. Ohlebusch: *A Lempel-Ziv-style compression method for repetitive texts*, in Proc. PSC 2017, 2017, pp. 96–107.
6. E. M. McCreight: *A space-economical suffix tree construction algorithm.* J. ACM, 23(2) 1976, pp. 262–272.
7. R. Nakamura, S. Inenaga, H. Bannai, T. Funamoto, M. Takeda, and A. Shinohara: *Linear-time off-line text compression by longest-first substitution.* Algorithms, 2(4) 2009, pp. 1429–1448.
8. J. Storer: *NP-completeness results concerning data compression*, Tech. Rep. 234, Department of Electrical Engineering and Computer Science, Princeton University, 1977.
9. J. Storer and T. Szymanski: *Data compression via textual substitution.* J. ACM, 29(4) 1982, pp. 928–951.

10. E. Ukkonen: *On-line construction of suffix trees.* Algorithmica, 14(3) 1995, pp. 249–260.
11. P. Weiner: *Linear pattern-matching algorithms*, in Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory, 1973, pp. 1–11.
12. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, IT-23(3) 1977, pp. 337–343.

Synchronizing Dynamic Huffman Codes

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, elina.opalinsk@ariel.ac.il

Abstract. Traditional dynamic Huffman 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, justifying the slow processing time. This, however, turns the encoded file into an extremely vulnerable one in the case of even a single bit error. Since the above mentioned assumption is not necessarily true, we explore blockwise dynamic Huffman variants, where the Huffman tree is periodically, rather than constantly, updated. Experiments show that avoiding the updates at every character and choosing larger blocks does not hurt the compression performance, and may even improve it at times. Moreover, the new scheme seems to be more robust against single errors introduced in the encoded file.

1 Introduction

One of the oldest, yet still popular, data compression techniques is Huffman coding [5]. The focus of the current research is on its dynamic version, where the code gets repeatedly updated while more characters of the input file are being processed. In particular, we consider the case that the compressed file has been transmitted over a network, and a communication error occurred within the encoded file, causing a change in one of its bits.

Given an input file which we shall call a text, even though the algorithm applies also to non-textual data, a first step is to decide how to parse the text into a set of *elements* to be encoded. Typically, these elements can be the characters of some standard alphabet Σ, like ASCII, but one could as well encode character pairs or triplets, or even words or phrases or word fragments, as long as there is a well-defined way to perform the parsing unambiguously.

This parsing can be used to derive the set of frequencies $\{w_1, \ldots, w_n\}$ of the n distinct elements of the text. Huffman's algorithm then assigns codeword lengths $\{\ell_1, \ldots, \ell_n\}$ to the corresponding elements of Σ, so that the average weighted length $\sum_{i=1}^{n} w_i \ell_i$ is minimized, yielding a minimum redundancy code. The code may be represented by a binary tree known as a *Huffman tree*, whose leaves are associated with the elements of the alphabet Σ. Edges in the tree pointing to a left or right child are labeled by 0 or 1, respectively, and the concatenation of the labels on the path from the root to a given leaf yields the corresponding codeword.

Static Huffman encoding thus requires a double pass over the data: the first for gathering the statistics on the distribution of the elements, on the basis of which the code can be built, and the second for the actual encoding process, once the code is given.

An alternative to this two-pass procedure could be an adaptive method in which both encoder and decoder maintain, independently, a copy of the current Huffman

Shmuel T. Klein, Elina Opalinsky, Dana Shapira: Synchronizing Dynamic Huffman Codes, pp. 27–37.
Proceedings of PSC 2018, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06484-9

tree, which is based on the frequencies of the elements processed so far. The trivial, and very costly, solution would be to reconstruct the Huffman tree from scratch in each iteration. Faller [1], Gallager [3] and Knuth [10] proposed, independently, essentially the same improved one-pass method, known as the FGK algorithm, for generating a Huffman tree according to the probabilities of the characters in the already processed portion of the file. The FGK algorithm was later enhanced by Vitter [15], who maintains the adaptive model and creates the encoding "on the fly".

One of the issues of transferring data over a network, is the ability to synchronize when communication errors occur. In [6], the problem of searching directly within a Huffman compressed file is investigated. Given a pattern P and a compressed file, instead of decompressing and searching for P in the decompressed file, the compression algorithm is applied to the pattern, and the resulting encoding is sought within the given encoded file. To announce a match of the pattern in the original file, the problem is to verify whether the detected occurrences are aligned on codeword boundaries. The algorithm proposed in [7] is based on the tendency of Huffman codes to resynchronize quickly after errors, even if decoding does not start at the beginning of a codeword [9].

Consider the case of a single bit error occurring during the transmission of the encoded file. In case the single error is a bit flip, and the file was encoded using some fixed length code, only a single codeword is affected. But in the case of a variable length code, like Huffman's, or if the error is the addition or deletion of a bit, synchronization with the correct decoding may also be lost. For example, let a part of the alphabet be $\{\ldots, \mathtt{M}, \mathtt{N}, \mathtt{O}, \mathtt{S}, \mathtt{U}, \ldots\}$ with codewords $\{\ldots, 0001, 1011, 010, 0010, 10010, \ldots\}$, respectively, and suppose the first bit of the encoding representing the string `MOON` has been lost. The correct decoding, is given in the lower part of Figure 1, whereas the erroneous decoding, `SUN`, is given in the upper part. Note that the codeword boundary just before `N`, in both decodings, is a synchronization point. We see that even a single bit error may have dramatic consequences, literally changing night into day...

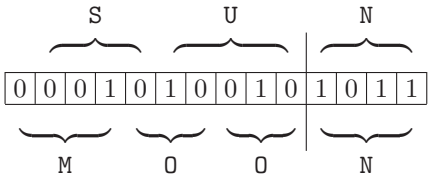


Figure 1. Synchronization in Static Huffman Encoded Files

Generally, suppose an error has occurred in one of the bits of the encoded file, which causes the decoding to be incorrect, and denote by c_1 the last codeword of the resulting wrong decoding before the synchronization point. Let c_2 be the last codeword of the correct decoding before the synchronization point. Then either c_1 must be a proper suffix of c_2 or vice versa. A code is said to have the *affix* property, if none of its codewords is either a prefix or a suffix of any other codeword. Therefore, the scenario described above about the codewords c_1 and c_2 ending both at a synchronization point is not possible if the code is affix, which is why such codes are also called *never-self-synchronizing* in [4]. But such affix codes are extremely rare [2].

Previous research [6] empirically shows that synchronization is regained after typically a few tens of bits. However, the situation is much more involved when dynamic Huffman coding is considered, due to the adaptive nature of the underlying model.

This paper modifies the dynamic Huffman coding procedure, so that the resulting algorithm is more robust against errors when compared to the original dynamic encoding. In fact, we generalize the dynamic Huffman compression of Vitter so that the resulting encoding can mostly synchronize with the correct decoding after only a few codewords. The paper is structured as follows. Section 2 presents the difficulty of the dynamic Huffman algorithm of Vitter [15] to synchronize after a single bit error. Section 3 proposes a generalized dynamic version which is more robust. Section 4 presents experimental results, and Section 5 concludes.

2 Synchronization in Dynamic Huffman Encoding

Dynamic Huffman encoding is highly vulnerable to occurrences of bit errors. In fact, even a single incorrect bit might change the dynamic Huffman tree in a way that will damage the remaining decoding completely.

Figure 2 visually presents an example of the effect of a bit flip introduced in the dynamic Huffman encoded file. The left side of Figure 2 is the decoding of a correctly encoded file (an excerpt from *Alice in Wonderland*), and the right side of Figure 2 is the decoding of the same file in which a *single* bit has been flipped. Characters that were decoded correctly appear in the same font as their correct counterparts, whereas incorrectly decoded characters are colored in red and **boldfaced**. As can be seen in this example, an error may trigger a snowball effect.

The single bit flip causes the character **x** to be mistakenly decoded as **F**, but then synchronization is seemingly regained, as for static Huffman coding. However, at this stage, the frequency counts of the two decodings already differ for certain characters, albeit only slightly. This small difference will trigger, 38 decoded characters later, another erroneous decoding, in which **xt_thing_w** will be substituted by **pnloci_lr**, where we use the underscore _ to visualize a blank. There are thus even more different updates, and the following wrong decoding is **plained_that_they_could_not_tas** which is replaced by **wh_iothn_o-n_oWnTay_ndtnlnha**. Clearly, the initially small changes have a cumulative impact, which materializes as gradually shorter correct stretches separating increasingly longer wrong ones, until the decoding becomes completely erroneous. In several other examples we tested, the situation is even worse, and the divergence is immediate rather than gradually as in Figure 2.

The following small example illustrates the difficulty of Vitter's algorithm to cope with errors. Consider the bit stream $\cdots 101111 \cdots$. The dynamically changing Huffman tree after having read these bits is presented in Figure 3. A dynamic Huffman tree is represented with a pair (*freq*, *char*) in its leaves, where *char* is the character represented by the leaf and *freq* is its frequency in the file so far. Each internal node contains the sum of the frequencies stored in its two children. A special leaf, labeled NYT, is used when a new, Not-Yet-Transmitted, symbol is detected, and its frequency is defined as zero.

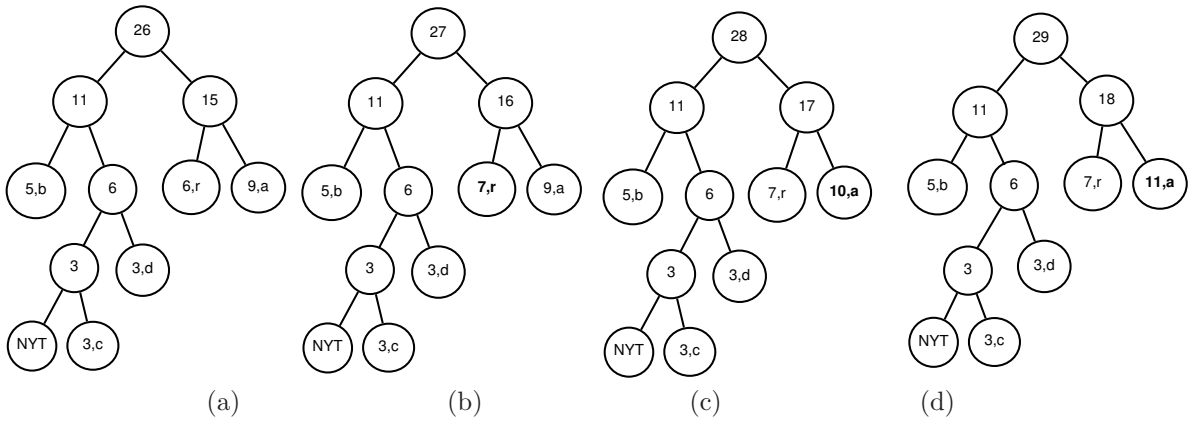
Suppose that at the beginning of the decoding process of these bits, the tree is the one given in Figure 3(a). Processing the first codeword 10, causes an increment of the number of occurrences of r from 6 to 7, as shown in Figure 3(b). The following bits are parsed into two consecutive codewords, 11, incrementing the number of the occurrences of a from 9 to 10 in Figure 3(c) and finally from 10 to 11, given in Figure 3(d).

Suppose that the encoded file has been corrupted and the second of the shown bits was flipped, so that the entire bit stream consists of 1s. The erroneous decoding

There was exactly one a-piece, all 'round.
The next thing was to eat the comfits; this caused some noise and confusion, as the large birds complained that they could not taste theirs, and the small ones choked and had to be patted on the back. However, it was over at last and they sat down again in a ring and begged the Mouse to tell them something more.
"You promised to tell me your history, you know," said Alice, "and why it is you hate–C and D," she added in a whisper, half afraid that it would be offended again.
"Mine is a long and a sad tale!" said the Mouse, turning to Alice and sighing.
"It_is_ a long tail, certainly," said Alice, looking down with wonder at the Mouse's tail, "but why do you call it sad?" And she kept on puzzling about it while the Mouse was speaking, so that her idea of the tale was something like this:–

There was e**F**actly one a-piece, all 'round.
The ne**Pnloci lr**as to eat the comfits; this caused some noise and confusion, as the large birds com**wh iothn o-n oWnTay ndtnlnha**te theirs, and the small ones choked and had to be patted on the back. **Ot"**ever, it was over at last and they sat down again in a ring and begged the **fmnrihdo** tell them something more.
"Wdnr v omised to tell me your history **uinrwan-heqe hint** Alice-**ks**nd why it is you hate**N ohe** and **Rkw** she added in a **pccwen cssd, a,rail** that it **baDh rfthfgnojs**gain**;e " o,othac a dv dita a sao laleAk hioa** the **o,** use-**l**urning **tll oose ano ecighingOe:e RNiisAhs d. lnint-Nn niorodkp aaio u tice- o lewi .n mr eane r.ton Urthe Aa Eov- nint'k e vrbad or aa jaadd** it sa**okcflttlea hwketr.genvhn ding aeY, it bcys olh h-ha rae frciSi nawl o,l hsdsi,a ve th etals se m msoci dtifc oceng"le eeed Sfu. y esil t,e a mdyhac j hoed**

Figure 2. Bit Flip in Dynamic Huffman Encoded Texts



(a) (b) (c) (d)

Figure 3. Correct decoding

is given in Figure 4. The parsing is now 11-11-11, thus incrementing the number of occurrences of a from 9 to 10, then from 10 to 11, and finally from 11 to 12, presented in Figures 4(a), (b) and (c), respectively. Before the number of occurrences of a changes to 12, the shape of the Huffman tree gets updated, and its underlying structure changes: before incrementing the contents of the leaf y with frequency 11, and that of all its ancestors, Vitter's algorithm calls for interchanging this leaf with the highest ranked node also containing the frequency 11, if there is one. The rank of a node is its index in a bottom-up, and within each level left-to-right, enumeration of all the nodes. In our case, the left child of the root has also frequency 11, so the left subtree and the leaf y are swapped, yielding the tree in Figure 4(d). Obviously, the following bits are completely out of synchronization.

Although the erroneous bit belongs to a single codeword, it directly affects the frequencies of *two* different codewords to start with, unlike for static Huffman coding, and might also trigger two snowball effects, so that the entire decoding may be

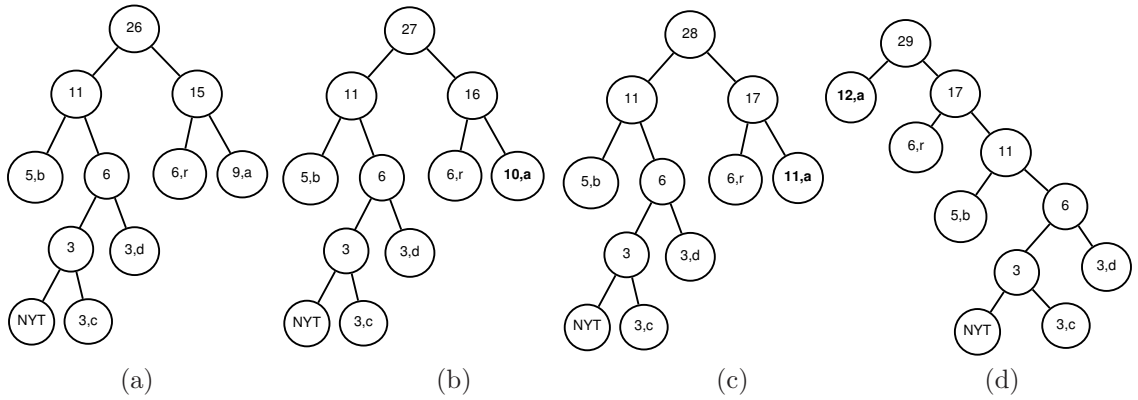


Figure 4. Incorrect decoding

damaged even faster. Going back to our example, the single error flipping 10 into 11 did not only reduce the number of occurrences of the symbol `r` corresponding to 10, but also increased the number of appearances of the codeword 11, corresponding to `a`. Since even for static Huffman coding a few codewords might be lost until synchronization is regained, each such mistaken codeword boundary again might cause an incorrect calculation of the frequencies of two codewords. Empirically checking this phenomenon on our datasets, we found that the structure of the Huffman tree is often changed before synchronization is attained, causing a complete loss of the remaining decoding. To overcome the problem we propose a relaxed variant of the dynamic Huffman compression that has the ability to cope with such errors, and generally synchronizes with the correct decoding after only a few codewords.

3 Proposed Algorithm

The main idea of our algorithm is to blur the frequencies with the objective of turning them to less sensitive to isolated errors. This is done on the one hand, by spacing out the updates to be done at the end of a *block* of several characters, and on the other hand by periodically rescaling the accumulated frequencies, for example, dividing each of them by 2. To avoid zero-frequencies, which would force a special treatment for disappearing and reappearing characters, one could initialize all frequencies with 1 and use upward rounding in the scaling, so that no frequency will fall below 1.

The rationale is the following. Spacing out the updates creates blocks within which the algorithm behaves essentially like static Huffman coding. There is thus a chance, if the error is not too close to a block boundary, that synchronization is regained even before it has changed the shape of the tree. As to scaling, if at a certain point, the distribution of the frequencies is $\{w_1, w_2, \ldots, w_n\}$, then considering $\{\lceil w_1/2 \rceil, \lceil w_2/2 \rceil, \ldots, \lceil w_n/2 \rceil\}$ instead will generally produce practically the same Huffman tree. Indeed, the shape of a Huffman tree is mainly determined by the *relative* sizes of the frequencies, rather than by their absolute values, which is why probabilities can be used instead of frequencies. Rescaling can thus achieve a triple goal:

1. it helps keeping the involved frequencies within bounded limits;
2. it gives higher weights to recently seen characters, as the last frequencies are divided by 2, but those accumulated in the block before are divided by 4, and,

generally, the number of occurrences of any character in block i when counted backwards from the last one, is divided by 2^i;

3. small fluctuations of ± 1 in the frequencies may either be corrected by the rounding, but even if not, there are good chances that the resulting Huffman trees are identical. If so, the error will not propagate and its effect may be corrected at the next synchronization point.

The fact that different, yet similar, frequency distributions may yield the same Huffman tree has been investigated by Longo and Galasso [12]: the set of probability distributions over a finite alphabet is given a "pseudometric", and an upper bound is derived for the distance from any probability distribution to the *dyadic* distribution (in which all the probabilities are powers of $\frac{1}{2}$) giving the same Huffman tree.

On the other hand, the disadvantage of rescaling seems to be that the construction of the Huffman tree will then not rely on true frequencies, but on approximate ones, which, would one think, might hurt the optimality of the Huffman procedure. It should however be kept in mind that Huffman codes are optimal only for static frequencies. In the adaptive variant, actually not only for Huffman, but also for arithmetic coding and for intrinsically adaptive methods like those of Ziv and Lempel [16,17], the basic assumption is that the distribution of elements in the text seen so far (the past) is identical, or at least a good estimate for, the distribution after the current point (the future), but there is no guarantee for such an assumption to be true!

The orthodox adherence to the exact frequencies is thus not necessarily justifiable, and it could well be that an approximation can produce results that are not inferior, and maybe even better at times. A similar observation has been mentioned in [8] in an application basing adaptive arithmetic coding on randomly chosen n out of $2n$ preceding characters, rather than the most recent n, without incurring any noticeable loss.

Once it has been agreed that our knowledge of the frequencies of the characters in the processed text does not need to be perfect, this reinforces the idea that we may change the constant updates after each read character, as advocated, e.g., in Vitter's algorithm, by periodic ones, to be performed only at the end of each block of b characters, for some fixed block size b. This has the obvious advantage of speeding up both compression and decompression, and our empirical results show that the expected loss is very low, much less than 1% on all our tests, even with large blocks like $b = 16K$. There were even instances in which the blockwise processing gave better compression than Vitter's variant, which corresponds to $b = 1$.

A first guess would be that the larger the blocksize b will be chosen, the less accurate our estimate will be, which is consistent with our assumption of dealing with a tradeoff: since a larger block implies obvious time savings, it is reasonable to assume that this gain in time comes at the price of a certain loss in compression efficiency. Our results, however, show that often, just the opposite is true! While for large blocks, the increase, if there was one, in the size of the compressed file was very small, it was for the small blocksizes that a significant loss occurred, increasing the file size by tens of percents. For example, for the two test files mentioned in the next section, the file size grew, for a block of size $b = 16$, by 22.4% and 44.8%, while for $b = 1K$, the increase was only by 0.2% and 2.2%, respectively.

To understand this behavior, recall that we start the dynamic encoding by assuming a uniform distribution of the characters, a quite arbitrary initialization which does not really matter if the blocksize is large enough. Recall also that all frequen-

cies are rescaled at the end of each block, so that the influence of the distribution of characters which are several blocks behind the current point is quickly vanishing. If the blocksize itself is small, only a part of the alphabet will appear in these few preceding blocks, and the frequencies on which the current encoding will be based will still include many elements with the initial frequency 1. For large enough blocks, such low frequency elements will practically have no influence, but for small blocks, they might be dominant, yielding an overall distribution which is still close to uniform, and therefore far from the real distribution within the input text.

Algorithm 1: Compression Algorithm

GENERALIZED DYNAMIC HUFF(T, b)
1 $HT \leftarrow$ Huffman Tree for uniform distribution of Σ
2 **while** input T is not exhausted **do**
3 **encode** the following b characters according to HT
4 **update** frequencies of Σ according to the last b characters
5 **divide** all frequencies by 2, rounding up
6 **update** HT according to updated frequencies

The formal algorithm, with parameters T, the input text, and b, the block size measured in number of characters, is given below. The tree reconstructed in the decompression phase maintains the same distribution as in the compression phase, and by agreeing to construct canonical trees [13] and keeping the symbols at each level in some agreed order, e.g., lexicographically, encoder and decoder are able to reconstruct the same tree.

4 Experimental Results

We applied our experiments on two text files:

1. the Bible (King James version) in English, denoted by *ebib*, over an alphabet of 52 characters, in which the text has been stripped of all punctuation signs except blank;
2. the French version of the European Union's JOC corpus, denoted by *ftxt*, which is a collection of pairs of questions and answers on various topics used in the ARCADE evaluation project [14], over an alphabet of 127 characters.

Table 1 presents the compression performance of the various algorithms on both data files. The second, third and fourth columns present the original file size, the size of the compressed file, using static Huffman, and the file size after applying Vitter's algorithm. The last columns give the sizes of the generalized dynamic algorithm for block sizes 2^iK, $0 \leq i \leq 4$, and the results when also division is applied is given in the line below. All figures are given in Bytes. The best compression results for each file are highlighted in bold. As can be seen, the efficiency of the blockwise compression algorithm is not inferior, on these examples, to the original one updating after each processed character, in both variants, with and without division. As fewer Huffman trees are constructed throughout the execution of the algorithms when larger blocks are considered, the savings in processing times are also obvious.

In order to assess the synchronization abilities of the different algorithms, several measures should be considered. The problem lies in the fact that the damage caused

File	**Original**	**Static**	**Vitter**	**Generalized**				
				K	2K	4K	8K	16K
ebib	3711020	1942474	1942008	1942247	1942628	1943383	1944885	1947814
				Generalized & Division				
				1945709	1942012	**1941187**	1941847	1948863
ftxt	7610765	4399422	4379161	**4378724**	4379065	4379748	4381080	4383696
				Generalized & Division				
				4476947	4429719	4411830	4406138	4406566

Table 1. Compression performance - ebib

by a single erroneous bit during the decoding process may be judged on different levels, each of which requires another definition. Indeed, assume a single such error has occurred, then at least one, and possibly several, codewords will be falsely interpreted. However,

1. the fact that there were wrong interpretations means that some frequencies will be incorrect, but this does not necessarily mean that the corresponding Huffman trees have changed; the damage might thus be locally restricted and have no long range impact.
2. Even if the changes in the frequencies are extended enough to trigger also a change in the shape of the trees, the set of codeword lengths, and at times, even the set of codewords themselves, may still remain unchanged, so that one *could* use other Huffman trees which are still identical. For instance, Vitter's algorithm constructs a very specific form of the Huffman tree, and a small perturbation in the frequencies may change its shape altogether, as in the example in Figure 4(d), but had canonical Huffman trees been used instead of Vitter's, these small alterations might have a lesser or no impact at all, and the original and altered frequencies might yield the same canonical tree.
3. Finally, the frequency fluctuations may be significant enough to imply even different canonical Huffman trees.

To deal with the first option, we need a measure D_1 evaluating some "distance" between the erroneous frequency distribution caused by the bit flip, and the correct distribution, according to which the file has been encoded. A well-known such measure is the *Kullback-Leibler* (KL) divergence [11]: for two probability distributions $E = \{e_1, \ldots, e_n\}$, which is possibly erroneous, and $T = \{t_1, \ldots, t_n\}$, which we assume to be the true one, define

$$D_1(E, T) = D_{KL}(E \| T) = \sum_{i=1}^{n} e_i \log \frac{e_i}{t_i} .$$

The KL divergence is a one-sided, asymmetric, non-negative distance from E to T, which equals zero if and only if $E = T$.

However, D_1 is not an appropriate measure for our application. First, it depends on the location of where the error has occurred. If this happened close to the beginning

of the file, the impact of a change of ± 1 on the yet small accumulated frequencies will be larger than if the error had occurred significantly later. Moreover, the involved probabilities are all very small, their ratios are close to 1 and overall, on all our tests, the values of the KL divergence were of the order of 10^{-5} to 10^{-10}, from which hardly any conclusion could be derived.

This suggests using a more descriptive measure for the distance between distributions, based on the absolute difference between corresponding frequencies, rather than on the relative one implied by the probabilities. If the erroneous and true frequency vectors are denoted $EF = \{ef_1, \ldots, ef_n\}$ and $TF = \{tf_1, \ldots, tf_n\}$, respectively, we define

$$D_2(EF, TF) = \sum_{i=1}^{n} |ef_i - tf_i|.$$

Table 2 presents the results of comparing the D_2 distances between the distributions produced by erroneous and true decodings by the three algorithms considered herein: Vitter's dynamic Huffman coding, with updates after each processed character, the blockwise dynamic algorithm using cumulative frequencies and blocksize $b = 1024$ encoded characters, and the same but using rescaling by dividing the frequencies by 2 after each block. The table gives the numbers for a typical example, presenting in the column headed i the distance D_2 as measured at the end of the ith block after the error. The test file was *ebib*, in which the first bit of codeword number 380245 was flipped.

File	1	2	3	4	5	6	7	8	9	10	20	30	40	50
Vitter	10	12	12	12	10	12	38	60	60	60	210	2344	4000	6114
blocks – cumulative	8	8	8	8	8	8	8	8	8	8	8	8	8	8
blocks – scaled	5	4	3	2	2	1	0	0	0	0	0	0	0	0

Table 2. Comparing the distance $D_2(EF, TF)$ between erroneous and true decoding

We see that for Vitter's algorithm, the sum of the absolute differences is about 10–12 at the beginning, but then increases exponentially yielding the snowball effect mentioned above. A large distance means that the distributions are completely different, in accordance with the example in Figure 2. On the other hand, the distance for the algorithm processing blocks stays constant at 8 for all the considered blocks, and even for hundreds thereafter. The frequencies, though, do increase gradually, just their difference remains constant, which indicates that synchronization has been regained. For the block variant with scaling, not only is there synchronization, but the difference also is zeroed by the repeated divisions.

As a different number of occurrences of the characters does not necessarily refer to an incorrect decoding, our final experiment is to measure the percentage of successful decodings for the various algorithms. We repeated the bit-flip test as the one reported in Table 2 one hundred times, with different flip positions, and checked not only the distance, but also whether there was ultimately synchronization after the error or not. To avoid a bias in the choice of the bit position of the error, the 2000th bit of 100 different blocks of b characters has been flipped, with $b \in \{1K, 2K, 4K, 8K\}$.

Table 3 presents the number of unsynchronized decodings as a function of block size for all three algorithms. Synchronization has been assessed in this experiment by

Block Size	**1K**	**2K**	**4K**	**8K**
Vitter	100	100	100	100
blocks – cumulative	23	17	17	16
blocks – scaled	22	15	13	12
intersection	4	2	3	7

Table 3. Number of unsynchronized decodings out of 100 tests as a function of block sizes.

comparing the last blocks at the end of the decoded files and checking that they are identical. The number appearing on the intersection line reports the number of cases that are unsynchronized in both blockwise algorithms.

In none of our experiments did Vitter's decoding synchronize with the correct decoding, whereas the block variants did get back on track in about 80% of the cases or more. This figure seems to be improving with growing block sizes. There seems also to be a small improvement of the variant using scaling over that using cumulative frequencies, though the full extent of the improvement might not be caught by our measure: the major advantage of periodically dividing the numbers is that erroneous fluctuations are ultimately "forgotten". This amnesic behavior allows the decoding to synchronize not only the text sent to the output file, but also the Huffman data structure used to enable the decoding. In the cumulative variant, the texts might match even for long stretches, but there is no guarantee that the differing Huffman trees will not, at some later stage, induce errors again.

5 Conclusion

Motivated by the lack of synchronization in the case of the occurrence of even a single bit error in a file that has been compressed by a standard dynamic Huffman code, we explored generalizations that get updated periodically, not necessarily after each character. Two blockwise dynamic variants were considered, with and without scaling, suggesting an improvement in processing time and robustness against single bit errors, without hurting the compression performance.

A synchronization point of a correct and incorrect decoding of static Huffman coding may be defined as the first position after that of the error in the encoded file, for which both decodings reach the root of the tree at the same time. However, locating the synchronization point in the dynamic variants was found to be a bit tricky, as two different Huffman trees should be compared in parallel. Moreover, small fluctuations in the frequencies may trigger later divergences of the decoding, even if temporarily synchronization has been restored. We therefore approximated the distances between both decodings by summing the frequency differences.

References

1. N. Faller: *An adaptive system for data compression*, in Record of the 7-th Asilomar Conference on Circuits, Systems and Computers, 1973, pp. 593–597.
2. A. Fraenkel and S. Klein: *Bidirectional Huffman coding.* The Computer Journal, 33(4) 1990, pp. 296–307.
3. R. Gallager: *Variations on a theme by Huffman.* IEEE Transactions on Information Theory, 24(6) 1978, pp. 668–674.
4. E. Gilbert and E. Moore: *Variable-length binary encodings.* The Bell System Technical Journal, 38, pp. 933–968.
5. D. Huffman: *A method for the construction of minimum redundancy codes.* Proc. of the IRE, 40 1952, pp. 1098–1101.
6. S. Klein and D. Shapira: *Pattern matching in Huffman encoded texts.* Inf. Process. Manage., 41(4) 2005, pp. 829–841.
7. S. Klein and D. Shapira: *Compressed pattern matching in JPEG images.* Int. J. Found. Comput. Sci., 17(6) 2006, pp. 1297–1306.
8. S. 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.
9. S. Klein and Y. Wiseman: *Parallel Huffman decoding with applications to JPEG files.* The Computer Journal, 46(5) 2003, pp. 487–497.
10. D. Knuth: *Dynamic Huffman coding.* Journal of Algorithms, 6(2) 1985, pp. 163–180.
11. S. Kullback and R. Leibler: *On information and sufficiency.* Annals of Mathematical Statistics, 22(1) 1951, pp. 79–86.
12. G. Longo and G. Galasso: *An application of informational divergence to Huffman codes.* IEEE Trans. Information Theory, 28(1) 1982, pp. 36–42.
13. E. Schwartz and B. Kallick: *Generating a canonical prefix encoding.* Commun. ACM, 7(3) 1964, pp. 166–169.
14. J. Véronis and P. Langlais: *Evaluation of parallel text alignment systems: The arcade project, in parallel text processing.* pp. 369–388.
15. J. Vitter: *Design and analysis of dynamic Huffman codes.* Journal of the ACM (JACM), 34(4) 1987, pp. 825–845.
16. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Trans. Information Theory, 23(3) 1977, pp. 337–343.
17. J. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Trans. Information Theory, 24(5) 1978, pp. 530–536.

A Faster V-order String Comparison Algorithm

Ali Alatabbi[1], Jacqueline W. Daykin[1,2,3], Neerja Mhaskar[4], M. Sohel Rahman[5], and William F. Smyth[1,4,6]

[1] Department of Informatics, King's College London, UK
ali.alatabbi@kcl.ac.uk, jackie.daykin@kcl.ac.uk
[2] Department of Computer Science
Aberystwyth University, Wales & Mauritius
[3] Department of Information Science
Stellenbosch University, South Africa
[4] Algorithms Research Group, Department of Computing & Software
McMaster University, Canada
pophlin@mcmaster.ca, smyth@mcmaster.ca
[5] Department of Computer Science & Engineering
Bangladesh University of Engineering & Science
msrahman@cse.buet.ac.bd
[6] School of Engineering & Information Technology
Murdoch University, Western Australia

Abstract. V-order is a total order on strings that determines an instance of Unique Maximal Factorization Families (UMFFs) [7–10], a generalization of Lyndon words [12]. V-order has also recently been proposed as an alternative to lexicographic order (lex-order) in the computation of suffix arrays and in the suffix-sorting induced by the Burrows-Wheeler Transform (BWT) [11]. The central problem of efficient V-ordering of strings was considered in [2–4, 9, 10], culminating in a remarkably simple, linear time, constant space comparison algorithm [1]. In this paper we improve on this result to achieve significant speed-up in almost all cases of interest.

Keywords: combinatorics, experiments, lexorder, linear, on-line algorithm, probability, string comparison, V-comparison, V-order

1 Introduction

This paper extends current knowledge on the non-lexicographic ordering technique known as V-order [6]. New combinatorial insights are obtained which are linked to computational settings. The first of these leads to an improvement on the linear-time V-order string comparison algorithm described in [1].

2 Preliminaries

We are given a finite totally ordered set of cardinality $\sigma = |\Sigma|$, called the *alphabet*, whose elements are *characters* (equivalently *letters*). A *string* is a sequence of zero or more characters over Σ. A string $\boldsymbol{x} = x_1 x_2 \cdots x_n$ of *length* $|\boldsymbol{x}| = n$ is represented by $\boldsymbol{x}[1..n]$, where $\boldsymbol{x}[i] \in \Sigma$ for $1 \leq i \leq n$. The set of all non-empty strings over the alphabet Σ is denoted by Σ^+. The *empty string* of zero length is denoted by ε, with $\Sigma^* = \Sigma^+ \cup \varepsilon$. If $\boldsymbol{x} = \boldsymbol{uwv}$ for strings $\boldsymbol{u}, \boldsymbol{w}, \boldsymbol{v} \in \Sigma^*$, then $\boldsymbol{u}$ is a *prefix*, $\boldsymbol{w}$ is a *substring* or *factor*, and $\boldsymbol{v}$ is a *suffix* of $\boldsymbol{x}$. We denote by $\boldsymbol{s}[i \ldots j]$, or $s_i \cdots s_j$, the substring of $\boldsymbol{s}$ that starts at position i and ends at position j. Notably, if $i > j$, $s[i \ldots j] = \varepsilon$; that is, $s[i \ldots j]$ is the empty string. If $\boldsymbol{x} = \boldsymbol{u}^k$ (a concatenation of k

Ali Alatabbi, Jacqueline W. Daykin, Neerja Mhaskar, M. Sohel Rahman, William F. Smyth: A Faster V-order String Comparison Algorithm, pp. 38–49.
Proceedings of PSC 2018, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06484-9

copies of $\boldsymbol{u}$) for some nonempty string $\boldsymbol{u}$ and some integer $k > 1$, then $\boldsymbol{x}$ is said to be a *repetition*; otherwise, $\boldsymbol{x}$ is *primitive*. For further stringological definitions, theory and algorithmics see [5, 13].

We define an order that is not lexorder (dictionary order), called V-order, as well as some of its notable properties, both combinatorial and algorithmic. V-order was explored in [6] and subsequently studied extensively in the literature both combinatorially and algorithmically [1–4, 7, 9–11].

Let $\boldsymbol{x} = x_1x_2\cdots x_n$ be a string over Σ. Define $h \in \{1,\ldots,n\}$ by $h = 1$ if $x_1 \le x_2 \le \cdots \le x_n$; otherwise, by the unique value such that $x_{h-1} > x_h \le x_{h+1} \le x_{h+2} \le \cdots \le x_n$. Let $\boldsymbol{x}^* = x_1x_2\cdots x_{h-1}x_{h+1}\cdots x_n$, where the star * indicates deletion of the letter x_h. We write $\boldsymbol{x}^{s*}$ for $(\ldots(\boldsymbol{x}^*)^*\ldots)^*$ with $s \ge 0$ stars. Let $g = \max\{x_1, x_2, \ldots, x_n\}$, and let k be the number of occurrences of g in $\boldsymbol{x}$. Then the sequence $\boldsymbol{x}, \boldsymbol{x}^*, \boldsymbol{x}^{2*}, \ldots$ ends in $g^k, \ldots, g^1, g^0 = \boldsymbol{\varepsilon}$. From all strings $\boldsymbol{x}$ over Σ, we use this process to form the *star tree*, where each string $\boldsymbol{x}$ labels a vertex, and there is a directed edge upward from $\boldsymbol{x}$ to $\boldsymbol{x}^*$, with the empty string $\boldsymbol{\varepsilon}$ as the root.

Definition 1. *[6, 7, 9, 10] We define* V-order $\prec$ *between distinct strings* $\boldsymbol{x}$, $\boldsymbol{y}$. *First* $\boldsymbol{x} \prec \boldsymbol{y}$ *if in the star tree* $\boldsymbol{x}$ *is in the path* $\boldsymbol{y}, \boldsymbol{y}^*, \boldsymbol{y}^{2*}, \ldots, \boldsymbol{\varepsilon}$. *If not, then there exist smallest* s, t *such that* $\boldsymbol{x}^{(s+1)*} = \boldsymbol{y}^{(t+1)*}$. *Let* $\boldsymbol{s} = \boldsymbol{x}^{s*}$ *and* $\boldsymbol{t} = \boldsymbol{y}^{t*}$; *then* $\boldsymbol{s} \ne \boldsymbol{t}$ *but* $|\boldsymbol{s}| = |\boldsymbol{t}| = m$ *say. Let* $j \in 1..m$ *be the greatest integer such that* $\boldsymbol{s}[j] \ne \boldsymbol{t}[j]$. *If* $\boldsymbol{s}[j] < \boldsymbol{t}[j]$ *in* Σ *then* $\boldsymbol{x} \prec \boldsymbol{y}$; *otherwise,* $\boldsymbol{y} \prec \boldsymbol{x}$. *Clearly* $\prec$ *is a total order on all strings in* Σ^*.

For instance, using the natural ordering of integers, if $\boldsymbol{x} = 2631$, then $\boldsymbol{x}^* = 263$, $\boldsymbol{x}^{2*} = 26$, $\boldsymbol{x}^{3*} = 6$, $\boldsymbol{x}^{4*} = \boldsymbol{\varepsilon}$, and so $26 \prec 2631$ while $2631 \prec 94$.

Definition 2. *[6, 7, 9, 10] The* V***-form*** *of a string* $\boldsymbol{x}$ *is defined as*

$$V_k(\boldsymbol{x}) = \boldsymbol{x} = \boldsymbol{x_0}g\boldsymbol{x_1}g\cdots\boldsymbol{x_{k-1}}g\boldsymbol{x_k}$$

for (possibly empty) strings $\boldsymbol{x_i}$, $i = 0, 1, \ldots, k$, *where* g *is the largest letter in* $\boldsymbol{x}$ — *thus we suppose that* g *occurs exactly* k *times. For clarity, when more than one string is involved, we use the notation* $g = \mathcal{L}_{\boldsymbol{x}}$, $k = \mathcal{C}_{\boldsymbol{x}}$.

Lemma 3. *[6, 7, 9, 10] Suppose we are given distinct strings* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *with corresponding* V*-forms as follows:*

$$\boldsymbol{x} = \boldsymbol{x}_0\mathcal{L}_{\boldsymbol{x}}\boldsymbol{x}_1\mathcal{L}_{\boldsymbol{x}}\boldsymbol{x}_2\cdots\boldsymbol{x}_{j-1}\mathcal{L}_{\boldsymbol{x}}\boldsymbol{x}_j,$$
$$\boldsymbol{y} = \boldsymbol{y}_0\mathcal{L}_{\boldsymbol{y}}\boldsymbol{y}_1\mathcal{L}_{\boldsymbol{y}}\boldsymbol{y}_2\cdots\boldsymbol{y}_{k-1}\mathcal{L}_{\boldsymbol{y}}\boldsymbol{y}_k,$$

where $j = \mathcal{C}_{\boldsymbol{x}}$, $k = \mathcal{C}_{\boldsymbol{y}}$.

Let $h \in 0..\max(j, k)$ *be the least integer such that* $\boldsymbol{x}_h \ne \boldsymbol{y}_h$. *Then* $\boldsymbol{x} \prec \boldsymbol{y}$ *if, and only if, one of the following conditions hold:*

(C1) $\mathcal{L}_{\boldsymbol{x}} < \mathcal{L}_{\boldsymbol{y}}$
(C2) $\mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$ *and* $\mathcal{C}_{\boldsymbol{x}} < \mathcal{C}_{\boldsymbol{y}}$
(C3) $\mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$, $\mathcal{C}_{\boldsymbol{x}} = \mathcal{C}_{\boldsymbol{y}}$ *and* $\boldsymbol{x}_h \prec \boldsymbol{y}_h$.

Observe the recursive nature of determining $\prec$ in (C3); that is, each substring pair $\boldsymbol{x}_h$, $\boldsymbol{y}_h$ can likewise be decomposed into V-forms.

Lemma 4. *[9, 10] For given strings $\boldsymbol{x}$ and $\boldsymbol{v}$, if $\boldsymbol{v}$ is a proper subsequence of $\boldsymbol{x}$, then $\boldsymbol{v} \prec \boldsymbol{x}$.*

Theorem 5. *[1, 4] For any strings $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{x}$, $\boldsymbol{y}$: $\boldsymbol{x} \prec \boldsymbol{y} \Leftrightarrow \boldsymbol{uxv} \prec \boldsymbol{uyv}$.*

Note that, according to this result, a comparison of two strings can ignore equal prefixes (or suffixes) — see Algorithm `COMPARE`.

Lemma 6. *[1] For any two strings $\boldsymbol{x}, \boldsymbol{y}$ with $\boldsymbol{x} \prec \boldsymbol{y}$ and any two letters λ, μ such that $\boldsymbol{y} \prec \boldsymbol{x}\lambda$:*

(i) *if $\lambda \leq \mu$, then $\boldsymbol{x}\lambda \prec \boldsymbol{y}\mu$;*
(ii) *if $\lambda > \mu$, then $\boldsymbol{y}\mu \prec \boldsymbol{x}\lambda$.*

Lemma 6 led to Algorithm `COMPARE` (Figure 1), described in [1][1]. Note that, in contrast to comments made in the Conclusion of [10], this tells us that in fact V-order comparison can be conducted in a positional manner as in lexorder comparison — in fact, `COMPARE` is an on-line algorithm [13] that requires only a one-character window. Notably, the first V-order comparison algorithm was presented in [9], and this was followed up by a couple of other interesting algorithms [2, 3, 10] before algorithm `COMPARE` was presented in [1].

```
procedure COMPARE(x, y, δ)
    i ← 1; j ← 1; δ ← 0
    while j ≤ |y| and i ≤ |x| and x[i] = y[j] do
        i ← i + 1; j ← j + 1
    if i > |x| or j > |y| then
        if |x| = |y| then
            δ ← 0; return
        if i > |x|  then
            δ ← −1; return
        δ ← 1; return
    while true do
        while j ≤ |y| and x[i] > y[j] do
            j ← j + 1
        if j > |y| then
            δ ← 1; return
        else
            i ← i + 1
        while i ≤ |x| and x[i] < y[j] do
            i ← i + 1
        if i > |x| then
            δ ← −1; return
        else
            j ← j + 1
```

Figure 1. Comparing $\boldsymbol{x}$ of length m and $\boldsymbol{y}$ of length n in V-order: $\boldsymbol{x} \prec, =, \succ \boldsymbol{y}$ according as $\delta = -1, 0, 1$.

Lemma 7. *[1–3, 9, 10] V-comparison requires linear time and constant space.*

[1] This version corrects the comparisons in lines 5 & 8 of the original presentation.

3 *V*-order String Comparison

In this section, we present a new algorithm to compare strings in V-order, which is sensitive to the structure of the input strings – the `COMPARE-Sensitive` algorithm (Figure 2). We show that probabilistically `COMPARE-Sensitive` runs faster than `COMPARE` in almost all cases of interest. We also show experimental results comparing the two algorithms. From these experiments we see that `COMPARE-Sensitive` runs at least twice as fast as `COMPARE`, in almost all cases of interest. Moreover, the new algorithm can easily be converted to work in an on-line setting, just like `COMPARE`. In addition to the new algorithm, we give a minor correction to the `COMPARE` algorithm (Figure 1).

3.1 `COMPARE-Sensitive` Algorithm

Suppose we are asked to compare two strings $\boldsymbol{x}$, $\boldsymbol{y}$ with V-forms

$$\boldsymbol{x} = \boldsymbol{x}_0\mathcal{L}_{\boldsymbol{x}}\boldsymbol{x}_1\mathcal{L}_{\boldsymbol{x}}\boldsymbol{x}_2\cdots\boldsymbol{x}_{j-1}\mathcal{L}_{\boldsymbol{x}}\boldsymbol{x}_j,$$
$$\boldsymbol{y} = \boldsymbol{y}_0\mathcal{L}_{\boldsymbol{y}}\boldsymbol{y}_1\mathcal{L}_{\boldsymbol{y}}\boldsymbol{y}_2\cdots\boldsymbol{y}_{k-1}\mathcal{L}_{\boldsymbol{y}}\boldsymbol{y}_k.$$

In this representation we can "most of the time" determine the order simply by applying conditions (C1) and (C2) of Lemma 3 to $\mathcal{L}_{\boldsymbol{x}}$ and $\mathcal{L}_{\boldsymbol{y}}$: if one maximum is greater than the other, or if the maxima are equal but have different frequencies of occurrences, then we are done. The `COMPARE-Sensitive` Algorithm (Figure 2) uses this observation to compare strings $\boldsymbol{x}$ and $\boldsymbol{y}$. The first few lines of the algorithm show the very simple preprocessing required for each string $\boldsymbol{x}, \boldsymbol{y}$ to check for conditions (C1) and (C2).

```
procedure COMPARE-Sensitive(x, y, δ)
    SCAN(x, |x|; L_x, occx)
    SCAN(y, |y|; L_y, occy)
    if L_x ≠ L_y then
        δ ← SIGN(L_x − L_y)
    else
        if occx ≠ occy then
            δ ← SIGN(occx − occy)
        else
            COMPARE-C3(x, y, L_x, δ)
```

Figure 2. Apply conditions (C1) and (C2) to $\boldsymbol{x}$ and $\boldsymbol{y}$: for $n \gg \sigma$, almost always $\mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$, while at the same time usually $occx \neq occy$, so that the routine `COMPARE-C3` does not need to be executed at all.

```
procedure SCAN(x, |x|; L_x, occx)
    L_x ← x[1]; occx ← 1
    for i ← 2 to |x| do
        if x[i] ≺ L_x then
            if x[i] = L_x then
                occx ← occx + 1
            else
                L_x ← x[i]; occx ← 1
```

Figure 3. Scan traverses the string $\boldsymbol{x}$ to compute the maximum $\mathcal{L}_{\boldsymbol{x}}$ and its frequency $occx$.

```
procedure COMPARE-C3(x, y, L_x, δ)
    i ← 1; j ← 1; δ ← 0
    while j ≤ |y| and i ≤ |x| and x[i] = y[j] do
        i ← i + 1; j ← j + 1
    if i > |x| or j > |y| then
        if |x| = |y| then
            δ ← 0; return
        if i > |x| then
            δ ← −1; return
        δ ← 1; return
    if x[i] = L_x then δ ← −1; return                    ▷ s(x_h) = ε
    if y[j] = L_x then δ ← 1; return                     ▷ s(y_h) = ε
    while true do
        while j ≤ |y| and x[i] > y[j] and y[j] < L_x do
            j ← j + 1
        if y[j] = L_x or j > |y| then                    ▷ L_x = L_y and y[j] ⊁ L_x
            δ ← 1; return
        else
            i ← i + 1
        while i ≤ |x| and x[i] < y[j] and x[i] < L_x do
            i ← i + 1
        if x[i] = L_x or i > |x| then                    ▷ x[i] ⊁ L_x
            δ ← −1; return
        else
            j ← j + 1
```

Figure 4. Comparing $\boldsymbol{x}$ of length m and $\boldsymbol{y}$ of length n in V-order, when $\mathcal{L}_x = \mathcal{L}_y$ and $occx = occy$: $\boldsymbol{x} \prec, =, \succ \boldsymbol{y}$ according as $\delta = -1, 0, 1$.

However if conditions (C1) and (C2) of Lemma 3 fail, we need to invoke the COMPARE algorithm to check for condition (C3). A potential disadvantage in invoking COMPARE is that it must rescan at least one of the strings in its full length. We propose a modification to the COMPARE algorithm, the COMPARE-C3 algorithm, which takes advantage of the values $\mathcal{L}_x = \mathcal{L}_y$ computed by SCAN to avoid this rescanning. COMPARE-C3 is motivated by the following observations:

Observation 1 *Recall that the condition (C3) of Lemma 3 first eliminates a common prefix (up to the beginning of substrings $\boldsymbol{x_h}$ and $\boldsymbol{y_h}$) in strings $\boldsymbol{x}$ and $\boldsymbol{y}$, and then compares only the substrings $\boldsymbol{x_h}$ and $\boldsymbol{y_h}$ containing the first mismatching letter in $\boldsymbol{x}$ and $\boldsymbol{y}$ while scanning the strings from left to right. Therefore, instead of comparing the entire strings $\boldsymbol{x}$ and $\boldsymbol{y}$ as in* COMPARE*, it suffices to first identify the substrings $\boldsymbol{x_h}$ and $\boldsymbol{y_h}$ and only compare them.*

Observation 2 *Observe that the substring $\boldsymbol{x_h}$ ($\boldsymbol{y_h}$) is either followed by the empty string (when $\boldsymbol{x_h}$ ($\boldsymbol{y_h}$) is a suffix of $\boldsymbol{x}$ ($\boldsymbol{y}$)), or $\mathcal{L}_x = \mathcal{L}_y$ (when $\boldsymbol{x_h}$ ($\boldsymbol{y_h}$) is not a suffix of $\boldsymbol{x}$ ($\boldsymbol{y}$)). We use this observation to identify the end of substring $\boldsymbol{x_h}$ ($\boldsymbol{y_h}$) in $\boldsymbol{x}$ ($\boldsymbol{y}$).*

Similar to COMPARE, the first **while** loop in COMPARE-C3 identifies a common prefix (alternatively the first mismatching position in strings $\boldsymbol{x}$ and $\boldsymbol{y}$ from the left). Note that this prefix might include a common prefix of substrings $\boldsymbol{x_h}$ and $\boldsymbol{y_h}$ identified under condition (C3) of Lemma 3. We denote the suffixes of $\boldsymbol{x_h}$ and $\boldsymbol{y_h}$ without a common prefix by $s(\boldsymbol{x_h})$ and $s(\boldsymbol{y_h})$, respectively. Then the second **while** loop in

`COMPARE-C3`, unlike `COMPARE`, only compares the suffixes $s(\boldsymbol{x_h})$ and $s(\boldsymbol{y_h})$, to compare strings $\boldsymbol{x}$ and $\boldsymbol{y}$. Therefore, it terminates when it encounters $\mathcal{L}_{\boldsymbol{x}}$ or ε, which marks the end of substrings $\boldsymbol{x_h}$ and $\boldsymbol{y_h}$ as seen in Observation 2. The two **if** statements before the second **while** loop ensure that `COMPARE-C3` returns correct results when either $s(\boldsymbol{x_h}) = \varepsilon$ or $s(\boldsymbol{y_h}) = \varepsilon$. For completeness we give the pseudocode for SCAN [2] (Figure 3), which is used in `COMPARE-Sensitive` to scan the string $\boldsymbol{x}$ ($\boldsymbol{y}$) to compute the maximum $\mathcal{L}_{\boldsymbol{x}}$ ($\mathcal{L}_{\boldsymbol{y}}$) and its frequency *occx* (*occy*).

If the conditions (C1) and (C2) of Lemma 3 do not hold, then we need to execute `COMPARE-C3` to compare strings $\boldsymbol{x}$ and $\boldsymbol{y}$. For such strings, scanning them and computing $\mathcal{L}_{\boldsymbol{x}}$ ($\mathcal{L}_{\boldsymbol{y}}$) and *occx* (*occy*) might result in an overhead. Therefore, such strings are possibly one of the worst case input strings for which the time required by `COMPARE-Sensitive` is the maximum. Therefore, we refer to them as "*bad strings*".

Lemma 3 *The probability of choosing a pair of bad strings $\boldsymbol{x}$ and $\boldsymbol{y}$ of length n from an alphabet of size σ is*

$$\frac{\sum_{\mathcal{L}_{\boldsymbol{x}}=1}^{\sigma}\sum_{k=1}^{n}\left(\binom{n}{k}(\mathcal{L}_{\boldsymbol{x}}-1)^{n-k}\right)^2}{\sigma^{2n}}, \tag{1}$$

where $\mathcal{L}_{\boldsymbol{x}}$ is the maximum letter in $\boldsymbol{x}$, and $k = occx$ is the number of times $\mathcal{L}_{\boldsymbol{x}}$ occurs in $\boldsymbol{x}$.

Proof.

For a given $\mathcal{L}_{\boldsymbol{x}}$ and k the number of strings of length n is computed as follows:

1. the k positions where $\mathcal{L}_{\boldsymbol{x}}$ occurs can be chosen in $\binom{n}{k}$ ways;
2. the remaining $n-k$ positions can be chosen in $(\mathcal{L}_{\boldsymbol{x}}-1)^{n-k}$ ways.

Then for fixed values of $\mathcal{L}_{\boldsymbol{x}}$ and k the total number of ways to choose $\boldsymbol{x}$ is

$$\binom{n}{k}(\mathcal{L}_{\boldsymbol{x}}-1)^{n-k}.$$

Since we choose $\boldsymbol{y}$ independently of $\boldsymbol{x}$, and $\mathcal{L}_{\boldsymbol{x}} = \mathcal{L}_{\boldsymbol{y}}$, the number of ways in which we can choose $\boldsymbol{y}$ for fixed values of $\mathcal{L}_{\boldsymbol{x}}$ and k is also $\binom{n}{k}(\mathcal{L}_{\boldsymbol{x}}-1)^{n-k}$.
Therefore the total number of ways to choose a pair of bad strings $\boldsymbol{x}$ and $\boldsymbol{y}$ for fixed values of $\mathcal{L}_{\boldsymbol{x}}$ and k is

$$\left(\binom{n}{k}(\mathcal{L}_{\boldsymbol{x}}-1)^{n-k}\right)^2.$$

Since $\mathcal{L}_{\boldsymbol{x}} \in [1..\sigma]$ and $k \in [1..n]$, the total number of ways to choose a pair of bad strings $\boldsymbol{x}$ and $\boldsymbol{y}$ is

$$\sum_{\mathcal{L}_{\boldsymbol{x}}=1}^{\sigma}\sum_{k=1}^{n}\left(\binom{n}{k}(\mathcal{L}_{\boldsymbol{x}}-1)^{n-k}\right)^2.$$

The total number of ways in which we can choose two strings over an alphabet of size σ is σ^{2n}. Therefore the probability of choosing a bad pair of strings $\boldsymbol{x}$ and $\boldsymbol{y}$ is

[2] For reasons of efficiency, the implementation of `SCAN` does not necessarily conform to the pseudocode given in (Figure 3).

$$\frac{\sum_{\mathcal{L}_{\boldsymbol{x}}=1}^{\sigma}\sum_{k=1}^{n}\left(\binom{n}{k}(\mathcal{L}_{\boldsymbol{x}}-1)^{n-k}\right)^2}{\sigma^{2n}}.$$

□

To simplify the computation in Lemma 3, we assume that $|x| = |y| = n$; that is, the strings to be compared are of the same lengths. However, Lemma 3 can easily be extended where $|x| \neq |y|$.

We computed the probabilities for the length of strings (n) ranging from 1 to 100 over alphabets of size $\sigma = 2, 4, 20$, and plotted graphs (see Appendix A) for the same. From the graphs we see that the probability of choosing a pair of bad strings $\boldsymbol{x}$ and $\boldsymbol{y}$ reaches a maximum value, and then drops significantly and stabilities to a constant value as n approaches 100. Since for all practical purposes the strings of interest are very large ($\gg 100$), we conclude that `COMPARE-Sensitive` will run faster than `COMPARE` in all cases of interest.

3.2 Experimental results

In this section, we show the results of the experiments conducted to compare the performance of `COMPARE` and `COMPARE-Sensitive` on different classes of bad strings. As expected from the Lemma 3, `COMPARE-Sensitive` runs much faster than `COMPARE`. In fact from the experimental results we see that `COMPARE-Sensitive` is twice as fast as `COMPARE`.

The experiments were conducted on a Windows 10 64-bit Operating System, with Intel(R) Xeon(R) CPU E31245 v3 @ 3.40GHz x64-based processor, having an installed memory (RAM) of 32.0 GB. The code was implemented in the `C++` language using Visual Studio 2017.

Strings $\boldsymbol{x}$ in the sample pair of strings $(\boldsymbol{x}, \boldsymbol{y})$ were chosen from the sample strings found at:

`http://www.cas.mcmaster.ca/~bill/strings/`,

and the string $\boldsymbol{y}$ was generated as a permutation of $\boldsymbol{x}$. Since $\boldsymbol{y}$ is a permutation of $\boldsymbol{x}$, the strings $\boldsymbol{x}$ and $\boldsymbol{y}$ fail conditions (C1) and (C2) of Lemma 3. Therefore each pair $(\boldsymbol{x}, \boldsymbol{y})$ is a pair of bad strings. In particular, strings $\boldsymbol{x}$ were chosen from the sample DNA, protein, random ($\sigma = 2$, $\sigma = 21$) and highly periodic strings available at the above URL. The lengths of the strings in a pair range from $n = 10K$ to $50K$. To minimize the effects of external factors (for example delays caused due to interrupts etc.) on the experiments, we executed the algorithms `COMPARE` and `COMPARE-Sensitive` on the same pair of strings ten times, and used the *minimum* time taken by each of them. In addition to this, we take the average of the time taken for 100 different pairs, to get the final data point for comparison in the graphs.

Let T_c and T_{cs} be the slopes of the `COMPARE` and `COMPARE Sensitive` lines seen in the graphs. Let

$$\alpha = \frac{T_c}{T_{cs}}.$$

Then the α values computed from the graphs for DNA strings (see Figure 5), random strings over alphabets 2 and twenty one (see Figures 6, 7), protein strings

(see Figure 8), and highly periodic strings (see Figure 9) are: $\alpha_{DNA} = 2.023$, $\alpha_{rand2} = 2.031$, $\alpha_{rand21} = 1.984$, $\alpha_{protein} = 1.954$, and $\alpha_{hp} = 2.009$, respectively. These α values suggest that `COMPARE-Sensitive` is twice as fast as `COMPARE`. Extrapolating for a value of $n = 1,000,000,000$ (1 billion) for a DNA string, the time taken by `COMPARE` is 24.44 10^{-3} secs while that for `COMPARE-Sensitive` is 12.06 10^{-3} secs. Moreover, it is seen that the size σ of the alphabet does not have any discernible affect on α.

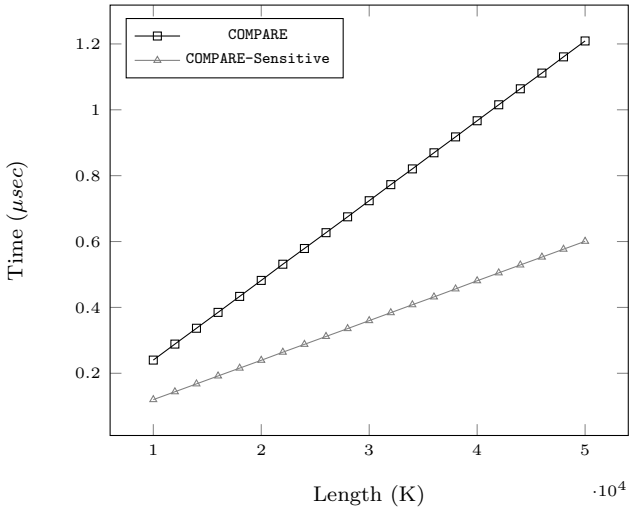


Figure 5. Runtime comparison of `COMPARE` and `COMPARE-Sensitive` on pairs of string $(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x}$ is randomly selected from DNA strings of length $2K$ to $50K$ and $\sigma = 4$, and y is a permutation of $\boldsymbol{x}$.

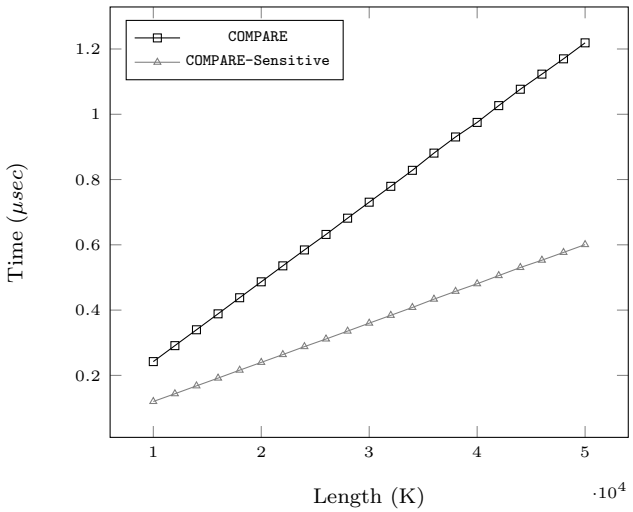


Figure 6. Runtime comparison of `COMPARE` and `COMPARE-Sensitive` on pairs of string $(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x}$ is randomly selected from random strings of length $2K$ to $50K$ and $\sigma = 2$, and y is a permutation of $\boldsymbol{x}$.

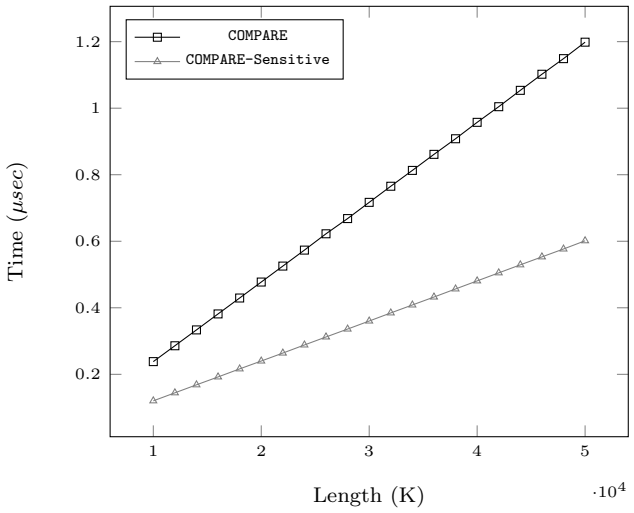


Figure 7. Runtime comparison of `COMPARE` and `COMPARE-Sensitive` on pairs of string $(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x}$ is randomly selected from random strings of length $2K$ to $50K$ and $\sigma = 21$, and y is a permutation of $\boldsymbol{x}$.

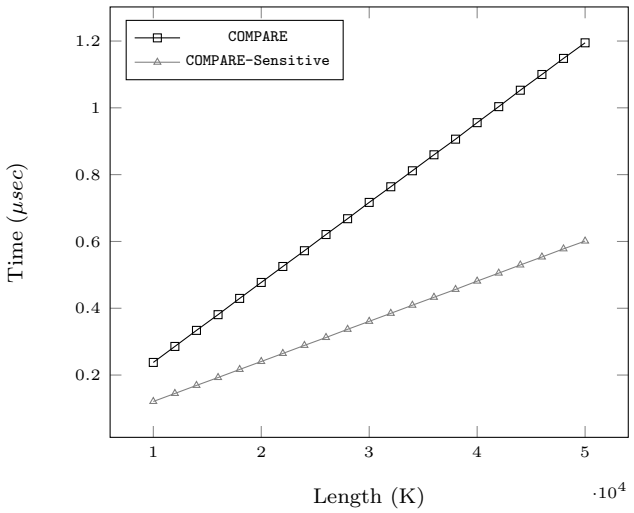


Figure 8. Runtime comparison of `COMPARE` and `COMPARE-Sensitive` on pairs of string $(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x}$ is randomly selected from protein strings of length $2K$ to $50K$ and $\sigma = 20$, and y is a permutation of $\boldsymbol{x}$.

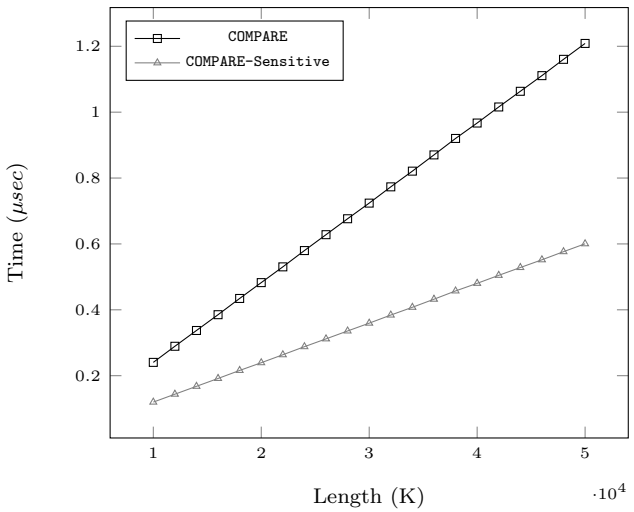


Figure 9. Runtime comparison of `COMPARE` and `COMPARE-Sensitive` on pairs of string $(\boldsymbol{x}, \boldsymbol{y})$ where $\boldsymbol{x}$ is randomly selected from highly periodic strings of length $2K$ to $50K$ and $\sigma = 2$, and y is a permutation of $\boldsymbol{x}$.

Acknowledgments

The third and fifth authors were funded by the NSERC Grant Number: 10536797. The second author was part-funded by the European Regional Development Fund through the Welsh Government.

References

1. A. Alatabbi, J. W. Daykin, J. Kärkkäinen, M. S. Rahman, and W. F. Smyth: *V–order: New combinatorial properties & a simple comparison algorithm.* Discrete Appl. Math., 215 2016, pp. 41–46.
2. A. Alatabbi, J. W. Daykin, M. S. Rahman, and W. F. Smyth: *Simple linear comparison of strings in V–order – (extended abstract)*, in International Workshop on Algorithms & Computation (WALCOM), vol. 8344 of Lecture Notes in Computer Science, Springer, 2014, pp. 80–89.
3. A. Alatabbi, J. W. Daykin, M. S. Rahman, and W. F. Smyth: *Simple linear comparison of strings in V–order.* Fundam. Inform., 139(2) 2015, pp. 115–126.
4. A. Alatabbi, J. W. Daykin, M. S. Rahman, and W. F. Smyth: *String comparison in V–order: New lexicographic properties & on-line applications.* arXiv:1507.07038, 2015.
5. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on Strings*, Cambridge University Press, New York, NY, USA, 2007.
6. T. N. Danh and D. E. Daykin: *The structure of V–order for integer vectors.* Congr. Numer.Ed. A.J.W. Hilton. Utilas Mat. Pub. Inc., Winnipeg, Canada, 113 (1996), 1996, pp. 43–53.
7. D. E. Daykin and J. W. Daykin: *Lyndon–like and V–order factorizations of strings.* J. Discrete Algorithms, 1(3–4) 2003, pp. 357–365.
8. D. E. Daykin, J. W. Daykin, and W. F. Smyth: *Combinatorics of Unique Maximal Factorization Families (UMFFs).* Fund. Inform. 97–3, Special Issue on Stringology, R. Janicki, S. J. Puglisi and M. S. Rahman (eds.), 2009, pp. 295–309.
9. D. E. Daykin, J. W. Daykin, and W. F. Smyth: *String comparison and Lyndon–like factorization using V–order in linear time*, in Symp. on Combinatorial Pattern Matching, vol. 6661, 2011, pp. 65–76.
10. D. E. Daykin, J. W. Daykin, and W. F. Smyth: *A linear partitioning algorithm for hybrid Lyndons using V–order.* Theoret. Comput. Sci., 483 2013, pp. 149–161.
11. J. W. Daykin and W. F. Smyth: *A bijective variant of the Burrows– Wheeler transform using V–order.* Theoret. Comput. Sci., 531 2014, pp. 77–89.
12. M. Lothaire: *Combinatorics on Words*, Reading, MA (1983); 2nd Edition, Cambridge University Press, Cambridge (1997)., Addison–Wesley, 1983.
13. B. Smyth: *Computing Patterns in Strings*, Pearson/Addison–Wesley, 2003.

A Figures for experiments conducted to compute probabilities of choosing a pair of bad strings of length n from an alphabet of size σ.

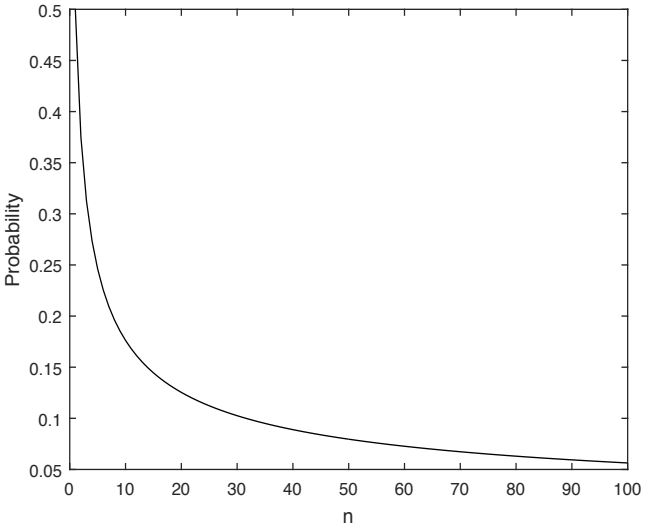


Figure 10. Probabilities computed for strings over alphabet of size $\sigma = \mathbf{2}$, and length $n \in [1..100]$ using Equation (1).

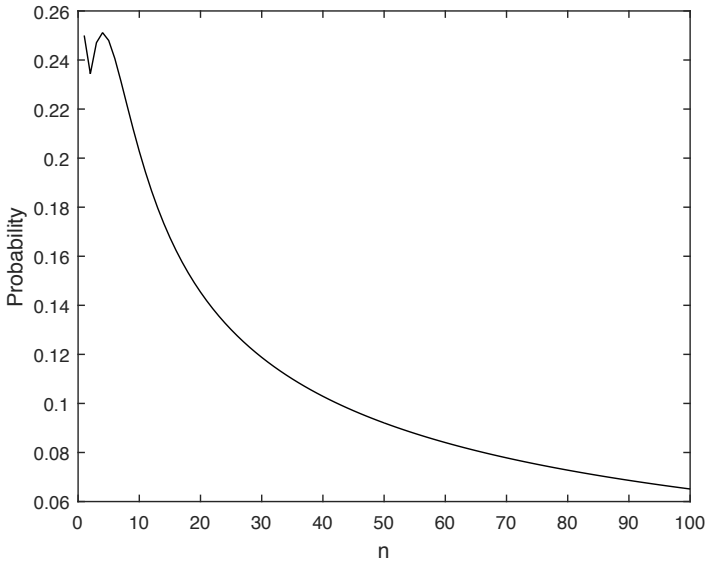


Figure 11. Probabilities computed for strings over alphabet of size $\sigma = \mathbf{4}$, and length $n \in [1..100]$ using Equation (1).

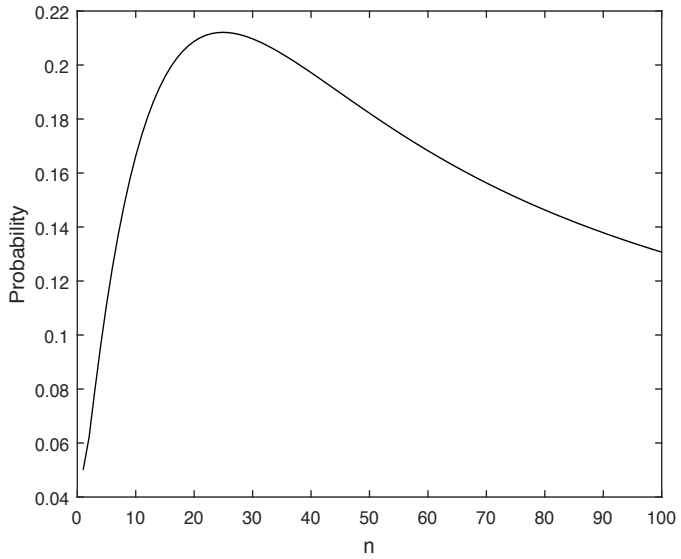


Figure 12. Probabilities computed for strings over alphabet of size $\sigma = \mathbf{20}$, and length $n \in [1..100]$ using Equation (1).

Fast and Simple Algorithms for Computing both LCS_k and LCS_{k+}

Filip Pavetić[1], Ivan Katanić[2], Gustav Matula[2], Goran Žužić[3], and Mile Šikić[2]

[1] Google Switzerland GmbH, Zürich, Switzerland
fpavetic@google.com
[2] Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia
{ivan.katanic, gustav.matula, mile.sikic}@fer.hr
[3] Carnegie Mellon University, Pittsburgh, USA
gzuzic@cs.cmu.edu

Abstract. Longest Common Subsequence (LCS) deals with the problem of measuring similarity of two strings. While this problem has been analyzed for decades, the recent interest stems from a practical observation that considering single characters is often too simplistic. Therefore, recent works introduce the variants of LCS based on shared substrings of length exactly or at least k (LCS_k and LCS_{k+} respectively). The main drawback of the state-of-the-art algorithms for computing LCS_k and LCS_{k+} is that they work well only in a limited setting: they either solve the average case well while being suboptimal in the pathological situations or they achieve a good worst-case performance, but fail to exploit the input data properties to speed up the computation. Furthermore, these algorithms are based on non-trivial data structures which is not ideal from a practitioner's point of view. We present a single algorithm to compute both LCS_k and LCS_{k+} which outperforms the state-of-the art algorithms in terms of runtime complexity and requires only basic data structures. In addition, we implement an algorithm to reconstruct the solution which offers significant improvement in terms of memory consumption. Our empirical validation shows that we save several orders of magnitude of memory on human genome data. The C++ implementation of our algorithms is made available at: https://github.com/google/fast-simple-lcsk.

Keywords: longest common subsequence, string similarity, efficient dynamic programming, bioinformatics, memory optimization

1 Introduction

Measuring the similarity of strings is one of the fundamental problems in computer science. It is very useful in many real-world applications such as DNA sequence comparison [5], differential file analysis and plagiarism detection [7]. One of the most popular techniques for efficient measurement of string similarity is the Longest Common Subsequence (LCS) [11,12,3].

LCS and extensions. Recently, there has been some critique of LCS being an oversimplified way to measure string similarity as it does not distinguish well between the sequences which consist mainly of consecutive characters and the ones which do not [4,16]. To overcome this limitation, extensions of LCS to more general variants have been proposed (see Example 1). In particular, Benson et al. [5,4] suggested LCS_k, which computes the similarity by counting the number of non-overlapping substrings of length k contained in both strings. Another extension was given by Pavetić et al. [14]: the LCS_{k++} computes similarity by summing the lengths of non-overlapping substrings of length **at least** k contained in both strings. These variants

Filip Pavetić, Ivan Katanić, Gustav Matula, Goran Žužić, Mile Šikić: Fast and Simple Algorithms for Computing both LCS_k and LCS_{k+}, pp. 50–62.
Proceedings of PSC 2018, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06484-9

have been applied in bioinformatics [15]. Later it has been renamed as $LCS_{\geq k}$ by Benson et al. [4] and as (used in this paper) LCS_{k+} by Ueki et al. [16][1].

Example 1 (Values of LCS_k and LCS_{k+} for various string pairs).

- $LCS_3(ABCBA, ABCBA) = 1$ (ABC) or (BCB) or (CBA)
- $LCS_{3+}(ABCBA, ABCBA) = 5$ $(ABCBA)$
- $LCS_2(ABXXXCDE, ABYYYCDE) = 2$ (AB, CD) or (AB, DE)
- $LCS_{2+}(ABXXXCDE, ABYYYCDE) = 5$ (AB, CDE)
- $LCS_1(AAA, AA) = LCS(AAA, AA) = 2$ (A, A)
- $LCS_{1+}(AAA, AA) = LCS(AAA, AA) = 2$ (A, A)

State of the art. For LCS_k and LCS_{k+} to be useful in practice, we need to be able to compute them efficiently. State-of-the-art algorithms are often parametrized by the total number of matching k-length substring pairs between the input strings, denoted by r. The observation that r is often limited in real-world data can be used to speed up the computation. Existing algorithms usually specialize for the situations when r is either low or high. Deorowicz and Grabowski [8] proposed several algorithms for efficient computation of LCS_k. Most notably, their *Sparse* algorithm allows both the computation of LCS_k and its reconstruction in $\mathcal{O}(m + n + r \log l)$ time and $\mathcal{O}(r)$ memory complexity, where l is the length of the optimal solution and m, n are the lengths of the two input strings. In their approach they adapt the Hunt-Szymanski [11] paradigm in a way that requires them to use a persistent red-black tree. Pavetić et al. [14,13] proposed an algorithm based on the Fenwick tree data structure [9] to efficiently compute LCS_{k+} in $\mathcal{O}(m + n + r \log r)$. Ueki et al. [16] proposed an algorithm which achieves a better worst-case complexity than Pavetić and Žužić [13] when $r \sim mn$, but a worse performance in the situations when r is small. The complexity of their algorithm is $\mathcal{O}(mn)$ and does not depend on r.

Improving the current state. The algorithms to compute LCS_k and LCS_{k+} are either of unsatisfactory runtime complexity or they rely on using complex data structures. Implementing these complex algorithms is extremely time consuming for a practitioner who usually has to make an experiment-based decision about which similarity measure is even useful for their cause. Therefore, it is valuable to have a simple and fast algorithm to compute both LCS_k and LCS_{k+}. Our contributions in this paper are:

- We propose an algorithm to compute LCS_k which achieves a favorable runtime complexity when compared to the previously known algorithms. The runtime complexity of the algorithm is $\mathcal{O}(\min(r \log l, r + ml))$, where l is the length of the optimal solution. (Section 3)
- We show that the same algorithm can be easily extended to compute LCS_{k+}. This unifies the solutions for both of these problems. (Section 4)
- We propose a heuristic to reduce memory needed to reconstruct the solution. Experiments on the human genome demonstrate that it reduces the memory usage by several orders of magnitude. (Section 5)
- Our algorithms do not rely on complex data structures such as Fenwick or a persistent red-black tree.

[1] Different authors used different names, however the definitions are the same.

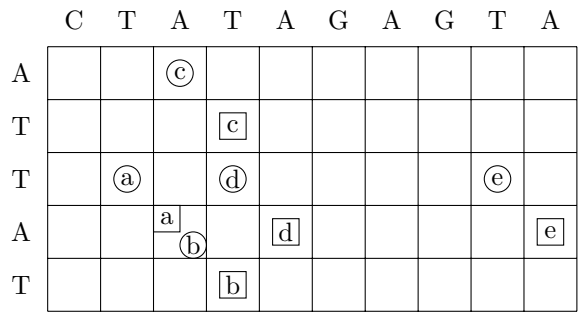


Figure 1. The Figure shows the start and end points of the match pairs produced by the two strings. In this example strings $A = ATTAT$ and $B = CTATAGAGTA$ construct exactly five match pairs for $k = 2$, denoted with a to e. Start points of the pairs are represented by circles and their end points are represented by squares. The following holds: "c precedes e", while the following does not hold: "a precedes b", "c precedes d" (Definition 5). Note that a start point of one match pair can share coordinates with the end point of another: e. g. end point of a and start point of b.

- To speed up future research on the topic, we made the implementation of our algorithms available at `https://github.com/google/fast-simple-lcsk`. As far as we know, this is the first widely accessible implementation of algorithms to compute LCS_k and LCS_{k+}.

2 Preliminaries

This section contains the definitions useful throughout the rest of the paper. Even though we inherit some of the definitions from other sources, we state them here for the sake of completeness. The inputs to all the algorithms are strings A of length m and B of length n, both over alphabet Σ. Without loss of generality we can assume that $m \leq n$. We use $X[i : j)$ to denote the substring of X, starting at (inclusive) index i and ending at (exclusive) index j. Using that notation it holds that $A[0 : m) = A$ and $B[0 : n) = B$.

Definition 2 (The LCS_k problem [4]). *Given two strings A and B of length m and n, respectively, and an integer $k \geq 1$, we say that C is a common subsequence in* exactly *k length substrings of A and B, if there exist $i_1, \ldots, i_t$ and $j_1, \ldots, j_t$ such that $A[i_s : i_s + k) = B[j_s : j_s + k) = C[p_s : p_s + k)$ for $1 \leq s \leq t$, and $i_s + k \leq i_{s+1}, j_s + k \leq j_{s+1}$ and $p_{s+1} = p_s + k$ for $1 \leq s < t, p_1 = 0$ and $|C| = p_t + k$. The longest common subsequence in* exactly *k length substrings (LCS_k) equals to maximum possible t, such that the mentioned conditions are met.*

Definition 3 (The LCS_{k+} problem [14,4,16]). *Given two strings A and B of length m and n, respectively, and an integer $k \geq 1$, we say that C is a common subsequence in* at least *k length substrings of A and B, if there exist $i_1, \ldots, i_t$, $j_1, \ldots, j_t$ and $l_1, \ldots, l_t$ such that $A[i_s : i_s + l_s) = B[j_s : j_s + l_s) = C[p_s : p_s + l_s)$ and $l_s \geq k$ for $1 \leq s \leq t$, and $i_s + l_s \leq i_{s+1}, j_s + l_s \leq j_{s+1}$ and $p_{s+1} = p_s + l_s$ for $1 \leq s < t, p_1 = 0$ and $|C| = p_t + l_t$. The longest common subsequence in* at least *k length substrings (LCS_{k+}) equals to maximum possible sum $\sum_{i=1}^{t} l_i$, such that the mentioned conditions are met.*

We state the recurrence relation to compute $LCS_k(i, j)$=$LCS_k(A[0 : i), B[0, j))$ for two strings A and B, given in [5]:

$$LCS_k(i,j) = \max \begin{cases} LCS_k(i-1,j) & \text{if } i \geq 1 \\ LCS_k(i,j-1) & \text{if } j \geq 1 \\ LCS_k(i-k,j-k)+1 & \text{if } A[i-k:i) = B[j-k:j) \end{cases} \quad (1)$$

The choice to add 1 or k in the last line of Equation 1 is arbitrary since it influences the final result only by a constant factor. We choose to add $+1$ to be consistent with prior definitions of LCS_k. When expanding the relation to LCS_{k+}, we need to adjust it as shown in [14,16]:

$$LCS_{k+}(i,j) = \max \begin{cases} LCS_{k+}(i-1,j) & i \geq 1 \\ LCS_{k+}(i,j-1) & j \geq 1 \\ \max\limits_{\substack{k \leq k' \leq min(i,j) \\ A[i-k':i) = B[j-k':j)}} LCS_{k+}(i-k',j-k')+k' & \end{cases} \quad (2)$$

Definition 4 (Match pair [5]). *Given the strings A, B and integer $k \geq 1$ we say that at (i,j) there is a **match pair** if $A[i:i+k) = B[j:j+k)$. (i,j) is also called the **start point** or the **start** of the match pair. $(i+k-1, j+k-1)$ is called the **end point** or the **end** of the match pair.*

Definition 5 (Precedence of match pairs). *Let $P{=}(i_P, j_P)$ and $G{=}(i_G, j_G)$ be match pairs. Then G **precedes** P if $i_G + k \leq i_P$ and $j_G + k \leq j_P$. In other words, G precedes P if the end of G is on the upper left side of the start of P in the dynamic programming table (see Figure 1).*

2.1 Efficient algorithms for computing LCS

The known efficient algorithms are based on the observation that in order to compute LCS via a classic dynamic programming approach[2], it is not always necessary to fill out the entire matrix. If it happens that many entries repeat in the cases when two strings have only a few pairs of matching characters, it is possible to design a structure which stores the matrix in a compressed form. We sketch the main ideas behind these techniques since we later use them as building blocks.

The algorithm by Hunt and Szymanski [11,6] traverses only the matching character pairs of the two strings in row-major order. The main idea is to maintain an array M such that M_d holds the minimum j such that there is some already processed row i for which $LCS(i,j) = d$. For simplicity we define $M_0 = 0$ and $M_d = \infty$ if no such j exists. In simpler terms, M is a compressed representation of the dynamic programming table, storing only the boundaries of same-value intervals. This is obviously useful when a row contains many repeated values. Note that M is non-decreasing. For every point (i,j) corresponding to matching character pairs in row i, let us find the biggest d such that $M_d < j$. Then there must exist a common subsequence of length d ending with (i',j') where $i' < i$ and $j' < j$. Such a subsequence can be extended by (i,j), so we know that after processing all the points in the current row we will have $M_{d+1} \leq j$. It will suffice to find a d such that $M_d < j \leq M_{d+1}$ (easily done using binary search), and set M_{d+1} to $\min(M_{d+1}, j)$ (since we're extending a subsequence

[2] This is usually done by filling out a matrix based on the relation which we get after we set $k = 1$ in either of the Equation 1 or Equation 2.

of length d ending at column M_d into a subsequence of length $d+1$ ending at column j). We note that the order in which the points of a row are processed matters: they should be ordered descending by column, so that the updates to M with the results of a new row happen effectively at the same time (otherwise queries are influenced by previous updates from the same row, which leads to incorrect results).

Hunt's algorithm was modified by Kuo and Cross [12] by replacing the binary search for each point in a row with a linear scan of M together with all the points in a row. Specifically, as we process the points of the current row, we also maintain an index d into M which we increment until $M_{d+1} \geq j$ for the current point (i, j). The increments of d amortize over the length of M, so the total complexity is $\mathcal{O}(r_i + l)$, where r_i is the number of points in row i and l is the length of the LCS. When $r_i \ll l$ this algorithm is performing worse than Hunt's variant (which would have a runtime complexity of $\mathcal{O}(r_i \log r_i)$). However, it does becomes a significant improvement as r_i approaches l.

3 Algorithm to compute LCS_k

In this section we show how to compute LCS_k efficiently. The main observation is that processing start and end points of the match pairs independently allows us to directly re-use the techniques for the efficient computation of LCS. Additionally, we dynamically adapt the computation between situations where the number of match pairs r is low as well as high, in order to secure a good worst-case performance. We achieve a runtime complexity of $\mathcal{O}(m + n + r + \min(r \log l, r + ml))$ and the memory complexity of $\mathcal{O}(l + m + n)$.

3.1 Decoupling the starts and ends of the match pairs

Another way to formulate the computation of LCS_k is to view it as a problem of finding the longest chain of match pairs. This formulation is an extension of the one previously used in the LCS literature [10]. Given a match pair P, it is possible to compute $LCS_k(P)$ using the following relation:

$$LCS_k(P) = \begin{cases} 1 & \text{if no match pair precedes P} \\ \max_G LCS_k(G) + 1 & \text{over all G preceding P} \end{cases} \tag{3}$$

In other words, we are looking for the longest chain of match pairs such that a pair which occurs earlier in the chain precedes the pairs which come later. The chain ending with a match pair P can be constructed in two ways: (i) P is added to some preceding shorter chain ending with G or (ii) a new chain containing only P is started.

Now that we have reformulated the relation for LCS_k, we show how to compute it efficiently. First we take a step back to the description of Hunt's algorithm for LCS. There we mentioned that the order in which the points within a row are processed matters since reads and updates to the helper array M happen interchangeably. If we allow two traversals of the points, the first one can only read and the second one can only update M. Note that this does not have much effect on the result of the LCS computation, but it only removes the restriction on the order in which we have to process the points within a row. The decoupling of the reads and the updates of

M is an idea which we use to generalize the algorithm to compute LCS_k. Namely, if we decouple the start and end points of all the match pairs and process them in row-major order, at every row we can: 1) do the reads of M for all the start points and then 2) update M with values of all the end points. This entire algorithm is summarized in Algorithm 1.

Algorithm 1 Computing LCS_k by decoupling start and end points

```
1: for 0 ≤ i < m do
2:     for all x = (i, j) ∈ StartPointsForRow(i) do
3:         P ← MatchPair(x)
4:         LCS_{k,start}(P) ← d s.t. M_d < j ≤ M_{d+1}
5:     end for
6:     for all x = (i, j) ∈ EndPointsForRow(i) do
7:         P ← MatchPair(x)
8:         LCS_{k,end}(P) ← LCS_{k,start}(P) + 1
9:         M_{LCS_{k,end}(P)} ← min(M_{LCS_{k,end}(P)}, j)
10:    end for
11: end for
12: return max_P LCS_{k,end}(P)
```

In the Algorithm 1 we use several quantities: 1) $LCS_{k,start}(P)$ stores the value read from M at the start point of match pair P, 2) $LCS_{k,end}(P)$ stores the value of LCS_k at the end point of P and 3) $MatchPair(x)$ retrieves a match pair for which x is a start or an end point. Lines 2-5 of the algorithm can be implemented in two different ways: 1) following Hunt's paradigm and performing binary search over the M array to do the reads and 2) following Kuo's paradigm and doing all the reads in one linear pass over array M. Since we can estimate the number of operations needed for both variants, at each row i we dynamically choose between the two options. Doing this picks up the benefits of both approaches and makes our algorithm work efficiently in both sparse and dense rows. $StartPointsForRow(i)$ and $EndPointsForRow(i)$ can be computed in $\mathcal{O}(n+m+r)$ time in multiple ways: 1) by using a suffix array based approach proposed by Deorowicz and Grabowski [8] or 2) hash the k-mers of B and create a hash table mapping to the indexes (if it happens that Σ^k is small enough to fit a computer word). Querying that table with k-mers from A trivially gives the start/end points for a wanted row. For more details see Appendix A.1.

3.2 Complexity

Generating all the match pairs takes $\mathcal{O}(m+n+r)$ time. Computing the update for row i takes $\mathcal{O}(min(r_i \ log \ l, r_i + l))$ time. Summing over all the rows implies that

$$\sum_{r=0}^{m-1} \min(r_i \ log \ l, r_i + l) \leq \mathcal{O}(\min(r \ log \ l, r + ml)) \tag{4}$$

Theorem 6. *The presented algorithm computes LCS_k with the runtime complexity of $\mathcal{O}(m+n+r+\min(r \ log \ l, r+ml))$. The memory complexity is $\mathcal{O}(l+m+n)$.*

Proof. The runtime complexity directly follows from adding up the complexities for generating the match pairs with the right side of Equation 4. Regarding memory consumption, we need to $\mathcal{O}(l)$ memory for array M, and $\mathcal{O}(m+n)$ memory to generate

the start/end points of the match pairs (same for both described approaches to generate them). This does not takes into account the memory needed for the reconstruction of the solution, which requires $\mathcal{O}(r)$ memory.

4 Algorithm to compute LCS_{k+}

In this section, we show how to modify the algorithm for LCS_k to compute LCS_{k+}. Similar to LCS_k, we look at the LCS_{k+} computation as finding chains of match pairs where consecutive ones precede or continue (see Definition 7) one another. This makes it possible to achieve the runtime complexity of $\mathcal{O}(m+n+r+\min(r(\log l+k), r+ml))$ and the memory complexity of $\mathcal{O}(l+m+n)$.

Definition 7 (Continuation of match pairs). *Let $P = (i_P, j_P)$ and $G = (i_G, j_G)$ be k-match pairs. Then P **continues** G if $i_P = i_G + 1$ and $j_P = j_G + 1$. P is only one down-right position from G, see Figure 1.*

We reproduce the relation for computing LCS_{k+} over match pairs from [14]:

$$dp(P) = \max \begin{cases} k \\ dp(G) + 1 & \text{if P is a continuation of G} \\ \max_G dp(G) + k & \text{over all G preceding P} \end{cases} \tag{5}$$

Note that this relation computes the actual number of characters in the LCS_{k+}, whereas the corresponding relation for LCS_k computes the number of blocks of length k. This difference requires us to reconsider the compressed representation M and make important changes. Instead of the array M, we introduce a new array N, defined as follows. Lets assume that we have finished processing the start and end points in row $i-1$. Let N_d denote the minimum j such that $dp(P) \geq d$ for some match pair P with end point (i', j). Furthermore, we let $N_0 = 0$ and $N_d = \infty$ when no such j exists. Note that by using '$\geq$' instead of '=' we make N non-decreasing. Indeed, since $LCS_{k+}(i-1, N_{d+1}) \geq d+1 \geq d$, N_d cannot be greater than N_{d+1}.

Using N to compute the maximum over preceding pairs in Equation 5 turns out to be the same as for LCS_k. However, updating N needs an adjustment. If we look at the end (i, j) of match pair P and suppose that we have calculated the corresponding $dp(P)$, the first temptation is to simply set $N_{dp(P)} \leftarrow \min(N_{dp(P)}, j)$. In order to maintain the non-decreasing property of N, we must ensure that $N_d \leq j$ for all $d \leq dp(P)$. However, it happens that we don't really need to update the whole prefix. If $dp(P)$ is computed from a preceding match pair G as $dp(G) + k$, upon processing the start point $(i-k+1, j-k+1)$ of P, we have $N_{dp(G)} < j-k+1 \leq j$. Since for any d, N_d may only decrease as we move from row to row, this will also hold when we reach the end point of P (in row i). Therefore, $N_{dp(G)} = N_{dp(P)-k}$ will already be smaller than or equal to j. The same holds for all indices less than $dp(G)$, as N is kept non-decreasing so we only need to set $N_{dp(P)-s} \leftarrow \min(N_{dp(P)-s}, j)$ for $s \in 0, .., k-1$. If we encounter $N_{dp(P)-s} = j$ we stop, as further values of N are already smaller than or equal to j. Since the pairs are sorted by column, this bounds the total time spent iterating through N for a single row by $O(l)$. In the case $dp(P) = dp(G)+1$ where P is a continuation of G we have $N_{dp(G)} \leq j-1 < j$, so it is enough to set $N_{dp(P)} \leftarrow \min(N_{dp(P)}, j)$.

i	N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9	N_{10}	N_{11}	N_{12}	N_{13}	N_{14}
42	0	5	5	5	5	33	43	43	43	43	44	49	49	49	49
45	0	5	5	5	5	31	31	31	31	43	44	49	49	49	49

Figure 2. Example used to highlight the difference in the updates between LCS_k and LCS_{k+}. It shows a typical situation where k entries of N need to be changed. The table shows the state of N after finishing with rows 42 and 45. Note that the content of the strings does not matter, we assume a situation where $k = 4$ and the only match pair in that range is $(42, 28)$ (N does not change between rows 43 and 44). When processing the start point of $P = (42, 28)$ in row 42, we find that $N_4 < 28 \leq N_5$, so we take $d = 4$. This means that using P we can extend a sequence of length 4 into a sequence of length 8. So when we reach row 45, and see the end point $(45, 31)$ of P, we calculate $dp(P) = d + k = 4 + 4 = 8$. Finally, we set N_5 through N_8 to $j = 31$.

Algorithm 2 Algorithm to compute LCS_{k+}

```
1: for 0 ≤ i < m do
2:     for all x = (i, j) ∈ StartPointsForRow(i) do
3:         P ← MatchPair(x)
4:         LCS_{k+,start}(P) ← d s.t. N_d < j ≤ N_{d+1}
5:     end for
6:     for all x = (i, j) ∈ EndPointsForRow(i) do
7:         P ← MatchPair(x)
8:         G ← a match pair s.t. P continues G
9:         LCS_{k+,end}(P) ← max(LCS_{k+,start}(P) + k, LCS_{k+,end}(G) + 1)
10:        for LCS_{k,end}(P) − k < z ≤ LCS_{k,end}(P) do
11:            N_z ← min(N_z, j)
12:        end for
13:    end for
14: end for
15: return max_P LCS_{k,end}(P)
```

We note that lines 2-5 (processing starts of match pairs) are identical for both Algorithm 1 and 2. The differences are in how the ends of the match pairs are processed, where two things happen: (i) continuations of match pairs are handled and (ii) the updates are done as described earlier in the section.

4.1 Complexity

In the sparse case, the time complexities of querying and updating N are $\mathcal{O}(r_i \log l)$ and $\mathcal{O}(kr_i)$ respectively, or $\mathcal{O}(r_i(\log l + k))$ in total. For the dense case we get $\mathcal{O}(r_i + l)$ for both, yielding $\mathcal{O}(\min(r_i(\log l + k), r_i + l))$ as the time complexity of processing a single row. Summing over all the rows, this is bounded by $\mathcal{O}(\min(r(\log l + k), r + ml))$. The runtime complexity of the whole algorithm is presented in Theorem 8.

Theorem 8. *The presented algorithm computes LCS_{k+} with the runtime complexity of $\mathcal{O}(m + n + r + \min(r(\log l + k), r + ml))$. The memory complexity is $\mathcal{O}(l + m + n)$.*

Proof. The analysis of memory complexity is the same as in Theorem 6. For time complexity, we again add up the complexities of generating match pairs and calculating the length of the LCS_{k+}.

Finally, we mention that one can achieve $\mathcal{O}(\log l)$ runtime complexity for querying N by using a more involved data structure (see Appendix A.2). However, we argue that the added complications are not worth it, since we are often in a setting where values

of k range in $\mathcal{O}(\log n)$. A good example of that are the DNA aligners operating on genomes having billions ($\log_2 10^9 \sim 30$) of nucleotides, with typical values of k ranging from 10 to 32 [18]. Additionally, too large values of k result in absence of match pairs, which does not make them useful [14].

5 Notes on the implementation

We implemented the algorithms described in this paper and made the code available at `https://github.com/google/fast-simple-lcsk`. This section briefly describes some details of our implementation which haven't been addressed in the rest of the paper.

The analysis of memory complexity for the reconstruction of the optimal solution in Section 3 shows that maintaining the chains of match pairs causes the biggest part of the memory consumption - if we store the match pairs until the end of the computation to do the reconstruction we need $\mathcal{O}(r)$ memory. For long inputs, this makes it impossible to fit the computation into RAM of a single computer. To reduce the memory requirements we can observe the following: at any moment of the processing, we will have a set of reconstruction paths ending with a match pair contained in the array M - we only need to keep the match pairs on these paths. As soon as there is no reconstruction path going from M to some match pair x, we can delete x. This is implemented as follows: every match pair is reference counted and has a pointer to its predecessor in the reconstruction path. The last match pair in every reconstruction path is pointed to from array M. As soon as M stops pointing to such match pair, its reference count drops to zero and it is deleted. This can further cause that the reference count of its predecessor dropped to zero so that one gets deallocated, etc.[3]

In order to demonstrate savings of the memory consumption on the real world data, we performed simple experiments on different chromosomes from the human genome[4]. We computed LCS_k of several chromosomes with themselves and compared the number of match pairs with the maximum number of match pairs kept in memory during computation for different values of k, in order to get an estimate on their relation in real data. Figure 3 shows that the savings of the memory consumption are significant. In particular, we can see that the described optimization can save 700-1000x of the memory. The savings tend to increase as the number of match pairs increases.

6 Conclusion and future work

We have demonstrated a single and simple algorithm for computing both LCS_k and LCS_{k+} of two strings. The algorithm beats the runtime complexity of the existing approaches and does not require complex data structures. Also, we have demonstrated a heuristic for reducing the memory needed to reconstruct the solution. Experiments on real data demonstrated savings of 700-1000x. Furthermore, we made our implementation widely accessible.

[3] In C++ this can be easily achieved by using std::shared_ptr.
[4] *Homo_sapiens.GRCh38 (Release 88)* obtained from *www.ensembl.org* [17]

k	chromosome	match pairs	max in memory	compression factor
30	1	5 627 330 181	7 802 719	721.20
30	2	5 737 185 065	8 186 269	700.83
30	3	3 575 560 336	6 778 074	527.51
29	1	6 428 219 516	8 090 660	794.52
29	2	6 368 795 382	8 494 144	749.78
29	3	3 971 642 925	7 020 437	565.72
28	1	7 374 448 317	8 399 480	877.96
28	2	7 108 071 229	8 824 093	805.52
28	3	4 440 640 300	7 277 931	610.15
27	1	8 540 954 582	8 732 809	978.03
27	2	8 012 803 096	9 179 547	872.89
27	3	5 014 652 783	7 554 106	663.83
26	1	9 954 502 925	9 092 744	1094.77
26	2	9 100 424 727	9 537 800	954.14
26	3	5 708 852 882	7 849 184	727.31

Figure 3. The comparison of the number of all the match pairs and the number of max match pairs kept in memory at any moment of the computation for the first few chromosomes from the human genome and varying values of k. The ratios of these numbers directly translates to the savings in the memory required for the reconstruction. We note that in our case of computing LCS_k of a chromosome with itself, the number of match pairs is going to be a sum of squares.

As a direction for future research we would like to pose few questions:

- Is it possible to create a link between the memory optimization we described with what is known as dominant points in LCS-related literature (see Appendix A.3)?
- It is not clear that all the match pairs are useful in the search for the optimal sequences. Can we speed up the algorithms by developing rules for discarding some of them, before even starting the computation?

Acknowledgments. The authors would like to thank Maria Brbić and Mario Lučić for the valuable comments on the manuscript.

References

1. A. Apostolico, S. Browne, and C. Guerra: *Fast linear-space computations of longest common subsequences.* Theor. Comput. Sci., 92(1) Jan. 1992, pp. 3–17.
2. A. Apostolico and C. Guerra: *The longest common subsequence problem revisited.* Algorithmica, 2(1) 1987, pp. 315–336.
3. B. S. Baker and R. Giancarlo: *Sparse dynamic programming for longest common subsequence from fragments.* J. Algorithms, 42(2) 2002, pp. 231–254.
4. G. Benson, A. Levy, S. Maimoni, D. Noifeld, and B. R. Shalom: *Lcsk: a refined similarity measure.* Theoretical Computer Science, 638 2016, pp. 11–26.
5. G. Benson, A. Levy, and B. R. Shalom: *Longest common subsequence in k length substrings*, in Proceedings of the 6th International Conference on Similarity Search and Applications - Volume 8199, SISAP 2013, New York, NY, USA, 2013, Springer-Verlag New York, Inc., pp. 257–265.
6. L. Bergroth, H. Hakonen, and T. Raita: *A survey of longest common subsequence algorithms*, in Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00), SPIRE '00, Washington, DC, USA, 2000, IEEE Computer Society, pp. 39–.
7. C. Y. Chen, J. Y. Yeh, and H. R. Ke: *Plagiarism detection using rouge and wordnet.* arXiv preprint arXiv:1003.4065, 2010.

8. S. Deorowicz and S. Grabowski: *Efficient algorithms for the longest common subsequence in k-length substrings.* Inf. Process. Lett., 114(11) Nov. 2014, pp. 634–638.
9. P. M. Fenwick: *A new data structure for cumulative frequency tables.* Software: Practice and Experience, 24(3) 1994, pp. 327–336.
10. C. M. Goeman Heiko: *A new practical linear space algorithm for the longest common subsequence problem.* Kybernetika, 38(1) 2002, pp. 45–66.
11. J. W. Hunt and T. G. Szymanski: *A fast algorithm for computing longest common subsequences.* Commun. ACM, 20(5) May 1977, pp. 350–353.
12. S. Kuo and G. R. Cross: *An improved algorithm to find the length of the longest common subsequence of two strings.* SIGIR Forum, 23(3-4) Apr. 1989, pp. 89–99.
13. F. Pavetić and G. Žužić: *lcskpp.* `https://github.com/fpavetic/lcskpp`, 2014.
14. F. Pavetić, G. Žužić, and M. Šikić: *LCSk++: Practical similarity metric for long strings.* CoRR, abs/1407.2407 2014.
15. I. Sović, M. Šikić, A. Wilm, S. N. Fenlon, S. Chen, and N. Nagarajan: *Fast and sensitive mapping of error-prone nanopore sequencing reads with graphmap.* bioRxiv, 2015.
16. Y. Ueki, Diptarama, M. Kurihara, Y. Matsuoka, K. Narisawa, R. Yoshinaka, H. Bannai, S. Inenaga, and A. Shinohara: *Longest common subsequence in at least k length order-isomorphic substrings*, in SOFSEM 2017: Theory and Practice of Computer Science, 2017, pp. 363–374.
17. A. Yates, W. Akanni, M. R. Amode, D. Barrell, K. Billis, D. Carvalho-Silva, C. Cummins, P. Clapham, S. Fitzgerald, L. Gil, C. G. Girn, L. Gordon, T. Hourlier, S. E. Hunt, S. H. Janacek, N. Johnson, T. Juettemann, S. Keenan, I. Lavidas, F. J. Martin, T. Maurel, W. McLaren, D. N. Murphy, R. Nag, M. Nuhn, A. Parker, M. Patricio, M. Pignatelli, M. Rahtz, H. S. Riat, D. Sheppard, K. Taylor, A. Thormann, A. Vullo, S. P. Wilder, A. Zadissa, E. Birney, J. Harrow, M. Muffato, E. Perry, M. Ruffier, G. Spudich, S. J. Trevanion, F. Cunningham, B. L. Aken, D. R. Zerbino, and P. Flicek: *Ensembl 2016.* Nucleic Acids Research, 44(D1) 2016, p. D710.
18. M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. A. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler: *Faster and more accurate sequence alignment with snap.* CoRR, abs/1111.5572 2011.

A Appendix

A.1 Generating the match pairs

Algorithm 1 assumes that it has a list of start and end points already available. Here we address how to obtain it. We offer two approaches which have the same runtime complexity, but different tradeoffs between the simplicity and the assumptions on the input data.

Deorowicz and Grabowski [8] described an algorithm for enumerating all the match pairs by using a suffix array built over the string $B\#A$[5]. The LCP (Longest Common Prefix) table is used to group all the suffixes of that string sharing a prefix of at least k. With careful bookkeeping it is possible to enumerate all the columns j which correspond to match pairs starting in row i. For more details please consult the referenced paper.

We note that in practice we are often in a setting where either the alphabet size Σ is small or useful values of k are small (e. g. DNA has $\Sigma = 4$ and the popular tools for aligning the DNA often set k to a range 10-32 [18]). If it happens that Σ^k is small enough to fit a 64-bit integer, we can easily and cheaply obtain perfect hashing of the k-mers by treating them as k-digit number in base Σ. Having that, we can build a hash table $\mathcal{H}$ mapping from the hashes of all the k-mers of B to the indices of their start position. Enumerating all the match pairs starting in a row i then comes down to hashing $A[i : i + k)$ and looking up all the indices from $\mathcal{H}$. This gives us a simpler algorithm, but still relevant in practice.

Both of the described approaches show how to generate the match pairs in $\mathcal{O}(n+m+r)$ time. We note that Algorithm 1 requires the start and end match points at every row. The start points for row i are obtained directly by generating the match pairs at row i. The end points for row i are obtained by generating the match pairs at row $i-k+1$.

A.2 An alternative to $\mathcal{O}(k)$ updates for LCS_{k+}

The operations we need are querying a single element of the array, and setting all elements in a given prefix $[0..i]$ to the minimum of a given value v and their present value.

One approach is to use a complete binary tree in which the leaves correspond to array elements in order. An internal node then represents an interval of the array. To simplify things, we think of leaves as intervals consisting of a single element. Every node of the tree has an associated value, initially set to ∞. We denote the leaf representing element N_i by $L(i)$. The algorithm is as follows:

- The update for prefix $[0, i]$ with value v is done by setting the values left siblings (if they exist) of nodes along the path from root to $L(i + 1)$ to the minimum of v and their present value (updates are lazily propagated to the leaves).
- The query for N_i is done by finding the minimum of the values along the path from root to the appropriate leaf $L(i)$.

[5] $B\#A$ = string B, concatenated with character '#', concatenated with string A

		a	a	a	a	a	a	a	a
	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	0	0
a	0	0	②	2	2	2	2	2	2
a	0	0	2	2	2	2	2	2	2
a	0	0	2	2	④	4	4	4	4
a	0	0	2	2	4	4	4	4	4
a	0	0	2	2	4	4	⑥	6	6
a	0	0	2	2	4	4	6	6	6
a	0	0	2	2	4	4	6	6	⑧

		b	b	a	a	d	d	c	c
	0	0	0	0	0	0	0	0	0
a	0	0	0	0	0	0	0	0	0
a	0	0	0	0	②	2	2	2	2
b	0	0	0	0	2	2	2	2	2
b	0	0	②	2	2	2	2	2	2
c	0	0	2	2	2	2	2	2	2
c	0	0	2	2	2	2	2	2	④
d	0	0	2	2	2	2	2	2	4
d	0	0	2	2	2	2	④	4	4

Figure 4. The full LCS_k table for two different string pairs, with $k = 2$. The encircled fields are called dominant points. For the purposes of reconstructing the solution, it is sufficient to only keep the match pairs ending at these locations.

Looking at the query and update together, we see that querying for i we will, for every previously applied update on $[0..j]$ such that $i \leq j$, encounter exactly one node that was affected by it, and thus correctly calculate the current value of N_i. Since the depth of the tree is $\mathcal{O}(\log l)$, that is also the time complexity of our operations.

By using this structure, we can reduce the runtime complexity of computing LCS_{k+} to $\mathcal{O}(m + n + r + \min(r \log l, r + ml))$.

A.3 Dominant points

The example strings shown in Figure 4 (left) generate match pairs at almost all the indexes. In this case $r = \Omega(mn)$, which makes the memory required to reconstruct the solution extremely high. Still, looking at the table, we can observe the following: in order to build any of the table entries with value 4, we only need to keep the topmost-leftmost entry with value 2, as opposed to keeping all of them. In fact, the points that we need to keep in order to reconstruct the solution are known in the LCS-related literature as *dominant points*.

Definition 9 (Dominant points [2,1]). *Lets look at a table* $T(i, j) = LCS_k(i, j)$ *or* $T(i, j) = LCS_{k+}(i, j)$. *A point* (i, j) *is then called* q*-dominant if* $T(i, j) = q$ *and for any other* (i', j') *such that* $T(i', j') = q$ *it holds that either* ($i' > i$ *and* $j' \leq j$) *or* ($i' \leq i$ *and* $j' > j$) *is true. The dominant points are then the union over* q*-dominant points over all different* q.

On Baier's Sort of Maximal Lyndon Substrings

Frantisek Franek[1], Michael Liut[1], and W. F. Smyth[1,2]

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

Abstract. We describe and analyze in terms of Lyndon words an elementary sort of maximal Lyndon factors of a string and prove formally its correctness. Since the sort is based on the first phase of Baier's algorithm for sorting of the suffixes of a string, we refer to it as Baier's sort.

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

1 Introduction

The computation of the maximal Lyndon substrings of a string has been actively researched since Bannai et al. presented a linear-time algorithm for computing runs [3]. Their algorithm relies on knowledge of all maximal Lyndon factors with respect to an order of the alphabet, and knowledge of all maximal Lyndon factors with respect to the inverse order. The maximal Lyndon factors of a string $x = x[1\,..\,n]$ are represented in the *Lyndon array* $L[1\,..\,n]$, where $L[i] =$ *the length of the maximal Lyndon factor starting at position* i: see [7,11] and references therein. Other linear-time algorithms for computing all the runs rely on Lempel-Ziv factorization, and so the comparative efficiency of the runs algorithms is determined by the comparison of computing the Lyndon array and computing the Lempel-Ziv factorization: see [6,4] and references therein.

There is only one known linear-time algorithm for computing the Lyndon array and it relies on suffix sorting: compute the suffix array, then the inverse suffix array, and then employ NSV (Next Smaller Value) algorithm to compute the Lyndon array from the inverse suffix array [12]. All other algorithms have $\mathcal{O}(n \log n)$ or $\mathcal{O}(n^2)$ worst case complexity: see [9,14] and references therein. The Lempel-Ziv factorization can be efficiently computed in linear time. Thus, our research focuses on linear computation of the Lyndon array without the need to build an unrelated global data structure such as the suffix array.

Baier [1,2] in 2016 presented an elementary though elaborate algorithm for suffix sorting.The algorithm works in two phases. Soon afterwards, Cristoph Diegelmann in

Frantisek Franek, Michael Liut, W. F. Smyth: On Baier's Sort of Maximal Lyndon Substrings, pp. 63–78.
Proceedings of PSC 2018, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06484-9

reaction to [9] was first to note that phase I of the Baier's suffix sorting algorithm in fact identifies and sorts the maximal Lyndon factors in linear time [10]. The performance of the algorithm was not great: Baier himself noted that his implementation of the suffix sort was about four times slower than the best linear algorithms for suffix sorting. Our analysis indicated that the first phase was the main culprit. We analyzed phase II in detail and presented a different implementation in [10].

There are three main goals for our study of Baier's sort. The first is to describe and formalize Baier's sort in terms of Lyndon words. The second is to provide a more formal and more detailed proof of the correctness of the sort in the framework of Lyndon words. And finally, the third goal is a detailed analysis of the method essential for a more efficient programming implementation in order to speed up the execution and lower the memory required. An additional goal is to see whether knowledge of the Lyndon array for a string can be utilized to speed up Baier's sort of the maximal Lyndon factors. This paper concerns the first two goals.

Lyndon words admit the standard factorization and so often are built from these two components in what is in essence a bottom-up approach [3,16]. However, maximal Lyndon factors of a string can be built in a top-down fashion not related to the standard factorization. In Baier's sort, the maximal Lyndon factors are determined using what we call the *water draining method*[1] since the process can be best visualized as if the string represented a bunch of hills in a water tank and we slowly drain the water from the tank. The maximal Lyndon factors are then built from the hills that the draining reveals. The *water draining method* has three steps:

(a) lower the water level by one
(b) extend the existing Lyndon factors
the revealed letters are used to extend the existing Lyndon factors where possible, or became Lyndon factors of length 1 otherwise;
(c) consolidate the new Lyndon factors
processed from the right, if several Lyndon factors are adjacent and can be joined to a longer Lyndon factor, they are joined.

In Fig. 1, we illustrate the process:

(1) We start with the string *abcdedbcdba* and a full tank of water.
(2) We drain one level, only e is revealed, nothing to extend, nothing to consolidate.
(3) We drain one more level and three d's are revealed, the first d extends e to de and the remaining two d's form Lyndon factors d of length 1, nothing to consolidate.
(4) We drain one more level and two c's are revealed, the first extends de to cde and the second extends d to cd, the consolidation then joins cde and d to $cded$ (5).

[1] This is not a pun on "graindraining" string used by Baier for illustration of his method.

(6) We drain one more level and three b's are revealed, the first extends *cded* to *bcded*, the second extends *cd* to *bcd*, the third is a Lyndon factor b of length 1, nothing to consolidate.

(7) We drain one more level and two a's are revealed, the first extends *bcded* to *abcded* and the second becomes a Lyndon factor a of length 1, in (8) *abcded* and *bcd* are joined to *abcdedbcd*, and we continue the consolidation in (9) where *abcdedbcd* and b are joined to *abcdedbcdb*, and the consolidation is complete.

So, during the process the following maximal Lyndon factors were identified: e at position 5, *de* at position 4, d at positions 6, 9, *cded* at position 3, *cd* at position 8, *bcded* at position 2, *bcd* at position 7, b at position 10, *abcdedbcdb* at position 1, and a at position 11. Note that all positions are accounted for, we really got all maximal Lyndon factors of the string *abcdedbcdba*. For a depiction of all maximal Lyndon factors of *abcdedbcdba*, see Fig. 2.

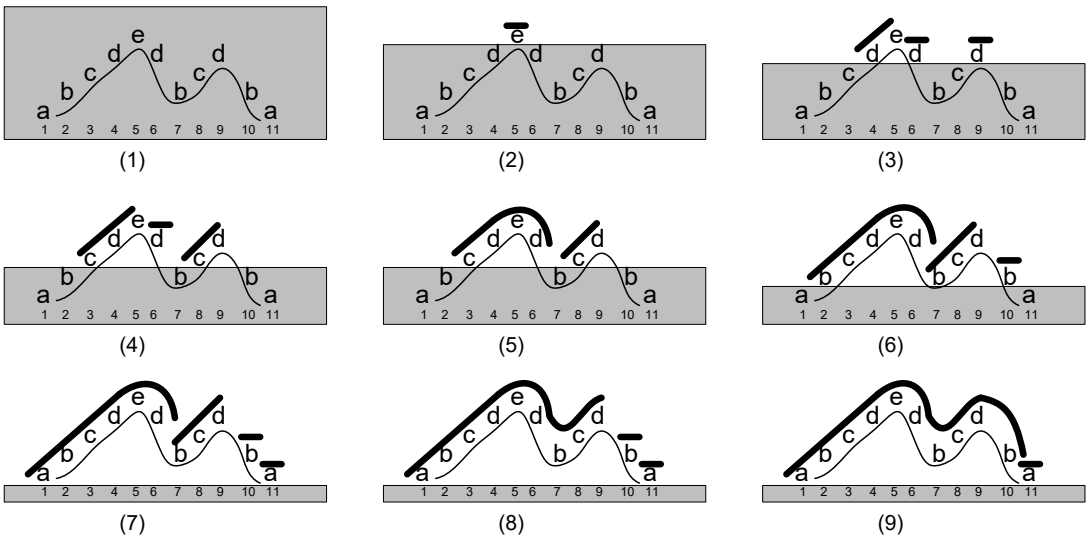


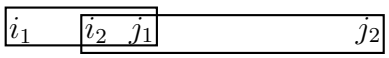
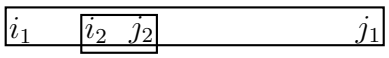
Figure 1. The water draining method for *abcdedbcdba*

The true ingenuity of Baier was to realize the water draining method by a simple linear mechanism based on the $prev()$ operator.

2 Background notation, notions and facts

In this section, we provide the description and definitions of the basic string notions and notation we are using in this paper, and the relevant basic facts. A ***string*** x is a sequence x[1..n] of ***letters*** x[i], 1 ¡ i ¡ n, each drawn from a set $\mathcal{A}$ called the ***alphabet***. The ***length*** of the string is n. $\mathcal{A}$ is a totally ordered set.

- The symbol ε denotes the empty string.
- A concatenation of two strings x and y is denoted by xy. A concatenation of k copies of u is is denoted as u^k, for an integer $k \geq 2$.
- If a string $x = uvw$, then u is a ***prefix*** of x, v is a ***factor*** or ***substring*** of x, and w is a ***suffix*** of x. A prefix (resp. suffix) is ***trivial*** if it is empty, and is ***proper*** if it is not the whole x.
- The fact that u is a prefix of x is denoted by $u \trianglelefteq x$, while the fact that u is a proper prefix of x is denoted by $u \triangleleft x$.
- A string w is a ***border*** of a string x, if w is both a proper prefix and a proper suffix of x. An empty string is a ***primitive*** border. If x has only a primitive border, it is ***unbordered***.
- A string x is ***primitive*** if there are no integer $k \geq 2$ and string u so that $x = u^k$.
- The expression $x \prec y$ denotes the fact that x is *lexicographically* smaller than y; that is, either x is a proper prefix of y or there is $i \leq \min\{|x|, |y|\}$ such that $x[1\,..\,i{-}1] = y[1\,..\,i{-}1]$ and $x[i] \prec y[i]$.
- The expression $x \preccurlyeq y$ denotes the fact that $x \prec y$ or $x = y$.
- The expression $x \prec\!\!\!\prec y$ denotes the fact that $x \prec y$, but x is not a prefix of y.
- For a string $x = wu$, uw is called a ***rotation*** of x, if either u or w is empty, than the rotation is ***trivial***.
- A string x is ***Lyndon*** if either $|x| = 1$, the so-called ***trivial*** Lyndon string, or x is lexicographically strictly smaller than any of its non-trivial rotations. Such a string is often called a *Lyndon word* [5].
- A Lyndon factor $u = x[i\,..\,j]$ of a string $x = x[1\,..\,n]$ is ***maximal*** if either $j = n$ or for any $j < \ell \leq n$, $x[i\,..\,\ell]$ is not Lyndon.
- For a string x, $\mathcal{A}_x$ denotes the ***alphabet*** of the string; that is, the set of letters occurring in x.
- For two factors $x[i_1\,..\,j_1]$ and $x[i_2\,..\,j_2]$ a string $x = x[1\,..\,n]$ with $i_1 \leq i_2$, we say that these two factors

  - are ***disjoint***, if $j_1 < i_2$ 
  - ***overlap***, if $i_2 \leq j_1$
  - ***intersect***, if $i_2 \leq j_1 \leq j_2$ 
  - the first ***includes*** the second (or, the second is ***included*** in the first), if $j_2 \leq j_1$ 

  Note that in our terminology

  $$\big(x \text{ and } y \textit{ overlap}\big) \Leftrightarrow \big((x \text{ and } y \textit{ intersect}) \text{ or } (\text{one includes the other})\big)$$

- A family of factors of a string x has the ***Monge*** property, if each two distinct factors from the family are either disjoint or one includes the other, or, equivalently, no two factors intersect.

Facts 1. Basic facts of Lyndon strings ([8,12,13,15])

(i) *For u of length > 1,*
- *u is Lyndon $\Rightarrow$ u is unbordered $\Rightarrow$ u is primitive*
- *u is Lyndon iff $u \prec u_2$ for any $u = u_1u_2$*
- *u is Lyndon iff $u_1 \prec u_2$ for any $u = u_1u_2$*
- *u is Lyndon $\Rightarrow$ there are Lyndon words u_1, u_2 so that $u_1 \prec u_2$ and $u = u_1u_2$. If u_2 is the largest possible such suffix, it is called the **standard factorization** of u.*

(ii) *Let $u = wcv$ be Lyndon and let $c \prec d$. Then $uwd = wcvwd$ is Lyndon.*

(iii) *Let $x = vu$ for non-empty v and u and let v be a Lyndon factor of x. Then v is a maximal Lyndon factor of x iff $u \prec vu$.*

(iv) *Let $u = u_1vu_2$ be Lyndon and let v be a maximal Lyndon factor of u. Then $u \prec v$.*

3 Basic Definitions and Notions

In this section we present the basic definitions and notions for the description and analysis of Baier's sort.

Definition 2. *Let $x = x[1\,..\,n]$ be a string.*

- *A **group** with a **context** u, denoted as G_u, is an ascending sequence of indices $i_1 < \cdots < i_k$ such that u is a prefix of $x[i_\ell\,..\,n]$ for every $\ell \in 1\,..\,k$.*
- *A sequence $\mathscr{C} = [G_{u(m)}, G_{u(m-1)}, \ldots, G_{u(1)}]$ is a **group configuration** for x if*
  (i) *for any $\ell \in 1\,..\,m$, $G_{u(\ell)}$ is a group with the context $u(\ell)$, and*
  (ii) *$u(m) \prec u(m-1) \prec \cdots \prec u(1)$, and*
  (iii) *for any $\ell \in 1\,..\,m$, $u(\ell)$ is a Lyndon substring of x, and*
  (iv) *all the groups are pairwise mutually disjoint, and*
  (v) *$\bigcup_{\ell=1}^{m} G_{u(\ell)} = 1\,..\,n$, and*
  (vi) *the family $\{\ \mathscr{C}\langle i\rangle \mid 1 \le i \le n\ \}$ has the Monge property, where $\mathscr{C}\langle i\rangle = x[i\,..\,i+|u(\ell)|-1]$ for the unique ℓ such that $i \in G_{u(\ell)}$, note that $\mathscr{C}\langle i\rangle = u(\ell)$.*
- *For $i, j \in G_{u(\ell)}$, $i \otimes j$ iff $|i - j| = |u(\ell)|$. The relation $\sim$ is defined as a transitive closure of $\otimes$. Thus, $i \sim j$ is an equivalence relation on $G_{u(\ell)}$. The symbol $[i]_\sim$ denotes the class of equivalence $\sim$ to which i belongs.*
- *For $i \in G_{u(\ell)}$, the **valence** of i is defined as $val_{\mathscr{C}}(i) = |[i]_\sim|$.*
- *$gr_{\mathscr{C}}(i)$ denotes the unique group of $\mathscr{C}$ the index i belongs to, i.e. $gr_{\mathscr{C}}(i) = G_{u(\ell)}$ iff $i \in G_{u(\ell)}$.*
- *A group $G_{u(\ell)}$ is **complete** if*
  - for any $i \in G_{u(\ell)}$, $\mathscr{C}\langle i\rangle$ is a maximal Lyndon factor of x, and
  - if $u(\ell) = x[j\,..\,j+|u|-1]$ is a maximal Lyndon factor of x, then $j \in G_{u(\ell)}$.

- *The operator* $prev_{\mathscr{C}}(i) = \max\{j < i \mid con_{\mathscr{C}}(gr_{\mathscr{C}}(j)) \prec con_{\mathscr{C}}(gr_{\mathscr{C}}(i))\}$*, if such a* j *exists,* `nil` *otherwise, where* $con_{\mathscr{C}}(G)$ *denotes the context of group* G *with respect to the configuration* $\mathscr{C}$*.*

Note that it is possible for an $i \in G_u$, that not only $x[i\,..\,i+|u|-1] = u$, but also $x[i+|u|\,..\,x[i+2|u|-1] = u$ and so forth. This is captured by the notion of the valence: if $i \in G_u$ and $val(i) = k$, the occurrence of u at position i is a part of a maximal repetition u^k.

It is easy to see, that for $j \in [i]_{\sim}$, $prev_{\mathscr{C}}(i) = prev_{\mathscr{C}}(j)$. Let $i, j \in G_u$. WLOG assume that $j = i+p|u|$ and so $x[i\,..\,j+|u|-1] = u^{p+1}$. Clearly, $prev_{\mathscr{C}}(i) \leq prev_{\mathscr{C}}(j)$. If $prev_{\mathscr{C}}(j) > prev_{\mathscr{C}}(i)$, it would indicate that a suffix of u must be lexicographically smaller than u, which contradicts the Lyndon property of u.

Together, (iv) and (v) assure that the groups of a group configuration form a disjoint partitioning of the string's index range $1\,..\,n$.

Since Definition 2 is quite involved, we present and illustrative example on a string $x = x[1..11] = abcdedbcdba$.

1	2	3	4	5	6	7	8	9	10	11
a	*b*	*c*	*d*	*e*	*d*	*b*	*c*	*d*	*b*	*a*

$G_a = \{11\}$, $G_{abcdedbcdb} = \{1\}$, $G_b = \{10\}$, $G_{bcd} = \{7\}$, $G_{bcded} = \{2\}$,
$G_{cd} = \{8\}$, $G_{cded} = \{3\}$, $G_d = \{6, 9\}$, $G_{de} = \{4\}$, $G_e = \{5\}$.

Take for instance the group $G_d = \{6, 9\}$, the context of the group is a string of length 1, d, and indeed, at the positions 6 and 9 we have substrings d starting. Moreover, G_d is complete, as $x[6]$ and $x[9]$ are both maximal Lyndon, and there is no other occurrence of a maximal Lyndon substring d. Similarly for $G_{abcdedbcdb} = \{1\}$, $x[1..10] = abcdedbcdb$, so it is the context, $abcdedbcdb$ is Lyndon, and $x[1..10] = abcdedbcdb$ is maximal, and there is no other occurrence of maximal $abcdedbcdb$, so $G_{abcdedbcdb}$ is complete. It is easy to see, that the whole set of the groups in the order given above is a group configuration: $a \prec abcdedbcdb \prec b \prec bcd \prec bcded \prec cd \prec cded \prec d \prec de \prec e$ and $G_a \cup G_{abcdedbcdb} \cup G_b \cup G_{bcd} \cup G_{bcded} \cup G_{cd} \cup G_{cded} \cup G_d \cup G_{de} \cup G_e = 1\,..\,11$.

Denote $\mathscr{C} = [G_a, G_{abcdedbcdb}, G_b, G_{bcd}, G_{bcded}, G_{cd}, G_{cded}, G_d, G_{de}, G_e]$. Consider the system of factors $\{\ \mathscr{C}\langle i\rangle \mid 1 \leq i \leq n\ \}$, let us demonstrate that it indeed has the Monge property:

$\mathscr{C}\langle 1\rangle = x[1\,..\,10] = abcdedbcdb$, $\mathscr{C}\langle 2\rangle = x[2\,..\,6] = bcded$,
$\mathscr{C}\langle 3\rangle = x[3\,..\,6] = cded$, $\mathscr{C}\langle 4\rangle = x[4\,..\,5] = de$,
$\mathscr{C}\langle 5\rangle = x[5] = e$, $\mathscr{C}\langle 6\rangle = x[6] = d$, $\mathscr{C}\langle 7\rangle = x[7\,..\,9] = bcd$,

$\mathscr{C}\langle 8\rangle = x[8\,..\,9] = cd$, $\mathscr{C}\langle 9\rangle = x[9] = d$, $\mathscr{C}\langle 10\rangle = x[10] = b$,
$\mathscr{C}\langle 11\rangle = x[11] = a$.

For instance, $\mathscr{C}\langle 1\rangle = x[1\,..\,10]$ includes $\mathscr{C}\langle 10\rangle = x[10]$, or $\mathscr{C}\langle 9\rangle = x[9]$ and $\mathscr{C}\langle 10\rangle = x[10]$ are disjoint, etc.

Lemma 3. *Let G_u be a group with a primitive context u for a string x. Let $i \in G_u$. Then $[i]_\sim = \{\ell \in G_u \mid \min [i]_\sim \le \ell \le \max [i]_\sim\}$.*

Proof. It suffices to prove that if $j - i = |u|$, then there is no $\ell \in G_u$ so that $i < \ell < j$. Assume there is such an ℓ. Then $x[i\,..\,i{+}|u|{-}1] = x[j\,..\,j{+}|u|{-}1] = u$ and since $j = i{+}|u|$, we have $x[i\,..\,i{+}2|u|{-}1] = 2u$. Moreover, $x[\ell\,..\,\ell{+}|u|{-}1] = u$ and $i < \ell < 2i$. Since u is primitive, this contradicts the synchronization principle for primitive strings. □

To illustrate Lemma 3, consider *abbbbabb*: $G_b = \{2, 3, 4, 5, 7, 8\}$. Consider $[3]_\sim = \{2, 3, 4, 5\}$, you can see that $[3]_\sim = \{i \in G_b \mid 2 \le i \le 5\}$. Similarly, $[7]_\sim = \{7, 8\} = \{i \in G_b \mid 7 \le i \le 8\}$.

Definition 4. *A group configuration $\mathscr{C} = [G_{u(m)}, G_{u(m-1)}, \ldots, G_{u(1)}]$ for a string x is r-**proper** for $1 \le r \le m$ if*

(vii) *all the groups $G_{u(r)}, \ldots, G_{u(1)}$ are complete; and*
(viii) *all the groups $G_{u(r-1)}, \ldots, G_{u(1)}$ have been processed; and*
(ix) *for any $i \in G_{u(r)}$, if $j = prev_{\mathscr{C}}(i)$, then $\mathscr{C}\langle j\rangle(\mathscr{C}\langle i\rangle)^t$ where $t = val_{\mathscr{C}}(i)$, is a prefix of $x[j\,..\,n]$.*

*To **process** the group $G_{u(r)}$ entails:*

(a) *Compute $P = \{prev_{\mathscr{C}_r}(i) \mid i \in G_{u(r)}$ and $prev_{\mathscr{C}_r}(i) \neq \mathtt{nil}\}$;*
(b) *Let $\approx$ be an equivalence on P defined by $i_1 \approx i_2$ iff $gr_{\mathscr{C}_r}(i_1) = gr_{\mathscr{C}_r}(i_2)$. Compute the disjoint partitioning of $P = P_1 \cup P_1 \cup \cdots \cup P_k$ into the classes of equivalence $\approx$. Let $P_1 \subseteq G_{u(n_1)}$, …, $P_k \subseteq G_{u(n_k)}$;*
(c) *For each $j \in 1\,..\,k$, let $\approx_j$ be an equivalence on P_j defined by $i_1 \approx_j i_2$ iff $i_1 = prev_{\mathscr{C}}(\ell_1)$ & $i_2 = prev_{\mathscr{C}}(\ell_2)$ & $val(\ell_1) = val(\ell_2)$. Let $[i_1]_{\approx_j}$ denote the class of equivalence of $\approx_j$ containing i_1. Define $val_{\mathscr{C}_r}([i_1]_{\approx_j}) = val(\ell_1)$ so that $i_1 = prev_{\mathscr{C}}(\ell_1)$. Compute the disjoint partitioning $P_j = P_{j,1} \cup P_{j,2} \cup \cdots \cup P_{j,t_j}$ into the classes of equivalence $\approx_j$. Moreover, let $val(P_{j,1}) > val(P_{j,2}) > \cdots > val(P_{j,t_j})$.*
(d) *for each $j \in 1\,..\,k$, for each $\ell \in 1\,..\,t_j$, move all indices of $P_{j,\ell}$ from $G_{u(n_j)}$ to a new group G'. The group G' is placed directly after $G_{u(n_j)}$ which is removed if it becomes empty. The context of G' is set to $u_{n_j}(u_r)^{val(P_{j,\ell})}$.*

Remark: Definition 4 may not seem sound, for the definition of *process* is only provided after it had been used in (viii). However, the definition is in fact recursive; that is for a 1-proper configuration, no group needs to be processed, then we process the last group to obtain a 2-proper configuration, and so on.

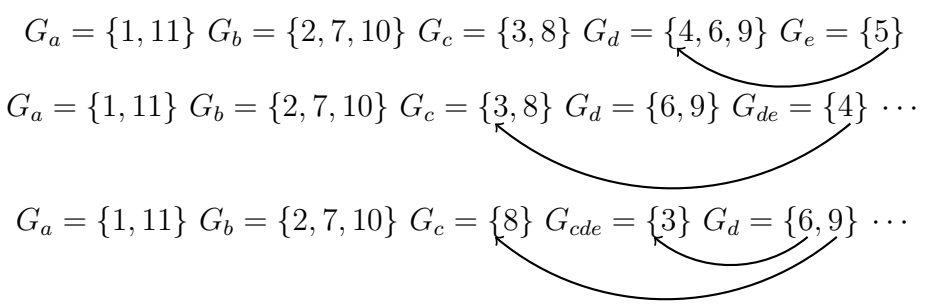
We will use a simple string *aabbabb* to illustrate the notions of the equivalence $\sim$, and the valence:

$$\begin{array}{c} 1\ 2\ 3\ 4\ 5\ 6\ 7 \\ a\ a\ b\ b\ a\ b\ b \end{array}$$

First consider the configuration $\mathscr{C}_1 = [G_a, G_b]$ where $G_a = \{1, 2, 5\}$ and $G_b = \{3, 4, 6, 7\}$. Then $val_{\mathscr{C}_1}(1) = val_{\mathscr{C}_1}(2) = val_{\mathscr{C}_1}(3) = val_{\mathscr{C}_1}(4) = val_{\mathscr{C}_1}(6) = val_{\mathscr{C}_1}(7) = 2$ and $val_{\mathscr{C}_1}(5) = 1$. It means, that for the processing of G_b we do not consider 4 and 7, only 3 and 6 and their valences of 2. Thus $P = \{2, 5\}$ as $2 = prev_{\mathscr{C}_1}(3) = prev_{\mathscr{C}_1}(4)$ and $5 = prev_{\mathscr{C}_1}(6) = prev_{\mathscr{C}_1}(7)$. Since $gr_{\mathscr{C}_1}(2) = gr_{\mathscr{C}_1}(5) = G_a$, 2 and 5 are both in the same P_j. So, after the refinement of G_a we get a new configuration $\mathscr{C}_2 = [G_a, G_{abb}, G_b]$ where $G_a = \{1\}$, $G_{abb} = \{2, 5\}$, and $G_b = \{3, 4, 6, 7\}$. Then $val_{\mathscr{C}_2}(2) = val_{\mathscr{C}_2}(5) = 2$, so $P = \{1\}$, as $1 = prev_{\mathscr{C}_2}(2) = prev_{\mathscr{C}_2}(5)$, and so we get a new and final configuration $\mathscr{C}_3 = [G_{aabbabb}, G_{abb}, G_b]$ where $G_{aabbabb} = \{1\}$, $G_{abb} = \{2, 5\}$, and $G_b = \{3, 4, 6, 7\}$.

For illustration, let us perform complete Baier's sort of *abcdedbcdba*, the arrows represent the *prev* operator. We start with the initial group configuration where each group groups all indices that start with the same letter. Processing from right, take the first unprocessed group from right, compute the *prev* values for indices in that group, and perform context concatenation wherever it points, partitioning the groups in the process. Then move to the next unprocessed group, until all groups except the very first one are processed.

$$\begin{array}{c} 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11 \\ a\ b\ c\ d\ e\ d\ b\ c\ d\ b\ a \end{array}$$



$$G_a = \{1, 11\}\ G_b = \{2, 7, 10\}\ G_{cd} = \{8\}\ G_{cded} = \{3\}\ \cdots$$

$$G_a = \{1, 11\}\ G_b = \{7, 10\}\ G_{bcded} = \{2\}\ G_{cd} = \{8\}\ \cdots$$

$$G_a = \{1, 11\}\ G_b = \{10\}\ G_{bcd} = \{7\}\ G_{bcded} = \{2\}\ \cdots$$

$$G_a = \{11\}\ G_{abcded} = \{1\}\ G_b = \{10\}\ G_{bcd} = \{7\}\ \cdots$$

$$G_a = \{11\}\ G_{abcded} = \{1\}\ G_b = \{10\}\ \cdots$$

$$G_a = \{11\}\ G_{abcdedb} = \{1\}\ \cdots$$

$$G_a = \{11\}\ \cdots$$

The processed (in bold) classes represent not only all maximal Lyndon factors of the string, but they are also lexicographically ordered:

$$G_a = \{11\}, G_{abcdedb} = \{1\}, G_b = \{10\}, G_{bcd} = \{7\}, G_{bcdded} = \{2\},$$
$$G_{cd} = \{8\}, G_{cded} = \{3\}, G_d = \{6, 9\}, G_{de} = \{4\}, G_e = \{5\}.$$

Compare it to Fig. 2 to see that all maximal Lyndon factors are accounted for.

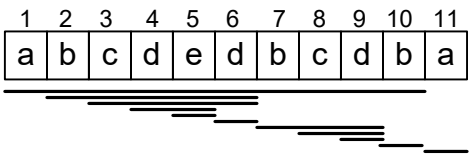


Figure 2. Maximal Lyndon Factors of *abcdedbcdba*

4 Properties of the group refinement

In this section we deal with the basic arrangement of group configurations referred in the text as ($\mathcal{B}$) given below:

Given a string $x[1\,..\,n]$ with the alphabet $\{a_1, \ldots, a_k\}$. For $\hat{r} \geq 1$,

$$\begin{aligned} &\mathscr{C}_1 = \left[Gu_{(1,m_1)}, Gu_{(1,m_1-1)} \ldots, Gu_{(1,1)}\right] \\ &\cdots \\ &\mathscr{C}_{\hat{r}} = \left[Gu_{(\hat{r},m_{\hat{r}})}, Gu_{(\hat{r},m_{\hat{r}}-1)} \ldots, Gu_{(\hat{r},1)}\right] \end{aligned} \qquad (\mathcal{B})$$

where $\mathscr{C}_1 = \left[Gu_{(1,m_1)}, Gu_{(1,m_1-1)} \ldots, Gu_{(1,1)}\right] = [G_{a_1}, \ldots, G_{a_k}]$;
for each $1 \leq r < \hat{r}$, $\mathscr{C}_r$ is an r-proper group configuration; and the

configuration $\mathscr{C}_{r+1} = \big[Gu_{(r+1,\, m_{r+1})}, Gu_{(r+1,\, m_{r+1}-1)} \ldots, Gu_{(r+1,\, 1)}\big]$ is produced by processing of the group $Gu_{(r,\, r)}$.

First, a few fundamental observations of the nature of "refinement" of the groups during processing.

Observation 5. *Referring to* $(\mathcal{B})$*, for any* $2 \le r \le \hat{r}$ *and any* $1 \le i, j \le n$*,*

(i) *If* $\mathscr{C}_r\langle i\rangle$ *is a proper suffix of* $\mathscr{C}_r\langle j\rangle$*, then* $\mathscr{C}_r\langle i\rangle = u_{(r,\, \xi)}$ *and* $\mathscr{C}_r\langle j\rangle = \mathscr{C}_\xi\langle j\rangle (u_{(r,\, \xi)})^t$ *where* $\xi < r$ *and* $t = val_{\mathscr{C}_\xi}(i)$*.*

(ii) *Either* $\mathscr{C}_r\langle i\rangle = \mathscr{C}_{r-1}\langle i\rangle$ *or* $\mathscr{C}_r\langle i\rangle = \mathscr{C}_{r-1}\langle i\rangle \big(u_{(r-1,\, r-1)}\big)^t$ *where* $t = val_{\mathscr{C}_{r-1}}(i)$*; consequently,* $\mathscr{C}_{r-1}\langle i\rangle$ *is a prefix of* $\mathscr{C}_r\langle i\rangle$*, while* $\mathscr{C}_r\langle i\rangle$ *cannot be a prefix of* $\mathscr{C}_{r-1}\langle i\rangle$*.*

(iii) *If* $|\mathscr{C}_{r+1}\langle i\rangle| > 1$*, then there exist* $\xi \le r$ *and* $1 \le \rho \le m_\xi$*, so that* $\mathscr{C}_{r+1}\langle i\rangle = u_{(\xi,\, \rho)} \big(u_{(\xi,\, \xi)}\big)^t$ *where* $t = val_{\mathscr{C}_\xi}(i + |u_{(\xi,\, \rho)}|)$*.*

Lemma 6 shows how the Monge property of the system of all occurrences of all the group contexts propagates through a group configuration arrangement.

Lemma 6. *Referring to* $(\mathcal{B})$*, then for any* $1 \le r \le \hat{r}$*, the system* $\{\, \mathscr{C}_r\langle i\rangle \mid 1 \le i \le n \,\}$ *has the Monge property.*

Proof. We are going to prove it by induction over r. First consider the case when $r = 1$.
Then each $u_{(1,\, \ell)} = c$ for some letter c of x. Thus each $\mathscr{C}_1\langle i\rangle = c$ for some letter c of x, and hence either $\mathscr{C}_1\langle i\rangle = \mathscr{C}_1\langle j\rangle$ or $\mathscr{C}_1\langle i\rangle \cap \mathscr{C}_1\langle j\rangle = \emptyset$ for any $i \ne j$.
Induction hypothesis: $\{\, \mathscr{C}_{r-1}\langle i\rangle \mid 1 \le i \le n \,\}$ has the Monge property.
Consider $\mathscr{C}_r\langle i\rangle$ and $\mathscr{C}_r\langle j\rangle$ for $i \ne j$. WLOG assume $i < j$.

- Case $\mathscr{C}_r\langle i\rangle = \mathscr{C}_{r-1}\langle i\rangle$ and $\mathscr{C}_r\langle j\rangle = \mathscr{C}_{r-1}\langle j\rangle$.
  By the induction hypothesis, either $\mathscr{C}_{r-1}\langle i\rangle \cap \mathscr{C}_{r-1}\langle j\rangle = \emptyset$ or $\mathscr{C}_{r-1}\langle i\rangle$ includes $\mathscr{C}_{r-1}\langle j\rangle$.
- Case $\mathscr{C}_r\langle i\rangle = \mathscr{C}_{r-1}\langle i\rangle \big(\mathscr{C}_{r-1}\langle \ell\rangle\big)^\rho$, $prev_{\mathscr{C}_{r-1}}(\ell) = i$, $\rho = val_{\mathscr{C}_{r-1}}(\ell)$, $\mathscr{C}_{r-1}\langle \ell\rangle = u_{(r-1,\, r-1)}$, and $\mathscr{C}_r\langle j\rangle = \mathscr{C}_{r-1}\langle j\rangle$. Since for all involved components, no two can intersect, the possible subcases are:
  - $\mathscr{C}_{r-1}\langle j\rangle$ is disjoint from $\mathscr{C}_{r-1}\langle i\rangle \big(\mathscr{C}_{r-1}\langle \ell\rangle\big)^\rho$.
  - $\mathscr{C}_{r-1}\langle j\rangle$ is included in a copy of $\mathscr{C}_{r-1}\langle \ell\rangle$.
  - $\mathscr{C}_{r-1}\langle j\rangle$ is included in $\mathscr{C}_{r-1}\langle i\rangle$.
- Case $\mathscr{C}_r\langle i\rangle = \mathscr{C}_{r-1}\langle i\rangle$ and $\mathscr{C}_r\langle j\rangle = \mathscr{C}_{r-1}\langle j\rangle \big(\mathscr{C}_{r-1}\langle \ell\rangle\big)^\rho$, $prev_{\mathscr{C}_{r-1}}(\ell) = j$, $\rho = val_{\mathscr{C}_{r-1}}(\ell)$, and $\mathscr{C}_{r-1}\langle \ell\rangle = u_{(r-1,\, r-1)}$. Since for all involved components, no two can intersect, the possible subcases are:
  - $\mathscr{C}_{r-1}\langle i\rangle$ and $\mathscr{C}_{r-1}\langle j\rangle$ are disjoint.
  - $\mathscr{C}_{r-1}\langle i\rangle$ includes $\mathscr{C}_{r-1}\langle j\rangle$ as a proper suffix. By Obs. 5(i), it means that $\mathscr{C}_{r-1}\langle j\rangle = u_{(r-2,\, r-2)}$. Since $prev_{\mathscr{C}_{r-1}}(\ell) = j$, it means $u_{(r-2,\, r-2)} \prec u_{(r-1,\, r-1)}$, which is a contradiction as $u_{(r-1,\, r-1)} \prec u_{(r-1,\, r-2)} = u_{(r-2,\, r-2)}$.

- $\mathscr{C}_{r-1}\langle i\rangle$ includes $\mathscr{C}_{r-1}\langle j\rangle\big(\mathscr{C}_{r-1}\langle \ell\rangle\big)^{\xi}$ for some $\xi \le \rho$ as a proper suffix. Then $\mathscr{C}_{r-1}\langle \ell\rangle$ is a suffix of $\mathscr{C}_{r-1}\langle i\rangle$ and by $(\mathcal{B})$, $\mathscr{C}_{r-1}\langle \ell\rangle = u_{(r-2,\,r-2)}$, i.e. $u_{(r-1,\,r-1)} = u_{(r-2,\,r-2)}$, a contradiction.
- $\mathscr{C}_{r-1}\langle i\rangle$ includes $\mathscr{C}_{r-1}\langle j\rangle\big(\mathscr{C}_{r-1}\langle \ell\rangle\big)^{\rho}$ but not as a suffix.

· Case $\mathscr{C}_r\langle i\rangle = \mathscr{C}_{r-1}\langle i\rangle\big(\mathscr{C}_{r-1}\langle \ell\rangle\big)^{\rho}$, $prev_{\mathscr{C}_{r-1}}(\ell) = i$, $\rho = val_{\mathscr{C}_{r-1}}(\ell)$, $\mathscr{C}_{r-1}\langle \ell\rangle = u_{(r-1,\,r-1)}$, and $\mathscr{C}_r\langle j\rangle = \mathscr{C}_{r-1}\langle j\rangle\big(\mathscr{C}_{r-1}\langle k\rangle\big)^{\xi}$, $prev_{\mathscr{C}_{r-1}}(k) = j$, $\xi = val_{\mathscr{C}_{r-1}}(k)$, $\mathscr{C}_{r-1}\langle k\rangle = u_{(r-1,\,r-1)}$. Since for all involved components, no two can intersect, the possible subcases are:

- $\mathscr{C}_{r-1}\langle i\rangle\big(u_{(r-1,\,r-1)}\big)^{\rho}$ is disjoint from $\mathscr{C}_{r-1}\langle j\rangle\big(u_{(r-1,\,r-1)}\big)^{\xi}$.
- $\mathscr{C}_{r-1}\langle i\rangle$ includes $\mathscr{C}_{r-1}\langle j\rangle\big(u_{(r-1,\,r-1)}\big)^{\xi}$ but not as a suffix, which is fine.
- $\mathscr{C}_{r-1}\langle i\rangle$ includes $\mathscr{C}_{r-1}\langle j\rangle\big(u_{(r-1,\,r-1)}\big)^{\xi}$ as a suffix, which is a contradiction as the ξ copies of $u_{(r-1,\,r-1)}$ are immediately followed by another ρ copies, so $\mathscr{C}_r\langle j\rangle$ should equal to $\mathscr{C}_{r-1}\langle j\rangle\big(\mathscr{C}_{r-1}\langle k\rangle\big)^{\xi+\rho}$.
- $\mathscr{C}_{r-1}\langle i\rangle$ includes $\mathscr{C}_{r-1}\langle j\rangle\big(u_{(r-1,\,r-1)}\big)^{\tau}$, $\tau < \xi$ as a proper suffix. But then $\xi - \tau = \rho$ and $\ell = k + \tau|u_{(r-1,\,r-1)}|$. Since $prev_{\mathscr{C}_{r-1}}(\ell) = i$ since $\mathscr{C}_r\langle i\rangle = \mathscr{C}_{r-1}\langle i\rangle\big(u_{(r-1,\,r-1)}\big)^{\rho}$ and $prev_{\mathscr{C}_{r-1}}(\ell) = j$ since $\mathscr{C}_r\langle j\rangle = \mathscr{C}_{r-1}\langle j\rangle\big(u_{(r-1,\,r-1)}\big)^{\xi}$, a contradiction.
- $\mathscr{C}_{r-1}\langle j\rangle$ is a suffix of $\mathscr{C}_{r-1}\langle i\rangle$ and $\rho = \xi$. Then $\ell = k$ and $prev_{\mathscr{C}_{r-1}}(\ell) = i$ since $\mathscr{C}_r\langle i\rangle = \mathscr{C}_{r-1}\langle j\rangle\big(u_{(r-1,\,r-1)}\big)^{\rho}$ and $prev_{\mathscr{C}_{r-1}}(\ell) = j$ since $\mathscr{C}_r\langle j\rangle = \mathscr{C}_{r-1}\langle j\rangle\big(u_{(r-1,\,r-1)}\big)^{\xi}$, a contradiction.

□

The process of refinement of the groups has a very particular property, namely $u_{(r,\,k)}$ is never followed immediately by $u_{(r-\xi,\,r-\xi)}$ for any $r-\xi \ge 1$ if $u_{(r,\,k)} \prec u_{(r-\xi,\,r-\xi)}$.

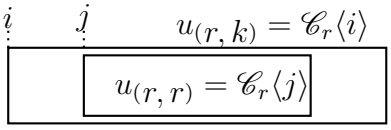
Lemma 7. *Referring to* $(\mathcal{B})$*, let* $1 \le r-\xi < r \le \hat{r}$*, let* $u_{(r,\,k)} \prec u_{(r-\xi,\,r-\xi)}$*, and let* $j = i+|u_{(r,\,k)}|$*. Then it is impossible to have* $\mathscr{C}_r\langle i\rangle = u_{(r,\,k)}$ *and* $\mathscr{C}_r\langle j\rangle = u_{(r-\xi,\,r-\xi)}$.

Proof. Arguing by contradiction assume to have it for some i, j, ξ, r, and k. Then $\mathscr{C}_{r-\xi}\langle i\rangle \trianglelefteq \mathscr{C}_r\langle i\rangle = u_{(r,\,k)} \prec u_{(r,\,r-\xi)} = u_{(r-\xi,\,r-\xi)}$, and so $prev_{\mathscr{C}_{r-\xi}}(j) \ge i$. If $prev_{\mathscr{C}_{r-\xi}}(j) = i$, then $\mathscr{C}_r\langle i\rangle \triangleleft \mathscr{C}_{r-\xi+1}\langle i\rangle \trianglelefteq \mathscr{C}_r\langle i\rangle$, a contradiction. Thus, $j_1 = prev_{\mathscr{C}_{r-\xi}}(j) > i$. So, $\mathscr{C}_{r-\xi+1}\langle j_1\rangle$ and $\mathscr{C}_r\langle i\rangle$ intersect, and so $\mathscr{C}_r\langle j_1\rangle$ and $\mathscr{C}_r\langle i\rangle$ intersect as $\mathscr{C}_{r-\xi+1}\langle j_1\rangle \trianglelefteq \mathscr{C}_r\langle j_1\rangle$. But that contradicts Lemma 6. □

We need to ascertain that the definition of the processing of $Gu_{(r,\,r)}$ can be carried out as defined in Def. 4, i.e. that the property of *prev* propagates through a group configurations arrangement. Lemma 8 shows that.

Lemma 8. *Referring to* ($\mathcal{B}$)*, for any* $1 \le r \le \hat{r}$*, for any* $i \in G_{u_{(r,\,r)}}$*, if* $j = prev_{\mathscr{C}_r}(i)$*, then* $x[j\,..\,n]$ *has* $\mathscr{C}_r\langle j\rangle(u_{(r,\,r)})^t$ *where* $t = val_{\mathscr{C}_r}(i)$*, as a prefix.*

Proof. It is clear that for $r = 1$ it is true: Let c be the largest letter in x, then $G_{u_{(1,\,1)}} = G_c$. Let $i \in G_c$ and let $j = prev_{\mathscr{C}_1}(i)$. Let $i_1 = \min\{\ell \le i \mid x[\ell] = c\}$, then $val_{\mathscr{C}_1}(i) = val_{\mathscr{C}_1}(i_1) = t$ and $prev_{\mathscr{C}_1}(i_1) = prev_{\mathscr{C}_1}(i) = j$. Then $j = i_1{-}1$, $\mathscr{C}_1\langle j\rangle = b$ for some $b \prec c$, and $\mathscr{C}_1\langle i_1\rangle = c^t$.
We are assuming it holds true for r.
We shall prove it for $r{+}1$, but first we need to prove three simple claims.
Though a bit stronger, in essence, the first claim states that the configuration illustrated below cannot happen.



Claim 1: *If* $1 \le r \le \hat{r}$*,* $i, j \in 1\,..\,n$*,* $i < j$*, and* $\xi \ge 0$*, then* $\mathscr{C}_r\langle i\rangle$ *cannot include* $\mathscr{C}_{r+\xi}\langle j\rangle = u_{(r+\xi,\,r+\xi)}$.

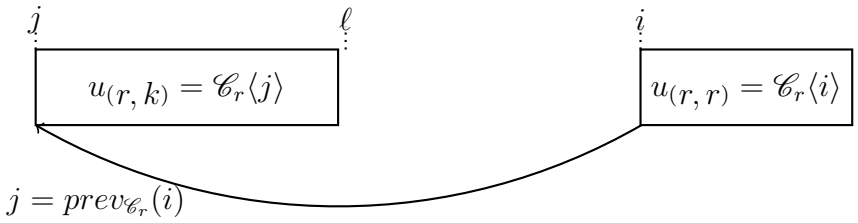
Arguing by contradiction, take the minimal r such that for some $i < j$, and some ξ, $\mathscr{C}_r\langle i\rangle$ includes $\mathscr{C}_{r+\xi}\langle j\rangle = u_{(r+\xi,\,r+\xi)}$. Let $\mathscr{C}_r\langle i\rangle = u_{(r,\,k)}$ for some k. Since $\mathscr{C}_r\langle i\rangle$ includes $\mathscr{C}_{r+\xi}\langle j\rangle$ and $\mathscr{C}_r\langle i\rangle \ne \mathscr{C}_{r+\xi}\langle j\rangle$ as $i < j$, $|\mathscr{C}_r\langle i\rangle| \ge 2$ and so $r > 1$. By Obs. 5(iii), there are $r' < r$, k', and $t \ge 1$ so that $u_{(r,\,k)} = \mathscr{C}_r\langle i\rangle = u_{(r',\,k')}\big(u_{(r',\,r')}\big)^t$. For $0 \le h < t$, define $\ell_h = i{+}|u_{(r',\,k')}| + h|u_{(r',\,r')}|$. Then $\mathscr{C}_{r'}\langle i\rangle = u_{(r',\,k')}$ and each $\mathscr{C}_{r'}\langle \ell_h\rangle = u_{(r',\,r')}$. For any $0 \le h < t$, by Lemma 6, $u_{(r+\xi,\,r+\xi)} = \mathscr{C}_{r+\xi}\langle j\rangle$ must be either disjoint from $u_{(r',\,r')} = u_{(r+\xi,\,r')} = \mathscr{C}_{r+\xi}\langle \ell_h\rangle$, or one must include the other. Let $\xi' = r{-}r'{+}\xi$, then $\xi' \ge 0$ and $r{+}\xi = r'{+}\xi'$.

- If $\mathscr{C}_{r+\xi}\langle j\rangle$ is disjoint from every $\mathscr{C}_{r+\xi}\langle \ell_h\rangle$, then $u_{(r'+\xi',\,r'+\xi')} = u_{(r+\xi,\,r+\xi)}$ must be included in $u_{(r',\,k')} = \mathscr{C}_{r'}\langle i\rangle$. So $u_{(r'+\xi',\,r'+\xi')}$ is included in $u_{(r',\,k')} = \mathscr{C}_{r'}\langle i\rangle$, which contradicts the minamility of r.
- Thus, $\mathscr{C}_{r+\xi}\langle j\rangle$ must be included in or include $\mathscr{C}_{r+\xi}\langle \ell_h\rangle$ for some $h \in 0..t{-}1$. If $\mathscr{C}_{r+\xi}\langle \ell_h\rangle = u_{(r',\,r')}$ included $\mathscr{C}_{r+\xi}\langle j\rangle = u_{(r'+\xi',\,r'+\xi')}$, we would have a contradiction with the minimality of r. Thus $\mathscr{C}_{r+\xi}\langle j\rangle = u_{(r'+\xi',\,r'+\xi')}$ must include $\mathscr{C}_{r+\xi}\langle \ell_h\rangle = u_{(r',\,r')}$, and so $j = \ell_h$, $u_{(r'+\xi',\,r'+\xi')} = u_{(r',\,r')}$, and so $r' = r'{+}\xi'$, a contradiction.

This concludes the proof of Claim 1.

Claim 2: *Let* $j = prev_{\mathscr{C}_r}(i)$*,* $\mathscr{C}_r\langle i\rangle = u_{(r,\,r)}$*, and for any* $j < i' < i$*,* $\mathscr{C}_r\langle i'\rangle \ne u_{(r,\,r)}$*. Then* $\mathscr{C}_r\langle j\rangle$ *and* $\mathscr{C}_r\langle i\rangle$ *are adjacent.*

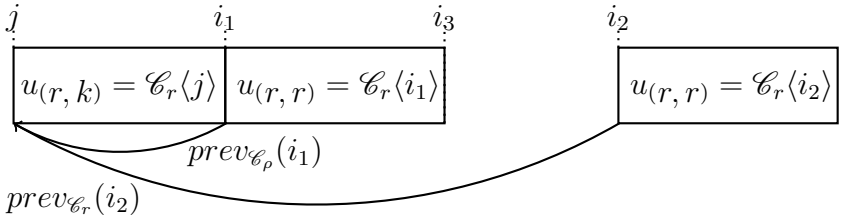
By Claim 1, $\mathscr{C}_r\langle j\rangle$ cannot include $\mathscr{C}_r\langle i\rangle$, and so, by Lemma 6, $\mathscr{C}_r\langle j\rangle$ and $\mathscr{C}_r\langle i\rangle$ are disjoint. By contradiction, we shall see that they must be adjacent. So assume that they are not adjacent,



i.e. $\mathscr{C}_r\langle\ell\rangle \succcurlyeq u_{(r,\,r)}$, therefore $\mathscr{C}_r\langle\ell\rangle = u_{(r,\,\rho)} = u_{(\rho,\,\rho)}$ for some $\rho \le r$. But since there is no occurrence of $u_{(r,\,r)}$ between j and i, $\rho < r$. But this is not possible by Lemma 7. This concludes the proof of Claim 2.

Claim 3: *Let $j = prev_{\mathscr{C}_r}(i_1) = prev_{\mathscr{C}_r}(i_2)$ and let $\mathscr{C}_r\langle i_1\rangle = \mathscr{C}_r\langle i_2\rangle = u_{(r,\,r)}$, and let $i_2 > i_1 + |u_{(r,\,r)}|$. Then $\mathscr{C}_r\langle i_1 + |u_{(r,\,r)}|\rangle = u_{(r,\,r)}$.*

Let $i_3 = i_1 + |u_{(r,\,r)}|$. Since $j = prev_{\mathscr{C}_r}(i_2)$, it follows that $i_3 \in \bigcup_{\xi=1}^{r} G u_{(r,\,\xi)}$. If $i_3 \in \bigcup_{\xi=1}^{r-1} G u_{(r,\,\xi)}$,



then $\mathscr{C}_r\langle i_3\rangle = u_{(r,\,\xi)} = u_{(\xi,\,\xi)}$ for some $\xi < r$. Since $\mathscr{C}_\xi\langle i_1\rangle \trianglelefteq \mathscr{C}_r\langle i_1\rangle \prec u_{(r,\,\xi)}$, it follows that $i_4 = prev_{\mathscr{C}_\xi}(i_3) \ge i_1$. If $prev_{\mathscr{C}_\xi}(i_3) = i_1$, then for some $t \ge 1$, $\mathscr{C}_{\xi+1}\langle i_1\rangle = u_{(r,\,r)}\big(u_{(\xi,\,\xi)}\big)^t \trianglelefteq \mathscr{C}_r\langle i_1\rangle = u_{(r,\,r)}$, a contradiction. Thus $i_4 > i_1$ and so $\mathscr{C}_{\xi+1}\langle i_4\rangle = x[i_4\,..\,i_3{-}1]\big(u_{(\xi,\,\xi)}\big)^t \trianglelefteq \mathscr{C}_r\langle i_4\rangle$ and so $\mathscr{C}_r\langle i_1\rangle$ and $\mathscr{C}_r\langle i_4\rangle$ intersect, a contradiction with Lemma 6. Thus, $i_3 \in G u_{(r,\,r)}$.

Now we can prove the induction step. Let $j = prev_{\mathscr{C}_r}(i)$, let $p = |u_{(r,\,r)}|$, and let $\mathscr{C}_r\langle i\rangle = u_{(r,\,r)}$. Let ℓ be the smallest i such that $j = prev_{\mathscr{C}_r}(i)$ and $\mathscr{C}_r\langle i\rangle = u_{(r,\,r)}$. Then by Claim 3, $x[i_1\,..\,i_1{+}tp{-}1] = \big(u_{(r,\,r)}\big)^t$ where $t = val_{\mathscr{C}_r}(\ell)$, moreover $x[\ell{-}p\,..\,\ell{-}1] \ne u_{(r,\,r)}$ and $x[i{+}tp\,..\,i{+}(t{+}1)p{-}1] \ne u_{(r,\,r)}$. By Claim 2, $\mathscr{C}_r\langle j\rangle$ and $\mathscr{C}_r\langle\ell\rangle$ are adjacent, and so $x[j\,..\,n]$ has $\mathscr{C}_r\langle j\rangle\big(u_{(r,\,r)}\big)^t$ as a prefix. □

Note that for $r = 1$, $G u_{(1,\,1)} = G_{a_k}$ where a_k is the largest letter of x, and so G_{a_k} is complete. Lemma 9 shows how the processing of $G u_{(r,\,r)}$ makes the group $G u_{(r{+}1,\,r{+}1)}$ complete, i.e. how it propagates through the refinement process.

Lemma 9. *Referring to* ($\mathcal{B}$)*, for any* $1 \leq r < \hat{r}$*, after processing the group* $Gu_{(r,\,r)}$*, the group* $Gu_{(r+1,\,r+1)}$ *is complete.*

Proof. Assume to have $i \in Gu_{(r+1,\,r+1)}$ so that $u_{(r+1,\,r+1)} = \mathscr{C}_{r+1}\langle i\rangle = \mathscr{C}_r\langle i\rangle = u_{(r,\,r+1)}$ is not maximal, i.e. $u_{(r+1,\,r+1)} \preccurlyeq x[j\,..\,n]$, where $j = i+|u_{(r+1,\,r+1)}|$. There are two cases.

- $Gu_{(r+1,\,r+1)} = Gu_{(r,\,r+1)}$.
  Then for some $t \geq 1$, and some ξ, $x[j\,..\,n]$ has $\big(u_{(r,\,r+1)}\big)^t u_{(r,\,\xi)}$ as a prefix and $u_{(r,\,\xi)}$ is not a prefix of $u_{(r,\,r+1)}$, and $\mathscr{C}_{r+1}\langle i\rangle = u_{(r+1,\,r+1)} = u_{(r,\,r+1)} \preccurlyeq \big(u_{(r,\,r+1)}\big)^t u_{(r,\,\xi)}$, and so $u_{(r,\,r+1)} \prec\!\cdot\, u_{(r,\,\xi)}$, and so $u_{(r,\,r)} \preccurlyeq u_{(r,\,\xi)}$ and $\xi \leq r$. Then we have $u_{(r+1,\,r+1)}$ immediately followed by $u_{(\xi,\,\xi)}$, $\xi < r{+}1$, and $u_{(r+1,\,r+1)} \prec u_{(\xi,\,\xi)}$, which contradicts Lemma 7.
- $Gu_{(r+1,\,r+1)}$ was created from $Gu_{(r,\,r+1)}$ and so $\mathscr{C}_{r+1}\langle i\rangle = u_{(r,\,r+1)}\big(u_{(r,\,r)}\big)^t$ for $t = val_{\mathscr{C}_r}(j)$. Let $\ell = i{+}t|u_{(r,\,r+1)}|$. There are three subcases:

  (α) For some $p \geq 1$, some $t_1 > t$, $x[j\,..\,n]$ has as a prefix $\Big(u_{(r,\,r+1)}\big(u_{(r,\,r)}\big)^t\Big)^p u_{(r,\,r+1)}\big(u_{(r,\,r)}\big)^{t_1}$. But then there is a group that has $u_{(r,\,r+1)}\big(u_{(r,\,r)}\big)^{t_2}$, $t_2 \geq t_1$, as the context, which means that the group with the context $u_{(r,\,r+1)}\big(u_{(r,\,r)}\big)^t$ cannot be $Gu_{(r+1,\,r+1)}$, a contradiction.

  (β) For some $p \geq 1$, some $t_1 < t$, $x[j\,..\,n]$ has as a prefix $\Big(u_{(r,\,r+1)}\big(u_{(r,\,r)}\big)^t\Big)^p u_{(r,\,r+1)}\big(u_{(r,\,r)}\big)^{t_1} u_{(r,\,\xi)}$ and $u_{(r,\,\xi)} \neq u_{(r,\,r)}$. Since $u_{(r,\,r+1)} \prec u_{(r,\,\xi)}$, it follows that $\xi \leq r$, but since $u_{(r,\,\xi)} \neq u_{(r,\,r)}$, we have $\xi < r$. So $u_{(r,\,\xi)} = u_{(\xi,\,\xi)}$. But then we have $u_{(r,\,r)}$ immediately followed by $u_{(\xi,\,\xi)}$ with $u_{(r,\,r)} \prec u_{(\xi,\,\xi)}$, which contradicts Lemma 7.

  (γ) For some $p \geq 1$ and some ξ, $x[j\,..\,n]$ has $\Big(u_{(r,\,r+1)}\big(u_{(r,\,r)}\big)^t\Big)^p u_{(r,\,\xi)}$ as a prefix and so that $u_{(r,\,r+1)} \prec\!\cdot\, u_{(r,\,\xi)}$. It follows that $\xi \leq r$. If $u_{(r,\,\xi)} = u_{(r,\,r)}$, then this would be case (α) as $x[j\,..\,n]$ would have $\Big(u_{(r,\,r+1)}\big(u_{(r,\,r)}\big)^t\Big)^{p-1} u_{(r,\,r+1)}\big(u_{(r,\,r)}\big)^{t+1}$ as a prefix. Thus we can assume that $\xi < r$ and so $u_{(r,\,\xi)} = u_{(\xi,\,\xi)}$. But then we have $u_{(r,\,r)}$ immediately followed by $u_{(\xi,\,\xi)}$ with $u_{(r,\,r)} \prec u_{(\xi,\,\xi)}$, which contradicts Lemma 7.

Thus, we showed that for any $i \in Gu_{(r+1,\,r+1)}$, $\mathscr{C}_{r+1}\langle i\rangle$ is a maximal Lyndon factor, i.e. $Gu_{(r+1,\,r+1)}$ is complete. □

Theorem 10. *Referring to* $(\mathcal{B})$, $\mathscr{C}_{\hat{r}} = \left[G_{u_{(}\hat{r}, m_{\hat{r}})}, G_{u_{(}\hat{r}, m_{\hat{r}}-1)} \ldots, G_{u_{(}\hat{r}, 1)}\right]$ *is an* $\hat{r}$*-proper configuration.*

Proof. It is straightforward to see that $\mathscr{C}_{\hat{r}}$ satisfies Def. 2(i)-(v). That it satisfies Def. 2(vi) follows from Lemma 6. Thus, $\mathscr{C}_{\hat{r}}$ is a group configuration. That Def. 4(vii) is satisfied follows from Lemma 9. That Def. 4(viii) is satisfied follows from the fact that $\mathscr{C}_{\hat{r}}$ was obtained by processing of $G_{u_{(}\hat{r}-1, \hat{r}-1)}$. Finally, the fact that Def. 4(ix) is satisfied follows from Lemma 8. □

Theorem 11. *For a string* x*, Baier's sort identifies and sorts all maximal Lyndon factors of* x*.*

Proof. Consider $(\mathcal{B})$ when the process of refinement stops. Consider an index i. There is a unique $1 \leq \ell \leq m_{\hat{r}}$ so that $i \in G_{u_{(}\hat{r}, \ell)}$. Thus $x[i\,..\,n]$ has $u_{(}\hat{r}, \hat{r})$ as a prefix. Since $G_{u_{(}\hat{r}, \ell)}$ is complete, $x[i\,..\,i+|u_{(}\hat{r}, \ell)|-1] = u_{(}\hat{r}, \ell)$ is a maximal Lyndon factor. Hence all maximal Lyndon factors of x are accounted for. □

5 Conclusion and future work

We showed how the process of refinement of the initial configuration propagates and is sustained as long as there are some i's in the group being processed such that $prev(i) \neq \texttt{nil}$. We showed that this process will identify and present in a sorted way all maximal Lyndon factors of a given string.

An algorithmic analysis of the process and implementation details are a necessary further step. Baier's implementation of phase I of his algorithm is linear in time, due to the use of the *prev* operator. The *prev*() values can be computed for the initial configuration in $\mathcal{O}(n)$ steps using a stack of size n, where n is the length of the input string, and then during the processing they can be updated in $\mathcal{O}(k)$ steps to work properly for the next level of refinement, where k is the the size of the group being processed. Our introduction of the *valence* makes the treatment of repetitions of the context of the group being processed more mathematically sound and straightforward and it lends itself to a simpler algorithmic treatment allowing to process the indices of the group being processed simply from the largest to the smallest with very little overhead. Our `C++` implementation uses all these mathematical insights and thus only uses memory of $13n$ integers, a significant improvement over Baier's implementation. Moreover, our implementation uses a static approach, so all required memory of $13n$ integers is allocated once and no further dynamic memory allocation is required, significantly speeding up the execution. Of course, rigorous comparison testing of the performances of the two implementations is needed before any conclusion can be drawn. The code can be obtained from

`http://www.cas.mcmaster.ca/~franek/research/bls.cpp`

References

1. U. Baier: *Linear-time suffix sorting — a new approach for suffix array construction.* M.Sc. Thesis, University of Ulm, 2015.
2. U. Baier: *Linear-time suffix sorting — a new approach for suffix array construction*, in 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016), R. Grossi and M. Lewenstein, eds., vol. 54 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2016, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 23:1–23:12.
3. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta: *The "Runs" Theorem.* SIAM J. COMPUT., 46 2017, pp. 1501–1514.
4. G. Chen, S. Puglisi, and W. Smyth: *Lempel-Ziv factorization using less time and space*, in Mathematics in Computer Science 1-4, J. Chan and M. Crochemore, eds., 2008, pp. 482–488.
5. K. T. Chen, R. H. Fox, and R. C. Lyndon: *Free differential calculus. iv. the quotient groups of the lower central series.* Annals of Mathematics, 68(1) 1958, pp. 81–95.
6. M. Crochemore, L. Ilie, and W. Smyth: *A simple algorithm for computing the Lempel-Ziv factorization*, in Proc. 18th Data Compression Conference, J. Storer and M. Marcellin, eds., 2008, pp. 482–488.
7. J. Daykin, F. Franek, J. Holub, A. S. Islam, and W. Smyth: *Reconstructing a string from its Lyndon arrays.* Theoretical Computer Science, 710 2018, pp. 44–51.
8. J.-P. Duval: *Factorizing words over an ordered alphabet.* J. Algorithms, 4(4) 1983, pp. 363–381.
9. F. Franek, A. S. Islam, M. S. Rahman, and W. Smyth: *Algorithms to compute the Lyndon array*, in Proceedings of Prague Stringology Conference 2016, PSC'16, 2016, pp. 172–184.
10. 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.
11. A. Glen, J. Simpson, and W. Smyth: *Counting Lyndon factors.* Electronic J. Combinatorics, 24 2017, pp. 3–28.
12. C. Hohlweg and C. Reutenauer: *Lyndon words, permutations and trees.* Theoretical Computer Science, 30(1) 2003, pp. 173–178.
13. M. Lothaire: *Combinatorics on words*, Addison-Wesley, Reading, Mass., 1983.
14. F. Louza, G. Manzini, W. Smyth, and G. Telles: *Lyndon array construction during Burrows-Wheeler inversion.* submitted for publication, 2017.
15. G. Melancon: *Lyndon word*, in Encyclopedia of Mathematics, M. Hazewinkel, ed., Springer Science+Business Media B.V. / Kluwer Academic Publishers, 2001.
16. J. Sawada and F. Ruskey: *Generating Lyndon brackets.* Journal of Algorithms, 46 2003, pp. 21–26.

Constrained Approximate Subtree Matching by Finite Automata

Eliška Šestáková, Bořivoj Melichar, and Jan Janoušek

Faculty of Information Technology,
Czech Technical University in Prague,
Thákurova 9, 160 00 Praha 6, Czech Republic
Eliska.Sestakova@fit.cvut.cz
Borivoj.Melichar@fit.cvut.cz
Jan.Janousek@fit.cvut.cz

Abstract. Processing tree data structures usually requires a pushdown automaton as a model of computation. Therefore, it is interesting that a finite automaton can be used to solve the constrained approximate subtree pattern matching problem. A systematic approach to the construction of such matcher by finite automaton, which reads input trees in prefix bar notation, is presented. Given a tree pattern and an input tree with m and n nodes, respectively, the nondeterministic finite automaton for the pattern is constructed and it is able to find all occurrences of the pattern to subtrees of the input tree with maximum given distance k. The distance between the pattern and subtrees of an input tree is measured by minimal number of restricted tree edit operations, called leaf nodes edit operations. The corresponding deterministic finite automaton finds all occurrences in time $\mathcal{O}(n)$ and has $\mathcal{O}(|A|^k m^{k+1})$ states, where A is an alphabet containing all possible node labels. Note that the size is not exponential in the number of nodes of the tree pattern but only in the number of errors. In practise, the number of errors is expected to be a small constant that is much smaller than the size of the pattern. To achieve better space complexity, it is also shown how dynamic programming approach can be used to simulate the nondeterministic automaton. The space complexity of this approach is $\mathcal{O}(m)$, while the time complexity is $\mathcal{O}(mn)$.

Keywords: finite automaton, approximate tree pattern matching, subtree matching, constrained tree edit distance, dynamic programming

1 Introduction

Exact tree pattern matching, the process of finding all matches of a tree pattern in an input tree, is an analogous problem to the string pattern matching. One of the approaches used for string pattern matching is to construct a finite automaton for the pattern [3,13]. The automata approach for solving exact tree pattern matching problem has also been studied in [5,9] using pushdown automaton as a model of computation. For other methods used to solve exact tree pattern matching problem see [6,7] and [2].

Approximate tree pattern matching problem is an extension of both exact tree pattern matching and approximate string pattern matching. The goal of the approximate string pattern matching problem is to find a substring in an input text with the minimal distance (string-to-string correction problem) to a given pattern. Similarly, as for the exact string matching problem, the automata approach can be used again to solve the approximate string pattern matching problem as well, see [12,13].

The goal of the approximate tree pattern matching problem is to find all occurrences of a given tree pattern in an input tree with the minimal distance. Similarly,

Eliška Šestáková, Bořivoj Melichar, Jan Janoušek: Constrained Approximate Subtree Matching by Finite Automata, pp. 79–90.
Proceedings of PSC 2018, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06484-9

approximate subtree pattern matching is the process of finding all occurrences of a tree pattern to subtrees of an input tree with the minimal distance. To measure the distance between two trees (tree-to-tree correction problem [17]), a tree edit distance is used. Common operations used are node rename, node insertion and node deletion, proposed in [10]. The recent algorithm proposed in [4] computes the tree edit distance between two rooted ordered trees with m and n nodes in $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space, where $m \leq n$.

Several authors also proposed restricted forms of tree-to-tree correction problem where only a limited set of edit operations is allowed. For example, Selkow in [14] introduced a constrained tree edit distance, sometimes referred to as 1-degree edit distance, where delete and insert operations are restricted to leaf nodes of a tree. Selkow's approach is recursive, so the distance between two trees can always be computed. For more details on the tree edit distance survey see [1,8].

Both tree-to-tree correction and approximate tree pattern matching problems have applications is several areas such as genetics, XML processing and databases, compiler optimization or natural language processing. Therefore, many solutions exist. For example, see [14,16,17,18] and [1,6,11]. However, most of them lack clear references to a systematic approach of the standard theory of formal languages and automata.

This paper shows that finite automata can be used to solve approximate subtree pattern matching problem with a constrained set of tree edit operations allowed. This is quite interesting since processing tree data structures usually requires a pushdown automaton as a model of computation. However, using only a special restricted set of tree edit operations allows a finite automaton to be the sufficient model of computation. The proposed method defines leaf nodes edit operations involving only simple tasks, such as node rename, leaf insertion and leaf deletion. However, it is not allowed to use these operations repeatedly such as Selkow [14]. In other words, it is not allowed to use operations leaf node insertion and leaf node deletion recursively to insert or delete a subtree of an arbitrary size. Therefore, the distance between two trees can be unknown, if the trees cannot be transformed to each other using only leaf nodes edit operations.

First of all, the proposed method forms linear notations for a given tree pattern and an input tree by traversing their tree structures in a sequential way. After that, a finite automaton for constrained approximate subtree matching, which is directly analogous to approximate string matching automata, is constructed. The proposed automaton is built over a tree pattern P based on a maximum distance (number of errors) k desired and is able to find all occurrences of subtree P within a given input tree T in time linear to the number of nodes of T.

The major issue in automata theory is often the size of the deterministic automaton, which can be exponential in the number of nodes of the tree pattern. However, the size of the automaton in this case is $\mathcal{O}(|A|^k m^{k+1})$, where m is the number of nodes of the tree pattern P, k is the maximum number of errors and A is an alphabet containing all possible node labels. In practise the number of errors is expected to be a small constant, that is much smaller than the size of the pattern. The paper also presents how dynamic programming can be used to simulate the nondeterministic finite automaton. This approach comes with the $\mathcal{O}(m)$ space complexity and $\mathcal{O}(mn)$ time complexity.

The rest of this paper is organised as follows. Basic definitions are given in Section 2. The problem definition is presented in Section 3. Section 4 introduces the nondeterministic finite automaton for the constrained approximate subtree pattern

matching. Following Section 5 deals with the dynamic programming approach used to simulate the nondeterministic automaton. Section 6 discusses the corresponding deterministic finite automaton and finally, Section 7 is the conclusion and future work discussion.

2 Basic Notions

An *alphabet* A is a finite non-empty set whose elements are called symbols. A *nondeterministic finite automaton* (NFA) is a 5-tuple $M = (Q, A, \delta, q_0, F)$, where Q is a finite set of states, A is an alphabet, δ is a state transition function from $Q \times A$ to the power set of Q, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states. A finite automaton is *deterministic* (DFA) if $\forall a \in A, q \in Q : |\delta(q, a)| \leq 1$.

A *rooted and directed tree* T is an acyclic connected directed graph $T = (N, E)$, where N is a set of nodes and E is a set of ordered pairs of nodes called directed edges. A *root* is a special node $r \in N$ with in-degree 0. All other nodes of a tree T have in-degree 1. There is just one path from the root r to every node $n \in N$, where $n \neq r$. A node n_1 is a *direct descendant* of a node n_2 if a pair $(n_2, n_1) \in E$.

T is called a *labelled* tree if there is a symbol from a finite alphabet A assigned to each node. T is called an *ordered* tree if a left-to-right sibling ordering in T is given. A *subtree* of a tree $T = (N, E)$ rooted at node $n \in N$ is a tree $T_n = (N_n, E_n)$, such that n is the root of T_n and N_n, E_n is the greatest possible subset of N, E, respectively.

The *size* of a tree $T = (N, E)$ denoted as $|T|$ is the cardinality of N. Any node of a tree with out-degree 0 is called a *leaf*. Leaves(T) stands for a set of all leaf nodes of the tree T. In the following, P and T will be used to denote rooted, ordered and labelled trees called a *tree pattern* and an *input tree*, respectively.

3 Problem Statement

A special type of the approximate tree pattern matching problem called approximate subtree matching is considered, where the goal is to find all occurrences of a tree pattern P to subtrees of an input tree T with maximum of k errors. Formally, the approximate subtree pattern matching problem for a maximum given distance is defined in the following way:

Definition 1 (Approximate subtree pattern matching with maximum of k errors). *A tree pattern P matches an input tree $T = (N, E)$ in a node $n \in N$ if the distance D between the pattern P and the subtree of T rooted at n is less than or equal to k, i.e., $D(P, T_n) \leq k$.*

The distance between a tree pattern and subtrees of an input tree is measured by minimal number of simple operations, called leaf nodes edit operations, applied to the tree pattern. Formal definitions of the leaf nodes edit distance and leaf nodes edit operations follows.

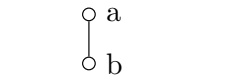
Definition 2 (Leaf nodes edit distance). *Let T_1 and T_2 be two rooted, ordered and labelled trees. The leaf nodes edit distance between T_1 and T_2, noted as $D_L(T_1, T_2)$, is the minimal number of leaf nodes edit operations needed to transform T_1 to T_2. The distance D_L is unknown if T_1 cannot be transformed to T_2 by using only leaf nodes edit operations.*

Definition 3 (Leaf nodes edit operations). *Let $T = (N, E)$ be a rooted, ordered and labelled tree and S_1, S_2, S_3 be sets such that $S_1 = N$, $S_2 = \text{Leaves}(T)$, $S_3 = N$. Then leaf nodes edit operations for T are defined as follows:*

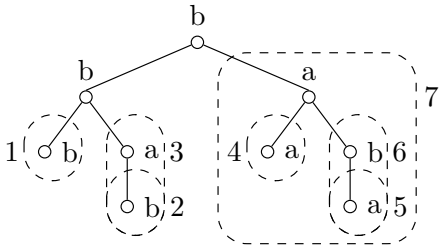
1. *node rename: change the label of a node $n \in S_1$ and assign $S_1 = S_1 \setminus \{n\}$,*
2. *leaf node deletion: delete a non-root leaf node $n \in S_2$ and assign $S_2 = S_2 \setminus \{n\}$,*
3. *leaf node insertion: insert a leaf node n_1 as a child of a node $n_2 \in S_3$. Do not update any set.*

In other words, it is not allowed to use operations node rename, leaf node insertion and leaf node deletion recursively to insert or delete a subtree of an arbitrary size.

Example 4. Let P and T be a tree pattern and an input tree as depicted in Figure 1a and Figure 1b, respectively. Having only leaf nodes edit operations allowed the tree pattern P can be modified as follows: (1) rename the root node a or the leaf node b, (2) delete the leaf node b, (3) add leaf nodes as children of the node b or as left or right siblings of the node b.



(a) Tree pattern P



(b) Input tree T

Figure 1: Graphical representation of the approximate subtree matching problem using leaf nodes edit distance $k = 2$

Thus, there are seven occurrences of the tree pattern P in the input tree T with maximum of $k = 2$ errors as shown in Figure 1b. These occurrences are marked with dashed lines and have numbers 1–7 assigned. Starting from the left, there are following errors in the occurrences of the tree pattern P to subtrees of the input tree T:

(1. match) 2 errors: rename the node a to b, delete the node b,
(2. match) 2 errors: rename the node a to b, delete the node b,
(3. match) exact match, 0 errors,
(4. match) 1 error: delete the node b,
(5. match) 1 error: delete the node b,
(6. match) 2 errors: rename the node a to b, rename the node b to a,
(7. match) 2 errors: add a leaf a as a child and as a left sibling of the node b.

Every sequential algorithm traverses a processed tree structure in a sequential order of nodes, which forms a corresponding linear notation of the tree structure. The proposed method uses the following linear notation of trees called the prefix bar notation which was introduced by Stoklasa, Janoušek and Melichar in [15].

Definition 5 (Prefix bar notation). *The prefix bar notation* pref_bar(T) *of a tree* T *is defined as follows:*

1. pref_bar(a) $= a \mid$ *if* a *is both the root and a leaf,*
2. pref_bar(T) $= a$ pref_bar(b_1) pref_bar(b_2) $\cdots$ pref_bar(b_n) $\mid$ *if* a *is the root of the tree* T *and* $b_1, b_2, \ldots, b_n$ *are direct descendants of* a*.*

Example 6. Let P and T be a tree pattern and an input tree as depicted in Figure 1a and Figure 1b, respectively. The prefix bar notations of P and T are described as follows: pref_bar(P) $= a\,b\,|\,|$ and pref_bar(T) $= b\,b\,b\,|\,a\,b\,|\,|\,|\,a\,a\,|\,b\,a\,|\,|\,|\,|$.

4 Nondeterministic Finite Automaton for Constrained Approximate Subtree Matching

This section deals with the constrained approximate subtree pattern matching by a nondeterministic finite automaton, which reads an input tree T in the prefix bar notation. The finite automaton is able to find all approximate occurrences of a tree pattern P with maximum of k errors to subtrees of an input tree T using leaf nodes edit distance.

The method is analogous to the construction of approximate string pattern matching automata. The NFA is built for the tree pattern P to recognize a language $A^*X(P,k)$, where A is an alphabet containing all possible node labels that may occur in both the tree pattern and the input tree. The alphabet also includes a special symbol called bar (noted as |), introduced in the definition of the prefix bar notation. $X(P,k)$ is a finite language generated for the number of allowed errors $k \geq 1$ from a given tree pattern P using leaf nodes edit operations, formally defined as $X(P,k) = \{\text{pref\_bar}(S) : \text{pref\_bar}(S) \in A^*, D_L(P,S) \leq k\}$. Thus, the proposed automaton can find all occurrences of $x \in X(P,k)$ in a given prefix bar notation of an input tree. The construction of the NFA is described by Algorithm 1 in detail.

Example 7. Let P be a tree pattern as shown in Figure 1a with its prefix bar notation pref_bar(P) $= a\,b\,|\,|$. The transition diagram of the NFA constructed for the tree pattern P and maximum number of errors $k = 2$ by Algorithm 1 is shown in Figure 2.

The NFA has a regular structure. State q_{ij} is at depth i (a position in the pattern) and on level j (number of errors). States $q_{i'j'}$ are assistant nodes used for insert leaf operations allowing inserting bars. Insert leaf operations are represented by two subsequent "vertical" transitions. The first transitions are labelled by all symbols of the alphabet A (except |) and they are followed by second transitions labelled by |. Rename node operations are represented by "diagonal" transitions labelled by those symbols of the alphabet A (except |) for which no direct transition to the next state exists. "Diagonal" ε-transitions represent delete leaf operations.

Input: A tree pattern P ($|P| = m$) and its prefix bar notation $\text{pref\_bar}(P) = p_1 p_2 \cdots p_{2m}$, maximum number k of errors allowed.

Output: NFA M accepting language $A^* X(P, k)$.

1. Let $M' = (\{q_0, q_1, \ldots, q_{2m}\}, A, \delta, q_0, \{q_{2m}\})$ be an exact pattern matching automaton, where
   (1) $\forall i,\ 0 \leq i < 2m :\ q_{i+1} \in \delta(q_i, p_{i+1})$,
   (2) $\forall a \in A :\ q_0 \in \delta(q_0, a)$.
2. Create a sequence of $k+1$ instances of M' such that: $M'_j = (Q_j, A, \delta_j, q_{0j}, F_j)$, $j = 0, 1, 2, \ldots, k$. $Q_j = \{q_{0j}, q_{1j}, \ldots, q_{2mj}\}, F_j = \{q_{2mj}\}$.
3. Construct the automaton $M = (Q, A, \delta, q_0, F)$ as follows:
   (a) $Q = \bigcup_{j=0}^{k} Q_j$, $q_0 = q_{00}$, $F = \bigcup_{j=0}^{k} F_j$
   (b) (copy) $\forall q \in Q, a \in A, j = 0, 1, 2, \ldots, k$ do $\delta(q, a) = \delta_j(q, a)$,
   (rename) $\forall i = 0, 1, \ldots, 2m-1, j = 0, 1, \ldots, k-1, a \in A \setminus \{p_{i+1}, |\}$ do
   if $(p_{i+1} \neq |)$ then $q_{i+1,j+1} \in \delta(q_{ij}, a)$,
   (delete) $\forall i = 0, 1, \ldots, 2m-2, j = 0, 1, \ldots, k-1$ do
   if $((p_{i+1} \neq |) \wedge (p_{i+2} = |))$ then $q_{i+2,j+1} \in \delta(q_{ij}, \varepsilon)$
   (insert) $\forall i = 1, 2, \ldots, 2m-1, j = 0, 1, \ldots, k-1, a \in A \setminus \{|\}$ do
   $Q = Q \cup \{q_{i'j'}\}, q_{i'j'} \in \delta(q_{ij}, a), q_{i,j+1} \in \delta(q_{i'j'}, |)$.
4. Remove all states inaccessible from state q_0 in M.

Algorithm 1: Construction of NFA that finds all approximate occurrences of a tree pattern P to subtrees of an input tree using leaf nodes edit distance.

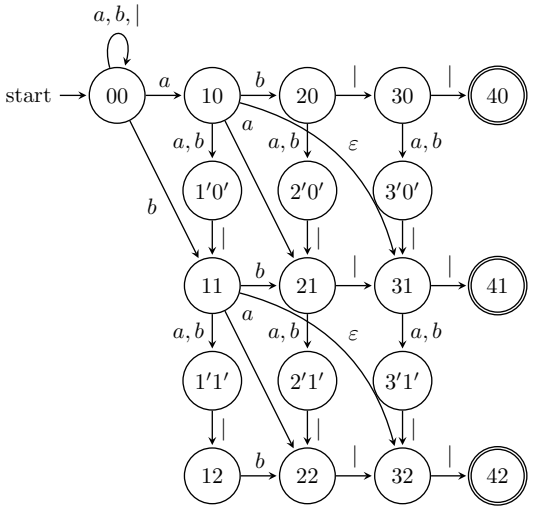


Figure 2: Transition diagram of NFA from Example 7 accepting all approximate occurrences of a tree pattern P in prefix bar notation $\text{pref\_bar}(P) = a\, b\, |\, |$ in an input tree using leaf nodes edit operations with maximum of $k = 2$ errors

5 Simulation of the Nondeterministic Finite Automaton for Constrained Approximate Subtree Matching

This section describes how the dynamic programming approach can be used to simulate the nondeterministic finite automaton for constrained approximate subreee matching. Given a tree pattern P with m nodes and an input tree T with n nodes, the algorithm computes a matrix D of size $(2m+1) \times (2n+1)$. Each element of the matrix $d_{i,j}$ $(0 \leq i \leq 2m, 0 \leq j \leq 2n)$ contains the constrained distance between the partial trees represented by prefix bar notations of the tree pattern of length i and the input tree of length j. Elements of the matrix D are computed as follows:

$$
\begin{aligned}
&d_{i,0} = k+1 \qquad 0 < i \leq 2m \\
&d_{0,j} = 0 \qquad 0 \leq j \leq 2n \\
&d_{i,j} = min \begin{cases}
\text{if } (p_i = t_j) \text{ then } d_{i-1,j-1} & \text{(match)} \\
\text{if } (p_i \neq t_j \wedge p_i, t_j \neq |) \text{ then } d_{i-1,j-1} + 1, & \text{(rename)} \\
\text{if } (i > 1 \wedge p_i = | \wedge p_{i-1} \neq |) \text{ then } d_{i-2,j} + 1, & \text{(delete)} \\
\text{if } (j > 1 \wedge t_j = | \wedge t_{j-1} \neq |) \text{ then } d_{i,j-2} + 1, & \text{(insert)} \\
k+1 & \text{(otherwise)}
\end{cases} \\
&\qquad\qquad\qquad\qquad\qquad\qquad 0 < i \leq 2m,\ 0 < j \leq 2n
\end{aligned}
$$

This formula represents the simulation of the nondeterministic finite automaton introduced in the previous section. If $d_{2m,j} \leq k$ then the tree pattern occurs in the input tree T with $d_{2m,j}$ errors ending at position j in the pref_bar(T). Note, that all values $d_{i,j} > k$ in the matrix D can be replaced by the value $k+1$ representing number of errors higher than k.

Example 8. Let P be a tree pattern (pref_bar(P) = $a\,b\,|\,|$) and T be an input tree (pref_bar(T) = $b\,b\,b\,|\,a\,b\,|\,|\,|\,a\,a\,|\,b\,a\,|\,|\,|\,|$) as shown in Figure 1a and Figure 1b, respectively. The matrix D computed for the tree pattern P and the input tree T with maximum number of errors $k = 2$ is shown in Table 1. The occurrences of the tree pattern P in the input tree T are marked with bold. All errors higher than 2 are in the matrix represented by number 3.

D	-	b	b	b	|	a	b	|	|	|	a	a	|	b	a	|	|	|	|
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	3	1	1	1	2	0	1	1	3	3	0	0	1	1	0	2	3	3	3
b	3	3	1	1	2	3	0	3	3	3	3	1	3	1	2	2	3	3	3
|	3	2	2	2	1	1	2	0	3	3	1	1	1	2	1	2	2	3	3
|	3	3	3	3	**2**	3	3	**2**	**0**	3	3	3	**1**	3	3	**1**	**2**	**2**	3

Table 1: Matrix D for a tree pattern P, input tree T and $k = 2$, where pref_bar(P) = $a\,b\,|\,|$ and pref_bar(T) = $b\,b\,b\,|\,a\,b\,|\,|\,|\,a\,a\,|\,b\,a\,|\,|\,|\,|$

Theorem 9. *The dynamic programming approach described by the formula can be used to simulate a run of the NFA for constrained approximate subtree matching using leaf nodes edit operations.*

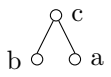
Proof. The NFA has a regular structure. It consists of depths i ($0 \leq i \leq 2m$) and levels j ($0 \leq j \leq 2n$). A depth represents a position in the pattern, while a level stands for the number of existing errors. Therefore, state q_{ij} is at depth i and on level j. In the matrix D, each row i represents the depth i and each column j corresponds to a j-th step of a run of the NFA (i.e., j symbols of pref_bar(T) read).

Every value $d_{i,j}$ of the matrix D stands for a level number (number of errors) of the topmost active state in i-th depth of the NFA and j-th step of the run of the NFA. If the value $d_{i,j} > k$ it simply means that there is no active state for the particular depth and step of the NFA.

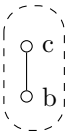
At the beginning only the initial state is active, which is in the formula represented by setting $d_{0,0} = 0$ and $d_{i,0} = k + 1$ ($0 < i \leq 2m$). The second part of the formula $d_{0,j} = 0$ ($0 \leq j \leq 2n$) simulates the self loop in the initial state. Third part of the formula describes the individual operations. The term $d_{i-1,j-1}$ simulates a matching transition – the value is copied to $d_{i,j}$ as the level has not changed and depth and step of the NFA is increased by 1. Term $d_{i-1,j-1} + 1$ corresponds to a rename transition – the level, depth and step are all increased by 1. Delete transition is represented by term $d_{i-2,j} + 1$, level is increased by 1, depth (position in the pattern) is increased by 2, but position in the text is not changed. The term $d_{i,j-2} + 1$ simulates insert transitions – level is again increased by 1, position in the text is increased by 2, but the depth is not increased. The last term sets $d_{i,j}$ to $k + 1$, which means there is no possible transition.

Therefore, all transitions of the NFA are considered. If $d_{2m,j} \leq k$ then a final state $q_{2m,d_{2m,j}}$ is active and the match is reported. The tree pattern occurs in the input tree T with $d_{2m,j}$ errors ending at position j in the pref_bar(T).
Possibly, there can exist more ways in the nondeterministic automaton leading to an accepting state. To ensure that values $d_{2m,j}$ are minimal possible, the third part of the formula contains minimum function *min*. Example 10 supports this statement. □

Example 10. Let P be a tree pattern and T be an input tree as shown in Figure 3a and Figure 3b, respectively, where pref_bar(P) = $c\,b\,|\,a\,|\,|$, pref_bar(T) = $c\,b\,|\,|$. Obviously, the pattern P occurs in the tree T just once with 1 error (delete leaf node a). However, the pattern P also occurs in T with 2 errors (delete leaf node b, rename a to b). Without minimum function in the dynamic programming formula, the second match would be reported instead the first one.



(a) Tree pattern P



(b) Input tree T

Figure 3: Graphical representation of Example 10

Theorem 11. *The simulation of the NFA for constrained approximate subtree matching using leaf nodes edit operations by dynamic programming has time complexity* $(2m + 1) \times (2n + 1) = \mathcal{O}(mn)$ *and space complexity* $4m = \mathcal{O}(m)$.

Proof. The dynamic programming approach builds a matrix D of size $(2m + 1) \times (2n + 1)$. All operations introduced in the formula can be done in constant time,

hence the time complexity of the simulation is $\mathcal{O}(mn)$. In the matrix D every column is computed using just two previous columns. Therefore, only $4m$ space is needed in order to compute all values and the space complexity results in $\mathcal{O}(m)$.

6 Deterministic Finite Automaton for Constrained Approximate Subtree Matching

To achieve better time complexity, the NFA can be turned into the DFA by using the standard determinisation algorithm (see [13], Algorithm 1.40). To compute the positions of all occurrences of the tree pattern P in the input tree T, the DFA is simply run on the prefix bar notation of the input tree T. The DFA reports a match every time it goes through a final state. All occurrences are located in time linear to the number of nodes of T.

Theorem 12. *Given an input tree T ($|T| = n$) and a tree pattern P ($|P| = m$), the deterministic automaton for approximate subtree matching that is obtained using standard determinisation algorithm over the NFA constructed by Algorithm 1 finds all approximate occurrences of the subtree P in the input tree T using leaf nodes edit distance in time $2n = \mathcal{O}(n)$.*

Proof. The prefix bar notation of the input tree T is in the searching phase read exactly once, symbol by symbol from left to right. The appropriate transition is taken each time a symbol is read, resulting in exactly $2n$ transitions. Approximate occurrences of the tree pattern P are reported each time that DFA goes through a final state.

In order to prove the upper bound of the state complexity of the deterministic finite automaton that finds all approximate occurrences of subtree P in an input tree T using leaf nodes edit distance, some of the results of the previous research concerning the dictionary matching problem will be used. The goal of the dictionary matching problem is to preprocess the dictionary, a finite set of words X, in order to locate words of X that occur in any given input word.

Crochemore at al. in [3] propose an algorithm for a direct construction of the deterministic dictionary matching automaton that recognizes a language A^*X and thus it can find all occurrences of words $x \in X$ in a given text. They have proven the automaton has $\mathcal{O}(\sum_{x \in X} |x|)$ states.

Later in [13], Melichar showed the equivalence of Crochemore's automaton to a finite automaton, that is created from a nondeterministic one, using standard determinisation algorithm based on a subset construction. The nondeterministic automaton has a tree-like structure with the self loop in the initial state for all symbols of the alphabet.

Moreover, it was shown that any acyclic automaton accepting language X can be transformed into a deterministic dictionary matching automaton accepting language A^*X by just adding the self loop in the initial state and using standard determinisation algorithm. It has been proven that the number of states of such created automaton in not greater than $\mathcal{O}(\sum_{x \in X} |x|)$.

The proposed nondeterministic automaton that finds all approximate occurrences of a tree pattern P to subtrees of an input tree T using leaf nodes edit distance can be viewed as a nondeterministic dictionary matching automaton for a dictionary $X(P, k)$. The finite language $X(P, k)$ was defined in the previous section as follows:

$$X(P,k) = \{\text{pref\_bar}(S) \,:\, \text{pref\_bar}(S) \in A^*, D_L(P,S) \leq k\}.$$

Therefore, the deterministic automaton M accepting language $A^*X(P,k)$, noted as $M(A^*X(P,k))$, has maximum of $\mathcal{O}(\sum_{x \in X(P,k)} |x|)$ states. Hence, the goal is to determine the size of the language $X(P,k)$ by finding the number of strings that represent prefix bar notations of trees created from the tree pattern P by using leaf nodes edit operations.

Theorem 13. *Given a tree pattern P ($|P| = m$), the number of strings, standing for prefix bar notations of trees, created from P by at most k rename node operations is $\mathcal{O}(|A|^k m^k)$.*

Proof. The set of strings created by exactly i rename node operations ($0 \leq i \leq k$) is made by replacing exactly i symbols of pref_bar(P) by other symbols. There are $\binom{m}{i}$ possibilities for choosing i symbols from pref_bar(P) and $|A| - 2$ possibilities for choosing the new symbol. Hence, the number of generated strings is at most $\binom{m}{i}(|A|-2)^i = \mathcal{O}(m^i)(|A|-2)^i = \mathcal{O}(|A|^i m^i)$. Therefore, the size of the set of strings created by at most k rename node operations is $\sum_{i=0}^{k} \mathcal{O}(|A|^i m^i) = \mathcal{O}(|A|^k m^k)$.

Theorem 14. *Given a tree pattern P ($|P| = m$), the number of strings, standing for prefix bar notations of trees, created from P by at most k delete leaf node operations is $\mathcal{O}(|\text{Leaves}(P)|^k)$.*

Proof. The set of strings created by exactly i delete leaf operations ($0 \leq i \leq k$) is made by deleting exactly $2i$ symbols from pref_bar(P). There are $\binom{|\text{Leaves}(P)|}{i} = \mathcal{O}(|\text{Leaves}(P)|^i)$ possibilities for choosing i leaf nodes from pref_bar(P). Therefore, the size of the set of strings created by at most k delete leaf operations can be specified as follows: $\sum_{i=0}^{k} \mathcal{O}(|\text{Leaves}(P)|^i) = \mathcal{O}(|\text{Leaves}(P)|^k)$.

Theorem 15. *Given a tree pattern P ($|P| = m$), the number of strings, standing for prefix bar notations of trees, created from P by at most k insert leaf node operations is $\mathcal{O}(|A|^k m^k)$.*

Proof. The number of strings created by exactly i insert leaf node operations can be transformed to the number of dipaths (directed paths) that can be found in a nondeterministic finite automaton constructed by Algorithm 1 with rename node and delete leaf nodes transitions removed. A dipath starts at the initial state and goes to some of final states. There are $2m$ steps needed to be done to the right (reading the pattern) and i steps down (insert operations, the two transitions representing an insert operation can be viewed as one step). Hence, every dipath is represented by a string containing $2m$ letters R (right) and i letters D (down). Moreover, the dipath needs to start and end with the letter R. The number of such strings is $\binom{2m+i-2}{i}$. Each letter D can represent $|A|-1$ inserted symbols, so the total number of strings created by exactly i insert leaf node operations is at most $\binom{2m+i-2}{i}(|A|-1)^i = \mathcal{O}(m^i|A|^i)$. Therefore, the size of the set of strings created by at most k insert leaf node operations is $\sum_{i=0}^{k} \mathcal{O}(|A|^i m^i) = \mathcal{O}(|A|^k m^k)$.

Theorem 16. *Given a tree pattern P ($|P| = m$), the number of strings, standing for prefix bar notations of trees, created from P by at most k operations node rename, leaf node insertion and leaf node deletion is $\mathcal{O}(|A|^k m^k)$.*

Proof. The number of such strings can be computed as follows:

$$\sum_{x=0}^{k}\sum_{y=0}^{k-x}\sum_{z=0}^{k-x-y} \underbrace{\mathcal{O}(|A|^x m^x)}_{\substack{\text{rename}\\ \text{x nodes}}} \underbrace{\mathcal{O}(|A|^y m^y)}_{\substack{\text{insert}\\ \text{y leaves}}} \underbrace{\mathcal{O}(|\text{Leaves}(P)|^z)}_{\substack{\text{delete}\\ \text{z leaves}}}$$

$$\sum_{x=0}^{k}\sum_{y=0}^{k-x}\sum_{z=0}^{k-x-y} \mathcal{O}(|A|^{x+y} m^{x+y} |\text{Leaves}(P)|^z)$$

$$\sum_{x=0}^{k}\sum_{y=0}^{k-x}\sum_{z=0}^{k-x-y} \mathcal{O}(|A|^{x+y} m^{x+y+z}) = \mathcal{O}(|A^k| m^k).$$

Theorem 17. *Let P be a tree pattern such that $|P| = m$ and k be the maximum number of errors allowed. The number of states of the deterministic finite automaton $M(A^*X(P,k))$ that is obtained using standard determinisation algorithm over NFA constructed by Algorithm 1 is $\mathcal{O}(|A|^k m^{k+1})$.*

Proof. As stated in [13], [3] the state complexity of this automaton is at most the same as the size of the language $X(P,k)$. Since $\forall x \in X(P,k) : |x| \leq 2m + 2k$, the size of the language $X(P,k)$ is

$$\mathcal{O}(\sum_{x \in X(P,k)} |x|) = \mathcal{O}((2m+2k)|A|^k m^k) = \mathcal{O}(|A|^k m^{k+1}).$$

7 Conclusion and Future Work

It was shown that finite automata can be used to solve constrained approximate subtree pattern matching problem. This is quite interesting since processing tree data structures usually requires a pushdown automaton as a model of computation. The proposed method creates a nondeterministic finite automaton for a given tree pattern P which is able to find all occurrences of a tree pattern P to subtrees of an input tree T with maximum distance (number of errors) k. The distance between a tree pattern P and subtrees of an input tree T is measured by minimal number of simple operations called leaf nodes edit operations.

The NFA can be turned into DFA by using the standard determinisation algorithm. The searching phase is afterwards performed in time $\mathcal{O}(n)$, where n is the number of nodes of an input tree T. In theory, the state complexity of the deterministic automaton can be exponential in the number of nodes of the tree pattern P. However, Section 6 gives the proof that the size of the proposed deterministic automaton is only $\mathcal{O}(|A|^k m^{k+1})$, where m is the number of nodes of the tree pattern P, k is the maximum number of errors and A is an alphabet containing all possible node labels. In practise the number of errors is expected to be a constant much smaller than the size of the pattern.

In Section 5, it was also shown how dynamic programming can be used to simulate the nondeterministic finite automaton. This approach has $\mathcal{O}(m)$ space complexity and $\mathcal{O}(mn)$ time complexity.

The proposed methods solve an approximate tree pattern matching subproblem since the operations used are a subset of general tree edit operations. Hence, the

techniques described here may also be relevant to other forms of approximate tree pattern matching problem, which we hope to explore in the future. Currently, we are working on the automata approach in respect to larger sets of edit operations allowed. In case of less restricted sets including general operations such as node insertion or node deletion the pushdown automata are required as models of computation.

Acknowledgements. This research has been partially supported by grant of CTU in Prague as project No. SGS17/209/OHK3/3T/18.

References

1. P. Bille: *A survey on tree edit distance and related problems.* Theoretical computer science, 337(1) 2005, pp. 217–239.
2. R. Cole and R. Hariharan: *Tree pattern matching to subset matching in linear time.* SIAM Journal on Computing, 32(4) 2003, pp. 1056–1066.
3. M. Crochemore and C. Hancart: *Automata for matching patterns*, in Handbook of formal languages, Springer, 1997, pp. 399–462.
4. E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann: *An optimal decomposition algorithm for tree edit distance.* ACM Trans. Algorithms, 6(1) Dec. 2009, pp. 2:1–2:19.
5. T. Flouri: *Pattern matching in tree structures*, PhD thesis, Czech Technical University, 2012.
6. C. M. Hoffmann and M. J. O'Donnell: *Pattern matching in trees.* Journal of the ACM (JACM), 29(1) 1982, pp. 68–95.
7. S. R. Kosaraju: *Efficient tree pattern matching*, in Foundations of Computer Science, 1989., 30th Annual Symposium on, IEEE, 1989, pp. 178–183.
8. L. Krčál: *Tree edit distance and approximate tree pattern matching problem.* Bachelor's thesis. Department of Computer Science and Engineering, Czech Technical University in Prague, Prague, Czech Republic, 2011.
9. J. Lahoda and J. Žďárek: *Simple tree pattern matching for trees in the prefix bar notation.* Discrete Applied Mathematics, 163 2014, pp. 343–351.
10. S. Y. Lu: *A tree-to-tree distance and its application to cluster analysis.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 1979, pp. 219–224.
11. F. Luccio and L. Pagli: *Simple solutions for approximate tree matching problems*, in Colloquium on Trees in Algebra and Programming, Springer, 1991, pp. 193–201.
12. B. Melichar: *Approximate string matching by finite automata*, in International Conference on Computer Analysis of Images and Patterns, Springer, 1995, pp. 342–349.
13. B. Melichar, J. Holub, and T. Polcar: *Text searching algorithms.* Available on: http://stringology.org/athens, 2005.
14. S. M. Selkow: *The tree-to-tree editing problem.* Information processing letters, 6(6) 1977, pp. 184–186.
15. J. Stoklasa, J. Janoušek, and B. Melichar: *Subtree pushdown automata for trees in bar notation, 2010.* London Stringology Days, 2010.
16. M. Svoboda and I. Holubová: *Refinement correction strategy for invalid XML documents and regular tree grammars*, in International Conference on Database and Expert Systems Applications, Springer, 2014, pp. 308–316.
17. K. C. Tai: *The tree-to-tree correction problem.* Journal of the ACM (JACM), 26(3) 1979, pp. 422–433.
18. K. Zhang, D. Shasha, and J. T. L. Wang: *Approximate tree matching in the presence of variable length don't cares.* Journal of Algorithms, 16(1) 1994, pp. 33–66.

Right-to-left Online Construction of Parameterized Position Heaps

Noriki Fujisato, Yuto Nakashima, Shunsuke Inenaga,
Hideo Bannai, and Masayuki Takeda

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

Abstract. Two strings of equal length are said to *parameterized match* if there is a bijection that maps the characters of one string to those of the other string, so that two strings become identical. The parameterized pattern matching problem is, given two strings T and P, to find the occurrences of substrings in T that parameterized match P. Diptarama et al. [Position Heaps for Parameterized Strings, CPM 2017] proposed an indexing data structure called *parameterized position heaps*, and gave a left-to-right online construction algorithm. In this paper, we present a *right-to-left* online construction algorithm for parameterized position heaps. For a text string T of length n over two kinds of alphabets Σ and Π of respective size σ and π, our construction algorithm runs in $O(n \log(\sigma+\pi))$ time with $O(n)$ space. Our right-to-left parameterized position heaps support pattern matching queries in $O(m \log(\sigma+\pi)+m\pi+pocc))$ time, where m is the length of a query pattern P and $pocc$ is the number of occurrences to report. Our construction and pattern matching algorithms are as efficient as Diptarama et al.'s algorithms.

1 Introduction

Text indexing is the task to preprocess the text string so that subsequent pattern matching queries can be answered efficiently. To date, a numerous number of text indexing structure for exact pattern matching have been proposed, ranging from classical data structures such as suffix trees [14], directed acyclic word graphs [2,3], and suffix arrays [10], to more advanced ones such as compressed suffix arrays [8] and FM index [7], just to mention a few.

Ehrenfeucht et al. [6] proposed a text indexing structure called *position heaps*. Ehrenfeucht et al.'s position heap is constructed in a *right-to-left* online manner, where a new node is incrementally inserted to the current position heap for each decreasing position $i = n, \ldots, 1$ in the input string T of length n. In other words, Ehrenfeucht et al.'s position heap is defined over a sequence $\langle \varepsilon, T[n..], \ldots, T[1..] \rangle$ of the suffixes of T in increasing order of their length, where ε is the empty string of length 0. Kucherov [9] proposed another variant of position heaps. Kucherov's position heap is constructed in a *left-to-right* online manner, where a new node is incrementally inserted to the current position heap for each increasing $i = 1, \ldots, n$. In other words, Kucherov's position heap is defined over a sequence $\langle T[1..], \ldots, T[n..], \varepsilon \rangle$ of the suffixes of T in decreasing order of their length. We will call Ehrenfeucht et al.'s position heap as the RL position heap, and Kucherov's position heap as the LR position heap. Both of the RL and LR position heaps for a text string T of length n require $O(n)$ space and can be constructed in $O(n \log \sigma)$ time, where σ is the alphabet size. By augmenting the RL and LR position heaps of T with auxiliary links called maximal reach pointers, pattern matching queries can be answered in $O(m \log \sigma + occ)$ time, where m is the length of a query pattern P and occ is the number of occurrences of P in T.

Noriki Fujisato, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda: Right-to-left Online Construction of Parameterized Position Heaps, pp. 91–102.
Proceedings of PSC 2018, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06484-9

Nakashima et al. [12] proposed position heaps for a set of strings that is given as a reversed trie, and proposed an algorithm that constructs the position heap of a given trie in $O(\sigma N)$ time and space, where N is the size of the input trie. Later, the same authors showed how to construct the position heap of a trie in $O(N)$ time and space, for integer alphabets of size polynomialy bounded in N [13].

Baker [1] introduced the *parameterized pattern matching* problem, that seeks for the occurrences of substrings of the text T that have the "same" structures as the given pattern P. Parameterized pattern matching is motivated by e.g., software maintenance and plagiarism detection [1]. More formally, we consider two distinct alphabets Σ and Π, and we call an element over $\Sigma \cup \Pi$ a p-string. The parameterized pattern matching problem is, given two p-strings T and P, to find all occurrences of substrings X of T that can be transformed to P by a bijection from $\Sigma \cup \Pi$ to $\Sigma \cup \Pi$ which is identity for Σ. For instance, if $T = \texttt{abzaxxbyaxxbazzax}$ and $P = \texttt{yazzbx}$ where $\Sigma = \{\texttt{a}, \texttt{b}\}$ and $\Pi = \{\texttt{x}, \texttt{y}, \texttt{z}\}$, then the positions to output are 3 and 8. To see why, observe that for the substring $T[3..8] = \texttt{zaxxby}$ there is a bijection $\texttt{z} \to \texttt{y}$, $\texttt{a} \to \texttt{a}$, $\texttt{x} \to \texttt{z}$, $\texttt{b} \to \texttt{b}$, and $\texttt{x} \to \texttt{y}$ that maps the substring to P. Also, observe that for the other substring $T[8..13] = \texttt{yaxxbz}$, there is a bijection $\texttt{y} \to \texttt{y}$, $\texttt{a} \to \texttt{a}$, $\texttt{x} \to \texttt{z}$, $\texttt{b} \to \texttt{b}$, and $\texttt{z} \to \texttt{x}$ that maps the substring to P as well.

Of various algorithms and indexing structures for the parameterized pattern matching (see [11] for a survey), we focus on Diptarama et al.'s *parameterized position heaps* [5]. Diptarama et al.'s *parameterized position heaps* are based on Kucherov's LR position heaps, which are constructed in a *left-to-right* online manner. Let us call their structure the *LR p-position heaps*. Diptarama et al. showed how to construct the LR p-position heap for a given text of length n in $O(n \log(\sigma + \pi))$ time with $O(n)$ space, where $\sigma = |\Sigma|$ and $\pi = |\Pi|$. They also showed that the LR p-position heap augmented with maximal reach pointers can support parameterized pattern matching queries in $O(m \log(\sigma + \pi) + m\pi + pocc)$ time, where $pocc$ is the number of occurrences to report.

In this paper, we propose *RL p-position heaps* which are constructed in a *right-to-left* online manner. We show how to construct our RL position heap for a given text string T of length n in $O(n \log(\sigma + \pi))$ time with $O(n)$ space. Our construction algorithm is based on Ehrenfeucht et al.'s construction algorithm for RL position heaps [6], and Weiner's suffix tree construction algorithm [14]. Namely, we use reversed suffix links defined for the nodes of RL p-position heaps. The key to our algorithm is how to label the reversed suffix links, which will be clarified in Definition 5. Using our RL p-position heap augmented with maximal reach pointers, one can perform parameterized pattern matching queries in $O(m \log(\sigma + \pi) + m\pi + pocc)$ time.

2 Preliminaries

2.1 Notations on strings

Let Σ and Π be disjoint sets called a *static alphabet* and a *parameterized alphabet*, respectively. Let $\sigma = |\Sigma|$ and $\pi = |\Pi|$. An element of Σ is called an *s-character*, and that of Π is called a *p-character*. In the sequel, both an s-character and a p-character are sometimes simply called a *character*. An element of Σ^* is called a *string*, and an element of $(\Sigma \cup \Pi)^*$ is called a *p-string*. The length of a (p-)string S is the number of characters contained in S. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a (p-)string $S = XYZ$, X, Y and Z are called a *prefix*, *substring*, and *suffix* of w,

respectively. The set of prefixes, substrings, and suffixes of a (p-)string S is denoted by $\mathsf{Prefix}(S)$, $\mathsf{Substr}(S)$, and $\mathsf{Suffix}(S)$, respectively. The i-th character of a (p-)string S is denoted by $S[i]$ for $1 \leq i \leq |S|$, and the substring of a (p-)string S that begins at position i and ends at position j is denoted by $S[i..j]$ for $1 \leq i \leq j \leq |S|$. For convenience, let $S[i..j] = \varepsilon$ if $j < i$. Also, let $S[i..] = S[i..|S|]$ for any $1 \leq i \leq |S|$.

2.2 Parameterized pattern matching

For any p-string X and $f : (\Sigma \cup \Pi) \to (\Sigma \cup \Pi)$, let $F(X) = f(X[1]) \cdots f(X[|X|])$. Two p-strings X and Y of length k each are said to *parameterized match* (*p-match*) iff there is a bijection f on $\Sigma \cup \Pi$ such that $f(a) = a$ for any $a \in \Sigma$ and $f(X[i]) = Y[i]$ for all $1 \leq i \leq k$. For instance, if $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $\Pi = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$, then $X = \mathtt{axbzzayx}$ and $Y = \mathtt{azbyyaxz}$ p-match since there is a bijection f such that $f(\mathtt{a}) = \mathtt{a}$, $f(\mathtt{b}) = \mathtt{b}$, $f(\mathtt{x}) = \mathtt{z}$, $f(\mathtt{y}) = \mathtt{x}$, and $f(\mathtt{z}) = \mathtt{y}$ and $F(X) = F(\mathtt{axbzzayx}) = \mathtt{azbyyaxz} = Y$. We write $X \approx Y$ iff X and Y p-match.

The *previous encoding* $\mathsf{prev}(S)$ of a p-string S of length n is a sequence of length n such that the first occurrence of each p-character x is replaced with 0 and any other occurrence of x is replaced by the distance to the previous occurrence of x in S, and each s-character remains the same. More formally, $\mathsf{prev}(S)$ is a sequence over $\Sigma \cup [0..n-1]$ of length n such that for each $1 \leq i \leq n$,

$$\mathsf{prev}(S)[i] = \begin{cases} S[i] & \text{if } S[i] \in \Sigma, \\ 0 & \text{if } S[i] \in \Pi \text{ and } S[i] \neq S[j] \text{ for any } 1 \leq j < i, \\ i - j & \text{if } S[i] \in \Pi, S[i] = S[j] \text{ and } S[i] \neq S[k] \text{ for any } j < k < i. \end{cases}$$

Observe that $X \approx Y$ iff $\mathsf{prev}(X) = \mathsf{prev}(Y)$. Using the same example as above, we have that $\mathsf{prev}(\mathtt{axbzzayx}) = \mathsf{prev}(\mathtt{azbyyaxz}) = \mathtt{a0b01a06}$.

Let T and P be p-strings of length n and m, respectively, where $n \geq m$. The *parameterized pattern matching* problem is to find all positions i in T such that $T[i..i+m-1] \approx P$.

3 Parameterized position heaps

Let $\mathcal{S} = \langle S_1, \ldots, S_k \rangle$ be a sequence of strings such that for any $1 < i \leq k$, $S_i \notin \mathsf{Prefix}(S_j)$ for any $1 \leq j < i$. For convenience, we assume that $S_1 = \varepsilon$.

Definition 1 (Sequence hash trees [4]). *The* sequence hash tree *of a sequence* $\mathcal{S} = \langle S_1, \ldots, S_k \rangle$ *of strings, denoted* $\mathsf{SHT}(\mathcal{S})$*, is a trie structure that is recursively defined as follows: Let* $\mathsf{SHT}(\mathcal{S})^i = (V_i, E_i)$*. Then*

$$\mathsf{SHT}(S)^i = \begin{cases} (\{\varepsilon\}, \emptyset) & \text{if } i = 1, \\ (V_{i-1} \cup \{p_i\}, E_{i-1} \cup \{(q_i, c, p_i)\}) & \text{if } 1 \leq i \leq k, \end{cases}$$

where q_i *is the longest prefix of* S_i *which satisfies* $q_i \in V_{i-1}$*,* $c = S_i[|q_i| + 1]$*, and* p_i *is the shortest prefix of* S_i *which satisfies* $p_i \notin V_{i-1}$*.*

Note that since we have assumed that each $S_i \in \mathcal{S}$ is not a prefix of S_j for any $1 \leq j < i$, the new node p_i and new edge (q_i, c, p_i) always exist for each $1 \leq i \leq k$. Clearly $\mathsf{SHT}(\mathcal{S})$ contains k nodes (including the root).

prev(T[17..])	0
prev(T[16..])	00
prev(T[15..])	a00
prev(T[14..])	0a03
prev(T[13..])	00a33
prev(T[12..])	000a33
prev(T[11..])	0100a33
prev(T[10..])	00104a33
prev(T[9..])	010104a33
prev(T[8..])	0013104a33
prev(T[7..])	01013104a33
prev(T[6..])	001313104a33
prev(T[5..])	0101313104a33
prev(T[4..])	00131313104a33
prev(T[3..])	002131313104a33
prev(T[2..])	0022131313104a33
prev(T[1..])	a0022131313104a33

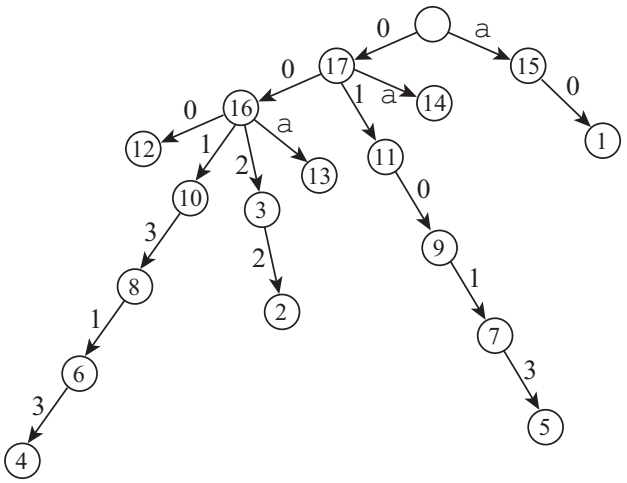


Figure 1. To the left is the list of prev(T[i..]) for p-string T = axyxyyxxyyxxzyazy of length 17, where $\Sigma = \{\mathtt{a}\}$ and $\Pi = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$. To the right is an illustration for PPH(T). The underlined prefix of each prev(T[i..]) in the left list denotes the longest prefix of prev(T[i..]) that was inserted to PPH(T[$i+1$..]) and hence, the node with id i represents this underlined prefix of prev(T[i..]).

In what follows, we will define our indexing data structure for a text p-string T of length n. Let $\mathcal{P}_T = \langle \varepsilon, \mathsf{prev}(T[n..]), \ldots, \mathsf{prev}(T[1..]) \rangle$ be the sequence of previous encoded suffixes of T arranged in *increasing* order of their length. It is clear that $\mathsf{prev}(T[i..]) \notin \mathsf{Prefix}(\mathsf{prev}(T[j..]))$ for any $1 \leq j < i$ and $\mathsf{prev}(T[i..]) \notin \mathsf{Prefix}(\varepsilon)$ for any $1 \leq i \leq n$. Hence we can naturally define the sequence hash tree for $\mathcal{P}_T$, and we obtain our data structure:

Definition 2 (Parameterized positions heaps). *The* parameterized position heap *(*p-position heap*) for a p-string T, denoted* PPH(T)*, is the sequence hash tree of $\mathcal{P}_T$ i.e.,* PPH(T) = SHT($\mathcal{P}_T$).

See Figure 1 for an example of our p-position heap.

Note that we can obtain $\mathcal{P}_{T[i-1..]}$ by adding $\mathsf{prev}(T[i-1..])$ at the beginning of $\mathcal{P}_{T[i..]}$. This also means that $\mathsf{PPH}(T[i..]) = \mathsf{SHT}(\mathcal{P}_{T[i..]})$ for each $1 \leq i \leq n$. Hence, we can construct PPH(T) by processing the input string T from right to left. We remark that we can easily compute $\mathsf{prev}(T[i-1..])$ from $\mathsf{prev}(T[i..])$ in a total of $O(n \log \pi)$ time for all $2 \leq i \leq n$ using $O(\min\{\pi, n\})$ extra space, e.g., by maintaining a balanced search tree that stores the distinct p-characters that have occurred in $T[i..]$ and records the leftmost occurrences of these p-character in the nodes.

Diptarama et al. [5] proposed another version of parameterized position heap for a sequence of previous encoded suffixes of the input p-string T arranged in *decreasing* order of their length. Since their algorithm processes T from left to right, we sometimes call their structure as a *left-to-right p-position heap* (*LR p-position heap*), while we call our PPH(T) as a *right-to-left p-position heap* (*RL p-position heap*) since our construction algorithm processes T from right to left.

For any p-string $P \in (\Sigma \cup [0..n-1])^+$, we say that P is *represented* by PPH(T) iff PPH(T) has a path which starts from the root and spells out P.

Lemma 3. *For any string T of length n,* PPH(T) *consists of exactly $n+1$ nodes. Also, there is a one-to-one correspondence between the positions $1, \ldots, n$ in T and the non-root nodes of* PPH(T).

Proof. Initially, $\mathsf{PPH}(\varepsilon)$ consists only of the root that represents ε. For each $1 \leq i \leq n$, since $|\mathsf{prev}(T[i..])| = n - i + 1 > n - j + 1 = |\mathsf{prev}(T[j..])|$ for any $1 \leq i < j \leq n$, it is clear that there is a prefix of $\mathsf{prev}(T[i..])$ that is not represented by $\mathsf{PPH}(T[i+1..])$. Therefore, when we construct $\mathsf{PPH}(T[i..])$ from $\mathsf{PPH}(T[i+1..])$, then exactly one node is inserted, which corresponds to position i. □

Let V be the set nodes of $\mathsf{PPH}(T)$. Based on Lemma 3, we define a bijection $\mathsf{id} : V \to [0..n]$ such that $\mathsf{id}(r) = 0$ for the root r and $\mathsf{id}(v) = i$ iff v was the node that was inserted when constructing $\mathsf{PPH}(T[i..])$ from $\mathsf{PPH}(T[i+1..])$.

Unlike our RL p-position heap, Diptarama et al.'s LR p-position heap can have *double nodes* to which two positions of the text p-string are associated.

We remark that the pattern matching algorithm of Diptarama et al. [5] can be applied to our RL p-position heap $\mathsf{PPH}(T)$ for a text p-string T, and this way one can solve the parameterized pattern matching problem in $O(m \log(\sigma + \pi) + m\pi + occ)$ time, where occ is the number of positions in text T such that the pattern p-string P of length m and the corresponding substring $T[i..i+m-1]$ p-match. We note that since our RL p-position heap does not have double nodes, the pattern matching algorithm can be somewhat simplified.

The following lemma is an analogue to Lemma 6 of [5] for Diptarama et al.'s LR p-position heap.

Lemma 4. *For any* $1 \leq i \leq j \leq n$ *if* $\mathsf{prev}(T[i..j])$ *is represented by* $\mathsf{PPH}(T)$*, then for any substring* X *of* $T[i..j]$*,* $\mathsf{prev}(X)$ *is represented by* $\mathsf{PPH}(T)$*.*

Proof. The lemma can be shown in a similar way to Lemma 6 of [5]. For the sake of completeness, we provide a full proof below.

First, we show that for any proper prefix $T[i..i+k]$ of $T[i..j]$ with $0 \leq k < j - i$, $\mathsf{prev}(T[i..i+k])$ is represented by $\mathsf{PPH}(T)$. It follows from the definition of previous encoding that $\mathsf{prev}(T[i..i+k]) = \mathsf{prev}(T[i..j])[1..k+1]$, and hence $\mathsf{prev}(T[i..i+k])$ is a prefix of $\mathsf{prev}(T[i..j])$. Since $\mathsf{prev}(T[i..j])$ is represented by $\mathsf{PPH}(T)$ and $i \leq i+k < j$, $\mathsf{prev}(T[i..i+k])$ is also represented by $\mathsf{PPH}(T)$.

Now it suffices for us to show that for any proper suffix $T[i+h..j]$ of $T[i..j]$ with $0 < h \leq j - i$, $\mathsf{prev}(T[i+h..j])$ is represented by $\mathsf{PPH}(T)$, since then we can inductively apply the above discussion for the prefixes. By the above discussions for the prefixes of $T[i..j]$, there exist positions $i = b_{j-i} < \cdots < b_0 \leq n$ in T such that $\mathsf{prev}(T[i..i+k]) = \mathsf{prev}(T[b_k..b_k+k])$ for $0 \leq k \leq j - i$. By the definition of $\mathsf{PPH}(T)$, the root has an out-going edge labeled by $\mathsf{prev}(T[b_1+1..b_1+1])$, and this is the base case for our induction. Since $\mathsf{prev}(T[i..i+k]) = \mathsf{prev}(T[b_k..b_k+k])$, we have $\mathsf{prev}(T[i+1..i+k]) = \mathsf{prev}(T[b_k+1..b_k+k])$. Now since $\mathsf{prev}(T[b_{k+1}+1..b_{k+1}+k+1]) = \mathsf{prev}(T[i+1..i+k+1])$ and $\mathsf{prev}(T[b_k+1..b_k+k]) = \mathsf{prev}(T[i+1..i+k])$, $\mathsf{prev}(T[b_k+1..b_k+k])$ is a prefix of $\mathsf{prev}(T[b_{k+1}+1..b_{k+1}+k+1])$. This implies that if $\mathsf{prev}(T[b_k+1..b_k+k])$ is represented by $\mathsf{PPH}(T)$, then $\mathsf{prev}(T[b_{k+1}+1..b_{k+1}+(k+1)])$ is also represented by $\mathsf{PPH}(T)$. By induction, we have that $\mathsf{prev}(T[b_{j-i}+1..b_{j-i}+j-i]) = \mathsf{prev}(T[i+1..j])$ is represented by $\mathsf{PPH}(T)$. Applying the same argument inductively, it is immediate that $\mathsf{prev}(T[i+h...j])$ with $2 \leq h \leq j - i$ are also represented by $\mathsf{PPH}(T)$. □

In the next section, we show how to construct our RL p-position heap $\mathsf{PPH}(T)$ for an input text p-string T of length n in $O(n \log(\sigma + \pi))$ time and $O(n)$ space.

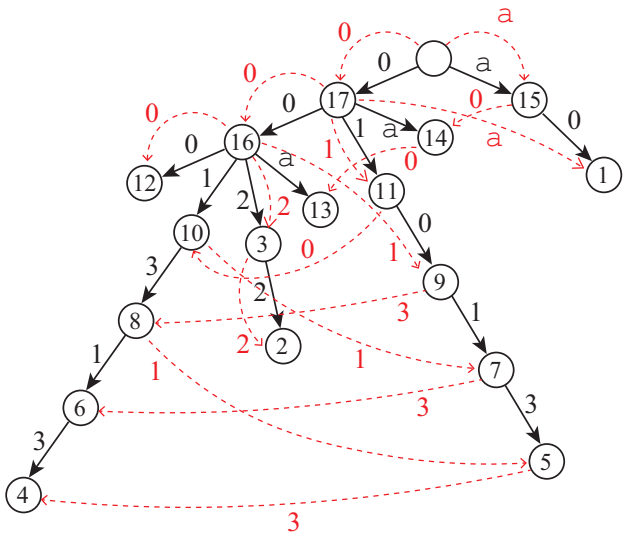


Figure 2. Illustration of the reversed suffix links of $\mathsf{PPH}(T)$ with the same p-string $T =$ `axyxyyxxyyxxzyazy` as in Figure 1. The reversed suffix links and their labels are shown in red.

4 Right to left construction of parameterized position heaps

In this section, we present our algorithm which constructs $\mathsf{PPH}(T)$ of a given p-string T in a right-to-left online manner. The key to our construction algorithm is the use of *reversed suffix links*, which will be defined in the following subsection.

4.1 Reversed suffix links

For convenience, we will sometimes identify each node v of $\mathsf{PPH}(T)$ with the path label from the root to v. In our right-to-left online construction of $\mathsf{PPH}(T)$, we use the *reversed suffix links*, which are a generalization of the *Weiner links* that are used in right-to-left construction of the suffix tree [14] for (standard) string matching:

Definition 5 (Reversed suffix links). *For any node v of $\mathsf{PPH}(T)$ and a character $a \in \Sigma \cup [0..n-1]$, let*

$$\mathsf{rsl}(a, v) = \begin{cases} av & \text{if } a \in \Sigma \cup \{0\} \text{ and } av \text{ is represented by } \mathsf{PPH}(T), \\ u & \text{if } a \in [1..n-1],\ v[a] = 0 \text{ and} \\ & u = 0v[1..a-1]av[a+1..|v|] \text{ is represented by } \mathsf{PPH}(T), \\ \textit{undefined} & \text{otherwise.} \end{cases}$$

It is clear that by taking one rsl link from a node, then the node depth (and hence the string length) increases exactly one.

Observe that the first case of of the definition of $\mathsf{rsl}(a, v)$ is a direct extension of the Weiner links, where $\mathsf{rsl}(a, v)$ points to the node av that is obtained by prepending a to v. The second case, however, is a special case that arises in parameterized pattern matching. The following lemma ensures that our reversed suffix links rsl are well defined:

Lemma 6. *For any node v in $\mathsf{PPH}(T)$ and a character $a \in \Sigma \cup [0..n-1]$, let $\mathsf{rsl}(a, v) = u$, where u is a node of $\mathsf{PPH}(T)$. Then, for any string X such that $\mathsf{prev}(X) = u$, $\mathsf{prev}(X[2..|X|]) = v$.*

Proof. In the first case of the definition of $\mathsf{rsl}(a, v)$ where $a \in \Sigma \cup \{0\}$, we have $\mathsf{prev}(X) = u = av$. Hence, $\mathsf{prev}(X[2..|X|]) = \mathsf{prev}(X)[2..|X|] = u[2..|u|] = v$.

In the second case of the definition of $\mathsf{rsl}(a, v)$ where $a \in [1..n-1]$, we have $\mathsf{prev}(X) = u = 0v[1..a-1]av[a+1..|v|]$, which implies that $X[1] = X[a+1]$ and $X[1] \neq X[i]$ for any $2 \leq i \leq a$. Thus, $\mathsf{prev}(X[2..|X|]) = v[1..a-1]0v[a+1..|v|] = v$. □

The next proposition shows that there is a monotonicity in the labels of the reversed suffix links that come from the nodes in the same path of $\mathsf{PPH}(T)$.

Proposition 7. *Suppose there is a reversed suffix link* $\mathsf{rsl}(a, v)$ *of a node* v *with* $a \in \Sigma \cup [0..n-1]$. *Let* u *be any ancestor of* v. *Then, if* $a \in \Sigma \cup \{0\}$, u *has a reversed suffix link* $\mathsf{rsl}(a, u)$. *Also, if* $a \in [1..n-1]$ *and* $|u| \geq a$, *then* u *has a reversed suffix link* $\mathsf{rsl}(a, u)$, *and if* $a \in [1..n-1]$ *and* $|u| < a$, *then* u *has a reversed suffix link* $\mathsf{rsl}(0, u)$.

Proof. It suffices for us to show that the lemma holds for the parent v' of v, since then the lemma inductively holds for any ancestor of v. Note that $v' = v[1..|v|-1]$. Let $w = \mathsf{rsl}(a, v)$.

If $a \in \Sigma \cup \{0\}$, then $w = av$. Hence, the parent of w is $w[1..|w|-1] = av[1..|v|-1] = av'$. Therefore, there is a reversed suffix link $\mathsf{rsl}(a, v')$.

If $a \in [1..n-1]$ and $|v'| = |v|-1 \geq a$, then it follows from the definition of $\mathsf{rsl}(a, v)$ that $v[a] = 0$ and $w = 0v[1..a-1]av[a+1..|v|]$. Since $|v'| \geq a$, we have that $v'[a] = 0$ and $|v| \geq a+1$. Thus $w[1..|w|-1] = 0v[1..a-1]av[a+1..|v|-1]$ is represented by $\mathsf{PPH}(T)$. Consequently, there is a reversed suffix link $\mathsf{rsl}(a, v')$.

If $a \in [1..n-1]$ and $|v'| = |v|-1 = a-1$, then it follows from the definition of $\mathsf{rsl}(a, v)$ that $v[a] = v[|v|] = 0$ and $w = 0v[1..|v|-1]a$. Thus $w[1..|w|-1] = 0v[1..|v|-1] = 0v'$ is represented by $\mathsf{PPH}(T)$. Consequently, there is a reversed suffix link $\mathsf{rsl}(a, v')$. □

4.2 Adding a new node

Our algorithm processes a given p-string T of length n from right to left and maintains $\mathsf{PPH}(T[i..])$ in decreasing order of $i = n, \ldots, 1$. Initially, we begin with $\mathsf{PPH}(\varepsilon)$ which consists of the root r representing the empty string ε. For convenience, we use an auxiliary node $\perp$ as a parent of the root r, and create reversed suffix links $\mathsf{rsl}(a, \perp) = r$ for every $a \in \Sigma \cup \{0\}$.

Now suppose we have constructed $\mathsf{PPH}(T[i..])$ for $1 < i \leq n$, and we will update it to $\mathsf{PPH}(T[i-1..])$. In so doing, we begin with node v_i such that $\mathsf{id}(v_i) = i$. We know the locus of this node v_i since v_i is the node that was inserted at the last step when $\mathsf{PPH}(T[i..])$ was constructed from $\mathsf{PPH}(T[i+1..])$. Note also that this node v_i is a leaf in $\mathsf{PPH}(T[i..])$. We climb up the path from v_i until finding its lowest ancestor v'_i that satisfies the following. There are three cases:

1. If $T[i-1] \in \Sigma$, then v'_i is the lowest ancestor of v_i such that $\mathsf{rsl}(T[i-1], v_i)$ is defined.
2. If $T[i-1] \in \Pi$ and $T[i-1] \neq T[j]$ for any $i \leq j \leq n$, then v'_i is the lowest ancestor of v_i such that $\mathsf{rsl}(0, v_i)$ is defined.
3. Otherwise, let $d = j - i$ where j is the smallest position such that $i \leq j \leq n$ and $T[i-1] = T[j]$. Then v'_i is the lowest ancestor of v_i such that $\mathsf{rsl}(d, v'_i)$ is defined if it exists, and v'_i is the lowest ancestor of v_i such that $\mathsf{rsl}(0, v'_i)$ is defined otherwise.

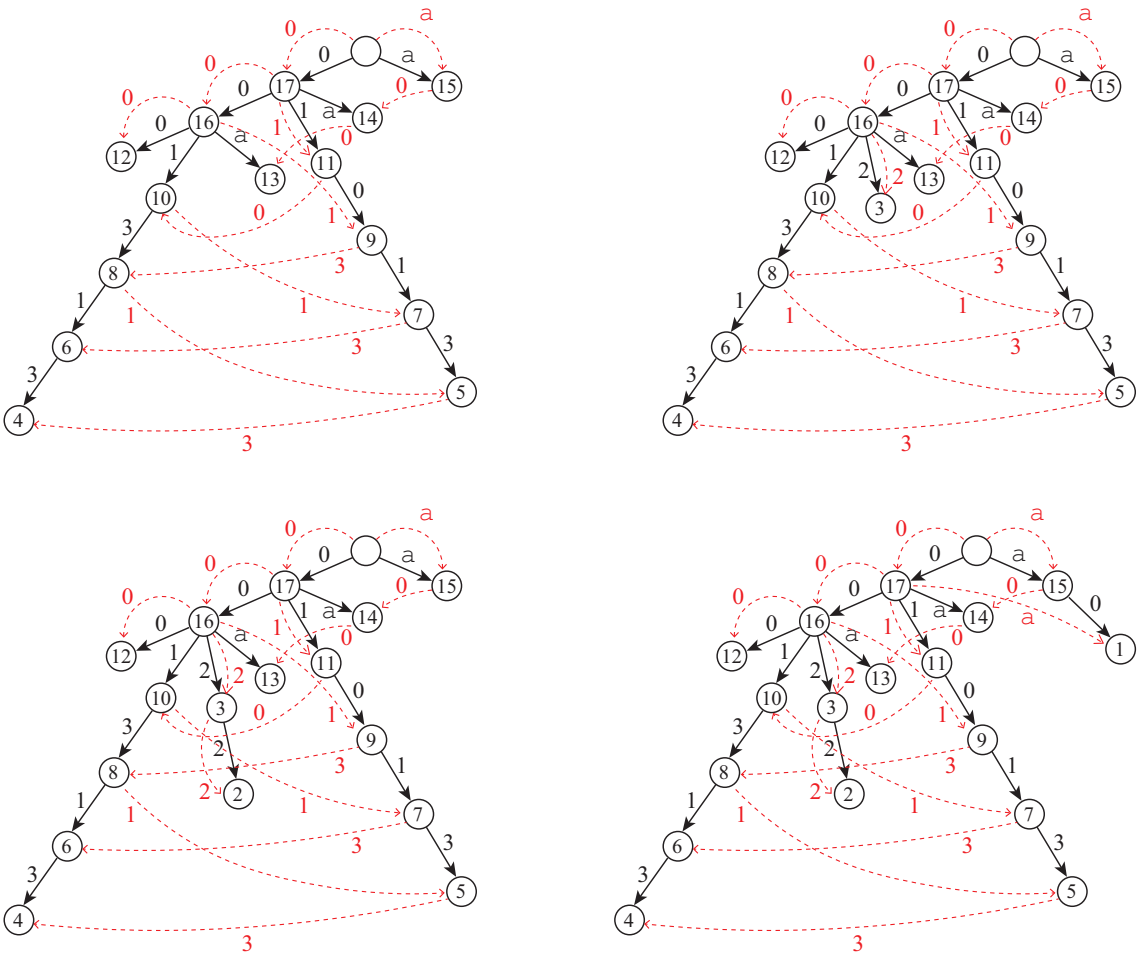


Figure 3. A snapshot of updating $\mathsf{PPH}(T[i..])$ for $i = 4, 3, 2, 1$ with the same p-string $T =$ axyxyyxxyyxxzyazy as in Figures 1 and 2. First, we update $\mathsf{PPH}(T[4..])$ (upper left) to $\mathsf{PPH}(T[3..])$ (upper right). Since $T[3] = T[5] =$ y and $d = 5 - 3 = 2$, we first try to find the lowest ancestor of the node with id 4 that has a reversed suffix link labeled with $d = 2$ by climbing up the path. However, it does not exist, and then we arrive at the lowest ancestor with id 17 whose depth is 1 (< 2). Hence the second sub-case of Case 3 is applied, and using its reversed suffix link we move to the node with id 16. The new node with id 3 is inserted as its child. Next, we update $\mathsf{PPH}(T[3..])$ (upper right) to $\mathsf{PPH}(T[2..])$ (lower left). Since $T[2] = T[4] =$ x and $d = 4 - 2 = 2$, we first try to find the lowest ancestor of the node with id 3 that has a reversed suffix link labeled with $d = 2$ by climbing up the path, and we arrive at the node with id 16. Hence the first sub-case of Case 3 is applied, and using its reversed suffix link we move to the node with id 3. The new node with id 2 is inserted as its child. Finally, we update $\mathsf{PPH}(T[2..])$ (lower left) to $\mathsf{PPH}(T[1..])$ (lower right). Since $T[1] =$ a $\in \Sigma$, Case 1 is applied. Thus we try to find the lowest ancestor of the node with id 2 that has a reversed suffix link labeled with a by climbing up the path, and we arrive at the root. Using its reversed suffix link, we move to the node with id 15. The new node with id 1 is inserted as its child.

Let u_i be the node of $\mathsf{PPH}(T[i..])$ that is pointed by the reversed suffix link of v'_i as above. Then, we create a new node v_{i-1} as a child of u_i such that $\mathsf{id}(v_{i-1}) = i - 1$. The new edge (u_i, v_{i-1}) is labeled by $\mathsf{prev}(T[i - 1..])[|u_i| + 1]$. We repeat the above procedure for all positions i in T in decreasing order. See also Figure 3 for concrete examples.

Lemma 8. *The above algorithm correctly updates* $\mathsf{PPH}(T[i..])$ *to* $\mathsf{PPH}(T[i - 1..])$.

Proof. Note that v_i and v'_i are prefixes of $\mathsf{prev}(T[i..])$. Let a be the character in $\Sigma \cup [0..n-1]$ that is used in the reversed suffix link as above.

In Cases 1 and 2 above, we have $a = T[i-1] \in \Sigma$ or $a = 0$. Then it is clear that av'_i is a prefix of $\mathsf{prev}(T[i-1..])$. Since v'_i is the lowest ancestor of v_i for which $\mathsf{rsl}(a, v'_i)$ is defined, $u_i = av'_i$ is the longest prefix of $\mathsf{prev}(T[i-1..])$ that is represented by $\mathsf{PPH}(T[i..])$. Hence, the new node v_{i-1} and its incoming edge labeled by $\mathsf{prev}(T[i-1..])[|u_i|+1]$ are correctly inserted.

Consider Case 3 above. We first try to find v'_i in the first sub-case, where $a = d \geq 1$. If it exists, then v'_i is the lowest ancestor of v_i such that $\mathsf{rsl}(d, v'_i)$ is defined, and thus $\mathsf{rsl}(d, v'_i) = 0v'_i[1..d-1]dv'_i[d+1..|v'_i|]$. It now follows from Lemma 4 that $u_i = 0v'_i[1..d-1]dv'_i[d+1..|v'_i|]$ is the longest prefix of $\mathsf{prev}(T[i-1..])$ that is represented by $\mathsf{PPH}(T[i..])$. Hence, the new node v_{i-1} and its incoming edge labeled by $\mathsf{prev}(T[i-1..])[|u_i|+1]$ are correctly inserted in this sub-case. It is clear that v'_i in the first sub-case is at least of depth d. Hence, if we arrive at the ancestor of v_i of depth $d-1$ without encountering the lowest ancestor satisfying the condition of the first sub-case, then we try to find the lowest ancestor of v_i that has a reversed suffix link labeled by 0 (second sub-case). Thus, by a similar argument to Case 2, the new node v_{i-1} its incoming edge labeled by $\mathsf{prev}(T[i-1..])[|u_i|+1]$ are correctly inserted in this second sub-case. □

4.3 Adding a new reversed suffix link

After inserting the new node v_{i-1}, we need to maintain the reversed suffix links corresponding to v_{i-1}.

Lemma 9. *There is exactly one reversed suffix link that points to the new node v_{i-1} in $\mathsf{PPH}(T[i-1..])$. Moreover, this reversed suffix link comes from the ancestor of v_i of depth $|v'_i|+1$.*

Proof. Suppose on the contrary that there are two distinct nodes x and y each of which has a reversed suffix link pointing to v_{i-1}. The label of any reversed suffix link that points to v_{i-1} is uniquely determined by the path label from the root to v_{i-1}. Therefore, the reversed suffix links of x and y that point to v_{i-1} are both labeled by the same symbol. This means that $x = y$, however, this contradicts the definition of the p-position heap. Hence, there is at most one node which has a reversed suffix link that points to v_{i-1}.

Let z_i be the ancestor of v_i of depth $|v'_i|+1$. Also, let $x = (T[i..])[|u_i|] = (T[i-1..])[|u_i|+1] = T[i+|u_i|-1]$, namely, x is the text character that corresponds to the label of the edge (v'_i, z_i) that is on the path from the root to v_i, and to the label of the new edge (u_i, v_{i-1}). If $x \in \Pi$ and $i+|u_i|-1$ is the smallest position in $T[i-1..]$ such that $T[i-1] = T[i+|u_i|-1]$, then (v'_i, z_i) is labeled with 0 while (u_i, v_{i-1}) is labeled with $|u_i|$. Otherwise, the label of the new edge (u_i, v_{i-1}) must be equal to that of (v'_i, z_i). It follows from the definition of reversed suffix links that in both cases the reversed suffix link to v_{i-1} comes from z_i. □

Lemma 10. *There is no reversed suffix link that comes from the new node v_{i-1} in $\mathsf{PPH}(T[i-1..])$.*

Proof. Suppose on the contrary that there is a reversed suffix link from v_{i-1} in $\mathsf{PPH}(T[i-1..])$, and let w be the node that is pointed by this reversed suffix link.

Notice that $|w| = |v_{i-1}|+1$. Let $T[j..]$ be the suffix of T for which this node w was inserted, namely, $\mathsf{id}(w) = j > i-1$. By Lemma 4, for any substring X of $T[j..j+|w|-1]$, $\mathsf{prev}(X)$ is represented by $\mathsf{PPH}(T[j..])$, and hence it is also represented by $\mathsf{PPH}(T[i..])$ since $j \leq i$. Recall that $\mathsf{prev}(T[j+1..j+|w|-1]) = \mathsf{prev}(T[i-1..i+|v_{i-1}|])$, which implies that the node v_{i-1} existed already in $\mathsf{PPH}(T[i..])$. However, this contradicts that v_{i-1} is the node that was inserted when $\mathsf{PPH}(T[i..])$ was updated to $\mathsf{PPH}(T[i-1..])$. □

Due to Lemmas 9 and 10, there is only one reversed suffix link that is newly inserted in $\mathsf{PPH}(T[i-1..])$.

4.4 Complexity analysis

Lemma 11. *The proposed algorithm runs in a total of $O(n\log(\sigma+\pi))$ time with $O(n)$ space.*

Proof. For each $i = n, \ldots, 1$, the algorithm updates $\mathsf{PPH}(T[i..])$ to $\mathsf{PPH}(T[i-1..])$. The update begins with node v_i such that $\mathsf{id}(v_i) = i$, and climbs up the path to v'_i. It takes a reversed suffix link from v'_i and moves to u_i of depth $|v'_i|+1$, and the new node v_{i-1} of depth $|v'_i|+2$ with $\mathsf{id}(v_{i-1}) = i-1$ is inserted. Hence the total number of nodes visited when updating $\mathsf{PPH}(T[i..])$ to $\mathsf{PPH}(T[i-1..])$ is $|v_i| - |v'_i| + 2 = |v_i| - |v_{i-1}| + 4$. Thus, the total number of nodes visited for all $i = n, \ldots, 1$ sums up to $\sum_{i=n}^{2}(|v_i| - |v_{i-1}| + 4) = |v_n| - |v_1| + 4(n-1) = O(n)$. At each node that we visit, it takes $O(\log(\sigma+\pi))$ time to search for the corresponding reversed suffix link, as well as inserting a new edge. Hence, the total time cost is $O(n\log(\sigma+\pi))$.

It is clear that the number of nodes in $\mathsf{PPH}(T)$ is $n+2$, including the root and the auxiliary node $\perp$. It follows from Lemmas 9 and 10 that the number of reversed suffix links coming out from the root, the internal nodes, and the leaves is $n+1$. As for the reversed suffix links that come from $\perp$ to the root, we add a new reversed suffix link labeled with $T[i-1]$ only if $T[i-1] \in \Sigma$ and $T[i-1] \neq T[j]$ for any $j < i-1$. This way, we can maintain these reversed suffix links from $\perp$ in an online manner, using $O(n)$ space. □

We have proven the following theorem, which is the main result of this paper.

Theorem 12. *For an input p-string T of length n, the proposed algorithm constructs $\mathsf{PPH}(T[i..])$ in a right-to-left online manner for $i = n, \ldots, 1$, in a total of $O(n\log(\sigma+\pi))$ time with $O(n)$ space.*

5 Parameterized pattern matching with augmented $\mathsf{PPH}(T)$

Ehrenfeucht et al. [6] introduced *maximal reach pointers*, which used for efficient pattern matching queries on position heaps. Diptarama et al. [5] introduced maximal reach pointers for their LR p-position heaps, and showed how to perform pattern matching queries in $O(m\log(\sigma+\pi) + m\pi + pocc)$ time, where m is the length of a given pattern p-string and *pocc* is the number of occurrences to report. We can naturally extend the notion of maximal reach pointers to our RL p-position heaps, as follows:

Definition 13 (Maximal reach pointers). *For each position $1 \leq i \leq n$ in T, the* maximal reach pointer *of the node v with $\mathsf{id}(v) = i$ points to the deepest node u of $\mathsf{PPH}(T)$ such that u is a prefix of $\mathsf{prev}(T[i..])$.*

prev(T[12..])	a
prev(T[11..])	0a
prev(T[10..])	00a
prev(T[9..])	a00a
prev(T[8..])	0a03a
prev(T[7..])	00a33a
prev(T[6..])	a00a33a
prev(T[5..])	0a03a33a
prev(T[4..])	00a33a33a
prev(T[3..])	a00a33a33a
prev(T[2..])	0a03a33a33a
prev(T[1..])	01a03a33a33a

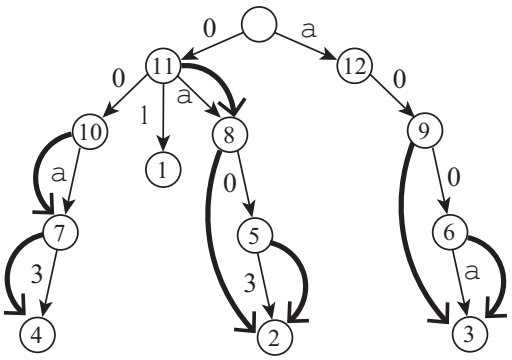


Figure 4. To the left is the list of prev(T[i..]) for p-string T = xxayxayxayxa of length 12, where $\Sigma = \{\mathtt{a}\}$ and $\Pi = \{\mathtt{x}, \mathtt{y}\}$. To the right is an illustration for augmented PPH(T), where the maximal reach pointers are indicated by the bold arrows. The wavy underlined prefix of each prev(T[i..]) in the left list denotes the longest prefix of prev(T[i..]) that is represented by PPH(T), and hence it is the destination of mrp(i).

We denote by mrp(i) the pointer of node v such that id(v) = i. The *augmented* PPH(T) is PPH(T) with the maximal reach pointers of all nodes. For simplicity, if mrp(i) points to the node with id i, then we omit this pointer. See Figure 4 for an example of maximal reach pointers and augmented PPH(T).

Lemma 14. *For every* $1 \le i \le n$*, we can compute* mrp(i) *in a total of* $O(n \log(\sigma+\pi))$ *time with* $O(n)$ *space.*

Proof. We compute mrp(i) for each position $i = 1, \ldots, n$ increasing order. In so doing, we use the *forward* suffix link that are the reversals of the reversed suffix links. For simplicity, we will call forward suffix links as suffix links. Since there is exactly one in-coming reversed suffix link to each node, there is also exactly one out-going suffix link from each node. Let sl(v) denote the node that the suffix link of v points to.

We begin with node v_1 such that id(v_1) = 1. Since we have built PPH(T[i..]) in decreasing order of i, v_1 is a leaf of PPH(T) and it is the deepest node that is a prefix of prev(T[1..]). Now we take the suffix link of v_1, and let u_1 = sl(v_1). Since prev(T[1..$|v_1|$]) = v_1, it follows from Lemma 6 that u_1 = prev(T[2..$|v_1|$]), which implies that u_1 is a prefix of prev(T[2..]). Then the deepest node v_2 that is a prefix of prev(T[2..]) can be found by traversing the corresponding path from node u_1. Then, we make a pointer to v_2 from the node w with id(w) = 2. We iteratively perform the same procedure for all positions i in increasing order.

To analyze the time complexity, we can use a similar argument as in Lemma 11. For each i, the number of nodes traversed is $|v_{i+1}|-|u_i|+1 = |v_{i+1}|-|v_i|+2$. Thus, the total number of nodes visited sums up to $\sum_{i=1}^{n-1}(|v_{i+1}|-|v_i|+2) = |v_n|-|v_1|+2(n-1) = O(n)$. Since it takes $O(\log(\sigma+\pi))$ time to search for each corresponding edge in the traversal, the total running time is $O(n \log(\sigma+\pi))$.

The space requirement is clearly $O(n)$. □

It is straightforward that by applying Diptarama et al.'s pattern matching algorithm to our PPH(T) augmented with maximal reach pointers, parameterized pattern matching can be done in $O(m \log(\sigma+\pi) + m\pi + pocc)$ time.

Corollary 15. *Using our augmented* $\mathsf{PPH}(T)$*, one can perform parameterized pattern matching queries in* $O(m \log(\sigma + \pi) + m\pi + pocc)$ *time.*

6 Conclusions and further work

This paper proposed a new indexing structure for parameterized pattern matching, called RL p-position heaps, that are built in a right-to-left online manner. We proposed a Weiner-type construction algorithm for our RL p-position heaps that runs in $O(n \log(\sigma + \pi))$ time with $O(n)$ space, for a given text p-string of length n over a static alphabet Σ of size σ and a parameterized alphabet Π of size π. The key to our efficient construction is how to label the reversed suffix links. By augmenting our position heap with maximal reach pointers, one can perform parameterized pattern matching in $O(m \log(\sigma + \pi) + m\pi + pocc)$ time, where m is the length of a query pattern and *pocc* is the number of occurrence to report.

Our future work includes the following:

- Would it be possible to shave the $m\pi$ term in the pattern matching time using parameterized position heaps? Other data structures such as parameterized suffix trees achieve better $O(m \log(\sigma + \pi) + pocc)$ time [1].
- Nakashima et al. [12] extended Ehrenfeucht et al.'s right-to-left position heaps [6] to a set of texts given as a trie. We are now working on extending our right-to-left p-position heaps to a set of texts given as a trie.

References

1. B. S. Baker: *Parameterized pattern matching: Algorithms and applications.* J. Comput. Syst. Sci., 52(1) 1996, pp. 28–42.
2. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas: *The smallest automaton recognizing the subwords of a text.* Theoretical Computer Science, 40 1985, pp. 31–55.
3. A. Blumer, J. Blumer, D. Haussler, R. Mcconnell, and A. Ehrenfeucht: *Complete inverted files for efficient text retrieval and analysis.* Journal of the ACM, 34(3) 1987, pp. 578–595.
4. E. Coffman and J. Eve: *File structures using hashing functions.* Communications of the ACM, 13 1970, pp. 427–432.
5. Diptarama, T. Katsura, Y. Otomo, K. Narisawa, and A. Shinohara: *Position heaps for parameterized strings*, in Proc. CPM 2017, 2017, pp. 8:1–8:13.
6. A. Ehrenfeucht, R. M. McConnell, N. Osheim, and S.-W. Woo: *Position heaps: A simple and dynamic text indexing data structure.* Journal of Discrete Algorithms, 9(1) 2011, pp. 100–121.
7. P. Ferragina and G. Manzini: *Indexing compressed text.* J. ACM, 52(4) 2005, pp. 552–581.
8. R. Grossi and J. S. Vitter: *Compressed suffix arrays and suffix trees with applications to text indexing and string matching.* SIAM J. Comput., 35(2) 2005, pp. 378–407.
9. G. Kucherov: *On-line construction of position heaps.* J. Discrete Algorithms, 20 2013, pp. 3–11.
10. U. Manber and G. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM J. Computing, 22(5) 1993, pp. 935–948.
11. J. Mendivelso and Y. Pinzón: *Parameterized matching: Solutions and extensions*, in Proc. PSC 2015, 2015, pp. 118–131.
12. Y. Nakashima, T. I, S. Inenaga, H. Bannai, and M. Takeda: *The position heap of a trie*, in Proc. SPIRE 2012, vol. 7608 of Lecture Notes in Computer Science, 2012, pp. 360–371.
13. 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.
14. P. Weiner: *Linear pattern-matching algorithms*, in Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory, 1973, pp. 1–11.

Parameterized Dictionary Matching with One Gap

B. Riva Shalom

Department of Software Engineering, Shenkar College, Ramat-Gan 52526, Israel.
rivash@shenkar.ac.il

Abstract. Dictionary Matching is a variant of the Pattern Matching problem where multiple patterns are simultaneously matched to a single text. In case the patterns contain sequences of don't care symbols, the problem is called Dictionary Matching with Gaps. Another famous variant of Pattern matching is the Parameterized Matching, where 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. In this paper we suggest the problem of Parameterized Dictionary Matching with one Gap, stemming from cyber security, where the patterns are the malware sequences we want to detect in the text, and the necessity of a parameterized match is due to their encryption. We present two algorithms solving the Prameterized Dictionary Matching with one Gap. The first solves the problem for dictionaries with variable length gaps and has query time of $O(n(\beta_{max} - \alpha_{min}) \log^2 d + occ)$, where n is the size of the text, d is the number of gapped patterns in the dictionary, $\beta_{max} - \alpha_{min}$ is the maximal size of gap and occ is the number of the gapped patterns reported as output. The second solution considers dictionaries with a single set of gap boundaries and has query time of $O(n(\beta - \alpha) + occ)$, where n is the size of the text, $\beta - \alpha$ is the size of the gap and occ is the number of the gapped patterns reported as output.

1 Introduction

Cyber security is a critical modern concern. It derives from cyber terroristic attacks, as well as economic dangers. Due to the importance of the problem, computer scientists develop various algorithms, dedicated to this struggle. Network intrusion detection systems 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 [24]. Having a list of gapped malware patterns yields the challenge of a dictionary matching with gaps.

In this paper we suggest an extension to the dictionary matching with one gap problem,(where every pattern in the dictionary has a single gap), where the gapped malware is encrypted, in order to evade virus scanners. We consider the case in which the encryption used is substitution cipher, by which units of plain text are replaced with ciphertext, according to a fixed system, and consider a parameterized mapping as a strategy of encryption, thus define the Parameterized Dictionary Matching with One Gap (pDMOG) problem. We suggest an algorithm for dictionary with variable length gaps and another lower time complexity for dictionaries where all patterns have gaps with identical boundaries.

Since the pDMOG problem is a combination of the Dictionary Matching with one gap problem and Parameterized Matching problem, we define hereafter each of the problems separately then form the combined definition.

Dictionary Matching with Gaps (DMOG) Let a gapped pattern be of the form $P = lp\{\alpha, \beta\}rp$, where both the left subpattern lp and the right subpattern rp

B. Riva Shalom: Parameterized Dictionary Matching with One Gap, pp. 103–116.
Proceedings of PSC 2018, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06484-9

are strings over alphabet Σ, and $\{\alpha, \beta\}$ denotes a sequence of at least α and at most β don't cares symbols between the subpatterns, where a don't care symbol can be matched to any text character from Σ. The formal definition follows.

Definition 1. The Dictionary Matching with One gap ($DMOG$) Problem*:*
Preprocess: *A dictionary D of total size $|D|$ over alphabet Σ consisting of d gapped patterns each containing a single gap.*
Query: *A text T of length n over alphabet Σ.*
Output: *All locations ℓ in T, where any gapped pattern ends.*

For example, let D be the set of patterns $\{P_1 = a\ b\ a\ \{2,4\}\ d\ d$, $P_2 = a\ b\ \{2,4\}\ c\ d$, $P_3 = b\ a\ \{2,4\}\ c\}$. 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 of 4 and also with gap of length 2, and of P_3 ending at locations 9, with gap length of 3.

Parameterized Matching The Parameterized Matching problem is a well known problem in computer science, where 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. Throughout the paper we denote a parameterized match by $p - match$. A formal definition follows.

Definition 2. Parameterized Matching Problem*(PM):*
Input: *A Text T of length n and a pattern P of length m, both over alphabet $\Sigma \bigcup \Pi$, where $\Sigma \bigcap \Pi = \emptyset$.*
Output: *All locations ℓ in T, where there exists a bijection $f : \Pi \to \Pi$ and the following hold:*
(1) $\forall P[i] \in \Sigma$, $P[i] = T[\ell + i - 1]$.
(2) $\forall P[i] \in \Pi$, $f(P[i]) = T[\ell + i - 1]$.

For example, let $\Sigma = \{a, b\}, \Pi = \{x, y, z\}$ 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 $\{4, 10\}$. The former implies mapping function $f(z) = x, f(x) = y$ while the latter implies mapping function $f(z) = y, f(x) = x$.

Parameterized Dictionary Matching with One Gap (pDMOG) The pDMOG problem is a combination of the above problems. Note, that according the motivation of the problem, we consider malicious code to appear on two packets, thus each part of the gapped pattern lp_i, rp_i, does not relate to the other part of the pattern, hence, they can be matched using different matching functions. The formal definition follows.

Definition 3. The Parametrized Dictionary Matching with One gap ($pDMOG$) Problem*:*
Preprocess: *A dictionary D consisting of d gapped patterns $\{P_i\}$ over alphabet $\Sigma \bigcup \Pi$, where $\Sigma \bigcap \Pi = \emptyset$ 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 $\Sigma \bigcup \Pi$, $\Sigma \bigcap \Pi = \emptyset$*
Output: *All locations ℓ in T, where there exists a bijection $f : \Pi \to \Pi$ and all the following hold for any P_i and a gap length $g \in [\alpha_i, \beta_i]$:*
(1) $\forall lp_i[j] \in \Sigma$, $lp_i[j] = T[\ell - |lp_i| - j]$.
(2) $\forall lp_i[j] \in \Pi$, $f(lp_i[j]) = T[\ell - |lp_i| - j]$.
(3) $\forall rp_i[j] \in \Sigma$, $rp_i[j] = T[\ell + g + j]$.
(4) $\forall rp_i[j] \in \Pi$, $f(rp_i[j]) = T[\ell + g + j]$.

For example, let $\Sigma = \{a, b\}, \Pi = \{q, u, v, w, z\}$ 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 former implies p-matching P_1 using mapping function $f(z) = u, f(x) = u$ for lp_1, a gap of length 3 and a mapping function $f(u) = w, f(q) = z$ for rp_1. The latter implies p-matching P_2 using mapping function $f(u) = v, f(q) = u$ for lp_2, a single character gap and a mapping function $f(u) = z, f(v) = w$ for rp_2.

We consider the alphabet to be of fixed size. If it is of variable size, a factor of $\log \sigma$ is to be multiplied to n in the query time of both solutions.

The paper is organized as follows. Section 2 scans previous work. Section 3 suggests the framework of the algorithm and some notations. The first part of the algorithm appears on Section 4 and the second part of the algorithm appears on Section 5. Section 6 concludes the paper and poses some open problems.

2 Previous Work

Dictionary matching has been amply researched (see e.g. [2,3,4,5,7,15]). When the patterns are gapped, and we consider the problem of Dictionary Matching with Gaps, there are several algorithms solving the problem, yet their definitions of the problem are not identical.

Rahman et al. [28] suggest an algorithm using AC automaton, and suffix arrays built over the text. Bille et al. [14], [13] improved time complexity, by using sorted lists of disjoint intervals, yet both solutions includes a factor of *socc* which is the total number of occurrences of the subpatterns in the text which can be very large. Kucherov and Rusinowitch [25] and Zhang et al. [29] solved the problem of matching a set of patterns with variable length of don't cares. Yet, they report a leftmost occurrence of a pattern if there exists one, while we are interested in all occurrences of the patterns in the text. Haapasalo et al. [20] gave an on-line algorithm for the general problem, yet, they report at most one occurrence for each pattern at each text position.

Amir et al. [9] solved the 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 query time of their second algorithm is $O(|T|(\beta - \alpha) + occ)$ and space of $O(d^2 + |D|)$, where d is the number of gapped patterns in dictionary D and *occ* is the number of patterns reported. Hon et al. [21] presented a similar solution, for dictionaries with variable length gaps, improving the space complexity to a linear space and requiring query time of $O(|T|\gamma \log \lambda \log d + occ)$, where γ denotes the number of distinct gap lengths and λ denotes the number of distinct lower and upper bounds of gap lengths.

Amir et al. [8] also considered the online version of the DMOG problem, where the text arrives online, a character at a time, and the requirement is to report all gapped patterns that are suffixes of the text that has arrived so far, before the next character arrives. In [10] Amir et al. considered the recognition version of the online DMOG problem, where each gapped pattern is reported at most once, during the entire online text scan.

Regarding 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.[11] The problem has many application in various fields, as detailed in [27], such as Image processing, where parameterized matching can help searching an icon on the screen, or improving ergonomy of databases of URLS. As a consequence, extensive work has been done on the problem and its various variants, some of which Lewenstein [26] and Mendivelso and Pinzon [27] scan. Among the parameterized matching extensions are, the work of Amir et al. [6] suggesting a parameterized version of KMP, Baker works [12], [11] regarding the maximal p-matches over a threshold length and a p-suffix tree, the parameterized fixed and dynamic dictionary problems presented by Idury and Schauffer [22], and improved by Ganguly et al. [19], the efficient parameterized text indexing, shown by Ferragina and Grossi [17], p-suffix arrays presented by Deguchi et al. [16], the Parameterized version of the LCS problem by Keller et al. [23] and many more.

3 Parameterized Dictionary Matching with One Gap - Framework

Throughout the paper we use the following notations. Let $D = \{P_1, \ldots, P_d\}$ be the dictionary, where every P_i is a gapped pattern of the form $lp_i\{\alpha_i, \beta_i\}rp_i$. In case the dictionary has a single set of gap boundaries $\{\alpha, \beta\}$, then $\forall 1 \leq i \leq d$, $\alpha_i = \alpha$ and $\beta_i = \beta$. We call lp_i the $left$ subpattern of P_i, and call rp_i the $right$ subpattern of P_i. We divide all subpatterns of the dictionary into two sets $Left =_{1 \leq i \leq d} \{lp_i\}$ where $d_{Left} = |Left| \leq d$ and $Right =_{1 \leq i \leq d} \{rp_i\}$ where $d_{Right} = |Right| \leq d$.

The solution for the DMOG problem, suggested in this paper, follows the frameworks of [9] and their improvement in [21]. Their algorithms consist of two parts: The first part is detecting separately all the left subpatterns and all the right subpatterns of the dictionary in the text. The second part is processing the subpattern occurrences, in order to efficiently report all gapped patterns P_i where both their subpatterns appear with a g sized gap between them, where $\alpha_i \leq g \leq \beta_i$.

For the first step they practiced the observation that matching two parts of a pattern P_i to a text, can be done by matching the reverse of the left subpatterns in $Left$ to the reverse of $T[1, \ldots, \ell]$ for all ℓs and matching the right subpatterns in $Right$ to $T[\ell + g + 1 \ldots n]$, where g is the size of the gap between the subpatterns occurrences. To this aim they constructed a generalized suffix tree of all the reverse of the $Left$ subpatterns, and a generalized suffix tree of all the $Right$ subpatterns.

For the second step, given a match of the reverse of some lp_i to $T[\ell \ldots 1]$ and a match of some rp_j to $T[\ell + g + 1 \ldots n]$, it is necessary to conclude which gapped patterns occurred, thus ought to be reported. Note, that several gapped patterns can be reported, such that their left subpattern is a suffix of lp_i and their right subpattern is a prefix of rp_j.

Hon et al. [21] suggested using range queries by rectangular stabbing for the second step of the algorithm. [9] suggested an additional technique, when all patterns share the same gap boundaries, in case query time is required to be $O(1 + occ)$ time per text location and gap length, where they use a look up table built in the preprocess stage.

The parameterized matching does not require exact matches between the Π characters, but rather to capture the characters order in the pattern. For this reason

Baker [11] 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 $s_i \in \Pi$, $prev(s_i) = 0$ if s_i is the leftmost position in S of this character, and $prev(s_i) = i - k$ if k is the previous position to the left at which the character s_i occurs. For example, let $\Sigma = \{a, b\}$, $\Pi = \{u, v\}$ and $S = a\ b\ u\ v\ a\ b\ u\ v\ u$, then $prev(S) = a\ b\ 0\ 0\ a\ b\ 4\ 4\ 2$. The string obtained by $prev(S)$ is called the p-string of S.

Lemma 4. *[11] Strings S_1, S_2 have $prev(S_1) = prev(S_2)$ iff they are p-matched.*

A direct result of this lemma is that using p-strings enables applying parameterize matching to various pattern matching techniques, with certain modifications required due to the behaviour of the $prev$ function, some of which were referred to in Section 2.

This paper follows the frameworks of [9], [21], yet adapts them to using parameterized matching while solving the DMOG problem. The algorithms suggested are described in the following sections, according to the parts of the solution.

4 p-Matching of Subpatterns

As mentioned in the previous section, both [9] and [21] construct two generalized suffix trees, one over the *Right* subpatterns and the other over the reverse of the the $Left$ subpatterns, which we call $Left^R$. They do not traverse the text query T using these suffix trees but rather require to attain the longest match of every suffix of T with the *Right* suffix tree and the longest match of every suffix of T^R and the $Left^R$ suffix tree. They do it in $O(n)$ time using Amir et. al. [5] technique of inserting all suffixes of T or T^R to the corresponding suffix trees. By inserting each suffix of T to the subpatterns generalized suffix tree, the needed information is gathered in linear time.

When considering the parameterized matching case, the parameterized suffix tree is considered as the mechanism of locating the $prev$ function of the subpatterns in $prev(T)$. Baker [11] showed the construction of a parameterized suffix tree. She used dynamic trees and lowest common ancestor queries to achieve the following results.

Lemma 5. *([11]) Given finite disjoint alphabets Σ, Π , a p-suffix tree can be built for a p-string S in time $O(|S| \log |S|)$ and linear space in the $|S|$. Given a p-string text query T, all p-matches of S in T can be reported in time $O(|T|)$ for fixed alphabets and in time $O(|T| \log(\min\{|S|, \sigma\}))$, where $\sigma = |\Sigma| + |\Pi|$ for variables alphabets.*

Other works, such as [19] considered an efficient construction and space consumption of a p-suffix tree, yet they did not reduce the construction time from $O(|S| \log |S|)$. The scheme we follow requires construction of p-suffix trees both in the preprocess and during query execution, where we insert suffixes of $prev(T)$ to a generalized p-suffix tree, thus, the first part of answering a query requires $O(n \log n)$ time. In order to decrease the query time, we consider another technique for dictionary matching for the first step of the algorithm, which is using the Aho-Corasick automaton [2].

Idury and Schaffer [22] constructed a modified Aho-Corasick automaton (AC) [2] suitable for p-strings. Their construction algorithm is similar to that of the original AC construction, yet important modifications were made to the goto and fail links of the automaton, adapting it to work with p-strings. Their p-AC automaton occupies $O(m \log m) = O(|D| \log |D|)$ bits, where m is the number of states in the automaton. They report all p-matches of patterns from dictionary D in text T in $O(|T| \log \sigma + occ)$ time, where occ are the number of reported occurrences. Note that in case we report

only the longest pattern located for each text location, the query is answered in $O(|T| \log \sigma)$ as the *occ* element is added since we require reporting all appearances of subpatterns that are suffixes of the longest subpattern recognized. Ganguly et. al. [18] suggested a space efficient data structure for the parameterized dictionary matching, improving the p-AC automaton of [22] by using sparsification technique. Their index requires $O(|D| \log \sigma + d \log |D|)$ bits and the report of all p-matches in text T requires $O(|T|(\log \sigma + \log_\sigma |D|) + occ)$. Due to our motivation in cyber security we use the data structure of Idury and Schaffer [22], guaranteeing a faster query time.

We calculate in linear time the p-string, $prev(lp_i)$ for every $lp_i \in Left$ and construct a p-AC automaton upon them, named $LpAC$. In addition we calculate $prev(T)$, thus by scanning $prev(T)$ using $LpAC$ we can locate all left subpatterns p-matching the text $T[1..\ell]$, where the p-match *ends* at location $T[\ell]$. For the $Right$ subpatterns, we need to locate all occurrences of $prev(rp_i)$ *starting* at location ℓ in $prev(T)$, hence, we need to scan the reverse of T and look for occurrences of the $prev(rp_i^R)$, therefore for every $rp_i \in Right$ we calculate in linear time the p-string of the reversed subpattern, $prev(rp_i^R)$ and construct a p-AC automaton upon them, named $RpAC$. In addition, the $prev$ function of the reverse of the text $prev(T^R)$ is calculated.

Note, that even in case the alphabets are not fixed, calculating the $prev$ function of a string S requires $O(|S|)$ time by using perfect hash tables for the position of the latest occurrence of a character in S. Each automaton consists of states, representing the p-strings of prefixes of the dictionary subpatterns. We consider the $p - label$ of a state to be the p-string of the sequence the state represents. A state p-labeled by a p-string of a subpattern from the dictionary is called an accepting state. Every state in the p-automata is numbered as will be described in the next section.

We scan $prev(T)$ using the $LpAC$ automaton and for every location ℓ in $prev(T)$, reached by the automaton, we save at array $Locc[\ell]$ the number of the current state in $LpAC$. Similarly, we scan $prev(T^R)$ using the $RpAC$ and save in $Rocc[\ell]$ the number of the current state reached by the automaton at $prev(T^R[\ell])$.

Lemma 6. *Performing the search with LpAC, RpAC yields for each text location ℓ, a state representing the longest prefix of some $prev(lp_i)$, p-matching the suffix of $prev(T[1 \dots \ell])$ and a state representing the longest prefix of some $prev(rp_j)$ p-matching the prefix of $prev(T[\ell \dots n])$, in linear time in the length of the text, for fixed alphabets, and with $O(|D| \log |D| + n)$ space requirements, where $|D|$ is the size of the dictionary and n is the size of the text.*

Proof. Scanning $prev(T)$ of the query text T with both of the p-automata requires $O(n)$, as [22] proves that scanning a p-text with a p-automaton requires linear time in the size of the text, for fixed alphabets. A single scan is sufficient using each p-automaton as the parameterized fail links, $pfail$ allow continuation of search from the point of a mismatch between the $prev(T)$ and the $prev$ of the current matched subpattern [22]. The p-automaton saves at every step of the scan the current state p-matching the current $prev(T)$ character, thus the longest prefix of a p-subpattern ending at the current text location. Using the reverse of T and the reverse of rp_j subpatterns we get that p-matching the longest $prev(rp_j^R)$ at the suffix of $prev(T[n \dots \ell])$ equals the p-matching of the longest rp_j starting at $prev(T[\ell \dots n])$.

Regarding space, the p-automaton is built over dictionary of size $|D|$, thus requires $O(|D| \log |D|)$ space, as proved in [22]. In addition we save the $Locc, Rocc$ arrays maintaining a pointer to a single state, for every text location. □

5 Results Calculation

Given the output of the p-automata scans at arrays *Locc*, *Rocc*, the second step of our algorithm is to report all gapped patterns where both their subpatterns p-matched the text, with a gap of size g between their occurrences. Hence, for a gap starting at text location $\ell+1$, we consider $Locc[\ell], Rocc[\ell+g+1]$ and want to report all gapped patterns P_i, where $prev(lp_i)$ is a suffix of the sequence associated with the state saved at $Locc[\ell]$, and $prev(rp_i^R)$ is a suffix of the sequence associated with the state saved at $Rocc[\ell+g+1]$, where $g \in [\alpha_i, \beta_i]$. In the following subsections, we suggest two algorithms for results calculation, the first follows the range query described in [21], enabling solving the pDMOG problem for dictionaries containing variable lengths gaps while the second follows the second solution of [9], enabling result calculation in $O(1+occ)$ time per a text location and a gap length, where *occ* is the number of reported patterns, yet it solves the pDMOG for dictionaries with a single set of gap boundaries.

5.1 Results Calculation by Rectangle Stabbing

Afshani et. al. [1] considered the problem of *Rectangular Stabbing*, where a set of k axis-aligned hyper rectangles are preprocessed, then, given a query k dimensional point, all t rectangles that contain the query point, can be easily reported. Note, that by the problem definition, a point on he boundary of a rectangle is not assumed to be contained in the rectangle. They proved the following lemma.

Lemma 7. *([1]) A set of d k-dimensional rectangles (where $k \geq 2$ is a constant) can be preprocessed into $O(d\log^{k-2} d)$ space data structure which can answer any rectangular stabbing query in $O(\log^{k-1} d + output)$*

Hon. et. al. [21] created a hyper rectangle region representing every gapped pattern in the dictionary. When given occurrences of some lp_i and rp_j, and a certain gap between the occurrences, they perform a rectangular stabbing query, and report all gapped pattern found in the text according to the given subpatterns and the gap between them. In order to have a single query of lp_i, rp_j and still retrieve all gapped patterns included in the query, that is all gapped patterns P_f where lp_f is a suffix of lp_i and rp_f is a prefix of rp_j, that appear with an appropriate gap between them, they numbered the nodes in the suffix trees they built over the $Left$ and $Right$ subpatterns, by their preorder rank. Such a numbering guarantees that a prefix of some rp_i has a smaller number than rp_i itself. In addition, due to the structure of suffix trees, they had that rp_i was in the subtree of all its prefixes. Therefore, by defining a dimension of the rectangle to be the number of a node in the suffix tree and the rightmost node in its subtree, they obtained the sought after reports.

However, for parameterized patterns, the case is more delicate. We need the rectangle related to a state p-labeled by $prev(lp_i)$ to be included in the rectangle related to the state p-labeled by $prev(lp_f)$, where $prev(lp_f)$ is a suffix of $prev(lp_i)$. Yet, the *prev* function does not preserve the suffix relation of the strings it is applied to. Consider x, y as two subpatterns, where x is a suffix of y. It is not guaranteed that $prev(x)$ is a suffix of $prev(y)$, due to the changes of the *prev* function when deleting characters from the beginning of the string. For example consider $lp_i = uuua$ and its suffix uua, so $prev(lp_i) = 011a$ yet, $prev(uua) = 01a$, which is not a suffix of $011a$.

Nevertheless, in the p-AC automaton, we can find the suffix of a p-subpattern by its $pfail$ link, as it points to a prefix of a p-subpattern that is a suffix of the

p-subpattern p-labeling the current state. Therefore, we construct for LpAC the trie $Lpfail$ and for $RpAC$ the trie $Rpfail$ respectively, where the nodes of the trie are the states of the p-automaton, the root of the trie correspond to the start state of the automaton and the children of a node x are all the states having a pfail link to x in the p-automaton. Obviously, the construction of these tries is done in linear time in the size of the p-automata. Numbering the nodes of $Lpfail$, $Rpfail$ by their preorder rank, yields the possibility to use the rectangular stabbing procedure efficiently for parameterized gapped dictionaries.

In the preprocess, we number each state x of $LpAC$ according to its preorder number in $Lpfail$ and denote it by $lnum(x)$. Similarly $rnum(y)$ is the preorder number of state y of $RpAC$ in the trie $Rpfail$. We name an $LpAC$ state, p-labeled by $prev(lp_i)$ by $lstate_{lp_i}$ and the $RpAC$ state p-labeled by $prev(rp_i^R)$ is named $rstate_{rp_i}$. Then, for every gapped pattern $P_i = lp_i\{\alpha_i, \beta_i\}rp_i \in D$ we construct a hyper rectangular region R_i in 3D where $R_i = [lnum(lstate_{lp_i}) - 1, lnum(x) + 1] \times [rnum(rstate_{rp_i}) - 1, rnum(y) + 1] \times [\alpha_i - 1, \beta_i + 1]$ where x is the rightmost leaf node in the subtree of $lstate_{lp_i}$ in $Lpfail$, y is the rightmost leaf node in the subtree of $rstate_{rp_i}$ in $Rpfail$ and α_i, β_i are the gap boundaries of P_i.

Lemma 8. *Given the filled $Locc, Rocc$ arrays, performing a Rectangular Stabbing query of point $(lnum(Locc[\ell]), rnum(Rocc[\ell + g + 1]), g)$ for $\alpha_{min} \leq g \leq \beta_{max}$ where $\alpha_{min} = \min_{1 \leq i \leq d}\{\alpha_i\}$, $\beta_{max} = \max_{1 \leq i \leq d}\{\beta_i\}$, yields all gapped patterns P_i p-matching text T, such that the occurrence of $prev(lp_i)$ ends at $prev(T[\ell])$ and there is a beginning of an occurrence of $prev(rp_i)$ after a gap of g characters.*

Such a query requires $O(\log^2 d + occ)$ time and space of $O(d \log d)$, where d is the number of gapped patterns and occ is the number of patterns reported as output.

Proof. Given the query point $(lnum(Locc[\ell]), rnum(Rocc[\ell + g + 1]), g)$, according to [1] all $R_i = [a, a'] \times [b, b'] \times [c, c']$ are retrieved, where $a < lnum(Locc[\ell]) < a'$, $b < rnum(Rocc[\ell + g + 1]) < b'$ and $c < g < c'$ holds. Suppose some $prev(lp_i)$ was located ending at location ℓ and $prev(rp_i^R)$ was located ending at location $\ell + g + 1$ in T^R, thus the query point is $(lnum(lstate_{lp_i}), rnum(rstate_{rp_i}), g)$. Obviously $lnum(lstate_{lp_i}) - 1 < lnum(lstate_{lp_i}) < lnum(x) + 1$, and $rnum(rstate_{rp_i}) - 1 < rnum(rstate_{rp_i}) < rnum(y) + 1$, when x, y are the rightmost leaves in the subtrees of $rstate_{lp_i}, rstate_{rp_i}$ in $Lpfail$, $Rpfail$ respectively, due to the preorder numbering. Hence, R_i is stabbed and P_i is reported if the gap length g between the subpatterns, is in accordance with boundaries α_i and β_i.

Another possible case is that $Locc[\ell] = f$, $Rocc[\ell + g + 1] = h$ and $prev(lp_i)$ is a suffix of the p-label of state f and $prev(rp_i^R)$ is a suffix of the p-label of state h, thus P_i needs to be reported in case the gap fits. Since $prev(lp_i)$ is a suffix of the p-label of state f, it follows that the state p-labeled by $prev(lp_i)$ is an ancestor of state f in the $Lpfail$ trie, thus $lnum(lstate_{lp_i}) < lnum(f)$ due to the preorder numbering. Moreover, as f is included in the subtree rooted by $lstate_{lp_i}$, we have that $lnum(f) < lnum($the rightmost leaf in the subtree rooted by $lstate_{lp_i}) + 1$. Similarly we have that $rnum(rstate_{rp_i}) < rnum(h)$ and $rnum(h) < rnum($the rightmost leaf in the subtree rooted by $rstate_{rp_i}) + 1$. It follows that the hyper rectangle R_i is stabbed by the query point, if the gap of length g between the located subpattern is in accordance with boundaries α_i and β_i, thus P_i is reported.

The time and space complexity of a query follow Lemma 7, considering the case of d hyper rectangles in $3D$, constructed in the preprocess. □

We perform such rectangular stabbing queries, for every text location $1 \leq \ell \leq n$ and for every possible gap size, $\alpha_{min} \leq g \leq \beta_{max}$ where $\alpha_{min} = \min_{1 \leq i \leq d}\{\alpha_i\}$, $\beta_{max} = \max_{1 \leq i \leq d}\{\beta_i\}$.

Lemma 6 and Lemma 8 yields Theorem 9.

Theorem 9. *The pDMOG problem for dictionary D with variable length gaps and text query T, can be solved in $O(|D| \log |D| + n)$ space, and with a query time of $O(n(\beta_{max} - \alpha_{min}) \log^2 d + occ)$, where n is the size of T, d is the number of gapped patterns in the dictionary and occ is the number of reported patterns.*

5.2 Results Calculation by Look-up Table

In case all gapped patterns share their gap boundaries and a query time is crucial, we suggest solving the intersection between the appearances of p-subpatterns using a lookup table named *out*, though it implies an increase in preprocessing time.

For an efficient filling of the lookup table, the subpatterns numbering has to satisfy the rule that the longer a subpattern, the higher its numbering, that is, $lnum(lstate_{lp_f}) > lnum(lstate_{lp_i})$, (where $lstate_{lp_i}$ is the state p-labeled by $prev(lp_i)$ in $LpAC$) *iff* $|lp_f| \geq |lp_i|$. Similarly, $rnum(rstate_{rp_h}) > rnum(rstate_{rp_j})$, (where $rstate_{rp_j}$ is the state p-labeled by $prev(rp_j^R)$ in $RpAC$), *iff* $|rp_h| \geq |rp_j|$. The numbering system from the previous subsection can be used as well as a simple BFS traversal over $LpAC/RpAC$.

The look up table consists of accepting states, yet, $Locc[\ell]$ and $Rocc[\ell + g + 1]$ can include any state in each of the p-automata, thus for each state x in each of the p-automata, that is *not an accepting state*, we save $accept(x)$ that is the accepting state with the longest p-label that is a suffix of the p-label of x. The $accept(x)$ are calculated, by a BFS traversal over the automaton. When reaching state x that is not an accepting state, we consider its pfail link, where $pfail(x)$ points to the longest p-labeled state that its p-label is a suffix of the p-label of x. In case $pfail(x)$ is an accepting state, then $accept(x) = pfail(x)$, otherwise $accept(x) = accept(pfail(x))$.

For every *accepting state* $lstate_{lp_i} \in LpAC$ we save a link $psuf(lp_i)$ that leads to the $lnum$ of an accepting state p-labeled by the longest lp_k such that $prev(lp_k)$ is a real suffix of $prev(lp_i)$, if it exists. We define $psuf(x) = lnum(pfail(x))$ if $pfail(x)$ is an accepting state and $psuf(x) = lnum(accept(pfail(x)))$ otherwise. Similarly, for every accepting state $rstate_{rp_j} \in Right$, we save a link $psuf(rp_i^R)$ that leads to the $rnum$ of an accepting state p-labeled by the longest rp_k such that $prev(rp_k^R)$ is a real suffix of $prev(rp_i^R)$, if it exists. This link is similarly calculated in the $Rpfail$ trie.

The *out* table is of size $d_{left} \times d_{right}$. Entry $out[f, h]$ refers to the set of all indices of gapped patterns that are reported when $prev(lp_i)$ is the longest subpattern that appears at the suffix of $prev(T[1 \ldots \ell])$ and $lnum(lstate_{lp_i}) = f$, and when $prev(rp_j^R)$ is the longest subpattern that appears at the suffix of $prev(T[n \ldots \ell + g + 1])$, and $rnum(lstate_{rp_j}) = h$ and $\alpha \leq g \leq \beta$. The *out* table is recursively filled in increasing order of indices, where filling $out[f, h]$ entry implies filling four fields:

1. Index field, $out[f, h].index = i$ iff $i = j$. (Note that at most one index can be saved at $out[f, h].index$ as two patterns are bound to differ by at least one subpattern, having a single set of gap boundaries.)
2. *up* link, where $out[f, h].up = [f', h]$ iff $prev(lp_k)$ is the longest suffix of $prev(lp_i)$ where $f' = lnum(lstate_{lp_k})$ and $k = j$.

3. $left$ link, where $out[f,h].left = [f,h']$ iff $prev(rp_k^R)$ is the longest sufix of $prev(rp_j^R)$ where $h' = rnum(rstate_{rp_k})$ and $k = i$.
4. $back$ link, where $out[f,h].back = [psuf^*(f), psuf^*(h))$, where $psuf^*(f)$ is the longest real suffix of $prev(lp_i)$ and $psuf^*(h)$ is the longest real suffix of $prev(rp_j^R)$ such either $psuf^*(f)$ or $psuf^*(h)$ form a gapped pattern with a the suffix of the other. ($psuf^*(f)$ can be obtained by recursively applying the $psuf$ links.)

The lookup table is filled by the following formal recursive rule.

The Recursive Rule

$$out[f,h].up = \begin{cases} [psuf(f),h] & \textbf{if } out[psuf(f),h].index \neq null \\ out[psuf(f),h].up & \textbf{otherwise} \end{cases}$$

$$out[f,h].left = \begin{cases} [f,psuf(h)] & \textbf{if } out[f,psuf(h)].index \neq null \\ out[f,psuf(h)].left & \textbf{otherwise} \end{cases}$$

$$out[f,h].back = \begin{cases} [psuf(f),psuf(h)] & \textbf{if } out[psuf(f),psuf(h)].index \neq null \\ & \text{or } out[psuf(f),psuf(h)].up \neq null \\ & \text{or } out[psuf(f),psuf(h)].left \neq null \\ out[psuf(f),psuf(h)].back & \textbf{otherwise} \end{cases}$$

Considering $Locc[\ell] = f, Rocc[\ell + g + 1] = h$, the results calculation is performed by consulting entry $out[f,h]$. We report $out[f,h].index$ if it exists, yet in order to report all relevant patterns, that their p-subpatterns are numbered by f or by h or that they are suffixes of the p-label of the states numbered by f, h, we follow the links saved at $out[f,h]$, as detailed in the procedure :

ResultsQuery(f, h **)**:

1. If $out[f,h].index \neq null$, report $out[f,h].index$.
2. If $out[f,h].back \neq null$
   ResultsQuery(f', h') for $[f',h'] = out[f,h].back$.
3. Let $f' \leftarrow f$, $h' \leftarrow h$.
4. **While** ($out[f',h].up \neq null$).
   (a) Let $[f',h] = out[f',h].up$.
   (b) Report $out[f',h].index$.
5. **While** ($out[f,h'].left \neq null$).
   (a) Let $[f,h'] = out[f,h'].left$.
   (b) Report $out[f,h'].index$.

Lemma 10. *The procedure* ResultsQuery*, given the gapped dictionary D, $Locc[\ell] = f, Rocc[\ell+g+1] = h$ and the $psuf$ function, reports all dictionary patterns appearing with gap of size g starting at $T[\ell+1]$.*

Proof. Due to the construction of the p-AC automata, we have that state numbered by f represents $prev(lp_i)$ and all its suffixes and the state numbered h represents a certain $prev(rp_j^R)$ and all its prefixes. According to the AC algorithm the subpatterns represented by these states are of maximal length [2].

In order to report all required patterns, entry $out[f,h]$ for $1 \leq f \leq d_{left}$,$1 \leq h \leq d_{right}$, has to contain links to all entries containing indices of patterns whose $left$ subpattern is represented by the state numbered f and its $right$ subpattern is represented by the state numbered h. There are 4 possible cases:

1. Case 1: lp_i and rp_j form a pattern P_i, (i=j) then $out[f,h].index = i$ and this pattern is reported.
2. Case 2: lp_i and rp_j form a pattern P_j, and $prev(lp_j)$ is a suffix of $prev(lp_i)$. If $psuf(lp_i) = lnum(lstate_{lp_j})$, then $out[f,h].up = [psuf(f), h]$, so we have a direct link to the entry containing pattern index j. If a shorter suffix of $prev(lp_i)$, whose state is numbered by f', forms a pattern with $prev(rp_j)$, such a suffix is a suffix of the p-label of the state numbered by $psuf(lp_i)$, where according to the numbering system $psuf(lstate_{lp_i}) < lnum(lstate_{lp_i})$, thus entry $out[psuf(f), h].up$ was already computed, and includes a link to $out[f', h]$. Note, that in case several $left$ p-subpatterns which are all suffixes of $prev(lp_i)$ form a pattern with rp_j, it implies all these suffixes, include each other as suffixes, thus can be reached by recursively following up links starting from $out[psuf(f), h]$.
3. Case 3: lp_i and rp_i form a pattern P_i, and $prev(rp_i^R)$ is a suffix of $prev(rp_j^R)$. If $psuf(rp_j) = rnum(rstate_{rp_i})$, then $out[f,h].left = [f, psuf(h)]$, so we have a direct link to the entry containing pattern index i. If a shorter suffix of $prev(rp_j^R)$, whose state is numbered by h' forms a pattern with $prev(lp_i)$, such a suffix is a suffix of the p-label of the state numbered by $psuf(rp_j^R)$, where according to the numbering system, $psuf(rstate_{rp_j}) < rnum(rstate_{rp_j})$, thus entry $out[f, psuf(h)].left$ was already computed, and includes a link to the $out[f, h']$. Note, that in case several $right$ p-subpatterns which are all suffixes of $prev(rp_j^R)$ form a pattern with lp_i, it implies all these subpatterns include each other as suffixes, thus can be reached by recursively following $left$ links starting from $out[f, psuf(h)]$.
4. Case 4: Some suffix of the $prev$ of the subpattern numbered by f and some suffix of the $prev$ of the reverse of subpattern numbered by h form gapped patterns. Note that it must be a real suffixes of the current subpatterns, as previous cases dealt with cases where lp_i or rp_j themselves where a part of reported patterns.
   (a) In case $out[psuf(f), psuf(h)].index \neq null$ the pattern index is reported.
   (b) In case an $out[psuf(f), psuf(h)]$ has a non null up link it implies that a suffix (or some suffixes) of the p-label of the state numbered by $psuf(f)$ forms a pattern with the the p-label of the state numbered by $psuf(h)$, recursively going over these up links we report all these patterns.
   (c) In case an $out[psuf(f), psuf(h)]$ has a non null $left$ link it implies that a suffix (or some suffixes) of the p-label of the state numbered by $psuf(h)$ forms a pattern with the p-label of the state numbered by $psuf(f)$, recursively going over these $left$ links we report all these patterns.
   (d) In case no pattern is formed by the p-label of the states numbered by either $psuf(f)$ nor $psuf(h)$, it must be that the patterns are formed by shorter suffixes of the subpatterns represented by states numbered f, h. They may be formed by state numbered $psuf(psuf(f))$ and state numbered $psuf(psuf(h))$ or by even sorter suffixes, which is $[psuf^*(f), psuf^*(h)]$. Nevertheless, by the definition of the $back$ link, the indices of the longest possible suffix of the p-label of the state numbered by $psuf(f)$ and the longest suffix of p-label of the state numbered by $psuf(h)$ which form a pattern, are saved in $out[psuf(f), psuf(h)].back$, which was already computed, due to the numbering system, so we follow $out[psuf(f), psuf(h)].back$, where we will have a pattern index or a link to an up or a left entry containing a pattern index.

These observations, can be easily proved by induction. □

Lemma 11. *The construction of the out table requires* $O(|D| \log |D| + d_{left} \times d_{right})$ *time and* $O(d_{left} \times d_{right})$ *space. Performing a query on the out table regarding p-subpatterns appearing adjacently to a gap starting at* $T[\ell + 1]$*, requires* $O(1 + occ)$ *time, where occ is the number of patterns reported.*

Proof. The preprocess requires numbering the accepting states of both p-AC automata and computing the *accept* and $psuf$ links, all can be done by performing a BFS traversal over p-automata and the $Lpfail, Rpfail$ tries, in linear time in the size of the p-automata and tries, $O(|D| \log |D|)$. Filling $d\ out[f, h]$ entries with index i when $lnum(lstate_{lp_i}) = f$ and $rnum(rstate_{rp_i}) = h$, can be done in $O(d)$ time. Filling each of the entries of the table $out[f, h]$ can be performed in $O(1)$ by the recursive rule.

The table query procedure is based on following links and reporting indices found. Every step of following an *up* or $left$ link implies that the linked entry contains a pattern index, needs to be reported. The *back* link either directs us to an entry including a pattern index, needs to be reported or it directs us to an entry containing an *up* or $left$ links. Hence, by following at most two links we encounter an index needs to be reported. Consequently, the time of following links is attributed to the size of the output.

The lookup table has $d_{left} \times d_{right}$ entries, each consists of 4 fields, yielding $O(d_{left} \times d_{right})$ space requirement.

□

For every text location $1 \leq \ell \leq n$ and for every possible gap size, $\alpha \leq g \leq \beta$ we perform a look up table query $out[lnum(Locc[\ell]), rnum(Rocc[\ell + g + 1])]$.

Lemma 6 and Lemma 11 yields Theorem 12.

Theorem 12. *The pDMOG problem for dictionary D with a single set of gap boundaries and text query T, can be solved in* $O(|D| \log |D| + d^2)$ *space and with query time* $O(n(\beta - \alpha) + occ)$*, where n is the size of the text and occ is the number of reported gapped patterns.*

6 Conclusions and Open Problems

This paper suggests the problem of dictionary matching with one gap where the matching technique is parameterized, a problem with tight relation to cyber security. The paper presents efficient and simple to program algorithms.

There are several interesting open problems related to the pDMOG problem, such as solving the DMOG problem for other methods of encrypted gapped patterns and solving the pDMG for patterns containing multiple gaps. Since the DMOG problem is a crucial bottleneck procedure in network intrusion detection system applications, these open problems should be addressed in the future.

References

1. P. Afshani, L. Arge, and K. G. Larsen: *Higher-dimensional orthogonal range reporting and rectangle stabbing in the pointer machine model*, in Symposuim on Computational Geometry 2012, SoCG '12, Chapel Hill, NC, USA, June 17-20, 2012, 2012, pp. 323–332.
2. A. V. Aho and M. J. Corasick: *Efficient string matching: An aid to bibliographic search.* Commun. ACM, 18(6) 1975, pp. 333–340.
3. A. Amir and G. Călinescu: *Alphabet-independent and scaled dictionary matching.* J. Algorithms, 36(1) 2000, pp. 34–62.
4. A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park: *Dynamic dictionary matching.* J. Comput. Syst. Sci., 49(2) 1994, pp. 208–222.
5. 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.
6. A. Amir, M. Farach, and S. Muthukrishnan: *Alphabet dependence in parameterized matching.* Inf. Process. Lett., 49(3) 1994, pp. 111–115.
7. 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.
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 2016, December 12-14, 2016, Sydney, Australia, 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 2017, Prague, Czech Republic, August 28-30, 2017, 2017, pp. 3–17.
11. B. S. Baker: *A theory of parameterized pattern matching: algorithms and applications*, in Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA, 1993, pp. 71–80.
12. B. S. Baker: *Parameterized duplication in strings: Algorithms and an application to software maintenance.* SIAM J. Comput., 26(5) 1997, pp. 1343–1362.
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 Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010, 2010, pp. 1297–1308.
15. G. S. Brodal and L. Gasieniec: *Approximate dictionary queries*, in Combinatorial Pattern Matching, 7th Annual Symposium, CPM 96, Laguna Beach, California, USA, June 10-12, 1996, Proceedings, 1996, pp. 65–74.
16. S. Deguchi, F. Higashijima, H. Bannai, S. Inenaga, and M. Takeda: *Parameterized suffix arrays for binary strings*, in Proceedings of the Prague Stringology Conference 2008, Prague, Czech Republic, September 1-3, 2008, 2008, pp. 84–94.
17. 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.
18. 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 2016, June 27-29, 2016, Tel Aviv, Israel, 2016, pp. 2:1–2:12.
19. A. Ganguly, W. Hon, and R. Shah: *A framework for dynamic parameterized dictionary matching*, in 15th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2016, June 22-24, 2016, Reykjavik, Iceland, 2016, pp. 10:1–10:14.
20. T. Haapasalo, P. Silvasti, S. Sippu, and E. Soisalon-Soininen: *Online dictionary matching with variable-length gaps*, in Experimental Algorithms - 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings, 2011, pp. 76–87.
21. 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.
22. R. M. Idury and A. A. Schäffer: *Multiple matching of parametrized patterns.* Theor. Comput. Sci., 154(2) 1996, pp. 203–224.

23. O. Keller, T. Kopelowitz, and M. Lewenstein: *On the longest common parameterized subsequence*. Theor. Comput. Sci., 410(51) 2009, pp. 5347–5353.
24. M. Krishnamurthy, E. S. Seagren, R. Alder, A. W. Bayles, J. Burke, S. Carter, and E. Faskha: *How to cheat at securing linux*, in Syngress Publishing, Inc., Elsevier, Inc, 2008, pp. 1–432.
25. G. Kucherov and M. Rusinowitch: *Matching a set of strings with variable length don't cares*. Theor. Comput. Sci., 178(1-2) 1997, pp. 129–154.
26. M. Lewenstein: *Parameterized pattern matching*, in Encyclopedia of Algorithms, 2016, pp. 1525–1530.
27. J. Mendivelso and Y. Pinzón: *Parameterized matching: Solutions and extensions*, in Proceedings of the Prague Stringology Conference 2015, Prague, Czech Republic, August 24-26, 2015, 2015, pp. 118–131.
28. 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 Computing and Combinatorics, 12th Annual International Conference, COCOON 2006, Taipei, Taiwan, August 15-18, 2006, Proceedings, 2006, pp. 146–155.
29. 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.

Three Strategies for the Dead-Zone String Matching Algorithm

Jacqueline W. Daykin[1,2,3,5], Richard Groult[4,3], Yannick Guesnet[3], Thierry Lecroq[3], Arnaud Lefebvre[3], Martine Léonard[3], Laurent Mouchard[3], Élise Prieur-Gaston[3], and Bruce Watson[5,6]

[1] Department of Computer Science, Aberystwyth Univ., Wales & Mauritius
[2] Department of Informatics, Kings College London, UK
[3] Normandie Univ., UNIROUEN, LITIS, 76000 Rouen, France
[4] Modélisation, Information et Systèmes (MIS), Univ. de Picardie Jules Verne, Amiens, France
[5] Department of Information Science, Stellenbosch Univ., South Africa
[6] CAIR, CSIR Meraka, Pretoria, South Africa

Abstract. Online exact string matching consists in locating all the occurrences of a pattern in a text where only the pattern can be preprocessed. Classical online exact string matching algorithms scan the text from start to end through a window whose size is equal to the pattern length. Exact string matching algorithms from the dead-zone family first locate the window in the middle of the text, compare the content of the window and the pattern and then recursively apply the same procedure on the left part and on the right part of the text while possibly excluding some parts of the text. We propose three different strategies for performing the symbol comparisons and, we compute the shifts for determining the left and right parts of the text at each recursive call.

Keywords: exact string matching algorithms, dead-zone strategy, online, recursion

1 Introduction

The string matching problem consists in finding one or, more usually, all the occurrences of a pattern $x = x[0\,..\,m-1]$ of length m in a text $y = y[0\,..\,n-1]$ of length n. It can occur for instance in information retrieval, bibliographic search and molecular biology. It has been extensively studied and numerous techniques and algorithms have been designed to solve this problem (see [9,3,5,6]). We are interested here in the problem where the pattern is given first and can then be searched in various texts. Thus a preprocessing phase is only allowed on the pattern.

Most string matching algorithms use a window to scan the text. The size of this window is equal to the length of the pattern. They first align the left ends of the window and the text. Then they check if the pattern occurs in the window (this specific work is called an *attempt*) and they *shift* the window to the right. They repeat the same procedure again until the right end of the window goes beyond the right end of the text. String matching algorithms mostly differ in the way they compare the pattern and the window content, in the way they compute the length of the shifts and in the quantity of information they store from one attempt to the other, leading to a great number of algorithms.

As early as 1997 a new family of algorithms has been designed that do not fit in the sliding window strategy: it consists in first locating the window in the middle of the text, performing an attempt and then recursively applying the same procedure on the left part and on the right part of the text, while possibly excluding some parts

Jacqueline W. Daykin, Richard Groult, Yannick Guesnet, Thierry Lecroq, Arnaud Lefebvre, Martine Léonard, Laurent Mouchard, Élise Prieur-Gaston, Bruce Watson: Three Strategies for the Dead-Zone String Matching Algorithm, pp. 117–128.
Proceedings of PSC 2018, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06484-9

of the text giving the "dead-zone" method [12,10,11,8]. Algorithms from this family are highly parallelizable.

This strategy has not attracted much attention. To address this, here we present three different methods for performing the symbol comparisons during the attempts and for computing the lengths of the shifts.

The remainder of the paper is organized as follows: in Section 2 we give the formal notions and notation used throughout the paper. Section 3 presents the basic techniques used for online string matching and recalls the famous Knuth-Morris-Pratt and Boyer-Moore algorithms. In the next three sections we give new strategies for the dead-zone method: a right-to-left memoryless strategy in Section 4, a right-to-left strategy with memory in Section 5 and an alternating strategy in Section 6. Finally we give our conclusion and perspectives in Section 7.

2 Notation

Consider a finite totally ordered alphabet Σ of constant size $|\Sigma| = \sigma$ which consists of a set of symbols. A *string* is a sequence of zero or more symbols over Σ. The set of all non-empty strings over Σ is denoted by Σ^+. The empty string is the null sequence of symbols (hence of zero length) and is denoted by ε; furthermore, $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.

A string x over Σ of length $|x| = m$ is represented by $x[0\,..\,m-1]$, where $x[i] \in \Sigma$ for $0 \leq i \leq m-1$ is the $(i+1)$-th symbol of x.

The reverse of a string x (i.e. the string read from right to left) is denoted by $\tilde{x}$.

The concatenation of two strings x and y is defined as the sequence of symbols of x followed by the sequence of symbols of y and is denoted by $x \cdot y$ or simply xy when no confusion is possible. A string x is a *factor* of y if $y = uxv$, where $u, v \in \Sigma^*$; specifically a string $x = x[0\,..\,m-1]$ is a factor of y if $x[0\,..\,m-1] = y[j\,..\,j+m-1]$ for some j.

Strings $u = y[0\,..\,i]$ are called *prefixes* of y, and strings $v = y[i\,..\,m-1]$ are called *suffixes* of y. The prefix u (respectively, suffix v) is a proper prefix (suffix) of a string y if $y \neq u, v$.

A border of a non-empty string x is a factor u of x which is both a proper prefix and a proper suffix of x. We denote by $border(x)$ the longest border of x and by $per(x) = |x| - border(x)$ the smallest period of x.

For $0 \leq i \leq m-1$:

- $pref[i]$ = length of the longest common prefix of x and $x[i\,..\,m-1]$;
- $suff[i]$ = length of the longest common suffix of x and $x[0\,..\,i]$.

Given a string x of length m, the values $border(x)$ and $per(x)$ and arrays $pref$ and $suff$ can be computed in $O(m)$ time (see [4]).

We are interested in finding all the occurrences of a pattern $x = x[0\,..\,m-1]$ of length $|x| = m$ in a text $y = y[0\,..\,n-1]$ of length $|y| = n$. We will apply the Dead-Zone strategy.

3 Background on online exact string matching algorithms

Exact string matching consists in finding one, or more generally, all the occurrences of a string x (usually called a *pattern* in the literature) of length m in a *text* y of length n. In its online version only the pattern x can be preprocessed before the searching phase.

3.1 Sliding window mechanism

Online exact string matching algorithms work as follows. They scan the text with the help of a *window* whose size is generally equal to m. They first align the left ends of the window and the text, then compare the symbols of the window with the symbols of the pattern — this specific work is called an *attempt* — and after a whole match of the pattern or after a mismatch they *shift* the window to the right. They repeat the same procedure again until the right end of the window goes beyond the right end of the text. This mechanism is usually called the *sliding window mechanism*. We associate each attempt with the positions j and $j+m-1$ in the text when the window is positioned on $y[j\,..\,j+m-1]$: we say that the attempt is at the left position j and at the right position $j+m-1$.

3.2 Knuth-Morris-Pratt algorithm

Consider an attempt at a left position j, that is when the window is positioned on the text factor $y[j\,..\,j+m-1]$. Assume that the first mismatch occurs between $x[i]$ and $y[i+j]$ with $0 < i < m$. Then, $x[0\,..\,i-1] = y[j\,..\,i+j-1] = u$ and $a = x[i] \neq y[i+j] = b$. When shifting, it is reasonable to expect that a prefix v of the pattern matches some suffix of the portion u of the text. Moreover, if we want to avoid another immediate mismatch, the symbol following the prefix v in the pattern must be different from a. The longest such prefix v is called the tagged border of u (it occurs at both ends of u followed by different symbols in x). This introduces the notation: let $bp_x[i]$ be the length of the longest border of $x[0\,..\,i-1]$ followed by a symbol c different from $x[i]$ and -1 if no such tagged border exists, for $0 < i \leq m$. Then, after a shift, the comparisons can resume between symbols $x[bp_x[i]]$ and $y[i+j]$ without missing any occurrence of x in y, and avoiding a backtrack on the text. The table bp_x can be computed in $O(m)$ space and time (see [4]) before the searching phase, applying the same searching algorithm to the pattern itself, as if $x = y$.

The best prefix array bp_x for x with $m+1$ elements is defined as follows for $0 \leq i \leq m$:

$$bp_x[i] = \begin{cases} -1 & \text{if } \ i = 0 \\ |border(x[0\,..\,i-1])| & \text{if } \ x[|border(x[0\,..\,i-1])|] \neq x[i] \\ bp_x[|border(x[0\,..\,i-1])|] & \text{otherwise.} \end{cases}$$

3.3 Boyer-Moore algorithm

The Boyer-Moore algorithm scans the symbols of the pattern from right to left beginning with the rightmost one. In the case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the window to the right. These two shift functions are called the *good suffix shift* (also called matching shift) and the *bad character shift* (also called occurrence shift).

Assume that a mismatch occurs between the character $x[i] = a$ of the pattern and the character $y[i+j] = b$ of the text during an attempt at the left position j. Then, $x[i+1\,..\,m-1] = y[i+j+1\,..\,j+m-1] = u$ and $x[i] \neq y[i+j]$. The good suffix shift consists in aligning the segment $y[i+j+1\,..\,j+m-1] = x[i+1\,..\,m-1]$ with its rightmost occurrence in x that is preceded by a symbol different from $x[i]$. If no such segment exists, the shift consists in aligning the longest suffix v of $y[i+j+1\,..\,j+m-1]$ with a matching prefix of x.

Formally we define two conditions, for $0 \leq i \leq m-1$ and $1 \leq d \leq m$ as follows. The suffix condition $suffCond_x$:

$$suffCond_x(i,d) = \begin{cases} 0 < d \leq i+1 \text{ and } x[i-d+1\,..\,m-d-1] \text{ is a suffix of } x \\ \text{or} \\ i+1 < d \text{ and } x[0\,..\,m-d-1] \text{ is a suffix of } x. \end{cases}$$

The occurrence condition $occCond_x$:

$$occCond_x(i,d) = \begin{cases} 0 < d \leq i \text{ and } x[i-d] \neq x[i] \\ \text{or} \\ i < d. \end{cases}$$

Then the good suffix array for x is defined as follows, for $0 \leq i \leq m-1$:

$$gs_x[i] = \min\{d \mid suffCond_x(i,d) \text{ and } occCond_x(i,d) \text{ are satisfied }\}.$$

The bad character shift consists in aligning the text symbol $y[i+j]$ with its rightmost occurrence in $x[0\,..\,m-2]$. If $y[i+j]$ does not occur in the pattern x, no occurrence of x in y can overlap with $y[i+j]$, and the left end of the window is aligned with the symbol immediately after $y[i+j]$, namely $y[i+j+1]$.

Note that the bad character shift can be negative, thus for shifting the window, the Boyer-Moore algorithm applies the maximum between the good suffix shift and the bad character shift.

4 Right-to-left searching strategy for the Dead-Zone method

One searching strategy for the Dead-Zone method consists in scanning the symbols of the window from right to left in the same manner as in the Boyer-Moore algorithm [2]. Then, when a mismatch occurs or when an occurrence of the pattern is found, a right shift and a left shift has to be applied in order to compute the right and left parts where the recursion will apply. The general situation is the following: a suffix v of the pattern has been matched in the text and a mismatch occurred between a symbol a at position i in the pattern and a symbol d in the text. The right shift consists in finding a re-occurrence of v in the pattern preceded by a symbol b different from a, and the left shift consists in finding the longest suffix v' of the pattern preceded by a symbol c different from a (see Figure 1).

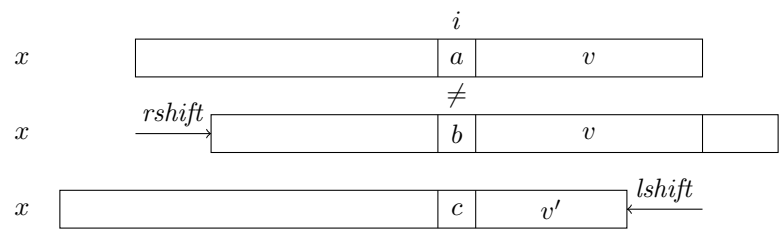


Figure 1. When a suffix v of the pattern has been matched in the text and a mismatch occurred between a symbol a at position i in the pattern. The right shift (*rshift*) consists in finding a re-occurrence of v in the pattern preceded by a symbol b different from a, and the left shift (*lshift*) consists in finding the longest suffix v' of the pattern preceded by a symbol c different from a.

The right shift can be computed by the good suffix array of the Boyer-Moore algorithm while the left shift can be computed by the best prefix array of the Knuth-Morris-Pratt algorithm [7] for the reverse pattern (see [4]).

Specifically, when a mismatch occurs at position i of the pattern the right shift is given by $gs_x[i]$ and the left shift is given by $m-1-i-bp_{\tilde{x}}[m-1-i]$ (where $bp_{\tilde{x}}$ is the best prefix array for $\tilde{x}$ the reverse of x). Algorithm DZ-R2L below implements this strategy. Parameters b and e are respectively the beginning and the end positions on the text and $C \geq 1$ is any small constant that will enable the recursion to stop. When the recursion stops the search is done by a brute force algorithm BF.

```
DZ-R2L(x, m, y, b, e)
 1  if e − b < Cm then
        ▷ Stop the recursion
 2      BF(x, m, y, b, e)
 3  else j ← (e + b + 1)/2 − m/2
 4      i ← m − 1
        ▷ Scan
 5      while i ≥ 0 and x[i] = y[j + i] do
 6          i ← i − 1
 7      if i < 0 then
 8          OUTPUT(j)
 9          rshift ← per(x)
10          lshift ← per(x)
11      else rshift ← gs_x[i]
12          lshift ← m − 1 − i − bp_x̃[m − 1 − i]
        ▷ Left recursive call
13      DZ-R2L(x, m, y, b, j + m − 1 − lshift)
        ▷ Right recursive call
14      DZ-R2L(x, m, y, j + rshift, e)
```

The following theorem can be easily proved.

Theorem 1. *The algorithm* DZ-R2L$(x, m, y, 0, n-1)$ *finds all the occurrences of* x *in* y *and runs in* $O(mn)$ *time.*

5 Right-to-left strategy with memory for the Dead-Zone method

For the Dead-Zone method, it is possible to implement a strategy with memory similalry to the Apostolico-Giancarlo (AG) algorithm [1] for the Boyer-Moore algorithm. The AG algorithm stores for each rightmost position of the window the length ℓ of the longest suffix of the pattern ending at that position. Then if, during the right-to-left scan of an attempt, it reaches a position where this length ℓ is non-null, the algorithm can make a decision and end the attempt without any further symbol comparisons as in most cases, otherwise it can jump over the factor of the text of length ℓ. In that case, symbols of the text are positively compared only once.

The main difference here is that for the AG algorithm there is only one window and we may access only the rightmost position of a previously matched suffix of the pattern, while for the Dead-Zone method we may access any symbol inside a

previously matched suffix of the pattern. Here, during an attempt at left position j when a mismatch occurs with position i of the pattern or $i = -1$ because an occurrence of the pattern has been found, when a symbol at a position k of the pattern x is matched with a symbol at position $j+k$ of the text y during the attempt we need to store, for this position $j + k$, the position k and the length $k - i$. These items of information are stored in arrays $skip_1$ and $skip_2$ respectively. Because when a match occurs at position k, the mismatch position i is not known yet, a stack is used during each attempt for storing all the matching positions. Then at the end of the attempt these positions are popped in order to fill the corresponding values of arrays $skip_1$ and $skip_2$.

Note that during an attempt at left position j, if a position $i + j$ is reached such that $k = skip_2[i + j] > 0$, it means that $x[k - \ell + 1\,..\,k] = y[i + j - \ell + 1\,..\,i + j]$ with $\ell = skip_1[i + j]$, and furthermore $x[k - \ell] \neq y[i + j - \ell]$ if $k \geq \ell$. We need to know whether $y[i + j - \ell + 1\,..\,i + j] = x[i - \ell + 1\,..\,i]$ and thus we need to know whether $x[k - \ell + 1\,..\,k] = x[i - \ell + 1\,..\,i]$: this information can be obtained by computing the longest common prefix of the suffixes of $\tilde{x}$ starting at positions $m - 1 - k$ and $m - 1 - i$. This can be done in constant time by computing the suffix array and the longest common prefixes (LCP) array of $\tilde{x}$ and preparing the latter for answering Range Minimum Queries (RMQ).

Then, let q be the length of the longest common prefix of $\tilde{x}[m - 1 - k\,..\,m - 1]$ and $\tilde{x}[m - 1 - i\,..\,m - 1]$. If $q = \ell$ then $x[k - \ell + 1\,..\,k] = x[i - \ell + 1\,..\,i]$ and if $k \geq \ell$ it also holds that $x[k - \ell] \neq x[i - \ell]$ then $y[i + j - \ell + 1\,..\,i + j] = x[i - \ell + 1\,..\,i]$ and $x[i - \ell]$ and $y[i + j - \ell]$ need to be compared.

Algorithm DZ-R2L-WITH-MEMORY below implements this strategy.

```
DZ-R2L-WITH-MEMORY(x, m, y, b, e)
 1  if e − b < Cm then
        ▷ Stop the recursion
 2      BF(x, m, y, b, e)
 3  else (i, j) ← (m − 1, (e + b + 1)/2 − m/2)
 4      S ← ∅
        ▷ Scan
 5      while i ≥ 0 do
 6          (k, ℓ) ← (skip_1[i + j], skip_2[i + j])
 7          if ℓ > 0 then
 8              q ← RMQ(SA_x̃, m − 1 − k, m − i − 1)
 9              if ℓ ≠ q then
10                  i ← i − min{ℓ, d}
11                  break
12              else i ← i − q
13          else if x[i] = y[i + j] then
14                  PUSH(S, i)
15                  i ← i − 1
16              else break
17      while S ≠ ∅ do
18          k ← POP(S)
19          (skip_1[j + k], skip_2[j + k]) ← (k, k − i)
20      if i < 0 then
21          OUTPUT(j)
22          (rshift, lshift) ← (per(x), per(x))
23      else (rshift, lshift) ← (gs_x[i], m − 1 − i − bp_x̃[m − 1 − i])
        ▷ Left recursive call
24      DZ-R2L-WITH-MEMORY(x, m, y, b, j + m − 1 − lshift)
        ▷ Right recursive call
25      DZ-R2L-WITH-MEMORY(x, m, y, j + rshift, e)
```

We can conjecture that, as with the AG algorithm, the DZ-R2L-WITH-MEMORY algorithm runs in linear time in the worst case.

6 Alternating searching strategy: right – left for the Dead-Zone method

In order to try to optimize the lengths of the right and left shifts, it is possible to design an alternating strategy for the Dead-Zone algorithm. At each attempt, the window is placed is the middle of the text. The scanning is performed by comparing alternately the right and the left ends of the window until a complete match or a mismatch occurs. After each attempt two recursive calls are performed on the left and on the right parts of the text.

Then, at each attempt, a mismatch can occur either on the left or on the right and accordingly a shift has to be computed in the left and in the right which leads to 4 shift functions stored in 4 arrays:

- right shift after a right mismatch stored in array *rsrm*;
- left shift after a right mismatch stored in array *lsrm*;

– right shift after a left mismatch stored in array *rslm*;
– left shift after a left mismatch stored in array *lslm*.

Let us define two conditions $suffCond'_x$ and $occCond'_x$ as follows. For $0 \le i \le m-1$ and $1 \le d \le m$,

$occCond'_x(i,d) = (0 < d \le i \text{ and } x[i-d] \ne x[i])$ or $(i < d)$

$$\begin{aligned} suffCond'_x(i,d) = (\ & 0 < d \le m-2-i \\ & \text{and } x[0\,..\,m-2-i-d] = x[d\,..\,m-2-i] \\ & \text{and } x[i-d+1\,..\,m-d-1] = x[i+1\,..\,m-1]\ ) \\ \text{or}\ & \\ (\ & m-2-i < d \le i+1 \\ & \text{and } x[i-d+1\,..\,m-d-1] = x[i+1\,..\,m-1]\ ) \\ \text{or}\ & \\ (\ & i+1 < d \text{ and } x[0\,..\,m-d-1] = x[d\,..\,m-1]\ \ ) \end{aligned}$$

The latter can be rewritten as

$$\begin{aligned} suffCond'_x(i,d) = (\ & 0 < d \le m-2-i \\ & \text{and } x[d\,..\,m-2-i] \text{ is a prefix of } x \\ & \text{and } x[i-d+1\,..\,m-d-1] \text{ is a suffix of } x\ \ ) \\ \text{or}\ & \\ (\ & m-2-i < d \le i+1 \\ & \text{and } x[i-d+1\,..\,m-d-1] \text{ is a suffix of } x\ \ ) \\ \text{or}\ & \\ (\ & i+1 < d \text{ and } x[0\,..\,m-d-1] \text{ is a suffix of } x\ ). \end{aligned}$$

Then the array *rsrm* can be defined as follows for $0 \le i \le m-1$:
$rsrm[i] = \min\{d \mid occCond'_x(i,d)$ and $suffCond'_x(i,d)$ are satisfied$\}$.
Three cases can arise as shown in Figure 2.

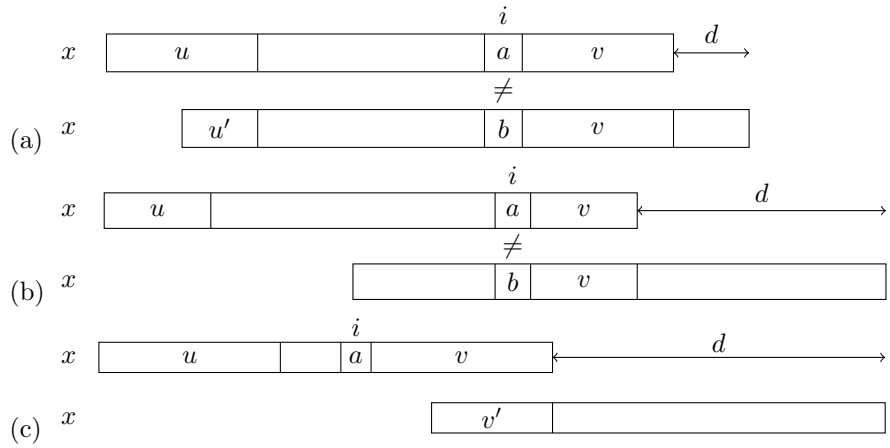


Figure 2. Right shift after a right mismatch: assume that prefix u and suffix v (of the same length) of x match the text and that a mismatch occurs with symbol a at position i of x: (a) in this case the suffix v of x reoccurs preceded by a symbol b different from a and a prefix u' of x matches a suffix of u; (b) in this case only a suffix v of x reoccurs preceded by a symbol b different from a; (c) in this case only a prefix v' of x matches a suffix of v.

Arrays *lsrm*, *rslm* and *lslm* can be defined similarly to array *rsrm*. Then algorithm DZ-Alt given below implements the alternating strategy.

```
DZ-ALT(x, m, y, b, e)
 1  if e − b < Cm then
        ▷ Stop the recursion
 2      BF(x, m, y, b, e)
 3  else j ← (e + b + 1)/2 − m/2
 4      i ← 0
 5      while i ≤ m/2 do
            ▷ Right scan
 6          if x[m − 1 − i] ≠ y[j + m − 1 − i] then
 7              rshift ← rsrm[m − 1 − i]
 8              lshift ← lsrm[m − 1 − i]
 9              break
            ▷ Left scan
10          if x[i] ≠ y[j + i] then
11              rshift ← rslm[i]
12              lshift ← lslm[i]
13              break
14          i ← i + 1
15      if i > m/2 then
16          OUTPUT(j)
17          rshift ← per(x)
18          lshift ← per(x)
        ▷ Left recursive call
19      DZ-ALT(x, m, y, b, j + m − 1 − lshift)
        ▷ Right recursive call
20      DZ-ALT(x, m, y, j + rshift, e)
```

6.1 Right Shift after a Right Mismatch

We will now show how to compute the array *rsrm* efficiently.

Lemma 2. *For* $0 \le i \le m-1$*:* $rsrm[i] \le m - suff[j]$ *where* $j = \max\{k \mid 0 \le k < m-1-i \text{ and } suff[k] = k+1\}$.

Proof. See Figure 3. We will show that $occCond'_x(j,d)$ and $suffCond'_x(j,d)$ are both satisfied with $j = i$ and $d = m-k-1$. If $0 \le k < m-1-i$ then $i < m-k-1$. If $suff[k] = k+1$ it means that $x[0\,..\,k]$ is a suffix of x. Then $occCond'_x(i, m-k-1)$ and $suffCond'_x(i, m-k-1)$ are both satisfied. Thus $rsrm[i] \le m-k-1$. □

The following two lemmas can be proved similarly.

Lemma 3. *For* $0 \le i \le m-1$*:* $rsrm[m-1-suff[i]] \le m-1-i$ *if* $m-1-i \ge suff[i]$.

Lemma 4. *For* $0 \le i \le m-1$*:* $rsrm[m-1-suff[i]] \le m-1-i$ *if* $m-1-i < suff[i]$ *and* $pref[m-1-i] \ge suff[i] - m + 1 + i$.

Then the following algorithm RSRM computes the array *rsrm*.

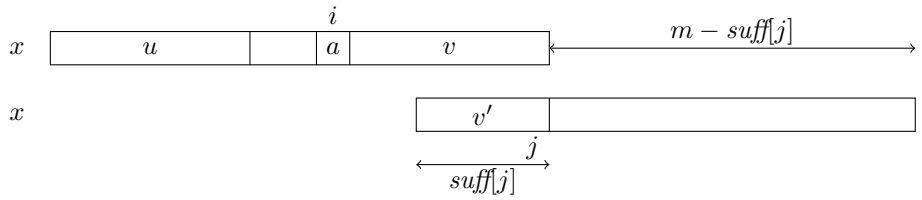


Figure 3. Right shift after a right mismatch: assume that prefix u and suffix v of x match the text and that a mismatch occurs with symbol a at position i of x then the length of the right shift cannot be larger than $m - \mathit{suff}[j]$.

```
RSRM(x, m, suff, pref)
 1  j ← 0
    ▷ Lemma 2
 2  for i ← m − 2 downto −1 do
 3     if i = −1 or suff[i] = i + 1 then
 4        while j < m − 1 − i do
 5           rsrm[j] ← m − 1 − i
 6           j ← j + 1
    ▷ Lemmas 3 and 4
 7  for i ← 0 to m − 2 do
 8     if m − 1 − i ≥ suff[i] or pref[m − 1 − i] ≥ suff[i] − m + 1 + i then
 9        rsrm[m − 1 − suff[i]] ← m − 1 − i
10  return rsrm
```

Lemma 5. *Algorithm* RSRM$(x, m, \mathit{suff}, \mathit{pref})$ *is correct and runs in linear time.*

Proof. Correctness comes from Lemmas 2–4. Time complexity analysis is similar to the analysis of the good suffix array (see [4]). □

6.2 Right Shift after a Left Mismatch

Let *rpref* be the array *pref* for $\tilde{x}$ and let *rsuff* be the array *suff* for $\tilde{x}$. The following four lemmas can be proved similarly to Lemmas 2–4.

For $0 \leq i \leq m - 1$:

Lemma 6. $\mathit{rslm}[i] < m - \mathit{suff}[j]$ *where* $j = \max\{k \mid 0 \leq k \leq i \text{ and } \mathit{suff}[k] = k + 1\}$.

Lemma 7. $\mathit{rslm}[i] \leq \min\{m - j \mid \mathit{suff}[j] \geq m - i\}$.

Lemma 8. $\mathit{rslm}[m-1-i+\mathit{rsuff}[i]] \leq m-1-i$ *if* $\mathit{rpref}[m-1-i] \geq m-1-i+\mathit{rsuff}[i]$.

Lemma 9. $\mathit{rslm}[m - 1 - \mathit{suff}[i]] \leq m - 1 - i$ *if* $\mathit{pref}[m - 1 - i] \geq \mathit{suff}[i] + m - 1 - i$.

Then the following algorithm RSLM computes the array *rslm*.

```
RSLM(x, m, suff, rpref, rsuff)
 1  j ← m/2 − 1
 2  for i ← m − 2 to −1 do
 3     if i = −1 or suff[i] = i + 1 then
 4        rslm[j] ← m − 1 − i
 5        j ← j − 1
 6  i ← 1
 7  j ← m − 2
 8  while i < m/2 do
 9     while j ≥ 0 and suff[j] < i do
10        j ← j − 1
11     if j < 0 then
12        break
13     else while i < m − j do
14           rslm[i − 1] ← m − 1 − j
15           i ← i + 1
16        j ← j − 1
17  for i ← 0 to m − 2 do
18     if rpref[m − 1 − i] ≥ rsuff[i] + m − 1 − i then
19        rslm[rsuff[i] + m − 1 − i] ← m − 1 − i
20  return rslm
```

Lemma 10. *Algorithm* RSLM$(x, m, suff, rpref, rsuff)$ *is correct and runs in linear time.*

Proof. Similar to the proof of Lemma 5. □

Arrays *lsrm* and *lslm* can be computed in linear time using similar methods. Thus the preprocessing phase of algorithm DZ-Alt requires linear time.

7 Conclusion and perspectives

We presented three strategies for the dead-zone exact string matching method: a memoryless right-to-left strategy, a right-to-left with memory strategy and an alternating strategy. It remains to state the time complexity exactly for the right-to-left with memory strategy that we conjectured to be linear. That would be the first linear strategy designed for the dead-zone method. An experimental study will also be necessary to assess the practical performances of the different strategies both in terms of running times and the number of symbol comparisons. Also the fact that such algorithms are highly parallelizable should be exploited.

Acknowledgements

The first author was part-funded by the European Regional Development Fund through the Welsh Government.

References

1. A. Apostolico and R. Giancarlo: *The Boyer-Moore-Galil string searching strategies revisited*. SIAM J. Comput., 15(1) 1986, pp. 98–105.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm*. Commun. ACM, 20(10) 1977, pp. 762–772.
3. C. Charras and T. Lecroq: *Handbook of exact string matching algorithms*, King's College London Publications, 2004.
4. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on strings*, Cambridge University Press, 2007.
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. Borzi, S. D. Mauro, and A. Maggio: *The string matching algorithms research tool*, in Proceedings of the Prague Stringology Conference 2016, Prague, Czech Republic, August 29-31, 2016, J. Holub and J. Zdárek, eds., Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2016, pp. 99–111.
7. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt: *Fast pattern matching in strings*. SIAM J. Comput., 6(1) 1977, pp. 323–350.
8. M. Mauch, D. G. Kourie, B. W. Watson, and T. Strauss: *Performance assessment of dead-zone single keyword pattern matching*, in 2012 South African Institute of Computer Scientists and Information Technologists Conference, SAICSIT '12, Pretoria, South Africa, October 1-3, 2012, J. H. Kroeze and R. de Villiers, eds., ACM, 2012, pp. 59–68.
9. G. Navarro and M. Raffinot: *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*, Cambridge University Press, 2002.
10. B. Watson and R. Watson: *A new family of string pattern matching algorithms*. South African Computer Journal, 30 2003, pp. 34–41.
11. B. W. Watson, D. G. Kourie, and T. Strauss: *A sequential recursive implementation of dead-zone single keyword pattern matching*, in Combinatorial Algorithms, 23rd International Workshop, IWOCA 2012, Tamil Nadu, India, July 19-21, 2012, Revised Selected Papers, S. Arumugam and W. F. Smyth, eds., vol. 7643 of Lecture Notes in Computer Science, Springer, 2012, pp. 236–248.
12. B. W. Watson and R. E. Watson: *A new family of string pattern matching algorithms*, in Proceedings of the Prague Stringology Club Workshop 1997, Prague, Czech Republic, July 7, 1997, J. Holub, ed., Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, 1997, pp. 12–23.

Author Index

Proceedings of the Prague Stringology Conference 2018
Edited by Jan Holub and Jan Žďárek
Published by: Prague Stringology Club
Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9, Praha 6, 160 00, Czech Republic.

ISBN 978-80-01-06484-9

URL: `http://www.stringology.org/`
E-mail: `psc@stringology.org` Phone: +420-2-2435-9811

Printed by Česká technika – Nakladatelství ČVUT
Zikova 4, Praha 6, 166 36, Czech Republic