Proceedings of the Prague Stringology Conference 2020

Edited by Jan Holub and Jan Žďárek

August 2020

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

ISBN 978-80-01-06749-9

Preface

The proceedings in your hands contains a collection of papers presented in the Prague Stringology Conference 2020 (PSC 2020) held on August 31–September 2, 2020 at the Czech Technical University in Prague, which organizes the event. The conference focused on stringology, i.e., a discipline concerned with algorithmic processing of strings and sequences, and related topics.

The submitted papers were reviewed by the program committee subject to originality and quality. The fourteen papers in this proceedings made the cut and were selected for regular presentation at the conference.

The PSC 2020 was quite unique as it was organized in both present and remote form. Due to COVID-19 pandemic situation there were restrictions for traveling. They were less and less strict as the date of the conference was approaching. So at the end, some participants arrived to Prague, some were connected remotely.

The Prague Stringology Conference has a long tradition. PSC 2020 is the twenty-third PSC conference. In the years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conferences (PSC's) in 2001–2006, 2008–2019 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 2020 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 2020. 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 2020

Jan Holub and Simone Faro

Conference Organisation

Program Committee

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

Organising Committee

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

External Referees

Golnaz Badkobeh	Diptarama Hendrian	Elise Prieur-Gaston
Hideo Bannai	Arnaud Lefebvre	Simon Puglisi
Itai Boneh	Neerja Mhaskar	Tatiana Starikovskaya

Table of Contents

Contributed Talks

New Compression Schemes for Natural Number Sequences

Sapir Asraf[1], Shmuel T. Klein[2], and Dana Shapira[1]

[1] Dept. of Computer Science, Ariel University, Ariel 40700, Israel
asrafsapir@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. Elias and Fano independently proposed a quasi-succinct representation for monotonic integer sequences. In case the standard deviation is high, we suggest using the well known C_γ code instead of the Unary code used by their solution. In case the integers are similar, not necessarily forming a monotonic sequence, we propose to apply the Haar transform as a preprocessing stage, to achieve additional savings. Experimental results support the additional savings carried out by using our method.

Keywords: lossless compression, universal codes, the Haar transform

1 Introduction

Fixed length codes, such as the *American Standard Code for Information Interchange* ASCII code, are the most popular method to store data, as they provide simplicity, direct access and the possibility for fast retrieval. When compression performance is of interest, variable length codes are usually more effective. Obviously, the codes should be *Uniquely Decipherable* (UD), meaning that there is no ambiguous decoding. In case no codeword is a prefix of any of the other codewords, the code is often called a *Prefix-free Code*, and such a code is also UD. The restriction to prefix-free codes does not hurt the compression performance. Famous prefix-free variable length codes are, for instance, Huffman [12], Elias [5] and Fibonacci [7] codes.

Elias [5] proposed mainly two *fixed*, universal, prefix codeword sets, named C_γ and C_δ, in which any integer x is represented by a binary codeword composed of two parts. The first part listing the number of bits in the binary representation of x, and the second storing the standard binary representation itself without its leading 1-bit. While the first part is encoded by C_γ using the Unary encoding, C_δ uses C_γ. There is no difference between C_γ and C_δ in the second part. The expected codeword lengths are within twice the optimal average codeword length for the same underlying source for C_γ, and only a log log factor away from optimal for C_δ. More precisely, C_γ requires $2\lfloor \log x \rfloor + 1$ and C_δ necessitates $\lfloor \log x \rfloor + 1 + 2\lfloor \log(\lfloor \log x \rfloor + 1) \rfloor$ bits to encode the number x.

A classical way to encode a monotonic set of integers is *differential encoding*, also called *gap encoding*. In this method, instead of encoding the original set $0 \leq x_1 \leq x_2 \leq \cdots \leq x_n \leq U$, the set of differences is encoded. If the sequence of differences is encoded by C_δ, then the number of bits used is no more than $n \log \frac{U}{n} + 2n \log\log \frac{U}{n} + O(n)$, which is close to the zero-order entropy of a bit-vector of size U with n 1-bits [14].

There are quite a few motivations for the interest in monotonic sequence, an *Inverted Index* is one of them. This is a powerful data structure commonly used in Information Retrieval to enhance the processing time of search engines. Given

Sapir Asraf, Shmuel T. Klein, Dana Shapira: New Compression Schemes for Natural Number Sequences, pp. 1–10.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9

a collection of documents, the inverted index is the list of documents where each element of the collection occurs in, possibly including the frequency as well as the exact positions of the element within each document. For a text T, the inverted index stores for each element w with n_w occurrences, the positions $x_1 < x_2 < \cdots < x_{n_w}$ within T where w occurs. As this list is usually given in order, the resulting sequence of integers is increasing.

Increasing sequences can also be found in *Compressed Suffix Arrays.* A suffix array (*SA*) for $T\$$, where T is a string of length n over Σ and $\$ \notin \Sigma$, is an array $SA[0:n-1]$ of the indices of the suffixes of $T\$$, stored in lexicographical order. Grossi and Vitter [10] improve the space requirements of a suffix array by decomposing it based on the *neighbor function* Φ. It has been shown that the values of Φ at consecutive positions referring to suffixes that start with the same symbol must be increasing. The implementation of CSA used in [1,13] applies differential encoding on the neighbor function. Improved compression results were proposed by Gog et al. [8] who suggest using the *Elias-Fano* encoding for storing the increasing Φ values of the CSA.

Our paper is constructed as follows. Section 2 recalls the details of Elias-Fano codes, and suggests a variant that uses $C\gamma$ rather than the Unary code used by Elias-Fano. We show that the original Elias-Fano is suitable for homogeneous series, while the new variant is effective for series with higher standard deviation. Section 3 then suggests the Haar transform as a preprocessing stage in order to convert homogeneous numbers to a series which is suitable for the new C_γ variant. Experimental results presented in Section 4 then support the savings of the proposed method.

2 Quasi-succinct representation for monotone sequences

Gap encoding can be used in order to compress inverted indices. Instead of encoding the non-decreasing list of integers $0 \le x_1 \le x_2 \le \cdots \le x_n$, directly pointing to the ordered set of documents, the differences $d_1 = x_1,\ d_2 = x_2 - x_1,\ \ldots,\ d_n = x_n - x_{n-1}$ are encoded, usually by universal codes such as Elias, Golomb [9] or Rice codes.

Elias [4] and Fano [6] independently proposed an efficient encoding method for representing a non-decreasing sequence $\mathcal{X}$ of positive integers

$$\mathcal{X} = \{0 \le x_1 \le x_2 \le \cdots \le x_n \le U\},$$

where U is a given upper bound on x_n, possibly equal to x_n.

The sequence $\mathcal{X}$ is represented by two separate bit-vectors $\mathcal{L}(\ell)$ and $\mathcal{U}(\ell)$, for storing the ℓ lower bits, and the differences between successive values of the remaining upper bits of each x_i, respectively. More precisely, the ℓ least significant bits of each $x_i \in \mathcal{X}$, which are $x_i \bmod 2^\ell$, are stored sequentially in $\mathcal{L}(\ell)$ in the same order as they appear in $\mathcal{X}$. The value of the binary representation of the remaining bits of each $x_i \in \mathcal{X}$, that is, the values $y_i = \lfloor \frac{x_i}{2^\ell} \rfloor$ are then considered, and the differences between adjacent values $\Delta(i) = y_i - y_{i-1}$, $1 \le i \le n$, are computed, setting $y_0 = 0$. $\mathcal{U}(\ell)$-Unary stores these differences $\Delta(i)$ in the same order as in $\mathcal{X}$ in a Unary encoding, that is, representing the integers $1, 2, 3, \ldots, i$ respectively by $1, 01, 001, \ldots, 0^{i-1}1$. Elias-Fano's method defines ℓ to be equal to $\max\{0, \lfloor \log(\frac{U}{n}) \rfloor\}$. A similar encoding of the indices of 1-bits in a sparse bit-vector, in which the sequence $\mathcal{U}(\ell)$ is replaced by a bit-vector, is described in [2].

Table 1 displays the representation of the Elias-Fano code applied on the monotonic sequence example 2, 3, 10, 16, and 52. The first and second lines of Table 1

give the sequence $\mathcal{X}$ and their binary representation $\mathcal{B}(x_i)$. According to Elias-Fano, $\ell = \max\{0, \lfloor \log(\frac{U}{n}) \rfloor\} = 3$ for our example, and the third line presents the lower bits vector $\mathcal{L}$ for $\ell = 3$. The next block of four lines are the stages for constructing $\mathcal{U}$-UNARY for this example, given at the end of this block. The following two lines, headed by $\mathcal{B}\big(\lfloor \frac{x_i}{2^\ell} \rfloor\big)$ and $\lfloor \frac{x_i}{2^\ell} \rfloor$, are the remaining bits in each x_i and their corresponding values after their 3 lower bits have been removed. The line headed by $\Delta(i)$ is the differences between adjacent values of the previous line. Elias-Fano uses 26 bits in total. The last five lines of Table 1 are explained below.

	x_1	x_2	x_3	x_4	x_5
$\mathcal{X}$	2	3	10	16	52
$\mathcal{B}(x_i)$	10	11	1010	10000	110100
$\mathcal{L}(3)$	010	011	010	000	100
$\mathcal{B}\big(\lfloor \frac{x_i}{2^\ell} \rfloor\big)$	0	0	1	10	110
$\lfloor \frac{x_i}{2^\ell} \rfloor$	0	0	1	2	6
$\Delta(i)$	0	0	1	1	4
$\mathcal{U}(3)$-UNARY	1	1	01	01	00001
$\mathcal{U}(3)$-C_γ	1	1	010	010	00101
$\mathcal{L}(2)$	10	11	10	00	00
$\mathcal{B}\big(\lfloor \frac{x_i}{2^\ell} \rfloor\big)$	0	0	10	100	1101
$\lfloor \frac{x_i}{2^\ell} \rfloor$	0	0	2	4	13
$\Delta(i)$	0	0	2	2	9
$\mathcal{U}(2)$-C_γ	1	1	011	011	0001010

Table 1. Quasi Succinct Encoding [4] for the sequence 2, 3, 10, 16 and 52

Exactly $\ell \cdot n$ bits are used for storing the lower bits vector $\mathcal{L}(\ell)$. Next, we compute the number of bits used by the upper bits vector $\mathcal{U}(\ell)$-UNARY. The Unary code records the values $y_i - y_{i-1} = \frac{x_i}{2^\ell} - \frac{x_{i-1}}{2^\ell}$. If this difference is c, then x_i is larger than x_{i-1} by at least $c \cdot 2^\ell$. The total differences can obviously not be larger than $\frac{x_n}{2^\ell}$, the latter being bounded in case the definition for ℓ is used, explained as follows.

$$\left\lfloor \frac{x_n}{2^\ell} \right\rfloor \leq \left\lfloor \frac{U}{2^\ell} \right\rfloor \leq \frac{U}{2^\ell} = \frac{U}{2^{\max\{0, \lfloor \log(U/n) \rfloor\}}}$$

If there exists an integer k so that $\frac{U}{n} = 2^k$ then $\frac{U}{2^{\max\{0, \lfloor \log(U/n) \rfloor\}}} = n$. Otherwise, $\lfloor \log(U/n) \rfloor = \lceil \log(U/n) \rceil - 1$, and $\frac{U}{2^{\max\{0, \lfloor \log(U/n) \rfloor\}}} \leq 2n$.

Each Unary codeword requires a single 1-bit, and each 0-bit within the Unary codeword represents an increase by 2^ℓ. At most n 1s and $2n$ 0s are written in the Unary representation, that is, 3 bits per integer x_i. This concludes that the representation of Elias-Fano uses at most $2 + \lceil \log(\frac{U}{n}) \rceil$ bits per element. Elias [4] proves that the Elias-Fano representation is close to optimal as the information theoretical lower bound for

a monotonic sequence of n integers is

$$\left\lceil \log \binom{U+n}{n} \right\rceil \approx n \log \left(\frac{U+n}{n} \right).$$

Although, Elias-Fano's encoding is considered *quasi-succinct*, that is, close to the optimal representation, which is the information theoretical bound [16], there is still place for improvements by replacing the Unary code by C_γ, as the former code is costly for large integers. The Unary encoding uses $i+1$ bits to encode the integer i, $i \geq 0$, i.e., the codeword for the integer i is $\mathtt{0}^i\mathtt{1}$. The ith codeword for C_γ refers to the binary representation of $i+1$, denoted by $\mathcal{B}(i+1)$, as the value zero may also be encoded. The number of bits in $\mathcal{B}(i+1)$ is encoded using its Unary form, followed by $\mathcal{B}(i+1)$ after the preceding 1-bit has been removed. The first several codewords of Unary and C_γ are

	0	1	2	3	4	5	6	7	8
U	1	01	001	0001	00001	000001	0000001	00000001	000000001
C_γ	1	01 0	01 1	001 00	001 01	001 10	001 11	0001 000	0001 001

where blanks are inserted between the unary and the binary parts for readability. Only for the codewords corresponding to values 1 and 3 are Elias' C_γ codewords longer than those of the Unary code; for all other values, C_γ is preferable to the Unary code. We therefore propose a different variant of the Elias-Fano encoding, which is especially useful for non-uniform monotonic sequences having large standard deviation.

The sequence $\mathcal{X}$ is still represented by two bit vectors $\mathcal{L}(i)$ and $\mathcal{U}(i)$ for storing, respectively, the i lower bits and the differences between successive values of the remaining upper bits of each x_i, but this time we shall not fix the number of bits i in advance and rather let it vary from 0 to $\ell = \max\{0, \lfloor \log(\frac{U}{n}) \rfloor\}$. Using the example of Table 1, the line headed $\mathcal{U}(3)$-C_γ refers to the case of $i = 3$ and presents the corresponding $\mathcal{U}(3)$ for C_γ. The lower bits vector $\mathcal{L}(3)$ remains the same, for a total of 28 bits, instead of 26 used by the original Elias-Fano. However, the representation for $i = 2$ uses only 25 bits, shown by the bottom block of Table 1. The lower bits vector $\mathcal{L}(2)$ uses 10 bits, and $\mathcal{U}(2)$-C_γ uses additional 15 bits, less than the 26 bits used by Elias-Fano.

ℓ	0	1	2	3	4	5	6
Elias-Fano-C_γ	35	34	31	36	37	38	41

Table 2. Elias-Fano-C_γ for the sequence $2, 3, 10, 16, 520$ where the original Elias-Fano uses 43 bits

The introduction of the Elias-Fano-C_γ variant was, however, not suggested for the savings of merely a single bit, and is rather suitable for sequences with larger standard deviation. Consider the same example in which the last element has been changed to 520. The standard deviation grows from 18.41 for the first one, to 204.96 for this new example. Elias-Fano-Unary requires 43 bits for the updated sequence, while Elias-Fano-C_γ only needs 31 bits, which is attained for $\ell = 2$. Table 2 gives the total number of bits required for encoding the sequence 2, 3, 10, 16, 520 by the new version, as a function of ℓ. It is interesting to see that the storage for all values of ℓ needs less space than the original Elias-Fano coding.

We thus see that there is an advantage for using C_γ for sequences with high variability. Obviously, for general data, the logarithmic encoding of C_γ will be preferable to the linear encoding of Unary, but for very uniform data, the differences encoded by $\mathcal{U}$ in the Elias-Fano scheme will tend to consist mainly of very small integers, for which the Unary variant is not so bad. In order to improve also the compression of more homogeneous integer sets, we apply an idea used repeatedly in other data compression applications, namely that of using a reversible transformation of the original input to produce an equivalent sequence that is more compressible. This has been used by applying the Burrows-Wheeler transform (BWT) for the compression of textual and other data [3], or the discrete cosine transform in lossy image compression by JPEG [15]. In our case, we aim at causing a set of integers to be less homogeneous. It turns out that the existence of an extreme element in a sequence is typical for the output of the Haar transform [11], suggested as a preprocessing stage in the next section.

3 The Haar Transform

The Haar wavelet transform, is a simple discrete transform, used in practical encoding applications such as the compression of digitized sound and images. Here it is applied for lossless compression of integer sequences. The Haar transform uses the basic scale function $\phi(t)$, and the basic wavelet function $\psi(t)$ defined as follows.

$$\phi(t) = \begin{cases} 1, & 0 \le t < 1 \\ 0, & otherwise. \end{cases} \qquad \psi(t) = \begin{cases} 1, & 0 \le t < 0.5 \\ -1, & 0.5 \le t < 1. \end{cases}$$

A target function $f(t)$ is approximated by an infinite linear combination of $\phi(t-k)$ and $\psi(2^j t - k)$, where the parameter k assumes all possible, positive, negative and zero, integer values:

$$f(t) = \sum_{k=-\infty}^{\infty} c_k \phi(t-k) + \sum_{k=-\infty}^{\infty} \sum_{j=0}^{\infty} d_{j,k} \psi(2^j t - k),$$

where c_k and $d_{j,k}$ are constants. The function is transformed to a low resolution average $\phi(t)$ and the high resolution detail $\psi(t)$. In this research we are interested in a particular non-normalized Haar transform, and refer to its matrix representation.

The Haar transform is related to a matrix of order $2^k \times 2^k$ for $k \ge 1$. The non-normalized Haar matrix $\mathcal{H}_2$ of order 2×2 is $\mathcal{H}_2 = \frac{1}{2}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$. The Haar matrix $\mathcal{H}_{2^k}$ of order $2^k \times 2^k$ is defined recursively by $\mathcal{H}_{2^k} = \begin{pmatrix} \mathcal{H}_{2^{k-1}} \otimes (1,1) \\ I_{2^{k-1}} \otimes (1,-1) \end{pmatrix}$, where $\otimes$ is the *Kronecker product* defined for an $n \times m$ matrix A and a $t \times r$ matrix B as the $nt \times mr$ matrix $A \otimes B$ obtained by:

$$\text{if} \quad A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{pmatrix} \quad \text{then} \quad A \otimes B = \begin{pmatrix} a_{1,1}B & \cdots & a_{1,n}B \\ \vdots & \ddots & \vdots \\ a_{m,1}B & \cdots & a_{m,n}B \end{pmatrix}.$$

Given a sequence of 2^k values $a_1, a_2, \ldots, a_{2^k}$, the Haar transform computes, for each pair of values a_{2i-1} and a_{2i}, $i = 1, \ldots, 2^{k-1}$, the quantities

$$avg(i) = \frac{a_{2i-1} + a_{2i}}{2} \qquad \text{and} \qquad \Delta(i) = \frac{a_{2i-1} - a_{2i}}{2}.$$

The resulting sequence is composed of the averages $avg(1), avg(2), \ldots, avg(2^{k-1})$, followed by the half-differences, $\Delta(1), \Delta(2), \ldots, \Delta(2^{k-1})$, and it is of the same length as the input sequence. The 2^{k-1} averages are recursively transformed into 2^{k-2} new averages followed by 2^{k-2} half-differences, and so on until only a single element remains. The produced single value followed by the 2^{j-1} half-differences obtained from all stages, $j = 2, \ldots k$, are concatenated to form the final transformed elements. Note that this single value, together with the sequences of half-differences, is sufficient to reconstruct the original sequence, so the Haar transform is reversible.

As an example consider the sequence $\mathcal{X} = \{1840, 1680, 1632, 1504, 1536, 1472, 1360, 1328\}$. The Haar transform applied on $\mathcal{X}$ is presented in Figure 1 resulting in the non-increasing sequence $\mathcal{H}(\mathcal{X}) = \{1544, 120, 96, 80, 80, 64, 32, 16\}$. The original series is given on the first line. The partition into pairs is depicted by curly braces, and their average is presented on the following line. The elements contributed to the resulting Haar vector, are shown in gray. The last line is the Haar transform outcome.

Algorithm 1: *Haar-C_γ*

HAAR-$C_\gamma(x_1, \ldots, x_{2^k})$

1 $(h_1, \ldots, h_{2^k}) \leftarrow$HAAR$(x_1, \ldots, x_{2^k})$
2 $U \leftarrow h_1$
3 $\ell \leftarrow \max\{0, \lfloor \log(\frac{U}{2^k}) \rfloor\}$
4 Encode $(h_{2^k}, \ldots, h_1)$ using ELIAS-FANO-$C_\gamma(i)$ for $0 \le i \le \ell$
choosing i that results with minimum number of bits

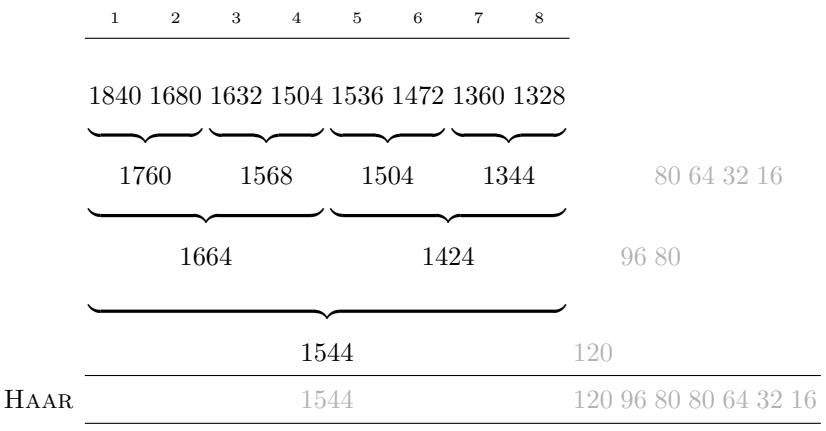


Figure 1. The Haar Transform for $\mathcal{X} = \{1840, 1680, 1632, 1504, 1536, 1472, 1360, 1328\}$

The resulting output of the Haar transform of Figure 1 is quite typical: a dominant first coefficient followed by others that are smaller by orders of magnitude, and most

importantly, with higher standard deviation than the original series. When the Haar transform is applied to an image, the averages of the disjoint successive pairs are commonly named the *coarse resolution* of the input image, while the differences of the pairs are called the *detail coefficients*. The Haar transform is effective for correlated pixels, as the coarse representation will resemble the original pixels, while the detail coefficients will be small. The small values tend to be more compressible than the original ones, and several compression techniques can be applied such as *Run-Length Encoding*, *Move-To-Front* and Huffman encoding for lossless compression, possibly adding quantization for lossy compression. For more details on the Haar transform we refer the reader to the book of Salomon [15].

In this research we suggest to apply Algorithm 1 in case the input integer series consists of similar numbers. Algorithm 1, which assumes that the input size is a power of 2, 2^k, starts by applying the Haar transform on the input sequence $(x_1, \ldots, x_{2^k})$ on line 1 and obtains the output sequence $(h_1, \ldots, h_{2^k})$ of the same length as a result. It then computes ℓ on line 3 as defined by Elias-Fano, and encodes the reverse sequence $(h_{2^k}, \ldots, h_1)$, to get an increasing sequence, with Elias-Fano-$C_\gamma(i)$, for i ranging from 0 to ℓ, choosing a value of i that results in the minimum number of bits.

Continuing our running example of Figure 1, we applied Algorithm 1 on the given sequence. The encoding of Haar-C_γ results in 62 bits, which was attained for $\ell = 3$. For comparison, Elias-Fano-Unary and Elias-Fano-C_γ on the *sorted* sequence of the original series gave 78 bits for $\ell = 7$ and 76 bits for $\ell = 6$, respectively. We also applied Elias-Fano-Unary on the resulting Haar vector, that, as noted above, is sorted for this example, which attained 76 bits for $\ell = 7$.

3.1 Encoding the Haar output using two blocks

In order to apply Algorithm 1, the Haar transform must result in a monotonic decreasing sequence, which is not necessarily the case. Algorithm 2 suggests the encoding by Elias-Fano-C_γ with only two different values for $\mathcal{U}(i)$-C_γ. That is, the sequence is partitioned into two buckets: the first containing only the first element, and all the others belonging to the second one. The corresponding values of $\mathcal{U}(i)$-C_γ are therefore all 0, so they can be omitted.

Interestingly, this encoding does not require a monotonic series as all coordinates, except the first, are written explicitly in the lower bits $\mathcal{L}$ array.

Algorithm 2: Bi-Haar-C_γ

Bi-Haar-$C_\gamma(x_1, \ldots, x_{2^k})$

1 $(h_1, h_2, \ldots, h_{2^k}) \leftarrow$Haar$(x_1, \ldots, x_{2^k})$
2 $m \leftarrow h_2$
3 $\ell \leftarrow \lfloor \log m \rfloor + 1$
4 Encode $(h_{2^k}, \ldots, h_1)$ using Elias-Fano-$C_\gamma(\ell)$, without encoding zeros in $\mathcal{U}$

Applying Algorithm 2 on our running example, $m = 120$, and the coefficients are encoded by $\lfloor \log m \rfloor + 1 = 7$ bits, for a total of 56 for $\mathcal{L}(7)$. The upper bits vector $\mathcal{U}(7)$-C_γ needs only to express the coarse coefficient, as the seven detail coefficients are with upper bit 0, which can be omitted for this method. In our example, the coefficient $\lfloor \frac{1544}{2^7} \rfloor = 12$, which is encoded by 13 bits in a Unary code, and by only 7 bits in C_γ, for a total of 69 and 63 bits, respectively.

3.2 Ensuring that the Haar transform results in integers

The Haar transform repeatedly computes the averages of number pairs in the input series, which may, obviously, produce non-integer numbers. Since Elias-Fano methods are restricted to integers, Algorithm 3 presents a variant of the Haar transform that makes sure the outcome consists only of integers. This is done by prepending a bit b_i, $1 \leq i \leq 2^k$ to each of the differences, indicating whether the corresponding average is exact or has been rounded. In fact, b_i is a parity bit as used in error correcting codes: in case the corresponding sum is even, b_i is set to 0, and if it is odd, $b_i = 1$. Concatenation is denoted by $\cdot$. The additional bits enables the reversibility of the Haar transform. The algorithm gets as input a sequence of $n = 2^k$ integers for some $k \geq 1$, and returns another sequence of n integers, the first being the (rounded) overall average, followed by $n - 1$ differences.

Algorithm 3: *Integer-Haar-C_γ*

INTEGER-HAAR-$C_\gamma(x_1, \ldots, x_{2^k})$

for $i \leftarrow 1$ **to** 2^{k-1} **do**
  $b_{2^{k-1}+i} \leftarrow (x_{2i} - x_{2i-1}) \bmod 2$
  $h_{2^{k-1}+i} \leftarrow \lfloor \frac{1}{2}(x_{2i} - x_{2i-1}) \rfloor$
  $z_i \leftarrow \lfloor \frac{1}{2}(x_{2i} + x_{2i-1}) \rfloor$
if $k = 1$ **then**
  return $(z_1, b_2 \cdot h_2)$
else
  $(y_1, \ldots, y_{2^{k-1}}) \leftarrow$HAAR$(z_1, \ldots, z_{2^{k-1}})$
  return $(y_1, \ldots, y_{2^{k-1}},\ b_{2^{k-1}+1} \cdot h_{2^{k-1}+1}, \ldots, b_{2^k} \cdot h_{2^k})$

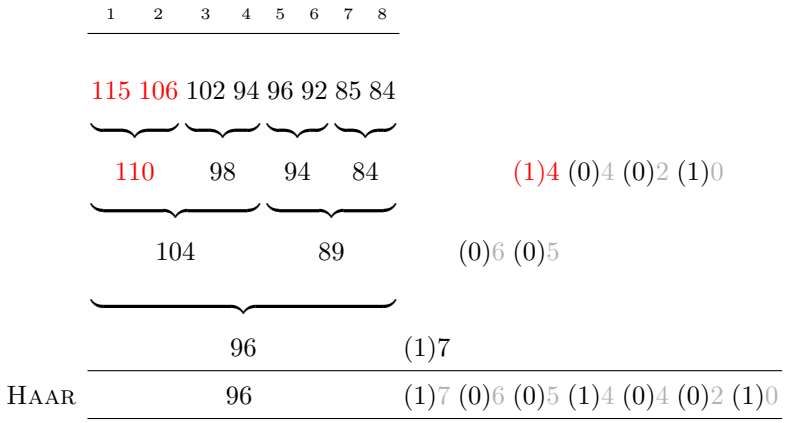


Figure 2. Haar Transform with two buckets

As example, consider the following sequence of integers $\mathcal{X} = 115, 106, 102, 94, 96, 92, 85, 84$, depicted in Figure 2. Each difference d on the right hand side is now preceded by a parity bit b in parentheses, which is set to 1 if and only if the corresponding average value a on the left hand side has been rounded, that is, the

sum of the two integers a' and a'' of the previous iteration, was odd, see the example in red in Figure 2. The reversibility means that we can recover a' and a'' from a, d and b. Indeed:

$$a' = a + d + b \qquad \text{and} \qquad a'' = a - d.$$

4 Experimental Results

In order to evaluate our proposed method, we considered randomly generated sequences of 256 elements. We defined U to be the largest element in the sequence and generated the Elias-Fano-C_γ encoding for varying values of ℓ from 1 to 30. All sequences presented a similar behavior, Figure 3 and Figure 4 depict the compression results of a typical representative.

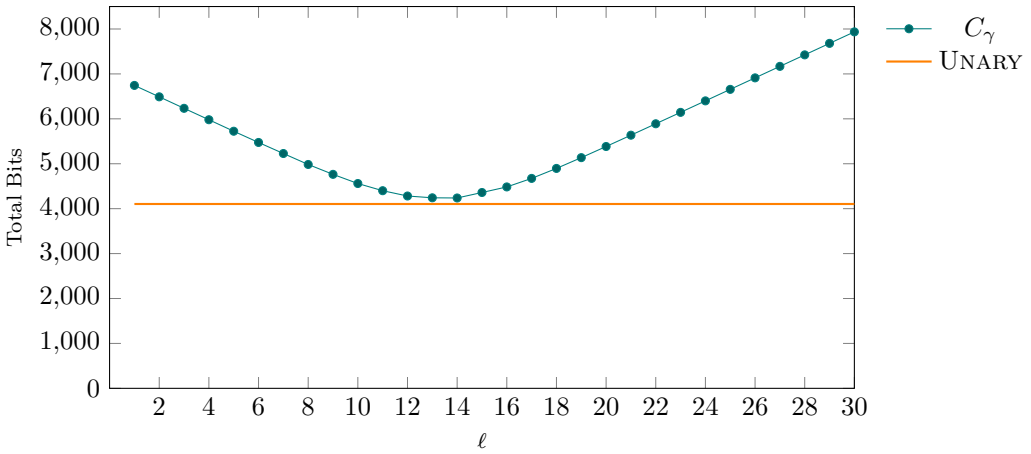


Figure 3. Elias-Fano-Unary vs. Elias-Fano-C_γ encoding for uniform random generated monotonic sequence of 256 elements

The general case is shown in Figure 3. Elias-Fano-Unary gives the best result, 4104 bits. The best encoding for Elias-Fano-C_γ is only slightly larger, 4238 bits, for $\ell = 14$, and it deteriorates for other values of ℓ. To get examples of more biased input sequences, the test was repeated again with randomly generated sequences, but to each of which one extreme element has been adjoined, thereby simulating the series handled by JPEG or after having applied the Haar transform. The corresponding graph of a typical example appears in Figure 4. Elias-Fano-Unary stores the input sequence using 6,144 bits with $\ell = 22$, while many Elias-Fano-C_γ values were lower, with a minimum achieved of 1,698 bits, for $\ell = 3$.

References

1. E. Benza, S. T. Klein, and D. Shapira: *Smaller compressed suffix arrays.* The Computer Journal, 2020.
2. A. Bookstein and S. T. Klein: *Compression of correlated bit-vectors.* Inf. Syst., 16(4) 1991, pp. 387–400.

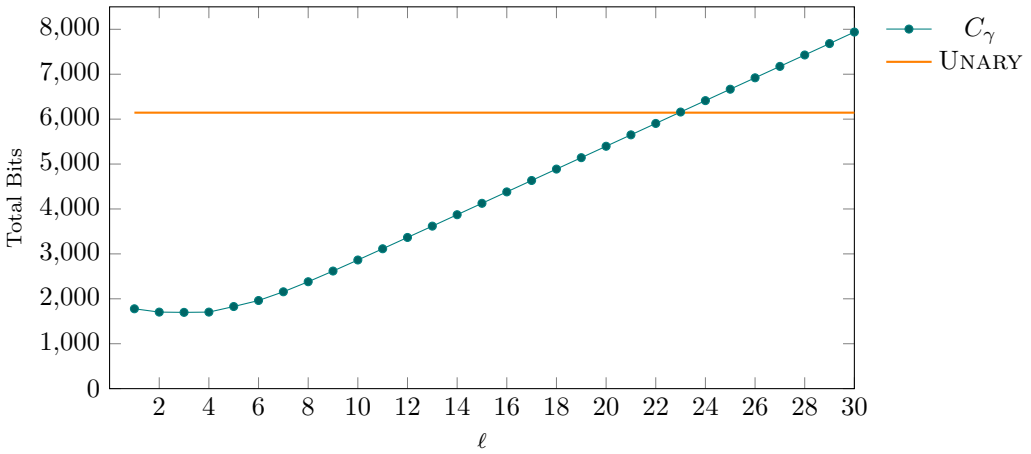


Figure 4. Elias-Fano-Unary vs. Elias-Fano-C_γ encoding for a random generated monotonic sequence of 256 elements, with an extreme element

3. M. Burrows and D. J. Wheeler: *A block sorting lossless data compression algorithm*, in SRC Technical Report **124**, Digital Equipment Corporation, Palo Alto, CA, 1994.
4. P. Elias: *Efficient storage and retrieval by content and address of static files.* J. ACM, 21(2) 1974, pp. 246–260.
5. P. Elias: *Universal codeword sets and representations of the integers.* IEEE Trans. Information Theory, 21(2) 1975, pp. 194–203.
6. R. Fano: *On the Number of Bits Required to Implement an Associative Memory*, Computation Structures Group Memo, MIT Project MAC Computer Structures Group, 1971.
7. A. S. Fraenkel and S. T. Klein: *Robust universal complete codes for transmission and compression.* Discrete Applied Mathematics, 64 1996, pp. 31–55.
8. S. Gog, A. Moffat, and M. Petri: *CSA++: fast pattern search for large alphabets*, in Proc. 19th Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, January 17-18, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2017, pp. 73–82.
9. S. W. Golomb: *Run-length encodings (corresp.).* IEEE Trans. Inf. Theory, 12(3) 1966, pp. 399–401.
10. 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.
11. A. Haar: *Zur Theorie der orthogonalen Funktionensysteme.* Mathematische Annalen, 69(3) 1910, pp. 331–371.
12. D. A. Huffman: *A method for the construction of minimum-redundancy codes.* Proceedings of the IRE, 40(9) 1952, pp. 1098–1101.
13. H. Huo, L. Chen, J. S. Vitter, and Y. Nekrich: *A practical implementation of compressed suffix arrays with applications to self-indexing*, in Proceeding of the Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26–28 March, IEEE Computer Society, Los Alamitos, CA, 2014, pp. 292–301.
14. G. Navarro: *Compact Data Structures - A Practical Approach*, Cambridge University Press, Cambridge UK, 2016.
15. D. Salomon, G. Motta, and D. Bryant: *Data Compression: The Complete Reference*, Molecular biology intelligence unit, Springer London, 2007.
16. S. Vigna: *Quasi-succinct indices*, in Sixth ACM International Conference on Web Search and Data Mining, WSDM 2013, Rome, Italy, February 4-8, 2013, S. Leonardi, A. Panconesi, P. Ferragina, and A. Gionis, eds., ACM, 2013, pp. 83–92.

Conversion of Finite Tree Automata to Regular Tree Expressions By State Elimination

Tomáš Pecka*, Jan Trávníček, and Jan Janoušek**

Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
{tomas.pecka,jan.travnicek,jan.janousek}@fit.cvut.cz

Abstract. Regular tree languages can be accepted and described by finite tree automata and regular tree expressions, respectively. We describe a new algorithm that converts a finite tree automaton to an equivalent regular tree expression. Our algorithm is analogous to the well-known state elimination method of the conversion of a string finite automaton to an equivalent string regular expression. We define a generalised finite tree automaton, the transitions of which read the sets of trees described by regular tree expressions. Our algorithm eliminates states of the generalised finite tree automaton, which is analogous to the elimination of states in converting the string finite automaton.

Keywords: regular tree languages, finite tree automata, regular tree expressions, state elimination method

1 Introduction

The theory of formal tree languages is an important part of computer science and has been extensively studied and developed since 1960s [5,6]. Trees are natural data structures for storing hierarchical data. Their applications range from areas such as natural language processing, interpretation of nonprocedural languages and code generation to processing markup languages such as XML.

Regular expressions are well-studied structures representing regular (string) languages in finite space [8,2]. The concept of expressions can be extended to regular tree languages as well. Regular tree expressions (RTEs) denote regular tree languages [5].

Standard computation models for problems on trees are various kinds of tree automata. An finite tree automaton (FTA) is an acceptor for the class of regular tree languages. The Kleene's theorem for tree languages states that the class of regular tree languages is equal to the class of languages that can be described by the RTEs [5]. Both formalisms are therefore equally powerful, but sometimes one of them is more convenient than the other one. For instance, language membership problem is easily solvable using the automaton. The expressions might be better in describing the language. Therefore it is suitable to find a way of converting one to another as we do in the area of strings.

The conversion of an RTE into an FTA was studied by several works. Algorithms presented in these papers are adaptations of well-known algorithms for the conversion of regular (string) expressions into finite (string) automata [9,10,3]. Note that there

* This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS20/208/OHK3/3T/18.

** The authors acknowledge the support of the OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics".

Tomáš Pecka, Jan Trávníček, Jan Janoušek: Conversion of Finite Tree Automata to Regular Tree Expressions By State Elimination, pp. 11–22.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9

also exist conversion algorithms for the problem of converting RTEs into pushdown (string) automata that accept trees in their linearised form [13,12].

The possibility of conversion in the other way, i.e. the conversion of an FTA to an RTE was introduced in the proof of the Kleene's theorem in [5]. The proof proposes a State elimination-like of conversion but it was not put into the algorithm and it was not discussed much. Recently, Guellouma and Cherroun studied regular tree equations [7] which they used to construct a regular tree equation system from an FTA. Solving this equation system yields an RTE describing the same language that was accepted by the original FTA. Unfortunately, the time complexity of the algorithm is not stated in the article.

In this paper, we build upon the idea of eliminating states from an FTA presented in the proof of Kleene's Theorem [5]. We present a practical algorithm for the problem of converting FTAs to RTEs that is inspired by the classical State elimination algorithm for finite (string) automata [8]. We define the notion of generalized finite tree automaton (GFTA) which differs from the FTA mainly in the transition function. Instead of the input alphabet symbols, the transitions now involve sets of trees described by the RTE. We use this model for the process of eliminating states. States of the GFTA are then eliminated one-by-one and the transitions of the automaton are modified in such way that the language the automaton accepts does not change. The transitions to the last remaining final state then define the equivalent RTE to the original automaton. Such approach can convert an FTA to an equivalent RTE in $O(|Q|^2 \cdot (|\Delta| + |Q_F|)$ time where Q and Q_F are the sets of all states and final states of the FTA, respectively, and Δ is the set of transitions of the FTA. Furthermore, the implementation of the algorithm is really simple and straightforward.

The following parts of this paper are organised as follows: Section 2 recalls basic definitions and notations. The new conversion algorithm yielding the RTE is presented in Section 3. Finally, the achieved results and ideas for future work are presented in the concluding section.

2 Background

A *ranked alphabet* Σ is a finite nonempty set of symbols, each of which is assigned with non-negative integer *arity* denoted by arity(a). The set Σ_n denotes the set of symbols from Σ with arity n. Elements of arity $0, 1, 2, \ldots, n$ are called nullary (also constants), unary, binary, ..., n-ary symbols, respectively. We assume that Σ contains at least one constant. We use numbers at the end of symbols for a short declaration of arity. For instance, $a2$ is a short declaration of a binary symbol a.

A labelled, ordered and ranked tree over a ranked alphabet Σ is defined on the concepts from graph theory [2]. A *directed ordered graph* G is a pair (N, R) where N is a set of nodes and R is a list of ordered pairs of edges. Elements of R are in the form $((f, g_1), (f, g_2), \ldots, (f, g_n))$, where $f, g_1, g_2, \ldots, g_n \in N$, $n \geq 0$. Such element denotes n edges leaving f with the first edge entering node g_1, the second entering g_2, and so forth. A sequence of nodes $(f_0, f_1, \ldots, f_n)$, $n \geq 1$ is a *path* of length n from node f_0 to f_n if there is an edge from f_i to f_{i+1} for each $0 \leq i < n$. A *cycle* is a path where $f_0 = f_n$. An *in-degree* of a node is a number of incoming edges. An *out-degree* is a number of outgoing edges. A node with out-degree 0 is called a *leaf*.

An ordered *directed acyclic graph (DAG)* is an ordered directed graph with no cycle. A *rooted DAG* is a DAG with a special node $r \in N$ called the *root*. The in-degree of r is 0, in-degree of every other node is 1 and there is just one path from the

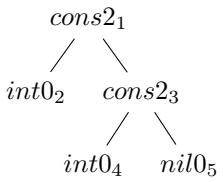


Figure 1. A directed, rooted, labelled, ranked, and ordered tree over $\Sigma = \{nil0, int0, cons2\}$.

root r to every $f \in N,\ f \neq r$. A *labelled ranked DAG* is a DAG where every node is labelled by a symbol $a \in \Sigma$ and the out-degree of a node $a \in \Sigma$ equals to arity(a). A *directed, ordered, rooted, labelled, and ranked tree* is rooted, labelled, and ranked DAG. All trees in this paper are considered to be directed, ordered, rooted, labelled, and ranked.

Due to simplicity we often represent trees in the well-known prefix notation (see Example 1).

Example 1. Let t from Figure 1 be a directed, rooted, labelled, ranked, and ordered tree with labels from ranked alphabet $\Sigma = \{nil0, int0, cons2\}$. Formally the tree is a graph $t = (\{cons2_1, int0_2, cons2_3, int0_4, nil0_5\}, \{(cons2_1, int0_2), (cons2_1, cons2_3), (cons2_3, int0_4), (cons2_3, nil0_5)\})$. The root of t is a $cons2_1$ node with an ordered pair of children $(int0_2, cons2_3)$. The prefix notation of t is $cons2(int0, cons2(int0, nil0))$. We omit the subscripted indexes and show the labels from Σ only.

A nondeterministic *bottom-up finite tree automaton (FTA)* over a ranked alphabet Σ is a $4-$tuple $\mathcal{A} = (Q, \Sigma, Q_F, \Delta)$, where Q is a finite set of states, $Q_F \subseteq Q$ is a set of final states, and Δ is a mapping $\Sigma_n \times Q^n \mapsto \mathcal{P}(Q)$ (where $\mathcal{P}$ denotes the powerset function), i.e., the transitions are in the form $f(q_1, q_2, \ldots, q_n) \to q$ where $f \in \Sigma_n$, $n \geq 0$, and $q, q_1, q_2, \ldots, q_n \in Q$. An FTA is a *deterministic FTA* if for every left hand side of the transition there is at most one target state.

The computation of the FTA starts at leaves and moves towards the root. Each subtree is mapped to a state. A *run* of an automaton is defined inductively: The leaves are mapped to states q by the transitions of the form $a \to q \in \Delta$, $a \in \Sigma_0$. If a root of a subtree is labelled with $f \in \Sigma_n$, $n \geq 1$, and its children are mapped to states $q_1, \ldots, q_n$, this subtree is mapped to q, where $f(q_1, q_2, \ldots, q_n) \to q \in \Delta$. *Language of a state* q (denoted by $L(q)$) is a set of subtrees mapped to state q. Obviously, an FTA *accepts* such trees that have their roots mapped to any final state, i.e., $L(\mathcal{A}) = \bigcup_{q \in Q_F} L(q)$.

The tree language $L(\mathcal{A})$ *recognised* by an FTA $\mathcal{A}$ is the set of trees accepted by the FTA $\mathcal{A}$. A tree language is *recognisable* if it is recognised by some FTA. It is recognisable if and only if it is a regular tree language (see [4,5] for the definition). Every nondeterministic FTA can be transformed to an equivalent deterministic FTA [5].

The transition function of FTA can be depicted using a diagram in a similar way as the diagram of a finite automaton. However, transitions of an FTA can have an arbitrary amount of source states. Therefore a *join node* of source states is added into the diagram. The order of source states is specified by a number on edges from states to join nodes.

Example 2. An example of an FTA is $\mathcal{A} = (\{I, L\}, \{int0, nil0, cons2\}, \{L\}, \Delta)$, where Δ consists of the following transitions: $int0 \to I$, $nil0 \to L$, and $cons2(I, L) \to L$. $\mathcal{A}$

accepts the language $L(\mathcal{A}) = \{nil, cons(int, nil), cons(int, cons(int, nil)), \ldots\}$. The automaton is depicted in Figure 2.

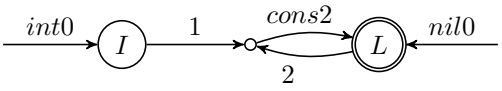


Figure 2. Visualisation of the FTA $\mathcal{A}$ from Example 2.

Regular tree expressions (RTEs) are defined (as in [5]) over two alphabets, $\mathcal{F}$ and $\mathcal{K}$. $\mathcal{F}$ is a ranked alphabet of symbols. $\mathcal{K}$ is a set of constants (special symbols with arity 0), $\mathcal{K} = \{\Box_1, \Box_2, \ldots, \Box_n\}$, $n \geq 0$, $\mathcal{F} \cap \mathcal{K} = \emptyset$. This alphabet is used to indicate the position where substitution operations take place.

Firstly, the substitution, i.e. replacing occurrences of $\Box_i$ by trees from a tree language L_j, is defined. Let $\mathcal{K} = \{\Box_1, \ldots, \Box_n\}$ and t be a tree over $\mathcal{F} \cup \mathcal{K}$, and $L_1, \ldots, L_n$ be tree languages. Then the *tree substitution* of $\Box_1, \ldots, \Box_n$ by $L_1, \ldots, L_n$ in t denoted by $t\{\Box_1 \leftarrow L_1, \ldots, \Box_n \leftarrow L_n\}$ is the tree language defined by the following identities:

- $\Box_i\{\Box_1 \leftarrow L_1, \ldots, \Box_n \leftarrow L_n\} = L_i$, for $i = 1, \ldots, n$,
- $a\{\Box_1 \leftarrow L_1, \ldots, \Box_n \leftarrow L_n\} = \{a\}, \forall a \in \mathcal{F}_0 \cup \mathcal{K}$ and $a \neq \Box_1, \ldots, a \neq \Box_n$,
- $f(s_1, \ldots, s_n)\{\Box_1 \leftarrow L_1, \ldots, \Box_n \leftarrow L_n\} = \{f(t_1, \ldots, t_n) \mid t_i \in s_i\{\Box_1 \leftarrow L_1, \ldots, \Box_n \leftarrow L_n\}\}$.

The tree substitution can be generalized to languages: $L\{\Box_1 \leftarrow L_1, \ldots, \Box_n \leftarrow L_n\} = \bigcup_{t \in L} t\{\Box_1 \leftarrow L_1, \ldots, \Box_n \leftarrow L_n\}$. The operation *alternation* of L_1 and L_2 is denoted by $L_1 + L_2$. It results in a set of trees obtained from the union of regular tree languages L_1 and L_2, i.e. $L_1 \cup L_2$. The operation *concatenation* of L_2 to L_1 through $\Box$, denoted by $\cdot\Box\,(L_1, L_2)$, is the set of trees obtained by substituting the occurrence of $\Box$ in trees of L_1 by trees of L_2, i.e. $\bigcup_{t \in L_1} t\{\Box \leftarrow L_2\}$. Given a tree language L over $\mathcal{F} \cup \mathcal{K}$ and $\Box \in \mathcal{K}$, the sequence $L^{n,\Box}$ is defined by the equalities $L^{0,\Box} = \{\Box\}$ and $L^{n+1,\Box} = \cdot\Box\,(L, L^{n,\Box})$. The operation *closure* is defined as $L^{*,\Box} = \bigcup_{n \geq 0} L^{n,\Box}$.

Finally, an RTE over alphabets $\mathcal{F}$ and $\mathcal{K}$ is defined inductively:

- the empty set ($\emptyset$) and a constant ($a \in \mathcal{F}_0 \cup \mathcal{K}$) are RTEs,
- if $E_1, E_2, \ldots, E_n$ are RTEs, $n > 0$, $f \in F_n$ and $\Box \in \mathcal{K}$, then: $E_1 + E_2$, $\cdot\Box\,(E_1, E_2)$, ${E_1}^{*,\Box}$, and $f(E_1, \ldots, E_n)$ are RTEs.

RTE E represents a language denoted by $L(E)$ and defined by the following equalities:

- $L(\emptyset) = \emptyset$,
- $L(a) = \{a\}$ for $a \in F_0 \cup K$,
- $L(f(E_1, \ldots, E_n)) = \{f(s_1, \ldots, s_n) \mid s_1 \in L(E_1), s_2 \in L(E_2), \ldots, s_n \in L(E_n)\}$,
- $L(E_1 + E_2) = L(E_1) \cup L(E_2)$,
- $L(\cdot\Box\,(E_1, E_2)) = L(E_1)\{\Box \leftarrow L(E_2)\}$,
- $L(E^{*,\Box}) = L(E)^{*,\Box}$.

We define the set $RTE(\mathcal{F}, \mathcal{K})$ to be a set of all RTEs over $\mathcal{F}$ and $\mathcal{K}$ alphabets. A regular tree language is recognisable if and only if it can be denoted by an RTE [5]. For the sake of simplicity we allow the alternation to act as an n-ary operator ($n \geq 2$), e.g., the RTE $(((E_1 + E_2) + E_3) + \cdots + E_n)$ can be written as $E_1 + E_2 + E_3 + \cdots + E_n$.

Example 3. Let $\mathcal{F} = \{nil0, int0, cons2\}$ and let $\mathcal{K} = \{\Box_1, \Box_2\}$. Then the RTE E from Figure3 denotes the language of lists of integers in LISP. $L(E) = \{nil0,$ $cons2(int0, nil0),$ $cons2(int0, cons2(int0, nil0)), \ldots\}$.

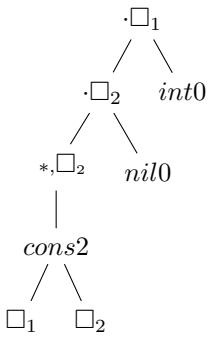


Figure 3. An RTE from Example 3 denoting the language of integer lists in LISP.

3 State elimination algorithm

The proof of Kleene's Theorem in Comon et al. [5] hinted that the conversion of FTAs to RTEs can be done by elimination of states. However, the algorithm itself is not presented in the book.

The following approach is inspired by the well-known state elimination algorithm for finite automata [8]. We eliminate states one by one and all paths that went through the eliminated state will be replaced by new transitions. Also, the transitions will now involve sets of trees described by an RTE (in the places where the classical string version would involve regular expressions) rather than individual symbols of the alphabet.

3.1 Generalized finite tree automaton

In order to represent a tree automaton whose transitions involve trees rather than symbols of the input alphabet we define an extension of the FTA called generalized finite tree automaton (GFTA). The GFTA differs from FTA mainly in the transition function which is defined over RTEs rather than over the individual symbols of a ranked alphabet.

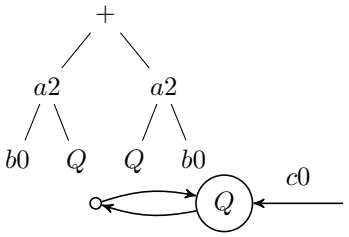
Definition 4. *Let $A' = (Q, \Sigma, Q_F, \Gamma)$ be a generalized finite tree automaton (GFTA). The meaning of Q, Σ and Q_F sets is the same as in an FTA and Γ is a mapping $RTE(\Sigma, Q) \times \mathcal{P}(Q) \mapsto Q$. The Q_F set is a singleton.*

Transition function Γ of a GFTA is in the form $E\{q_1, \ldots, q_n\} \to q$ where E is an $RTE(\Sigma, Q)$ and $q, q_1, \ldots, q_n \in Q$. The substitution symbols of RTEs act as references to the languages of the corresponding states and the constant alphabet of RTEs is therefore equal to the set of states, i.e., Q.

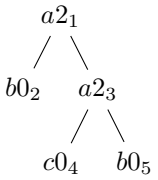
The order of transition's source states is no longer important as it is defined in the RTE because the children in RTEs are ordered. For this reason, it is no longer necessary to maintain the vector of source states of a transition ordered, and it can be converted to a set.

The run of GFTA is defined similarly to the run of FTA. A subtree t is labelled with the state q only if there exists a transition $E\{q_1 \ldots q_n\} \to q$ and $t \in L(E)$. Recall that the symbols from Q set in the RTE act as the references to the states of the automaton. A tree t is accepted by GFTA if t is labelled with the final state of the automaton. Language of a GFTA is the set of trees accepted by the automaton.

We use almost the same rules for depicting GFTAs as for FTAs. Only the edges leading to *join nodes* are no longer labelled with their position because they are no longer ordered.



(a) Example GFTA.



(b) Example tree.

Figure 4. Example GFTA and a sample tree it accepts.

Example 5. Consider the simple GFTA depicted in Figure 4a and the input tree t from Figure 4b. Subtree $c0_4$ of t is trivially labelled with state Q. Subtrees $a2_3$ and $a2_1$ are also labelled with Q state because both subtrees correspond to the RTE leading to the state Q. Note that the Q symbol in the RTE corresponds to the subtree labelled with state Q. Subtrees $b0_2$ and $b0_5$ are not labelled with a state.

Lemma 6. *An FTA $A = (Q, \Sigma, Q_F, \Delta)$ can be converted to an equivalent GFTA $A' = (Q \cup q_f, \Sigma, \{q_f\}, \Gamma)$, $q_f \notin Q$.*

Proof. Every transition $f_n(q_1, \ldots, q_n) \to q \in \Delta$ is transformed to $E\{q_1, \ldots, q_n\} \to q \in \Gamma$ where E is an RTE $f_n(q_1, \ldots, q_n)$. In order to have only one final state we also add a new state q_f and we create a transition $E\{q\} \to q_f$ where $E = q$ from every old final state $q \in Q_F$ to q_f. This is similar to adding ε-transitions in the string variant. It is easy to see that the languages accepted by A and A' are equal. □

The transformation of an FTA to a GFTA is obvious from the proof of Lemma 6, but we still formalise it in Algorithm 1. We also formulate Lemma 8, which states that every single-state GFTA can be converted to an RTE.

Algorithm 1: FTA to GFTA

input : FTA $A = (Q, \Sigma, Q_F, \Delta)$
output : GFTA $A' = (Q', \Sigma, Q'_F, \Gamma)$ corresponding to A
function *ExtendFTA*$(A = (Q, \Sigma, Q_F, \Delta))$**:**
  $Q' = Q \cup \{q_f\}$ $(q_f \notin Q)$ `// new final state`
  $Q'_F = \{q_f\}$
  $\Gamma = \{E\{q_1, \ldots, q_n\} \to q \text{ where } E = f_n(q_1, \ldots, q_n) \mid \forall f_n(q_1, \ldots, q_n) \to q \in \Delta\}$
  $\Gamma = \Gamma \cup \{E'\{q\} \to q_f \text{ where } E' = q \mid \forall q \in Q_F\}$
  return $A' = (Q', \Sigma, Q'_F, \Gamma)$

Example 7. Figure 5 shows the tree automaton from Example 2 converted to GFTA by Algorithm 1.

Lemma 8. *Let $A = (Q, \Sigma, Q_F, \Gamma)$ be a GFTA with only 1 state ($|Q| = 1$) that is also the final state ($Q = Q_F$) and transitions in the form $E_i \to q$. One can generate an RTE E such that $L(E) = L(A)$.*

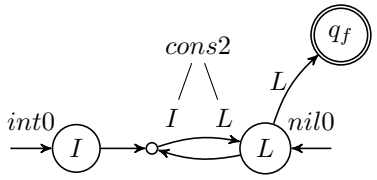


Figure 5. Visualisation of the GFTA created from the FTA from Example 2.

Proof. Multiple transitions of the form $E'_i \to q$ can be transformed into a single transition in the form $E \to q$ where $E = E'_1 + \cdots + E'_n$ and E'_i corresponds to the RTE of the i-th original transition.

Now it is clear that any tree accepted by the automaton must also be in the language denoted by the RTE corresponding to the only transition of the automaton and vice versa. Therefore the language denoted by the RTE E is equivalent to the language of the GFTA. □

3.2 Elimination of a single state

Before we state the full algorithm, we must define the process of eliminating a single non-final state from a GFTA. The elimination of a single state q, $q \in Q \setminus Q_F$ from a GFTA modifies the GFTA in such a way that the state q is no longer present. Therefore the transitions involving the state q are removed as well. However, the language of the automaton must remain unchanged. To compensate, some new transitions between the remaining states are added.

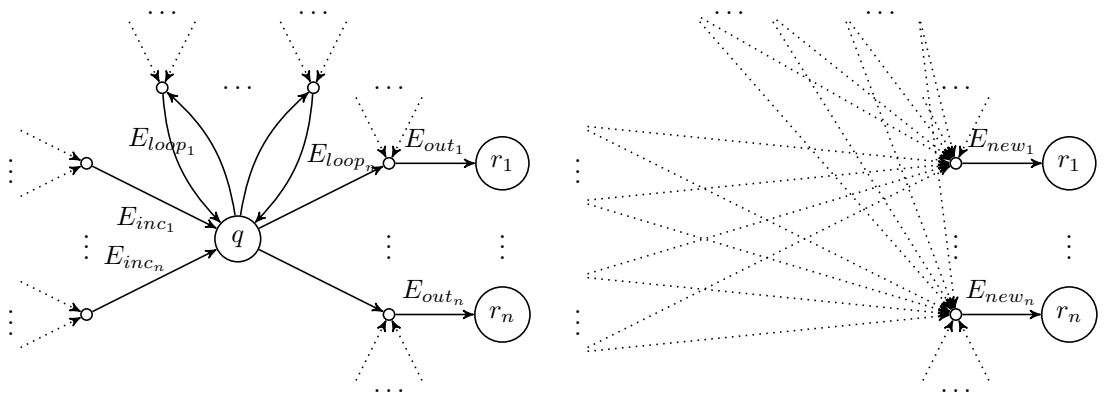
With respect to the state q we classify the transitions of the automaton into the following four groups: A transition is *incoming* if q is only a target state but not a source. If q is among the source states of a transition and also a target state, then the transition is classified as *looping*. If q is only a source state but not a target, it is called an *outgoing* transition. If q has no part in the transition, it is classified as an *other* transition. It is obvious that *other* transitions are left intact when q is eliminated. Definition 9 formalises the classification.

Definition 9. *Function* trtype *classifies the transition of a GFTA w.r.t. the state* $q \in Q$.

$$\mathrm{trtype}(E\{p_1, \ldots, p_n\} \to p, q) = \begin{cases} \text{incoming} & \text{if } q \notin \{p_1, \ldots, p_n\} \wedge q = p \\ \text{outgoing} & \text{if } q \in \{p_1, \ldots, p_n\} \wedge q \neq p \\ \text{looping} & \text{if } q \in \{p_1, \ldots, p_n\} \wedge q = p \\ \text{other} & \text{if } q \notin \{p_1, \ldots, p_n\} \wedge q \neq p \end{cases}$$

Example 10. Let $A = (Q, \Sigma, Q_F, \Gamma)$ be the GFTA from Figure 5. According to the Definition 9 the transitions from Γ with respect to state L are classified as follows: $\mathrm{trtype}(int0 \to I, L) = \text{other}$, $\mathrm{trtype}(nil0 \to L, L) = \text{incoming}$, $\mathrm{trtype}(cons2\{I, L\} \to L, L) = \text{looping}$, $\mathrm{trtype}(L \to q_f, L) = \text{looping}$.

Now consider the fragment of GFTA visualised in Figure 6a. Arbitrary non-final state q of the GFTA may have incoming transitions (such transitions are labelled in the picture by the RTEs E_{inc_i}), looping transitions (E_{loop_i}) and outgoing transitions (E_{out_i}). The other transitions are obviously left intact by the process of eliminating q



(a) Before the elimination of state q.

(b) After the elimination of state q by Lemma 12.

Figure 6. Fragment of a GFTA before and after the elimination of state q.

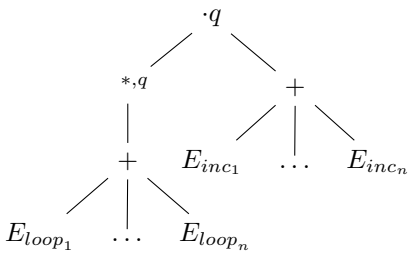


Figure 7. RTE equivalent to the language of a state of GFTA.

because q is not involved in those transitions. Let us state the following lemmas that will define the elimination process.

Lemma 11. *The concatenation operation in RTEs is associative, i.e., for RTEs x, y, z the following holds: $\cdot\Box\,(\cdot\Box\,(x, y), z) = \cdot\Box\,(x, \cdot\Box\,(y, z))$.*

Proof. No mater the order of application of the concatenation operation in either RTEs $\cdot\Box\,(\cdot\Box\,(x, y), z)$ and $\cdot\Box\,(x, \cdot\Box\,(y, z))$, the occurrences of $\Box$ in the RTE x are the place of substitution of a language given by RTE y and the occurrences of $\Box$ in the RTE y are the place of substitution of a language given by RTE z. □

Lemma 12. *Let $A = (Q, \Sigma, Q_F, \Gamma)$ be a GFTA and $q \in Q \setminus Q_F$. Let $E_{loop_1}, \ldots, E_{loop_n}$ be RTEs collected from looping transitions w.r.t. the state q, let $E_{inc_1}, \ldots, E_{inc_n}$ be RTEs from incoming transitions w.r.t. the state q and let $E_{out_1}, \ldots, E_{out_n}$ be RTEs collected from outgoing transitions w.r.t. the state q. Then the language of state q can be represented as an RTE $E_q = \cdot q\,((E_{loop_1} + \cdots + E_{loop_n})^{*,q}, (E_{inc_1} + \cdots + E_{inc_n}))$ (for clarity, the RTE fragment is visualised in Figure 7). Then the references to q in E_{out_i} can be replaced (using the concatenation operation) with an RTE denoting the language of state q. The E_{out_i} transition then becomes $E_{new_i} = \cdot q\,(E_{out_i}, E_q)$. Note that empty alternation equals to $\emptyset$.*

Proof. The proof is essentially the same as the proof of Kleene's Theorem in [5, Prop. 2.2.7]. We only use different RTE fragment for the E_{new_i} because we find it more intuitive. The fragment proposed in [5], i.e., $E_{new_i} = \cdot q\,(\cdot q\,(E_{out_i}, (E_{loop_1} + \cdots + E_{loop_n})^{*,q}), (E_{inc_1} + \cdots + E_{inc_n}))$, is equivalent as follows from Lemma 11. □

Lemma 13. *After eliminating state $q \in Q$, all occurrences of symbol q in the automaton's transition function are bounded by some concatenations over q.*

Proof. References to state q can appear either at outgoing and looping transitions of state q or at incoming and looping transitions of a state $p \in Q, p \neq q$ of the automaton. Firstly, by eliminating the state q the references at outgoing and looping transitions are surely bounded under concatenation nodes (by Lemma 12). Now for the state p. If q appears at a incoming or a looping transition of the state p then (by Definition 9) the transition is also an outgoing transition of state q. Therefore the reference is properly bounded by eliminating state q. □

The state elimination algorithm is intentionally designed not to replace the references inside the RTE but rather to utilize the concatenation operation. Using this approach, there is no need to traverse the RTEs to replace all the occurrences. Furthermore, when multiple references are present, the replacement is not copied to multiple places. This approach is formalised in Algorithm 2. Function *Alternate* is used in Algorithm 2. For a set of transitions of labelled with RTEs E_i the function returns an alternation of those, i.e., an RTE $E_1 + \cdots + E_n$.

Algorithm 2: Elimination of a single state in a GFTA

input : GFTA $A = (Q, \Sigma, Q_F, \Gamma)$, $q \in Q$
output : GFTA $A' = (Q \setminus \{q\}, \Sigma, Q_F, \Gamma')$, i.e., A without state q and $L(A) = L(A')$
function *EliminateState*$(A = (Q, \Sigma, Q_F, \Gamma),\ q)$**:**
  $incoming, looping, sources, \Gamma' = \emptyset, \emptyset, \emptyset, \emptyset$
  foreach $E\{q_1, \ldots, q_n\} \to r \in \Gamma$ **do**
    if trtype$(E\{q_1, \ldots, q_n\} \to r, q) =$ incoming **then**
      $incoming = incoming \cup \{E\{q_1, \ldots, q_n\} \to r\}$
      $sources = sources \cup \{q_1, \ldots, q_n\}$
    else if trtype$(E\{q_1, \ldots, q_n\} \to r, q) =$ looping **then**
      $looping = looping \cup \{E\{q_1, \ldots, q_n\} \to r\}$
      $sources = sources \cup \{q_1, \ldots, q_n\}$
    else if trtype$(E\{q_1, \ldots, q_n\} \to r, q) =$ other **then**
      $\Gamma' = \Gamma' \cup \{E\{q_1, \ldots, q_n\} \to r\}$ `// not involved`
  foreach $E\{q_1, \ldots, q_n\} \to r \in$ outgoing **do**
    $E_{new} = \cdot q\ (E, (\cdot q\ (Alternate(looping)^{*,q}, Alternate(incoming))))$ `// Lemma 12`
    $\Gamma' = \Gamma' \cup \{E_{new}\{sources \setminus \{q\} \cup \{q_1, \ldots, q_n\}\} \to r\}$
  return $A' = (Q \setminus \{q\}, \Sigma, Q_F, \Gamma')$

Lemma 14. *Applying Algorithm 2 to a GFTA $A = (Q, \Sigma, Q_F, \Gamma)$ does not increase the cardinality of Γ.*

Proof. The claim follows directly from Algorithm 2. For every other and outgoing transition one transition is added to Γ'. For every incoming and looping transition no new transitions are added. Therefore it always holds that $|\Gamma'| \leq |\Gamma|$. □

Lemma 15. *Algorithm 2 runs in $O(|Q| \cdot |\Gamma|)$ time for input GFTA $A = (Q, \Sigma, Q_F, \Gamma)$.*

Proof. Both for loops obviously iterates over at most $|\Gamma|$ elements and also require some work for merging two sets of size at most Q. Therefore, the upper bound of running time is $O(|Q| \cdot |\Gamma|)$. □

3.3 State elimination algorithm

The previous subsection stated an algorithm for the elimination of a single non-final state from a GFTA. This process can be repeated (in arbitrary order of states) until we obtain a single-state automaton with such transitions that allow us to directly apply Lemma 8. Algorithm 3 formalises this simple process.

Algorithm 3: State elimination of a GFTA

```
   input  : FTA A = (Q, Σ, Q_F, Δ)
   output : RTE E such that L(A) = L(E)
1  function StateElimination(A = (Q, Σ, Q_F, Δ)):
2  |  GFTA A' = (Q', Σ', {q_f}, Γ) = ExtendFTA(A)          // Algorithm 1
3  |  foreach q ∈ Q' \ {q_f} do
4  |  |  A' = EliminateState(A', q)                       // Algorithm 2
5  |  return Alternate({E_i | ∀E_i{q_1, ..., q_n} → q_f} ∈ Γ)   // remaining transitions
```

Theorem 16. *Algorithm 3 converts an FTA $A = (Q, \Sigma, Q_F, \Delta)$ to an RTE E such that $L(A) = L(E)$.*

Proof. The algorithm creates a single-state GFTA. Therefore the claim immediately follows from Lemmas 8, 12 and 13.

Theorem 17. *The total running time of Algorithm 3 is $O(|Q|^2 \cdot (|\Delta| + |Q_F|))$ time.*

Proof. The running time consists of converting the original FTA to GFTA and $|Q|$ invocations of Algorithm 2. The largest value for $|\Gamma|$ parameter of Algorithm 2 is in the first iteration where $|\Gamma| = |\Delta| + |Q_F|$ (Lemma 6). After the elimination of a single state, the number of transitions can only decrease or remain the same (Lemmas 12 and 14). Using time complexity of Algorithm 2 stated in Lemma 15, the total upper bound on the complexity of Algorithm 3 is $O(|Q| \cdot |Q| \cdot (|\Delta| + |Q_F|))$. □

Example 18. Let A be the GFTA from Figure 2. The trace run of the algorithm is shown in Figure 8. The states of A are eliminated in lexicographical order, i.e., I, L.

4 Conclusion

We presented a simple full algorithm for the construction of a regular tree expression (RTE) equivalent to given finite tree automaton (FTA) by eliminating states. Both the idea and implementation are also very similar to the original State elimination for finite (string) automata and regular expressions [8]. This construction was originally hinted in [5] to prove the Kleene Theorem, i.e., the equivalence between languages of RTEs and FTAs. We showed that the idea of eliminating states one by one from an FTA forms an easy and intuitive algorithm that can be easily implemented.

The presented algorithm runs in $O(|Q|^2 \cdot (|\Delta| + |Q_F|))$ time, i.e., it is proportional to the number of states and the size of the transition function of the converted automaton. Also, different order of elimination may create different RTE but all of them denote the same language. However, this also holds for the original string algorithm [8,11].

Future work may focus on the Algorithm 2. Better data structures may help in improving the time complexity upper bound. Another interesting problem is finding

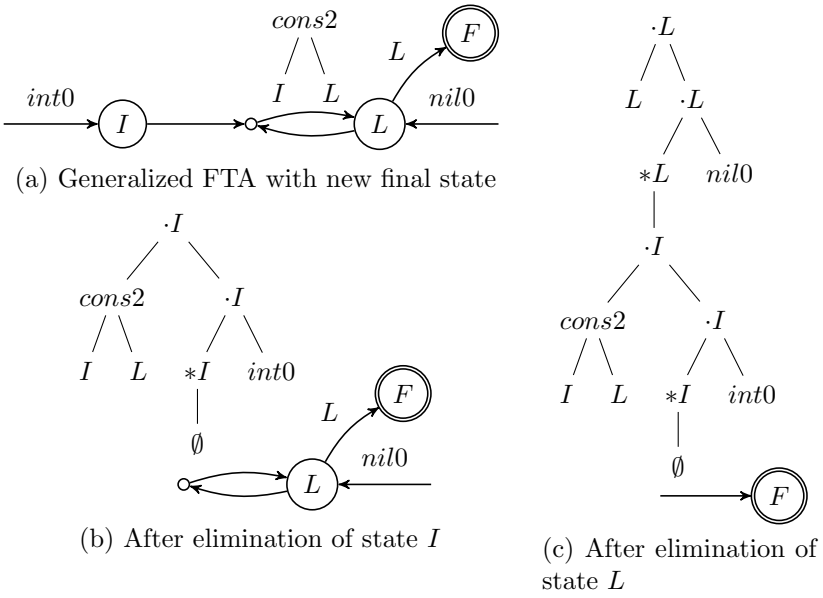


Figure 8. Example run of the algorithm on the FTA from Figure 2.

the best elimination order. Different orderings give different resulting RTEs and some of them are smaller than others. Similar experimental research was done in the string elimination method [11].

You can find the `C++` implementation of the presented algorithm in the latest versions of Algorithms Library Toolkit project [1]. We also tested the implementation by converting various random FTAs to RTEs using this algorithm and then back using the algorithm from [12] adapted to FTAs.

References

1. *Algorithms Library Toolkit*: `https://alt.fit.cvut.cz`.
2. A. V. Aho and J. D. Ullman: *The theory of parsing, translation, and compiling. 1: Parsing*, Prentice-Hall, 1972.
3. A. Belabbaci, H. Cherroun, L. Cleophas, and D. Ziadi: *Tree pattern matching from regular tree expressions*. Kybernetika, 54(2) 2018, pp. 221–242.
4. L. Cleophas: *Tree Algorithms: Two Taxonomies and a Toolkit*, PhD thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, Apr. 2008.
5. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi: *Tree automata techniques and applications*, 2007, Release October 2007.
6. F. Gécseg and M. Steinby: *Tree Languages*, vol. 3 of Handbook of Formal Languages, Springer, 1997, pp. 1–68.
7. Y. Guellouma and H. Cherroun: *From tree automata to rational tree expressions*. Int. J. Found. Comput. Sci., 29(6) 2018, pp. 1045–1062.
8. J. E. Hopcroft, R. Motwani, and J. D. Ullman: *Introduction to automata theory, languages, and computation (2. ed)*, Addison-Wesley, 2003.
9. D. Kuske and I. Meinecke: *Construction of tree automata from regular expressions*, in Developments in Language Theory, 12th International Conference, DLT 2008, Kyoto, Japan, September 16-19, 2008. Proceedings, 2008, pp. 491–503.
10. É. Laugerotte, N. O. Sebti, and D. Ziadi: *From regular tree expression to position tree automaton*, in Language and Automata Theory and Applications - 7th International Conference, LATA 2013, Bilbao, Spain, April 2-5, 2013. Proceedings, 2013, pp. 395–406.
11. N. Moreira, D. Nabais, and R. Reis: *State elimination ordering strategies: Some experimental results*, vol. 31, 08 2010, pp. 139–148.

12. T. Pecka, J. Trávníček, R. Polách, and J. Janoušek: *Construction of a pushdown automaton accepting a postfix notation of a tree language given by a regular tree expression.* 2018, pp. 6:1–6:12.
13. R. Polách, J. Janoušek, and B. Melichar: *Regular tree expressions and deterministic pushdown automata*, in Proceedings of the 7th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, 2011, pp. 70–77.

Enumerative Data Compression with Non-Uniquely Decodable Codes*

M. Oğuzhan Külekci[1], Yasin Öztürk[1], Elif Altunok[1], and Can Yılmaz Altıniğne[2]

[1] Informatics Institute
Istanbul Technical University, Istanbul, Turkey
{kulekci,ozturky17,altunok}@itu.edu.tr
[2] School of Computer and Communication Sciences
EPFL, Zurich, Switzerland
can.altinigne@epfl.ch

Abstract. Non-uniquely decodable codes can be defined as the codes that cannot be uniquely decoded without additional disambiguation information. These are mainly the class of non-prefix-free codes, where a codeword can be a prefix of other(s), and thus, the codeword boundary information is essential for correct decoding. Although the codeword bit stream consumes significantly less space when compared to prefix–free codes, the additional disambiguation information makes it difficult to catch the performance of prefix-free codes in total. Previous studies considered compression with non-prefix-free codes by integrating rank/select dictionaries or wavelet trees to mark the code-word boundaries. In this study we focus on another dimension with a block–wise enumeration scheme that improves the compression ratios of the previous studies significantly. Experiments conducted on a known corpus showed that the proposed scheme successfully represents a source within its entropy, even performing better than the Huffman and arithmetic coding in some cases. The non-uniquely decodable codes also provides an intrinsic security feature due to lack of unique-decodability. We investigate this dimension as an opportunity to provide compressed data security without (or with less) encryption, and discuss various possible practical advantages supported by such codes.

1 Introduction

A coding scheme basically replaces the symbols of an input sequence with their corresponding codewords. Such a scheme can be referred as non-uniquely decodable if it is not possible to uniquely decode the codewords back into the original data without using a disambiguation information. We consider non-uniquely decodable non–prefix–free (NPF) codes in this study as the most simple representative of that family. In NPF coding, a codeword can be a prefix of other(s), and the ambiguities may arise since the codeword boundaries cannot be determined without explicit specification of the individual codeword lengths.

Due to the lack of that unique decodability feature, NPF codes has received very limited attention [4,12,1] in the data compression area. Although the codewords become smaller when compared to their prefix-free versions, they should be augmented with the disambiguation information for proper decoding, and the additional space consumption of that auxiliary data structures unfortunately eliminates the advantage of short codewords. Thus, the challenge here is to devise an efficient way of representing the codeword boundaries.

* This study has been supported by the TÜBİTAK research fund under the grant number 117E865.

M. Oğuzhan Külekci, Yasin Öztürk, Elif Altunok, Can Yılmaz Altıniğne: Enumerative Data Compression with Non-Uniquely Decodable Codes, pp. 23–32.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9

The data structures to bring unique decodability for NPF codes was studied in [12]. More recently, the compression performance of NPF codes, which are augmented with wavelet trees [17] or rank/select dictionaries [18] to mark the code-word boundaries, had been compared with Huffman and arithmetic coding in [1]. It should be noted that using succinct bit arrays to mark the code-word boundaries had also been independently mentioned in some previous studies as well [5,6]. Although such NPF coding schemes are performing a bit worse in terms of compression, they support random-access on compressed data.

In this work, we study improving the compression performance of non-uniquely decodable codes with the aim to close the gap with the prefix-free codes in terms of compression ratio. We propose an enumerative coding [2,10] scheme to mark the codeword boundaries as an alternative of using wavelet trees or a rank/select dictionaries. Instead of representing the length of every codeword on the encoded bit stream, the codeword boundaries are specified in blocks of d consecutive symbols for a predetermined d value. Assume the codeword lengths of the symbols in a block are shown with a d–dimensional vector. The sum of the d individual codeword lengths is denoted by p, and the vector can be specified by its rank q among all d–dimensional vectors having an inner sum of p according to an enumeration scheme. Thus, a tuple $\langle p, q \rangle$, can specify the codeword boundaries in a d symbol long block.

The method introduced in this study represents an input data by replacing every symbol with a NPF codeword and then compressing the corresponding $\langle p, q \rangle$ tuples efficiently. Experiments conducted on a known corpus [1] showed that the compression ratios achieved with the proposed method reaches the entropy bounds and improve the arithmetic and Huffman coding ratios. To the best of our knowledge, this is the first study revealing that non-prefix-free codes can catch compression ratios quite close to entropy of the data.

In recent years, compressive data processing, which can be defined as operating directly on compressed data for some purpose, had been mentioned as a primary tool to keep pace with ever growing size in big data applications [15]. For instance, many database vendors are focusing on compressed databases [21] to cope with the massive data management issues. On the other hand, it is becoming a daily practice to benefit from cloud services both for archival and processing of data. Obviously, the primary concern in using such a third-party remote service is the privacy and security of the data, which can be achieved simply by encryption. Encrypted compressed data is both space efficient and secure. However, the encryption level introduces several barriers in processing the underlying compressed data. Alternative solutions that investigate the privacy of the data without incorporating an encryption scheme have also been considered [11,8,7,16]. Thus, new compression schemes respecting the data privacy without damaging the operational capabilities on the compressed data may find sound applications [14] in practice. For instance, similarity detection of documents without revealing their contents [13] and privacy preserving storage with search capabilities are some potential applications based on those non-uniquely decodable codes.

The outline of the paper is as follows. We start by defining the non-prefix-free codes from a compression perspective, and then proceed by introducing our enumeration scheme to represent the disambiguation information. The proposed compression method as a whole is described next, which is then followed by the experimental results and discussions addressing the opportunities and future work.

[1] Manzini's corpus available at `http://people.unipmn.it/manzini/lightweight/corpus/index.html`.

2 The Non-Prefix-Free Coding

$T = NONPREFIXFREE$

$\Sigma = \{E, R, F, N, I, O, P, X\}$
$O = \{3, 2, 2, 2, 1, 1, 1, 1\}$

$\Sigma \rightarrow W$

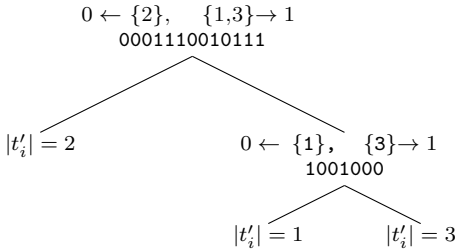
Σ:	E	R	F	N	I	O	P	X
W :	0	1	00	01	10	11	000	001

T =	N	O	N	P	R	E	F	I	X	F	R	E	E
T' =	01	11	01	000	1	0	00	10	001	00	1	0	0
L =	2	2	2	3	1	1	2	2	3	2	1	1	1

a) *The $T' = NPF(T)$ coding of a sample text T.*

T′ = 01110100010001000100100
B = 10101010011101010010111

b) *Code-word boundaries in $T' = NPF(T)$ marked on a bit array B.*



c) *Code-word lengths array L in T' represented with a wavelet tree.*

Figure 1. The NPF coding and code-word boundary representation alternatives with bitmap and wavelet tree.

$T = t_1 t_2 \cdots t_n$ is a sequence of symbols, where $t_i \in \Sigma = \{\epsilon_1, \epsilon_2, \ldots, \epsilon_\sigma\}$. Each symbol $\epsilon_i \in \Sigma$ requires $\lceil \log \sigma \rceil$ bits in fixed-length coding, and the total length of T then becomes $n \cdot \lceil \log \sigma \rceil$ bits. Without loss of generality, assume the symbols of the alphabet Σ are ordered according to their number of occurrences on T such that ϵ_1 is the most and ϵ_σ is the least frequent ones.

Let's assume a code word set $W = \{w_1, w_2, \ldots, w_\sigma\}$, where each w_i denotes the minimal binary representation of $(i + 1)$ as $w_i = MBR(i + 1)$. The minimal binary representation of an integer $i > 1$ is the bit string $MBR(i) = b_1 b_2 \cdots b_{\log i}$ such that $i = 2^{\log i} + \sum_{a=1}^{\log i} b_a \cdot 2^{\log i - a}$. For example, $MBR(13) = 101$, which is actually the binary representation of 13 omitting the leftmost 1 bit.

This definition generates $W = \{0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$, where the code words $w_i \in W$ has varying bit lengths, and W is not prefix free as some code words appear as the prefixes of others. The Kraft's inequality [9], which states the necessary condition that a code-word set W is uniquely decodable if $\sum_{i=1}^{\sigma} 2^{-|w_i|} \leq 1$, does not hold on this W. For each code-word length $\ell_k \in \{1, 2, 3, \ldots, \lfloor \log(\sigma + 1) \rfloor\}$, there are 2^{ℓ_k} code words except the last code-word length by which less symbols might be represented when $\sigma \neq 2^h - 2$ for some h. Thus, it is clear that $2^{-1} + 2^{-1} + 2^{-2} + 2^{-2} + 2^{-2} + 2^{-2} + 2^{-3} + \cdots + 2^{-(\lceil \log(\sigma+2) \rceil - 1)} \geq 1$ when $\sigma > 2$.

The non-prefix-free coding of T is the transformation obtained by replacing each $t_i = \epsilon_j$ with $t'_i = w_j$ according to the $\Sigma \rightarrow W$ mapping for all $1 \leq i \leq n$ as shown by $NPF(T) = T' = t'_1 t'_2 \cdots t'_n$, $t'_i \in W$. In T', the most significant two symbols from Σ are shown by 1 bit, and the following four symbols are denoted by 2 bits, and so on.

The total number of bits in the non-prefix-free coded sequence T is $|NPF(T)| = 1 \cdot (o_1 + o_2) + 2 \cdot (o_3 + \cdots + o_6) + \cdots + (\lceil \log(\sigma + 2) \rceil - 1) \cdot (o_{2^{\lceil \log(\sigma+2) \rceil - 1} - 1} + \cdots + o_\sigma)$, where o_i is the number of appearances of ϵ_i in T.

The code word boundaries on T' are not self-delimiting and cannot be determined without additional information. Previous approaches [12,1] used wavelet trees and rank/select dictionaries to mark the boundaries are shown in Figure 1. Although these compressed data structures are very useful to support random access on the compressed sequence, it had been observed in [1] that the compression ratios achieved by these methods are a bit worse than the Huffman and arithmetic coding. In this study, we incorporate an enumerative coding to specify the codeword boundaries.

Assume a list of items are ordered according to some definition, and it is possible to reconstruct any of the items from its rank in the list. In such a case, transmitting the index instead of the original data makes sense, and provides compression once representing the rank takes less space than the original data. That is actually the main idea behind enumerative coding [3]. We apply this scheme to represent the code-word boundaries in a sequence of NPF codewords. Empirical observations, as can be followed in the experimental results section, revealed that the usage of the proposed enumerative scheme can compress data down to its entropy.

3 Enumerative Coding to Mark Codeword Boundaries

One simple thing that can be achieved to mark the codeword boundaries is to store the codeword lengths of individual symbols on the input text T. These lengths vary from minimum codeword length 1 to a maximum of $\ell_{max} = \lfloor \log(\sigma + 1) \rfloor$ bits. The sequence of n codeword length information can then be compressed via a Huffman or arithmetic codec. However, our initial experiments showed that this coding does not provide a satisfactory compression ratio, where the total compression ratio cannot reach the entropy of the source sequence. With the motivation of marking the boundaries of multiple symbols instead of single individuals may improve the compression performance, we decided to test whether such a block-wise approach would help.

A block is defined as consecutive d symbols, and thus, there are $r = \lceil \frac{n}{d} \rceil$ blocks on T. When n is not divisible by d, we pad the sequence with the most frequent symbol. We maintain a list of r tuples as $R = \{\langle p_1, q_1 \rangle, \langle p_2, q_2 \rangle, \ldots \langle p_r, q_r \rangle\}$ such that

- $p_i = |t'_{(i-1)d+1}| + |t'_{(i-1)d+2}| + \cdots + |t'_{i \cdot d}|$ for $1 \leq i \leq r$, where $|t'_j|$ denotes the bit length of the codeword corresponding to symbol t_j, and
- q_i represents the rank of the vector $\langle |t'_{(i-1)d+1}|, |t'_{(i-1)d+2}|, \ldots, |t'_{i \cdot d}| \rangle$ among all possible d-dimensional vectors whose elements sum up to p_i.

For example on the example shown in Figure 1, if we assume a block size of $d = 3$, then $p_1 = 2+2+2 = 6$ since the codeword lengths of the first three symbols (`NON`) are all 2 bits. All possible 3-dimensional vectors whose elements are in range $[1 \ldots 3]$ and sum up to 6 can be listed in lexicographic order as $\langle 1, 2, 3 \rangle$, $\langle 1, 3, 2 \rangle$, $\langle 2, 1, 3 \rangle$, $\langle 2, 2, 2 \rangle$, $\langle 2, 3, 1 \rangle$, $\langle 3, 1, 2 \rangle$, and $\langle 3, 2, 1 \rangle$. We observe that $\langle 2, 2, 2 \rangle$ is the fourth item in this list, and thus $q_1 = 4$. Similarly the lengths of the next block $\langle 3, 1, 1 \rangle$ can be shown by $\langle 5, 6 \rangle$ since $3 + 1 + 1 = 5$ and $\langle 3, 1, 1 \rangle$ is the sixth item in the lexicographically sorted possibilities list $\langle 1, 1, 3 \rangle$, $\langle 1, 2, 2 \rangle$, $\langle 1, 3, 1 \rangle$, $\langle 2, 1, 2 \rangle$, $\langle 2, 2, 1 \rangle$, and $\langle 3, 1, 1 \rangle$.

In such a block-wise approach we need to devise an enumeration strategy to convert an input vector to an index and vice versa. We explain the building blocks in the following subsections.

3.1 Number of Distinct Vectors

Let $\psi(k, d, v)$ return the number of distinct d dimensional vectors, in which each dimension can take values from 1 to k, and they sum up to v in total. The total sum v should satisfy $d \leq v \leq (k \cdot d)$ since each dimension is at least 1 and at most k. If $v = d$ or $d = 1$, then there can be only one possible vector in which all dimensions are set to 1 in the former case and to k in the later case as there is only one dimension. The $\psi(k, d, v)$ function can be computed with a recursion such that $\psi(k, d, v) = \sum_{i=\alpha}^{\beta} \psi(k, d-1, v-i)$. This is based on setting one, say first, dimension to one of the possible value i and then counting the remaining $(d-1)$ dimensional vectors whose elements sum up to $(v-i)$. The pseudo code of this calculation is given in Algorithm 1.

3.2 Vector to Index

Assume we are given a d dimensional vector as $\langle v_1, v_2, \ldots, v_d \rangle$, where each $1 \leq v_i \leq k$ for a known k. We would like to find the lexicographical rank of this vector among all possible d–dimensional vectors with an inner sum of $v = v_1 + v_2 + \cdots + v_d$. First we can count how many of the d-dimensional vectors have a smaller number than v_1 in their first dimension. Next step is to count the number of vectors that have the same v_1 in the first dimension, but less than v_2 in the second position. We repeat the same procedure on remaining dimensions, and the sum of the computed vectors return the rank of our vector. This can be achieved via a recursion, which is shown in Algorithm 2, that uses the $\psi()$ function described above. As an example, for $d = 3$, and $k = 3$, assume we want to find the rank of $2, 2, 2$. First we count the number of vectors that has a 1 in its first position with an inner sum of 6 via the $\psi(k = 3, d = 2, v = 5)$ function, which returns us 2. Next, we count the vectors that has a 2 in first position and a value less than 2, which can take value only 1, in its second position. This can also be computed via $\psi(k = 3, d = 1, v = 3)$ function, which returns 1 since we have only one dimension to set. Now we know that there are 3 vectors that are enumerated before our input on the possibilities list. We do not need to search for the last dimension since it is not free and its value is already determined.

3.3 Index to Vector

In this case we are given a number I representing the rank of a d dimensional vector in a set of d dimensional vectors with a known inner sum v, and we aim to generate this vector. We start by setting the first dimension to the minimum value 1, and count how many possibilities exits by the $\psi(k, d-1, s-1)$. If this number is less than I, we decrease I by this value, set 2 for the first position and keep counting the possibilities in the same way until detecting the first value at which I is no longer larger. Thus, we have found the first dimension of the vector, we repeat the same procedure to detect the other dimensions. The pseudo code of this calculation is given at Algorithm 3.

Algorithm 1: $\psi(k, d, v)$

Input:
k: Maximum value of a dimension.
d: The number of dimensions.
v: The inner sum of the vectors.
Output:
Number of distinct d dimensional vectors with an inner sum of v.

```
1  if (v > k·d)||(v < d) then return 0;
2  if (d = 1)||(v = d) then return 1;
3  if (v = d + 1) then return d;
4  if (1 < v + k − k·d) then
5  |  α = v + k − k·d
6  else
7  |  α = 1
8  if (k < v − d + 1) then
9  |  β = k
10 else
11 |  β = v − d + 1
12 sum = 0;
13 for (i = α; i ≤ β; i+ = 1) do
14 |  sum+ =
   |    ψ(k, d − 1, v − i);
15 end
16 return sum;
```

Algorithm 2: VectorToIndex$(\langle v_1, v_2, \ldots, v_d\rangle, d, k)$

Input: k: Maximum value of a dimension. d: The number of dimensions. $v_1 \cdots v_d$: Input vector.
Output: Rank of the input vector among lexicographically sorted vectors with the same inner sum of $\sum_i v_i$.

```
1  v = v1 + v2 + ··· + vd ;
2  if (d = 1)||(v = d) then return 0;
3  index = 0;
4  for (i = 1; i < v1; i+ = 1) do
5  |  index+ = ψ(k, d − 1, v − i);
6  end
7  index+ = VectorToIndex(⟨v2, v3, ..., vd⟩, d − 1, k);
8  return index;
```

Algorithm 3: $IndexToVector(k, d, v, index)$

Input: k: Maximum value of a dimension d: The number of dimensions. v: The inner sum of the vectors. $index$: The rank of the vector among all possible vectors.
Output: The $\langle v_1, v_2, \ldots, v_d\rangle$ vector with $v_1 + v_2 + \cdots + v_d = v$, and rank $index$ among all possible vectors with inner sum v.

```
1  for (i = 1; i < d; i+ = 1) do
2  |  v_i = 1;
3  |  while (z = ψ(k, d − i, v − v_i) < index) do
4  |  |  index− = z;
5  |  |  v_i = v_i + 1;
6  |  end
7  |  v = v − v_i;
8  end
9  v_d = v;
```

4 The Complete Method

The pseudo-codes of the proposed encoding and decoding with the Non-uniquely decodable codes are given in Algorithms 4 and 5.

In the encoding phase, the NPF codeword stream B is simply the concatenation of the NPF codewords. At each d symbols long block, the total length of the codewords is the corresponding p_i value, which can take values from d to $k \cdot d$. This p_i value is encoded to the $Pstream$ by an adaptive compressor.

The index corresponding to the vector of the latest d codeword lengths is computed with the $VectorToIndex$ function as described in the enumeration section by using the p_i value as the inner sum. This q_i is then encoded into the $Qstream$ with p_i assumed to be the context in this compression. Notice that according to the p_i value, the number of possible vectors change, where there appears relatively small candidates for small p_i. When all the codewords in the block are 1 bit long, which means $p_i = d$, then there is no need to encode q_i since there is only one possibility. Similarly, $p_i = k \cdot d$ implies all codeword are maximum length k, and again nothing is required to add into the compressed $Qstream$.

The decoding phase is performed accordingly, where first the p_i value is extracted from $Pstream$. If the extracted p_i is equal to d or $k \cdot d$, this implies nothing has been added to the $Qstream$ in the coding phase since the target vectors are determined with single options. Otherwise, by using the p_i value as the context, the corresponding q_i is extracted from $Qstream$ followed by the $IndexToVector$ operation.

Algorithm 4:
Encode(T, d)

Input: $T = t_1 t_2 \cdots t_n$ is the input data, where $t_i \in \Sigma = \{\epsilon_1, \epsilon_2, \ldots, \epsilon_\sigma\}$. d is the chosen block length.
Output: The codeword bit-stream and the compressed $\langle p_i, q_i \rangle$ list.

$r = \lceil \frac{n}{d} \rceil$;
$B = \emptyset$;
Generate the NPF codeword set $W = \{w_1, w_2, \ldots, w_\sigma\}$;
$k = \lfloor \log(\sigma + 1) \rfloor$;
for $(i = 0; i < r; i+ = 1)$ **do**
    $p_i = 0$;
    for $(j = 0; j < d; j+ = 1)$ **do**
        $\epsilon_h = T[i \cdot d + j + 1]$;
        $B \leftarrow Bw_h$;
        $vec[j + 1] = |w_h|$;
        $p_i+ = vec[j + 1]$;
    end
    Encode p_i into $Pstream$ with an adaptive coder;
    if $(p_i \neq d)\&\&(p_i \neq k \cdot d)$ **then**
        $q_i = VectorToIndex(vec[], d, k)$;
        Encode q_i into $Qstream$ with an adaptive coder by using the *sum* value as the context;
end

Algorithm 5:
Decode($B, Pstream, Qstream, d, n, W$)

Input: B is the NPF codeword bit stream. $Pstream$ is the compressed p_i values. $Qstream$ is the compressed q_i values. $W = \{w_1, w_2, \ldots, w_\sigma\}$ is the NPF codeword set.
Output: The original data sequence $T = t_1 t_2 \cdots t_n$

$r = \lceil \frac{n}{d} \rceil$;
$k = \lfloor \log(\sigma + 1) \rfloor$;
for $(i = 0; i < r; i+ = 1)$ **do**
    Decode p_i from the $Pstream$;
    if $p_i = d$ **then**
        $\langle v_1, v_2, \ldots, v_d \rangle = \langle 1, 1, \ldots, 1 \rangle$;
    else if $p_i = k \cdot d$ **then**
        $\langle v_1, v_2, \ldots, v_d \rangle = \langle k, k, \ldots, k \rangle$;
    else
        Decode q_i from the $Qstream$ by using p_i as the context ;
        $\langle v_1, v_2, \ldots v_d \rangle \leftarrow IndexToVector(k, d, p_i, q_i)$;
    end
    for $(j = 1; j \leq d; j+ = 1)$ **do**
        $w_h \leftarrow$ Read next v_j bits from B;
        $t_{i \cdot d + j} = \epsilon_h$;
    end
end

5 Implementation and Experimental Results

Being directly proportional to the imbalance of the symbol frequencies in the source, the codewords with short lengths are expected to appear more, and thus, the d–dimensional vectors are in general filled with small numbers with small p_i values as a consequence. Figure 2 shows the distribution of block lengths and their corresponding number of distinct vectors by assuming $d = 6$ and $k = 7$. On the same figure also the observed frequencies of possible block bit lengths on a 100 megabyte English text are depicted, where the most frequent bit block length seems 15 here. We present the distribution of q_i values in the context of $p = 15$ on Figure 3 to give an idea about the imbalance that increases the success of representing codeword boundaries over the non–prefix–free codeword stream.

We have implemented the proposed scheme and compared compression ratio against both static and adaptive versions of the Huffman and arithmetic codes (AC) on the test corpus. While compressing the $Pstream$ and $Qstream$, we have used the adaptive arithmetic encoder of [19], and tested our scheme with different block sizes of $d = 2$, $d = 4$, and $d = 6$. Table 1 shows the compression performance of each scheme on various files in terms of bits spent per each symbol in total.

The experiments showed that for $d = 6$, the compression ratio of the Non-uniquely decodable codes generally improves the others. However, AC seems achieving better ratios on three files. On `howto` file the difference in between the AC and Non-uniquely decodable codes are very small as being in thousands decimal, which is not found to be meaningful. On `howto.bwt`, which is the same `howto` file after Burrows-Wheeler transform, the difference is sharper. This is mainly due to the fact that the runs in the

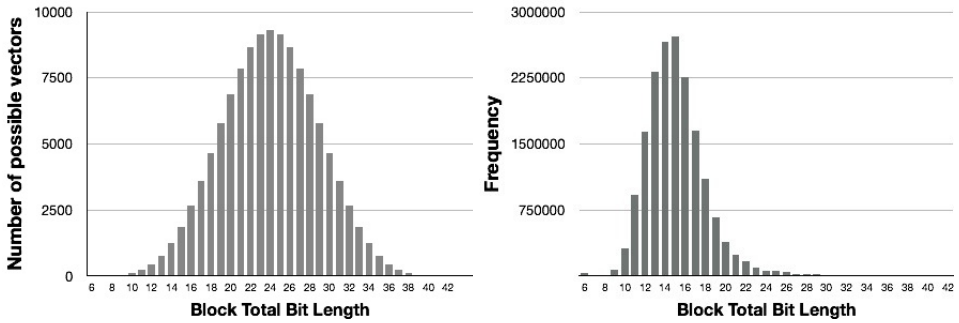


Figure 2. Assuming a block size of $d = 6$ symbols, and a maximum codeword length $k = 7$, a) presents possible bit block lengths and corresponding number of distinct vectors per each, b) presents the *observed* bit block lengths and their number of occurrences on 100MB of English text (etext file from the Manzini's corpus).

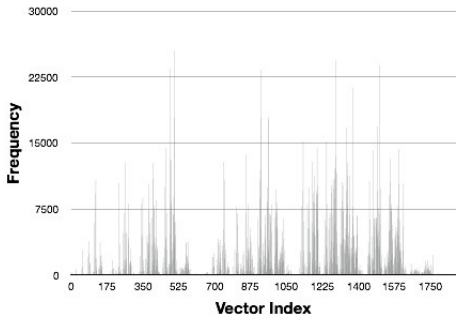


Figure 3. The distribution are 1875 distinct 6–dimensional vectors, where each dimension can take values from 1 to $k = 7$ with an inner sum of 15 on 100MB of English text according to our enumeration scheme.

BWT string may introduce an advantage for the adaptive codes. Notice that both files are around 40 megabytes and shorter than the other files except the `chr22.dna` file, on which our method performs clearly worse. In the current experimental observations it is thought that the performance of the proposed coding becomes better on large files with larger alphabets.

It is possible to increase the block size, particularly on larger volumes. However, when d becomes larger current implementation suffers from the slow down due to the recursive function implementations to find the enumerative index of a vector, and vice versa. Considering that our compression scheme is composed of three main components as the base non-prefix-free code stream, and over that the $Pstream$ and $Qstream$, we would like to monitor their respective space occupation on the final compressed size. Table 2 includes the diffraction of these three components for different d values tested. There appears a trade off such that the $Pstream$ gets better compressed with increased block size, where the reverse works for $Qstream$.

6 Discussions and Conclusions

This study has shown that non-prefix-free codes with an efficient representation of the codeword boundaries can reach the entropy bounds in compression as is the case

File	Size	Symbols	Entropy	Huffman		Arithmetic		NPF		Non-uniquely decodable		
				Stat.	Adapt.	Stat.	Adapt.	RS	WT	d=2	d=4	d=6
sprot34.dat	109MB	66 (k=6)	4.762	4.797	4.785	4.764	4.749	5.434	5.178	4.869	4.790	**4.698**
chr22.dna	34MB	5 (k=2)	2.137	2.263	2.195	2.137	**1.960**	2.957	2,616	2.468	2.466	2.462
etext99	105MB	146 (k=7)	4.596	4.645	4.595	4.604	4.558	5.140	4,553	4.632	4.570	**4.553**
howto	39MB	197 (k=7)	4.834	4.891	4.779	4.845	**4.731**	5.300	4.215	4.856	4.759	4.736
howto.bwt	39MB	198 (k=7)	4.834	4.891	3.650	4.845	**3.471**	5.300	4.215	4.143	3.950	3.949
jdk13c	69MB	113 (k=6)	5.531	5.563	5.486	5.535	5.450	6.404	5.658	5.577	5.460	**5.275**
rctail96	114MB	93 (k=6)	5.154	5.187	5.172	5.156	5.139	5.766	5.408	5.164	5.020	**4.818**
rfc	116MB	120 (k=6)	4.623	4.656	4.573	4.626	4.529	5.094	4.853	4.685	4.555	**4.463**
w3c2	104MB	256 (k=8)	5.954	5.984	5.700	5.960	5.659	6.648	5.820	5.826	5.686	**5.617**

Table 1. Compression ratio comparison between the proposed scheme, NPF rank/select and wavelet tree [1], arithmetic, and Huffman coding in terms of bits/symbol.

File	Codeword Stream	Pstream			Qstream		
		d=2	d=4	d=6	d=2	d=4	d=6
sprot34.dat	2.686	1.476	0.909	0.659	0.707	1.196	1.353
chr22.dna	1.494	0.718	0.504	0.399	0.256	0.468	0.568
etext99	2.516	1.316	0.789	0.580	0.800	1.265	1.457
howto	2.618	1.451	0.885	0.655	0.787	1.256	1.464
howto.bwt	2.618	1.183	0.781	0.604	0.342	0.552	0.726
jdk13c	3.263	1.449	0.871	0.642	0.866	1.327	1.370
rctail96	2.878	1.462	0.893	0.659	0.824	1.250	1.281
rfc	2.516	1.472	0.911	0.677	0.697	1.128	1.271
w3c2	3.436	1.548	0.949	0.706	0.841	1.301	1.475

Table 2. The diffraction of the codeword stream, Pstream, and Qstream on the number of bits used per symbol for different d values.

for prefix–free codes. It is not possible to decode the codeword stream without the codeword boundary information encoded in *Pstream* and *Qstream*. This property may make sense to achieve a level of data security with less encryption, particularly, on high-entropy data [14]. There is no need to encrypt the codeword stream when one would like to secure the data, which can significantly reduce the encryption load. More than that, it is still possible to make some operations such as search and similarity computations on the codeword stream. Yet another opportunity might appear in the distributed storage of the data, where keeping the NPF codewords and codeword boundary informations in different sites can help in providing the privacy. Same idea may also apply in content delivery networks.

Besides the compression ratio, where this study mainly concentrated, the memory usage and the speed of compression are surely important parameters in practice. Current implementation is slow due to two main facts as NPF codewords are not byte–aligned, and the vector to/from index enumerations are consuming additional time. The former problem is common to all variable length codes, which can be overcome by benefiting from the Huffman coding tables idea [20]. The enumeration time consumption can also be decreased by using tables which include the precomputed vector to/from index calculations by sacrificing a bit more memory. The algorithm engineering of the proposed scheme along with the possible applications in data security area are possible venues of research as a next step. Surely, better data structures to encode the codeword boundaries is open for improvement.

References

1. B. Adaş, E. Bayraktar, and M. O. Külekci: *Huffman codes versus augmented non-prefix-free codes*, in Experimental Algorithms, Springer, 2015, pp. 315–326.
2. J. Cleary and I. Witten: *A comparison of enumerative and adaptive codes.* IEEE Transactions on Information Theory, 30(2) 1984, pp. 306–315.
3. T. Cover: *Enumerative source encoding.* IEEE Transactions on Information Theory, 19(1) 1973, pp. 73–77.
4. M. Dalai and R. Leonardi: *Non prefix-free codes for constrained sequences*, in Information Theory, 2005. ISIT 2005. Proceedings. International Symposium on, IEEE, 2005, pp. 1534–1538.
5. P. Ferragina and R. Venturini: *A simple storage scheme for strings achieving entropy bounds*, in Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 2007, pp. 690–696.
6. K. Fredriksson and F. Nikitin: *Simple compression code supporting random access and fast string matching*, in Experimental Algorithms, Springer, 2007, pp. 203–216.
7. D. W. Gillman, M. Mohtashemi, and R. L. Rivest: *On breaking a huffman code.* IEEE Transactions on Information theory, 42(3) 1996, pp. 972–976.
8. J. Kelley and R. Tamassia: *Secure compression: Theory\ & practice.* IACR Cryptology ePrint Archive, 2014 2014, p. 113.
9. L. Kraft: *A device for quantizing, grouping, and coding amplitude-modulated pulses*, Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering, 1946.
10. M. O. Kulekci: *Enumeration of sequences with large alphabets.* arXiv preprint arXiv:1211.2926, 2012.
11. M. O. Külekci: *On scrambling the burrows–wheeler transform to provide privacy in lossless compression.* Computers & Security, 31(1) 2012, pp. 26–32.
12. M. O. Külekci: *Uniquely decodable and directly accessible non-prefix-free codes via wavelet trees*, in Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on, IEEE, 2013, pp. 1969–1973.
13. M. O. Külekci, I. Habib, and A. Aghabaiglou: *Privacy–preserving text similarity via non-prefix-free codes*, in International Conference on Similarity Search and Applications, Springer, 2019, pp. 94–102.
14. M. O. Külekci and Y. Öztürk: *Applications of non-uniquely decodable codes to privacy-preserving high-entropy data representation.* Algorithms, 12(4) 2019, p. 78.
15. P.-R. Loh, M. Baym, and B. Berger: *Compressive genomics.* Nature biotechnology, 30(7) 2012, pp. 627–630.
16. R. B. Muralidhar: *Substitution cipher with non-prefix-free codes*, Master's thesis, San Jose State University, 2011.
17. G. Navarro: *Wavelet trees for all.* Journal of Discrete Algorithms, 25 2014, pp. 2–20.
18. D. Okanohara and K. Sadakane: *Practical entropy-compressed rank/select dictionary*, in Proceedings of the Meeting on Algorithm Engineering & Expermiments, Society for Industrial and Applied Mathematics, 2007, pp. 60–70.
19. A. Said: *Introduction to arithmetic coding-theory and practice*, Tech. Rep. HPL–2004–76, Hewlett Packard Laboratories, Palo Alto, CA, April 2004.
20. A. Siemiński: *Fast decoding of the huffman codes.* Information Processing Letters, 26(5) 1988, pp. 237–241.
21. T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte: *The implementation and performance of compressed databases.* ACM Sigmod Record, 29(3) 2000, pp. 55–67.

Fast Exact Pattern Matching in a Bitstream and 256-ary Strings

Igor O. Zavadskyi

Taras Shevchenko National University of Kyiv
Kyiv, Ukraine
2d Glushkova ave.
ihorza@gmail.com

Abstract. A few known techniques of exact pattern matching, such as 2-byte read, fast loop, and sliding search windows, are improved and applied to two related sub-problems. At first, we present a new family of pattern matching algorithms, performing efficiently over 256-ary alphabets. Taking them as an underlying solution, we build the algorithms for searching a string in a bitstream. It turns out that in both cases our algorithms outperform all the other tested methods for all tested pattern lengths.

1 Introduction

Finding all occurrences of a given substring in a larger body of text is one of the most fundamental problems in computer science. In this presentation we consider an important sub-problem consisting in searching a bitstream pattern in a bitstream text. It is of interest, since a vast variety of data is presented in a binary form, e.g. images, videos, archived data etc. The recent invention of data compression codes, which perform close to entropy and at the same time support data search in a compressed file [1], also actualizes the demand for bitstream pattern matching.

The first non-trivial bitstream pattern matching algorithm was presented in 2007 by S.T. Klein and M.K. Ben-Nissan [16]. Since then, S. Faro and T. Lecroq developed a number of improved bitstream pattern matching techniques implemented in the binary-hash and binary-skip algorithms (both presented in [8]) and the most advanced Binary-Faro-Lecroq (BFL) algorithm [7]. Also, the adaptation of the FED algorithm (Fast matching with Encoded DNA sequences) [15] to binary search has to be mentioned, although it is somewhat inferior to BFL.

The general idea of any non-trivial bitstream pattern matching method consists in avoiding time consuming bit-level operations as far as possible. A search is performed on the byte level, and only when a candidate substring is found, the bit-level procedure checks if a true pattern occurrence takes place. Therefore, any bitstream pattern matching technique is based on the underlying algorithm of pattern matching on a 256-ary alphabet (assuming a byte is 8 bits). For example, the algorithm [16] is based on Boyer-Moore search method, while the BFL algorithm combines a multi-pattern version of the BNDM algorithm [17] with the simplified shift strategy of Commentz-Walter algorithm [3].

Of course, an underlying byte-level algorithm has to be implemented in a multi-pattern version, since it searches not a single pattern but 8 patterns corresponding to 8 possible alignments of a binary substring towards the byte boundaries. However, the performance of a multi-pattern version of an algorithm strongly depends on the performance of its single-pattern version, and thereby the development of pattern

Igor O. Zavadskyi: Fast Exact Pattern Matching in a Bitstream and 256-ary Strings, pp. 33–47.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9

matching algorithms that efficiently perform on 256-ary alphabets is a key for solving the bitstream pattern-matching problem.

According to [9] and our own experiments, the following algorithms are of the most interest when the alphabet consists of 256 characters, in different testing environments and for different pattern lengths: comparison-based Fast Search (FS, [2]), Franek-Jennings-Smyth [10] and variations of algorithms exploiting the bit-parallelism idea: Simplified BNDM with q-grams (SBNDMq, [4]), hybrid of SBNDM and Boyer-Moore-Horspool algorithms (SBNDM-BMH, [11]), Forward SBNDM (FSBNDM, [6]), SD-NDM with a "greedy" fast loop (GSBNDM, [19]) and BNDM for long patterns (LB-NDM, [18]).

As shown in [19], the performance of BNDM-type algorithms can be improved by applying the technique of *2-byte read*, which is of special interest when $|\Sigma| = 256$. However, the performance of 2-byte reads suffers from the expansion of shift tables. As experiments show, the running time of an algorithm increases significantly when its shift tables together with some other preprocessed data cease to fit into processor L1 cache, which is typically $16 - 64$ KB. For a 256-ary alphabet, the size of a shift table with 2-byte indices is 64 KB, which exceeds the "L1 cache limit" in most cases.

Two other techniques, which can significantly improve the algorithm performance for different alphabets and pattern lengths, are based on using multiple sliding search windows [13] and skipping the occurrence check with a help of so called *fast loop* [14].

For a 256-ary alphabet, featuring a simple comparison-based method with these techniques can outperform the algorithms based on bit-parallelism. Indeed, the experimental results in Tables 1−2 show that the comparison-based Fast Search algorithm with 6 or 8 sliding windows performs faster than all other known methods (except for those hereinafter developed) for almost all tested pattern lengths.

In this presentation we improve all three aforementioned techniques. At first, we present a "compromise" solution between 2-byte and 1-byte reads, forming the index of a search table from the values of more than 1 but less than 2 sequential bytes of a text, typically 13-15 bits. We call this method a *1.5-byte read*. It allows us to increase the average length of a shift comparing to "1-byte read" algorithms, while spending rather less memory than the 2-byte read approach requires. Similar ideas were discussed at Stringmasters [21], although not published yet. Then we offer a few tricks, which improve the performance of the fast loop and sliding windows techniques. As a result, we construct a family of comparison-based algorithms, which outperform all other known solutions in discovering patterns of different lengths in texts with uniformly distributed characters from a byte-based 256-ary alphabet. These algorithms are called *Z-Byte* and discussed in Section 2. A family of bitstream search algorithms, based upon Z-Byte, is constructed in Section 3. We call these algorithms *Z-Bit*. Finally, the results of algorithm benchmarking are presented in Section 4.

Let us note that an attempt to combine multiple-character reads and multiple search windows has been done in our previous work [23]. However, the methods presented hereinafter are simpler, and when applying to 256-ary alphabet, faster.

Throughout the entire presentation we use the following notations:

- Σ - alphabet of an input text and a pattern
- $|\Sigma|$ - size of the alphabet
- b - number of bits in a byte, by default $b = 8$
- k - number of significant bits in a 1.5-byte read, typically $b < k < 2b$

2 Pattern matching over an alphabet of 256 characters

In this section we assume that each character of a text occupies one byte of memory, and all bits of this byte are significant, i.e. $|\Sigma| = 256$. By $T[0\,..\,n-1]$ and $P[0\,..\,m-1]$ we denote a text and a pattern respectively.

2.1 1.5-byte read

At first, let us discuss the essence of the "1-byte read", "2-byte read" approaches and their "1.5-byte read" modification. Let Z be a one-dimensional shift table for some pattern matching algorithm and i is the index of some character of a text. The 1-byte read approach assumes the value $Z[T[i]]$ to be the shift length, as in the Boyer-Moore-Horspool algorithm (BMH, [12]), or other data the shift depends on. BMH method is shown schematically in Alg. 1.

Algorithm 1: Boyer-Moore-Horspool algorithm

```
1 foreach c ∈ Σ do Z[c] ← m;                          // Preprocessing
2 for i ← 0 to m − 2 do Z[P[i]] ← m − 1 − i;
3 pos ← 0;                                             // Search
4 while pos ≤ n − m do
5 |   check the occurrence at pos;
6 |   pos ← pos + Z[T[pos + m − 1]];
```

Line 6 corresponds to a "bad character" shift, which maximal possible value is m. When $|\Sigma| = 256$ and the pattern is short, the probability of a maximal BMH shift is high enough. E.g. it equals $(255/256)^m$ for random text and random pattern. But when the pattern is longer, the "1-byte" bad character shift becomes not sufficient. For example, if $m = 512$, $(255/256)^m \approx 0.135$.

The situation can be amended by using 2 sequential bytes of a text as a basis of a bad-character shift. E.g. for $m = 512$, the probability of a maximal shift of a random pattern over a random text becomes greater than 0.99. The computationally efficient implementation of this approach is discussed in [19]: the expression $Z[T[i]]$ is transformed into $Z[word(T[i], T[i+1])]$, where the function *word* converts two sequential bytes of memory into a two-byte word in a processor register. In C programming language this function can be implemented by the type-casting mechanism, i.e. $word(T[i], T[i+1])$ equals to `*(unsigned short*)(T+i)`. In fact, the time complexity of calculating the $Z[T[i]]$ and $Z[word(T[i], T[i+1])]$ values is the same, while the memory complexity is significantly increased from 256 bytes needed for 1-byte reads to 64 KB occupying by a shift table with 2-byte indices.

However, for reasonable pattern lengths, e.g. less than 1000 bytes, the memory complexity can be significantly reduced with a little impact on the shift length. This can be achieved by using not all bits of a 2-byte word in the index of a shift table. Some of bits can be suppressed by applying the mask: $Z[word(T[i], T[i+1])\&mask]$. For example, if $m = 512$ and the mask contains 14 'one' bits, the probability of a maximal shift for random text and random pattern will be $((2^{14}-1)/2^{14})^{511} \approx 0.97$, which is only 0.022 less than that one for 2-byte read. At the same time, the shift table will contain 2^{14} vs. 2^{16} elements, i.e. 4 times less. Of course, one extra operation $\&mask$ has to be performed in the search loop, but in most cases this expense will be more than covered by the fact that the shift table fits into L1 cache.

2.2 Double fast loop

The other disadvantage of Algorithm 1 consists in the necessity to check the possible occurrence of a pattern at each iteration of the search loop. However, the occurrence check can be avoided by applying the so called "fast loop", which was first introduced in [14]. This technique is implemented in a number of algorithms and is particularly effective when more than one character of a text is processed at each iteration, e.g. in the EBOM [6] and SBNDM [11] algorithms. In this case the probability of a maximal shift is high enough, and this implies that the exact length of a shift may not be stored in a shift table. Instead, we only determine whether a shift of some constant length, e.g. m or $m-1$, is *safe*, i.e. cannot cause missing the pattern occurrence. If it is, we make this constant length shift, otherwise the occurrence has to be checked, and the value of a next shift is calculated by some other algorithm. This approach allows us to avoid loading the shift length value from memory to a computer register, which is quite consuming operation.

Featuring the BMH method with the fast loop of aforementioned type and 1.5-byte reads, we get the Algorithm 2. The shift table Z contains not the shift lengths, but some "flag" information. That is, if $Z[T[i]] = 1$, the shift of the search window $m-1$ characters right is safe. The fast loop is implemented in lines 7 and 8. Although two bytes of a text are read in line 7, only $k < 2b$ bits of each two-byte word are used to form the index of the shift table Z. After exiting the fast loop, we check the occurrence and get the shift value from the Quick Search shift table (QS, [20]). To exit the fast and main loops at the end of a text correctly, it should be appended by a stop-pattern.

Algorithm 2: Search algorithm with the fast loop and 1.5-byte reads

```
1  mask ← 2^k − 1;                                          // Preprocessing
2  foreach i ∈ [0; 2^k) do Z[i] ← 1;
3  for i ← 0 to m − 2 do
4  |  Z[word(P[i], P[i + 1])] ← 0
5  pos ← m − 2;                                             // Search
6  while pos < n do
7  |  while Z[word(T[pos], T[pos + 1])&mask] ≠ 0 do
8  |  |  pos ← pos + m − 1;
9  |  check the occurrence at pos − m + 2;
10 |  pos ← pos + QS[pos + 2];
```

We tested the Alg. 2 performance for $|\Sigma| = 256$ and different pattern lengths. The results show that this algorithm often outperform other known Boyer-Moore-based algorithms, such as FS or FJS, although the latter ones implement more sophisticated techniques to process the situation when the fast loop is terminated. However, taking into account that the 1.5-byte read implies high probability of a maximal shift, we developed other simple and efficient method to process the exit from the fast loop. The main idea is based upon the assumption that the inequality $Z[word(T[i-1], T[i])\&mask] \neq 0$ holds with the same probability as the inequality $Z[word(T[i], T[i+1])]\&mask] \neq 0$. It implies that even if the fast loop condition fails, we can make one step back and, very likely, continue the fast loop from that position. This "double fast loop" is shown in Alg. 3, which preprocessing phase is the same as in Alg. 2. The double fast loop is especially efficient for long patterns, when the probability of the fast loop termination is higher, while the relative impact of the

left shift in line 7 is less. Note that this loop always starts from adding $m-1$ to the current position *pos* in line 4, and to compensate this addition, we subtract $m-1$ from the QS shift length in line 10 and initialize *pos* with the value -1 in line 1.

Algorithm 3: Search phase of the algorithm with the double fast loop

```
1  pos ← −1;
2  while pos < n do
3  |  repeat
4  |  |  pos ← pos + m − 1;
5  |  |  while Z[word(T[pos], T[pos + 1])&mask] ≠ 0 do
6  |  |  |  pos ← pos + m − 1;
7  |  |  pos ← pos − 1;
8  |  until Z[word(T[pos], T[pos + 1])&mask] ≠ 0 ;
9  |  check the occurrence at pos − m + 3;
10 |  pos ← pos + QS[pos + 3] − m + 2;
```

We denote the Alg. 3 by Zk-Byte, where k is the number of 'one' bits in the mask ($8 < k < 16$). If $k = 16$, lines 5 and 8 contain full 2-byte reads and variable *mask* is not needed, while in algorithm Z8 the reference to the shift table looks as $Z[T[pos]]$ and the variable *pos* can be incremented by m in lines 4 and 6. In the next subsection we discuss how to shift the search window m characters right for any value of k.

2.3 Longer shifts

Let $T[pos]$ and $T[pos+1]$ be the two last characters of a search window. If the condition $Z[word(T[pos], T[pos+1])\&mask] \neq 0$ in lines 5 and 8 of Alg. 3 is satisfied, these two bytes of a text cannot belong to the pattern together. Still, the character $T[pos+1]$ can coincide with $P[0]$ and that's why the search window can be shifted safely by $m-1$ characters at most, not by m. If the pattern is short, this decrement of a maximal safe shift length can have a meaningful effect on algorithm performance.

This situation can be amended by adjusting the array Z: we can assign 0 to all elements of the form $Z[word(c, P[0])\&mask]$, $c \in \Sigma$. As a result, if the last byte of a search window coincides with $P[0]$, the shift will be considered as non-maximal, and m can be assumed to be the value of the maximal shift. It seems that this will increase the probability of a non-maximal shift in a random text by 1/256 at most, which is not too much. However, the *endianness* of a machine, i.e., an order in which the bytes of a value are loaded from memory into a processor register, has to be taken into account. Let us examine the function *word*, used in lines 5 and 8 of Alg. 3. It converts two sequential bytes of memory into a two-byte number in a processor register. In most programming languages this function can be implemented by the type-casting mechanism, e.g. `(unsigned short*)` operator in C language. However, its result depends on the endianness. On a little endian machine (e.g. x86 processor) bytes of a value are loaded into a register in the reverse order (Fig. 1). This is an unwanted situation if we compose a shift table index of the full last byte of a search window and a part of the second to last, because a resultant value will be shifted in a register to the left (Fig. 1 (a)), which makes the size of the shift table the same as for the 2-byte read. However, the situation in Fig. 1 (b) - not full last byte and full second to last - is also unwanted, because only the part of the last byte of a search window should coincide with the part of $P[0]$ byte to make the shift non-maximal. Then, the increase of the non-maximal shift probability for a random text will be up

to $1/2^{k-8}$, which significantly reduces the probability of a fast loop continuation. For example, if $k = 12$, the latter probability will be reduced by up to 1/16.

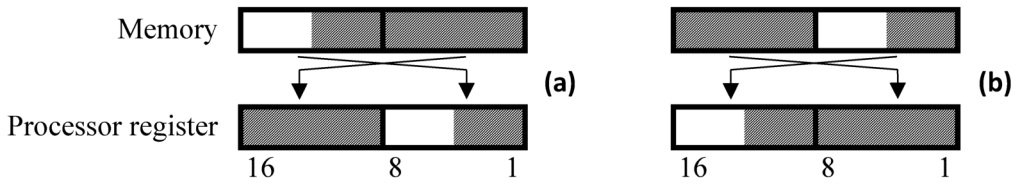


Figure 1. Loading a 2-byte value from memory on a little endian machine. 'White' bits are reset to zero by *mask*, 'grey' bits remain significant. *mask* resets the bits of the highest byte (a) or lowest byte (b).

Nonetheless, on a little endian machine the permutation shown in Fig. 1 (b) becomes admissible if the text is searched from right to left. Then, extending the maximal shift from $m-1$ to m decreases the probability of a fast loop termination by up to 1/256, and the index of a shift table is composed of the low bits of a two-byte word. We call right-to-left search algorithms *reverse* and denote them by the letter "R", e.g. RZ12-Byte. The reverse search method RZk-Byte is shown in Alg. 4.

Algorithm 4: The reverse search algorithm RZk-Byte

```
1  mask ← 2^k − 1;                                          // Preprocessing
2  foreach i ∈ [0; 2^k) do Z[i] ← 1;
3  for i ← 0 to m − 2 do
4  |  Z[word(P[i], P[i + 1])] ← 0
5  for i ← 0 to 2^(k−b) do
6  |  Z[(i << b)|P[m − 1]] ← 0
7  foreach c ∈ Σ do RQS[c] ← m + 1;
8  for i ← m − 1 downto 0 do RQS[P[i]] ← i + 1;
9  pos ← n;                                                 // Search
10 repeat
11 |  repeat
12 |  |  pos ← pos − m;
13 |  |  while Z[word(T[pos], T[pos + 1]) & mask] ≠ 0 do
14 |  |  |  pos ← pos − m;
15 |  |  pos ← pos + 1;
16 |  until Z[word(T[pos], T[pos + 1]) & mask] ≠ 0;
17 |  pos ← pos − 1;
18 |  check the occurrence at pos;
19 |  pos ← pos − RQS[T[pos − 1]] + m;
20 until pos ≥ m;
```

The text T is assumed to be prepended with a stop pattern ($T[-m \dots -1] = P[0 \dots m-1]$) to exit the double fast loop, given in lines 11−16, when the search finishes. The search window starts at the position $n - m$ and moves to the left. The variable *pos* always addresses the beginning of a search window in which first two bytes are used to determine the possibility of a maximal shift by m characters. The algorithm steps 1 character forward between the internal and external fast loops (line 15) and reverts this step in line 17 if the external fast loop is terminated. After

checking the occurrence we make the "Reverse Quick Search" shift using the shift table RQS, which is filled in lines 7−8. Lines 5−6 intended to block the maximal shift when the first character of a search window coincides with the last character of the pattern; this code is equivalent to $Z[word(c, P[m-1])\&mask] \leftarrow 0, c \in \Sigma$. Finally, lines 1−4 of the preprocessing stage are similar to those ones in Alg. 3.

2.4 Sliding windows

The performance of Z-Byte and RZ-Byte algorithms can be improved significantly by implementing a two sliding windows technique [13]. However, for reverse methods it should be slightly changed, since we can search a text from right to left only, while in standard method windows are moving towards each other until they meet. The search phase of Alg. 4 can be rewritten as shown in Alg. 5. Both sliding windows are moving towards the beginning of a text. The first window specified by $pos1$ starts in the middle of a text, while the second one specified by $pos2$ starts in the end. The main loop is finished when the first window reaches the beginning of a text. After that, the second window may be moved further if it has not reached the middle position yet (lines 17−22). The preprocessing phase of Alg. 5 is just the same as of Alg. 4.

Algorithm 5: The search phase of the reverse search algorithm with 2 sliding windows RZk-Byte-w2

```
1  pos1 ← ⌊n/2⌋;
2  pos2 ← n − m;
3  while pos1 ≥ 0 do
4  |   while Z[word(T[pos1], T[pos1 + 1]) & mask] ≠ 0 &
   |    Z[word(T[pos2], T[pos2 + 1]) & mask] ≠ 0 do
5  |   |   pos1 ← pos1 − m;
6  |   |   pos2 ← pos2 − m;
7  |   if Z[word(T[pos1 + 1], T[pos1 + 2]) & mask] = 0 then
8  |   |   check the occurrence at pos1;
9  |   |   pos1 ← pos1 − RQS[T[pos1 − 1]];
10 |   else
11 |   |   pos1 ← pos1 − m + 1;
12 |   if Z[word(T[pos2 + 1], T[pos2 + 2]) & mask] = 0 then
13 |   |   check the occurrence at pos2;
14 |   |   pos2 ← pos2 − RQS[T[pos2 − 1]];
15 |   else
16 |   |   pos2 ← pos2 − m + 1;
17 while pos2 > ⌊n/2⌋ do
18 |   while Z[word(T[pos2], T[pos2 + 1]) & mask] ≠ 0 do
19 |   |   pos2 ← pos2 − m;
20 |   if pos2 > ⌊n/2⌋ then
21 |   |   check the occurrence at pos2;
22 |   pos2 ← pos2 − RQS[T[pos2 − 1]];
```

Let us note that filling the shift table Z with zeros and ones allows us to join the 1.5-character checks with the bitwise '&' operation (line 4) instead of the slower logical *AND* as in "classical" sliding windows approach [13]. Also, the external fast

loop is replaced with separate 'if-else' blocks for each of two sliding windows (lines 7−11 and 12−16). This is done to take the advantage of the situation when the maximal shift can be made only in one of two sliding windows.

We denote the Alg. 5 as RZk-Byte-w2 - the reverse search algorithm with a k-bit read and 2 sliding windows. As opposed to "classical" approach, we can use an odd number of sliding windows. For example, the parallel searches in the algorithm RZk-Byte-w3 will start at positions $\lfloor n/3 \rfloor$, $\lfloor 2n/3 \rfloor$ and n. Of course, each pair of sliding windows in non-reverse Z-algorithms can be moved towards each other. However, for large enough texts the experiments show no significant difference in performance between "bi-directional" and "uni-directional" sliding windows methods.

3 Pattern matching in a bitstream

In this section we denote the array of full bytes of a pattern by $P[0\,..\,m-1]$, and $T[0\,..\,n-1]$ denotes the input text. The last bytes $P[m]$ and $T[n-1]$ are not full and padded with zeros if the pattern and/or text bit length is not a factor of 8. Otherwise, $P[m]$ is assumed to be 0, while $T[n-1]$ is a full byte. The byte next to the text, $T[n]$, is always 0 as well as $P[m+1]$. The bit length of a pattern we denote by l, and $p[0\,..\,l-1]$ is the array of pattern bits. By "search window" we mean a $(m-1)$-byte substring of a text that is supposed to belong to the pattern.

Hereinafter we assume a little endianness and discuss the "right-to-left" bitstream search by the example of RZk-Bit algorithm. Its general structure is similar to the structure of the underlying RZk-Byte algorithm. At each iteration of the fast loop we try to move the search window as far as possible to the left. *pos* addresses the leftmost byte of a search window. Thus, the window occupies the bytes $T[pos], \ldots, T[pos + m-2]$, while bytes $T[pos-1]$, $T[pos+m-1]$, and possibly $T[pos+m]$, may contain the left and right "tails" of the pattern.

The search window can be safely moved $m-1$ bytes to the left, if two conditions are met (taking into account the endianness and applying the *mask*): (a) the pair of bytes $(T[pos], T[pos+1])$ does not belong to the pattern, and (b) some prefix of the pair $(T[pos], T[pos+1])$ of length greater than 8 does not coincide with the pattern suffix. The shift table Z is filled in accordance with this conditions at the preprocessing phase (Alg. 8) and checked during the search phase (lines 5 and 8 of Alg. 6). The whole block of code in lines $3-8$ of Alg. 6 implements the double fast loop discussed in Subsection 2.2.

After the exit from the double fast loop, we check if the pattern can be aligned with the search window shifted q bits to the left, $q = 0, \ldots, b-1$. This check is performed by the procedure $CheckMatch(q, pos)$ invoked in line 11 of Alg. 6. It is not efficient to test all b possible values of q. Instead, we store in the set $\lambda[c]$ all values $q < b$ such that the factor $p[q] \ldots p[q+b-1]$ of a pattern coincides with the byte c. This implies that all possible occurrences such that the byte $T[pos]$ is the leftmost full byte of a text that belongs to the pattern, are checked in lines 10 and 11 of Alg. 6.

After the occurrence check, *pos* is safely moved 1 byte to the left after the addition in line 12 and subtraction in line 4 on the next iteration of the main loop.

Algorithm 6: The search phase of the RZk-Bit algorithm

```
1  pos ← n − 1;
2  repeat
3  |  repeat
4  |  |  pos ← pos − m + 1;
5  |  |  while Z[word(T[pos], T[pos + 1]) & mask] ≠ 0 do
6  |  |  |  pos ← pos − m + 1;
7  |  |  pos ← pos + 1;
8  |  until Z[word(T[pos], T[pos + 1]) & mask] ≠ 0;
9  |  pos ← pos − 1;
10 |  foreach q ∈ λ[T[pos]] do
11 |  |  CheckMatch(q, pos);
12 |  pos ← pos + m − 2;
13 until pos ≥ m − 1;
```

The body of the procedure $CheckMatch(q, pos)$ is given in Alg. 7. We have to check if a window, occupying q lowest bits of the byte $T[pos-1]$ and the next $l-q$ bits of a text, coincides with the pattern. The function *byte* in line 2 truncates a 2-byte value to its lowest byte. It can be be implemented as a type-casting `(unsigned char)` in C language. Generally, in line 2 we compose a series of bytes from pairs of the adjacent bytes of a text taking q lowest bits of the byte $T[j]$ and $b-q$ highest bits of the byte $T[j+1]$. The composed bytes are compared with the bytes of the pattern. If they all match, the last byte is composed and compared with $P[m]$ in line 5. Since the byte $P[m]$ is not full, the composed byte is truncated by the mask $lastMask$ to significant bits of $P[m]$ only.

Algorithm 7: Procedure $CheckMatch(q, pos)$

```
1  start ← j ← pos − 1;
2  while j − start < m AND
     byte((T[j] << (b − q))|(T[j + 1] >> q)) = P[j − start] do
3  |  j ← j + 1
4  if j − start = m then
5  |  if ((T[j] << (b − q))|(T[j + 1] >> q))&lastMask = P[m] then
6  |  |  output(start · k + q)
```

The value $lastMask$ as well as other values and tables, which remain constant through the search phase, is calculated at the preprocessing phase, shown in Alg. 8. Let us explain how the loop in lines 4−11 of Alg. 8 works, in which the shift table Z is constructed. For each 16-bit substring of a pattern, starting at the bit position i, the corresponding element of the array Z is specified with the flag 0, which denotes a non-maximal shift (the short suffixes of the pattern, $l-i \leq b$, are not processed, since their possible alignments with search window prefixes have no impact on a safe shift of the length $m-1$). During the search phase it will be checked if this substring coincides with some substring of a text. Each of these substrings does not consist of a continuous sequence of significant bits, but has a "hole" of insignificant ones shown in the upper part of Fig. 1 (b). However, the function $bitWord$ invoked in line 5 of Alg. 8 performs the bits permutation shown if Fig. 1 (b) and returns the mentioned substring in the compact form shown in the lower part of that figure.

If such substring is aligned to the boundaries of a two-byte word, this permutation becomes trivial and can be completed by the type-casting. This is how the function *word* is calculated in the search phases of RZ-Byte and RZ-Bit methods. However, in a bitstream the mentioned substring may intersect with 3 adjacent bytes of a pattern, and the permutation becomes trickier. It is explained in the comments to Alg. 9.

Algorithm 8: The preprocessing phase of the RZk-Bit algorithm

```
1  mask ← 2^k − 1; m ← ⌊l/b⌋; lastMask ← byte((2^b − 1) << (b − l mod b));
2  foreach i ∈ [0; b) do c ← (P[0] << i)|(P[1] >> (b − i)) λ[c] ← λ[c] ∪ {i};
3  foreach i ∈ [0; 2^k) do Z[i] ← 1;
4  foreach i ∈ [0; l − b) do
5  |  t ← bitWord(i, mask);
6  |  if l − i ≥ 2b then
7  |  |  Z[t] ← 0;
8  |  else
9  |  |  if l − i > 3b − k then s ← 2b − (l − i);          // 2 (a)
10 |  |  else s ← k − b;                                   // 2 (b)
11 |  |  foreach j ∈ [0; 2^s) do Z[t|(j << b)] ← 0;
```

The formatted substring is assigned to the variable t in line 5 of Alg. 8. If $l-i \geq 2b$, the whole substring belongs to the pattern, and $Z[t] = 0$ (line 7). Otherwise, the substring is truncated at the end of the pattern, and after the *bitWord* permutation a "hole" of insignificant bits may appear inside the formatted substring, Fig. 2 (a). The length s of this "hole" is calculated in lines 9 (Fig. 2 (a)) or 10 (Fig. 2 (b)). In line 11 the "hole" in the substring t is filled with all possible 2^s values. This way, the set of indices of zero elements of the array Z is constructed.

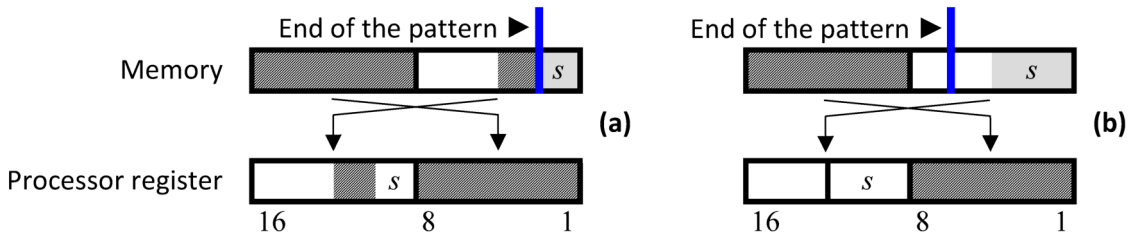


Figure 2. The *bitWord* permutation (Alg. 9) of the right tail of a pattern. The bits, remaining significant after the permutation, highlighted with dark grey. The bits that would have remained significant if not for the end of a pattern, highlighted with light grey. (a) The pattern ends within significant bits of the lowest byte; (b) the pattern ends within insignificant bits.

Algorithm 9: Function $bitWord(i, mask)$, little endian machine

```
1  r ← ⌊i/b⌋;                                   // index of the highest byte
2  s ← (P[r] << 2b)|(P[r + 1] << b)|P[r + 2];   // load 3 bytes from memory
3  s ← s << (i mod b);      // shift to the left edge of a 3-byte word
4  mask1 ← (2^b − 1) << 2b;      // mask for the highest byte, 0xff0000
5  mask2 ← mask&((2^b − 1) << b);            // mask for the middle byte
6  return ((s&mask1) >> 2b)|(s&mask2); // move highest byte to lowest
```

4 Benchmarking experiments

The experimental algorithms execution times are shown in Table 1 (256-ary search) and Table 2 (bitstream search). The algorithms were implemented in C programming language and compiled with GNU GCC compiler. The C codes of some Z-algorithms are published in [22], while most of other algorithms have been taken from [5].

4.1 Remarks on implementation

There are three adjustments in C versions of our algorithms, which are not shown in Alg. 2−6 for simplicity.

The first adjustment deals with addressing the characters of a text. It is reasonable to assume that addressing a character via pointer like `*ptr` will be a bit faster than addressing the array element like `T[pos]`, because the latter expression in fact means `*(T+pos)` and requires one extra addition. Whilst other operations such as additions and comparisons have the same time complexity either for pointers or indices. Indeed, for our algorithms, the experiments show that the gain from using pointers instead of indices is up to 20% for patterns of length 2, up to 6% for patterns of length 4, 0.5−1.5% for patterns of length 8 and insignificant for longer patterns. For other algorithms, the results are rather contradictory. That is why all Z/RZ algorithms were tested in "pointer" versions, while other algorithms have been run in their original versions. Thus, to make a comparison more relevant, the timing of R/RZ algorithms may be increased in above-mentioned proportions. However, even after this adjustment Z/ZR algorithms still remain the fastest ones for all pattern lengths.

The second adjustment has to be made in reverse algorithms in order to correctly process the algorithm stop condition. In algorithms $4-6$ the text was assumed to be prepended by a stop pattern. However, it can be problematic due to "left-to-right" organization of memory structures in programming languages: we can easily allocate extra memory after the text to append a string to it, but there is no technique to allocate extra memory just before the address stored in a given pointer T addressing the beginning of a text, except for shifting the whole text right or taking into account this specificity before invoking a search function. The other option is to: (1) before the main search loop, backup the beginning of a text and replace it with a stop pattern; (2) after the main loop, restore the beginning of a text and search the pattern in it using some other simple algorithm. This approach has been implemented in the tested reverse algorithms.

And the last adjustment relates to short patterns, less than 3 bytes in length. In this case, the length of a maximal shift in Zk-Byte or RZ-Bit algorithms is equal to $m - 1 = 1$. And since in the double fast loop we step 1 character back, this loop may become endless. Therefore, in Zk-Byte ($8 < k < 16$) and RZ-Bit algorithms, the short patterns are considered as a special case and processed with a single fast loop.

4.2 Testing environment

The executables have been run on 40 computers with different processors and L1 cache varying from 24KB up to 64KB; all computers were little endian machines. The experimental 10MB text contains a set of English Wikipedia articles encoded by the multi-delimiter code $D_{2,4-\infty}$ [1]. Although this code compresses texts not far from entropy ($2-3$% away), it makes possible the direct data search in a compressed

file without its decompression, just like Fibonacci codes, (s, c)-dense codes and other codes with delimiters. This assigns a special meaning to a pattern matching problem in application to delimiter-encoded data. Also, the algorithms have been tested on a random 10MB text, however, the results are not shown in this presentation, since they differ from the timing given below insignificantly and do not change the general picture of algorithms superiority.

The results, given in milliseconds, represent a weighted average time of 500×40 runs of each algorithm searching patterns of length $2 - 512$ randomly taken from the text, a new pattern for each run. In a bitstream search the pattern starts from a random bit. Weight numbers has been taken on pro rata basis of average computer benchmark. In other words, the below presented time values have been calculated as follows.

1. Get the average time $T_{a,m,c}$ of 500 runs of each algorithm a for each pattern length m on each computer c.
2. Multiply $T_{a,m,c}$ by $total\_time/40c\_time$, where $total\_time$ is the sum of all values $T_{a,m,c}$ for all computers / algorithms / pattern lengths, c_time is the sum of all values $T_{a,m,c}$ for a specific computer c, and 40 is the number of computers.
3. For each pair (a, m), get the average value of $T_{a,m,c}$ with respect to all computers.

This approach allows us to interpret all computers as units of equal importance independently of their actual benchmarks.

4.3 Experimental results

Generally, 44 different variations of Z-Byte and RZ-Byte algorithms were tested, with $1 - 6$ sliding windows and the parameter k varying in the range $12 - 15$. Also, the algorithms Z8 and Z16 based on full byte reads have been tested in 1, 2 and 3 sliding windows versions. It appears that when $m \geq 8$, the 13-bit or 14-bit reads are most efficient. That's why only the results for 13 and 14-bit versions of Z/RZ-Byte algorithms with 1.5 read are shown in Table 1 and only for those numbers of sliding windows, which give the optimal result for at least one pattern length. Let us note that when a pattern becomes longer, the efficiency of 14-bit reads in comparison to 13-bit reads increases. And when the pattern length is very short, $m \leq 4$, the 1.5-byte read appears to be superfluous and the algorithm Z8-w2, supporting 1-byte read and 2 sliding windows, demonstrates the best performance.

A number of known algorithms were tested for comparison, both in original and "2-byte read" versions, if applicable. As it is seen, a 2-byte read does not improve an algorithm's performance for $|\Sigma| = 256$. At the same time, different representatives of Z/RZ-families outperform all other known algorithms on all examined pattern lengths. This testifies that our "1.5-byte read" approach appears to be more efficient. Indeed, if the pattern is not extremely long, the probability of a maximal shift based on analysing 2 byte suffix of a search window is not much higher than if we analyse $13-14$ bits. However, in the latter case the shift table fits into L1 cache, which is rather more important factor. This is also confirmed by the performance of Z16-Byte-w2, the fastest algorithm among Z-algorithms with 2-byte read, which always under-performs some 1.5-read algorithms.

Among other algorithms the comparison-based Fast Search algorithms with 6 or 8 sliding windows demonstrate the best results for all pattern lengths. The ratio of running times of the best algorithm not belonging to Z/RZ-families to the winner is

shown in the last row of Table 1 for each pattern length. It is significant for short ($m \leq 8$) and long ($m \geq 128$) patterns, however, for middle-sized ones the Z/RZ timing is only slightly less than that of the FS with 6 or 8 sliding windows.

Thus, we can conclude that the bit-parallelism approach to pattern matching is not too efficient on a 256-ary alphabet. The main reason for that is that comparison-based algorithms provide quite good average shift length having at the same time less operational complexity. Instead of implementing the bit-parallelism, the better way for improvement consists in (a) involving bits of more than one character into bad-character comparisons; (b) making use of multiple sliding search windows; (c) tuning the fast loop technique.

This is confirmed by the performance of bitstream pattern matching algorithms, presented in Table 2. As seen for all investigated pattern lengths, the RZ-Bit algorithms, based on the principles (a)-(c), outperform the BFL method, based upon the bit-parallel algorithm BNDM. And this outperformance is essential even for no-sliding windows versions of RZ-Bit. Apart from the above mentioned reasons, the superiority of the RZ-Bit family may be explained by the following factors: (d) although the

Pattern length m	2	4	8	16	32	64	128	256	512
Z8-Byte-w2	**40.63**	**26.22**	19.64	16.10	14.58	14.02	12.52	13.16	12.35
RZ13-Byte	67.50	35.70	21.21	14.91	11.80	10.67	8.82	6.42	4.51
RZ14-Byte	67.92	35.80	21.16	14.94	11.85	10.77	8.92	6.40	4.40
RZ13-Byte-w2	49.34	26.29	15.90	12.05	10.36	9.66	8.33	5.85	3.50
RZ14-Byte-w2	49.59	26.48	15.97	12.06	10.47	9.85	8.41	5.88	**3.41**
RZ13-Byte-w3	49.45	26.42	**15.67**	11.17	9.59	9.00	7.93	5.65	3.86
RZ14-Byte-w3	49.78	26.70	15.91	11.26	9.74	9.15	8.02	5.65	3.70
RZ13-Byte-w5	50.89	27.18	16.04	**11.08**	9.50	9.03	8.09	5.80	4.32
RZ14-Byte-w5	51.38	27.50	16.31	11.27	9.70	9.14	8.19	5.73	4.06
Z13-Byte-w3	94.31	30.75	15.95	11.20	**9.44**	**8.92**	**7.90**	**5.60**	3.83
Z14-Byte-w3	99.79	31.04	16.21	11.28	9.60	9.09	8.03	5.62	3.71
RZ16-Byte-w2	49.87	28.34	17.94	14.03	10.86	9.91	8.29	6.60	4.43
FSBNDM	69.76	35.97	20.29	13.63	11.61	11.67	12.04	12.21	12.26
FSBNDM-2byte	90.33	50.59	30.55	20.43	16.47	16.58	17.02	17.23	17.13
FSBNDM31	224.37	75.28	34.50	18.01	12.81	12.88	13.13	13.26	13.24
FSBNDM31-2byte	215.67	79.00	40.54	23.91	17.54	17.68	18.10	18.32	18.21
GSBNDMq2	–	43.05	20.84	13.53	11.45	11.53	11.70	11.82	11.82
GSBNDMq2-2byte	–	54.03	29.15	19.32	15.67	15.75	16.06	16.30	16.19
SBNDMq2	130.04	44.40	21.44	13.83	11.50	11.56	11.69	11.82	11.82
SBNDMq2-2byte	199.07	72.79	37.28	23.01	16.89	16.98	17.32	17.55	17.43
SBNDM/BMH	152.10	78.45	43.68	25.89	19.86	19.97	20.38	20.68	20.51
LBNDM	118.87	64.43	38.63	25.03	18.93	15.80	13.83	9.88	7.33
FJS	201.94	121.84	70.95	39.17	23.03	17.99	16.48	19.16	19.46
FS	259.70	130.52	68.52	36.15	21.24	17.33	16.53	22.84	21.14
FSw4	77.47	39.73	21.91	13.01	9.96	9.64	9.23	10.22	8.80
FSw6	62.46	32.35	18.23	11.44	9.65	9.47	8.97	8.97	7.56
FSw8	57.15	30.93	19.11	12.65	9.62	9.46	8.88	8.72	7.07
Best non-Z to best Z ratio	1.41	1.18	1.16	1.03	1.02	1.06	1.12	1.56	2.07

Table 1. Algorithms running times, $|\Sigma| = 256$ (milliseconds)

maximal length of a long-shift in the BFL algorithm is $2m-1$, in practice it is often $2m-3$ or less, while any total maximal shift in two sliding windows of RZ-Bit algorithm is $2m-2$, i.e. longer in average; (e) when the pattern is shortened below 60 bits, and importance of a long-shift decreases, the timing difference between RZ-Bit and BFL becomes especially large, which indicates a higher efficiency of a bit-alignment checking technique implemented in the RZ-Bit algorithms.

Pattern length	20	40	60	80	100	200	300	400	500
RZ13-Bit	157.65	48.28	34.83	26.03	22.22	15.27	13.17	12.52	12.04
RZ14-Bit	148.76	44.52	32.22	23.74	20.37	13.90	12.14	11.49	11.15
RZ15-Bit	145.22	42.73	31.23	23.10	19.86	13.74	12.12	11.51	11.13
RZ13-Bit-w2	129.49	35.91	26.81	20.88	18.25	13.89	12.29	11.84	11.54
RZ14-Bit-w2	120.65	33.24	24.81	19.07	**16.90**	13.08	11.78	11.35	11.10
RZ15-Bit-w2	116.87	32.57	24.50	18.96	17.03	13.15	11.94	11.53	11.31
RZ13-Bit-w3	135.61	38.50	28.40	21.96	19.09	13.90	12.25	11.82	11.51
RZ14-Bit-w3	126.74	35.19	26.08	19.79	17.25	12.74	11.43	11.03	10.77
RZ15-Bit-w3	122.25	34.03	25.47	19.44	17.10	**12.72**	**11.41**	**11.03**	**10.77**
RZ16-Bit-w2	**102.02**	**30.56**	**23.76**	**18.91**	17.05	13.35	12.26	11.95	11.66
BFL	277.77	91.46	63.80	43.23	36.63	23.38	21.65	25.29	31.32
FED	397.07	130.04	86.42	52.13	46.26	31.14	28.40	27.97	28.95
Best non-Z to best Z ratio	2.72	2.99	2.68	2.28	2.16	1.83	1.89	2.29	2.68

Table 2. Running times on a bitstream (milliseconds)

5 Conclusions

An efficient approach to pattern matching in a bitstream has been investigated and tested as well as underlying algorithms of string matching in 256-ary texts. It relies on improving the 2-byte read principle as well as tuning the fast loop and sliding windows techniques. The averaged results of testing provided on 40 computers show that different representatives of the developed families of algorithms outperform all other tested solutions for all studied pattern lengths.

References

1. A. V. Anisimov and I. O. Zavadskyi: *Variable-length prefix codes with multiple delimiters.* IEEE Transactions on Information Theory, 63(5) 2017, p. 2885–2895.
2. D. Cantone and S. Faro: *Fast-search algorithms: New efficient variants of the Boyer-Moore pattern-matching algorithm.* Journal of Automata, Languages and Combinatorics, 10(5/6) 2005, pp. 589–608.
3. B. Commentz-Walter: *A string matching algorithm fast on the average*, in Proceedings of International Colloquium on Automata, Languages, and Programming, 1979, pp. 118–132.
4. B. Durian, J. Holub, H. Peltola, and J. Tarhio: *Tuning BNDM with q-grams*, in Proceedings of the Workshop on Algorithm Engineering and Experiments, I. Finocchi and J. Hershberger, Eds. SIAM, New York, New York, USA, 2009, pp. 29–37.
5. S. Faro and T. Lecroq: *String matching research tool: Exact string matching algorithms.* https://www.dmi.unict.it/ faro/smart/algorithms.php.
6. S. Faro and T. Lecroq: *Efficient variants of the backward-oracle-matching algorithm*, in Proceedings of the Prague Stringology Conference, J. Holub and J. Zdarek, Eds. Czech Technical University in Prague, Czech Republic, 2008, pp. 146–160.

7. S. Faro and T. Lecroq: *An efficient matching algorithm for encoded DNA sequences and binary strings*, in Proceedings of Combinatorial Pattern Matching, G. Kucherov and E. Ukkonen, Eds., 2009, p. 106–115.
8. S. Faro and T. Lecroq: *Efficient pattern matching on binary strings*, in 35th International Conference on Current Trends in Theory and Practice of Computer Science, 2009, p. Poster.
9. S. Faro and T. Lecroq: *The exact online string matching problem: a review of the most recent results.* ACM Computing Surveys (CSUR), 45(2) 2013, p. article 13.
10. F. Franek, C. Jennings, and W. Smyth: *A simple fast hybrid pattern-matching algorithm.* J. Discret. Algorithms, 5(4) 2007, pp. 682–695.
11. J. Holub and B. Durian: *Fast variants of bit parallel approach to suffix automata.* Paper presented at the Second Haifa Annual International Stringology Research Workshop of the Israeli Science Foundation, 2005.
12. N. R. Horspool: *Practical fast searching in strings.* Soft.Pract.Exp., 10(6) 1980, pp. 501–506.
13. A. Hudaib, R. Al-khalid, D. Suleiman, M. A. A. Itriq, and A. Al-Anani: *A fast pattern matching algorithm with two sliding windows (tsw).* Journal of Computer Science, 4(5), p. 393–401.
14. A. Hume and D. Sunday: *Fast string searching.* Softw. Pract. Exp., 21(11) 1991, pp. 1221–1248.
15. J. Kim, E. Kim, and K. Park: *Fast matching method for DNA sequences*, in Proceedings of Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, 2007, p. 271–281.
16. S. Klein and M. Ben-Nissan: *Accelerating Boyer Moore searches on binary texts*, in Proceedings of International Conference on Implementation and Application of Automata, CIAA-07, 2007, p. 130–143.
17. G. Navarro and M. Raffinot: *A bit-parallel approach to suffix automata: Fast extended string matching*, in Proceedings of Combinatorial Pattern Matching, 1998, pp. 14–33.
18. H. Peltola and J. Tarhio: *Alternative algorithms for bit-parallel string matching*, in Proceedings of the 10th International Symposium on String Processing and Information Retrieval SPIRE'03, 2003, pp. 80–94.
19. H. Peltola and J. Tarhio: *String matching with lookahead.* Discrete Applied Mathematics, 163 2014, pp. 352–360.
20. D. Sunday: *A very fast substring search algorithm.* Comm. ACM, 33(8) 1990, pp. 132–142.
21. B. W. Watson, F. Franek, J. Holub, C. Iliopoulos, and B. Smyth: *Fractional n-grams for shifting.* Idea presented at StringMasters 2014 at McMaster University, Hamilton, Ontario, Canada, 2014.
22. I. O. Zavadskyi: *The Z-family algorithms: Implementation in C programming language.* https://github.com/zavadsky/stringology.
23. I. O. Zavadskyi: *A family of exact pattern matching algorithms with multiple adjacent search windows*, in Proceedings of the Prague Stringology Conference, J. Holub and J. Zdarek, Eds. Czech Technical University in Prague, Czech Republic, 2017, p. 152–166.

Fast Practical Computation of the Longest Common Cartesian Substrings of Two Strings

Simone Faro[1], Thierry Lecroq[2], and Kunsoo Park[3]

[1] University of Catania, Department of Mathematics and Computer Science, Italy
faro@dmi.unict.it
[2] Normandie Univ, UNIROUEN, LITIS, 76000 Rouen, France
thierry.lecroq@univ-rouen.fr
[3] Seoul National University, Seoul, Korea
kpark@theory.snu.ac.kr

Abstract. Cartesian trees have been introduced 40 years ago. They are associated to strings of numbers. They are structured as heap and original strings can be recovered by symmetrical traversal of the trees. The Cartesian tree matching problem appeared very recently. It consists of finding all substrings of a given text which have the same Cartesian tree as that of a given pattern. Here we present two methods for computing the longest common Cartesian substrings of two strings. The first method is a classical linear suffix tree based method. The alternative method runs in quadratic worst case time but is more space economical. Experiments show that the alternative solution runs faster for short strings.

1 Introduction

Cartesian trees have been introduced by Vuillemin [11]. They are associated to strings of numbers. They are structured as heap and original strings can be recovered by symmetrical traversal of the trees. They have many applications including finding patterns in time series data such as share prices in stock markets. It has been shown that they are connected to Lyndon trees [7,3], to Range Minimum Queries [4] or to parallel suffix tree construction [9]. Recently new results on Cartesian pattern matching appear [8,10,6]. It consists of finding substrings of a text that have the same Cartesian tree as a pattern. Recent studies concern finding periods in Cartesian tree matching [1].

In this article we are interested in computing the longest common Cartesian substrings of two strings which means common substrings of maximal length and that have the same Cartesian tree. A usual linear time method for computing longest common substrings for classical strings consists of building the generalized suffix tree of the two strings and the deepest internal nodes (in terms of string depth) having leaves for suffixes of both strings identify longest common substrings. This method can be applied for computing longest common Cartesian substrings of two strings. However the suffix tree has to be built on the parent-distance representation of the two strings and classical suffix tree construction algorithms cannot be used. We propose a quadratic time algorithm for computing the longest Cartesian substrings of two strings that uses only constant extra space in addition to the two strings and their parent-distance representation. Experimental results show that for short strings and in most practical settings our alternative solution is faster than the suffix tree based method.

This article is organized as follows. Section 2 presents the notations and definitions used throughout the rest of the article. Section 3 presents the method for constructing

Simone Faro, Thierry Lecroq, Kunsoo Park: Fast Practical Computation of the Longest Common Cartesian Substrings of Two Strings, pp. 48–60.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9

the Cartesian tree of a string. Section 4 briefly presents the suffix tree based method for computing longest common Cartesian substrings. Section 5 describes our new alternative solution. Section 6 presents experimental results. Section 7 concludes the article.

2 Notations and Definitions

A string is a sequence of symbol drawn from an alphabet Σ, which is a set of integers. We assume that a comparison between any two symbols of the alphabet can be done in constant time. For a string x, $x[i]$ represents the i-th symbol of x, and $x[i\,..\,j]$ represents the substring of x starting from position i and ending at position j. We denote by $x_i = x[1\,..\,i]$ the prefix of x of length i.

Let x be a string of numbers of length m. The Cartesian tree $\mathcal{CT}(x)$ of x is the binary tree where:

- the root corresponds to the index i of the minimal element of x (if there are several occurrences of the minimal element, the leftmost one is chosen);
- the left subtree of the root corresponds to the Cartesian tree of $x[1\,..\,i-1]$;
- the right subtree of the root corresponds to the Cartesian tree of $x[i+1\,..\,m]$.

For simplicity in what follows we will use the symbol $x[i]$ to refer to both the i-th character of x and the node of $\mathcal{CT}(x)$ whose key is i, depending on the context.

The following definition of the *right path* of a binary tree is particularly relevant to this paper.

Definition 1 (Right path). *The right path, $rp(T)$, of a binary tree T is the sequence of nodes encountered starting from the root of the tree and always going right.*

3 Construction of the Cartesian Tree

The construction of the Cartesian tree of a string x of length m can be done by means of an iterative procedure which iterates over the elements of x, proceeding from left to right, and computes the Cartesian tree of x_{i+1} from the Cartesian tree of x_i, for $1 \leq i < m$.

To better describe such approach we observe that the Cartesian tree of $x[1]$ consists of a single node, while $\mathcal{CT}(x_{i+1})$ can be computed from $\mathcal{CT}(x_i)$ by identifying the number of nodes that are on the right path of $\mathcal{CT}(x_i)$ but not on the right path of $\mathcal{CT}(x_{i+1})$.

Going deeper into the details, let $rp(\mathcal{CT}(x_i)) = \langle x[j_1], x[j_2], \ldots, x[j_k] \rangle$ be the right path of $\mathcal{CT}(x_i)$, with $1 \leq k \leq i$ and where $x[j_1]$ is the root of the Cartesian tree and $j_k = i$. Assume that $x[j_u] < x[i+1] < x[j_{u+1}]$, for some $0 \leq u \leq k$. We can distinguish three cases, as depicted in Figure 1:

1. if $u = 0$ then $x[i+1]$ is less than $x[j_1]$, the current root of the tree, and $x[i+1]$ becomes the new root of $\mathcal{CT}(x_{i+1})$;
2. if $0 < u < k$ then $x[i+1]$ is the smallest value on the right of $x[j_u]$ and all elements in the substring $x[j_u+1\,..\,i]$ are greater than $x[i+1]$. Then $x[i+1]$ is inserted as the right child of $x[j_u]$ and the subtree rooted at $x[j_u+1]$ is moved on the left of $x[i+1]$.

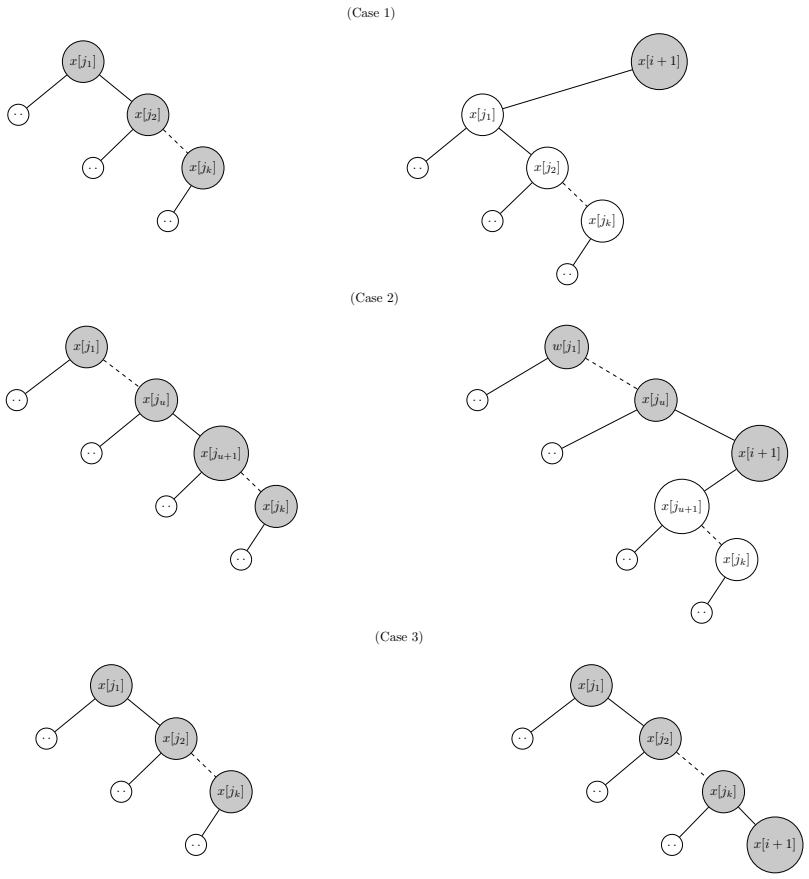


Figure 1. The three cases occurring when computing $\mathcal{CT}(x_{i+1})$ from $\mathcal{CT}(x_i)$. (Case 1) $x[i+1]$ is less than the current root of the tree and it is added as the new root; (Case 2): we have $x[j_u] < x[i+1] < x[j_{u+1}]$; (Case 3): we have $x[i+1]$ is greater than $x[i]$. In all cases $x[i+1]$ becomes the last node of the right path of the tree. Nodes belonging to the right path are filled in gray.

3. if $u = k$ then $x[i+1]$ is greater than $x[j_k]$, the rightmost character in the right path of x_i, so that $x[i+1]$ is added as the right child of $x[j_k]$ in $\mathcal{CT}(x_{i+1})$.

Example 2. Let $x = \langle 4, 22, 3, 5, 2, 23, 15, 1\rangle$ be a numeric sequence of length 8. Figure 3 shows the Cartesian trees computed by such incremental procedure.

Lemma 3 ([5]). *Given a numeric string x, of length m, the Cartesian tree of x can be computed in $O(m)$-time.*

Proof. In order to analyse the time complexity for the computation of the Cartesian tree of a string we refer to the algorithm whose pseudocode, presented in Figure 2, was described in [5].

The for cycle of line 4 is executed $m-1$ times. The while loop of line 7 consists of scanning upward the right path of the tree. Each iteration of this loop decreases the current length of the right path by one and the scanned node will not be scanned again thus the overall number of iterations of the while loop over all the iterations of the for loop is bounded by m. All the other operations can be done in constant time. Therefore the time complexity of the algorithm for building the Cartesian tree of a string of length m is $O(m)$.

```
BUILD-CARTESIAN-TREE(x, m)
 1  root ← NEW-NODE()
 2  ELEMENT(root) ← x[1]
 3  q ← root
 4  for i ← 2 to m do
        ▷ q is the last node of the right path
 5      p ← NEW-NODE()
 6      ELEMENT(p) ← x[i]
 7      while q ≠ nil and x[i] < ELEMENT(q) do
 8          q ← PARENT(q)
 9      if q = nil then
            ▷ Case 1
10          LEFT(p) ← root
11          PARENT(root) ← p
12          root ← p
13      else ▷ Cases 2 and 3
14          if RIGHT(q) ≠ nil then
15              PARENT(RIGHT(q)) ← p
16          LEFT(p) ← RIGHT(q)
17          RIGHT(q) ← p
18          PARENT(p) ← q
19      q ← p
20  return root
```

Figure 2. The iterative procedure BUILD-CARTESIAN-TREE for building the Cartesian tree of a string x of length m. A node of the Cartesian tree has 4 components: PARENT, ELEMENT, LEFT and RIGHT. The function NEW-NODE() creates a new node and initializes its 4 components to nil.

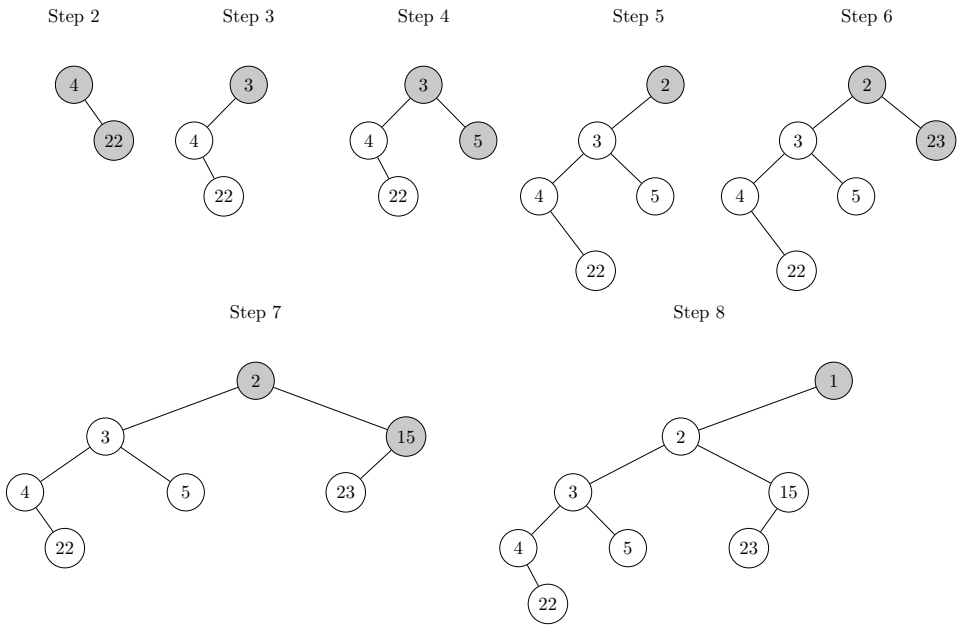


Figure 3. Different steps of the construction of $\mathcal{CT}(x)$ when $x = \langle 4, 22, 3, 5, 2, 23, 15, 1 \rangle$.

For the sake of completeness we point out that the Cartesian tree of a string x can be also computed by iterating over the elements of x and proceeding from right to left (instead of from left to right) by means of a symmetrical procedure.

For realizing the algorithm given in Figure 2 the right path can be implemented as a stack so that there is no need to have a link to its parent for each node of the tree.

Instead of building the Cartesian tree for every position in the text to solve Cartesian tree matching, Park et al. [8] introduced the following representation for a Cartesian tree.

Definition 4 (Parent-distance representation). *The parent-distance representation of a string $x[1\,..\,m]$ is a function PD_x, which is defined as follows:*

$$PD_x(i) = \begin{cases} i - \max_{1 \le j < i}\{j \mid x[j] \le x[i]\} & \text{if such } j \text{ exists} \\ 0 & \text{otherwise.} \end{cases}$$

Example 5. The following table gives the parent-distance representation for $x = \langle 4, 22, 3, 5, 2, 23, 15, 1\rangle$.

i	1	2	3	4	5	6	7	8
$x[i]$	4	22	3	5	2	23	15	1
$PD_x(i)$	0	1	0	1	0	1	2	0

Since the parent-distance representation has a one-to-one mapping to the Cartesian tree [8], it can replace the Cartesian tree without any loss of information. It can be computed and stored in a table in linear time and space using the algorithm given in Figure 4 (see [8]).

```
COMPUTE-PARENT-DISTANCE(x, m)
 1  ST ← empty stack
 2  for i ← 1 to m do
 3     while ST is not empty do
 4        (value, index) ← ST.top
 5        if value ≤ x[i] then
 6           break
 7        ST.pop
 8     if ST is empty then
 9        PD_x[i] ← 0
10     else PD_x[i] ← i − index
11     ST.push((x[i], i))
12  return PD_x
```

Figure 4. Computation of the parent-distance representation for a string x of length m.

4 Longest common Cartesian substrings: suffix tree based solution

The Cartesian suffix tree of a string has to be built on the parent-distance representation of the string. The parent-distance representation of a substring of x can be easily computed as follows (see [8]):

$$PD_{x[i..j]}[k] = \begin{cases} 0 & \text{if } PD_x[i+k-1] \ge k \\ PD_x[i+k-1] & \text{otherwise.} \end{cases}$$

This can be used for getting all the suffixes of the parent-distance representation for building its suffix tree. However classical linear time suffix tree construction algorithms cannot be used because the distinct right context property should hold in order to apply these algorithms, which means that the suffix link of every internal node should point to an explicit node. In other words if $lcp(x[i\,..\,m], x[j\,..\,m]) = \ell$ then $lcp(x[i{+}1\,..\,m], x[j{+}1\,..\,m]) = \ell{-}1$ for $1 \leq i, j \leq m$ where $lcp(u, v)$ is the length of the longest prefix common to two strings u and v. The Cartesian suffix tree does not have the distinct right context property meaning that if $lcp(PD_x[i\,..\,m], PD_x[j\,..\,m]) = \ell$ then $lcp(PD_x[i+1\,..\,m], PD_x[j+1\,..\,m])$ can be greater than $\ell - 1$.

Example 6. With $x = \langle 4, 22, 3, 5, 2, 23, 15, 1\rangle$,

$$lcp(PD_{x[5..8]}, PD_{x[6..8]}) = lcp(\langle 0, 1, 2, 0\rangle, \langle 0, 0, 0\rangle) = 1$$

and

$$lcp(PD_{x[6..8]}, PD_{x[7..8]}) = lcp(\langle 0, 0, 0\rangle, \langle 0, 0\rangle) = 2.$$

A linear time construction algorithm for suffix tree with missing suffix links was first given in [2]. It can be used for building Cartesian suffix trees. These Cartesian suffix trees can be used to compute longest common Cartesian substrings of two strings x and y: for instance, by building the generalized Cartesian suffix tree of PD_x and PD_y. Then the internal nodes with the largest string depth having leaves corresponding to both PD_x and PD_y identify longest common Cartesian substrings of x and y. This can be done during a traversal of the tree. Thus longest common Cartesian substrings of two strings can be computed in linear time and in linear space. The space overhead, in addition to the two strings and their parent-distance representation, is constituted by the generalized suffix tree.

5 Longest common Cartesian substrings: alternative solution

Let x and y be two strings of numbers of length m and n respectively. We are interested in finding the longest substrings of x and y having the same Cartesian tree. We will describe a solution based on dynamic programming. This solution also uses the parent-distance representation. We will show that if $x[i'\,..\,i-1]$ and $y[j'\,..\,j-1]$ are the longest suffixes of $x[1\,..\,i-1]$ and $y[1\,..\,j-1]$ having the same Cartesian tree then the longest suffixes of $x[1\,..\,i]$ and $y[1\,..\,j]$ having the same Cartesian tree can easily be computed. Let us first state that if two substrings have the same parent-distance so have their suffixes.

Fact 7 *If* $PD_{x[i'..i]} = PD_{y[j'..j]}$ *then* $PD_{x[i-\ell..i]} = PD_{y[j-\ell..j]}$ *for* $0 \leq \ell < i - i'$.

Proof. Let $0 \leq \ell < i - i'$ and $i - \ell - i' + 1 < k < i - i'$, then only two cases have to be considered:

1. $PD_{x[i'..i]}[k] = PD_{y[j'..j]}[k] < k - i + \ell$ then $PD_{x[i-\ell..i]}[k - i + \ell + i'] = PD_{x[i'..i]}[k] = PD_{y[j'..j]}[k] = PD_{y[j-\ell..j]}[k - i + \ell + i']$ (see Figure 5) or
2. $PD_{x[i'..i]}[k] = PD_{y[j'..j]}[k] \geq k{-}i{+}\ell$ then $PD_{x[i-\ell..i]}[k{-}i{+}\ell{+}i'] = 0 = PD_{y[j-\ell..j]}[k{-}i + \ell + i']$ (see Figure 6).

In both cases $PD_{x[i-\ell..i]}[p] = PD_{y[j-\ell..j]}[p]$ for $1 \leq p \leq \ell + 1$ thus $PD_{x[i-\ell..i]} = PD_{y[j-\ell..j]}$.

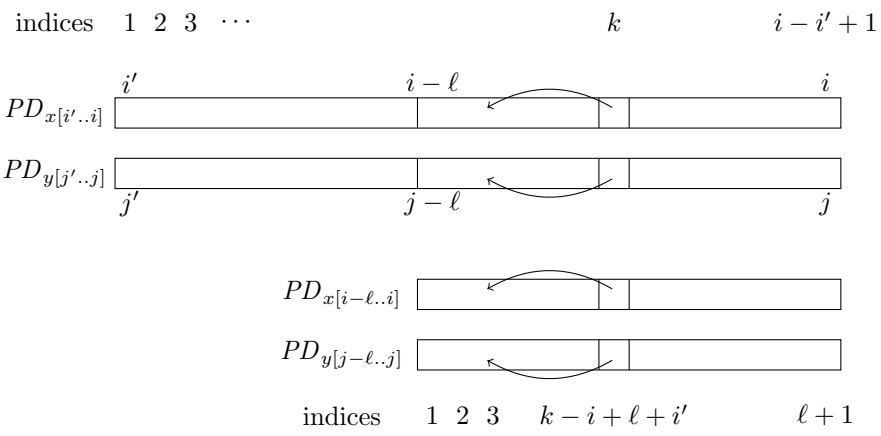


Figure 5. $PD_{x[i'..i]}[k] = PD_{y[j'..j]}[k] < k - i + \ell$

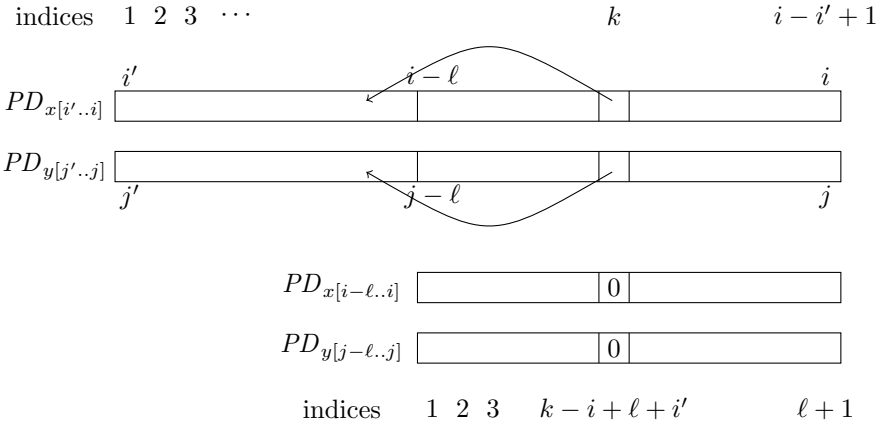


Figure 6. $PD_{x[i'..i]}[k] = PD_{y[j'..j]}[k] \geq k - i + \ell$

We can now state the next lemma.

Lemma 8. *Let $x[i'..i-1]$ and $y[j'..j-1]$ be the longest suffixes of $x[1..i-1]$ and $y[1..j-1]$ having the same Cartesian tree. Let $\ell_x = PD_{x[i'..i]}[i - i' + 1]$, $\ell_y = PD_{y[j'..j]}[j - j' + 1]$ and $\ell = \min\{\ell_x, \ell_y\}$.*

The longest suffixes of $x[1..i]$ and $y[1..j]$ having the same Cartesian tree are:

1. *$x[i'..i]$ and $y[j'..j]$ if $\ell_x = \ell_y$;*
2. *$x[i - \ell_y + 1..i]$ and $y[j - \ell_y + 1..j]$ if $\ell_x \neq \ell_y$ and $\ell_x = 0$;*
3. *$x[i - \ell_x + 1..i]$ and $y[j - \ell_x + 1..j]$ if $\ell_x \neq \ell_y$ and $\ell_y = 0$;*
4. *$x[i - \ell + 1..i]$ and $y[j - \ell + 1..j]$ if $\ell_x \neq \ell_y$ and $\ell_x \neq 0$ and $\ell_y \neq 0$.*

Proof. If $x[i'..i-1]$ and $y[j'..j-1]$ have the same Cartesian tree then $PD_{x[i'..i-1]} = PD_{y[j'..j-1]}$. We will detail the 4 cases:

1. If $\ell_x = \ell_y$ then $PD_{x[i'..i]} = PD_{y[j'..j]}$ and thus $x[i'..i]$ and $y[j'..j]$ have the same Cartesian tree. Thus $x[i'..i]$ and $y[j'..j]$ are the longest suffixes of $x[1..i]$ and $y[1..j]$ having the same Cartesian tree. Longer suffixes with the same Cartesian tree would contradict the maximality of the length of $x[i'..i-1]$ and $y[j'..j-1]$.

2. If $\ell_x \neq \ell_y$ and $\ell_x = 0$ then $PD_{y[k..j]}[j-k+1] = \ell_y \neq \ell_x$ for $j' \leq k \leq j - \ell_y$ and $PD_{y[j-\ell_y+1..j]}[\ell_y] = 0 = \ell_x$. Thus by Fact 7, $x[i-\ell_y+1\,..\,i]$ and $y[j-\ell_y+1\,..\,j]$ are the longest suffixes of $x[1\,..\,i]$ and $y[1\,..\,j]$ having the same Cartesian tree.
3. If $\ell_x \neq \ell_y$ and $\ell_y = 0$ then $PD_{x[k..i]}[i-k+1] = \ell_x \neq \ell_y$ for $i' \leq k \leq i - \ell_x$ and $PD_{x[i-\ell_x+1..i]}[\ell_x] = 0 = \ell_y$. Thus by Fact 7, $x[i-\ell_x+1\,..\,i]$ and $y[j-\ell_x+1\,..\,j]$ are the longest suffixes of $x[1\,..\,i]$ and $y[1\,..\,j]$ having the same Cartesian tree.
4. If $\ell_x \neq \ell_y$ and $\ell_x \neq 0$ and $\ell_y \neq 0$ then $PD_{x[k..i]}[i-k+1] \neq PD_{y[k'..j]}[j-k'+1]$ for $i' \leq k \leq i - \ell$ and for $j' \leq k' \leq j - \ell$ and $PD_{x[i-\ell+1..i]}[\ell] = 0 = PD_{y[j-\ell+1..j]}[\ell]$. Thus by Fact 7, $x[i-\ell+1\,..\,i]$ and $y[j-\ell+1\,..\,j]$ are the longest suffixes of $x[1\,..\,i]$ and $y[1\,..\,j]$ having the same Cartesian tree.

A diagonal d corresponds to a pair of factors $x[i\,..\,s]$ and $y[j\,..\,t]$ such $1 \leq i \leq s \leq m$, $1 \leq j \leq t \leq n$, $s - i = t - j$ and $d = j - i$. Since factors of length 1 always constitute common Cartesian substrings between two strings, the algorithm Compute-Longest-Cartesian-Substring given in Figure 7 processes diagonals from $-m+2$ to $n-2$. For each diagonals it uses 4 indices i, i', j and j' to compare $x[i'\,..\,i]$ and $y[j'\,..\,j]$ starting with the first factors of length 2 of the current diagonal. It updates indices i, i', j and j' according to Lemma 8. It only computes the length ℓ of the longest common Cartesian substrings of x and y but could easily be computed to report two positions i in x and j in y such that $x[i\,..\,i+\ell-1]$ and $y[j\,..\,j+\ell-1]$ have the same Cartesian tree with the same complexities.

Theorem 9. *Given two strings x and y of numbers of length m and n respectively, the longest substrings of x and y having the same Cartesian tree can be computed in time $O(mn)$ and in space $O(m+n)$.*

Proof. Given x and y, the parent-distance representations PD_x and PD_y can be computed in space and time $O(m)$ and $O(n)$ respectively. Then the results of Lemma 8 can be applied on any pair of ending positions of substrings of x and y. There are $O(mn)$ such pairs and the computation for one pair takes constant time if the result for the correct previous pair is available. By performing the computation diagonal-wise the result follows.

Example 10. $x = \langle 70, 84, 63, 74, 86, 97\rangle$ and $y = \langle 50, 83, 76, 39, 90, 67, 1, 6\rangle$. Then $PD_x = \langle 0, 1, 0, 1, 1, 1\rangle$ and $PD_y = \langle 0, 1, 2, 0, 1, 2, 0, 1\rangle$. Let us look at starting positions $i' = 3$ in x and $j' = 4$ in y:

- $PD_{x[3..3]}[1] = 0$ and $PD_{y[4..4]}[1] = 0$
- $PD_{x[3..4]}[2] = 1$ and $PD_{y[4..5]}[2] = 1$
- $PD_{x[3..5]}[3] = 1$ and $PD_{y[4..6]}[3] = 2$ then i' becomes 5 and j' becomes 6
- $PD_{x[5..6]}[2] = 1$ and $PD_{y[6..7]}[2] = 0$ then i' becomes 6 and j' becomes 7

thus the longest Cartesian substring of $x[3\,..\,6]$ and $y[4\,..\,7]$ has length 2.

6 Experiments

The two methods have been implemented in C programming language. The experiments were performed on a computer running MacOS 10.12.6 with an Intel Core i5 1.3 GHz processor and 4GB RAM. We want to compute the length of the longest common Cartesian substring of x of length m and of y of length n. Assume w.l.o.g. that $m < n$.

```
COMPUTE-LONGEST-CARTESIAN-SUBSTRING(x, m, y, n)
 1  PD_x ← COMPUTE-PARENT-DISTANCE(x, m)
 2  PD_y ← COMPUTE-PARENT-DISTANCE(y, n)
 3  maxlength ← 1
 4  for d ← −m + 2 to n − 2 do
      ▷ Initialization of i, j, i′, j′
 5    (i, j) ← (2, 2)
 6    if d < 0 then
 7      i ← −d + 2
 8    else if d > 0 then
 9          j ← d + 2
10    (i′, j′) ← (i − 1, j − 1)
      ▷ Processing diagonal d
11    while i ≤ m and j ≤ n do
12      (ℓ_x, ℓ_y) ← (PD_{x[i′..i]}[i − i′ + 1], PD_{y[j′..j]}[j − j′ + 1])
        ▷ Application of Lemma 8
13      if ℓ_x ≠ ℓ_y then
14        if ℓ_x = 0 then
15          ℓ ← ℓ_y
16        else if ℓ_y = 0 then
17              ℓ ← ℓ_x
18          else ℓ ← min{ℓ_x, ℓ_y}
19        (i′, j′) ← (i − ℓ + 1, j − ℓ + 1)
20      else ▷ maxlength can only increase when ℓ_x = ℓ_y
21        maxlength ← max{maxlength, i − i′ + 1}
22      (i, j) ← (i + 1, j + 1)
23  return maxlength
```

Figure 7. Computation of the length of the longest Cartesian substrings of x of length m and y of length n.

The suffix tree with missing suffix links from [2] has been implemented naively without back propagation and with a simple hashing function. Our implementation is not linear in the worst case but we argue that for short strings it is faster than the linear method which is quite intricate and will lead to an increase in runtime for such short strings. The method for finding the longest Cartesian substring between two strings is then the following: the parent distance representation PD_x and PD_y are computed. Then we construct $PD_{yx} = PD_y\$_1 PD_x\$_2$ where $\$_1$ and $\$_2$ are terminators that do not occur in PD_x and PD_y. Then the suffix tree of PD_{yx} is build for the part corresponding of the longest string y. For the remaining part the suffix tree is scanned to determine the fork where to insert the tail of the current suffix. The string depth of this fork is used to compute the length of the longest common Cartesian substring. The tail is then not inserted since it is not necessary for our purposes and will only lead to a loss of time.

For our alternative solution, main long diagonals are processed first. Shortest diagonals corresponding to prefixes and suffixes of y are processed in a second time. During the computation of a diagonal, when the length of the remaining part of the diagonal is too short and will not possibly contribute to a longest common Cartesian substring, we processed to the next diagonals.

6.1 Random data

We built random strings of integers with 4 different alphabets $[0; 10[$, $[0; 100[$, $[0; 1000[$ and $[0; 10000[$. Then we consider 3 different values for n: 50, 125 and 200. For each value of n we consider 4 values of m: $n/10$, $n/5$, $n/2.5$ and $n/1.25$.

Figure 8 to 10 show the running times of the two solutions: σ denotes the alphabet size, alt the times for the alternative solution and ST the times for the suffix tree based solution. Times are expressed in μseconds.

σ	m	5	10	20	40
10	alt	4283	6189	13621	24711
	ST	36120	33230	46744	56796
100	alt	4963	7210	13702	25644
	ST	40437	39994	46495	58973
1000	alt	4779	7618	14708	25621
	ST	39886	42869	50787	58650
10000	alt	4280	7083	14636	25825
	ST	36074	40522	51166	58184

Figure 8. Experiments with random data for $n = 50$

σ	m	12	24	48	96
10	alt	16589	34500	71790	150029
	ST	98865	95663	108624	132481
100	alt	15577	37178	77217	139539
	ST	96253	106864	121404	128341
1000	alt	14420	35633	72197	141706
	ST	85163	101248	111795	126697
10000	alt	15112	34439	74415	140266
	ST	94438	95248	114660	128906

Figure 9. Experiments with random data for $n = 125$

σ	m	20	40	80	160
10	alt	38677	91698	203764	436527
	ST	127756	150781	181238	261578
100	alt	50687	114319	204344	444743
	ST	197489	209726	196585	280190
1000	alt	46502	104542	190889	393760
	ST	172624	183040	174690	216348
10000	alt	43269	97887	198550	414818
	ST	150034	160674	181007	236281

Figure 10. Experiments with random data for $n = 200$

For $n = 50$ our alternative solution is always fastest than the suffix tree solution. For $n = 125$ our alternative solution is fastest than the suffix tree solution for values of m up to $n/2$. For $n = 200$ our alternative solution is fastest than the suffix tree solution for values of m up to $n/4$.

6.2 Real data

We use data from the COVID-19 pandemic.[1] We extracted the numbers of cases and numbers of deaths for the 15 most infected countries at that time. Data were given in reverse chronological order. We trimmed the data for the tailing runs of 0s and 1s at the end.

Table 1 gives the country numbers and the number of days for cases and deaths for each country.

♯	Country	Cases	Deaths
1	China	102	98
2	France	62	53
3	Germany	62	49
4	Iran	68	68
5	Italy	66	65
6	Korea	72	67
7	Spain	63	54
8	Turkey	45	40
9	UK	105	98
10	USA	67	58
11	Russia	47	32
12	Brazil	54	41
13	Canada	63	42
14	Belgium	57	47
15	Netherlands	60	50

Table 1. Country number, number of cases and deaths considered.

Then we computed the length of the longest common Cartesian substrings for each pair of countries.

	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	11	8	11	11	10	10	13	9	13	14	9	9	10	11
2		9	12	11	8	9	16	11	11	10	9	11	7	12
3			8	11	8	11	9	11	9	8	12	8	8	11
4				11	10	9	12	8	14	12	8	9	8	10
5					9	10	10	13	11	11	10	12	9	10
6						9	7	9	8	8	8	9	7	8
7							9	10	9	9	9	9	11	11
8								9	12	15	8	9	7	12
9									10	10	9	14	12	11
10										14	9	10	9	10
11											9	9	8	10
12												11	10	8
13													9	8
14														9

Figure 11. Length of the longest common Cartesian substring between countries for the number of cases of the COVID-19.

Figure 11 shows the results for the number of cases. These results have been computed in 9381 μs with the Suffix Tree method and in 5904 μs with our alternative method.

[1] We downloaded data from `https://www.ecdc.europa.eu`. on 27th April 2020

	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	9	8	14	7	10	10	10	10	8	10	8	9	13	10
2		10	10	9	7	7	10	7	9	9	9	9	12	14
3			9	9	8	11	9	11	10	8	7	8	10	7
4				8	9	10	13	11	9	9	8	9	10	8
5					8	8	7	10	14	10	11	10	8	11
6						8	7	9	7	9	8	7	8	8
7							10	8	9	9	10	9	9	8
8								7	9	10	7	7	11	8
9									8	9	10	10	9	9
10										8	11	8	11	9
11											9	8	10	9
12												7	7	9
13													9	10
14														8

Figure 12. Length of the longest common Cartesian substring between countries for the number of deaths of the COVID-19.

Figure 12 shows the results for the number of deaths. These results have been computed in 8727 μs with the Suffix Tree method and in 4758 μs with our alternative method.

Again, these results show that for such short strings our alternative solution is faster than the suffix tree based method.

In our experiments, our alternative solution is faster than the suffix tree solution in 38 cases out of 50 cases (on both random and real data); when it is faster, our alternative solution is 3.81 times faster on average (up to 8.43 times for the maximum), and when it is slower, it is 1.3 times slower on average than the suffix tree solution (up to 1.82 for the maximum).

7 Conclusion

In this article we presented a classical suffix tree based solution for computing the longest Cartesian substrings between two strings. This solution is based on the parent-distance representation of the two strings and runs in linear time and linear extra-space in addition to the two strings and their parent-distance representation. Then we proposed an alternative solution based on dynamic programming that runs in quadratic time in the worst case and in constant extra-space in addition to the two strings and their parent-distance representation. We presented experiments that show that our alternative solution runs faster for short strings than the suffix tree based solution. Further works would include the search for the longest approximate common Cartesian substrings between two strings but the notion of approximation in this context has to be defined.

References

1. M. Bataa, S. G. Park, A. Amir, G. M. Landau, and K. Park: *Finding periods in Cartesian tree matching*, in Combinatorial Algorithms - 30th International Workshop, IWOCA 2019, Pisa, Italy, July 23-25, 2019, Proceedings, C. J. Colbourn, R. Grossi, and N. Pisanti, eds., vol. 11638 of Lecture Notes in Computer Science, Springer, 2019, pp. 70–84.
2. R. Cole and R. Hariharan: *Faster suffix tree construction with missing suffix links.* SIAM J. Comput., 33(1) 2003, pp. 26–42.

3. M. Crochemore and L. M. S. Russo: *Cartesian and Lyndon trees.* Theor. Comput. Sci., 806 2020, pp. 1–9.
4. E. D. Demaine, G. M. Landau, and O. Weimann: *On Cartesian trees and Range Minimum Queries.* Algorithmica, 68(3) 2014, pp. 610–625.
5. H. N. Gabow, J. L. Bentley, and R. E. Tarjan: *Scaling and related techniques for geometry problems*, in Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA, R. A. DeMillo, ed., ACM, 1984, pp. 135–143.
6. G. Gu, S. Song, S. Faro, T. Lecroq, and K. Park: *Fast multiple pattern Cartesian tree matching*, in WALCOM: Algorithms and Computation - 14th International Conference, WALCOM 2020, Singapore, March 31 - April 2, 2020, Proceedings, M. S. Rahman, K. Sadakane, and W. Sung, eds., vol. 12049 of Lecture Notes in Computer Science, Springer, 2020, pp. 107–119.
7. C. Hohlweg and C. Reutenauer: *Lyndon words, permutations and trees.* Theor. Comput. Sci., 307(1) 2003, pp. 173–178.
8. S. G. Park, A. Amir, G. M. Landau, and K. Park: *Cartesian tree matching and indexing*, in 30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, June 18-20, 2019, Pisa, Italy., N. Pisanti and S. P. Pissis, eds., vol. 128 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019, pp. 16:1–16:14.
9. J. Shun and G. E. Blelloch: *A simple parallel Cartesian tree algorithm and its application to parallel suffix tree construction.* ACM Trans. Parallel Comput., 1(1) 2014, pp. 8:1–8:20.
10. S. Song, C. Ryu, S. Faro, T. Lecroq, and K. Park: *Fast Cartesian tree matching*, in String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings, N. R. Brisaboa and S. J. Puglisi, eds., vol. 11811 of Lecture Notes in Computer Science, Springer, 2019, pp. 124–137.
11. J. Vuillemin: *A unifying look at data structures.* Commun. ACM, 23(4) 1980, pp. 229–239.

Forward Linearised Tree Pattern Matching Using Tree Pattern Border Array

Jan Trávníček, Robin Obůrka, Tomáš Pecka*, and Jan Janoušek**

Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Praha 6
Czech Republic
{Jan.Travnicek, oburkrob, Tomas.Pecka, Jan.Janousek}@fit.cvut.cz

Abstract. We define a tree pattern border array as a property of linearised trees analogous to border arrays from the string domain. We use it to define a new forward tree pattern matching algorithm for ordered trees, which finds all occurrences of a single given linearised tree pattern in a linearised input tree. As with the classical Morris-Pratt algorithm, the tree pattern border array is used to determine shift lengths in the searching phase of the tree pattern matching algorithm. We compare the new algorithm with the best performing previously existing algorithms based on backward linearised tree pattern matching algorithms, (non-)linearised tree pattern matching algorithms using finite tree automata or stringpath matchers. We show that the presented algorithm outperforms these for single tree pattern matching.

Keywords: tree processing, tree linearisation, Morris-Pratt algorithm

1 Introduction

Trees are one of the fundamental data structures used in Computer Science and the theory of formal tree languages has been extensively studied and developed since the 1960s [9,11]. Tree pattern matching on node-labeled trees is an important algorithmic problem with applications in many tasks such as compiler code selection, interpretation of nonprocedural languages, implementation of rewriting systems, or markup languages processing. Trees can be represented as a string by various linearisations [16]. Such a linear notation can be obtained by a corresponding tree traversal. Moreover, every sequential algorithm on a tree traverses its nodes in a sequential order, which corresponds to some linear notation. Such a linear representation need not be built explicitly.

Tree patterns are trees whose leaves can be labelled by a special wildcard, the nullary symbol S, which serves as a placeholder for any subtree. Since the linear notation of a subtree of a tree is a substring of the linear notation of that tree, the subtree matching and tree pattern matching problems are in many ways similar to the string pattern matching problem. We note that the tree pattern matching problem is more complex than the string matching one because there can be at most $n(n-1)/2$ distinct substrings of a string of size n, whereas there can be at most $2^{n-1}+n$ distinct tree patterns which match a tree of size n [13].

* This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS20/208/OHK3/3T/18.

** The authors acknowledge the support of the OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics".

Jan Trávníček, Robin Obůrka, Tomáš Pecka, Jan Janoušek: Forward Linearised Tree Pattern Matching Using Tree Pattern Border Array, pp. 61–73.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9

Border array and borders in general are one of well-studied string properties used in various efficient single pattern string pattern matching algorithms (Morris-Pratt and Knuth-Morris-Pratt algorithms, etc.) [17,15,4], and multi-pattern ones (Aho-Corasick) [1].

For unrestricted tree pattern sets, among the fastest pattern matching algorithms in practice are algorithms based on deterministic frontier-to-root (bottom-up) tree automata (DFRTAs) [6,8,12] and Hoffmann-O'Donnell-style stringpath matchers [2,12]. The latter uses Aho-Corasick pattern matching algorithm, but processes the tree in its natural representation.

Tree pattern matching algorithms processing a linearised representation of a tree exist as well. They either use pushdown automata [10] or, in the case of single pattern matching, adapt ideas known from backward string pattern matching [20]. However, no existing tree pattern matching algorithm uses the border array constructed for a linearised tree pattern directly on linearised trees.

The best performing algorithms using deterministic tree automata or deterministic pushdown automata generally run in $\Theta(n+occ)$ time, where n is the size of the subject tree [8]. On the other hand, the backward tree pattern matching algorithm require $\Omega(n/m + occ)$ and $O(m \cdot n + occ)$ time, where m is the size of the tree pattern [20].

While modifying forward string pattern matching to forward subtree matching (searching for occurrences of given subtrees) is straightforward, this is not the case for forward tree pattern matching, where complications arise due to the use of nullary symbol S and matched subtrees being possibly recursively nested.

In this paper, a definition of tree pattern border array is presented. The size of the tree pattern border array table is linear with the size of the linearised pattern. The tree pattern border array is later used in the design of a new forward tree pattern matching algorithm. The presented forward tree pattern matching algorithm is a modification of the Morris-Pratt algorithm from the string domain and even though it does not keep the linear complexity of the searching phase with respect to the size of the subject tree n, it often performs better in practice than sublinear backward tree pattern matching algorithm [20]. Even though the Knuth-Morris-Pratt algorithm is a straight-forward extension of the Morris-Pratt algorithm, this is not the case in trees and therefore the presented algorithm is based on the Morris-Pratt algorithm. Our experimental results show that the presented algorithm outperforms even the aforementioned DFRTAs and Aho-Corasick stringpath matchers in single-pattern matching case.

The paper is organised as follows: Section 2 recalls basic definitions and properties of trees and the Morris-Pratt algorithm. Section 3 defines the tree pattern border array and presents the new forward linearised tree pattern matching algorithm based on the Morris-Pratt algorithm. Section 4 compares the results with other state-of-the-art algorithms. Some concluding remarks are presented in Section 5.

2 Basic notions

An *alphabet* is a finite nonempty set of *symbols*. In a *ranked* alphabet $\mathcal{A}$, each symbol ℓ is assigned a nonnegative *arity* or *rank* denoted by $Arity(\ell)$. The set of symbols of arity p is denoted by $\mathcal{A}_p$. Elements of arity $0, 1, 2, \ldots, p$ are called nullary (constants), unary, binary, ..., p-ary symbols, respectively. We assume that $\mathcal{A}$ contains at least one constant. In the examples, we use numbers at the end of identifiers for a short declaration of symbols with arity. For instance, $a2$ is a short declaration of a binary symbol a.

A *string* s is a sequence of n symbols $s_1 s_2 s_3 \cdots s_n$ from a given alphabet, where n is the size of the string. A sequence of zero symbols is called the empty string. The empty string is denoted by symbol ε. A *s[i .. j]* is a substring (factor) $s_i \cdots s_j$ of s, note that ε substring of s is obtained by $s[i..i-1]$. A prefix and a suffix of a string s of length n is a substring $s[1..j]$ and $s[i..n]$, respectively, where $1 \leq j \leq n$ and $1 \leq i \leq n$. A proper prefix and a proper suffix of s is a prefix and a suffix, respectively, which is not equal to s.

Based on concepts and notations from graph theory [3], a *rooted tree* t is a weakly connected acyclic directed graph $t = (V, E)$ with a special node $r \in V$, called the *root*, such that r has in-degree 0, all other nodes of t have in-degree 1, and there is just one path from the root r to every $f \in V$ and $f \neq r$, where a path from a node f_0 to a node f_n is a sequence of nodes $(f_0, f_1, ..., f_n)$ for $n > 0$ and $(f_i, f_{i+1}) \in E$ for each $0 \leq i < n$. Nodes of a rooted tree with out-degree 0 are called *leaves*.

A node g is a *direct descendant* of node f if a pair $(f, g) \in E$ and *descendant* of node f if $(f, f_1, f_2, ..., f_n, g)$ for $n \geq 0$ is a path in t.

A tree is an *ordered, ranked and labelled* rooted tree with nodes labelled by symbols from a ranked alphabet satisfying that the out-degree of a node f labelled by symbol $\ell \in \mathcal{A}$ equals $Arity(\ell)$ and with the direct descendants $g_1, g_2, \ldots, g_n$ of a node f ordered.

A *subtree* (a complete subtree) of tree $t = (V, E)$ is any tree $t' = (V', E')$ such that:

1. V' is an nonempty subset of V,
2. $E' = (V' \times V') \cap E$, and
3. no node of $V \setminus V'$ is a descendant of a node in V'.

The *prefix notation* $pref(t)$ of a tree t is defined as follows:

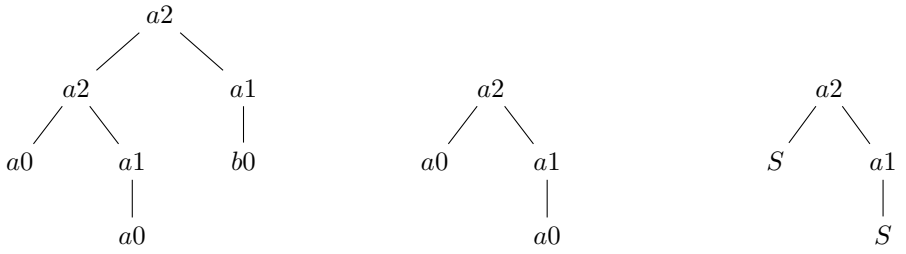
1. $pref(\ell) = \ell 0$ if ℓ is a leaf,
2. $pref(t) = \ell n\ pref(t_1)\ pref(t_2) \cdots pref(t_n)$, where ℓ is the label of the root of tree t, $n = Arity(\ell)$ and $t_1, t_2, \ldots, t_n$ are direct descendants of the root of t.

Let $s = s_1 s_2 \cdots s_n$, $n \geq 1$, be a string over a ranked alphabet $\mathcal{A}$. Then, the *arity checksum* $ac(s) = \sum_{i=1}^{n} Arity(s_i) - n + 1$. Let $pref(t)$ and s of size n be a tree t in prefix notation and a substring of $pref(t)$, respectively. Then, s is the prefix notation of a subtree of t, if and only if $ac(s) = 0$, and $ac(s[1..j]) \geq 1$ for all $1 \leq j < n$ [16].

Example 1. Consider a tree t_{1r} over a ranked alphabet $\mathcal{A} = \{a2, a1, a0\}$, $pref(t_{1r}) =$ $a2\ a2\ a0\ a1\ a0\ a1\ a0$. Trees can be represented graphically, as is done for tree t_{1r} in Figure 1a.

To define a *tree pattern*, we use a special wildcard symbol $S \notin \mathcal{A}$, $Arity(S) = 0$, which serves as a placeholder for any subtree. A tree pattern is defined as a labelled ordered tree over ranked alphabet $\mathcal{A} \cup \{S\}$. We will assume that the tree pattern contains at least one node labelled by a symbol from $\mathcal{A}$. Note that the wildcard symbol can only label leaves of tree pattern.

A tree pattern p with $k \geq 0$ occurrences of the symbol S *matches* a subject tree t at node n if there exist subtrees $t_1, t_2, \ldots, t_k$ (not necessarily the same) of t such that the tree p', obtained from p by substituting the subtree t_i for the i-th occurrence of S in p, $i = 1, 2, \ldots, k$, is equal to the subtree of t rooted at n.



(a) Tree t_{1r} over a ranked alphabet from Example 1.

(b) Subtree (tree) p_{1r} from Example 2.

(c) Tree pattern p_{2r} from Example 2.

Figure 1. Trees and tree patterns over a ranked alphabet from Example 1 and Example 2.

Example 2. Consider a tree t_{1r} from Example 1, which is illustrated in Figure 1a. Consider a subtree p_{1r} over ranked alphabet $\mathcal{A}$, $pref(p_{1r}) = a2\ a0\ a1\ a0$ and a tree pattern p_{2r} over ranked alphabet $\mathcal{A} \cup \{S\}$, $pref(p_{2r}) = a2\ S\ a1\ S$, which are illustrated in Figure 1b and Figure 1c. Tree pattern p_{1r} occurs once in t_{1r} — match is at node 2 of t_{1r}. Tree pattern p_{2r} occurs twice in t_{1r}, it matches at nodes 1 and 2 of t_{1r}.

2.1 Forward string pattern matching algorithm

A classical representant of a forward string pattern matching algorithm is the Morris-Pratt algorithm [17,15]. The algorithm's execution splits into preprocessing and searching phases.

A precomputed table to determine the length of shift and also to know the number of symbols not required to be matched again in the following match attempt is constructed during the preprocessing phase. The table used in the algorithm is the border array [18,5]. The border array for a string of length m can be constructed in $O(m)$ time.

Note the border array is defined as in [5] without explicitly defining borders to ease transition from strings to linearised tree patterns where concepts of prefix and suffix do not apply.

Definition 3 (border array ([5]) $\mathcal{B}(s)$). *Let s be a string of length n. The border array $\mathcal{B}(s)$ is defined for each index $1 \leq i \leq n$ such that $\mathcal{B}(s)[1] = 0$ and otherwise $\mathcal{B}(s)[i] = max(\{0\} \cup \{k : s[1..k] = s[i-k+1..i] \wedge k \geq 1 \wedge i-k+1 > 1\})$.*

From the definition, the substrings $s[1..k]$ and $s[i-k+1..i]$ are referred to as borders.

Example 4. Consider a string $s = ababc$ of length 5. The border array $\mathcal{B}(s) = 0, 0, 1, 2, 0$.

The algorithm locates all occurrences of a pattern in a text in a searching phase. Note that the presented algorithms deviates from the classical presentation of the Morris-Pratt algorithm to simplify the transition from its string version to a tree version.

The searching phase of the Morris-Pratt algorithm has time complexity $O(m+n)$. The algorithm makes at most $2n-1$ comparisons during the searching phase [17,15]. Line 10 of Algorithm 1 represents the shift and line 11 of Algorithm 1 represents a carry of information of how many symbols do not need to be matched again.

Algorithm 1: Morris-Pratt matching function.

Input: The subject string s of size n, the pattern string p of size m, the border array table $\mathcal{B}(p)$
Result: A list of matches.

```
1  begin
2      i := 0
3      j := 1
4      while i ≤ n − m do
5          while j ≤ m and s[i + j] = p[j] do
6              j += 1
7          end
8          if j > m then yield i + 1
9          if j ≠ 1 then
10             i += j − B(p)[j − 1] − 1
11             j := B(p)[j − 1] + 1
12         else
13             i += 1
14         end
15     end
16 end
```

3 Forward Linearised Tree Pattern Matching

The pattern occurrences in linear notation are represented by substrings of trees in the linear notation. They can contain "gaps" given by a special wildcard symbol S, which serves as a placeholder for any subtree.

The string pattern matching algorithm must be modified to handle these gaps. The wildcard symbol S represents any subtree. Moreover, matched subtrees may be possibly nested. The wildcard symbol S therefore needs special care.

In order to handle these gaps a *Subtree jump table* structure is defined. The structure was introduced in [14].

Definition 5 (subtree jump table for prefix notation $sjt(pref(t))$). *Let t and $pref(t) = \ell_1\ell_2\cdots\ell_n$, $n \geq 1$, be a tree and its prefix notation, respectively. A* subtree jump table for prefix notation *$sjt(pref(t))$ is a mapping from a set $\{1..n\}$ into a set $\{2..n+1\}$. If $\ell_i\ell_{i+1}\cdots\ell_{j-1}$ is the prefix notation of a subtree of tree t, then $sjt(pref(t))[i] = j$, $1 \leq i < j \leq n+1$.*

Note that the definition of subtree jump table for prefix notation is applicable to tree patterns without changes as well.

Informally, the subtree jump table contains an entry for each subtree r of tree t. The entry for subtree r is located at the position of its root in the prefix notation $pref(t)$ of the tree t. The entry stores an index into string $pref(t)$ to a symbol that is one after the last of the subtree r, i.e., position of the root of r plus the length of $pref(r)$. This structure has the same size as the prefix notation of the tree t. The construction time is $O(n)$ where n is the length of $pref(t)$ [14].

3.1 Linearised tree border

The Morris-Pratt algorithm uses shift heuristics based on borders. One needs to define a tree pattern border array in order to obtain similar shift heuristics in the Morris-Pratt algorithm modification for trees.

In order to define the tree pattern border array, first, let us define equivalence of linear representations of a tree pattern and its factor.

Definition 6 (matches relation *s matches r*)**.** *Let S be a wildcard symbol representing a complete subtree in prefix ranked notation of trees. Two strings s and r are in relation* matches *if:*

$$
\begin{array}{lll}
s = \ell s' & r = \ell r' & \text{and } s' \text{ matches } r' \\
 & & \text{and } \ell \in \mathcal{A}, \\
s = Ss' & r = Sr' & \text{and } s' \text{ matches } r', \\
s = \ell_1 \cdots \ell_m s' & r = Sr' & \text{and } ac(\ell_1 \cdots \ell_m) = 0 \\
 & & \text{and } \forall k, 1 \leq k < m, ac(\ell_1 \cdots \ell_k) \geq 1 \\
 & & \text{and } s' \text{ matches } r', \\
s = Ss' & r = \ell_1 \cdots \ell_m r' & \text{and } ac(\ell_1 \cdots \ell_m) = 0 \\
 & & \text{and } \forall k, 1 \leq k < m, ac(\ell_1 \cdots \ell_k) \geq 1 \\
 & & \text{and } s' \text{ matches } r', \\
s = Ss' & r = \ell_1 \cdots \ell_m & \text{and } \forall k, 1 \leq k \leq m, ac(\ell_1 \cdots \ell_k) \geq 1, \\
s = \varepsilon \text{ or } r = \varepsilon & &
\end{array}
$$

The two strings s and r are in relation matches if symbols of strings r and s compare on corresponding positions, wildcards in string r correspond to complete subtrees in string s, and wildcards in string s correspond to complete subtrees within string r and possibly incomplete subtree at the end of string r. Note that the corresponding symbols may not be on the same indexes in strings s and r as a subtree corresponding to wildcard S may be longer than a single symbol.

Informally, the two strings s and r representing prefixes of prefix notation of a tree pattern are in relation matches if the corresponding tree pattern subgraphs structurally and symbol-wise align.

Definition 7 (tree pattern border array $\mathcal{B}(pref(p))$). *Let $pref(p)$ be a tree pattern in a prefix notation of length n. The $\mathcal{B}(pref(p))$ is defined for each index $1 \leq i \leq n$ such that $\mathcal{B}(pref(p))[1] = 0$ and otherwise $\mathcal{B}(pref(p))[i] = max(\{0\} \cup \{k : pref(p) \text{ matches } pref(p)[i-k+1..i] \wedge k \geq 1 \wedge i-k+1 > 1\})$.*

Notice that the definition of tree pattern border array is different from the definition of the border array for strings in the use of the matches relation and in use of the complete linear representation of tree pattern as its left argument.

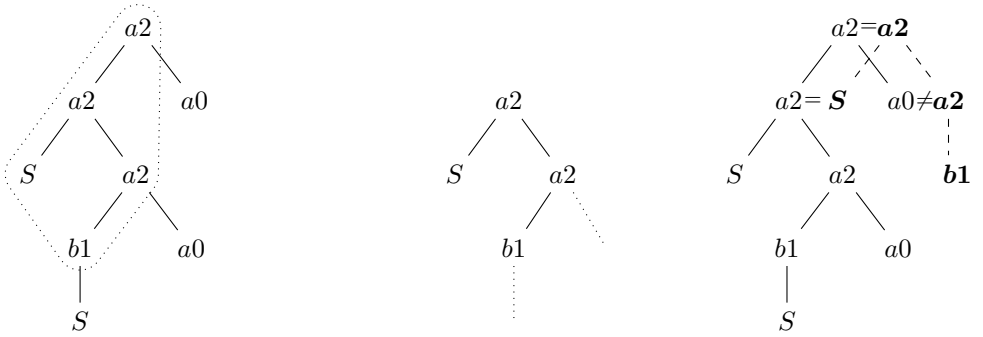
	1	2	3	4	5	6	7	8			
pref(p)	$a2$	$a2$	S	$a2$	$b1$	S	$a0$	$a0$			
pref(p)[2..5]	$a2$	$\vdash$		S			$\dashv$	$a2$			mismatch at position 8
pref(p)[3..5]	$\vdash$			S				$\dashv$	$a2$	$b1$	match
pref(p)[4..5]	$a2$	$b1$									mismatch at position 2
pref(p)[5..5]	$b1$										mismatch at position 1
pref(p)[6..5]											match because pref(p)[6..5] = ε

Table 1. Trace of naive computation of $pref(p)$ *matches* $pref(p)[j+1..5]$ for $1 \leq j \leq 5$ and $pref(p) = a2\ a2\ S\ a2\ b1\ S\ a0\ a0$.

Example 8. Let $pref(p) = a2\ a2\ S\ a2\ b1\ S\ a0\ a0$ be a prefix notation of a tree pattern p. In order to compute $\mathcal{B}(pref(p))[5]$ according to Definition 7 the computation of $pref(p)$ *matches* $pref(p)[j+1..5]$ for $1 \leq j \leq 5$ is necessary.

The minimal j for which $pref(p)$ *matches* $pref(p)[j+1..5]$ is 2, therefore, the $\mathcal{B}(pref(p))[5] = 5-2 = 3$. The process of computation of relation matches is depicted in Table 1.

The visualisation of the alignment of $pref(p)$ and $pref(p)[2..5]$, internally computed by the *matches* relation is depicted in Figure 2 on non-linearised tree pattern p and tree pattern p subgraph corresponding to $pref(p)[2..5]$. The other alignment visualisations can be depicted in a similar manner.



(a) Tree pattern p and the subgraph of p corresponding to the prefix of $pref(p)$ relevant to the computation of relation *matches*.

(b) The subgraph of p corresponding to $pref(p)[2..5]$.

(c) Visualisation of the alignment.

Figure 2. Visualisation of a single alignment of p and subgraph corresponding to $pref(p)[2..5]$ in computation of $pref(p)$ *matches* $pref(p)[j+1..5]$) for $1 \leq j \leq 5$.

The $\mathcal{B}(pref(p))$ can naively be implemented using Definition 7. The relation *matches* is easy to implement using iteration with the help of $sjt(pref(p))$. The naive computation requires $O(m^3)$ time, where m is the length of $pref(p)$.

The computation of $\mathcal{B}(pref(p))$ can also be done in quadratic time with respect to the length of $pref(p)$. All the factors beginning at *offset* are tested in one iteration through the pattern $pref(p)$ to determine the result of $pref(p)$ *matches* $pref(p)[offset+1..i]$ $\forall\, 1 \leq offset \leq i \leq m$. This approach avoids repeating some comparisons. One iteration through $pref(p)$ takes $O(m)$ time and has to be repeated $m-1$ times for different values of *offset*. The procedure of creating $\mathcal{B}(pref(p))$ is formalised in Algorithm 2.

3.2 Forward linearised tree pattern matching algorithm

One of the usages of the border array is in the Morris-Pratt algorithm. The string version of the algorithm consists of two alternating components – occurrence check and shift computation. Both can be adapted to trees with the later requiring the tree pattern border array defined.

The Algorithm 3 is a modification of the Morris-Pratt algorithm for strings. It uses index j into the $pref(p)$ and two indexes i and *offset* into the $pref(s)$. The index i holds the position of the current attempt and *offset* holds the position of currently compared symbol. Index *offset* is needed due to the "elasticity" of the pattern caused by the wildcards. The shift distances are precomputed but otherwise the computation is unchanged, however, the number of symbols that are not required to be matched again is derived from the border array and is limited by the first occurrence of the wildcard due to variability of the subtree in place of the wildcard.

Algorithm 2: Computation of tree pattern border array.

Input: A pattern tree $pref(p)$ (pattern) of size m and a vector of integers $sjt(pref(p))$
Output: A vector of integers $\mathcal{B}(pref(p))$ indexed as $[1..m]$

```
1  begin
2      for i := 1 to m do B(pref(p))[i] := 0
3      for offset := 1 to m do
4          i := 1                                   /* index into full pref(p) */
5          j := offset + 1                          /* index into a factor of pref(p) */
6          while True do
7              if i > m then
8                  while j ≤ m do
9                      B(pref(p))[j] := max(B(pref(p))[j], j − offset)
10                     j += 1
11                 end
12                 break
13             else if j > m then
14                 break
15             else if pref(p)[i] = pref(p)[j] then
16                 B(pref(p))[j] := max(B(pref(p))[j], j − offset)
17                 i += 1
18                 j += 1
19             else if pref(p)[i] = S ∨ pref(p)[j] = S then
20                 for k := j to sjt(pref(p))[j] − 1 do
21                     B(pref(p))[k] := max(B(pref(p))[k], k − offset)
22                 end
23                 i := sjt(pref(p))[i]                 /* skip S */
24                 j := sjt(pref(p))[j]
25             else
26                 break                                /* mismatch */
27             end
28         end
29     end
30 end
```

Theorem 9. *Given a tree pattern p in* prefix notation *$pref(p)$ and tree pattern border array $\mathcal{B}(pref(p))$ constructed by Algorithm 2, the Algorithm 3 does not skip any occurrence of the pattern p in an input tree t.*

Proof. Assume that the match attempt found a mismatch on j-th symbol of the pattern, therefore the $shift$ is either by single position if $j = 1$, which is always safe, or, according to the tree pattern border array, $j - \mathcal{B}(pref(p))[j-1] - 1$ if $j \geq 1$.

Assume, that for the shift where $j \geq 1$ there is a shorter shift by k positions, where $0 < k < j - \mathcal{B}(pref(p))[j-1] - 1$. It must therefore be possible to match $pref(p)[k..j-1]$ with the pattern $pref(p)$ itself. However, the shift for mismatch at j-th position is derived from the $\mathcal{B}(pref(p))[j-1]$, which according to the Definition 7 failed to match $pref(p)[k..j-1]$ with the pattern $pref(p)$ itself by Definition 6 for each $0 < k < j - \mathcal{B}(pref(p))[j-1] - 1$. □

3.3 Example

Example 10. Consider a tree pattern p and a subject s with their respective representations in prefix notation $pref(p) = a2\ a2\ S\ a2\ b1\ S\ a0\ a0$ and $pref(s) = a2\ a2$

Algorithm 3: Forward tree pattern matching algorithm

Input: The subject tree in $pref(s)$ notation of size n, the tree pattern in $pref(p)$ notation of size m, and a vector of integers $sjt(pref(p))$
Input: A vector of integers $\mathcal{B}(pref(p))$
Result: Locations of occurrences of the tree pattern p in the subject tree s.

```
1  begin
2      Spos := min({j : pref(p)[j] = S ∧ 1 ≤ j ≤ m})
3      shift[1] := 1
4      for i := 2 to m + 1 do shift[i] := i − B(pref(p))[i − 1] − 1
5      i := 0
6      j := 1
7      while i ≤ n − m do
8          offset := i + j
9          while j ≤ m and offset ≤ n do
10             if pref(p)[j] = pref(s)[offset] then
11                 j += 1
12                 offset += 1
13             else if pref(p)[j] = S then
14                 offset := sjt(pref(s))[offset]
15                 j += 1
16             else
17                 break
18             end
19         end
20         if j > m then yield i + 1
21         i += shift(pref(p))[j]
22         j := max(1, min(Spos, j) − shift(pref(p))[j])
23     end
24 end
```

id	1	2	3	4	5	6	7	8	9
pref(p)	$a2$	$a2$	S	$a2$	$b1$	S	$a0$	$a0$	
$\mathcal{B}(pref(p))$	0	1	2	2	3	4	5	6	
shift(pref(p))	1	1	1	1	2	2	2	2	2

Table 2. The tree pattern border array $\mathcal{B}(pref(p))$ and $shift(pref(p))$ used in Algorithm 3.

$a2$ $a0$ $a2$ $b1$ $b0$ $a0$ $a0$ $a2$ $a2$ $a0$ $a2$ $b1$ $b0$ $a0$ $a0$. Table 2 shows the tree pattern border array and derived shift function values for pattern p and Table 3 shows the run of the tree pattern matching algorithm. Matches are at indexes 2 and 10.

3.4 Time complexity

Consider a pattern p of length m and a subject s of length n. The time complexity of the preprocessing phase (construction of $sjt(pref(p))$ and Algorithm 2) is $O(m^2)$.

The classical Morris-Pratt algorithm runs in linear time with respect to the subject size thanks to the saving some subject to pattern symbol comparisons arising from the border array properties. Since the Forward linearised tree pattern matching algorithm skips some parts of the subject tree where wildcards are and the information about symbols inside the skipped subtree is not known while doing so, the number of symbols not needed to be matched in the next pattern to subject alignment is limited by the first subtree wildcard in the tree pattern. Matching itself therefore takes $O(m \cdot n + occ)$

id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
pref(s)	$a2$	$a2$	$a2$	$a0$	$a2$	$b1$	$b0$	$a0$	$a0$	$a2$	$a2$	$a0$	$a2$	$b1$	$b0$	$a0$	$a0$
sjt	18	10	9	5	9	8	8	9	10	18	17	13	17	16	16	17	18
1	$a2$	$a2$	$\vdash$			S		$\dashv$	$a2$								
2		$a2$	$a2$	S	$a2$	$b1$	S	$a0$	$a0$								
3				$a2$													
4					$a2$	$a2$											
5						$a2$											
6							$a2$										
7								$a2$									
8									$a2$								
9										$a2$	$a2$	S	$a2$	$b1$	S	$a0$	$a0$

Table 3. Algorithm 3 run for the subject and the pattern from Example 10.

in general and $\Theta(n + occ)$ time when the pattern tree does not contain any wildcard. Time required for construction of $sjt(pref(s))$ is included.

4 Some empirical results

An existing Forest FIRE toolkit and accompanying FIRE Wood graphical user interface [7,19] were extended with the implementation of the presented algorithm. Many tree pattern matching algorithms based on automata, like those described in [2,6,8,12] and others, are already implemented within the toolkit. Single pattern matching algorithm based on linearisations of both pattern tree and subject tree utilising a backward shift heuristics [20] is also present. Performance of the presented algorithm is compared with some of the best-performing algorithms in the toolkit based on automata, according to the results in [8], and with the algorithm based on linearisation of tree structures utilising a backward shift heuristics. The running time of the pattern preprocessing was not measured as it is done only once for all queries by the pattern on many subjects.

We have measured the following running times of searching phases: 1) our new forward tree pattern matching algorithm based on linearisations of pattern and subject tree (FLTPM); 2) an algorithm based on linearisation of pattern and subject tree utilising a backward shift heuristics (BLTPM); 3) an algorithm based on the use of a *deterministic frontier-to-root (bottom-up) tree automaton* constructed for the pattern (DFRTA); and 4) an algorithm based on the use of a *Aho-Corasick automaton* constructed for the pattern's stringpath set (AC). The construction of a subtree jump table is included in the running time for both FLTPM and BLTMP algorithms and these algorithms' running times were recorded with trees represented in their prefix notation. Additionally, a modified version of the DFRTA algorithm reading prefix notation of a subject was implemented in the Forest FIRE toolkit as another reference algorithm (DFRTA Prefix). Since this algorithm doesn't need subtree jump table, its construction isn't included in the running time of DFRTA Prefix algorithm.

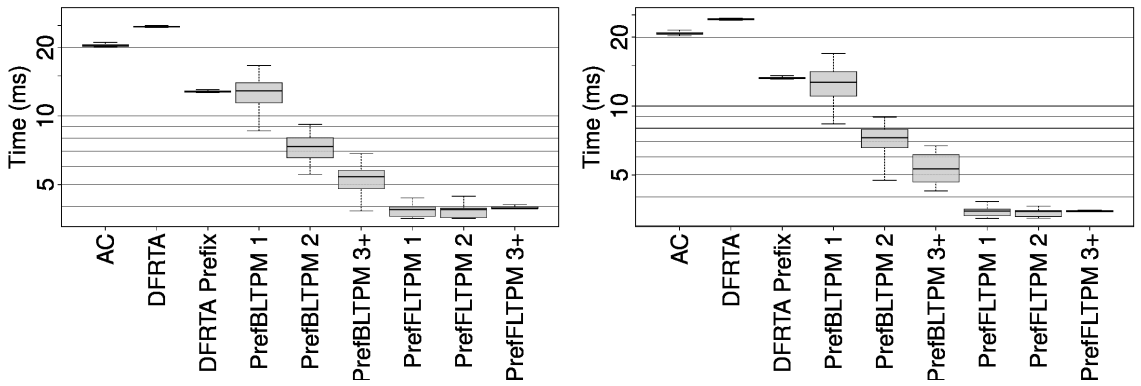
The performance of the new algorithm was measured using a pattern set previously used to measure the performance of preexistent algorithms in the Forest FIRE toolkit. This pattern set was obtained by taking the Mono project's X86 instruction set grammar and, for each grammar production, taking the tree in the production's right-hand side, and replacing any nonterminal occurrences by wildcard symbol occurrences. The resulting pattern set consists of 460 tree patterns of varying sizes.

Two sets of subject trees were used previously to measure the performance of Forest FIRE toolkit and the same two sets were used in the benchmarking of the new algorithm. The two subject sets were a set of 150 trees of approximately 500 nodes each and a set of 500 trees of approximately 150 nodes each.

As in the case of the BLTPM algorithm, the new algorithm is a single-pattern one. All chosen algorithms were executed with each pattern from the pattern set and each subject tree from two subject sets individually. The running times of the pattern matching algorithms were aggregated for a single tree pattern and all subject trees.

Benchmarking was conducted on a 2 GHz Intel Core i7 with 24 GB of RAM running OpenSUSE GNU/Linux version 15.2 using Java SE 11.

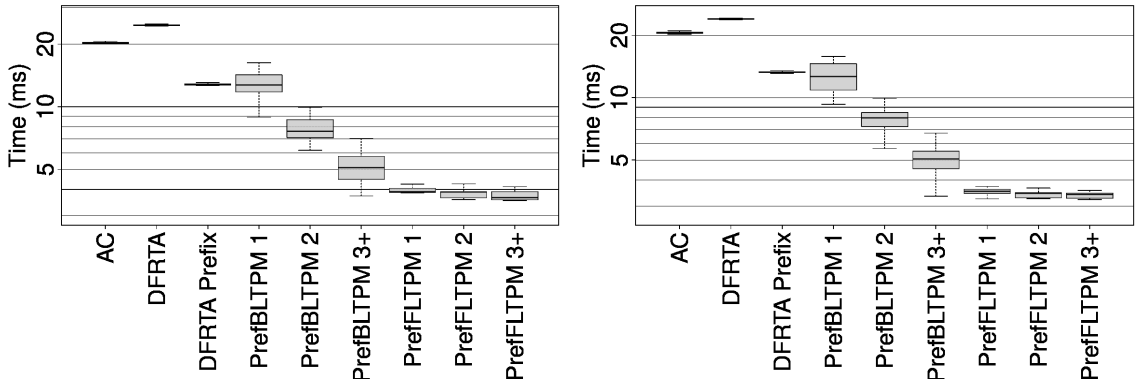
The linearised representations of subject trees and pattern trees were constructed in-memory and are linear in size with respect to the sizes of the subject trees and the pattern trees. The time required to construct the linear representation was not included in the running time of the searching phase. Also, because the search time was our primary concern, we do not consider memory use. Figure 3a and Figure 3b show the search times of tree patters with a wildcard symbol as boxplots. The figures of BLTPM and FLTPM algorithms were split to three based on the distance of the first wildcard symbol from the beginning of the pattern in its prefix representation to present this distance affects the running time. The distances are one symbol, two symbols, and three and more symbols. Similarly, Figure 4a and Figure 4b show the search times of tree patterns without a wildcard symbol as boxplots. The figures for BLTPM and FLTPM algorithms were split to three based on the length of the tree patterns, to patterns of length one, two, and three and more.



(a) Results on 150 trees of ca. 500 nodes each. (b) Results on 500 trees of ca. 150 nodes each.

Figure 3. Distributions of pattern matching times for the respective algorithms and patterns with wildcards.

The BLTPM algorithm generally runs faster for longer tree patterns without a wildcard or with a wildcard further from the beginning of the pattern whereas the FLTPM algorithm is unaffected by the wildcard position nor by the length of the pattern. The plots are clearly showing that on average, our new forward linearised tree pattern matching algorithm considerably outperforms the existing ones based on the automata approach for the single-pattern case (note the logarithmic scale) and even the backward linearised tree pattern matching algorithm.



(a) Results on 150 trees of ca. 500 nodes each. (b) Results on 500 trees of ca. 150 nodes each.

Figure 4. Distributions of pattern matching times for the respective algorithms and patterns without wildcards.

5 Concluding remarks

We presented a property of linearised trees similar to border arrays from strings. Using tree pattern border arrays, a new forward tree pattern matching algorithm similar to the Morris-Pratt algorithm for strings is defined. The algorithm was designed for trees represented in the prefix notation, but the idea can be adapted to other notations. The algorithm was empirically compared with other pattern matching algorithms and was shown to perform well in practice. Future work should focus on the identification of properties of the tree pattern border array to improve the preprocessing time and modification of the shift heuristics similar to the one used in the Knuth-Morris-Pratt algorithm. Future work should also include an investigation into a shift heuristics for multiple tree patterns, i.e., into a modification of the Aho-Corasick algorithm.

References

1. A. Aho and M. Corasick: *Efficient string matching: An aid to bibliographic search.* Commun. ACM, 18 06 1975, pp. 333–340.
2. A. V. Aho, M. Ganapathi, and S. W. K. Tjiang: *Code generation using tree matching and dynamic programming.* ACM Trans. Program. Lang. Syst., 1989, pp. 491–516.
3. A. V. Aho and J. D. Ullman: *The theory of parsing, translation, and compiling*, Prentice-Hall, 1972.
4. R. Beal and D. Adjeroh: *Border array for structural strings*, in Combinatorial Algorithms, S. Arumugam and W. F. Smyth, eds., Berlin, Heidelberg, 2012, Springer Berlin Heidelberg, pp. 189–205.
5. R. Beal and D. Adjeroh: *Border array for structural strings*, in Combinatorial Algorithms, S. Arumugam and W. F. Smyth, eds., Berlin, Heidelberg, 2012, Springer Berlin Heidelberg, pp. 189–205.
6. D. R. Chase: *An improvement to bottom-up tree pattern matching*, in POPL, ACM Press, 1987, pp. 168–177.
7. L. Cleophas: *Forest FIRE and FIRE Wood: Tools for tree automata and tree algorithms*, in FSMNLP, J. Piskorski, B. W. Watson, and A. Yli-Jyrä, eds., vol. 19 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2008, pp. 191–198.
8. L. Cleophas: *Tree Algorithms: Two Taxonomies and a Toolkit*, PhD thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, Apr. 2008.
9. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi: *Tree automata: Techniques and applications*, 2007, `http://www.grappa.univ-lille3.fr/tata/`.

10. T. Flouri, J. Janoušek, B. Melichar, C. S. Iliopoulos, and S. P. Pissis: *Tree template matching in ranked ordered trees by pushdown automata*, in Implementation and Application of Automata, B. Bouchou-Markhoff, P. Caron, J.-M. Champarnaud, and D. Maurel, eds., vol. 6807 of Lecture Notes in Computer Science, Springer Verlag, 2011, pp. 273–281.
11. F. Gécseg and M. Steinby: *Tree Languages*, vol. 3 of Handbook of Formal Languages, Springer, 1997, pp. 1–68.
12. C. M. Hoffmann and M. J. O'Donnell: *Pattern matching in trees.* Journal of the ACM, 29(1) January 1982, pp. 68–95.
13. J. Janoušek: *Arbology: Algorithms on trees and pushdown automata*, PhD thesis, habilitation thesis, Brno University of Technology, 2010, submitted, 2010.
14. J. Janoušek, B. Melichar, R. Polách, M. Poliak, and J. Trávníček: *A full and linear index of a tree for tree patterns*, in Descriptional Complexity of Formal Systems, H. Jürgensen, J. Karhumäki, and A. Okhotin, eds., vol. 8614 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 198–209.
15. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt: *Fast pattern matching in strings.* SIAM journal on computing, 6(2) 1977, pp. 323–350.
16. B. Melichar, J. Janoušek, and T. Flouri: *Arbology: trees and pushdown automata.* Kybernetika, 48, No.3 2012, pp. 402–428.
17. J. Morris Jr and V. Pratt: *A linear pattern-matching algorithm*, Technical Report 40, University of California, Berkeley, 1970.
18. W. F. Smyth: *Computing Patterns in Strings*, Addison-Wesley-Pearson Education Limited, 2003.
19. R. Strolenberg: *ForestFIRE & FIREWood, A Toolkit & GUI for Tree Algorithms*, Master's thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, June 2007, `http://alexandria.tue.nl/extra1/afstversl/wsk-i/strolenberg2007.pdf`.
20. J. Trávníček, J. Janoušek, B. Melichar, and L. Cleophas: *On modification of boyer-moore-horspool's algorithm for tree pattern matching in linearised trees.* Theoretical Computer Science, 830-831 2020, pp. 60 – 90.

Greedy versus Optimal Analysis of Bounded Size Dictionary Compression and On-the-Fly Distributed Computing

Sergio De Agostino

Computer Science Department
Sapienza University of Rome
Via Salaria 113, 00198 Rome, Italy
deagostino@di.uniroma1.it

Abstract. Scalability and robustness are not an issue when compression is applied for massive data storage, in the context of distributed computing. Speeding up on-the-fly compression for data transmission is more controversial. In such case, a compression technique merging together an adaptive and a non-adaptive approach has to be considered. A practical implementation of LZW (Lempel, Ziv and Welch) compression, called LZC (C indicates the Unix command 'compress'), has this characteristic. The non-adaptive phases work with bounded size prefix dictionaries built by LZW factorizations during the adaptive ones. In order to improve the compression effectiveness, we suggest to apply LZMW (Lempel, Ziv, Miller and Wegman) factorization to LZC compression (LZCMW) during the adaptive phases since it builds better dictionaries than LZW. The LZMW heuristic was originally thought with a dictionary bounded by the least recently used strategy. We introduce the LZCMW heuristic in order to have non-adaptive phases. All the heuristics mentioned above employ the greedy approach. We show, finally, a worst case analysis of the greedy approach with respect to the optimal solution decodable by the LZC decompressor. Such analysis suggests parallelization of on the fly compression is not suitable for highly disseminated data since the non-adaptive phases are too far from optimal.

Keywords: on-the-fly compression, factorization, distributed system, scalability

1 Introduction

Massive data are, usually, defined as big when the order of magnitude is greater than a terabyte. Compression is considered advantageous and actionable by the big data community since it is possible to evaluate many predicates without having to decompress. However, the locality principle stated by Zipf's law [26] for arbitrarily large datasets provides another good reason (perhaps an even better reason in the near future) for massive data compression, that is, compression and, more importantly, decompression that are highly parallelizable. In fact, using the computational resources to speed up compression and decompression could be more practical than employing sophisticated techniques to query compressed data (as, for example, compressed pattern matching). Moreover, the locality principle is so general that parallelism could be applied to any kind of data and with any compression technique, even with smaller order of magnitudes than a petabyte. Indeed, any sequential compression technique could be applied in parallel and independently to relatively large pieces of data on standard small, medium and large scale distributed systems.

While speeding up massive data compression for storage on a distributed system is not an issue for the reasons mentioned above, speeding up on-the-fly compression

Sergio De Agostino: Greedy versus Optimal Analysis of Bounded Size Dictionary Compression and On-the-Fly Distributed Computing, pp. 74–83.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9

for data transmission is more controversial. In such case, it seems that the only way to exploit the full computational power of a distributed system is to employ a technique comprising an adaptive phase followed by a non-adaptive one which can process the data until it becomes obsolete. Then, iteratively, the two phases can be repeated. This iteration of the sequential procedure can be parallelized if we divide the input into blocks of data long enough to run one step of the iteration on each block. Each block is, obviously, split in a first sub-block where the adaptive phase is run and a second one to be compressed in a non-adaptive way. A general parallel approach for this kind of compression technique is to compress the first sub-block with only one processor and, afterwards, to parallelize massively the compression of the second sub-block. We will see that this is possible if we employ a practical implementation of LZW compression [24], called LZC [1], since it has this characteristic. The non-adaptive phases work with bounded size prefix dictionaries built by LZW factorizations during the adaptive ones. In order to improve the compression effectiveness, we suggest to apply LZMW (Lempel, Ziv, Miller and Wegman) factorization to LZC compression (LZCMW) during the adaptive phases since it builds better dictionaries than LZW. The LZMW heuristic was originally thought with a dictionary bounded by the least recently used strategy [21]. We introduce the LZCMW heuristic in order to have non-adaptive phases.

All the heuristics mentioned above employ the greedy approach. We show, finally, a worst case analysis of the greedy approach with respect to the optimal solution decodable by the LZC decompressor. Such analysis suggests parallelization of on-the-fly compression is not suitable for highly disseminated data since the non-adaptive phases are too far from optimal. Massive data compression for storage and the locality principle are discussed in Section 2, where implementations of Zip and LZW compressors are described. How to speed-up on-the-fly LZW compression with distributed computing is faced in Section 3, where the improvement by means of the LZMW heuristic is discussed. The greedy versus optimal analysis is given in Section 5. Conclusion and future work of this pure theory paper are given in Section 6.

2 Distributed Massive Data Compression

Distributed systems have two types of complexities, the interprocessor communication and the input-output mechanism. While the input/output issue is inherent to any parallel algorithm and has standard solutions, the communication cost of the computational phase after the distribution of data among the processors and before the output of the final result is obviously algorithm-dependent. So, we need to limit the interprocessor communication and involve more local computation to design a practical algorithm. The simplest model for this phase is, of course, a simple array of processors with no interconnections and, therefore, no communication cost. The arguments of next subsection imply that compressing massive data for storage is not an issue for any technique on such a distributed system and the most popular compressors belonging to the Zip family are described. On the other hand, improving the speed-up of on-the-fly compression for massive data transmission is more controversial and requires characteristics that, to our knowledge, only LZW compression has. We describe the unbounded and bounded memory versions of this technique in the second and third subsection so that the issue of massive data transmission and distributed computing can be faced in the next section.

2.1 The Locality Principle

Zipf's law states that the frequency of any word in a collection is inversely proportional to its rank in the frequency table. The most frequent word occurs twice as often as the second most frequent, and so on. The effect of this uneven distribution of byte patterns is evident in the effectiveness of common compression programs, as for example, the family of Zip compressors based on the Lempel-Ziv sliding window factorization technique [18], [19], [23]. Such factorization of a string S is $S = f_1 f_2 \cdots f_i \cdots f_m$ where f_i is the longest match with a substring occurring previously in the prefix $f_1 f_2 \cdots f_i$ if $f_i \neq \lambda$ (empty string), otherwise f_i is the alphabet character next to $f_1 f_2 \cdots f_{i-1}$. f_i is encoded by the pointer $q_i = (d_i, \ell_i)$, where d_i is the displacement back to the copy of the factor and ℓ_i is the length of the factor. If $d_i = 0$, ℓ_i is the alphabet character. In other words, there is a window sliding its right end over the input string and all the substrings of the prefix read so far in the computation are potential reference copies for the current factor. In practice, the work space must be bounded and this is done by sliding a fixed length window and by bounding the match length. Simple real time implementations are realized by means of hashing techniques providing a specific position in the window where a good approximation of the longest match is found on realistic data. In [25], the two current characters are hashed and collisions are chained via an offset array. The Unix gzip compressor chains collisions too but hashes three characters [16].

While gzip stores 64 kB of history, it averages approximately 64 percent compression. Instead, bzip2 stores between 100 kB and 900 kB of history and averages 66 percent compression, a very small gain in comparison with the increase of memory. Therefore, 64 kB of history are enough to compress data and this can be generalized to any compression technique we want to use. Such locality principle is critical if we want to speed up compression and, more importantly, decompression on a large scale network. Indeed, we can apply compression in parallel to data blocks relatively small (a few hundreds kilobytes) since the locality principle holds for arbitrarily large datasets on the basis of Zipf's law and scalability and robustness are guaranteed. This follows from the fact that the loss of compression effectiveness due to the lack of history at the beginning of each block is amortized if the block length is one order of magnitude greater than the amount of past data to which we need to refer for compression [8]. Then, distributing data blocks of size approximately half a megabyte among the nodes of a network allows an efficient application of any adaptive compression method and guarantees scalability and robustness for massive data (for the interested reader, improved variants of the sliding window factorization technique exist employing either fixed-length codewords [2], [20] or variable-length ones [3], [13], [14], [15], [17]). Parallel decompression is symmetrical.

2.2 LZW Compression

Ziv-Lempel compression is a dictionary-based technique [18], [27], using a string factorization process where the factors of the string are substituted by *pointers* to copies stored in a dictionary which are called *targets*. The *standard* LZW (Lempel-Ziv-Welch) factorization of a string S is $S = f_1 f_2 \cdots f_i \cdots f_m$ where each factor f_i is the longest match with the concatenation of a previous factor and the next character [24]. f_i is encoded by a pointer q_i whose target is such concatenation (LZW compression). LZW compression can be implemented in real time by storing the dictionary with a trie data structure. When the string length goes to infinity, also the dictionary size

does. Such unbounded version was proved to be P-complete [4], [5], [6], [7], [8], meaning that such factorization is hard to parallelize even on a shared memory random access machine or on a highly interconnected network. Therefore, we need to consider bounded memory versions of LZW compression in order to make such technique suitable for distributed computing.

2.3 Bounded Size Dictionaries

In practical implementations the dictionary size is bounded by a constant and the pointers have equal size. Let $d + \alpha$ be the cardinality of the fixed size dictionary where α is the cardinality of the alphabet. With the most naive approach, there is a first phase of the factorization process where the dictionary is filled up and "frozen". Afterwards, the factorization continues in a non-adaptive way using the factors of the frozen dictionary. In other words, the factorization of a string S is $S = f_1 f_2 \cdots f_i \cdots f_m$ where f_i is the longest match with the concatenation of a previous factorf_j, with $j \leq d$, and the next character. The shortcoming of this heuristic is that after processing the string for a while the dictionary often becomes obsolete. If the dictionary elements were removed with a heuristic, new elements could be added.

The best deletion heuristic is the LRU (last recently used) strategy [22]. The LRU deletion heuristic removes elements from the dictionary by deleting at each step of the factorization the least recently used factor, which is not a proper prefix of another one. If the size of the dictionary is $\mathrm{O}(\log^k n)$, the LRU strategy is log-space hard for SC^k [7],[8],[12]. SC is the class of problems solvable simultaneously in polynomial time and polylogarithmic space and SC^k is the class of problems solvable simultaneously in polynomial time and $\mathrm{O}(\log^k n)$ space for a fixed k. LZW compression using a dictionary of size $\mathrm{O}(\log^k n)$ and the LRU deletion heuristic belongs to SC^{k+1}. Indeed, the LZW algorithm with LRU deletion heuristic on a dictionary of size $\mathrm{O}(\log^k n)$ can be performed in polynomial time and $\mathrm{O}(\log^k n \log(\log n))$ space, where n is the length of the input string. The trie requires $\mathrm{O}(\log^k n)$ space by using an array implementation since the number of children for each node is bounded by the alphabet cardinality. The $\log(\log n)$ factor is required to store the information needed for the LRU deletion heuristic since each node must have a different age, which is an integer value between 0 and the dictionary size.

The hardness result is not so relevant for the space complexity analysis since $\Omega(\log^k n)$ is an obvious lower bound to the work space needed for the computation. Much more interesting is what can be said about the parallel complexity analysis. In [12] it was shown that LZW compression using the LRU deletion heuristic with a dictionary of size c can be performed in parallel either in $\mathrm{O}(\log n)$ time with $2^{\mathrm{O}(c \log c)} n$ processors or in $2^{\mathrm{O}(c \log c)} \log n$ time with $\mathrm{O}(n)$ processors on a shared memory random access machine or a highly connected network. This means that if the dictionary size is constant the compression problem belongs to NC, the class of problems solvable in polylogarithmic parallel time with a polynomial number of processors on a random access shared memory machine or a highly connected network. NC and SC are classes that can be viewed in some sense symmetric and are believed to be incomparable. Since log-space reductions are in NC, the compression problem cannot belong to NC when the dictionary size is polylogarithmic if NC and SC are incomparable. We want to point out that the dictionary size c figures as an exponent in the parallel complexity of the problem. This is not by accident. If we believe that SC is not included in NC, then the SC^k-hardness of the problem when c is $\mathrm{O}(\log^k n)$ implies the exponentiation

of some increasing and diverging function of c. Indeed, without such exponentiation either in the number of processors or in the parallel running time, the problem would be SC^k-hard and in NC when c is $\mathrm{O}(\log^k n)$. Observe that the P-completeness of the problem, which requires a superpolylogarithmic value for c, does not suffice to infer this exponentiation since c can figure as a multiplicative factor of the time function. Moreover, this is a unique case so far where somehow we use hardness results to argue that practical algorithms of a certain kind (NC in this case) do not exist because of huge multiplicative constant factors occurring in their analysis. If in theory LZW compression with the LRU deletion heuristic cannot be parallelized on any kind of parallel or distributed system for the reasons explained above, in practice the locality principle allows an effective distributed implementation. Considering the fact that a practical bound to the dictionary size is 2^{16} and that about 300 kilobytes are enough to fill up a dictionary of this size on realistic data, we can argue that compressing independently blocks of at least 600 kilobytes is a sufficiently robust approach. However, the process of adding and removing dictionary elements at each step never works in a non-adaptive way. The deletion heuristic providing such method is RESTART. After the dictionary is filled up, the RESTART deletion heuristic starts a non-adaptive phase and monitors the compression ratio. When the ratio deteriorates, the heuristic deletes all the elements from the dictionary but the alphabet characters and restarts a new adaptive phase. Let $S = f_1 f_2 \cdots f_j \cdots f_i \cdots f_m$ be the factorization of the input string S computed by the LZW compression algorithm using the RESTART deletion heuristic. Let j be the highest index less than i where a restart operation happens. Then, f_j is an alphabet character and f_i is the longest match with the concatenation of a previous factor f_h, with $h \geq j$, and the next character (1 and $k+1$ are considered restart positions by default). This heuristic, called LZC [1], is used by the Unix command Compress since it has a good compression effectiveness and it is easy to implement. Since the dictionary size is 2^{16} the number of different concatenations of a factor with the next character between f_h and f_t is equal to 2^{16} decreased by the alphabet size, with h and t two consecutive positions where the restart operation happens (f_t is not counted). Usually, the dictionary performs well in a non-adaptive way on a block long enough to learn another dictionary of the same size. This is what is done by the SWAP deletion heuristic. When the other dictionary is filled, they swap their roles on the successive block. The improvement introduced by the SWAP heuristic cannot be utilized with distributed computing since the processing of the successive block cannot start until the dictionary is learned from the previous block. In conclusion, LZC compression is the version we will adopt in the next section.

3 Speeding up On-the-Fly Data Compression

We have seen in the previous section that LZC compression is a technique comprising an adaptive phase followed by a non-adaptive one which can process the data until it becomes obsolete. Since the dictionary size is 2^{16} in practical implementations and about 300 kB are enough to fill up a dictionary of this size on realistic data, another 300 kB can be considered a robust lower bound to the amount of data for which the non-adaptive way works properly [9]. Then, we divide conceptually the input into blocks of about 600 kB with each block split in a first half of about 300 kB, where the adaptive phase is run sequentially, and a second one to be compressed in a non-adaptive way with the dictionary just learned, using the computational power of the distributed system. The second half is, therefore, partitioned into sub-blocks. The

architecture of the distributed system could be modeled as a star network where the central node runs the sequential adaptive phase until the dictionary is filled. At each factorization step, the central node sends the current factor concatenated with the next character to the adjacent nodes to update their own copy of the dictionary. To speed up the broadcasting of such concatenation its longest proper prefix (that is, the current factor) can be compressed with the pointer having such prefix as target. This means that the processors receiving the factors store the dictionary in a trie using an auxiliary perfect hashing table where the pointers are the keys and the targets are the values. Then sub-blocks of the next 300 kilobytes are broadcasted to the adjacent nodes. We show how to implement the non-adaptive phase on small and medium scale systems in the next subsection. Then, we scale up the system and modify the approach to keep its robustness in the second subsection. The third subsection considers decompression. Improvements by the LZMW approach are discussed in the last subsection.

3.1 Small and Medium Scale

After having just learned the dictionary, 300 kB have to be compressed in a non-adaptive way using such dictionary. In [10], it is shown that if we distribute sub-blocks among different processors to compress them independently then the sub-block length must be the order of a few kilobytes to guarantee robustness. Ten processors and one hundred processors are the orders of magnitudes for small and medium scale distributed systems, respectively. Therefore, robustness is guaranteed. On the other hand, the size of a large scale system is the order of magnitude of a thousand implying that the sub-block length has the order of magnitude of 100 bytes. In such case, overlapping of adjacent sub-blocks and a preprocessing of the boundaries are necessary. We experimented in [10] that, when compressing megabytes of English text, the LZC average compression ratio is 0.42 while the distributed approach has a one percent loss in both cases.

3.2 Large Scale

Both overlapping of adjacent sub-blocks and a preprocessing of the boundaries are necessary since the boundary positions are arbitrary and, therefore, likely not to be at the beginning of natural factors. Consequently, a sub-block factorization starts with factors much smaller than the average factor length. This initial disadvantage is amortized if the sub-block length has the order of magnitude of a kilobyte, which is not the case of large scale distributed systems as explained in the previous subsection. Again, after having just learned the dictionary, 300 kB have to be compressed in a non-adaptive way using such dictionary. Generally speaking, sub-blocks of length $M(k+2)$, except for the first one and the last one which are $M(k+1)$ long, are broadcasted to the processors, with k a positive integer and M the maximum factor length. Each sub-block overlaps on $2M$ characters with the adjacent ones to the left and to the right, respectively (obviously, the first one overlaps only to the right and the last one only to the left). We call a *boundary match* a factor either covering positions in the first and second half of the $2M$ characters shared by two adjacent sub-blocks or being a suffix of the first half. The processors execute the following algorithm to compress each sub-block:

- for each sub-block, every corresponding processor computes the longest boundary matches to the left and to the right (only to the right (left) if the sub-block is the first (last) one).
- each processor computes the greedy factorization from the end of the boundary match on the left boundary of its sub-block to the beginning of the boundary match on the right boundary.

The parallel running time of the preprocessing phase computing the boundary matches is $O(M^2)$ by brute force. In [10], it is shown experimentally that for $k = 10$ the compression ratio achieved by such factorization is about the same as the sequential one. Considering that typically the average match length is about 10, one processor can compress 100 bytes independently and this is why this approach is suitable for a large scale distributed system.

3.3 On-the-Fly Decompression

To decode on-the-fly the compressed data on a distributed system, after the sequential decompression of the first half of a block, it is enough to use during the compressing phase a special mark occurring in the sequence of pointers each time the coding of a sub-block ends in the second half of the block. A copy of the dictionary is stored in every processor since the sequential decompression of the first half is run again by the central node of the star network, sending a new dictionary element to the adjacent nodes at each step. Differently from the coding phase, a perfect hashing table rather than a trie is used to store the dictionary. Moreover, the special marks allow the broadcasting of the subsequences of pointers coding each sub-block of the second half to the adjacent nodes. Therefore, the decoding of the sub-blocks is straightforward.

3.4 Improving On-the-Fly Compression of Massive Data

The prefix dictionaries built by LZW have the disadvantage to include useless elements. To ameliorate this, the LZMW factorization builds better dictionaries by updating it at each step with the concatenation of the last two factors. Therefore, the LZMW factorization of a string S is $S = f_1 f_2 \cdots f_i \cdots f_m$ where each factor f_i is the longest match with the concatenation of two consecutive previous factors. In practical implementations, the LZMW heuristic was originally thought with a dictionary bounded by the least recently used strategy and f_i was encoded by a pointer q_i whose target is in such dictionary [21], improving even ten per cent on LZW in some cases. We propose the LZCMW heuristic in order to have non-adaptive phases, where the dictionary is bounded to 2^{16} elements by the RESTART deletion heuristic. This makes LZMW compression suitable for parallel implementations of on-the-fly compression on small and large scale distributed systems in a similar fashion as for LZC. Therefore, the amelioration provided by LZMW compression in the sequential case is kept on every scale distributed system since the loss of compression effectiveness of the RESTART deletion heuristic with respect to LRU is quite limited [1]. Running the LZCMW heuristic on different types of data could be an interesting future experimental work.

4 The Greedy versus Optimal Analysis

A *feasible* d-restarted LZW factorization $S = f_1 \cdots f_m$ is such that the number of different concatenations of a factor with the next character between f_h and f_t (f_t is

not counted) is less or equal than d decreased by the alphabet size, with h and t two consecutive positions where the restart operation happens, and each factor f_i with $h < i < t$ is equal to $f_j c$, where c is the first character of f_{j+1} and $h \leq j < i$ (1 and $k+1$ are considered restart positions by default). We define *optimal* the feasible d-restarted LZW factorization with the smallest number of factors. A practical algorithm to compute the optimal solution is not known.

A trivial upper bound to the approximation multiplicative factor of a feasible factorization with respect to the optimal one is the maximum factor length of the optimal solution, that is, the height of the trie storing the dictionary. Such upper bound is $\Theta(d)$, where d is the dictionary size ($O(d)$ follows from the feasibility of the factorization and $\Omega(d)$ from the factorization of the unary string). We give worst case examples for the procedures of section 3 using only two characters a and b. It follows that the worst case analysis is valid for any given alphabet of cardinality greater than 1. If any of the procedures described in subsections 3.1 and 3.2 is applied to the input block of length d^2

$$b^{d^2/4-d/2}(\prod_{i=0}^{d/2-1} ab^iba^i)(\prod_{i=1}^{d} a^{d/2})$$

where the dictionary is learned in the first half and employed to compress in a non-adaptive way the second one, then the factorization of the first half of the block, that is, $b^{d^2/4-d/2}(\prod_{i=0}^{d/2-1} ab^iba^i)$ is

$$b, bb, \ldots, b^\ell, b^{\ell'}, a, b, ab, ba, abb, baa, \ldots, ab^i, ba^i, \ldots, ab^{d/2-1}, ba^{d/2-1}$$

where $\ell' \leq \ell + 1$ and the dictionary is filled if we assume its size is $d + \ell + 3$. The non-adaptive factorization of the second half is $a, a, \cdots a, a$ and the total cost of the factorization of the block is $\ell + 1 + d + d^2/2$, which is $\Theta(d^2)$. On the other hand, the cost of the optimal solution on the block is $\ell + 5d/2$ which is $\Theta(d)$ since the factorization of the first half is

$$b, bb, \ldots, b^\ell, b^{\ell'}, a, b, ab, b, a, abb, b, aa, \ldots, ab^i, b, a^i, \ldots, ab^{d/2-1}, b, a^{d/2-1}$$

Observe that the $O(d)$ approximation multiplicative factor depends on the non-adaptive phase and this happens when the dictionary learned on the first half of the block performs badly on the second half, that is in practice, when the data are highly disseminated. In such case, the *totally adaptive* feasible d-restarted LZW factorization (restarting as soon as the dictionary is filled with a greedy choice at each factorization step) is much more appropriate, as shown by the following theorem (the proof employs techniques similar to the ones for the unbounded dictionary case of [11]), but unfortunately does not seem to be parallelizable.

Theorem 1. *The totally adaptive d-restarted LZW factorization is an $O(\sqrt{d})$ approximation of the optimal one, where d is the dictionary size.*

Proof. Let S be the input string and T be the trie storing the dictionary of factors of the optimal d-restarted LZW factorization Φ of S between two consecutive positions where the restart operation happens. Each dictionary element (but the alphabet characters) corresponds to the concatenation of a factor f of the optimal factorization with the first character of the next factor, that we call an *occurrence* of the dictionary

element (node of the trie) in Φ. We call an element of the dictionary built by the greedy process *internal* if its occurrence is contained in the occurrence of a node of T and denote with M_T the number of internal occurrences. The number of non-internal occurrences is less than the number of factors of Φ. Therefore, we can consider only the internal ones. An occurrence f' of the greedy factorization internal to an factor f of Φ is represented by a subpath of the path representing f in T. Let u be the endpoint at the lower level in T of this subpath (which, obviously, represents a prefix of f). Let $d(u)$ be the number of subpaths representing internal phrases with endpoint u and let $c(u)$ be the total sum of their lengths. All the occurrences of the greedy factorization are different from each other between two consecutive positions where the restart operation happens. Since two subpaths with the same endpoint and equal length represent the same factor, we have $c(u) \geq d(u)(d(u)+1)/2$. Therefore

$$1/2 \sum_{u \in T} d(u)(d(u)+1) \leq \sum_{u \in T} c(u) \leq 2n \leq 2|\Phi| H_T$$

where H_T is the height of T, $|\Phi|$ is the number of phrases of Φ and the multiplicative factor 2 is due to the fact that occurrences of dictionary elements may overlap. We denote with $|T|$ the number of nodes in T; since $M_T = \sum_{u \in T} d(u)$, we have

$$M_T^2 \leq |T| \sum_{u \in T} d(u)^2 \leq |T| \sum_{u \in T} d(u)(d(u)+1) \leq 4|T||\Phi| H_T$$

where the first inequality follows from the fact that the arithmetic mean is less than the quadratic mean. Then

$$M_T \leq \sqrt{4|T||\Phi| H_T} = |\Phi| \sqrt{\frac{4|T| H_T}{|\Pi|}} \leq 2|\Phi| \sqrt{H_T}$$

Since the trie height is $\Theta(d)$ at worst, the theorem statement follows. □

We wish to point out that the proof works for any alphabet cardinality, so the comparative analysis of the procedures of section 3 with the totally adaptive d-restarted LZW factorization is the same for every non-unary alphabet on the highly disseminated data illustrated above.

5 Conclusion

We discussed massive data compression and decompression in the context of distributed computing. The locality principle implies that scalability and robustness are not an issue when compression is applied for massive data storage. Speeding up on-the-fly compression for data transmission is more controversial and, in such case, it seems we need compression techniques merging together an adaptive and a non-adaptive approach. We proposed a practical implementation of LZW compression, called LZC, as the most suitable to our knowledge since it has this characteriistic and introduced a new version of LZC compression, employing LZMW factorization in order to improve the compression effectiveness. However, a greedy versus optimal analysis shows such approach is not suitable for highly disseminated data. As future work, different techniques could be developed or existing ones could be adapted to work in this fashion with the purpose of improving compression effectiveness further.

References

1. T. C. Bell and I. H. Witten: *Text Compression*, Prentice Hall, 1990.
2. M. Crochemore, A. Langiu, and F. Mignosii: *Note on the greedy parsing optimality for dictionary-based text compression.* Theoretical Computer Science, 525 2014, pp. 55–59.
3. M. Crochemore, G. M., A. Langiu, F. Mignosi, and A. Restivo: *Dictionary-simbolwise flexible parsing.* Journal of Discrete Algorithms, 14 2012, pp. 74–90.
4. S. DeAgostino: *P-complete problems in data compression.* Theoretical Computer Science, 127 1994, pp. 181–186.
5. S. DeAgostino: *Sub-linear algorithms and complexity issues for lossless data compression*, 1994.
6. S. DeAgostino: *Parallelism and data compression via textual substitution*, 1995.
7. S. DeAgostino: *Parallelism and dictionary-based data compression.* Information Sciences, 135 2001, pp. 43–56.
8. S. DeAgostino: *Lempel-ziv data compression on parallel and distributed systems.* Algorithms, 4 2011, pp. 183–199.
9. S. DeAgostino: *Lzw data compression on large scale and extreme distributed systems*, in Proceedings Prague Stringology Conference, 2012, pp. 18–27.
10. S. DeAgostino: *The greedy approach to dictionary-based static text compression on a distributed system.* Journal of Discrete Algorithms, 34 2015, pp. 54–61.
11. S. DeAgostino and R. Silvestri: *A worst case analisys of the lz2 compression algorithm*, 1997.
12. S. DeAgostino and R. Silvestri: *Bounded size dictionary compression:* SC^k*-completeness and* nc *algorithms.* Information and Computation, 180 2003, pp. 101–112.
13. A. Farrugia, P. Ferragina, A. Frangioni, and R. Venturini: *Bicriteria data compression*, in Proceedings SIAM-ACM Symposium on Discrete Algorithms (SODA 14), 2014, pp. 1582–1585.
14. P. Ferragina, I. Nittoi, and R. Venturini: *On optimally partitioning a text to improve its compression.* Algorithmica, 61 2011.
15. P. Ferragina, I. Nittoi, and R. Venturini: *On the bit-complexity of lempel-ziv compression.* SIAM Journal on Computing, 42 2013.
16. J. Gailly and M. Adler: *http://www.gzip.org*, 1991.
17. A. Langiu: *On parsing optimality for dictionary-based text compression - the zip case.* Journal of Discrete Algorithms, 20 2013, pp. 65–70.
18. A. Lempel and J. Ziv: *On the complexity of finite sequences.* IEEE Transactions on Information Theory, 22 1976, pp. 75–81.
19. A. Lempel and J. Ziv: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23 1977, pp. 337–343.
20. Y. Matias and C. S. Sahinalp: *On the optimality of parsing in dynamic dictionary-based data compression*, in Proceedings SIAM-ACM Symposium on Discrete Algorithms (SODA 99), 1999, pp. 943–944.
21. V. S. Miller and M. N. Wegman: *Variations on theme by ziv-lempel*, 1985.
22. J. A. Storer: *Data Compression: Methods and Theory*, Computer Science Press, 1988.
23. J. A. Storer and T. G. Szymanski: *Data compression via textual substitution.* Journal of ACM, 29 1982, pp. 928–951.
24. T. A. Welch: *A technique for high-performance data compression.* IEEE Computer, 17 1984, pp. 8–19.
25. D. Whiting, G. A. George, and G. E. Ivey: *Data compression apparatus and method*, 1991.
26. G. K. Zipf: *The Ppsycology of Language*, Houghton-Mifflin, 1935.
27. J. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Transactions on Information Theory, 24 1978, pp. 530–536.

Left Lyndon Tree Construction

Golnaz Badkobeh[1] and Maxime Crochemore[2,3]

[1] Department of Computing
Goldsmiths University of London
United Kingdom
g.badkobeh@gold.ac.uk

[2] Department of Informatics
King's College London
United Kingdom
Maxime.Crochemore@kcl.ac.uk

[3] Université Gustave Eiffel
Marne-la-Vallée, France

Abstract. We extend the left-to-right Lyndon factorisation of a word to the left Lyndon tree construction of a Lyndon word. It yields an algorithm to sort the prefixes of a Lyndon word according to the infinite ordering defined by Dolce et al. (2019). A straightforward variant computes the left Lyndon forest of a word. All algorithms run in linear time on a general alphabet (letter-comparison model).

1 Lyndon words

In this article we consider algorithmic questions related to Lyndon words. Introduced in the field of combinatorics by Lyndon (see [11]) and used in algebra, these words have shown their usefulness for designing efficient algorithms on words. The notion of Lyndon tree associated with the decomposition of a Lyndon word, for example, has been used by Bannai et al. [1] to solve a conjecture of Kolpakov and Kucherov [9] on the maximal number of runs (maximal periodicities) in words, following a result in [2].

The key result in [1] is that every run in a word y contains as a factor, a Lyndon root (according to the alphabet order or its inverse), that corresponds to a node of the associated Lyndon tree. Since the Lyndon tree has a linear number of nodes according to the length of y, browsing all its nodes leads to a linear-time algorithm in order to report all the runs occurring in y. However, the time complexity of this technique also depends on the time it takes to build the tree and to extend a potential root to an actual run.

Here we consider the left Lyndon tree of a Lyndon word y. This tree has a single node if y is reduced to a single letter, otherwise its structure corresponds recursively to the left standard factorisation (see Viennot [13]) of y as uv where u is the longest proper Lyndon prefix of y.

The dual notion of the right Lyndon tree of a Lyndon word y (based on the factorisation $y = uv$ where v is the longest proper Lyndon suffix of y) is strongly related to the sorted list of suffixes of y. Indeed, Hohlweg and Reutenauer [8] showed that the Lyndon tree is the Cartesian tree built from the ranks of suffixes in their sorted list (sometimes called the inverse suffix array of the word). The list corresponds to the standard permutation of suffixes of the word and is the main component of the suffix array (see [4]), one of the major data structures for text indexing.

Golnaz Badkobeh, Maxime Crochemore: Left Lyndon Tree Construction, pp. 84–95.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9

Inspired by a result of Ufnarovskij [12], Dolce et al. [6] showed that the left Lyndon tree is also a Cartesian tree built from ranks of prefixes sorted according to an order they call the infinite order.

The main result of this article is to show that sorting prefixes of a Lyndon word according to the infinite order can be attained in linear time in the letter-comparison model (comparing letters is assumed to be carried out in constant time). This produces the prefix standard permutation of the word. The algorithm is based on the Lyndon factorisation of words by Duval [7] and it extends naturally to build the Lyndon forest of a word.

Definitions

Let A be an alphabet with an ordering $<$ and A^+ be the set of all non-empty words over A. The length of a word w is denoted by $|w|$. Let ϵ denotes the empty word, i.e., word of length 0. We say that uv is a non-trivial factorisation of a word w if $uv = w$ and u, v are non-empty words.

A word is said to be strongly smaller than a word v, $u \ll v$, if there are words r, s and t, and letters a and b with $u = ras$, $v = rbt$ and $a < b$. A word u is smaller than a word v, $u < v$, if either $u \ll v$ or u is a proper prefix of v. In addition to this usual lexicographical ordering the infinite order $\prec$ (see [5,6]) is defined by: $u \prec v$ if $u^\infty < v^\infty$ or both $u^\infty = v^\infty$ and $|u| > |v|$. Note that $u^\infty = v^\infty$ implies that u and v are powers of the same word, consequence of Fine and Wilf's Periodicity lemma (see [10, Proposition 1.3.5]). For example, if $u = \mathtt{abba}$, $v = \mathtt{abb}$, then $u^\infty = \mathtt{abbaabbaabba}\ldots$ and $v^\infty = \mathtt{abbabbabb}\ldots$, therefore $u^\infty < v^\infty$ and consequently $u \prec v$. If $u = \mathtt{ababab}$, $v = \mathtt{abab}$, then $u^\infty = v^\infty = \mathtt{abababab}\ldots$ and $u \prec v$.

The next proposition defines Lyndon words that are not reduced to a single letter. Condition in item (i) is the original definition and condition in item (iii) is by Ufnarovskij [12].

Proposition 1. *The following conditions are equivalent and define a Lyndon word w, $|w| > 1$: for any non-trivial factorisation uv of w, (i) $w < vu$, (ii) $w < v$, (iii) $u^\infty < w^\infty$.*

2 Lyndon suffix table

This section recalls known algorithms. The algorithms presented in this article strongly use the notion of Lyndon suffix table of a word, which is denoted by $LynS_y$ or simply by $LynS$ for the generic word y. Table $LynS$ of a word y is defined, for each position j on y, by

$$LynS[j] = \max\{|w| \mid w \text{ Lyndon suffix of } y[0\,..\,j]\}.$$

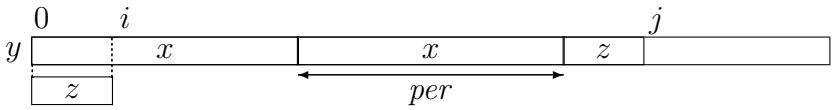
Example 2. Let $y_0 = \mathtt{ababbababbabac}$ on the alphabet of constant letters $\{\mathtt{a}, \mathtt{b}, \ldots\}$ ordered as usual $\mathtt{a} < \mathtt{b} < \cdots$. The corresponding $LynS_{y_0}$ table is as follows:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$y[j]$	a	b	a	b	b	a	b	a	b	b	a	b	a	c
$LynS_{y_0}[j]$	1	2	1	2	5	1	2	1	2	5	1	2	1	14

The $LynS$ table is the dual notion of the Lyndon table l in [1] or Lyn in [3] used to detect maximal periodicities (also called runs) in words: $Lyn[j]$ is the maximal length of the Lyndon prefixes of $y[j\,..\,|y|-1]$.

The computation of *LynS* is a mere extension of the algorithm for testing if a word is the prefix of a Lyndon word. It includes the key point of the factorisation algorithm in [7] and is recalled first as Algorithm LyndonWordPrefix that works online on its input word y.

```
LyndonWordPrefix(y non-empty word of length n)
 1  (per, i) ← (1, 0)
 2  for j ← 1 to n − 1 do
 3       if y[j] < y[i] then          ▷ y[i] = y[j − per]
 4            return FALSE
 5       elseif y[j] > y[i] then
 6            (per, i) ← (j + 1, 0)
 7       else i ← i + 1 mod per
 8  return TRUE
```



The key feature of the method stands in lines 5-6 of the algorithm and is illustrated on the above picture. If $y[j] > y[i] = y[j - per]$, not only the periodicity *per* of $y[0\,..\,j-1]$ breaks but $y[0\,..\,j]$ is a Lyndon word with period $j+1$. This feature is a consequence of the following known properties.

Proposition 3. *(i) Let z be a word and a a letter for which za is a prefix of the Lyndon word x. Let b be a letter with $a < b$. Then zb is a Lyndon word.*
(ii) Let u and v be two Lyndon words with $u < v$, then uv is Lyndon word.

Algorithm LyndonSuffix computes the Lyndon suffix table of a Lyndon word. This algorithm results from a minor modification of Algorithm LyndonWordPrefix and can be easily enhanced to compute also the smallest period of all non-empty prefixes of the input.

```
LyndonSuffix(y Lyndon word of length n)
 1  LynS[0] ← 1
 2  (per, i) ← (1, 0)
 3  for j ← 1 to n − 1 do
 4       if y[j] ≠ y[i] then          ▷ y[j] > y[i] = y[j − per]
 5            LynS[j] ← j + 1
 6            (per, i) ← (j + 1, 0)
 7       else LynS[j] ← LynS[i]
 8            i ← i + 1 mod per
 9  return LynS
```

Proposition 4. *Algorithm* LyndonSuffix *computes the Lyndon suffix table of a Lyndon word of length n in time $O(n)$ in the letter-comparison model.*

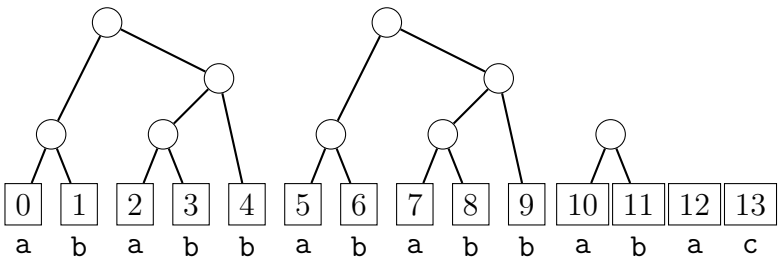
3 Left Lyndon tree construction

The left Lyndon tree $\mathcal{L}(y)$ of a Lyndon word y represents recursively the left standard factorisation of Lyndon words. Leaves of the tree are positions on the word and internal nodes correspond to concatenations of Lyndon factors of the word, and as such can be viewed as interpositions. Namely, $\mathcal{L}(y) = (0)$ if $|y| = 1$ else it is $(p, \mathcal{L}(u), \mathcal{L}(v))$ where the root $p \in \{|y|\,..\,2|y| - 2\}$ is an integer and uv is the left standard factorisation of y, that is, u is the longest proper Lyndon prefix of y (v is then a Lyndon word).

Subtrees of $\mathcal{L}(y)$ are handled from positions on y in the following manner. The subtree associated with position j is $\mathcal{L}(y[i\,..\,j])$ with root root$[j]$, where $y[i\,..\,j]$ is the longest Lyndon suffix of $y[0\,..\,j]$, i.e. $j - i + 1 = LynS[j]$. Position j on y is the rightmost leaf of the subtree and $LynS[j]$ is its width.

It is known that y, $|y| > 1$, is of the form $x^k zb$ where x is a Lyndon word of length $per = period(x^k z)$, $k > 0$, z is a proper prefix of x and b is a letter greater than letter a following z in x (za is a prefix of x) [7].

The construction of $\mathcal{L}(y)$ is achieved with the help of the table $LynS$ of y. It is done by processing y from left to right building first $\mathcal{L}(x)$ and reproducing that tree or part of it up to z. The picture displays the subtrees built for the word $(\texttt{ababb})^2\texttt{aba}$.

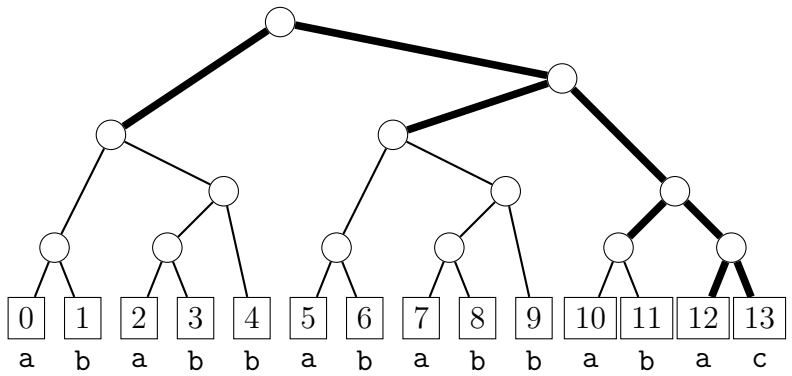


The main step of the procedure, in addition to computing $LynS$ by Algorithm LyndonSuffix, is to aggregate partial Lyndon trees when processing the last letter b of y; to create the final tree as a bundle of all subtrees. In fact, this step is also carried out when dealing with $x^k z$ at each position j for which $LynS[j] > 1$. In order to aggregate the subtrees, the second property of Proposition 3 is applied iteratively, processing the subtrees from right to left. Instruction of this step appear at lines 10-15 in Algorithm LeftLyndonTree, in which left$[q]$ and right$[q]$ are respectively the left and right children of the internal node q of the tree.

The process of bundling can be viewed as a translation into the tree structure of the proof of the key feature of Algorithm LyndonWordPrefix. Even so the latter deals with this process in constant time, which is not the case here, the iteration of bundling instructions does not affect the asymptotic running time of the present algorithm.

```
LeftLyndonTree(y Lyndon word of length n)
 1  (LynS[0], root[0]) ← (1, 0)
 2  (per, i) ← (1, 0)
 3  for j ← 1 to n − 1 do
 4        root[j] ← j
 5        if y[j] ≠ y[i] then          ▷ y[j] > y[i] = y[j − per]
 6              LynS[j] ← j + 1
 7              (per, i) ← (j + 1, 0)
 8        else LynS[j] ← LynS[i]
 9              i ← i + 1 mod per
10        (ℓ, k) ← (1, j − 1)
11        while ℓ < LynS[j] do
12              q ← new node ≥ n
13              (left[q], right[q]) ← (root[k], root[j])
14              root[j] ← q
15              (ℓ, k) ← (ℓ + LynS[k], k − LynS[k])
16  return root[n − 1]
```

The picture below shows thick links and nodes created by the final round of instructions at lines 10-15 in Algorithm LeftLyndonTree.



Proposition 5. *Algorithm* LeftLyndonTree *builds the left Lyndon tree of a Lyndon word of length n in time $O(n)$ in the letter-comparison model.*

Proof. All instructions inside the **for** loop are executed in constant time except the **while** loop. In addition, since each execution of instructions in the **while** loop takes constant time and leads to the creation of an internal node of the final tree, twinned with the fact that there are exactly $n - 1$ internal nodes, the total running time is $O(n)$.

4 Sorting prefixes

We show that Algorithm LeftLyndonTree can be adapted to sort the prefixes of a Lyndon word according to the infinite ordering $\prec$. This is a consequence of Theorem 7 below.

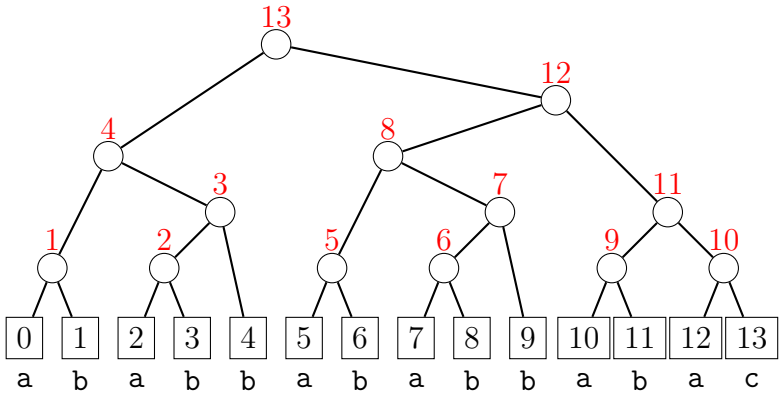
An internal node p of the left Lyndon tree of a Lyndon word y is the root of a Lyndon subtree associated with a Lyndon factor w, $|w| > 1$ of y. This factor is

obtained by concatenating two consecutive occurrences of Lyndon factors u and v. If the concerned occurrence of w ends at position j on y, node p is identified with the prefix of y ending at position j. The correspondence between internal nodes of the tree and proper non-empty prefixes of y is clearly one-to-one because internal nodes are identified with interpositions, pairs $(i, i+1)$ of positions on y.

Labelling internal nodes with the $\prec$-ranks of their associated prefixes transforms the tree into a heap, i.e. ranks are increasing from leaves to the root. The relation between the infinite order and left Lyndon trees is established by the next result [6].

Theorem 6 (Dolce, Restivo, Reutenauer, 2019). *The tree of internal nodes of the left Lyndon tree of a Lyndon word y in which nodes are labelled by the ranks of proper prefixes of y sorted according to the infinite order is the Cartesian tree of the ranks.*

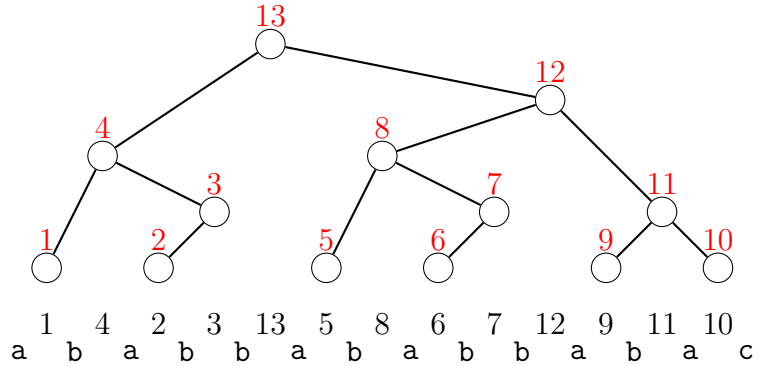
The following picture shows the left Lyndon tree of $y_0 =$ `ababbababbabac` and the $\prec$-rank labels of its internal nodes.



Denoting a prefix of y_0 by the position of its last letter (length minus 1), the table below shows $\prec$-ranks of proper non-empty prefixes of the word and their sorted list, inverse of Rank. The sorted list is `a` $\prec$ `aba` $\prec$ `abab` $\prec$ `ab` $\prec$ `ababba` $\prec$ `ababbaba` $\prec$ `ababbabab` $\prec$ `ababbab` $\prec$ `ababbababba` $\prec$ `ababbababbaba` $\prec$ `ababbababbab` $\prec$ `ababbababb` $\prec$ `ababb`.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$y[j]$	a	b	a	b	b	a	b	a	b	b	a	b	a	c
rank$[j]$	1	4	2	3	13	5	8	6	7	12	9	11	10	
prefix list	0	2	3	1	5	7	8	6	10	12	11	9	4	

The tree below is the Cartesian tree of prefix $\prec$-ranks.



Note that Algorithm LeftLyndonTree processes the resulting tree in left-to-right post-order. The next result links this order to prefix $\prec$-ranks.

Theorem 7. *Algorithm* LeftLyndonTree *on a Lyndon word y of length $n > 1$, creates and processes internal nodes of the tree in the order of their corresponding prefix ranks according to the ordering $\prec$.*

Proof. A Lyndon word that is not reduced to a single letter, y is of the form $x^k zb$ where x is a Lyndon word of length $period(x^k z)$, $k > 0$, z is a proper prefix of x and b is a letter greater than letter a following prefix z in x [7].

Algorithm LeftLyndonTree processes nodes of the Lyndon trees $\mathcal{L}(y)$ as follows. Initially, it builds $\mathcal{L}(x)$ and Lyndon trees of the next occurrences of x in a left-to-right manner. It continues with the tree related to z. Eventually during the last bundling (run of instructions at lines 10-15) the algorithm builds $\mathcal{L}(zb)$ and follows with the nodes corresponding to the concatenations $x \cdot zb$, $x \cdot xzb$, ..., $x \cdot x^{k-1}zb$ in that order.

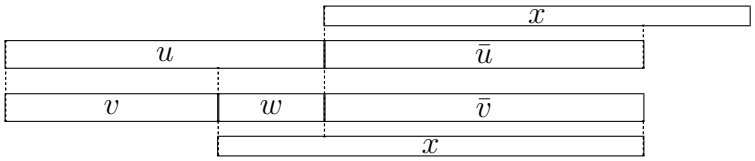
The statement is proved by induction on the length of the period $|x|$ of $x^k z$. If is $|x| = 1$, x is reduced to a single letter and y is of the form $a^k b$ for two letters a and b with $a < b$. Nodes associated with prefixes a^k, a^{k-1}, ..., a are processed in this order, which matches the $\prec$-order of prefixes, $a^k \prec a^{k-1} \prec \cdots \prec a$, as expected.

We then assume $|x| > 1$ and consider disjoint groups of non-empty proper prefixes of y. For $e = 0, 1, \ldots, k$, let

$$P_e = \{x^e u \text{ prefix of } y \mid e|x| < |x^e u| < \min\{(e+1)|x|, |y|\}\}.$$

The main part of the proof relies on three claims that we prove first.

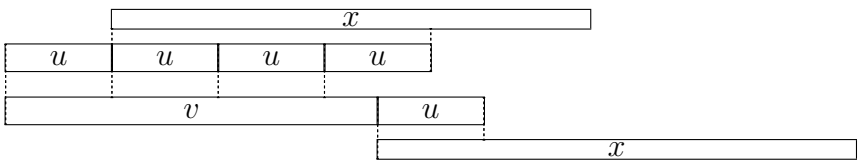
Claim 1: prefixes $x^e u \in P_e$, $0 < e \leq k$, are in the same relative $\prec$-order as prefixes $u \in P_0$. Let $u, v \in P_0$ with $u \prec v$ and let us show $x^e u \prec x^e v$ considering two cases.



Case $u^\infty = v^\infty$ and $|u| > |v|$. By the Periodicity lemma u, v and $v^{-1}u$ are powers of the same word. Let $w = v^{-1}u$, $\bar{v} = w^{-1}x$ and $\bar{u}$ the prefix of x of length $|\bar{v}|$ (see picture). Since x is a Lyndon word, $\bar{u} < x < \bar{v}$, which implies $ux < vx$ because w is a prefix of x. Therefore we have $(x^e u)^\infty < (x^e v)^\infty$, that is, $x^e u \prec x^e v$.

Case $u^\infty < v^\infty$. Assume u is shorter than v and let h be the largest exponent for which u^h is a prefix of v. It is a proper prefix because $u^\infty < v^\infty$ and then $w = (u^h)^{-1}v$ is not empty.

If $|u| \leq |w|$, we have $u \ll w$, which implies $ux \ll vx$ and $(x^e u)^\infty < (x^e v)^\infty$, that is, $x^e u \prec x^e v$. (This case can happen if for example, $u = \texttt{ab}$, $v = \texttt{abababbbb}$, then $w = \texttt{bbb}$ which means $|u| < |w|$)

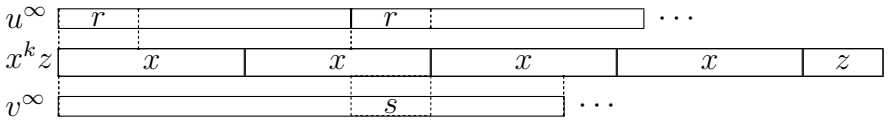


If $|u| > |w|$, v is a proper prefix of u^{h+1} but u^{h+1} shorter than vu cannot be a prefix of it due to the Periodicity lemma applied on periods $|u|$ and $|v|$ of u^{h+1}. Then

$u \ll wu$ and since u is a prefix of x it implies $ux \ll vx$ and $(x^e u)^\infty < (x^e v)^\infty$, that is, $x^e u \prec x^e v$ as before.

The situation in which u is longer than v is fairly symmetric and treated similarly. Therefore again $u \prec v$ implies $x^e u \prec x^e v$, which proves the claim.

Claim 2: prefixes in P_e are $\prec$-smaller than prefixes in P_f when $0 \le e < f \le k$. Let $u \in P_e$ and $v \in P_f$. We have to compare u and v according to $\prec$, that is, to compare u^∞ and v^∞.



When $e > 0$, u is longer than x. Let r be the prefix of u for which $|ur| = |x^{e+1}|$ (see picture in which $u \in P_1$ and $v \in P_2$) and s the suffix of x of the same length. Comparing u^∞ and v^∞ amounts to compare r and s, because u is a prefix of v. Since r is a prefix and s a suffix of the Lyndon word x, we have $r < s$ and even more, $r \ll s$, then $u^\infty < v^\infty$ and $u \prec v$.

When $e = 0$, u is shorter than x. Let h be the largest integer for which u^h is a prefix of x. It is a proper prefix because x is a Lyndon word and $w = (u^h)^{-1}x$ is not empty. As in the proof of previous claim, u^{h+1} cannot be prefix of xu that is a prefix of v. The same conclusion follows, that it, $u^{h+1} \ll vu$ and eventually $u \prec v$.

Claim 3: prefixes in P_e, $0 \le e \le k$, are $\prec$-smaller than prefixes x^f, $0 < f \le k$. To prove the claim, in view of the statement of Claim 2 and the fact $x^k \prec x^{k-1} \prec x$ by definition, it is enough to show that $P_k \prec x^k$. Note that if P_k is empty the proof can be shown with P_{k-1} instead, and if in addition $k = 1$ then we are left with an element in the proof of Claim 2.

Let $x^k u \in P_k$, $s = u^{-1}x$ and r the prefix of x of length $|s|$. As prefix and suffix of x, r and s satisfy $r < s$. Since $x^k ur < x^k us = x^{k+1}$ and r is a prefix of x, it results in $(x^k u)^\infty < x^\infty$ and eventually $x^k u \prec x^k$. This proves the claim.

To summarise, claims show

$$P_0 \prec P_1 \prec \cdots \prec P_k \prec x^k \prec x^{k-1} \prec \cdots \prec x.$$

Let us go back to induction. By induction hypothesis, the result holds for internal nodes of $\mathcal{L}(x)$ corresponding to prefixes in P_0.

Consider the next occurrences of x. Since the Lyndon suffix table for each of them is copied from that of prefix x due to the instruction at line 8 in Algorithm LeftLyndonTree, the Lyndon trees of all occurrences of x have the same structure. Therefore, both from the induction hypothesis and from Claim 1, the order in which internal nodes of the eth occurrence of x are processed and created matches the $\prec$-order of prefixes in P_e, for $0 < e \le k$.

The algorithm processes occurrences of x from left to right, which corresponds to the result of Claim 2. The treatment of zb is done at the beginning of the bundling run, which also corresponds to the fact that prefixes in P_k are $\prec$-larger than all prefixes that have been considered before.

Finally, the last part of the bundling creates nodes associated with x^k, x^{k-1}, ..., x in that order, which matches the order $x^k \prec x^{k-1} \prec \cdots \prec x$.

This ends the proof of the theorem.

A consequence of the theorem is that Algorithm LeftLyndonTree can be downgraded to compute directly the $\prec$-sorted list of non-empty proper prefixes of a Lyndon word. Dolce et al. [6] call this ordered list the prefix standard permutation. The following algorithm computes the prefix standard permutation.

```
PrefixStandardPermutation(y Lyndon word of length n)
 1  S ← ()
 2  (LynS[0], per, i) ← (1, 1, 0)
 3  for j ← 1 to n − 1 do
 4       if y[j] ≠ y[i] then          ▷ y[j] > y[i] = y[j − per]
 5            LynS[j] ← j + 1
 6            (per, i) ← (j + 1, 0)
 7       else LynS[j] ← LynS[i]
 8            i ← i + 1 mod per
 9       (k, m) ← (j − 1, 1)
10       while m < LynS[j] do
11            S ← S · (j − m)
12            m ← m + LynS[k]
13            k ← k − LynS[k]
14  return S
```

Corollary 8. *Sorting the proper non-empty prefixes of a Lyndon word of length n according to the infinite order $\prec$ can be carried out in time $O(n)$ in the letter-comparison model.*

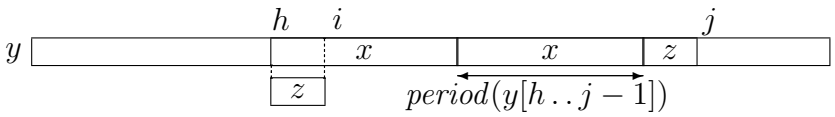
Proof. It essentially suffices to substitute the handling of the sequence to the processing of internal nodes of the Lyndon tree of the word in Algorithm LeftLyndonTree, which is equivalent to do a left-to-right post-order traversal of the tree. The change is realised by Algorithm PrefixStandardPermutation.

5 Lyndon forest

When the non-empty word y is not a Lyndon word, the above process can be carried out on each factor of its Lyndon factorisation, a decreasing list of Lyndon factors of the word. Lyndon factorisation of y is a list $x_1, x_2, \ldots, x_k$ for which both $x_1 x_2 \cdots x_k = y$ and $x_1 \geq x_2 \geq \cdots \geq x_k$. This factorisation is unique (see [10, Chen-Fox-Lyndon theorem]).

The factorisation and its algorithm by Duval [7] is the guiding thread of previous algorithms. The Lyndon forest of word y is the list of Lyndon trees $\mathcal{L}(x_1)$, $\mathcal{L}(x_2)$, …, $\mathcal{L}(x_k)$. Its computation uses again the Lyndon suffix table of the word, computed by Algorithm LongestLyndonSuffix whose input is not necessarily a Lyndon word.

It is a revision of Algorithm LyndonSuffix. The change stands in instructions on lines 4-7. They reset the computation to the suffix $y[h\,..\,n-1]$ of the input after the factorisation of the prefix $y[0\,..\,h-1]$ is definitely achieved.



```
LongestLyndonSuffix(y non-empty word of length n)
 1  LynS[0] ← 1
 2  (per, h, i, j) ← (1, 0, 0, 1)
 3  while j < n do
 4      if y[j] < y[i] then
 5          h ← j − (i − h)
 6          LynS[h] ← 1
 7          (per, i, j) ← (1, h, h + 1)
 8      elseif y[j] > y[i] then
 9          LynS[j] ← j − h + 1
10          j ← j + 1
11          (per, i) ← (j − h, h)
12      else LynS[j] ← LynS[i]
13          (i, j) ← (h + (i − h + 1 mod per), j + 1)
14  return LynS
```

Proposition 9. *Algorithm* LongestLyndonSuffix *computes the Lyndon suffix table of a word of length $n > 0$ in time $O(n)$ in the letter-comparison model.*

Proof. Let us consider values of expression $h+j$ and show they form a strictly increasing sequence after each iteration in the **while** loop. This claim holds if the condition at line 4 is false, because j is incremented by at least one unit (on line 10 or on line 13) and h remains unchanged. This claim also holds if the condition at line 4 is true, because h is incremented by at least $period(y[h\,..\,j-1])$ while j is decremented by less than the same value.

Since $h + j$ goes from 1 to at most $2n - 1$ the running time is $O(n)$.

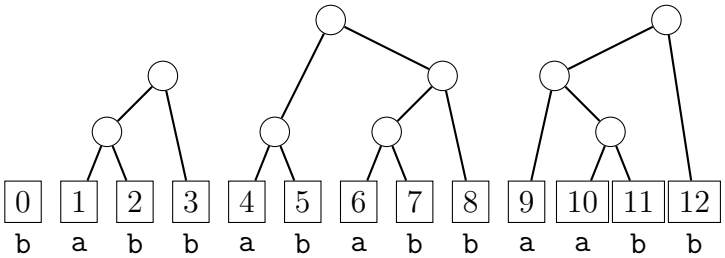
Note the Lyndon factorisation of a word y can be retrieved from its $LynS$ table by sequentially tracing back the starting position of the previous factor, starting from $|y|$. The list of starting positions of factors, in reverse order, is $i_k = |y| - LynS[|y|-1]$, $i_{k-1} = i_k - LynS[i_{k-1} - 1], \ldots, 0$.

Example 10. The Lyndon suffix table of $y_1 =$ `babbababbaabb` is as follows.

j	0	1	2	3	4	5	6	7	8	9	10	11	12
$y[j]$	b	a	b	b	a	b	a	b	b	a	a	b	b
$LynS[j]$	1	1	2	3	1	2	1	2	5	1	1	3	4

Starting positions of factors of its Lyndon factorisation are $9 = 13 - LynS[12]$, $4 = 9 - LynS[8]$, $1 = 4 - LynS[3]$, $0 = 1 - LynS[0]$. The factorisation is `b·abb·ababb·aabb`.

The following depicts the Lyndon forest corresponding to y_1.



Algorithm LeftLyndonForest is merely adapted from the previous algorithm in order to manage Lyndon tree constructions of factors of the Lyndon factorisation while computing the latter. The next proposition is a direct consequence of Proposition 9.

Proposition 11. *Algorithm* LeftLyndonForest *computes the Lyndon forest of a word of length $n > 0$ in time $O(n)$ in the letter-comparison model.*

```
LeftLyndonForest(y non-empty word of length n)
 1  (LynS[0], root[0]) ← (1, 0)
 2  (per, h, i, j) ← (1, 0, 0, 1)
 3  while j < n do
 4      root[j] ← j
 5      if y[j] < y[i] then
 6          h ← j − (i − h)
 7          LynS[h] ← 1
 8          (per, i, j) ← (1, h, h + 1)
 9      elseif y[j] > y[i] then
10          LynS[j] ← j − h + 1
11          j ← j + 1
12          (per, i) ← (j − h, h)
13      else LynS[j] ← LynS[i]
14          (i, j) ← (h + (i − h + 1 mod per), j + 1)
15      ▷ Bundle
16      (p, m, k) ← (root[j], 1, j − 1)
17      while m < LynS[j] do
18          q ← new node ≥ n
19          (left[q], right[q]) ← (root[k], p)
20          (p, m) ← (q, m + LynS[k])
21          k ← k − LynS[k]
22  return root[n − 1]
```

6 Conclusions

In this paper, algorithm LyndonSuffix computes, for a given Lyndon word, its Lyndon suffix table. The Lyndon suffix table is an essential part of algorithm LeftLyndonTree which constructs the left Lyndon tree of a Lyndon word in linear time. We further investigated the prefix standard permutation, initially introduced by Dolce et al. [6], and its relation to the left Lyndon tree. This study resulted in a linear-time algorithm for computing prefix standard permutation in the letter-comparison model. In addition, we exhibited a strong connection between the prefix ranks and the left Lyndon tree. This connection dictates that the order in which the internal nodes of the left Lyndon tree are created and processed coincides with that of the prefix ranks according to infinite ordering.

We finally endeavoured to design a linear-time algorithm LyndonForest which computes the Lyndon forest of a given word. This process entailed modifications

of algorithm LyndonSuffix to create algorithm LongestLyndonSuffix, which enables us to construct the Lyndon suffix table of also non-Lyndon words.

Many interesting questions remain, for example, is there a connection between runs and the internal nodes of the Lyndon forest? Is there a relation between the left and the right Lyndon trees?

References

1. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta: *The "runs" theorem.* SIAM J. Comput., 46(5) 2017, pp. 1501–1514.
2. M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen: *The maximal number of cubic runs in a word.* J. Comput. Syst. Sci., 78(6) 2012, pp. 1828–1836.
3. M. Crochemore, T. Lecroq, and W. Rytter: *One Twenty Five Problems in Text Algorithms*, Cambridge University Press, 2020, In press.
4. M. Crochemore and L. M. S. Russo: *Cartesian and Lyndon trees.* Theoretical Computer Science, 806 February 2020, pp. 1–9.
5. F. Dolce, A. Restivo, and C. Reutenauer: *On generalized lyndon words.* Theor. Comput. Sci., 777 2019, pp. 232–242.
6. F. Dolce, A. Restivo, and C. Reutenauer: *Some variations on Lyndon words.* CoRR, abs/1904.00954 2019.
7. J. Duval: *Factorizing words over an ordered alphabet.* J. Algorithms, 4(4) 1983, pp. 363–381.
8. C. Hohlweg and C. Reutenauer: *Lyndon words, permutations and trees.* Theor. Comput. Sci., 307(1) 2003, pp. 173–178.
9. R. M. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in 40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA, IEEE Computer Society, 1999, pp. 596–604.
10. M. Lothaire: *Combinatorics on Words*, Addison-Wesley, 1983, Reprinted in 1997.
11. R. C. Lyndon: *On Burnside problem i.* Trans. Amer. Math. Soc., 77 1954, pp. 202–215.
12. V. A. Ufnarovskij: *Combinatorial and asymptotic methods in algebra*, in Algebra VI: Combinatorial and Asymptotic Methods of Algebra. Non-Associative Structures, A. Kostrikin and I. Shafarevich, eds., vol. 57 of Encyclopaedia of Mathematical Sciences, Springer, Berlin, 2011, pp. 1–196.
13. G. Viennot: *Algèbres de Lie libres et monoïdes libres*, vol. 691 of Lecture Notes in Mathematics, Springer-Verlag, Berlin, 1978.

On Arithmetically Progressed Suffix Arrays*

Jacqueline W. Daykin[1], Dominik Köppl[2], David Kübel[3], and Florian Stober[4]

[1] Department of Computer Science, Aberystwyth University, UK; Department of Information Science, Stellenbosch University, South Africa
jwd6@aber.ac.uk; jackie.daykin@gmail.com,
[2] Kyushu University, Japan Society for Promotion of Science, Japan
dominik.koeppl@inf.kyushu-u.ac.jp,
[3] University of Bonn, Institute of Computer Science, Germany
kuebel@cs.uni-bonn.de,
[4] University of Stuttgart, Institute for Formal Methods of Computer Science (FMI), Germany
florian.stober@t-online.de

Abstract. We characterize those strings whose suffix arrays are based on arithmetic progressions, in particular, arithmetically progressed permutations where all pairs of successive entries of the permutation have the same difference modulo n. We show that an arithmetically progressed permutation P coincides with the suffix array of a unary, binary, or ternary string. We further analyze the conditions of a given P under which we can find a uniquely defined string over either a binary or ternary alphabet having P as its suffix array. These results give rise to numerous future research directions.

Keywords: arithmetic progression, suffix array, string combinatorics

1 Introduction

The integral relationship between the suffix array [18] (SA) and Burrows-Wheeler transform [4] (BWT) is explored in [1], which also illustrates the versatility of the BWT beyond its original motivation in lossless block compression [4]. BWT applications include compressed index structures using backward search pattern matching, multimedia information retrieval, bioinformatics sequence processing, and it is at the heart of the bzip2 suite of text compressors. By its association with the BWT, this also indicates the importance of the SA data structure and hence our interest in exploring its combinatorial properties.

These combinatorial properties can be useful when checking the performance or integrity of string algorithms or data structures on string sequences in testbeds when properties of the employed string sequences are well understood. In particular, due to current trends involving massive data sets, indexing data structures need to work in external memory (e.g., [3]), or on distributed systems (e.g. [11]). For devising a solution adaptable to these scenarios, it is crucial to test whether the computed index (consisting of the suffix array or the BWT, for instance) is correctly stored on the hard disk or on the computing nodes, respectively. This test is more cumbersome than in the case of a single machine working with RAM. One way to test is to compute the index for an instance, whose index shape can be easily verified. For example, one could check the validity of the computed BWT on a Fibonacci word since the shape of its BWT is known [20,5,23].

Other studies based on Fibonacci words are the suffix tree [22] or the Lempel-Ziv 77 (LZ77) factorization [2]. In [16], the suffix array and its inverse of each even

* This work was initiated at the open problems session of StringMasters colocated with IWOCA 2019 (International Workshop on Combinatorial Algorithms).

Jacqueline W. Daykin, Dominik Köppl, David Kübel, Florian Stober: On Arithmetically Progressed Suffix Arrays, pp. 96–110.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9

Fibonacci word is studied as an arithmetic progression. In this study, the authors, like many at that time, did not append the artificial $ delimiter (also known as a sentinel) to the input string, thus allowing suffixes to be prefixes of other suffixes. This small fact makes the definition of $\mathsf{BWT}_\mathsf{T}[i] = \mathsf{T}[\mathsf{SA}_\mathsf{T}[i] - 1]$ for a string T with suffix array SA_T incompatible with the traditional BWT defined on the BWT matrix, namely the lexicographic sorting of all cyclic rotations of the string T.

For instance, $\mathsf{SA}_{\mathtt{bab}} = [2, 3, 1]$ with $\mathsf{BWT}_{\mathtt{bab}} = \mathtt{bab}$, while $\mathsf{SA}_{\mathtt{bab\$}} = [4, 2, 3, 1]$ and $\mathsf{BWT}_{\mathtt{bab\$}} = \mathtt{bba\$}$ with $\mathtt{\$} < \mathtt{a} < \mathtt{b}$. However, the traditional BWT constructed by reading the last characters of the lexicographically sorted cyclic rotations $[\mathtt{ab}\boldsymbol{\mathtt{b}}, \mathtt{ba}\boldsymbol{\mathtt{b}}, \mathtt{bb}\boldsymbol{\mathtt{a}}]$ of $\mathtt{bab}$ yields $\mathtt{bba}$, which is equal to $\mathsf{BWT}_{\mathtt{bab\$}} = \mathtt{bba\$}$ after removing the $ character.

Note that not all strings are in the BWT image. An $O(n \log n)$-time algorithm is given by Giuliani et al. [13] for identifying all the positions in a string S that a $ can be inserted into so that S becomes the BWT image of a string ending with $.

Despite this incompatibility in the suffix array based definition of the BWT, we can still observe a regularity for even Fibonacci words [16, Sect. 5]. Similarly, both methods for constructing the BWT are compatible when the string T consists of a Lyndon word. The authors of [16, Remark 1] also observed similar characteristics for other, more peculiar string sequences. For the general case, the $ delimiter makes both methods equivalent, however the suffix array approach is typically preferred as it requires $\Theta(n)$ time [21] compared to $\Theta(n^2)$ with the BWT matrix method [4]. By utilizing combinatorial properties of the BWT, an in-place algorithm is given by Crochemore et al. [8], which avoids the need for explicit storage for the suffix sort and output arrays, and runs in $O(n^2)$ time using $O(1)$ extra memory (apart from storing the input text). Köppl et al. [15, Sect. 5.1] adapted this algorithm to compute the traditional BWT within the same space and time bounds.

Up to now, it has remained unknown whether we can formulate a class of string sequences for which we can give the shape of the suffix array as an arithmetic progression (independent of the $ delimiter). With this article, we catch up on this question, and establish a correspondence between strings and suffix arrays generated by arithmetic progressions. Calling a permutation of integers $[1..n]$ *arithmetically progressed* if all pairs of successive entries of the permutation have the same difference modulo n, we show that an arithmetically progressed permutation coincides with the suffix array of a unary, binary or ternary string. We analyze the conditions of a given arithmetically progressed permutation P under which we can find a uniquely defined string T over either a unary, a binary, or ternary alphabet having P as its suffix array.

The simplest case is for unary alphabets: Given the unary alphabet $\Sigma := \{\mathtt{a}\}$ and a string T of length n over Σ, $\mathsf{SA}_\mathsf{T} = [n, n-1, \ldots, 1]$ is an arithmetically progressed permutation with ratio $-1 \equiv n - 1 \mod n$.

For the case of a binary alphabet $\{\mathtt{a}, \mathtt{b}\}$, several strings of length n exist that solve the problem. Trivially, the solutions for the unary alphabet also solve the problem for the binary alphabet. However, studying those strings of length n whose suffix array is $[n, n-1, \ldots, 1]$, there are now multiple solutions: each $\mathsf{T} = \mathtt{b}^r\mathtt{a}^s$ with $r, s \in [0..n]$ such that $r + s = n$ has this suffix array. Similarly, $\mathsf{T} = \mathtt{a}^{n-1}\mathtt{b}$ has the suffix array $\mathsf{SA}_\mathsf{T} = [1, 2, \ldots, n]$, which is an arithmetically progressed permutation with ratio 1.

In what follows, we present a comprehensive analysis of strings whose suffix arrays are arithmetically progressed permutations (under the standard lexicographic order). In practice, such knowledge can reduce the $O(n)$ space for the suffix array to $O(1)$.

The structure of the paper is as follows. In Section 2 we give the basic definitions and background, and also deal with the elementary case of a unary alphabet. The

main results are presented in Section 3: we justify the need for coprimality, then cover ternary and binary alphabets followed by considering inverse permutations, and finally link the binary characterization to Fibonacci words. We conclude in Section 4 and propose a list of open problems and research directions, showing there is plenty of scope for further investigation. We proceed to the foundational concepts.

2 Preliminaries

Let Σ be an alphabet with size $\sigma := |\Sigma|$. An element of Σ is called a *character*[1]. Let Σ^+ denote the set of all nonempty finite strings over Σ. The *empty string* of length zero is denoted by ε; we write $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. Given an integer $n \geq 1$, a *string*[2] of length n over Σ takes the form $\mathsf{T} = t_1 \cdots t_n$ with each $t_i \in \Sigma$. We write $\mathsf{T} = \mathsf{T}[1..n]$ with $\mathsf{T}[i] = t_i$. The length n of a string T is denoted by $|\mathsf{T}|$. If $\mathsf{T} = \mathsf{uwv}$ for some strings $\mathsf{u}, \mathsf{w}, \mathsf{v} \in \Sigma^*$, then u is a *prefix*, w is a *substring*, and v is a *suffix* of T; we say u (resp. w and v) is *proper* if $\mathsf{u} \neq \mathsf{T}$ (resp. $\mathsf{w} \neq \mathsf{T}$ and $\mathsf{v} \neq \mathsf{T}$). We say that a string T of length n has *period* $p \in [1..n-1]$ if $\mathsf{T}[i] = \mathsf{T}[i+p]$ for every $i \in [1..n-p]$ (note that we allow periods larger than $n/2$). If $\mathsf{T} = \mathsf{uv}$, then vu is said to be a *cyclic rotation* of T. A string that is both a proper prefix and a proper suffix of a string $\mathsf{T} \neq \varepsilon$ is called a *border* of T; a string is *border-free* if the only border it has is the empty string ε.

If Σ is a totally ordered alphabet with order $<$, then this order $<$ induces the *lexicographic ordering* $\prec$ on Σ^* such that $\mathsf{u} \prec \mathsf{v}$ for two strings $\mathsf{u}, \mathsf{v} \in \Sigma^*$ if and only if either u is a proper prefix of v, or $\mathsf{u} = \mathsf{r}a\mathsf{s}$, $\mathsf{v} = \mathsf{r}b\mathsf{t}$ for two characters $a, b \in \Sigma$ such that $a < b$ and for some strings $\mathsf{r}, \mathsf{s}, \mathsf{t} \in \Sigma^*$. In the following, we select a totally ordered alphabet Σ having three characters $\mathsf{a}, \mathsf{b}, \mathsf{c}$ with $\mathsf{a} < \mathsf{b} < \mathsf{c}$.

A string T is a *Lyndon word* if it is strictly least in the lexicographic order among all its cyclic rotations [17]. For instance, `abcac` and `aaacbaabaaacc` are Lyndon words, while the string `aaacbaabaaac` with border `aaac` is not.

For the rest of the article, we take a string T of length $n \geq 2$. The suffix array $\mathsf{SA} := \mathsf{SA}_\mathsf{T}[1..n]$ of T is a permutation of the integers $[1..n]$ such that $\mathsf{T}[\mathsf{SA}[i]..n]$ is the i-th lexicographically smallest suffix of T. We denote with ISA its inverse, i.e., $\mathsf{ISA}[\mathsf{SA}[i]] = i$. By definition, ISA is also a permutation. The string BWT with $\mathsf{BWT}[i] = \mathsf{T}[\mathsf{SA}[i] - 1 \mod n]$ is the (SA-based) BWT of T.

The Fibonacci sequence is a sequence of binary strings $\{\mathsf{F}_m\}_{m \geq 1}$ with $\mathsf{F}_1 := \mathsf{b}$, $\mathsf{F}_2 := \mathsf{a}$, and $\mathsf{F}_m := \mathsf{F}_{m-1}\mathsf{F}_{m-2}$. Then F_m and $f_m := |\mathsf{F}_m|$ are called the m-th *Fibonacci word* and the m-th *Fibonacci number*, respectively.

The focus of this paper is on arithmetic progressions. An *arithmetic progression* is a sequence of numbers such that the differences between all two consecutive terms are of the same value: Given an arithmetic progression $\{p_i\}_{i \geq 1}$, there is an integer $k \geq 1$ such that $p_{i+1} = p_i + k$ for all $i \geq 1$. We call k the *ratio* of this arithmetic progression. Similarly to sequences, we can define permutations that are based on arithmetic progressions: An *arithmetically progressed permutation* with ratio $k \in [1..n-1]$ is an array $P := [p_1, \ldots, p_n]$ with $p_{i+1} = p_i + k \mod n$ for all $i \in [1..n]$, where we stipulate that $p_{n+1} := p_1$.[3] Here $x \bmod n := x$ if $x \leq n$ and $x - n \mod n$ otherwise for an integer $x \geq 1$. In what follows, we want to study (a) strings whose suffix arrays are

[1] Also known as *letter* or *symbol* in the literature.
[2] Also known as *word* in the literature.
[3] We can also support negative values of k: Given a negative $k < 0$, we exchange it with $k' := n - k \bmod n \in [1..n]$ and use k' instead of k.

arithmetically progressed permutations, and (b) the shape of these suffix arrays. For a warm-up, we start with the unary alphabet:

Theorem 1. *Given the unary alphabet* $\{\mathtt{a}\}$*, the suffix array of a string of length* n *over* $\{\mathtt{a}\}$ *is uniquely defined by the arithmetically progressed permutation* $[n, n-1, \ldots, 1]$ *with ratio* $n-1$*.*

Conversely, given the arithmetically progressed permutation $P = [n, n-1, \ldots, 1]$, we want to know the number of strings from a general totally ordered alphabet $\Sigma = [1..\sigma]$ with the natural order $1 < 2 < \cdots < \sigma$, having P as their suffix array. For that, we fix a string T of length n with $\mathsf{SA}_\mathsf{T} = P$. Let $s_j \geq 0$ be the number of occurrences of the character $j \in \Sigma$ appearing in T. Then $\sum_{j=1}^{\sigma} s_j = n$. By construction, each character j has to appear after all characters k with $k > j$. Therefore, $\mathsf{T} = \sigma^{s_\sigma}\sigma^{s_{\sigma-1}} \cdots 1^{s_1}$ such that the position of the characters are uniquely determined. In other words, we can reduce this problem to the classic stars and bars problem [10, Chp. II, Sect. 5] with n stars and σ bars, yielding $\binom{n+\sigma-1}{n}$ possible strings. Hence we obtain:

Theorem 2. *There are* $\binom{n+\sigma-1}{n}$ *strings of length* n *over an alphabet with size* σ *having the suffix array* $[n, n-1, \ldots, 1]$*.*

As described above, strings of Theorem 2 have the form $\sigma^{s_\sigma}\sigma^{s_{\sigma-1}} \cdots 1^{s_1}$. The BWT based on the suffix array is $1^{s_1-1}2^{s_2} \cdots \sigma^{s_\sigma}1$. For $s_1 \geq 2$, it does not coincide with the BWT based on the rotations since the lexicographically smallest rotation is $1^{s_1}\sigma^{s_\sigma} \cdots 2^{s_2}$, and hence the first entry of this BWT is 2. For $s_1 = 1$, the last character '1' acts as the dollar sign being unique and least among all characters, making both BWT definitions equivalent.

For the rest of the analysis, we omit the arithmetically progressed permutation $[n, n-1, \ldots, 1]$ of ratio $k = n-1$ as this case is complete. All other permutations (including those of ratio $k = n-1$) are covered in our following theorems whose results we summarized in Fig. 1.

p_1	k	Min. Size of Σ	Properties of Strings	Reference
1		2	unique, Lyndon word	Theorem 11
$k+1$		2	unique, period $(n-k)$	Theorem 11
n	$\neq (n-1)$	2	unique, period $(n-k)$	Theorem 11
	$= (n-1)$	1	trivially periodic	Theorem 2
$\notin \{1, k+1, n\}$		3	unique	Theorem 4

Figure 1. Characterization of strings whose suffix array is an arithmetic progression $P = [p_1, \ldots, p_n]$ of ratio k. The choice of p_1 determines the minimum size of the alphabet and whether a string is unique, periodic or a Lyndon word. The column *Min. Size of* Σ denotes the smallest possible size of Σ for which there exists such a string whose characters are drawn from Σ.

3 Arithmetically Progressed Suffix Arrays

We start with the claim that each arithmetically progressed permutation coincides with the suffix array of a string on a ternary alphabet. Subsequently, given an arithmetically progressed permutation P, we show that either there is precisely one string T

Rotation	T	P	$p_1-k-1 \bmod n$	$\mathsf{BWT_T}$
	1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8		
(1)	b a b b a b a c	[5, 2, 7| 4, 1, 6, 3| 8]	7	$\mathtt{b}^4\mathtt{ca}^3$
(2)	b a b a c b a c	[2, 7, 4| 1, 6, 3| 8, 5]	4	$\mathtt{b}^3\mathtt{c}^2\mathtt{a}^3$
(3)	a c b a c b a c	[7, 4, 1| 6, 3| 8, 5, 2]	1	$\mathtt{b}^2\mathtt{c}^3\mathtt{a}^3$
(4)	a c b a c a c c	[4, 1, 6| 3| 8, 5, 2, 7]	6	$\mathtt{bc}^4\mathtt{a}^3$
(5)	a b a b b a b b	[1, 6, 3| 8, 5, 2, 7, 4]	3	$\mathtt{b}^5\mathtt{a}^3$
(6)	c c a c c a c b	[6, 3| 8| 5, 2, 7, 4, 1]	8	$\mathtt{c}^5\mathtt{a}^2\mathtt{b}$
(7)	c c a c b c c b	[3| 8, 5| 2, 7, 4, 1, 6]	5	$\mathtt{c}^5\mathtt{ab}^2$
(8)	b a b b a b b a	[8, 5, 2| 7, 4, 1, 6, 3]	2	$\mathtt{b}^5\mathtt{a}^3$

Figure 2. T of Eq. 1 for each arithmetically progressed permutation P of length $n = 8$ with ratio $k = 5$, starting with $p_1 := P[1] = k = 5$. The permutation of the k-th row is the k-th cyclic rotation of the permutation P in the first row. The splitting of P into the subarrays is visualized by the | symbol. For (5) and (8), the alphabet is binary and the BWTs are the same. The strings of (3) and (8) are periodic with period $n - k$, since the last text position of each subarray is at most as large as $n - k = 3$ (*cf.* the proof of Theorem 11). For $i \in [1..n]$, $\mathsf{BWT_T}[i] = \mathsf{T}[P[i + n - k^{-1} \bmod n]] = \mathsf{T}[P[i + 3 \bmod n]]$ with $k^{-1} = k = 5$ defined in Section 3.4.

with $\mathsf{SA_T} = P$ whose characters are drawn from a *ternary* alphabet, or, if there are multiple candidate strings, then there is precisely one whose characters are drawn from a *binary* alphabet. For this aim, we start with the restriction on k and n to be coprime:

3.1 Coprimality

Two integers are *coprime*[4] if their greatest common divisor (gcd) is one. An *ideal* $k\mathbb{N} := \{ki\}_{i\in\mathbb{N}}$ is a subgroup of $([1..n], +)$. It *generates* $[1..n]$ if $|k\mathbb{N}| = n$, i.e., $k\mathbb{N} = [1..n]$. Fixing one element $P[1] \in k\mathbb{N}$ of an ideal $k\mathbb{N}$ generating $[1..n]$ induces an arithmetically progressed permutation $P[1..n]$ with ratio k by setting $P[i + 1] \leftarrow P[i] + k$ for every $i \in [1..n - 1]$. On the contrary, each arithmetically progressed permutation with ratio k induces an ideal $k\mathbb{N}$ (the induced ideals are the same for two arithmetically progressed permutations that are shifted). Consequently, there is no arithmetically progressed permutation with ratio k if k and n are not coprime since in this case $\{(ki) \bmod n \mid i \geq 1\} \subsetneq [1..n]$, from which we obtain:

Lemma 3. *The numbers k and n must be coprime if there exists an arithmetically progressed permutation of length n with ratio k.*

3.2 Ternary Alphabet

Given an arithmetically progressed permutation $P := [p_1, \ldots, p_n]$ with ratio k, we define the ternary string $\mathsf{T}[1..n]$ by splitting P right after the values $n - k$ and $(p_1 - k - 1) \bmod n$ into the three subarrays A, B, and C (one of which is possibly empty) such that $P = ABC$. Subsequently, we set

$$\mathsf{T}[p_i] := \begin{cases} \mathtt{a} & \text{if } p_i \in A, \text{ or} \\ \mathtt{b} & \text{if } p_i \in B, \text{ or} \\ \mathtt{c} & \text{if } p_i \in C. \end{cases} \tag{1}$$

Figure 2 gives an example of induced ternary/binary strings.

[4] Also known as *relatively prime* in the literature.

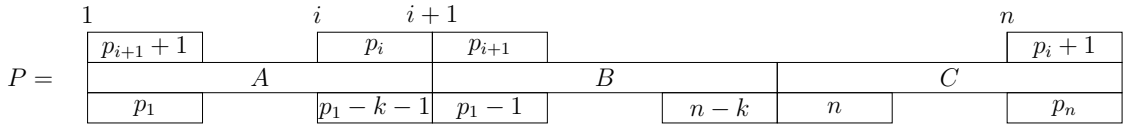


Figure 3. Setting of the proof of Theorem 4 with the condition $p_i + 1 = p_1 - k = p_n$. In the Figure we assume that the entry $p_1 - k - 1$ appears before $n - k$ in P.

Theorem 4. *Given an arithmetically progressed permutation* $P := [p_1, \ldots, p_n] \neq [n, n-1, \ldots, 1]$ *with ratio* k, $\mathsf{SA}_\mathsf{T} = P$ *for* T *defined in Eq. 1.*

Proof. Suppose we have constructed SA_T. Since $\mathtt{a} < \mathtt{b} < \mathtt{c}$, according to the above assignment of T, the suffixes starting with $\mathtt{a}$ lexicographically precede the suffixes starting with $\mathtt{b}$, which lexicographically precede the suffixes starting with $\mathtt{c}$. Hence, $\mathsf{SA}[1..|A|], \mathsf{SA}[|A|+1..|A|+|B|]$ and $\mathsf{SA}[|A|+|B|+1..n]$ store the same text positions as A, B, and C, respectively. Consequently, it remains to show that the entries of each subarray (A, B or C) are also sorted appropriately. Let p_i and p_{i+1} be two neighboring entries within the same subarray. Thus, $\mathsf{T}[p_i] = \mathsf{T}[p_{i+1}]$ holds, and the lexicographic order of their corresponding suffixes $\mathsf{T}[p_i..n]$ and $\mathsf{T}[p_{i+1}..n]$ is determined by comparing the subsequent positions, starting with $\mathsf{T}[p_i + 1]$ and $\mathsf{T}[p_{i+1} + 1]$. Since we have $(p_{i+1} + 1) - (p_i + 1) = p_{i+1} - p_i = k$, we can recursively show that these entries remain in the same order next to each other in the suffix array until either reaching the last array entry or a subarray split, that is, (1) $p_i + 1 = p_1 - k$ or (2) $p_i = n - k$.

1. When $p_i + 1$ becomes $p_1 - k \mod n = p_n$ (the last entry in SA), p_{i+1} is in the subsequent subarray of the subarray of $p_i = p_1 - k - 1$ (remember that A or B ends directly after $p_1 - k - 1$, *cf.* Fig. 3). Hence $\mathsf{T}[p_i] < \mathsf{T}[p_{i+1}]$, and $\mathsf{T}[p_i..n] \prec \mathsf{T}[p_{i+1}..n]$.
2. The split at the value $n - k$ ensures that when reaching $p_i = n - k$ and $p_{i+1} = n$, we can stop the comparison here as there is no character following $\mathsf{T}[p_{i+1}]$. The split here ensures that we can compare the suffixes $\mathsf{T}[n-k..n]$ and $\mathsf{T}[n]$ by the characters $\mathsf{T}[n-k] < \mathsf{T}[n]$. If we did not split here, $\mathsf{T}[n] = \mathsf{T}[n-k]$, and the suffix $\mathsf{T}[n]$ would be a prefix of $\mathsf{T}[n-k..n]$, resulting in $\mathsf{T}[n] \prec \mathsf{T}[n-k..n]$ (which yields a contradiction unless $p_n = n$).

To sum up, the text positions stored in each of A, B and C are in the same order as in SA_T since the $j-1$ subsequent text positions of each consecutive pair of entries p_i and p_{i+1} are consecutive in P for the smallest integer $j \in [1..n]$ such that $p_{i+1} + jk \in \{p_1 - 1, n\}$.

□

Knowing the suffix array of the ternary string T of Eq. 1, we can give a characterization of its BWT. We start with the observation that both BWT definitions (rotation based and suffix array based) coincide for the strings of Eq. 1 (but do not in general as highlighted in the introduction, *cf.* Fig. 4), and then continue with insights in how the BWT looks like.

Theorem 5. *Given an arithmetically progressed permutation* $P := [p_1, \ldots, p_n] \neq [n, n-1, \ldots, 1]$ *with ratio* k *and the string* T *of Eq. 1, the BWT of* T *defined on the BWT matrix coincides with the BWT of* T *defined on the suffix array.*

BWT matrix of `babbabac`:	BWT matrix of `ccaccacb`:	BWT matrix of `bbabbabb`:
`abacbabb`	`acbccacc`	`abbabbbb`
`abbabacb`	`accacbcc`	`abbbbabb`
`acbabbab`	`bccaccac`	`babbabbb`
`babacbab`	`cacbccac`	`babbbbab`
`babbabac`	`caccacbc`	`bbabbabb`
`bacbabba`	`cbccacca`	`bbabbbba`
`bbabacba`	`ccacbcca`	`bbbabbab`
`cbabbaba`	`ccaccacb`	`bbbbabba`

Figure 4. BWTs defined by the lexicographic sorting of all rotations of strings whose suffix arrays are cyclic rotations. This figure shows (from left to right) the BWT matrices of the strings of Rotation (1) and (6) of Fig. 2 as well as of Case (2) from Fig. 6. Reading the last column of a BWT matrix (whose characters are italic) from top down yields the BWT defined on the BWT matrix. While the BWT defined on the BWT matrix and the one defined by the suffix array coincides for the strings of Eq. 1 due to Theorem 5, this is not the case in general for the binary strings studied in Section 3.3, where we observe that $\mathsf{BWT}_{\texttt{bbabbabb}} = \texttt{bbbbbaab}$ defined by the suffix array differs from `bbbbbaba` (the last column on the right)

Proof. According to Theorem 4, $\mathsf{SA}_\mathsf{T} = P$, and therefore the BWT of T defined on the suffix array is given by $\mathsf{BWT}_\mathsf{T}[i] = \mathsf{T}[p_i - 1 \bmod n]$. The BWT matrix is constituted of the lexicographically ordered cyclic rotations of T. The BWT $\mathsf{BWT}_{\text{matrix}}$ based on the BWT matrix is obtained by reading the last column of the BWT matrix from top down (see Fig. 4). Formally, $\mathsf{BWT}_{\text{matrix}}[i] = \mathsf{T}[Q[i] - 1 \mod n]$, where $Q[i]$ is the starting position of the lexicographically i-th smallest rotation $\mathsf{T}[Q[i]..n]\mathsf{T}[1..Q[i]-1]$. We prove the equality $P = Q$ by showing that, for all $i \in [1..n-1]$, the rotation $R_i := \mathsf{T}[p_i..n]\mathsf{T}[1..p_i - 1]$ starting at $p_i = \mathsf{SA}_\mathsf{T}[i]$ is lexicographically smaller than the rotation $R_{i+1} := \mathsf{T}[p_{i+1}..n]\mathsf{T}[1..p_{i+1} - 1]$ starting at $p_{i+1} = \mathsf{SA}_\mathsf{T}[i+1]$. We do that by comparing both rotations R_i and R_{i+1} characterwise:

Let j be the first position where R_i and R_{i+1} differ, i.e., $R_i[j] \neq R_{i+1}[j]$ and $R_i[q] = R_{i+1}[q]$ for every $q \in [1..j)$.

First we show that $j \neq p_n - p_i + 1 \bmod n$ by a contradiction: Assuming that $j = p_n - p_i + 1 \bmod n$, we conclude that $j \neq 1$ by the definition of $i \in [1..n-1]$. Since k is the ratio of P, we have

$$R_i[j-1] = \mathsf{T}[p_i + j - 2 \bmod n] = \mathsf{T}[p_n - 1 \bmod n] = \mathsf{T}[p_1 - k - 1 \bmod n]$$

and $R_{i+1}[j-1] = \mathsf{T}[p_1 - 1 \bmod n]$. By Eq. 1, $p_1 - k - 1 \mod n$ and $p_1 - 1 \mod n$ belong to different subarrays of P, therefore $\mathsf{T}[p_1 - k - 1] \neq \mathsf{T}[p_1 - 1]$ and $R_i[j-1] \neq R_{i+1}[j-1]$, contradicting the choice of j as the first position where R_i and R_{i+1} differ.

This concludes that $j \leq n$ (hence, $R_i \neq R_{i+1}$) and $j \neq p_n - p_i + 1 \bmod n$. Hence, $R_i[j] = \mathsf{T}[p_i + j - 1 \bmod n]$ and $R_{i+1}[j] = \mathsf{T}[p_{i+1} + j - 1 \bmod n] = \mathsf{T}[p_i + j - 1 + k \bmod n]$ are characters given by two consecutive entries in SA_T, i.e., $\mathsf{SA}_\mathsf{T}[q] = p_i + j - 1 \bmod n$ and $\mathsf{SA}_\mathsf{T}[q+1] = p_i + j - 1 + k \bmod n$ for a $q \in [1..n-1]$. Thus $R_i[j] \leq R_{i+1}[j]$, and by definition of j we have $R_i[j] < R_{i+1}[j]$, leading finally to $R_i \prec R_{i+1}$. Hence, $Q = P$. □

Lemma 6. *Let $P := [p_1, \ldots, p_n] \neq [n, n-1, \ldots, 1]$ be an arithmetically progressed permutation with ratio k. Further, let $\mathsf{T}[1..n]$ be given by Eq. 1 such that $\mathsf{SA}_\mathsf{T} = P$ according to Theorem 4. Given that $p_t = p_1 - 1 - k \mod n$ for a $t \in [1..n]$, BWT_T is*

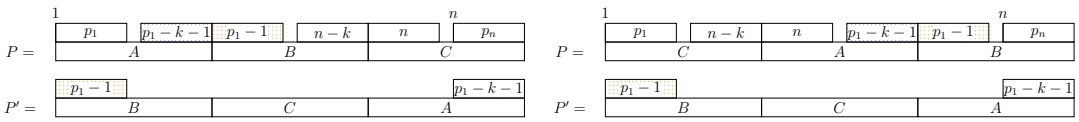


Figure 5. Setting of Eq. 1 with the distinction whether the entry $p_1 - k - 1$ appears before (left) or after (right) $n - k$ in P, yielding a different shape of the $\mathsf{BWT_T}$ defined as $\mathsf{BWT_T}_{[i]} = \mathsf{T}[P'[i]]$ with $P'[i] = \mathsf{SA_T}[i] - 1 \mod n$.

given by the t-th rotation of $\mathsf{T}[\mathsf{SA}[1]] \cdots \mathsf{T}[\mathsf{SA}[n]]$, *i.e.,* $\mathsf{BWT_T}[i] = \mathsf{T}[P[i + t \mod n]]$ *for* $i \in [1..n]$.

Proof. Since P is an arithmetically progressed permutation with ratio k then so is the sequence $P' := [p'_1, \ldots, p'_n]$ with $p'_i = p_i - 1 \mod n$. In particular, P' is a cyclic shift of P with $p'_n = p_1 - 1 - k \mod n$ because $p'_1 = p_1 - 1$. However, $p_1 - 1 - k$ is a split position of one of the subarrays A, B, or C, meaning that P' starts with one of these subarrays and ends with another of them (*cf.* Fig. 5). Consequently, there is a t such that $p_t = p'_n$, and we have the property that $\mathsf{BWT_T}$ with $\mathsf{BWT_T}[i] = \mathsf{T}[P'[i]]$ is the t-th rotation of $\mathsf{T}[\mathsf{SA}[1]] \cdots \mathsf{T}[\mathsf{SA}[n]]$. □

We will determine the parameter $t = n - k^{-1} \mod n$ after Eq. 3 in Section 3.4, where k^{-1} is defined such that $k \cdot k^{-1} \mod n = 1 \mod n$. With Lemma 6, we obtain the following corollary which shows that the number of runs in $\mathsf{BWT_T}$ for T defined in Eq. 1 are minimal:

Corollary 7. *For an arithmetically progressed permutation* $P := [p_1, \ldots, p_n] \neq [n, n-1, \ldots, 1]$ *and the string* T *defined by Eq. 1,* $\mathsf{BWT_T}$ *consists of exactly 2 runs if* T *is binary, while it consists of exactly 3 runs if* T *is ternary.*

Theorem 8. *Given an arithmetically progressed permutation* $P := [p_1, \ldots, p_n]$ *with ratio* k *such that* $p_1 \notin \{1, k+1, n\}$, *the string* T *given in Eq. 1 is unique.*

Proof. The only possible way to define another string T' would be to change the borders of the subarrays A, B, and C. Since $p_1 \notin \{1, n\}$, $n - k$ and n, as well as $p_1 - k - 1$ and $p_1 - 1$, are stored as a consecutive pair of text positions in P.

- If P is not split between its consecutive text positions $n-k$ and n, then $\mathsf{T}'[n-k] = \mathsf{T}'[n]$. Consequently, we have the contradiction $\mathsf{T}'[n] \prec \mathsf{T}'[n-k..n]$.
- If P is not split between its consecutive text positions $(p_1-k-1) \mod n$ and $(p_1-1) \mod n$, then $\mathsf{T}'[p_1 - k - 1 \mod n] = \mathsf{T}'[p_1 - 1 \mod n]$. Since $p_1 \neq k+1$, and $\mathsf{T}'[p_1 - k \mod n] = \mathsf{T}'[p_n] > \mathsf{T}'[p_1]$, this leads to the contradiction $\mathsf{T}'[(p_1 - k - 1 \bmod n)..n] \succ \mathsf{T}'[(p_1 - 1 \bmod n)..n]$, *cf.* Fig. 3.

□

Following this analysis of the ternary case we proceed to consider binary strings. A preliminary observation is given in Fig. 2, which shows, for the cases p_1 is 1 and n in Theorem 8, namely Rotations (5) and (8), that a rotation of $n - k$ in the permutation gives a rotation of one in the corresponding binary strings. We formalize this observation in the following lemma, drawing a connection between binary strings whose suffix arrays are arithmetically progressed and start with 1 or n.

Lemma 9. *Let $P := [p_1, \ldots, p_n]$ be an arithmetically progressed permutation with ratio k and $p_1 = 1$ for a binary string T over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ with $\mathsf{SA}_\mathsf{T} = P$. Suppose that the number of $\mathtt{a}$'s in T is m and that $\mathsf{T}' = \mathsf{T}[2] \cdots \mathsf{T}[n]\mathsf{T}[1]$ is the first rotation of T. Then $\mathsf{SA}_{\mathsf{T}'}$ is the m-th rotation of P with $\mathsf{SA}_{\mathsf{T}'}[1] = n$. Furthermore, $\mathsf{BWT}_\mathsf{T} = \mathsf{BWT}_{\mathsf{T}'}$.*

Proof. Since $p_1 = 1$, $\mathsf{T}[1] = \mathtt{a}$ and $\mathsf{T}[n] = \mathtt{b}$. In the following, we show that $P' = \mathsf{SA}_{\mathsf{T}'}$ for $P' := [p'_1, \ldots, p'_n] := [p_1 - 1 \bmod n, \ldots, p_n - 1 \bmod n]$ with $p'_1 = p_1 - 1 = n$ (since $\mathsf{T}'[n] = \mathtt{a}$). For that, we show that each pair of suffixes in SA_T is kept in the same relative order in P' (excluding $\mathsf{SA}_\mathsf{T}[1] = 1$):

Consider two text positions $p_i, p_j \in [p_2, \ldots, p_n]$ with $\mathsf{T}[p_i..n] = u_1 \cdots u_s \prec \mathsf{T}[p_j..n] = v_1 \cdots v_t$.

- If $u_h \neq v_h$ for the least $h \in [1..\min\{s,t\}]$, then $u_1 \cdots u_s\mathtt{a} \prec v_1 \cdots v_t\mathtt{a}$.
- Otherwise, $u_1 \cdots u_s$ is a proper prefix of $v_1 \cdots v_t = u_1 \cdots u_s v_{s+1} \cdots v_t$.
  - If $v_{s+1} = \mathtt{b}$, then $u_1 \cdots u_s\mathtt{a} \prec u_1 \cdots u_s\mathtt{b}v_{s+2} \cdots v_t\mathtt{a} = v_1 \cdots v_t\mathtt{a}$.
  - Otherwise ($v_{s+1} = \mathtt{a}$), $u_1 \cdots u_s\mathtt{a}$ is a proper prefix of $v_1 \cdots v_t\mathtt{a}$, and similarly $u_1 \cdots u_s\mathtt{a} \prec u_1 \cdots u_s\mathtt{a}v_{s+2} \cdots v_t\mathtt{a} = v_1 \cdots v_t\mathtt{a}$.

Hence the relative order of these suffixes given by $[p_2, \ldots, p_n]$ and $[p'_2, \ldots, p'_n]$ is the same. In total, we have $p'_i = p_i - 1 \mod n$ for $i \in [1..n]$, hence P' is an arithmetically progressed permutation with ratio k. Given the first m entries in P index represent all suffixes of T starting with $\mathtt{a}$, P' is the m-th rotation of P since $p'_1 = n$ is the $(m+1)$-th entry of P, i.e., the smallest suffix starting with $\mathtt{b}$ in T. Finally, since the strings T and T' are rotations of each other, their BWTs are the same. □

Like the parameter t of Lemma 6, we will determine the parameter m after Eq. 3 in Section 3.4.

3.3 Binary Alphabet

We start with the construction of a binary string from an arithmetically progressed permutation:

Theorem 10. *Given an arithmetically progressed permutation $P := [p_1, \ldots, p_n] \neq [n, n-1, \ldots, 1]$ with ratio k such that $p_1 \in \{1, k+1, n\}$, we can modify T of Eq. 1 to be a string over the binary alphabet $\{\mathtt{a}, \mathtt{b}\}$ with $\mathsf{SA}_\mathsf{T} = P$.*

Proof. If $p_1 = 1$, then P is split after the occurrences of the values $n-k$ and $-k = n-k \bmod n$, which gives only two non-empty subarrays. If $p_1 = n$, P is split after the occurrence of $n-k-1$, which implies that C is empty since $p_n = n-k$. Hence, T can be constructed with a binary alphabet in those cases, *cf.* Fig. 2.

For the case $p_1 = k+1$, P is split after the occurrences of the values $n-k$ and $k+1-k-1 \mod n = n \mod n$, so B contains only the text position n. By construction, the requirement is that the suffix $\mathsf{T}[n]$ is smaller than all other suffixes starting with $\mathtt{c}$. So instead of assigning the unique symbol $\mathsf{T}[n] \leftarrow \mathtt{b}$ as in Theorem 4, we can assign $\mathsf{T}[n] \leftarrow \mathtt{c}$, which still makes $\mathsf{T}[n]$ the smallest suffix starting with $\mathtt{c}$. We conclude this case by converting the binary alphabet $\{\mathtt{a}, \mathtt{c}\}$ to $\{\mathtt{a}, \mathtt{b}\}$. *Cf.* Fig. 2, where T in Rotation (6) has become $\mathtt{bbabbabb}$ with period $n-k = 3$. □

The main result of this section is the following theorem. There, we characterize all binary strings whose suffix arrays are arithmetically progressed permutations. More precisely, we identify which of them are unique[5], periodic, or a Lyndon word.

[5] The exact number of these binary strings is not covered by Theorem 8.

Case	T	SA	p_s	s
	1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8		
(1)	b a b b a b b a	[8, 5, 2, 7, 4, 1, 6, 3]	2	3
(2)	b b a b b a b b	[6, 3, 8, 5, 2, 7, 4, 1]	3	2
(3)	a b a b b a b b	[1, 6, 3, 8, 5, 2, 7, 4]	3	3

Figure 6. All binary strings of length 8 whose suffix arrays are arithmetically progressed permutations with ratio $k = 5$. Theorem 11 characterizes these strings (and also gives the definition of p_s). Cases (1) and (3) also appear in Fig. 2 at Rotation (8) and (5), respectively, while Case (2) can be obtained from Rotation (6) by exchanging the last character with c. Cases (1) and (2) both have period $n - k = 3$, and Case (3) is a Lyndon word.

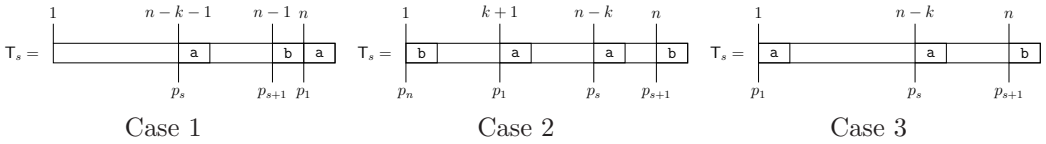


Figure 7. Sketches of the cases of Theorem 11. T_s is uniquely determined if the suffix array SA of T_s is arithmetically progressed with ratio k and the first entry $\mathsf{SA}[1] \in \{1, k+1, n-k\}$ is given.

Theorem 11. *Let n and $k \in [1..n-1]$ be two coprime integers. If $k \neq n-1$, there are* exactly three *binary strings of length n whose suffix arrays are arithmetically progressed permutations with ratio k. Each such solution $\mathsf{T}_s \in \{a, b\}^+$ is characterized by*

$$\mathsf{T}_s[i] = \begin{cases} \mathtt{a} & \text{for } i \in \mathsf{SA}_{\mathsf{T}_s}[1..s], \text{ or} \\ \mathtt{b} & \text{otherwise,} \end{cases} \tag{2}$$

for all text positions $i \in [1..n]$ and an index $s \in [1..n-1]$ (the split *index).*

The individual solutions are obtained by fixing the values for p_1 and p_s, the position of the lexicographically largest suffix starting with a*, of $\mathsf{SA}_{\mathsf{T}_s} = [p_1, \ldots, p_n]$:*

1. *$p_1 = n$ and $p_s = n - k - 1$,*
2. *$p_1 = k + 1$ and $p_s = n - k$, and*
3. *$p_1 = 1$ and $p_s = n - k$.*

The string T_s has period $n - k$ in Cases 1 and 2, while T_s of Case 3 is a Lyndon word, which is not periodic by definition.

For $k = n - 1$, Cases 2 and 3 each yields exactly one binary string, but Case 1 yields n binary strings according to Theorem 2.

Proof. Let S be a binary string of length n, and suppose that $\mathsf{SA}_{\mathsf{S}} = P := [p_1, \ldots, p_n]$ is an arithmetically progressed permutation with ratio k. Further let p_s be the position of the largest suffix of S starting with a. Then $\mathsf{S}[p_i..n] \prec \mathsf{S}[p_{i+1}..n]$ and thus $\mathsf{S}[p_i] \leq \mathsf{S}[p_{i+1}]$. We have $\mathsf{S}[j] = \mathsf{S}[j + k \mod n]$ for all $j \in [1..n] \setminus \{p_n, p_s\}$ since

- $p_n = \mathsf{SA}_{\mathsf{S}}[n]$ is the starting position of the largest suffix ($\mathsf{S}[p_n] = \mathtt{b} \neq \mathtt{a} = \mathsf{S}[p_1] = \mathsf{S}[p_n + k]$).
- $\mathsf{S}[p_s..n]$ and $\mathsf{S}[p_{s+1}..n]$ are the lexicographically largest suffix starting with a and the lexicographically smallest suffix starting with b, respectively, such that $\mathsf{S}[p_s] = \mathtt{a} \neq \mathtt{b} = \mathsf{S}[p_{s+1}]$.

To sum up, since $\mathsf{S}[p_s..n] \prec \mathsf{S}[p_{s+1}..n]$ by construction, $\mathsf{S}[p_i..n] \prec \mathsf{S}[p_{i+1}..n]$ holds for $p_i > p_{i+1}$ whenever $p_i \neq p_n$. This, together with the coprimality of n and k,

determines p_s uniquely in the three cases (*cf.* Fig. 6 for the case that $n = 8$ and $k = 5$ and Fig. 7 for sketches of the proof):

Case 1: We first observe that the case $k = n-1$ gives us $P = [n, n-1, \ldots, 1]$, and this case was already treated with Theorem 1. In the following, we assume $k < n-1$, and under this assumption we have $s > 1$, $\mathsf{T}_s[n] = \mathsf{T}_s[p_1] = \mathtt{a}$ and $\mathsf{T}_s[n-1] = \mathtt{b}$ (otherwise $\mathsf{T}_s[n-1..n]$ would be the second smallest suffix, i.e., $P[2] = n-1$ and hence $k = n-1$). Consequently, $\mathsf{T}_s[n-1..n] = \mathsf{T}_s[p_{s+1}..n]$ is the smallest suffix starting with $\mathtt{b}$, namely $\mathtt{ba}$, and therefore $p_s = n-1-k$.

Case $p_1 \neq n$: If $p_1 \neq n$, then $\mathsf{T}_s[n] = \mathtt{b}$ (otherwise $\mathsf{T}_s[n] \prec \mathsf{T}_s[p_1..n]$). Therefore, $\mathsf{T}_s[n]$ is the smallest suffix starting with $\mathtt{b}$, and consequently $p_s = n-k$.

For the periodicity, with $\mathsf{T}_s[j] = \mathsf{T}_s[j+k \mod n] = \mathsf{T}_s[j-(n-k) \mod n]$ for $j \in [1..n] \setminus \{p_1, p_s\}$ we need to check two conditions:

- If $p_n - (n-k) > 0$, then $\mathsf{T}_s[p_n-(n-k)] = \mathsf{T}_s[p_1] \neq \mathsf{T}_s[p_n]$ breaks the periodicity.
- If $p_s - (n-k) > 0$, then $\mathsf{T}_s[p_s-(n-k)] = \mathtt{a} \neq \mathtt{b} = \mathsf{T}_s[p_{s+1}]$ breaks the periodicity.

For Case 1, $p_n = n-k$ and $p_s = n-k-1$ (hence $p_n-(n-k) = 0$ and $p_s-(n-k) = -1$), thus Case 1 is periodic.
Case 2 is analogous to Case 1.
For Case 3, T_s does not have period $n-k$ as $p_n = n-k+1$, and hence $p_n-(n-k) > 0$. It cannot have any other period since Case 3 yields a Lyndon word (because the lexicographically smallest suffix $\mathsf{T}_s[p_1..n] = \mathsf{T}_s[1..n]$ starts at the first text position). Note that Case 3 can be obtained from Case 2 by setting $\mathsf{T}_s[1] \leftarrow \mathtt{a}$ (the smallest suffix $\mathsf{T}_s[k+1..n]$ thus becomes the second smallest suffix).

Finally, we need to show that no other value for p_1 admits a binary string S having an arithmetically progressed permutation $P := [p_1, \ldots, p_n]$ with ratio k as its suffix array. So suppose that $p_1 \notin \{1, k+1, n\}$, then this would imply the following:

- $\mathsf{S}[p_1] = \mathtt{a}$ because the smallest suffix starts at text position p_1, and
- $\mathsf{S}[p_1-1] = \mathtt{b}$ because of the following: First, the text position $\mathsf{S}[p_1-1]$ exists due to $p_1 > 1$. Second, since $p_1 < n$, there is a text position $j \in [p_1+1..n]$ such that $\mathsf{S}[p_1] = \ldots = \mathsf{S}[j-1] = \mathtt{a}$ and $\mathsf{S}[j] = \mathtt{b}$ (otherwise $\mathsf{S}[n]$ would be the smallest suffix). If $\mathsf{S}[p_1-1] = \mathtt{a}$, then the suffix $\mathsf{S}[p_1-1..n]$ starting with $\mathtt{a}^{j-p_1+1}\mathtt{b}$ is lexicographically smaller than the suffix $\mathsf{S}[p_1..n]$ starting with $\mathtt{a}^{j-p_1}\mathtt{b}$. Hence, $\mathsf{S}[p_1-1] = \mathtt{b}$ must hold.
- $p_n - 1 \geq 1$ (since $p_1 \neq k+1$) and $\mathsf{S}[p_n-1] = \mathtt{a}$. If $\mathsf{S}[p_n-1] = \mathtt{b}$, then the suffix $\mathsf{S}[p_n-1..n]$ has a longer prefix of $\mathtt{b}$'s than the suffix $\mathsf{S}[p_n..n]$, and is therefore lexicographically larger.

Since $\mathsf{S}[p_n-1] = \mathtt{a}$ and $\mathsf{S}[p_1-1] = \mathtt{b}$ with $p_n-1+k \mod n = p_1-1$, the smallest suffix starting with $\mathtt{b}$ is located at index p_1-1. This is a contradiction as $p_1 \neq n$ implies $\mathsf{S}[n] = \mathtt{b}$ (if $\mathsf{S}[n] = \mathtt{a}$, then $\mathsf{SA}[1] = n$ instead of $\mathsf{SA}[1] = p_1$) and thus the smallest suffix starting with $\mathtt{b}$ is located at index n (this is a contradiction since we assumed that this suffix starts at $p_1 - 1 \in [1..n-1]$). This establishes the claim for p_1. □

For a given arithmentically progressed permutation with ratio k, and first entry $p_1 \in \{1, k+1, n\}$, the string T_s of Theorem 11 coincides with T of Theorem 10.

3.4 Inverse Permutations

Since the inverse P^{-1} of a permutation P with $P^{-1}[P[i]] = i$ is also a permutation, one may wonder whether the inverse P^{-1} of an arithmetically progressed permutation is also arithmetically progressed. We affirm this question in the following. For that, we use the notion of the *multiplicative inverse* k^{-1} of an integer k (to the congruence class $[1..n] = \mathbb{Z}/n\mathbb{Z}$), which is given by $k^{-1} \cdot k \mod n = 1 \mod n$. The multiplicative inverse k^{-1} is uniquely defined if k and n are coprime.

Theorem 12. *The inverse P^{-1} of an arithmetically progressed permutation P with ratio k is an arithmetically progressed permutation with ratio k^{-1} and $P^{-1}[1] = (1 - P[n]) \cdot k^{-1} \bmod n$.*

Proof. Let $x := P[i]$ for an index $i \in [1..n]$. Then $P[i + k^{-1} \bmod n] = x - k \cdot k^{-1} \bmod n = x - 1 \bmod n$. For the inverse permutation P^{-1} this means that $P^{-1}[x] = i$ and $P^{-1}[x-1 \bmod n] = i + k^{-1} \bmod n$. Thus the difference $P^{-1}[x-1 \bmod n] - P^{-1}[x]$ is k^{-1}.

Since $P[i] = j \iff P[n] + ik \mod n = j$ holds for all indices $i \in [1..n]$, we have (using $i \leftarrow P^{-1}[1]$ and $j \leftarrow 1$ in the above equivalence)

$$\begin{aligned} P[P^{-1}[1]] = 1 \mod n &\iff P[n] + P^{-1}[1] \cdot k = 1 \mod n \\ &\iff P^{-1}[1] \cdot k = 1 - P[n] \mod n \\ &\iff P^{-1}[1] = (1 - P[n]) \cdot k^{-1} \mod n. \end{aligned}$$

□

Consequently, using the split index s of p_s for SA and

$$\begin{aligned} \mathsf{ISA}[i] &= \mathsf{ISA}[1] + (i-1)k^{-1} \mod n \\ &= (1 - \mathsf{SA}[n]) \cdot k^{-1} + (i-1)k^{-1} \mod n \\ &= (i - \mathsf{SA}[n]) \cdot k^{-1} \mod n, \end{aligned}$$

we can rewrite T_s defined in Eq. 2 as

$$\mathsf{T}_s[i] = \begin{cases} \mathsf{a} & \text{if } \mathsf{ISA}[i] \leq p_s, \text{ or} \\ \mathsf{b} & \text{otherwise} \end{cases} \tag{3}$$

where SA and ISA denote the suffix array and the inverse suffix array of T_s, respectively. Another result is that $\mathsf{ISA}[p_s] = s$ is the number of a's in T_s, for which we split the study into the cases of Theorem 11:

1. If $\mathsf{SA}[1] = n$ and $p_s = n - k - 1$, then $\mathsf{SA}[n] = n - k$ and $\mathsf{ISA}[i] = (i - n + k)k^{-1} \bmod n$. Consequently, $\mathsf{ISA}[p_s] = (-1)k^{-1} \bmod n = n - k^{-1} \bmod n$.
2. If $\mathsf{SA}[1] = k + 1$ and $p_s = n - k$, then $\mathsf{SA}[n] = 1$ and $\mathsf{ISA}[i] = (i-1)k^{-1} \bmod n$. Consequently, $\mathsf{ISA}[p_s] = (n - k - 1)k^{-1} \bmod n = nk^{-1} - 1 - k^{-1} \bmod n = n - 1 - k^{-1} \bmod n$.
3. If $\mathsf{SA}[1] = 1$ and $p_s = n - k$, then $\mathsf{SA}[n] = n - k + 1$ and $\mathsf{ISA}[i] = (i - n + k - 1)k^{-1} \bmod n$. Consequently, $\mathsf{ISA}[p_s] = (-1)k^{-1} \bmod n = n - k^{-1} \bmod n$ as in Case (1).

For Fig. 6 with $k = 5$ and $n = 8$, we know that the number of a's is $\mathsf{ISA}[p_s] = 3$ in Cases (1) and (3), and $\mathsf{ISA}[p_s] = 2$ in Case (2) because $k^{-1} = 5 \Leftrightarrow k \cdot k^{-1} \bmod n = 1 \bmod n$. This also determines the constant m used in Lemma 9. Finally, we can fix the parameter t in Lemma 6 defined such that $p_t = p_1 - 1 - k \bmod n$: For that, write $\mathsf{ISA}[i] = (i - p_n)k^{-1} \bmod n = (i + k - p_1)k^{-1} \bmod n$ and compute $\mathsf{ISA}[p_t] = \mathsf{ISA}[p_1 - 1 - k] = (-1)k^{-1} \bmod n = n - k^{-1} \bmod n$.

3.5 Relation to the Fibonacci word sequence

Köppl and I [16, Thm. 1] observed that the suffix array of F_m for even m is the arithmetically progressed permutation $\mathsf{SA}_{\mathsf{F}_m}$ with ratio $f_{m-2} \bmod f_m$ and $\mathsf{SA}_{\mathsf{F}_m}[1] = f_m$. Theorem 11 generalizes this observation by characterizing all binary strings whose suffix arrays are arithmetically progressed. Hence, F_m must coincide with Case 1 of Theorem 11 since it ends with character $\mathtt{a}$.

Lemma 13. *The Fibonacci word F_m for even m is given by*

$$\mathsf{F}_m[i] = \begin{cases} \mathtt{a} & \text{if } 1 + i \cdot f_{m-2} \bmod f_m \le f_{m-1}, \text{ or} \\ \mathtt{b} & \text{otherwise.} \end{cases}$$

Proof. We use the following facts:

- The greatest common divisor of f_i and f_j is the Fibonacci number whose index is the greatest common divisor of i and j [24, Fibonacci numbers]. Hence, f_{m-1} and f_m are coprime for every $m \ge 2$.
- $f_{m-2}^2 \bmod f_m = 1$ holds for every even $m \ge 3$ [14]. Hence, $k^{-1} = k = f_{m-2}$.
- By definition, $\mathsf{F}_m[f_m] = \mathtt{a}$ if m is even, and therefore $\mathsf{SA}_{\mathsf{F}_m}[1] = f_m$.

The split position p_s is $p_s = f_m - k = f_{m-1}$. So $\mathsf{SA}_{\mathsf{F}_m}[f_m] = f_m - k = f_m - f_{m-2}$. By Theorem 12, $\mathsf{ISA}_{\mathsf{F}_m}[i] = if_{m-2} - \mathsf{SA}_{\mathsf{F}_m}[f_m]f_{m-2} \bmod f_m = if_{m-2} + 1 \bmod f_m$, where $-\mathsf{SA}_{\mathsf{F}_m}[f_m]f_{m-2} \bmod f_m = (f_{m-2} - f_m)f_{m-2} \bmod f_m = 1 - f_m f_{m-2} \bmod f_m = 1$. The rest follows from Eq. 3. □

Let $\bar{\mathsf{F}}_m$ denote the m-th Fibonacci word whose $\mathtt{a}$'s and $\mathtt{b}$'s are exchanged, i.e., $\bar{\mathsf{F}}_m = \mathtt{a} \Leftrightarrow \mathsf{F}_m = \mathtt{b}$.

Lemma 14. $\mathsf{SA}_{\bar{\mathsf{F}}_m}$ *is arithmetically progressed with ratio f_{m-2} for odd m.*

Proof. Since $\bar{\mathsf{F}}_m[|\bar{\mathsf{F}}_m|] = \mathtt{a}$ for odd m, $\bar{\mathsf{F}}_m[f_m..]$ is the lexicographically smallest suffix. Hence, $\mathsf{SA} := \mathsf{SA}_{\bar{\mathsf{F}}_m} = |\bar{\mathsf{F}}_m|$. If SA is arithmetically progressed with ratio k, then its split position must be $p_s = n-k-1$ according to Theorem 11. We show that $k = f_{m-2}$ by proving

$$\bar{\mathsf{F}}_m[f_m..] \prec \bar{\mathsf{F}}_m[f_m + f_{m-2} \bmod f_m..] \prec \bar{\mathsf{F}}_m[f_m + 2f_{m-2} \bmod f_m..] \prec \dots \\ \prec \bar{\mathsf{F}}_m[f_m + (f_m - 1)f_{m-2} \bmod f_m..]$$

in a way similar to [16, Lemma 8]. For that, let $\bar{\mathsf{S}}$ of a binary string $\mathsf{S} \in \{\mathtt{a}, \mathtt{b}\}^*$ denote S after exchanging $\mathtt{a}$'s and $\mathtt{b}$'s (i.e., $\bar{\mathsf{S}} = \mathtt{a} \Leftrightarrow \mathsf{S} = \mathtt{b}$). Further, let $\lessdot$ be the relation on strings such that $\mathsf{S} \lessdot \mathsf{T}$ if and only if $\mathsf{S} \prec \mathsf{T}$ and S is *not* a prefix of T. We need this relation since $\mathsf{S} \lessdot \mathsf{T} \iff \bar{\mathsf{S}} \gtrdot \bar{\mathsf{T}}$ while $\mathsf{S} \prec \mathsf{T}$ and $\bar{\mathsf{S}} \prec \bar{\mathsf{T}}$ holds if S is a prefix of T.

- For $i \in [1..f_{m-1})$, we have $\mathsf{F}_m[i..] \gtrdot \mathsf{F}_m[i + f_{m-2}..]$ due to [16, Lemma 7], thus $\bar{\mathsf{F}}_m[i..] \lessdot \bar{\mathsf{F}}_m[i + f_{m-2}..]$.
- For $i \in (f_{m-1}..f_m]$, since $\mathsf{F}_m = \mathsf{F}_{m-1}\mathsf{F}_{m-2} = \mathsf{F}_{m-2}\mathsf{F}_{m-3}\mathsf{F}_{m-2}$, $\mathsf{F}_m[i..]$ is a prefix of $\mathsf{F}_m[i - f_{m-1}..] = \mathsf{F}_m[i + f_{m-2} \bmod f_m..]$. Therefore, $\bar{\mathsf{F}}_m[i..]$ is a prefix $\bar{\mathsf{F}}_m[i + f_{m-2} \bmod f_m..]$ and $\bar{\mathsf{F}}_m[i..] \prec \bar{\mathsf{F}}_m[i + f_{m-2} \bmod f_m..]$.

Since f_m and f_{m-2} are coprime, $\{i + f_{m-2} \bmod f_m \mid i > 0\} = [1..n]$. Starting with the smallest suffix $\bar{\mathsf{F}}_m[f_m..]$, we end up at the largest suffix $\bar{\mathsf{F}}_m[f_m + (f_m - 1)f_{m-2} \bmod f_m..]$ after $m - 1$ arithmetic progression steps of the form $\bar{\mathsf{F}}_m[f_m + if_{m-2} \bmod f_m..]$ for $i \in [0..f_m - 1]$. By using one of the two above items we can show that these arithmetic progression steps yield a list of suffixes sorted in lexicographically ascending order. □

4 Conclusion and Problems

Given an arithmetically progressed permutation P with ratio k, we studied the minimum alphabet size and the shape of those strings having P as their suffix array. Only in the case $P = [n, n-1, \ldots, 1]$, a unary alphabet suffices. For general $P = [p_1, \ldots, p_n] \neq [n, n-1, \ldots, 1]$, there is exactly one such string on the binary alphabet if and only if $p_1 \in \{1, k+1, n\}$. In all other cases, there is exactly one such string on the ternary alphabet.We conclude by proposing some research directions.

- Prove which of the solutions for binary strings (if any) yield *balanced words*, i.e., binary strings T such that for each character $c \in \{\mathtt{a}, \mathtt{b}\}$, the number of occurrences of c in U and in V differ by at most one, for all pairs of substrings U and V with $|\mathsf{U}| = |\mathsf{V}|$ of the infinite concatenation $\mathsf{T} \cdot \mathsf{T} \cdots$ of T.
- A natural question arising from this research is to characterize strings having arithmetic progression properties for the run length exponents of their BWTs, particularly for the bijective [12] or extended BWT [19], which are always invertible.

  For example, given the arithmetically progressed permutation 3214, then the run-length compressed string $\mathtt{a}^3\mathtt{c}^2\mathtt{\$}\mathtt{b}^4$ (a) matches the permutation 3214 and (b) is a BWT image because its inverse is $\mathtt{b}^2\mathtt{c}\mathtt{b}^2\mathtt{c}\mathtt{a}^3\mathtt{\$}$, which can be computed by the Last-First mapping. However, for the same permutation, $\mathtt{a}^3\mathtt{b}^2\mathtt{\$}\mathtt{b}^4$ does not work since it is not a BWT image.
- Arithmetic properties can likewise be considered for the following stringology integer arrays:
  - Firstly the *longest common prefix* (LCP) array LCP, whose entry $\mathsf{LCP}[i]$ is the length of the longest common prefix of the *lexicographically* i-th smallest suffix with its lexicographic predecessor for $i \in [2..n]$.
  - Given a string $\mathsf{T} \in \Sigma^+$ of length n, the *prefix table* P_T of T is given by $P_\mathsf{T}[i] = \mathsf{LCP}(\mathsf{T}, \mathsf{T}[i..n])$ for $i \in [1..n]$; equivalently, the *border table* B_T of T is defined by
  $$B_\mathsf{T}[i] = \max\{|\mathsf{S}| \mid \mathsf{S} \text{ is a border of } \mathsf{T}[1..i]\} \text{ for } i \in [1..n].$$
  - Integer *prefix lists* are more concise than prefix tables and give the lengths of overlapping LCPs of T and suffixes of T (cf. [7]).
  - The i-th entry of the *Lyndon array* $\lambda = \lambda_\mathsf{T}[1..n]$ of a given string $\mathsf{T} = \mathsf{T}[1..n]$ is the length of the longest Lyndon word that is a prefix of $T[i..]$ – reverse engineering in [9] includes a linear-time test for whether an integer array is a Lyndon array. Likewise, the *Lyndon factorization array* $\mathsf{F} = \mathsf{F}_\mathsf{T}[1..n]$ of T in [6] gives at each position i the number of factors in the Lyndon factorization starting at i, that is the number of factors in the suffix $\mathsf{F}_\mathsf{T}[i..n]$. The problems are to characterize those arithmetic progressions which define a valid Lyndon array, respectively Lyndon factorization array. For example, consider the string $\mathsf{T} = \mathtt{azyx}$, then its Lyndon array is $\lambda_\mathsf{T} = [4, 1, 1, 1]$, while the Lyndon factorization array is $\mathsf{F}_\mathsf{T} = [1, 3, 2, 1]$. Trivially, for $\mathsf{T} = \mathtt{abc}\ldots\mathtt{z}$ the Lyndon array is an arithmetic progression and likewise for the Lyndon factorization array of $\mathsf{T} = \mathtt{z}^t\mathtt{y}^t\mathtt{x}^t \ldots \mathtt{a}^t$.
- A challenging research direction is to consider arithmetic progressions for multi-dimensional suffix arrays and Fibonacci word sequences.

Acknowledgements

We thank Gabriele Fici for the initial help in guiding this research started at String-Masters.

Funding: This research was part-funded by JSPS KAKENHI with grant number JP18F18120, and by the European Regional Development Fund through the Welsh Government [Grant Number 80761-AU-137 (West)]:

References

1. D. Adjeroh, T. Bell, and A. Mukherjee: *The Burrows-Wheeler Transform: Data compression, Suffix arrays, and Pattern matching*, Springer, 2008.
2. J. Berstel and A. Savelli: *Crochemore factorization of Sturmian and other infinite words*, in Proc. MFCS, vol. 4162 of LNCS, 2006, pp. 157–166.
3. T. Bingmann, J. Fischer, and V. Osipov: *Inducing suffix and LCP arrays in external memory.* ACM Journal of Experimental Algorithmics, 21(1) 2016, pp. 2.3:1–2.3:27.
4. M. Burrows and D. J. Wheeler: *A block sorting lossless data compression algorithm*, Tech. Rep. 124, Digital Equipment Corporation, Palo Alto, California, 1994.
5. M. Christodoulakis, C. S. Iliopoulos, and Y. J. P. Ardila: *Simple algorithm for sorting the Fibonacci string rotations*, in Proc. SOFSEM, vol. 3831 of LNCS, 2006, pp. 218–225.
6. A. Clare and J. W. Daykin: *Enhanced string factoring from alphabet orderings.* Inf. Process. Lett., 143 2019, pp. 4–7.
7. J. Clément and L. Giambruno: *Representing prefix and border tables: results on enumeration.* Mathematical Structures in Computer Science, 27(2) 2017, pp. 257–276.
8. M. Crochemore, R. Grossi, J. Kärkkäinen, and G. M. Landau: *Computing the Burrows-Wheeler transform in place and in small space.* J. Discrete Algorithms, 32 2015, pp. 44–52.
9. J. W. Daykin, F. Franek, J. Holub, A. S. M. S. Islam, and W. F. Smyth: *Reconstructing a string from its Lyndon arrays.* Theor. Comput. Sci., 710 2018, pp. 44–51.
10. W. Feller: *An introduction to probability theory and its applications*, Wiley, 1968.
11. J. Fischer and F. Kurpicz: *Lightweight distributed suffix array construction*, in Proc. ALENEX, 2019, pp. 27–38.
12. J. Y. Gil and D. A. Scott: *A bijective string sorting transform.* ArXiv 1201.3077, 2012.
13. S. Giuliani, Z. Lipták, and R. Rizzi: *When a dollar makes a BWT*, in Proc. ICTCS, vol. 2504 of CEUR Workshop Proceedings, 2019, pp. 20–33.
14. V. Hoggatt and M. Bicknell-Johnson: *Composites and Primes Among Powers of Fibonacci Numbers increased or decreased by one.* Fibonacci Quarterly, 15 1977, p. 2.
15. D. Köppl, D. Hashimoto, D. Hendrian, and A. Shinohara: *In-place bijective Burrows-Wheeler transformations*, in Proc. CPM, LIPIcs, 2020, p. to appear.
16. D. Köppl and T. I: *Arithmetics on suffix arrays of Fibonacci words*, in Proc. WORDS, vol. 9304 of LNCS, 2015, pp. 135–146.
17. M. Lothaire: *Combinatorics on Words*, Cambridge Mathematical Library, Cambridge University Press, 2 ed., 1997.
18. U. Manber and E. W. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM J. Comput., 22(5) 1993, pp. 935–948.
19. S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino: *An extension of the Burrows-Wheeler transform.* Theor. Comput. Sci., 387(3) 2007, pp. 298–312.
20. S. Mantaci, A. Restivo, and M. Sciortino: *Burrows-Wheeler transform and Sturmian words.* Inf. Process. Lett., 86(5) 2003, pp. 241–246.
21. G. Nong, S. Zhang, and W. H. Chan: *Linear suffix array construction by almost pure induced-sorting*, in Proc. DCC, 2009, pp. 193–202.
22. W. Rytter: *The structure of subword graphs and suffix trees of Fibonacci words.* Theor. Comput. Sci., 363(2) 2006, pp. 211–223.
23. J. Simpson and S. J. Puglisi: *Words with simple Burrows-Wheeler transforms.* Electr. J. Comb., 15(1) 2008.
24. D. Wells: *Prime Numbers: The Most Mysterious Figures in Math*, Wiley, 2005.

Pointer-Machine Algorithms for Fully-Online Construction of Suffix Trees and DAWGs on Multiple Strings

Shunsuke Inenaga

Department of Informatics, Kyushu University, Japan
PRESTO, Japan Science and Technology Agency, Japan
inenaga@inf.kyushu-u.ac.jp

Abstract. We deal with the problem of maintaining the *suffix tree* indexing structure for a fully-online collection of strings, where a new character can be prepended to any string in the collection at any time. The only previously known algorithm for the problem, recently proposed by Takagi et al. [Algorithmica 82(5): 1346-1377 (2020)], runs in $O(N \log \sigma)$ time and $O(N)$ space *on the word RAM* model, where N denotes the total length of the strings and σ denotes the alphabet size. Their algorithm makes heavy use of the nearest marked ancestor (NMA) data structure on semi-dynamic trees, that can answer queries and supports insertion of nodes in $O(1)$ amortized time on the word RAM model. In this paper, we present a simpler fully-online right-to-left algorithm that builds the suffix tree for a given string collection in $O(N(\log \sigma + \log d))$ time and $O(N)$ space, where d is the maximum number of in-coming *Weiner links* to a node of the suffix tree. We note that d is bounded by the height of the suffix tree, which is further bounded by the length of the longest string in the collection. The advantage of this new algorithm is that it works on *the pointer machine model*, namely, it does not use the complicated NMA data structures that involve table look-ups. As a byproduct, we also obtain a pointer-machine algorithm for building the *directed acyclic word graph* (*DAWG*) for a fully-online left-to-right collection of strings, which runs in $O(N(\log \sigma + \log d))$ time and $O(N)$ space again without the aid of the NMA data structures.

1 Introduction

1.1 Suffix trees and DAWGs

Suffix trees are a fundamental string data structure with a myriad of applications [10]. The first efficient construction algorithm for suffix trees, proposed by Weiner [21], builds the suffix tree for a string in a right-to-left online manner, by updating the suffix tree each time a new character is prepended to the string. It runs in $O(n \log \sigma)$ time and $O(n)$ space, where n is the length of the string and σ is the alphabet size.

One of the most interesting features of Weiner's algorithm is a very close relationship to Blumer et al.'s algorithm [2] that builds the *directed acyclic word graph* (*DAWG*) in a left-to-right online manner, by updating the DAWG each time a new character is prepended to the string. It is well known (c.f. [4,5]) that the DAG of the Weiner links of the suffix tree of T is equivalent to the DAWG of the reversal $\overline{T}$ of T, or symmetrically, the suffix link tree of the DAWG of $\overline{T}$ is equivalent to the suffix tree of T. Thus, right-to-left online construction of suffix trees is essentially equivalent to left-to-right construction of DAWGs. This means that Blumer et al.'s DAWG construction algorithm also runs in $O(n \log \sigma)$ time and $O(n)$ space [2].

DAWGs also support efficient pattern matching queries, and have been applied to other important string problems such as local alignment [6], pattern matching with

Shunsuke Inenaga: Pointer-Machine Algorithms for Fully-Online Construction of Suffix Trees and DAWGs on Multiple Strings, pp. 111–124.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9

variable-length don't cares [14], dynamic dictionary matching [11], compact online Lempel-Ziv factorization [23], finding minimal absent words [8], and finding gapped repeats [18].

1.2 Fully online construction of suffix trees and DAWGs

Takagi et al. [17] initiated the generalized problem of maintaining the suffix tree for a collection of strings in a *fully-online manner*, where a new character can be prepended to any string in the collection at any time. This fully-online scenario arises in real-time database systems e.g. for sensor networks or trajectories. Takagi et al. showed that a direct application of Weiner's algorithm [21] to this fully-online setting requires one to visit $\Theta(N \min(K, \sqrt{N}))$ nodes, where N is the total length of the strings and K is the number of strings in the collection. Note that this leads to a worst-case $\Theta(N^{1.5} \log \sigma)$-time construction when $K = \Omega(\sqrt{N})$.

In their analysis, it was shown that Weiner's original algorithm applied to a fully-online string collection visits a total of $\Theta(N \min(K, \sqrt{N}))$ nodes. This means that the amortization argument of Weiner's algorithm for the number of nodes visited in the climbing process for inserting a new leaf, does not work for multiple strings in the fully-online setting. To overcome difficulty, Takagi et al. proved the three following statements: (1) By using σ nearest marked ancestor (NMA) structures [22], one can skip the last part of the climbing process; (2) All the σ NMA data structures can be stored in $O(n)$ space; (3) The number of nodes explicitly visited in the remaining part of each climbing process can be amortized to $O(1)$ per new added character. This led to their $O(N \log \sigma)$-time and $O(N)$-space fully-online right-to-left construction of the suffix tree for multiple strings.

Takagi et al. [17] also showed that Blumer et al.'s algorithm [2,3] applied to a fully-online left-to-right DAWG construction requires at least $\Theta(N \min(K, \sqrt{N}))$ work as well. They also showed how to maintain an *implicit* representation of the DAWG of $O(N)$ space which supports fully-online updates and simulates a DAWG edge traversal in $O(\log \sigma)$ time each. The key here was again the non-trivial use of the aforementioned σ NMA data structures over the suffix tree of the reversed strings.

As states above, Takagi et al.'s construction heavily relies on the use of the NMA data structures [22]. Although NMA data structures are useful and powerful, all known NMA data structures for (static and dynamic) trees that support $O(1)$ (amortized) time queries and updates [9,12,22] are quite involved, and they are valid only on the word RAM model as they use look-up tables that explicitly store the answers for small sub-problems. Hence, in general, it would be preferable if one could achieve similar efficiency without NMA data structures.

1.3 Our contribution

In this paper, we show how to maintain the suffix tree for a right-to-left fully-online string collection in $O(N(\log \sigma + \log d))$ time and $O(N)$ space, where d is the maximum number of in-coming *Weiner links* to a node of the suffix tree. Our construction does not use NMA data structures and works in the *pointer-machine model* [19,20], which is a simple computational model without address arithmetics. We note that d is bounded by the height of the suffix tree. Clearly, the height of the suffix tree is at most the maximum length of the strings. Hence, the d term can be dominated by the σ term when the strings are over integer alphabets of polynomial size in N, or when a

large number of strings of similar lengths are treated. To achieve the aforementioned bounds on the pointer-machine model, we reduce the problem of maintaining incoming Weiner links of nodes to the *ordered split-insert-find problem*, which maintains dynamic sets of sorted elements allowing for split and insert operations, and find queries, which can be solved in a total of $O(N \log d)$ time and $O(N)$ space.

As a byproduct of the above result, we also obtain the *first* non-trivial algorithm that maintains an *explicit* representation of the DAWG for fully-online left-to-right multiple strings, which runs in $O(N(\log \sigma + \log d))$ time and $O(N)$ space. By an explicit representation, we mean that every edge of the DAWG is implemented as a pointer. This DAWG construction does not require complicated table look-ups and thus also works on the pointer machine model.

2 Preliminaries

2.1 String notations

Let Σ be a general ordered alphabet. Any element of Σ^* is called a *string*. For any string T, let $|T|$ denote its length. Let ε be the empty string, namely, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma \setminus \{\varepsilon\}$. If $T = XYZ$, then X, Y, and Z are called a *prefix*, a *substring*, and a *suffix* of T, respectively. For any $1 \leq i \leq j \leq |T|$, let $T[i..j]$ denote the substring of T that begins at position i and ends at position j in T. For any $1 \leq i \leq |T|$, let $T[i]$ denote the ith character of T. For any string T, let $\mathsf{Suffix}(T)$ denote the set of suffixes of T, and for any set $\mathcal{T}$ of strings, let $\mathsf{Suffix}(\mathcal{T})$ denote the set of suffixes of all strings in $\mathcal{T}$. Namely, $\mathsf{Suffix}(\mathcal{T}) = \bigcup_{T \in \mathcal{T}} \mathsf{Suffix}(T)$. For any string T, let $\overline{T}$ denote the reversed string of T, i.e., $\overline{T} = T[|T|] \cdots T[1]$. For any set $\mathcal{T}$ of strings, let $\overline{\mathcal{T}} = \{\overline{T} \mid T \in \mathcal{T}\}$.

2.2 Suffix trees and DAWGs for multiple strings

For ease of description, we assume that each string T_i in the collection $\mathcal{T}$ terminates with a unique character $\$_i$ that does not appear elsewhere in $\mathcal{T}$. However, our algorithms work without $\$_i$ symbols at the right end of strings as well.

A compacted trie is a rooted tree such that (1) each edge is labeled by a non-empty string, (2) each internal node is branching, and (3) the string labels of the out-going edges of each node begin with mutually distinct characters. The *suffix tree* [21] for a text collection $\mathcal{T}$, denoted $\mathsf{STree}(\mathcal{T})$, is a compacted trie which represents $\mathsf{Suffix}(\mathcal{T})$. The *string depth* of a node v of $\mathsf{Suffix}(\mathcal{T})$ is the length of the substring that is represented by v. We sometimes identify node v with the substring it represents. The suffix tree for a single string T is denoted $\mathsf{STree}(T)$.

$\mathsf{STree}(\mathcal{T})$ has at most $2N - 1$ nodes and thus $2N - 2$ nodes, since every internal node of $\mathsf{STree}(\mathcal{T})$ is branching and there are N leaves in $\mathsf{STree}(\mathcal{T})$. By representing each edge label x with a triple $\langle k, i, j \rangle$ of integers such that $x = T_k[i..j]$, $\mathsf{STree}(\mathcal{T})$ can be stored in $O(N)$ space.

We define the *suffix link* of each non-root node av of $\mathsf{STree}(\mathcal{T})$ with $a \in \Sigma$ and $v \in \Sigma^*$, by $\mathsf{slink}(av) = v$. For each explicit node v and $a \in \Sigma$, we also define the reversed suffix link (a.k.a. *Weiner link*) by $\mathsf{W\_link}_a(v) = avx$, where $x \in \Sigma^*$ is the shortest string such that avx is a node of $\mathsf{STree}(\mathcal{T})$. $\mathsf{W\_link}_a(v)$ is undefined if av is not a substring of strings in $\mathcal{T}$. A Weiner link $\mathsf{W\_link}_a(v) = avx$ is said to be *hard* if $x = \varepsilon$, and *soft* if $x \in \Sigma^+$.

See the left diagram of Figure 1 for an example of $\mathsf{STree}(\mathcal{T})$ and Weiner links.

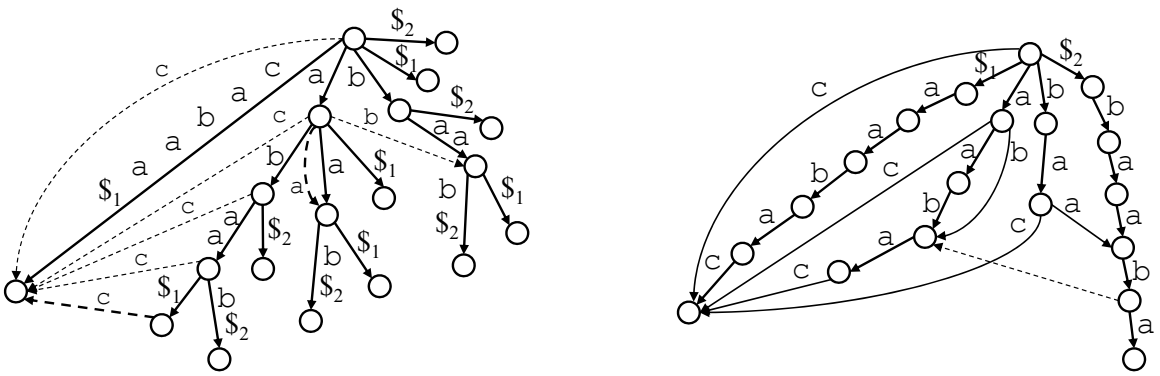


Figure 1. Left: $\mathsf{STree}(\mathcal{T})$ for $\mathcal{T} = \{\texttt{cabaa\$}_1, \texttt{abaab\$}_2\}$. The bold dashed arrows represent hard Weiner links, while the narrow dashed arrows represent soft Weiner links. Not all Weiner links are shown for simplicity. Right: $\mathsf{DAWG}(\mathcal{S})$ for $\mathcal{S} = \overline{\mathcal{T}} = \{\$_1\texttt{aabac}, \$_2\texttt{baaba}\}$. The dashed arrow represents a suffix link. Not all suffix links are shown for simplicity.

The *directed acyclic word graph* (*DAWG* for short) [2,3] of a text collection $\mathcal{S}$, denoted $\mathsf{DAWG}(\mathcal{S})$, is a (partial) DFA which represents $\mathsf{Suffix}(\mathcal{S})$. It is proven in [3] that $\mathsf{DAWG}(\mathcal{S})$ has at most $2N - 1$ nodes and $3N - 4$ edges for $N \geq 3$. Since each DAWG edge is labeled by a single character, $\mathsf{DAWG}(\mathcal{S})$ can be stored with $O(N)$ space. The DAWG for a single string S is denoted $\mathsf{DAWG}(S)$.

A node of $\mathsf{DAWG}(\mathcal{S})$ corresponds to the substrings in $\mathcal{S}$ which share the same set of ending positions in $\mathcal{S}$. Thus, for each node, there is a unique longest string represented by that node. For any node v of $\mathsf{DAWG}(\mathcal{S})$, let $\mathit{long}(v)$ denote the longest string represented by v. An edge (u, a, v) in the DAWG is called *primary* if $|\mathit{long}(u)| + 1 = |\mathit{long}(v)|$, and is called *secondary* otherwise. For each node v of $\mathsf{DAWG}(\mathcal{S})$ with $|\mathit{long}(v)| \geq 1$, let $\mathsf{slink}(v) = y$, where y is the longest suffix of $\mathit{long}(v)$ which is not represented by v.

Suppose $S = \overline{\mathcal{T}}$. It is known (c.f. [2,3,5]) that there is a node v in $\mathsf{STree}(\mathcal{T})$ iff there is a node x in $\mathsf{DAWG}(\mathcal{S})$ such that $\mathit{long}(x) = \overline{v}$. Also, the hard Weiner links and the soft Weiner links of $\mathsf{STree}(\mathcal{T})$ coincide with the primary edges and the secondary edges of $\mathsf{DAWG}(\mathcal{S})$, respectively. In a symmetric view, the reversed suffix links of $\mathsf{DAWG}(\mathcal{S})$ coincide with the suffix tree $\mathsf{STree}(\mathcal{T})$ for $\mathcal{T}$.

See Figure 1 for some concrete examples of the aforementioned symmetry. For instance, the nodes $\texttt{abaa}$ and $\texttt{baa}$ of $\mathsf{STree}(\mathcal{T})$ correspond to the nodes of $\mathsf{DAWG}(\mathcal{S})$ whose longest strings are $\overline{\texttt{abaa}} = \texttt{aaba}$ and $\overline{\texttt{baa}} = \texttt{aab}$, respectively. Observe that both $\mathsf{STree}(\mathcal{T})$ and $\mathsf{DAWG}(\mathcal{S})$ have 19 nodes each. The Weiner links of $\mathsf{STree}(\mathcal{T})$ labeled by character $\texttt{c}$ correspond to the out-going edges of $\mathsf{DAWG}(\mathcal{S})$ labeled by $\texttt{c}$. To see another example, the three Weiner links from node $\texttt{a}$ in $\mathsf{STree}(\mathcal{T})$ labeled $\texttt{a}$, $\texttt{b}$, and $\texttt{c}$ correspond to the three out-going edges of node $\{\texttt{a}\}$ of $\mathsf{DAWG}(\mathcal{S})$ labeled $\texttt{a}$, $\texttt{b}$, and $\texttt{c}$, respectively. For the symmetric view, focus on the suffix link of the node $\{\$_2\texttt{baab}, \texttt{baab}\}$ of $\mathsf{DAWG}(\mathcal{S})$ to the node $\{\texttt{aab}, \texttt{ab}\}$. This suffix link reversed corresponds to the edge labeled $\texttt{b}\$_2$ from the node $\texttt{baa}$ to the node $\texttt{baab}\$_2$ in $\mathsf{STree}(\mathcal{T})$.

We now see that the two following tasks are essentially equivalent:

(A) Building $\mathsf{STree}(\mathcal{T})$ for a fully-online right-to-left text collection $\mathcal{T}$, using hard and soft Weiner links.
(B) Building $\mathsf{DAWG}(\mathcal{S})$ for a fully-online left-to-right text collection $\mathcal{S}$, using suffix links.

2.3 Pointer machines

A pointer machine [19,20] is an abstract model of computation such that the state of computation is stored as a directed graph, where each node can contain a constant amount of data (e.g. integers, symbols) and a constant number of pointers (i.e. out-going edges to other nodes). The instructions supported by the pointer machine model are basically creating new nodes and pointers, manipulating data, and performing comparisons. The crucial restriction in the pointer machine model, which distinguishes it from the word RAM model, is that pointer machines cannot perform address arithmetics, namely, memory access must be performed only by an explicit reference to a pointer. While the pointer machine model is apparently weaker than the word RAM model that supports address arithmetics and unit-cost bit-wise operations, the pointer machine model serves as a good basis for modeling linked structures such as trees and graphs, which are exactly our targets in this paper. In addition, pointer-machines are powerful enough to simulate list-processing based languages such as LISP and Prolog (and their variants), which have recurrently gained attention.

3 Brief reviews on previous algorithms

To understand why and how our new algorithms to be presented in Section 4 work efficiently, let us briefly recall the previous related algorithms.

3.1 Weiner's algorithm and Blumer et al.'s algorithm for a single string

First, we briefly review how Weiner's algorithm for a single string T adds a new leaf to the suffix tree when a new character a is prepended to T. Our description of Weiner's algorithm slightly differs from the original one, in that we use both hard and soft Weiner links while Weiner's original algorithm uses hard Weiner links only and it instead maintains Boolean vectors indicating the existence of soft Weiner links.

Suppose we have already constructed $\mathsf{STree}(T)$ with hard and soft Weiner links. Let ℓ be the leaf that represents T. Given a new character a, Weiner's algorithm climbs up the path from the leaf ℓ until encountering the deepest ancestor v of ℓ that has a Weiner link $\mathsf{W\_link}_a(v)$ defined. If there is no such ancestor of ℓ above, then a new leaf representing aT is inserted from the root r of the suffix tree. Otherwise, the algorithm follows the Weiner link $\mathsf{W\_link}_a(v)$ and arrives at its target node $u = \mathsf{W\_link}_a(v)$. There are two sub-cases:

(1) If $\mathsf{W\_link}_a(v)$ is a hard Weiner link, then a new leaf $\hat{\ell}$ representing aT is inserted from u.
(2) If $\mathsf{W\_link}_a(v)$ is a soft Weiner link, then the algorithm splits the incoming edge of u into two edges by inserting a new node y as a new parent of u such that $|y| = |v| + 1$ (See also Figure 2). A new leaf representing aT is inserted from this new internal node y. We also copy each *out-going* Weiner link $\mathsf{W\_link}_c(u)$ from u with a character c as an out-going Weiner link $\mathsf{W\_link}_c(y)$ from y so that their target nodes are the same (i.e. $\mathsf{W\_link}_c(u) = \mathsf{W\_link}_c(y)$). See also Figure 3. Then, a new *hard* Weiner link is created from v to y with label a, in other words, an old soft Weiner link $\mathsf{W\_link}_a(v) = u$ is *redirected* to a new hard Weiner link $\mathsf{W\_link}_a(v) = y$. In addition, all the old soft Weiner links of ancestors z of v such that $\mathsf{W\_link}_a(z) = u$ in $\mathsf{STree}(T)$ have to be redirected to new soft Weiner links $\mathsf{W\_link}_a(z) = y$ in $\mathsf{STree}(aT)$. These redirections can be done by keeping climbing

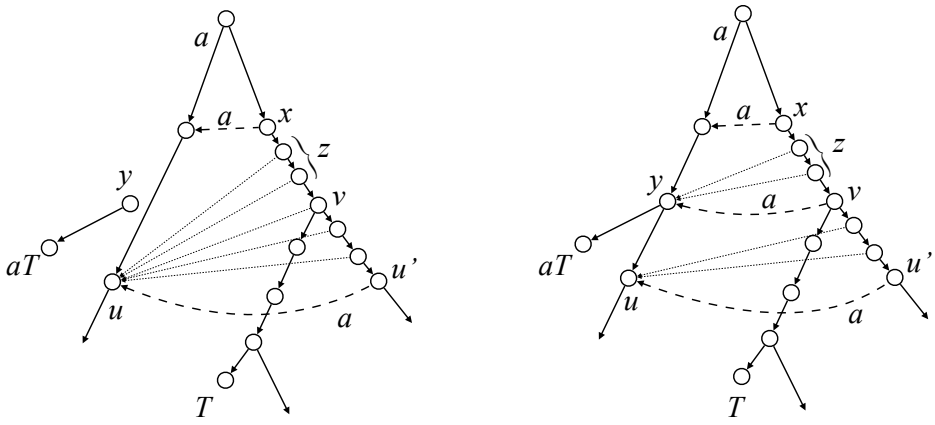


Figure 2. Left: Illustration for $\mathsf{STree}(T)$ of Case (2) before inserting the new leaf representing aT. Right: Illustration for $\mathsf{STree}(aT)$ of Case (2) after inserting the new leaf representing aT. In both diagrams, thick dashed arrows represent hard Winer links, and narrow dashed arrows represent soft Weiner links. All these Winer links are labeled by a. Also, new Weiner links labeled a are created from the nodes between the leaf for T and v to the new leaf for aT (not shown in this diagram).

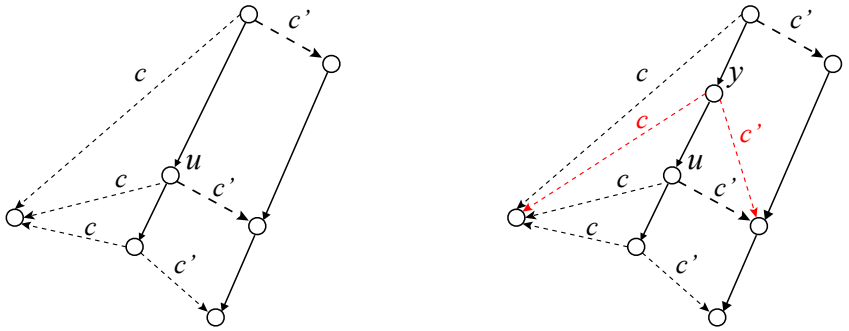


Figure 3. Illustration of the copy process of the out-going Weiner links of u to its new parent y in Case (2). Left: Out-going Weiner links of node u before the update. Right: Each out-going Winer link of node u is copied to its new parent y, represented by a red dashed arrow.

up the path from v until finding the deepest node x that has a hard Weiner link with character a pointing to the parent of u in $\mathsf{STree}(T)$.

In both Cases (1) and (2) above, new soft Weiner links $\mathsf{W\_link}_a(x) = \hat{\ell}$ are created from every node x in the path from ℓ to the child of v.

The running time analysis of the above algorithm has three phases.

(a) In both Cases (1) and (2), the number of nodes from leaf ℓ for T to v is bounded by the number of newly created soft Weiner links. This is amortized $O(1)$ per new character since the resulting suffix tree has a total of $O(n)$ soft Weiner links [2], where $n = |T|$.

(b) In Case (2), the number of out-going Weiner links copied from u to y is bounded by the number of newly created Weiner links, which is also amortized $O(1)$ per new character by the same argument as (a).

(c) In Case (2), the number redirected soft Weiner links is bounded by the number of nodes from v to x. The analysis by Weiner [21] shows that this number of nodes from v to x can be amortized $O(1)$.

Wrapping up (a), (b), and (c), the total numbers of visited nodes, created Weiner links, and redirected Weiner links through constructing $\mathsf{STree}(T)$ by prepending n characters are $O(n)$. Thus Weiner's algorithm constructs $\mathsf{STree}(T)$ in a right-to-left online manner in $O(n \log \sigma)$ time with $O(n)$ space, where the $\log \sigma$ term comes from the cost of maintaining Weiner links of each node in the lexicographically sorted order by e.g. a standard balanced binary search tree.

Since this algorithm correctly maintains all (hard and soft) Weiner links, it builds $\mathsf{DAWG}(S)$ for the reversed string $S = \overline{T}$ in a left-to-right manner, in $O(n \log \sigma)$ time with $O(n)$ space. In other words, this version of Weiner's algorithm is equivalent to Blumer et al.'s DAWG online construction algorithm.

We remark that the aforementioned version of Weiner's algorithm, and equivalently Blumer et al.'s algorithm, work on the pointer machine model as they do not use address arithmetics nor table look-ups.

3.2 Takagi et al.'s algorithm for multiple strings on the word RAM

When Weiner's algorithm is applied to fully-online right-to-left construction of $\mathsf{STree}(\mathcal{T})$, the amortization in Analysis (c) does not work. Namely, it was shown by Takagi et al. [17] that the number of redirected soft Weiner links is $\Theta(N \min(K, \sqrt{N}))$ in the fully-online setting for multiple K strings. A simpler upper bound $O(NK)$ immediately follows from an observation that the insertion of a new leaf for a string T_i in $\mathcal{T}$ may also increase the depths of the leaves for all the other $K-1$ strings $T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_K$ in $\mathcal{T}$. Takagi et al. then obtained the aforementioned improved $O(N \min(K, \sqrt{N}))$ upper bound, and presented a lower bound instance that indeed requires $\Omega(N \min(K, \sqrt{N}))$ work. It should also be noted that the original version of Weiner's algorithm that only maintains Boolean indicators for the existence of soft Weiner links, must also visit $\Theta(N \min(K, \sqrt{N}))$ nodes [17].

Takagi et al. gave a neat way to overcome this difficulty by using the nearest marked ancestor (NMA) data structure [22] for a rooted tree. This NMA data structure allows for making unmarked nodes, splitting edges, inserting new leaves, and answering NMA queries in $O(1)$ amortized time each, in the word RAM model of machine word size $\Omega(\log N)$. Takagi et al. showed how to skip the nodes between v to x in $O(1)$ amortized time using a single NMA query on the NMA data structure associated to a given character a that is prepended to T. They also showed how to store σ NMA data structures for all σ distinct characters in $O(N)$ total space. Since the amortization argument (c) is no more needed by the use of the NMA data structures, and since the analyses (a) and (b) still hold for fully-online multiple strings, the total number of visited nodes was reduced to $O(N)$ in their algorithm. This led to their construction in $O(N \log \sigma)$ time and $O(N)$ space, in the word RAM model.

Takagi et al.'s $\Theta(N \min(K, \sqrt{N}))$ bound also applies to the number of visited nodes and that of redirected secondary edges of $\mathsf{DAWG}(\mathcal{S})$ for multiple strings in the fully-online setting. Instead, they showed how to simulate secondary edge traversals of $\mathsf{DAWG}(\mathcal{S})$ in $O(\log \sigma)$ amortized time each, using the aforementioned NMA structures. We remark that their data structure is only an implicit representation of $\mathsf{DAWG}(\mathcal{S})$ in the sense that the secondary edges are not explicitly stored.

4 Simple fully-online constructions of suffix trees and DAWGs on the pointer-machine model

In this section, we present our new algorithms for fully-online construction of suffix trees and DAWGs for multiple strings, which work on the pointer-machine model.

4.1 Right-to-left suffix tree construction

In this section, we present our new algorithm that constructs the suffix tree for a fully-online right-to-left string collection.

Consider a collection $\mathcal{T}' = \{T_1, \ldots, T_K\}$ of K strings. Suppose that we have built $\mathsf{STree}(\mathcal{T}')$ and that for each string $T_i \in \mathcal{T}'$ we know the leaf ℓ_i that represents T_i.

In our fully-online setting, any new character from Σ can be prepended to any string in the current string collection $\mathcal{T}$. Suppose that a new character $a \in \Sigma$ is prepended to a string T in the collection $\mathcal{T}'$, and let $\mathcal{T} = (\mathcal{T}' \setminus \{T\}) \cup \{aT\}$ be the collection after the update. Our task is to update $\mathsf{STree}(\mathcal{T}')$ to $\mathsf{STree}(\mathcal{T})$.

Our approach is to reduce the sub-problem of redirecting Weiner links to the *ordered split-insert-find problem* that operates on ordered sets over dynamic universe of elements, and supports the following operations and queries efficiently:

- Make-set, which creates a new list that consists only of a single element;
- Split, which splits a given set into two disjoint sets, such that the elements in one set are all smaller than those in the other set;
- Insert, which inserts a new single element into a given set;
- Find, which reports the name of the set that a given element belongs to.

Recall our description of Weiner's algorithm in Section 3.1 and see Figure 2. Consider the set of in-coming Weiner links of node u before updates (the left diagram of Figure 2), and assume that these Weiner links are sorted by the length of the origin nodes. After arriving at the node v in the climbing up process from the leaf for T, we take the Weiner link with character a and arrive at node u. Then we access the set of in-coming Weiner-links of u by a find query. When we create a new internal node y as the parent of the new leaf for aT, we split this set into two sets, one as the set of in-coming Weiner links of y, and the other as the set of in-coming Weiner links of u (see the right diagram of Figure 2). This can be maintained by a single call of a split operation.

Now we pay our attention to the copying process of Weiner links described in Figure 3. Observe that each newly copied Weiner link can be inserted by a single find operation and a single insert operation into the set of in-coming Weiner links of $\mathsf{W\_link}_u(c)$ for each character c where $\mathsf{W\_link}_u(c)$ is defined.

Now we prove the next lemma:

Lemma 1. *Let f denote the operation and query time of a linear-space algorithm for the ordered split-insert-find problem. Then, we can build the suffix tree for a fully-online right-to-left string collection of total length N in a total of $O(N(f + \log \sigma))$ time and $O(N)$ space.*

Proof. The number of split operations is clearly bounded by the number of leaves, which is N. Since the number of Weiner links is at most $3N - 4$, the number of insert operations is also bounded by $3N - 4$. The number of find queries is thus bounded by $N + 3N - 4 = 4N - 4$. By using a linear-space split-insert-find data structure, we

can maintain the set of in-coming Weiner links for all nodes in a total of $O(Nf)$ time with $O(N)$ space.

Given a new character a to prepend to a string T, we climb up the path from the leaf for T and find the deepest ancestor v of the leaf for which $\mathsf{W\_link}_a(v)$ is defined. This can be checked in $O(\log \sigma)$ time at each visited node, by using a balanced search tree. Since we do not climb up the nodes z (see Figure 2) for which the soft Weiner links with a are redirected, we can use the same analysis (a) as in the case of a single string. This results in that the number of visited nodes in our algorithm is $O(N)$. Hence we use $O(N \log \sigma)$ total time for finding the deepest node which has a Weiner link for the prepended character a.

Overall, our algorithm uses $O(N(f + \log \sigma))$ time and $O(N)$ space. □

Our ordered split-insert-find problem is a special case of the union-split-find problem on ordered sets, since each insert operation can be trivially simulated by make-set and union operations. Link-cut trees of Sleator and Tarjan [15] for a dynamic forest support make-tree, link, cut operations and find-root queries in $O(\log d)$ time each. Since link-cut trees can be used to path-trees, make-set, insert, split, and find in the ordered split-insert-find problem can be supported in $O(\log d)$ time each. Since link-cut trees work on the pointer machine model, this leads to a pointer-machine algorithm for our fully-online right-to-left construction of the suffix tree for multiple strings with $f = O(\log d)$. Here, in our context, d denotes the maximum number of in-coming Weiner links to a node of the suffix tree.

A potential drawback of using link-cut trees is that in order to achieve $O(\log d)$-time operations and queries, link-cut trees use some auxiliary data structures such as splay trees [16] as its building block. Yet, in what follows, we show that our ordered split-insert-find problem can be solved by a simpler balanced tree, AVL-trees [1], retaining $O(N(\log \sigma + \log d))$-time and $O(N)$-space complexities.

Theorem 2. *There is an AVL-tree based pointer-machine algorithm that builds the suffix tree for a fully-online right-to-left multiple strings of total length N in $O(N(\log \sigma + \log d))$ time with $O(N)$ space, where d is the maximum number of in-coming Weiner links to a suffix tree node and σ is the alphabet size.*

Proof. For each node u of the suffix tree $\mathsf{STree}(\mathcal{T}')$ before update, let $S(u) = \{|x| \mid \mathsf{W\_link}_a(x) = u\}$ where $a = u[1]$, namely, $S(u)$ is the set of the string depths of the origin nodes of the in-coming Weiner links of u. We maintain an AVL tree for $S(u)$ with the node u, so that each in-coming Weiner link for u points to the corresponding node in the AVL tree for $S(u)$. The root of the AVL tree is always linked to the suffix tree node u, and each time another node in the AVL tree becomes the new root as a result of node rotations, we adjust the link so that it points to u from the new root of the AVL tree.

This way, a find query for a given Weiner link is reduced to accessing the root of the AVL tree that contains the given Weiner link, which can be done in $O(\log S(u)) \subseteq O(\log d)$ time.

Inserting a new element to $S(u)$ can also be done in $O(\log S(u)) \subseteq O(\log d)$ time.

Given an integer k, let S_1 and S_2 denote the subset of $S(u)$ such that any element in S_1 is not larger than k, any element in S_2 is larger than k, and $S_1 \cup S_2 = S(u)$. It is well known that we can split the AVL tree for $S(u)$ into two AVL trees for S_1 and for S_2 in $O(\log S(u)) \subseteq O(\log d)$ time (c.f. [13]). In our context, k is the string depth of the deepest node v that is a Weiner link with character a in the upward path from

the leaf for T. This allows us to maintain $S_1 = S(y)$ and $S_2 = S(u)$ in $O(\log d)$ time in the updated suffix tree $\mathsf{STree}(\mathcal{T})$.

When we create the in-coming Weiner links labeled a to the new leaf $\hat{\ell}$ for aT, we first perform a make-set operation which builds an AVL tree consisting only of the root. If we naïvely insert each in-coming Weiner link to the AVL tree one by one, then it takes a total of $O(N \log d)$ time. However, we can actually perform this process in $O(N)$ total time even on the pointer machine model: Since we climb up the path from the leaf ℓ for T, the in-coming Weiner links are already sorted in decreasing order of the string depths of the origin nodes. We create a family of maximal complete binary trees of size $2^h - 1$ each, arranged in decreasing order of h. This can be done as follows: Initially set $r \leftarrow |S(\hat{\ell})|$. We then greedily take the largest h such that $2^h - 1 \leq r$, and then update $r \leftarrow r - (2^h - 1)$ and search for the next largest h and so on. These trees can be easily created in $O(|S(\hat{\ell})|)$ total time by a simple linear scan over the sorted list of the in-coming Weiner links. Since the heights h of these complete binary search trees are monotonically decreasing, and since all of these binary search trees are AVL trees, one can merge all of them into a single AVL tree in time linear in the height of the final AVL tree (c.f. [13]), which is bounded by $O(h) = O(\log S(\hat{\ell}))$. Thus, we can construct the initial AVL tree for the in-coming Weiner links of each new leaf $\hat{\ell}$ in $O(|S(\hat{\ell})|)$ time. Since the total number of Weiner links is $O(N)$, we can construct the initial AVL trees for the in-coming Weiner links of all new leaves in $O(N)$ total time.

Overall, our algorithm works in $O(N(\log \sigma + \log d))$ time with $O(N)$ space. □

4.2 Left-to-right DAWG construction

The next theorem immediately follows from Theorem 2.

Theorem 3. *There is an AVL-tree based pointer-machine algorithm that builds an explicit representation of the DAWG for a fully-online left-to-right multiple strings of total length N in $O(N(\log \sigma + \log d))$ time with $O(N)$ space, where d is the maximum number of in-coming edges of a DAWG node and σ is the alphabet size. This representation of the DAWG allows each edge traversal in $O(\log \sigma + \log d)$ time.*

Proof. The correctness and the complexity of construction are immediate from Theorem 2.

Given a character a and a node v in the DAWG, we first find the out-going edge of v labeled a in $O(\log \sigma)$ time. If it does not exist, we terminate. Otherwise, we take this a-edge and arrive at the corresponding node in the AVL tree for the destination node u for this a-edge. We then perform a find query on the AVL tree and obtain u in $O(\log d)$ time. □

We emphasize that Theorem 3 gives the *first* non-trivial algorithm that builds an explicit representation of the DAWG for fully-online multiple strings. Recall that a direct application of Blumer et al.'s algorithm to the case of fully-online K multiple strings requires to visit $\Theta(N \min(K, \sqrt{N}))$ nodes in the DAWG, which leads to $O(N \min(K, \sqrt{N}) \log \sigma) = O(N^{1.5} \log \sigma)$-time construction for $K = \Theta(\sqrt{N})$.

It should be noted that after all the N characters have been processed, it is easy to modify, in $O(N)$ time in an offline manner, this representation of the DAWG so that each edge traversal takes $O(\log \sigma)$ time.

4.3 On optimality of our algorithms

It is known that sorting a length-N sequence of σ distinct characters is an obvious lower bound for building the suffix tree [7] or alternatively the DAWG. This is because, when we build the suffix tree or the DAWG where the out-going edges of each node are sorted in the lexicographical order, then we can obtain a sorted list of characters at their root. Thus, $\Omega(N \log \sigma)$ is a comparison-based model lower bound for building the suffix tree or the DAWG. Since Takagi et al.'s $O(N \log \sigma)$-time algorithm [17] works only on the word RAM model, in which faster integer sorting algorithms exist, it would be interesting to explore some cases where our $O(N(\log \sigma + \log d))$-time algorithms for a weaker model of computation can perform in optimal $O(N \log \sigma)$ time.

It is clear that the maximum number d of in-coming Weiner links to a node is bounded by the total length N of the strings. Hence, in case of integer alphabets of size $\sigma = N^{O(1)}$, our algorithms run in optimal $O(N \log \sigma) = O(N \log N)$ time.

For the case of smaller alphabet size $\sigma = \text{polylog}(N)$, the next lemma can be useful:

Lemma 4. *The maximum number d of in-coming Weiner links is less than the height of the suffix tree.*

Proof. For any node u in the suffix tree, all in-coming Weiner links to u is labeled by the same character a, which is the first character of the substring represented by u. Therefore, all in-coming Weiner links to u are from the nodes in the path between the root and the node $u[2..|u|]$. □

We note that the height of the suffix tree for multiple strings is bounded by the length of the longest string in the collection. In many applications such as time series from sensor data, it would be natural to assume that all the K strings in the collection have similar lengths. Hence, when the collection consists of $K = N/\text{polylog}(N)$ strings of length $\text{polylog}(N)$ each, we have $d = \text{polylog}(N)$. In such cases, our algorithms run in optimal $O(N \log \sigma) = O(N \log \log N)$ time.

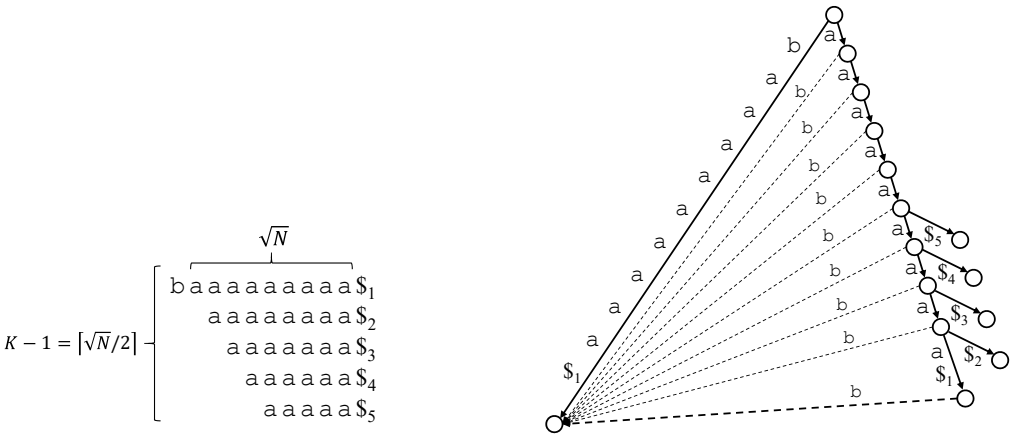


Figure 4. Left: The $K-1 = \lceil \sqrt{N}/2 \rceil$ strings where character `b` has been prepended only to the first string T_1. Right: The corresponding part of the suffix tree. Dashed arrows represent Weiner links with character `b`.

The next lemma shows some instance over a binary alphabet of size $\sigma = 2$, which requires a certain amount of work for the splitting process.

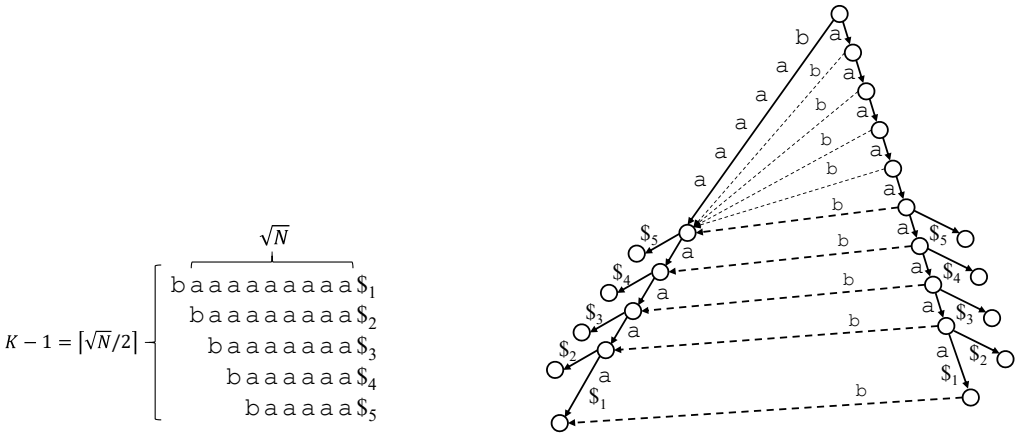


Figure 5. Left: The $K-1 = \lceil\sqrt{N}/2\rceil$ strings where character b has been prepended to all of them. Right: The corresponding part of the suffix tree after the updates. Each time a new leaf is created, $\Theta(\sqrt{N})$ in-coming Weiner links were involved in a split operation on the AVL tree and it takes $O(\log N)$ time.

Lemma 5. *There exist a set of fully-online multiple strings over a binary alphabet such that the node split procedure of our algorithms takes $O(\sqrt{N}\log N)$ time.*

Proof. Let $K = 1 + \lceil\sqrt{N}/2\rceil$.

For the time being, we assume that each string T_i is terminated with a unique symbol $\$_i$. Consider a subset $\{T_1, \ldots, T_{K-1}\}$ of $K-1 = \lceil\sqrt{N}/2\rceil$ strings such that for each $1 \le i \le K-1$, $T_i = \mathtt{a}^{\sqrt{N}-i+1}\$_i$. We then prepend the other character b from the binary alphabet $\{\mathtt{a}, \mathtt{b}\}$ to each T_i in increasing order of $i = 1, \ldots, K-1$. For $i = 1$, $\sqrt{N}$ Weiner links to the new leaf for $\mathtt{b}T_1 = \mathtt{ba}^{\sqrt{N}}\$_1$, each labeled b, are created. See Figure 4 for illustration of this step.

Then, for each $i = 2, \ldots, K-1$, inserting a new leaf for $\mathtt{b}T_i$ requires an insertion of a new internal node as the parent of the new leaf. This splits the set of in-coming Weiner links into two sets: one is a singleton consisting of the Winer link from node $\mathtt{a}^{\sqrt{N}-i+1}$, and the other consists of the Weiner links from the shallower nodes. Each of these $K-2$ split operations can be done by a simple deletion operation on the corresponding AVL tree, using $O(\log\sqrt{N}) = O(\log N)$ time each. See Figure 5 for illustration.

Observe also that the same analysis holds even if we remove the terminal symbol $\$_i$ from each string T_i (in this case, there is a non-branching internal node for each T_i and we start the climbing up process from this internal node).

The total length of these $K-1$ strings is approximately $3N/8$. We can arbitrarily choose the last string T_K of length approximately $5N/8$ so that it does not affect the above split operations (e.g., a unary string $a^{5N/8}$ or $b^{5N/8}$ would suffice).

Thus, there exists an instance over a binary alphabet for which the node split operations require $O(\sqrt{N}\log N)$ total time. □

Since $\sqrt{N}\log N = o(N)$, the $\sqrt{N}\log N$ term is always dominated by the $N\log\sigma$ term. It is left open whether there exists a set of strings with $\Theta(N)$ character additions, each of which requires splitting a set that involves $N^{O(1)}$ in-coming Weiner links. If such an instance exists, then our algorithm must take $\Theta(N\log N)$ time in the worst case.

5 Conclusions and future work

In this paper we considered the problem of maintaining the suffix tree and the DAWG indexing structures for a collection of multiple strings that are updated in a fully-online manner, where a new character can be added to the left end or the right end of any string in the collection, respectively. Our contributions are simple pointer-machine algorithms that work in $O(N(\log \sigma + \log d))$ time and $O(N)$ space, where N is the total length of the strings, σ is the alphabet size, and d is the maximum number of in-coming Weiner links of a node in the suffix tree. The key idea was to reduce the sub-problem of re-directing in-coming Weiner links to the ordered split-insert-find problem, which we solved in $O(\log d)$ time by AVL trees. We also discussed the cases where our $O(N(\log \sigma + \log d))$-time solution is optimal.

A major open question regarding the proposed algorithms is whether there exists an instance over a small alphabet which contains $\Theta(N)$ positions each of which requires $\Theta(\log N)$ time for the split operation, or requires $\Theta(N)$ insertions each taking $\Theta(\log N)$ time. If such instances exist, then the running time of our algorithms may be worse than the optimal $O(N \log \sigma)$ for small σ. So far, we have only found an instance with $\sigma = 2$ that takes sub-linear $O(\sqrt{N} \log N)$ total time for split operations.

Acknowledgements

This work is supported by JSPS KAKENHI Grant Number JP17H01697 and JST PRESTO Grant Number JPMJPR1922.

References

1. G. Adelson-Velsky and E. Landis: *An algorithm for the organization of information.* Proceedings of the USSR Academy of Sciences (in Russian), 146 1962, pp. 263–266, English translation by Myron J. Ricci in Soviet Mathematics - Doklady, 3:1259–1263, 1962.
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.* J. ACM, 34(3) 1987, pp. 578–595.
4. M. T. Chen and J. Seiferas: *Efficient and elegant subword-tree construction*, in Combinatorial Algorithms on Words, vol. 12 of NATO ASI Series, 1985, pp. 97–107.
5. M. Crochemore and W. Rytter: *Text Algorithms*, Oxford University Press, 1994.
6. H. H. Do and W. Sung: *Compressed directed acyclic word graph with application in local alignment.* Algorithmica, 67(2) 2013, pp. 125–141.
7. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan: *On the sorting-complexity of suffix tree construction.* J. ACM, 47(6) 2000, pp. 987–1011.
8. Y. Fujishige, Y. Tsujimaru, S. Inenaga, H. Bannai, and M. Takeda: *Computing DAWGs and minimal absent words in linear time for integer alphabets*, in MFCS 2016, 2016, pp. 38:1–38:14.
9. H. N. Gabow and R. E. Tarjan: *A linear-time algorithm for a special case of disjoint set union.* Journal of Computer and System Sciences, 30 1985, pp. 209–221.
10. D. Gusfield and J. Stoye: *Linear time algorithms for finding and representing all the tandem repeats in a string.* J. Comput. Syst. Sci., 69(4) 2004, pp. 525–546.
11. D. Hendrian, S. Inenaga, R. Yoshinaka, and A. Shinohara: *Efficient dynamic dictionary matching with DAWGs and AC-automata.* Theor. Comput. Sci., 792 2019, pp. 161–172.
12. H. Imai and T. Asano: *Dynamic orthogonal segment intersection search.* J. Algorithms, 8(1) 1987, pp. 1–18.

13. D. Knuth: *The Art Of Computer Programming, vol. 3: Sorting And Searching, Second Edition*, Addison-Wesley, 1998.
14. 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.
15. D. D. Sleator and R. E. Tarjan: *A data structure for dynamic trees.* J. Comput. Syst. Sci., 26(3) 1983, pp. 362–391.
16. D. D. Sleator and R. E. Tarjan: *Self-adjusting binary search trees.* J. ACM, 32(3) 1985, pp. 652–686.
17. T. Takagi, S. Inenaga, H. Arimura, D. Breslauer, and D. Hendrian: *Fully-online suffix tree and directed acyclic word graph construction for multiple texts.* Algorithmica, 82(5) 2020, pp. 1346–1377.
18. Y. Tanimura, Y. Fujishige, T. I, S. Inenaga, H. Bannai, and M. Takeda: *A faster algorithm for computing maximal α-gapped repeats in a string*, in SPIRE 2015, 2015, pp. 124–136.
19. R. E. Tarjan: *A class of algorithms which require nonlinear time to maintain disjoint sets.* J. Comput. Syst. Sci., 18(2) 1979, pp. 110–127.
20. R. E. Tarjan: *Data structures and network algorithms*, vol. 44 of CBMS-NSF regional conference series in applied mathematics, SIAM, 1983.
21. P. Weiner: *Linear pattern-matching algorithms*, in Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory, 1973, pp. 1–11.
22. J. Westbrook: *Fast incremental planarity testing*, in ICALP 1992, 1992, pp. 342–353.
23. J. Yamamoto, T. I, H. Bannai, S. Inenaga, and M. Takeda: *Faster compact on-line Lempel-Ziv factorization*, in STACS 2014, 2014, pp. 675–686.

Simple KMP Pattern-Matching on Indeterminate Strings*

Neerja Mhaskar[1] and W. F. Smyth[1,2]

[1] Algorithms Research Group, Department of Computing & Software
McMaster University, Canada
pophlin@mcmaster.ca, smyth@mcmaster.ca
[2] School of Engineering & Information Technology
Murdoch University, Western Australia

Abstract. In this paper we describe a simple, fast, space-efficient approach to finding all matches of an indeterminate pattern $\boldsymbol{p} = \boldsymbol{p}[1..m]$ in an indeterminate string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, where both $\boldsymbol{p}$ and $\boldsymbol{x}$ are defined on a "small" ordered alphabet Σ — say, $\sigma = |\Sigma| \leq 9$. A preprocessing phase replaces Σ by an integer alphabet Σ_I of size $\sigma_I = \sigma$ that (reversibly, in time linear in string length) maps both $\boldsymbol{x}$ and $\boldsymbol{p}$ into equivalent regular strings $\boldsymbol{y}$ and $\boldsymbol{q}$, respectively, on Σ_I, whose maximum (indeterminate) letter can be expressed in a 32-bit word (for $\sigma \leq 4$, thus for DNA sequences, an 8-bit representation suffices). We then describe an efficient version KMP_INDET of the venerable Knuth-Morris-Pratt algorithm to find all occurrences of $\boldsymbol{q}$ in $\boldsymbol{y}$ (that is, of $\boldsymbol{p}$ in $\boldsymbol{x}$), but, whenever necessary, using the prefix array, rather than the border array, to control shifts of the transformed pattern $\boldsymbol{q}$ along the transformed string $\boldsymbol{y}$. Although requiring $\mathcal{O}(m^2 n)$ time in the theoretical worst case, in cases of practical interest KMP_INDET executes in $\mathcal{O}(n)$ time. A noteworthy feature is the very small additional space requirement: $\Theta(m)$ words in all cases. We conjecture that a similar approach may yield practical and efficient indeterminate equivalents to other well-known pattern-matching algorithms, especially Boyer-Moore and its variants.

Keywords: indeterminate, degenerate, conservative degenerate, pattern-matching, KMP, indeterminate encoding

1 Introduction

Given a fixed finite alphabet $\Sigma = \{\lambda_1, \lambda_2, \ldots, \lambda_\sigma\}$, a ***regular letter***, also called a ***character***, is any single element of Σ, while an ***indeterminate letter*** is any subset of Σ of cardinality greater than one. A ***regular string*** $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on Σ is an array of regular letters drawn from Σ. An ***indeterminate string*** $\boldsymbol{x}[1..n]$ on Σ is an array of letters drawn from Σ, of which at least one is indeterminate. Whenever entries $\boldsymbol{x}[i]$ and $\boldsymbol{x}[j]$, $1 \leq i, j \leq n$, both contain the same character (possibly other characters as well), we say that $\boldsymbol{x}[i]$ ***matches*** $\boldsymbol{x}[j]$ and write $\boldsymbol{x}[i] \approx \boldsymbol{x}[j]$.

In this paper we describe a simple transformation of Σ that permits all subsets of Σ to be replaced by single integer values, while maintaining matches and non-matches between all transformed entries $\boldsymbol{x}[i_1]$ and $\boldsymbol{x}[i_2], 1 \leq i_1, i_2 \leq n$, in $\boldsymbol{x}$. The method is effective on small alphabets (say $|\Sigma| = \sigma \leq 9$), including in particular the important case of DNA sequences ($\Sigma_{DNA} = \{a, c, g, t\}$). Thus, in many cases, cumbersome and time-consuming matches of indeterminate letters can be efficiently handled. For background on pattern-matching in indeterminate strings, see [6,1,8,11,9,2,16,17,4,5].

* Supported by Grant No. 105–36797 from the Natural Sciences & Engineering Research Council of Canada (NSERC). Also the authors thank anonymous reviewers for several valuable suggestions.

Neerja Mhaskar, W. F. Smyth: Simple KMP Pattern-Matching on Indeterminate Strings, pp. 125–133.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9

We will make use of a mapping $\mu_j \leftarrow \lambda^{(j)},\ j = 1, 2, \ldots, \sigma$, of the letters $\lambda^{(j)}$ of Σ chosen in some order, where μ_j is the j^{th} prime number ($\mu_1 = 2, \mu_2 = 3$, and so on). Then, given $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on Σ (the ***source string***), we can apply the mapping to compute $\boldsymbol{y} = \boldsymbol{y}[1..n]$ (the ***mapped string***) according to the following rule:

(R) For every $\boldsymbol{x}[i] = \{\lambda_1, \lambda_2, \ldots, \lambda_k\}$, $1 \leq k \leq \sigma$, $1 \leq i \leq n$, where $\lambda_h \in \Sigma, 1 \leq h \leq k$, set

$$\boldsymbol{y}[i] \leftarrow \prod_{h=1}^{k} \mu_{\lambda_h}.$$

When $k = \sigma$, $\boldsymbol{y}$ achieves the maximum value, which we denote by $P_\sigma = \prod_{j=1}^{\sigma} \mu_j$ (often called a ***hole*** [2]). More generally, since the mapping π yields all possible products of the first σ prime numbers, it imposes an order on indeterminate letters drawn from Σ: $\boldsymbol{x}[i_1] < \boldsymbol{x}[i_2] \Leftrightarrow \boldsymbol{y}[i_1] < \boldsymbol{y}[i_2]$.

For example, consider a DNA source string $\boldsymbol{x} = a\{a, c\}g\{a, t\}t\{c, g\}$, over Σ_{DNA}. Then $\sigma = 4$, and applying (R) for $1 \leq k \leq 4$ (based on the mapping $\mu : 2 \leftarrow a, 3 \leftarrow c, g \leftarrow 5, 7 \leftarrow t$), we compute a mapped string $\boldsymbol{y} = 2/6/5/14/7/15$, so that

$$a < g < \{a, c\} < t < \{a, t\} < \{c, g\}.$$

On the other hand, a different mapping (say, $\mu : 2 \leftarrow t, 3 \leftarrow c, 5 \leftarrow a, 7 \leftarrow g$) would yield $\boldsymbol{y} = 5/15/7/10/2/21$ and a quite different ordering

$$t < a < g < \{a, t\} < \{a.c\} < \{c, g\}.$$

Lemma 1 *Let k_i denote the number of letters in $\boldsymbol{x}[i]$. Then Rule (R) computes $\boldsymbol{y}$ in time $\Theta(K_{\boldsymbol{x}})$, where $K_{\boldsymbol{x}} = \sum_{i=1}^{n} k_i$.*

Note that when the letters in $\boldsymbol{x}$ are strongly indeterminate — that is, $K_{\boldsymbol{x}} \in \Theta(\sigma n)$ —, then the approach proposed here (replacing $\boldsymbol{x}$ by $\boldsymbol{y}$) has the advantage that subsequent processing of $\boldsymbol{y}$ requires access only to a single integer at each position.

Lemma 2 *If $\boldsymbol{y}$ is computed from $\boldsymbol{x}$ by Rule (R), then for every $i_1, i_2 \in 1..n$, $\boldsymbol{x}[i_1] \approx \boldsymbol{x}[i_2]$ if and only if $\gcd(\boldsymbol{y}[i_1], \boldsymbol{y}[i_2]) > 1$.*

Proof.
($\Rightarrow$) By contradiction. Suppose $\boldsymbol{x}[i_1] \approx \boldsymbol{x}[i_2]$, $1 \leq i_1, i_2 \leq n$, but $\gcd(\boldsymbol{y}[i_1], \boldsymbol{y}[i_2]) = 1$; that is, $\boldsymbol{y}[i_1]$ and $\boldsymbol{y}[i_2]$ have no common divisor. Since for every i, the letter $\boldsymbol{y}[i]$ is a product of the prime numbers assigned to the characters in $\boldsymbol{x}[i]$, we see that therefore $\boldsymbol{x}[i_1]$ and $\boldsymbol{x}[i_2]$ can have no character in common; that is, $\boldsymbol{x}[i_1] \not\approx \boldsymbol{x}[i_2]$, a contradiction.
($\Leftarrow$) By the reverse argument. □

Two strings $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$ of equal length n are said to be ***isomorphic*** if and only if for every $i, j \in \{1, \ldots, n\}$,

$$\boldsymbol{x_1}[i] \approx \boldsymbol{x_1}[j] \iff \boldsymbol{x_2}[i] \approx \boldsymbol{x_2}[j]. \tag{1}$$

We thus have:

Observation 3 *If $\boldsymbol{x}$ is an indeterminate string on Σ, and $\boldsymbol{y}$ is the numerical string constructed by applying Rule (R) to $\boldsymbol{x}$, then $\boldsymbol{x}$ and $\boldsymbol{y}$ are isomorphic.*

Observation 4 *By virtue of Lemma 2 and (1),* $\boldsymbol{y}$ *can overwrite the space required for* $\boldsymbol{x}$ *(and vice versa) with no loss of information.*

Observation 5 *Suppose* ℓ_1 *and* ℓ_2 *are integers representable in at most* B *bits. Then* $\gcd(\ell_1, \ell_2)$ *can be computed in time bounded by* $\mathcal{O}(M_B \log B)$[1]*, where* M_B *denotes the maximum time required to compute* $\ell_1 \ell_2$ *over all such integers.*

Observation 6 *For* $\sigma = 9$ *corresponding to the first nine prime numbers*

$$2, 3, 5, 7, 11, 13, 17, 19, 23$$

$P_\sigma = 223,092,870$*, a number representable in less than* $B = 32$ *bits, a single computer word. Thus by Observation 5, the time required to match any two indeterminate letters is bounded by* $\mathcal{O}(5M_{32})$*. When* $\sigma = 4$*, corresponding to* Σ_{DNA}*,* $2 \times 3 \times 5 \times 7 = 210 < 256$*, and so* $B = 8$ *and the matching time reduces to* $\mathcal{O}(3M_8)$*.*

Observation 7 *We assume therefore that, for* $\sigma \leq 9$*, computing a match between* $\boldsymbol{x}[i_1]$ *and* $\boldsymbol{x}[i_2]$ *on* Σ *(that is, between* $\boldsymbol{y}[i_1]$ *and* $\boldsymbol{y}[i_2]$ *computed using Rule (R)) requires time bounded above by a (small) constant.*

Other models to represent indeterminate strings have been proposed [14,10]. For example, the model proposed in [14] maps all the non-empty letters (both regular and indeterminate) over the DNA alphabet $\Sigma_{DNA} = \{A, C, G, T\}$ to the IUPAC symbols $\Sigma_{IUPAC} = \{A, C, G, T, R, Y, S, W, K, M, B, D, H, V, N\}$. Then given an indeterminate string over Σ_{DNA}, it constructs an isomorphic regular string over Σ_{IUPAC}. In [10], the model proposed maps each symbol in the DNA alphabet to a 4-bit integer power of 2; that is, $\{A, C, G, T\}$ is mapped to $\{2^0, 2^1, 2^2, 2^3\}$. Then a non-empty indeterminate letter over Σ_{DNA} is represented as $\Sigma_{\{s \in \mathcal{P}(\Sigma)\}} s$ of maximum size $15 = 1111_2$. Also, instead of using the natural order on integers, [10] uses a Gray code [7] to order indeterminate letters over Σ_{DNA}. Note that with the Gray code two successive values differ by only one bit, such as 1100 and 1101, which enables minimizing the number of separate intervals associated with each of the four symbols of Σ_{DNA}.

2 Pattern Matching Algorithm for Indeterminate Strings

In this section we describe a simple, fast, space-efficient algorithm KMP_INDET that, in order to compute all occurrences of a source pattern $\boldsymbol{p} = \boldsymbol{p}[1..m]$ in a source string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, computes all the positions at which the corresponding mapped pattern $\boldsymbol{q} = \boldsymbol{q}[1..m]$ occurs in the mapped string $\boldsymbol{y} = \boldsymbol{y}[1..n]$. We begin with the following result:

Lemma 8 *For alphabet* Σ *of size* $\sigma \leq 9$*, the positions of occurrence of* $\boldsymbol{p}$ *in* $\boldsymbol{x}$ *can be computed in* $O(mn)$ *time.*

Proof. By Lemma 2 and Observation 5, the positions of occurrence of $\boldsymbol{q}$ in $\boldsymbol{y}$ can be trivially computed in $O(mnM_B \log B) = O(mn)$ time, with constant of proportionality $M_B \log B$. □

[1] https://en.wikipedia.org/wiki/Greatest_common_divisor#Complexity

As noted earlier, many pattern matching algorithms have been proposed for indeterminate strings. In [12] Iliopoulos and Radoszewski propose an $\mathcal{O}(n \log m)$ algorithm for a constant alphabet. This is the best theoretical bound known so far for pattern matching algorithms on indeterminate strings over a constant alphabet. Recently, pattern matching algorithms for ***conservative*** indeterminate strings, where the number of indeterminate letters in text $\boldsymbol{x}$ and pattern $\boldsymbol{p}$ is bounded above by a constant k, have been proposed [4,5]. In [4], Crochemore et. al present an $\mathcal{O}(nk)$ algorithm which uses suffix trees and other auxiliary data structures to search for $\boldsymbol{p}$ in $\boldsymbol{x}$. In [5], Daykin et. al propose a pattern matching algorithm by first constructing the Burrows Wheeler Transform (BWT) of $\boldsymbol{x}$ in $\mathcal{O}(mn)$ time, and use it to find all occurrences of $\boldsymbol{p}$ in $\boldsymbol{x}$ in $\mathcal{O}(km^2 + q)$ time, where q is the number of occurrences of the pattern in $\boldsymbol{x}$, and $\mathcal{O}(km^2)$ is the time required to compute it.

2.1 Definitions

We give here a few essential definitions.

Given $\boldsymbol{x}[1..n]$, then for $1 \leq i \leq n$ and $1 \leq j \leq n$, $\boldsymbol{u} = \boldsymbol{x}[i..j]$ is called a ***substring*** of $\boldsymbol{x}$, an ***empty*** substring $\boldsymbol{\varepsilon}$ if $j < i$. If $i = 1$, $\boldsymbol{u}$ is a ***prefix*** of $\boldsymbol{x}$, a ***suffix*** if $j = n$. A string $\boldsymbol{x}$ has a ***border*** $\boldsymbol{u}$ if $|\boldsymbol{u}| < |\boldsymbol{x}|$ and $\boldsymbol{x}$ has both prefix and suffix equal to $\boldsymbol{u}$. Note that a border of $\boldsymbol{x}$ may be empty.

A ***border array*** $\beta_{\boldsymbol{x}} = \beta_{\boldsymbol{x}}[1..n]$ of $\boldsymbol{x}$ is an integer array where for every $i \in [1..n]$, $\beta_{\boldsymbol{x}}[i]$ is the length of the longest border of $\boldsymbol{x}[1..i]$. A ***prefix array*** $\pi_{\boldsymbol{x}} = \pi_{\boldsymbol{x}}[1..n]$ of $\boldsymbol{x}$ is an integer array where for every $i \in [1..n]$, $\pi_{\boldsymbol{x}}[i]$ is the length of the longest substring starting at position i that matches a prefix of $\boldsymbol{x}$. See Figure 1 for an example.

	1	2	3	4	5	6	7	8	9	10	11	12	13
$\boldsymbol{x}$	a	a	b	a	a	b	a	a	{a, b}	b	a	a	{a, c}
$\beta_{\boldsymbol{x}}$	0	1	0	1	2	3	4	5	6	3	4	5	2
$\pi_{\boldsymbol{x}}$	13	1	0	6	1	0	3	5	1	0	2	2	1

Figure 1. Border array $\beta_{\boldsymbol{x}}$, and Prefix array $\pi_{\boldsymbol{x}}$ computed for the string $\boldsymbol{x} = aabaabaa\{a, b\}baa\{a, c\}$.

In Lemmas 9 and 10, we rephrase earlier results on running times for computing the border array and prefix array of a string of length n.

Lemma 9 ([15,16]) *The border array and prefix array of a regular string of length m can be computed in $\mathcal{O}(m)$ time.*

Lemma 10 ([15,16]) *The border array and prefix array of an indeterminate string of length m can be computed in $\mathcal{O}(m^2)$ time in the worst-case, $\mathcal{O}(m)$ in the average case.*

For completeness we give in Figure 2 the KMP algorithm for regular strings $\boldsymbol{x} = \boldsymbol{x}[1..n]$. In case of a mismatch or after a full match, KMP computes the shift of the pattern $\boldsymbol{p} = \boldsymbol{p}[1..m]$ along $\boldsymbol{x}$ by using the border array of $\boldsymbol{p}$, which as we have seen is computable in $\mathcal{O}(m)$ time. Thus KMP runs in $\mathcal{O}(n)$ time.

```
function KMP(x, n, p, m) : Integer List
i ← 0; j ← 0
indexlist ← ∅ ▷ List of indices where p occurs in x
β_p ← Border array of pattern p
while i < n do
    if p[j + 1] = x[i + 1] then
        j ← j + 1; i ← i + 1
        if j = m then
            indexlist ← indexlist ∪ {i−j+1}
            j ← β_p[j]
    else
        if j = 0 then i ← i+1
        else
            j ← β_p[j]
return indexlist
```

Figure 2. KMP checks whether the regular pattern $\boldsymbol{p}$ occurs in the regular text $\boldsymbol{x}$. If it does, then it outputs the set of indices at which p occurs in $\boldsymbol{x}$; otherwise returns an empty set.

2.2 KMP Algorithm for Indeterminate Strings

We now describe KMP_INDET (see Figure 3), which searches for pattern $\boldsymbol{q} = \boldsymbol{q}[1..m]$ in text $\boldsymbol{y} = \boldsymbol{y}[1..n]$, outputting the indices at which $\boldsymbol{q}$ occurs in $\boldsymbol{y}$ (thus, at which $\boldsymbol{p}$ occurs in $\boldsymbol{x}$). Thus our algorithm implements the KMP algorithm [13] on indeterminate strings that have been transformed using Rule (R). However, note that this transformation is not necessary for the algorithm to work: we use it to improve space and time efficiency. The algorithm also works with other indeterminate string encoding/transformations mentioned in the previous section. While scanning $\boldsymbol{y}$ from left to right and performing letter comparisons, KMP_INDET checks whether the prefix of $\boldsymbol{q}$ and the substring of $\boldsymbol{y}$ currently being matched are both regular. If so, then it uses the border array $\beta_{\boldsymbol{q}_\ell}$ of the longest regular prefix $\boldsymbol{q}_\ell$ of $\boldsymbol{q}$ of length ℓ, to compute the shift; if not, it constructs a new string $\boldsymbol{q'}$, which is a concatenation of the longest proper prefix of the matched prefix of $\boldsymbol{q}$ and the longest proper suffix of the matched substring of $\boldsymbol{y}$, using the prefix array $\pi_{\boldsymbol{q'}}$ of $\boldsymbol{q'}$ to compute the shift. The COMPUTE_SHIFT function given in Figure 4 implements this computation.

In order to determine whether or not indeterminate letters are included in any segment $\boldsymbol{q'} = \boldsymbol{q}[1..j{-}1]\boldsymbol{y}[i{-}j{+}2..i]$, two variables are employed: $indet_y$ and the length ℓ of the longest regular prefix $\boldsymbol{q}_{\boldsymbol{\ell}}$ of $\boldsymbol{q}$. $indet_y$ is a Boolean variable that is true if and only if the current segment $\boldsymbol{y}[i{-}j{+}2..i]$ contains an indeterminate letter; ℓ is pre-computed in $\mathcal{O}(m)$ time as a byproduct of the one-time calculation of $\boldsymbol{q}_{\boldsymbol{\ell}}$.

If $\boldsymbol{y}$ and $\boldsymbol{q}$ are both regular, then KMP_INDET reduces to the KMP algorithm [13]. Otherwise, it checks whether indeterminate letters exist in the matched prefix of $\boldsymbol{q} = \boldsymbol{q}[1..j{-}1]$, or the matched substring of $\boldsymbol{y} = \boldsymbol{y}[i{-}j{+}2..i]$. If they do, then the shift in $\boldsymbol{q}$ is equal to the maximum length of the prefix of $\boldsymbol{q}[1..j-1]$ that matches with a suffix of $\boldsymbol{y}[i{-}j{+}2..i]$. To compute this length, the algorithm first builds a new string $\boldsymbol{q'} = \boldsymbol{q}[1..j{-}1]\boldsymbol{y}[i{-}j{+}2..i]$ and, based on an insight given in [16], computes its *prefix* array $\pi_{\boldsymbol{q'}}$ rather than its border array. To compute the shift only the last j entries of $\pi_{\boldsymbol{q'}}$ are examined; that is, entries $k = j{+}1$ to $2(j-1)$. Note that we need to consider only those entries k in $\pi_{\boldsymbol{q'}}[j{+}1..2(j-1)]$, where a prefix of $\boldsymbol{q'}$ matches the suffix at

```
function KMP_INDET(y, n, q, m) : Integer List
i ← 0; j ← 0; indet_y ← false
indexlist ← ∅ ▷ List of indices where q occurs in y
q_ℓ ← longest regular prefix of q of length ℓ
β_q ← Compute_β(q_ℓ) ▷ Border Array of q_ℓ
while i < n do
    if q[j + 1] ≈ y[i + 1] then
        if INDET(y[i + 1]) then indet_y ← true
        j ← j + 1; i ← i + 1
        if j = m then
            indexlist ← indexlist ∪ {i−j+1}
            j ← Compute_Shift(indet_y, y, q, i, j, β_q, ℓ)
            indet_y ← false
    else
        if j = 0 then i ← i+1
        else
            j ← Compute_Shift(indet_y, y, q, i, j, β_q, ℓ)
            indet_y ← false
return indexlist
```

Figure 3. KMP_INDET checks whether the pattern $\boldsymbol{q}$ occurs in the text $\boldsymbol{y}$ (both possibly indeterminate). If it does, then it outputs the set of indices at which q occurs in $\boldsymbol{y}$; otherwise returns an empty set.

k ($\boldsymbol{q'}[k..2(j-1)]$); that is, the entries where $\pi_{\boldsymbol{q'}}[k] = 2j-k-1$. The shift is simply the maximum over such entries in $\pi_{\boldsymbol{q'}}$. (Recall that computing the border array for an indeterminate string is not useful as the matching relation $\approx$ is not transitive [8].)

```
function COMPUTE_SHIFT(indet_y, y, q, i, j, β_q, ℓ) : Integer
▷ ℓ is length of longest regular prefix of q.
if indet_y or j > ℓ then
    q' = q[1..j−1]y[i−j+2..i]
    π_q' ← Compute_π(q') ▷ Prefix Array of pattern q'
    max ← 0
    for k = j to 2(j − 1)
        if max < π_q'[k] and π_q'[k] = 2j−k−1 then
            max ← π_q'[k]
    j ← max
else ▷ prefix of q & substring of y are regular
    j ← β_q[j]
return j
```

Figure 4. COMPUTE_SHIFT computes the shift in the pattern when a mismatch occurs or the end of pattern is reached.

Figure 5 represents the processing of the text $\boldsymbol{x} = aabaabaa\{a,b\}baa\{a,c\}$ and pattern $\boldsymbol{p} = aabaa$ corresponding to the processing of $\boldsymbol{y}$ and $\boldsymbol{q}$ by KMP_INDET. KMP_INDET first computes $\beta_{\boldsymbol{p}} = (0,1,0,1,2)$ and $\ell = 5$. Initially the pattern is aligned with $\boldsymbol{x}$ at position 1. Since it matches with the text ($j = 5$), and $indet_x = false$ and $5 \leq (\ell = 5)$, we compute the shift from $\beta_{\boldsymbol{p}}[5] = 2$. Therefore, the pattern

is then aligned with $\boldsymbol{x}$ at position $i = 4$. Analogously, the pattern is next aligned with $\boldsymbol{x}$ at position $i = 7$. Since a mismatch occurs at $i + 1 = 10, j + 1 = 4$, and because $indet_x = true$, we construct $\boldsymbol{p'} = \boldsymbol{p}[1..2]\boldsymbol{x}[8..9] = aba\{a, b\}$ and compute $\pi_{\boldsymbol{p'}} = (4, 0, 2, 1)$. Then shift is equal to 2. Therefore the pattern is aligned with $\boldsymbol{x}$ at 8. Since it matches (and because it is the last match), KMP_INDET returns the list $\{1, 4, 8\}$.

KMP_INDET contains a function INDET that determines whether or not the current position $\boldsymbol{y}[i+1]$ is indeterminate. To enable this query to be answered efficiently, we suppose that an array $P = P[1..9]$ has been created with $P[t]$ equal to the t^{th} prime number in the range 2..23 (for $\sigma = 9$). Then $\boldsymbol{y}[i+1]$ is indeterminate if and only if it exceeds 23 or else does not occur in P. The worst case time requirement for INDET is therefore $\log_2 9$ times a few microseconds, the time for a binary search.

	1	2	3	4	5	6	7	8	9	10	11	12	13
$\boldsymbol{x}$	a	a	b	a	a	b	a	a	{a, b}	b	a	a	{a, c}
	a	a	b	a	a								
				a	a	b	a	a					
							a	a	b	x			
								a	a	b	a	a	

Figure 5. The figure simulates the execution of KMP_INDET on the text $\boldsymbol{x} = aabaabaa\{a, b\}baa\{a, c\}$ and pattern $\boldsymbol{p} = aabaa$. After execution, KMP_INDET returns the list of positions $\{1, 4, 8\}$ at which $\boldsymbol{p}$ occurs in $\boldsymbol{x}$. 'x' in the third alignment identifies a mismatch.

Now we discuss the running time of algorithm KMP_INDET. It is clear that the running time of KMP_INDET for a regular pattern and regular text is linear. Otherwise, when a matched prefix of $\boldsymbol{q}$ or a matched substring of $\boldsymbol{y}$ contains an indeterminate letter, then the algorithm constructs the prefix array of a new string $\boldsymbol{q'}$ which is a concatenation of the matched strings. In the worst case we might need to construct the prefix array of $\boldsymbol{q'}$ for each iteration of the **while** loop. By Lemma 10 and because $\boldsymbol{q'}$ can be of length at most $2(m-1)$, in the worst case the total time required for the execution of KMP_INDET is $\mathcal{O}(m^2 n)$. Theorem 11 states these conclusions:

Theorem 11 *Given text $\boldsymbol{y} = \boldsymbol{y}[1..n]$ and pattern $\boldsymbol{q} = \boldsymbol{q}[1..m]$ on an alphabet of constant size σ,* KMP_INDET *executes in $\mathcal{O}(n)$ time when $\boldsymbol{y}$ and $\boldsymbol{q}$ are both regular; otherwise, when both are indeterminate, the worst-case upper bound is $\mathcal{O}(m^2 n)$. The algorithm's additional space requirement is $\mathcal{O}(m)$, for the pattern $\boldsymbol{q'}$ and corresponding arrays $\beta_{\boldsymbol{q'}}$ and $\pi_{\boldsymbol{q'}}$.*

An improved theoretical bound to compute the prefix array for a string over a constant alphabet is given in [12], and is summarized in Lemma 12. Using Lemma 12 we restate Theorem 11 resulting in an improved run time complexity for KMP_INDET.

Lemma 12 ([12]) *The prefix array of an indeterminate string of length n over a constant-sized alphabet can be computed in $\mathcal{O}(n\sqrt{n})$ time and $\mathcal{O}(n)$ space.*

Theorem 13 *Given text $\boldsymbol{y} = \boldsymbol{y}[1..n]$ and pattern $\boldsymbol{q} = \boldsymbol{q}[1..m]$ on an alphabet of constant size σ,* KMP_INDET *executes in $\mathcal{O}(n)$ time when $\boldsymbol{y}$ and $\boldsymbol{q}$ are both regular; otherwise, when both are indeterminate, the worst-case upper bound is $\mathcal{O}(nm\sqrt{m})$.*

The algorithm's additional space requirement is $\mathcal{O}(m)$, for the pattern $\boldsymbol{q'}$ and corresponding arrays $\beta_{\boldsymbol{q'}}$ and $\pi_{\boldsymbol{q'}}$.

We provide context for the result given in Theorems 11 and 13 by the following:

Remark 14 *One of the features that makes this algorithm truly practical is that, apart from the $\mathcal{O}(n)$ time in-place mapping of $\boldsymbol{x}$ into $\boldsymbol{y}$ and $\boldsymbol{p}$ into $\boldsymbol{q}$, there is no preprocessing and no auxiliary data structure requirement. As a result, processing is direct and immediate, requiring negligible additional storage.*

Remark 15 *The worst case time requirement is predicated on a requirement for $\mathcal{O}(n)$ (short) shifts of $\boldsymbol{q}$ along $\boldsymbol{y}$, each requiring a worst-case $\mathcal{O}(m^2)$ prefix array calculation. For example, this circumstance could occur with $\boldsymbol{p} = \{a, b\}c^{m-1}$ and $\boldsymbol{x} = a^n$ or with $\boldsymbol{p} = ab$ and $\boldsymbol{x} = \{a, c\}^n$.*

Remark 16 *Indeed, given a regular pattern and a string $\boldsymbol{x}$ containing Q indeterminate letters (a case considered in both [4] and [5]),* KMP_Indet *may make as many as mQ shifts, each requiring $\mathcal{O}(m^2)$ processing, thus $\mathcal{O}(m^3Q)$ overall. Therefore, if m^3Q is small with respect to m^2n — Q small with respect to n/m — then* KMP_Indet *will execute in $\mathcal{O}(n)$ time.*

3 Conclusion

We have described a simple procedure, based on the KMP algorithm, to do pattern-matching on indeterminate strings that is very time-efficient in cases that arise in practice and moreover uses negligible $\Theta(m)$ space in all cases. We conjecture that a similar approach is feasible for the Boyer-Moore algorithm [3], together with its numerous variants (BM-Horspool, BM-Sunday, BM-Galil, Turbo-BM): see [15, Ch. 8] and

`https://www-igm.univ-mlv.fr/~lecroq/string/`

It would also be of interest to optimize KMP_Indet for the conservative indeterminate strings mentioned in Section 2. And we look forward to experimental comparison of the running times of existing indeterminate pattern-matching algorithms with those of KMP_Indet, assuming various frequencies of indeterminate letters.

References

1. K. Abrahamson: *Generalized string matching.* SIAM Journal of Computing, 16(6) 1987, pp. 1039–1051.
2. F. Blanchet-Sadri: *Algorithmic Combinatorics on Partial Words*, Chapman & Hall CRC, 2008.
3. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Communications of the ACM, 20(10) 1977, pp. 762–772.
4. M. Crochemore, C. S. Iliopoulis, R. Kundu, M. Mohamed, and F. Vayani: *Linear algorithm for conservative degenerate pattern matching.* Eng. Appls. of Artificial Intelligence, 51 2016, pp. 109–114.
5. J. W. Daykin, R. Groult, Y. Guesnet, T. Lecroq, A. Lefebvre, M. Léonard, L. Mouchard, E. Prieur-Gaston, and B. Watson: *Efficient pattern matching in degenerate strings with the Burrows-Wheeler transform.* Information Processing Letters, 147 2019, pp. 82–87.

6. M. Fischer and M. Paterson: *String matching and other products*, in Complexity of Computation,, R. Karp, ed., American Mathematical Society, 1974, pp. 113–125.
7. F. Gray: *Pulse code communication*. Hughes Aircraft Company, U.S. Patent no. 2632058, 1953.
8. J. Holub and W. F. Smyth: *Algorithms on indeterminate strings*. Proc. 14th Australasian Workshop on Combinatorial Algs. (AWOCA), 2003, pp. 36–45.
9. J. Holub, W. F. Smyth, and S. Wang: *Hybrid pattern-matching algorithms on indeterminate strings*, in London Algorithmics and Stringology, J. W. Daykin, M. Mohamed, and K. Steinhofel, eds., King's College Texts in Algorithmics, 2006, pp. 115–133.
10. L. Huang, V. Popic, and S. Batzoglou: *Short read alignment with populations of genomes*. Bioinformatics, 29(13) 06 2013, pp. i361–i370.
11. C. S. Iliopoulos, M. Mohamed, L. Mouchard, W. F. Smyth, K. G. Perdikuri, and A. K. Tsakalidis: *String regularities with don't cares*. Nordic J. Computing, 10(1) 2003, pp. 40–51.
12. C. S. Iliopoulos and J. Radoszewski: *Truly subquadratic-time extension queries and periodicity detection in strings with uncertainties*, in CPM, 2016.
13. D. E. Knuth, J. H. Morris, and V. R. Pratt: *Fast pattern matching in strings*. SIAM Journal of Computing, 6(2) 1977, pp. 323–350.
14. P. Procházka and J. Holub: *On-line searching in IUPAC nucleotide sequences*, in Proceedings of the 12th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2019) - Volume 3: BIOINFORMATICS, Prague, Czech Republic, February 22-24, 2019, E. D. Maria, A. L. N. Fred, and H. Gamboa, eds., SciTePress, 2019, pp. 66–77.
15. B. Smyth: *Computing Patterns in Strings*, Pearson/Addison–Wesley, 2003.
16. W. F. Smyth and S. Wang: *New perspectives on the prefix array*. Proc. 15th String Processing & Inform. Retrieval Symp. (SPIRE), 5280 2008, pp. 133–143.
17. W. F. Smyth and S. Wang: *An adaptive hybrid pattern-matching algorithm on indeterminate strings*. Internat. J. Foundations of Computer Science, 20(6) 2009, pp. 985–1004.

Re-Pair in Small Space

Dominik Köppl[1], Tomohiro I[2], Isamu Furuya[3], Yoshimasa Takabatake[2], Kensuke Sakai[2], and Keisuke Goto[4]

[1] Kyushu University, Japan Society for Promotion of Science
dominik.koeppl@inf.kyushu-u.ac.jp,
[2] Kyushu Institute of Technology, Japan
tomohiro@ai.kyutech.ac.jp, takabatake@ai.kyutech.ac.jp,
k_sakai@donald.ai.kyutech.ac.jp
[3] Graduate School of IST, Hokkaido University, Japan
furuya@ist.hokudai.ac.jp
[4] Fujitsu Laboratories Ltd., Kawasaki, Japan
goto.keisuke@fujitsu.com

Abstract. Re-Pair is a grammar compression scheme with favorably good compression rates. The computation of Re-Pair comes with the cost of maintaining large frequency tables, which makes it hard to compute Re-Pair on large scale data sets. As a solution for this problem we present, given a text of length n whose characters are drawn from an integer alphabet with size $\sigma = n^{\mathcal{O}(1)}$, an $\mathcal{O}(n^2) \cap \mathcal{O}(n^2 \lg\log_\tau n \lg\lg\lg n / \log_\tau n)$ time algorithm computing Re-Pair with $\max((n/c)\lg n, n\lceil\lg\tau\rceil) + \mathcal{O}(\lg n)$ bits of working space including the text space, where $c \geq 1$ is a fix user-defined constant and τ is the sum of σ and the number of non-terminals.

1 Introduction

Re-Pair [16] is a grammar deriving a single string. It is computed by replacing the most frequent bigram in this string with a new non-terminal, recursing until no bigram occurs more than once. Despite this simple-looking description, both the merits and the computational complexity of Re-Pair are intriguing. As a matter of fact, Re-Pair is currently one of the most well-understood grammar schemes.

Besides the seminal work of Larsson and Moffat [16], there are a couple of articles devoted to the compression aspects of Re-Pair: Given a text T of length n whose characters are drawn from an integer alphabet of size $\sigma := n^{\mathcal{O}(1)}$, the output of Re-Pair applied to T is at most $2nH_k(T) + o(n\lg\sigma)$ bits with $k = o(\log_\sigma n)$ when represented naively as a list of character pairs [19], where H_k denotes the empirical entropy of the k-th order. Using the encoding of Kieffer and Yang [12], Ochoa and Navarro [20] could improve the output size to at most $nH_k(T)+o(n\lg\sigma)$ bits. Other encodings were recently studied by Ganczorz [9]. Since Re-Pair is a so-called *irreducible* grammar, its grammar size, i.e., the sum of the symbols on the right hand side of all rules, is upper bounded by $\mathcal{O}(n/\log_\sigma n)$ [12, Lemma 2], which matches the information-theoretic lower bound on the size of a grammar for a string of length n. Comparing this size with the size of the smallest grammar, its approximation ratio has $\mathcal{O}((n/\lg n)^{2/3})$ as an upper bound [5] and $\Omega(\lg n/\lg\lg n)$ as a lower bound [1]. On the practical side, Yoshida and Kida [26] presented an efficient fixed-length code for compressing the Re-Pair grammar.

Although conceived as a grammar for compressing texts, Re-Pair has been successfully applied for compressing trees [17], matrices [23], or images [7]. For different settings or for better compression rates, there is a great interest in modifications to

Dominik Köppl, Tomohiro I, Isamu Furuya, Yoshimasa Takabatake, Kensuke Sakai, Keisuke Goto: Re-Pair in Small Space, pp. 134–147.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9

Re-Pair. Charikar et al. [5, Sect. G] gave an easy variation to improve the size of the grammar. Another variant, proposed by Claude and Navarro [6], runs in a user defined working space ($> n \lg n$ bits), and shares with our proposed solution the idea of a table that (a) is stored with the text in the working space, and (b) grows in rounds. The variant of González et al. [11] is specialized on compressing an array of integers delta-encoded (i.e., by the differences of subsequent entries). Sekine et al. [22] provide an adaptive variant whose algorithm divides the input into blocks, and processes each block based on the rules obtained from the grammars of its preceding blocks. Subsequently, Masaki and Kida [18] gave an *online* algorithm producing a grammar mimicking Re-Pair. Ganczorz and Jez [10] modified the Re-Pair grammar by disfavoring the replacement of bigrams that cross Lempel-Ziv-77 (LZ77) [27] factorization borders, which allowed the authors to achieve practically smaller grammar sizes. Recently, Furuya et al. [8] presented a variant, called *MR-Re-Pair*, in which a most frequent maximal repeat is replaced instead of a most frequent bigram.

1.1 Related Work

Re-Pair is a grammar proposed by Larsson and Moffat [16], who presented an algorithm computing it in expected linear time with $5n+4\sigma^2+4\sigma'+\sqrt{n}$ words of working space, where σ' is the number of non-terminals (produced by Re-Pair). González et al. [11, Sect. 4.1] gave another linear time algorithm using $12n + \mathcal{O}(p)$ bytes of working space, where p is the maximum number of distinct bigrams considered at any time. The large space requirements got significantly improved by Bille et al. [3], who presented a randomized linear time algorithm taking $(1 + \epsilon)n + \sqrt{n}$ words on top of the rewriteable text space for a constant ϵ with $0 < \epsilon \leq 1$. Subsequently, they improved their algorithm in [2] to include the text space within the $(1 + \epsilon)n + \sqrt{n}$ words of working space. However, they assume that the alphabet size σ is constant and $\lceil \lg \sigma \rceil \leq w/2$, where w is the machine word size. They also provide a solution for $\epsilon = 0$ running in expected linear time. Recently, Sakai et al. [21] showed how to convert an arbitrary grammar (representing a text) into the Re-Pair grammar in compressed space, i.e., without decompressing the text. Combined with a grammar compression that can process the text in compressed space in a streaming fashion, this result leads to the first Re-Pair computation in compressed space.

1.2 Our Contribution

In this article,[1] we propose an algorithm that computes the Re-Pair grammar in $\mathcal{O}(n^2) \cap \mathcal{O}(n^2 \lg \log_\tau n \lg \lg \lg n / \log_\tau n)$ time (cf. Theorem 3 and Theorem 5) with $\max((n/c) \lg n,\ n \lceil \lg \tau \rceil) + \mathcal{O}(\lg n)$ bits of working space including the text space, where $c \geq 1$ is a fix user-defined constant and τ is the sum of the alphabet size σ and the number of non-terminals σ'.

Given that the characters of the text are drawn from a large integer alphabet with size $\sigma = \Omega(n)$,[2] the algorithm works in-place. In this setting, we obtain the first non-trivial in-place algorithm, as a trivial approach on a text T of length n would

[1] Parts of this work have already been presented as a poster [15] at the Data Compression Conference 2020 (`https://sigport.org/documents/re-pair-small-space`).

[2] We consider the alphabet as not effective, i.e., a character does not have to appear in the text, as this is a common setting in Unicode texts such as Japanese text. For instance, $n^2 = \Omega(n) \cap n^{\mathcal{O}(1)} \neq \emptyset$ could be such an alphabet size.

compute the most frequent bigram in $\Theta(n^2)$ time by computing the frequency of each bigram $T[i]T[i+1]$ for every integer i with $1 \leq i \leq n-1$, keeping only the most frequent bigram in memory. This sums up to $\mathcal{O}(n^3)$ total time, since there can be $\Theta(n)$ different bigrams considered for replacement by Re-Pair.

To achieve our goal of $\mathcal{O}(n^2)$ total time, we first provide a trade-off algorithm (cf. Lemma 2) finding the d most frequent bigrams in $\mathcal{O}(n^2 \lg d/d)$ time for a trade-off parameter d. We subsequently run this algorithm for increasing values of d, and show that we need to run it $\mathcal{O}(\lg n)$ times, which gives us $\mathcal{O}(n^2)$ time if d is increasing sufficiently fast. Our major tools are appropriate text partitioning, elementary scans, and sorting steps, which we visualize in Section 2.5 by an example, and practically evaluate in Section 2.6. When $\tau = o(n)$, a different approach using word-packing and bit-parallel techniques becomes attractive, leading to an $\mathcal{O}(n \lg \log_\tau n \lg \lg \lg n / \log_\tau n)$ time algorithm, which we explain in Section 3.

1.3 Preliminaries

We use the word RAM model with a word size of $\Omega(\lg n)$ for an integer $n \geq 1$. We work in the restore model [4], in which algorithms are allowed to overwrite the input, as long as they can restore the input to its original form.

Strings. Let T be a text of length n whose characters are drawn from an integer alphabet Σ of size $\sigma = n^{\mathcal{O}(1)}$. A bigram is an element of Σ^2. The *frequency* of a bigram B in T is the number of *non-overlapping* occurrences of B in T, which is at most $|T|/2$. For instance, the frequency of the bigram $\mathtt{aa} \in \Sigma^2$ in the text $T = \mathtt{a} \cdots \mathtt{a}$ consisting of n $\mathtt{a}$'s is $\lfloor n/2 \rfloor$.

Re-Pair. We reformulate the recursive description in the introduction by dividing a Re-Pair construction algorithm into *turns*. Stipulating that T_i is the text after the i-th turn with $i \geq 1$ and $T_0 := T \in \Sigma_0^+$ with $\Sigma_0 := \Sigma$, Re-Pair replaces one of the most frequent bigrams (ties are broken arbitrarily) in T_{i-1} with a non-terminal in the i-th turn. Given this bigram is $\mathtt{bc} \in \Sigma_{i-1}^2$, Re-Pair replaces all occurrences of $\mathtt{bc}$ with a new non-terminal X_i in T_{i-1}, and sets $\Sigma_i := \Sigma_{i-1} \cup \{X_i\}$ with $\sigma_i := |\Sigma_i|$ to produce $T_i \in \Sigma_i^+$. Since $|T_i| \leq |T_{i-1}| - 2$, Re-Pair terminates after $m < n/2$ turns such that $T_m \in \Sigma_m^+$ contains no bigram occurring more than once.

2 Algorithm

A major task for producing the Re-Pair grammar is to count the frequencies of the most frequent bigrams. Our work horse for this task is a frequency table. A *frequency table* in T_i of length f stores pairs of the form $(\mathtt{bc}, x)$, where $\mathtt{bc}$ is a bigram and x the frequency of $\mathtt{bc}$ in T_i. It uses $f \lceil \lg(\sigma_i^2 n_i/2) \rceil$ bits of space since an entry stores a bigram consisting of two characters from Σ_i and its respective frequency, which can be at most $n_i/2$. Throughout this paper, we use an elementary in-place sorting algorithm like heapsort:

Lemma 1 ([25]). *An array of length n can be sorted in-place in $\mathcal{O}(n \lg n)$ time.*

2.1 Trade-Off Computation

By embracing the frequency tables, we present a solution with a trade-off parameter:

Lemma 2. *Given an integer d with $d \geq 1$, we can compute the frequencies of the d most frequent bigrams in a text of length n whose characters are drawn from an alphabet of size σ in $\mathcal{O}(\max(n,d)n \lg d/d)$ time using $2d\lceil \lg(\sigma^2 n/2)\rceil + \mathcal{O}(\lg n)$ bits.*

Proof. Our idea is to partition the set of all bigrams appearing in T into $\lceil n/d\rceil$ subsets, compute the frequencies for each subset, and finally merge these frequencies. In detail, we partition the text $T = S_1 \cdots S_{\lceil n/d\rceil}$ into $\lceil n/d\rceil$ substrings such that each substring has length d (the last one has a length of at most d). Subsequently, we extend S_j to the left (only if $j > 1$) such that S_j and S_{j+1} overlap by one text position, for $1 \leq j < \lceil n/d\rceil$. By doing so, we take the bigram on the border of two adjacent substrings S_j and S_{j+1} for each $j < \lceil n/d\rceil$ into account. Next, we create two frequency tables F and F', each of length d for storing the frequencies of d bigrams. These tables are at the beginning empty. In what follows, we fill F such that after processing S_i, F stores the most frequent d bigrams among those bigrams occurring in $S_1, \ldots, S_i$ while F' acts as a temporary space for storing candidate bigrams that can enter F.

With F and F', we process each of the n/d substrings S_j as follows: Let us fix an integer j with $1 \leq j \leq \lceil n/d\rceil$. We first put all bigrams of S_j into F' in *lexicographic* order. We can perform this within the space of F' in $\mathcal{O}(d \lg d)$ time since there are at most d different bigrams in S_j. We compute the frequencies of all these bigrams in the *complete* text T in $\mathcal{O}(n \lg d)$ time by scanning the text from left to right while locating a bigram in F' in $\mathcal{O}(\lg d)$ time with a binary search. Subsequently, we interpret F and F' as one large frequency table, sort it with respect to the frequencies while discarding duplicates such that F stores the d most frequent bigrams in $T[1..jd]$. This sorting step can be done in $\mathcal{O}(d \lg d)$ time. Finally, we clear F' and are done with S_j. After the final merge step, we obtain the d most frequent bigrams of T stored in F.

Since each of the $\mathcal{O}(n/d)$ merge steps takes $\mathcal{O}(d \lg d + n \lg d)$ time, we need $\mathcal{O}(\max(d,n) \cdot (n \lg d)/d)$ time. For $d \geq n$, we can build a large frequency table and perform one scan to count the frequencies of all bigrams in T. This scan and the final sorting with respect to the counted frequencies can be done in $\mathcal{O}(n \lg n)$ time.

2.2 Algorithmic Ideas

With Lemma 2, we can compute T_m in $\mathcal{O}(mn^2 \lg d/d)$ time with additional $2d\lceil \lg(\sigma_m^2 n/2)\rceil$ bits[3] of working space on top of the text for a parameter d with $1 \leq d \leq n$. In the following, we show how this leads us to our first algorithm computing Re-Pair:

Theorem 3. *We can compute Re-Pair on a string of length n in $\mathcal{O}(n^2)$ time with $\max((n/c)\lg n, n\lceil \lg \tau\rceil) + \mathcal{O}(\lg n)$ bits of working space including the text space as a rewriteable part in the working space, where $c \geq 1$ is a fixed constant and $\tau = \sigma_m$ is the sum of the alphabet size σ and the number of non-terminal symbols.*

In our model, we assume that we can enlarge the text T_i from $n_i\lceil \lg \sigma_i\rceil$ bits to $n_i\lceil \lg \sigma_{i+1}\rceil$ bits without additional extra memory. Our main idea is to store a growing frequency table using the space freed up by replacing bigrams with non-terminals. In detail, we maintain a frequency table F in T_i of length f_k for a growing variable f_k, which is set to $f_0 := \mathcal{O}(1)$ in the beginning. The table F takes $f_k\lceil \lg(\sigma_i^2 n/2)\rceil$ bits,

[3] The variable τ used in the abstract and in the introduction is interchangeable with σ_m, i.e., $\tau = \sigma_m$.

which is $\mathcal{O}(\lg(\sigma^2 n)) = \mathcal{O}(\lg n)$ bits for $k = 0$. When we want to query it for a most frequent bigram, we linearly scan F in $\mathcal{O}(f_k) = \mathcal{O}(n)$ time, which is not a problem since (a) the number of queries is $m \leq n$, and (b) we aim for $\mathcal{O}(n^2)$ overall running time. A consequence is that there is no need to sort the bigrams in F according to their frequencies, which simplifies the following discussion.

Frequency Table F. With Lemma 2, we can compute F in $\mathcal{O}(n \max(n, f_k) \lg f_k/f_k)$ time. Instead of recomputing F on every turn i, we want to recompute it only when it no longer stores a most frequent bigram. However, it is not obvious when this happens as replacing a most frequent bigram during a turn (a) removes this entry in F and (b) can reduce the frequencies of other bigrams in F, making them possibly less frequent than other bigrams not tracked by F. Hence, the variable i for the i-th turn (creating the i-th non-terminal) and the variable k for recomputing the frequency table F the $(k+1)$-st time are loosely connected. We group together all turns with the same f_k and call this group the *k-th round* of the algorithm. At the beginning of each round, we enlarge f_k and create a new F with a capacity for f_k bigrams. Since a recomputation of F takes much time, we want to end a round only if F is no longer useful, i.e., when we no longer can guarantee that F stores a most frequent bigram. To achieve our claimed time bounds, we want to assign all m turns to $\mathcal{O}(\lg n)$ different rounds, which can only be done if f_k grows sufficiently fast.

Algorithm Outline. At the beginning of the k-th round and the i-th turn, we compute the frequency table F storing f_k bigrams, and keep additionally the lowest frequency of F as a threshold t_k, which is treated as a constant during this round. During the computation of the i-th turn, we replace the most frequent bigram (say, $\texttt{bc} \in \Sigma_i^2$) in the text T_i with a non-terminal X_{i+1} to produce T_{i+1}. Thereafter, we remove $\texttt{bc}$ from F and update those frequencies in F which got decreased by the replacement of $\texttt{bc}$ with X_{i+1}, and add each bigram containing the new character X_{i+1} into F if its frequency is at least t_k. Whenever a frequency in F drops below t_k, we discard it. If F becomes empty, we move to the $(k+1)$-st round, and create a new F for storing f_{k+1} frequencies. Otherwise (F still stores an entry), we can be sure that F stores a most frequent bigram. In both cases, we recurse with the $(i+1)$-st turn by selecting the bigram with the highest frequency stored in F. We show in Algorithm 1 a pseudo code of this outlined algorithm. We describe in the following how we update F and how large f_{k+1} can become at least.

2.3 Algorithmic Details

Suppose that we are in the k-th round and in the i-th turn. Let t_k be the lowest frequency in F computed at the beginning of the k-th round. We keep t_k as a constant threshold for the invariant that all frequencies in F are at least t_k during the k-th round. With this threshold we can assure in the following that F is either empty or stores a most frequent bigram. Now suppose that the most frequent bigram of T_i is $\texttt{bc} \in \Sigma_i^2$, which is stored in F. To produce T_{i+1} (and hence advancing to the $(i+1)$-st turn), we enlarge the space of T_i from $n_i \lceil \lg \sigma_i \rceil$ to $n_i \lceil \lg \sigma_{i+1} \rceil$, and replace all occurrences of $\texttt{bc}$ in T_i with a new non-terminal X_{i+1}. Subsequently, we would like to take the next bigram of F. For that, however, we need to update the stored frequencies in F. To see this necessity, suppose that there is an occurrence of $\texttt{abcd}$ with two characters $\texttt{a}, \texttt{d} \in \Sigma_i$ in T_i. By replacing $\texttt{bc}$ with X_{i+1},

Algorithm 1: Algorithmic outline of our proposed algorithm working on a text T with a growing frequency table F. The constants α_i and β_i are explained in Section 2.3. The same section shows that the outer while loop is executed $\mathcal{O}(\lg n)$ times.

```
1  k ← 0, i ← 0
2  f_0 ← O(1)
3  T_0 ← T
4  while highest frequency of a bigram in T is greater than one do      ▷ during the k-th round
5      F ← frequency table of Lemma 2 with d := f_k
6      t_k ← minimum frequency stored in F
7      while F ≠ ∅ do                                                  ▷ during the i-th turn
8          bc ← most frequent bigram stored in F
9          T_{i+1} ← T_i.replace(bc, X_{i+1})                          ▷ create rule X_{i+1} → bc
10         i ← i + 1                                                   ▷ introduce the (i+1)-th turn
11         remove all bigrams with frequency lower than t_k from F
12         add new bigrams to F having X_i as left or right character and a frequency of at
             least t_k
13     f_{k+1} ← f_k + max(2/β_i, (f_k − 1)/(2β_i))/α_i
14     k ← k + 1                                                       ▷ introduce the (k+1)-th round
15 Invariant: i = m (the number of non-terminals)
```

1. the frequencies of `ab` and `cd` decrease by one[4], and
2. the frequencies of `a`X_{i+1} and X_{i+1}`d` increase by one.

Updating F. We can take care of the former changes (1) by decreasing the respective bigram in F (in case that it is present). If the frequency of this bigram drops below the threshold t_k, we remove it from F as there may be bigrams with a higher frequency that are not present in F. To cope with the latter changes (2), we track the characters adjacent to X_{i+1} after the replacement, count their numbers, and add their respective bigrams to F if their frequencies are sufficiently high. In detail, suppose that we have substituted `bc` with X_{i+1} exactly h times. Consequently, with the new text T_{i+1} we have additionally $h \lg \sigma_{i+1}$ bits of free space[5], which we call D in the following. Subsequently, we scan the text and put the characters of Σ_{i+1} appearing to the left of each of the h occurrences of X_{i+1} into D. After sorting the characters in D lexicographically, we can count the frequency of `a`X_{i+1} for each character `a` $\in \Sigma_{i+1}$ preceding an occurrence of X_{i+1} in the text T_{i+1} by scanning D linearly. If the obtained frequency of such a bigram `a`X_{i+1} is at least as high as the threshold t_k, we insert `a`X_{i+1} into F, and subsequently discard a bigram with the currently lowest frequency in F if the size of F has become f_k+1. In case that we visit a run of X_{i+1}'s during the creation of D, we must take care of not counting the overlapping occurrences of $X_{i+1}X_{i+1}$. Finally, we can count analogously the occurrences of X_{i+1}`d` for all characters `d` $\in \Sigma_i$ succeeding an occurrence of X_{i+1}.

Capacity of F. After the above procedure we have updated the frequencies of F. When F becomes empty, all bigrams stored in F have been replaced or have a frequency

[4] For the border case `a` = `b` = `c` (resp. `b` = `c` = `d`), there is no need to decrement the frequency of `ab` (resp. `cd`).

[5] The free space is consecutive after shifting all characters to the left.

that became less than t_k. Subsequently, we end the k-th round and continue with the $(k+1)$-st round by (a) creating a new frequency table F with capacity f_{k+1}, and (b) setting the new threshold t_{k+1} to the minimal frequency in F. In what follows, we (a) analyze in detail when F becomes empty (as this determines the sizes f_k and f_{k+1}), and (b) show that we can compensate the number of discarded bigrams with an enlargement of F's capacity from f_k bigrams to f_{k+1} bigrams for the sake of our aimed total running time.

Next, we analyze how many characters we have to free up (i.e., how many bigram occurrences we have to replace) to gain enough space for storing an additional frequency. Let $\delta_i := \lg(\sigma_{i+1}^2 n_i/2)$ be the number of bits needed to store one entry in F, and let $\beta_i := \min(\delta_i/\lg\sigma_{i+1}, c\delta_i/\lg n)$ be the minimum number of characters that need to be freed to store one frequency in this space. To understand the value of β_i, we look at the arguments of the minimum function in the definition of β_i and simultaneously at the maximum function in our aimed working space of $\max(n\lceil\lg\sigma_m\rceil, (n/c)\lg n) + \mathcal{O}(\lg n)$ bits (cf. Theorem 3):

1. The first item in this maximum function allows us to spend $\lg\sigma_{i+1}$ bits for each freed character such that we obtain space for one additional entry in F after freeing $\delta_i/\lg\sigma_{i+1}$ characters.
2. The second item allows us to use $\lg n$ additional bits after freeing up c characters.[6] Hence, after freeing up $c\delta_i/\lg n$ characters, we have space to store one additional entry in F.

With $\beta_i = \min(\delta_i/\lg\sigma_{i+1}, c\delta_i/\lg n) = \mathcal{O}(\log_\sigma n) \cap \mathcal{O}(\log_n \sigma) = \mathcal{O}(1)$ we have the sufficient condition that replacing a constant number of characters gives us enough space for storing an additional frequency.

If we assume that replacing the occurrences of a bigram stored in F does not decrease the other frequencies stored in F, the analysis is now simple: Since each bigram in F has a frequency of at least two, $f_{k+1} \geq f_k + f_k/\beta_i$. Since $\beta_i = \mathcal{O}(1)$, this lets f_k grow exponentially, meaning that we need $\mathcal{O}(\lg n)$ rounds. In what follows, we show that this is also true in the general case.

Lemma 4. *Given the frequency of all bigrams in F drop below the threshold t_k after replacing the most frequent bigram* `bc`*, then its frequency has to be at least* $\max(2, |F| - 1/2)$*, where $|F| \leq f_k$ is the number of frequencies stored in F.*

Proof. If the frequency of `bc` in T_i is x, then we can reduce at most $2x$ frequencies of other bigrams (both the left character and the right character of each occurrence of `bc` can contribute to an occurrence of another bigram). Since a bigram must occur at least twice in T_i to be present in F, the frequency of `bc` has to be at least $\max(2, (f_k - 1)/2)$ for discarding all bigrams of F.

Suppose that we have enough space available for storing the frequencies of $\alpha_i f_k$ bigrams, where α_i is a constant (depending on σ_i and n_i) such that F and the working space of Lemma 2 with $d = f_k$ can be stored within this space. With β_i and Lemma 4

[6] This additional treatment helps us to let f_k grow sufficiently fast in the first steps to save our $\mathcal{O}(n^2)$ time bound, as for sufficiently small alphabets and large text sizes, $\lg(\sigma^2 n/2)/\lg\sigma = \mathcal{O}(\lg n)$, which means that we might run the first $\mathcal{O}(\lg n)$ turns with $f_k = \mathcal{O}(1)$, and therefore already spend $\mathcal{O}(n^2 \lg n)$ time.

with $|F| = f_k$, we have

$$\begin{aligned}\alpha_i f_{k+1} &= \alpha_i f_k + \max(2/\beta_i, (f_k - 1)/(2\beta_i)) \\ &= \alpha_i f_k \max(1 + 2/(\alpha_i\beta_i f_k), 1 + 1/(2\alpha_i\beta_i) - 1/(2\alpha_i\beta_i f_k)) \\ &\geq \alpha_i f_k(1 + 2/(5\alpha_i\beta_i)) =: \gamma_i\alpha_i f_k \text{ with } \gamma_i := 1 + 2/(5\alpha_i\beta_i),\end{aligned}$$

where we used the equivalence $1+2/(\alpha_i\beta_i f_k) = 1+1/(2\alpha_i\beta_i)-1/(2\alpha_i\beta_i f_k) \Leftrightarrow 5 = f_k$ to estimate the two arguments of the maximum function.

Since we let f_k grow by a factor of at least $\gamma := \min_{1\leq i\leq m}\gamma_i > 1$ for each recomputation of F, $f_k = \Omega(\gamma^k)$, and therefore $f_k = \Theta(n)$ after $k = \mathcal{O}(\lg n)$ steps. Consequently, after reaching $k = \mathcal{O}(\lg n)$, we can iterate the above procedure a constant number of times to compute the non-terminals of the remaining bigrams occurring at least twice.

Time Analysis. In total we have $\mathcal{O}(\lg n)$ rounds. At the start of the k-th round, we compute F with the algorithm of Lemma 2 with $d = f_k$ on a text of length at most $n - f_k$ in $\mathcal{O}(n(n - f_k)\cdot \lg f_k/f_k)$ time with $f_k \leq n$. Summing this up, we yield

$$\mathcal{O}\left(\sum_{k=0}^{\mathcal{O}(\lg n)} \frac{n - f_k}{f_k} n \lg f_k\right) = \mathcal{O}\left(n^2 \sum_{k}^{\lg n} \frac{k}{\gamma^k}\right) = \mathcal{O}\big(n^2\big) \text{ time.} \tag{1}$$

In the i-th turn, we update F by decreasing the frequencies of the bigrams affected by the substitution of the most frequent bigram `bc` with X_{i+1}. For decreasing such a frequency, we look up its respective bigram with a linear scan in F, which takes $f_k = \mathcal{O}(n)$ time. However, since this decrease is accompanied with a replacement of an occurrence of `bc`, we obtain $\mathcal{O}(n^2)$ total time by charging each text position with $\mathcal{O}(n)$ time for a linear search in F. With the same argument, we can bound the total time for sorting the characters in D to $\mathcal{O}(n^2)$ overall time: Since we spend $\mathcal{O}(h \lg h)$ time on sorting h characters preceding or succeeding a replaced character, and $\mathcal{O}(f_k) = \mathcal{O}(n)$ time on swapping a sufficiently large new bigram composed of X_{i+1} and a character of Σ_{i+1} with a bigram with the lowest frequency in F, we charge each text position again with $\mathcal{O}(n)$ time. Putting all time bounds together gives the claim of Theorem 3.

2.4 Storing the Output In-Place

Finally, we show that we can store the computed grammar in text space. More precisely, we want to store the grammar in an auxiliary array A packed at the end of the working space such that the entry $A[i]$ stores the right hand side of the non-terminal X_i, which is a bigram. Thus the non-terminals are represented implicitly as indices of the array A. We therefore need to subtract $2\lg\sigma_i$ bits of space from our working space $\alpha_i f_k$ after the i-th turn. By adjusting α_i in the above equations, we can deal with this additional space requirement as long as the frequencies of the replaced bigrams are at least three (we charge two occurrences for growing the space of A).

When only bigrams with frequencies of at most two remain, we switch to a simpler algorithm, discarding the idea of maintaining the frequency table F: Suppose that we work with the text T_i. Let λ be a text position, which is 1 in the beginning, but will be incremented in the following turns while holding the invariant that $T[1..\lambda]$ does not contain a bigram of frequency two. We scan $T_i[\lambda..n]$ linearly from left to right

and check, for each text position j, whether the bigram $T_i[j]T_i[j+1]$ has another occurrence $T_i[j']T_i[j'+1] = T_i[j]T_i[j+1]$ with $j' > j+1$, and if so,

(a) append $T_i[j]T_i[j+1]$ to A,
(b) replace $T_i[j]T_i[j+1]$ and $T_i[j']T_i[j'+1]$ with a new non-terminal X_{i+1} to transform T_i to T_{i+1}, and
(c) recurse on T_{i+1} with $\lambda := j$ until no bigram with frequency two is left.

The position λ, which we never decrement, helps us to skip over all text positions starting with bigrams with a frequency of one. Thus, the algorithm spends $\mathcal{O}(n)$ time for each such text position, and $\mathcal{O}(n)$ time for each bigram with frequency two. Since there are at most n such bigrams, the overall running time of this algorithm is $\mathcal{O}(n^2)$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	c	a	b	a	a	c	a	b	c	a	b	a	a	c	a	a	a	b	c	a	b	ab:5	ca:5	aa:3
2	c	X_1		a	a	c	X_1		c	X_1		a	a	c	a	a	X_1		c	X_1		ab:0	ca:1	aa:3
																	D							
3	c	X_1	a	a	c	X_1	c	X_1	a	a	c	a	a	X_1	c	X_1								aa:3
4	c	X_1	a	a	c	X_1	c	X_1	a	a	c	a	a	X_1	c	X_1	c	c	c	a	c			aa:3
5	c	X_1	a	a	c	X_1	c	X_1	a	a	c	a	a	X_1	c	X_1	a	c	c	c	c			aa:3
6	c	X_1	a	a	c	X_1	c	X_1	a	a	c	a	a	X_1	c	X_1							cX_1:4	aa:3
7	c	X_1	a	a	c	X_1	c	X_1	a	a	c	a	a	X_1	c	X_1	a	c	a	c			cX_1:4	aa:3
8	c	X_1	a	a	c	X_1	c	X_1	a	a	c	a	a	X_1	c	X_1	a	a	c	c			cX_1:4	aa:3
9	c	X_1	a	a	c	X_1	c	X_1	a	a	c	a	a	X_1	c	X_1							cX_1:4	aa:3
																						F		

Figure 1. Step-by-step execution of the first turn of our algorithm on the string $T =$ `cabaacabcabaacaaabcab`. The turn starts with the memory configuration given in Row 1. Positions 1 to 21 are text positions, positions 22 to 24 belong to F ($f_0 = 3$, and it is assumed that a frequency fits into a text entry). Subsequent rows depict the memory configuration during Turn 1. A comment to each row is given in Section 2.5.

2.5 Step-by-Step Execution

Here, we present an exemplary execution of the first turn (of the first round) on the input $T =$ `cabaacabcabaacaaabcab`. We visualize each step of this turn as a row in Fig. 1. A detailed description of each row follows:

Row 1: Suppose that we have computed F, which has the constant number of entries $f_0 = 3$.[7] The highest frequency is five achieved by `ab` and `ca`. The lowest frequency represented in F is three, which becomes the threshold t_0 for a bigram to be present in F such that bigrams whose frequencies drop below t_0 are removed from F. This threshold is a constant for all later turns until F is rebuilt (in the following round). During Turn 1, the algorithm proceeds now as follows:

Row 2: Choose `ab` as a bigram to replace with a new non-terminal X_1 (break ties arbitrarily). Replace every occurrence of `ab` with X_1 while decrementing frequencies in F accordingly to the neighboring characters of the replaced occurrence.

[7] In the later turns when the size f_k becomes larger, F will be put in the text space.

	Prefix Size in KiB				
Data Set	64	128	256	512	1024
ESCHERICHIA_COLI	20.68	130.47	516.67	1708.02	10112.47
CERE	13.69	90.83	443.17	2125.17	9185.58
COREUTILS	12.88	75.64	325.51	1502.89	5144.18
EINSTEIN.DE.TXT	19.55	88.34	181.84	805.81	4559.79
EINSTEIN.EN.TXT	21.11	78.57	160.41	900.79	4353.81
INFLUENZA	41.01	160.68	667.58	2630.65	10526.23
KERNEL	20.53	101.84	208.08	1575.48	5067.80
PARA	20.90	175.93	370.72	2826.76	9462.74
WORLD_LEADERS	11.92	21.82	167.52	661.52	1718.36
`aa`$\cdots$`a`	0.35	0.92	3.90	14.16	61.74

Table 1. Experimental evaluation of our implementation described in Section 2.6. Table entries are running times in seconds. The last line is the benchmark on the unary string `aa`$\cdots$`a`.

Row 3: Remove from F every bigram whose frequency falls below the threshold. Obtain space for D by aligning the compressed text T_1. (The process of Row 2 and Row 3 can be done simultaneously.)

Row 4: Scan the text and copy each character preceding an occurrence of X_1 in T_1 to D.

Row 5: Sort characters in D lexicographically.

Row 6: Insert new bigrams (consisting of a character of D and X_1) whose frequencies are at least as large as the threshold.

Row 7: Scan the text again and copy each character succeeding an occurrence of X_1 in T_1 to D (symmetric to Row 4).

Row 8: Sort all characters in D lexicographically (symmetric to Row 5).

Row 9: Insert new bigrams whose frequencies are at least as large as the threshold (symmetric to Row 6).

2.6 Implementation

At `https://github.com/koeppl/repair-inplace`, we provide a simplified implementation in C++17. The simplification is that we (a) fix the bit width of the text space to 16 bit, and (b) assume that Σ is the byte alphabet. We further skip the step increasing the bit width of the text from $\lg \sigma_i$ to $\lg \sigma_{i+1}$. This means that the program works as long as the characters of Σ_m fit into 16 bits. The benchmark, whose results are displayed in Table 1, was conducted on a Mac Pro Server with an Intel Xeon CPU X5670 clocked at 2.93GHz running Arch Linux. The implementation was compiled with `gcc-8.2.1` in the highest optimization mode `-O3`. Looking at Table 1, we observe that the running time is super-linear to the input size on all text instances, which we obtained from the Pizza&Chili corpus (`http://pizzachili.dcc.uchile.cl/`). Table 2 gives some characteristics about the used data sets. We see that the number of rounds is the number of turns plus one for every unary string $\mathtt{a}^{2^k}$ with an integer $k \geq 1$ since the text contains only one bigram with a frequency larger than two in each round. Replacing this bigram in the text makes F empty such that the algorithm recomputes F after each turn. Note that the number of rounds can drop while scaling the prefix length based on the choice of the bigrams stored in F.

		Turns /1000 Prefix Size in KiB					Rounds Prefix Size in KiB				
Data Set	σ	2^6	2^7	2^8	2^9	2^{10}	2^6	2^7	2^8	2^9	2^{10}
ESCHERICHIA_COLI	4	1.8	3.2	5.6	10.3	18.1	6	9	9	12	12
CERE	5	1.4	2.8	5.0	9.2	15.1	13	14	14	14	14
COREUTILS	113	4.7	6.7	10.2	16.1	26.5	15	15	15	14	14
EINSTEIN.DE.TXT	95	1.7	2.8	3.7	5.2	9.7	14	14	15	16	16
EINSTEIN.EN.TXT	87	3.3	3.5	3.8	4.5	8.6	16	15	15	15	17
INFLUENZA	7	2.5	3.7	9.5	13.4	22.1	11	12	14	13	15
KERNEL	160	4.5	8.0	13.9	24.5	43.7	10	11	14	14	13
PARA	5	1.8	3.2	5.8	10.1	17.6	12	12	13	13	14
WORLD_LEADERS	87	2.6	4.3	6.1	10.0	42.1	11	11	11	11	14
$\mathtt{a}\mathtt{a}\cdots\mathtt{a}$	1	15	16	17	18	19	16	17	18	19	20

Table 2. Characteristics of our data sets used in Section 2.6. The number of turns and rounds are given for each of the prefix sizes 128, 256, 512, and 1024 KiB of the respective data sets. The number of turns reflecting the number of non-terminals is given in units of thousands. The turns of the unary string $\mathtt{a}\mathtt{a}\cdots\mathtt{a}$ are in plain units (not divided by thousand).

3 Bit-Parallel Algorithm

In the case that $\tau = \sigma_m$ is $o(n)$ (and therefore $\sigma = o(n)$), a word-packing approach becomes interesting. We present techniques speeding up previously introduced operations on chunks of $\mathcal{O}(\log_\tau n)$ characters from $\mathcal{O}(\log_\tau n)$ time to $\mathcal{O}(\lg\lg\lg n)$ time. In the end, these techniques allow us to speed up the sequential algorithm of Theorem 3 from $\mathcal{O}(n^2)$ time to the following:

Theorem 5. *We can compute Re-Pair on a string of length n in $\mathcal{O}(n^2 \lg\log_\tau n \lg\lg\lg n/\log_\tau n)$ time with $\max((n/c)\lg n, n\lceil\lg\tau\rceil) + \mathcal{O}(\lg n)$ bits of working space including the text space, where $c \geq 1$ is a fixed constant and $\tau = \sigma_m$ is the sum of the alphabet size σ and the number of non-terminal symbols.*

Note that the $\mathcal{O}(\lg\lg\lg n)$ time factor is due to the popcount function [24, Algo. 1], which has been optimized to a single instruction on modern computer architectures.

3.1 Broadword Search

First, we deal with accelerating the computation of the frequency of a bigram in T by exploiting broadword search thanks to the word RAM model. We start with the search of single characters and subsequently extend this result to bigrams:

Lemma 6. *We can count the occurrences of a character $c \in \Sigma$ in a string of length $\mathcal{O}(\log_\sigma n)$ in $\mathcal{O}(\lg\lg\lg n)$ time.*

See the full version [14] for a proof, which is a variation of broadword searching zero bytes [13, Sect. 7.1.3]. Having Lemma 6, we show that we can compute the frequency of a bigram in T in $\mathcal{O}(n\lg\lg\lg n/\log_\sigma n)$ time. For that, we interpret $T \in \Sigma^n$ of length n as a text $T \in (\Sigma^2)^{\lceil n/2\rceil}$ of length $\lceil n/2\rceil$. Then we partition T into strings fitting into a computer word, and call each string of this partition a *chunk*. For each chunk, we can apply Lemma 6 by treating a bigram $c \in \Sigma^2$ as a single character. The result is, however, not the frequency of the bigram c in general. For computing the frequency a bigram $\mathtt{bc} \in \Sigma^2$, we distinguish the cases $\mathtt{b} \neq \mathtt{c}$ and $\mathtt{b} = \mathtt{c}$.

Case $\mathtt{b} \neq \mathtt{c}$. By applying Lemma 6 to find the character $\mathtt{bc} \in \Sigma^2$ in a chunk S (interpreted as a string of length $\lfloor q/2 \rfloor$ on the alphabet Σ^2), we obtain the number of occurrences of $\mathtt{bc}$ starting at odd positions in S. To obtain this number for all even positions, we apply the procedure to $\mathtt{d}S$ with $\mathtt{d} \in \Sigma \setminus \{\mathtt{b}, \mathtt{c}\}$. Additional care has to be taken at the borders of each chunk matching the last character of the current chunk and the first character of the subsequent chunk with $\mathtt{b}$ and $\mathtt{c}$, respectively.

Case $\mathtt{b} = \mathtt{c}$. This case is more involving as overlapping occurrences of $\mathtt{bb}$ can occur in S, which we must not count. To this end, we watch out for *runs* of $\mathtt{b}$'s, i.e., substrings of maximal lengths consisting of the character $\mathtt{b}$ (here, we consider also maximal substrings of $\mathtt{b}$ with length 1 as a run). We separate these runs into runs ending either at even or at odd positions. We do this because the frequency of $\mathtt{bb}$ in a run of $\mathtt{b}$'s ending at an even (resp. odd) position is the number of occurrences of $\mathtt{bb}$ within this run ending at an even (resp. odd) position. We can compute these positions similarly to the approach for $\mathtt{b} \neq \mathtt{c}$ by first (a) hiding runs ending at even (resp. odd) positions, and then (b) counting all bigrams ending at even (resp. odd) positions. Runs of $\mathtt{b}$'s that are a prefix or a suffix of S are handled individually if S is neither the first nor the last chunk of T, respectively. That is because a run passing a chunk border starts and ends in different chunks. To take care of those runs, we remember the number of $\mathtt{b}$'s of the longest suffix of every chunk, and accumulate this number until we find the end of this run, which is a prefix of a subsequent chunk. With the aforementioned analysis of the runs crossing chunk borders, we can extend this procedure to count the frequency of $\mathtt{bb}$ in T. We conclude:

Lemma 7. *We can compute the frequency of a bigram in a string T of length n whose characters are drawn from an alphabet of size σ in $\mathcal{O}(n \lg\lg\lg n / \log_\sigma n)$ time.*

3.2 Bit-Parallel Adaption

Similarly to Lemma 2, we present an algorithm computing the d most frequent bigrams, but now with the word-packed search of Lemma 7.

Lemma 8. *Given an integer d with $d \geq 1$, we can compute the frequencies of the d most frequent bigrams in a text of length n whose characters are drawn from an alphabet of size σ in $\mathcal{O}(n^2 \lg\lg\lg n / \log_\sigma n)$ time using $d\lceil \lg(\sigma^2 n/2)\rceil + \mathcal{O}(\lg n)$ bits.*

Proof. We allocate a frequency table F of length d. For each text position i with $1 \leq i \leq n-1$, we compute the frequency of $T[i]T[i+1]$ in $\mathcal{O}(n \lg\lg\lg n / \log_\sigma n)$ time with Lemma 7. After computing a frequency, we insert it into F if it is one of the d most frequent bigrams among the bigrams we have already computed. We can perform the insertion in $\mathcal{O}(\lg d)$ time if we sort the entries of F by their frequencies, yielding $\mathcal{O}((n \lg\lg\lg n / \log_\sigma n + \lg d)n)$ total time.

Studying the final time bounds of Eq. (1) for the sequential algorithm of Section 2, we see that we spend $\mathcal{O}(n^2)$ time in the first turn, but spend less time in later turns. Hence, we want to run the bit-parallel algorithm only in the first few turns until f_k becomes so large that the benefits of running Lemma 2 outweigh the benefits of the bit-parallel approach of Lemma 8. In detail, for the k-th round, we set $d := f_k$ and run the algorithm of Lemma 8 on the current text if d is sufficiently small, or otherwise

the algorithm of Lemma 2. In total, we yield

$$\mathcal{O}\left(\sum_{k=0}^{\mathcal{O}(\lg n)} \min\left(\frac{n-f_k}{f_k} n \lg f_k, \frac{(n-f_k)^2 \lg\lg\lg n}{\log_\tau n}\right)\right) = \mathcal{O}\left(n^2 \sum_{k=0}^{\lg n} \min\left(\frac{k}{\gamma^k}, \frac{\lg\lg\lg n}{\log_\tau n}\right)\right)$$
$$= \mathcal{O}\left(\frac{n^2 \lg\log_\tau n \lg\lg\lg n}{\log_\tau n}\right) \text{ time in total,} \tag{2}$$

where $\tau = \sigma_m$ is the sum of the alphabet size σ and the number of non-terminals, and $k/\gamma^k > \lg\lg\lg n/\log_\tau n \Leftrightarrow k = \mathcal{O}(\lg(\lg n/(\lg\tau \lg\lg\lg n)))$.

To obtain the claim of Theorem 5, it is left to show that the k-th round with the bit-parallel approach uses $\mathcal{O}(n^2 \lg\lg\lg n/\log_\tau n)$ time, as we now want to charge each text position with $\mathcal{O}(n/\log_\tau n)$ time with the same amortized analysis as after Eq. (1). We target $\mathcal{O}(n/\log_\tau n)$ time for

(1) replacing all occurrences of a bigram,
(2) shifting freed up text space to the right,
(3) finding the bigram with the highest or lowest frequency in F,
(4) updating or exchanging an entry in F, and
(5) looking up the frequency of a bigram in F.

Items (1) and (2) can be solved by applying elementary bit-parallel techniques.[8]

For the remaining points, our trick is to represent F by a minimum and a maximum heap, both realized as array heaps. For the space increase, we have to lower α_i (and γ_i) adequately. Each element of an array heap stores a frequency and a pointer to a bigram stored in a separate array B storing all bigrams consecutively. A pointer array P stores pointers to the respective frequencies in both heaps for each bigram of B. The total data structure can be constructed at the beginning of the k-th round in $\mathcal{O}(f_k)$ time, and hence does not worsen the time bounds. While B solves Item (5), the two heaps with P solve Items (3) and (4) even in $\mathcal{O}(\lg f_k)$ time.

In case that we want to store the output in working space, we follow the description of Section 2.4, where we now use word-packing to find the second occurrence of a bigram in T_i in $\mathcal{O}(n/\log_{\sigma_i} n)$ time.

4 Conclusion

In this article, we proposed an algorithm computing Re-Pair in-place in sub-quadratic time for small alphabet sizes. Our major tools are simple, which allowed us to parallelize our algorithm or adapt it in the external memory model.

Acknowledgments

This work is funded by the JSPS KAKENHI Grant Numbers JP18F18120 (Dominik Köppl), 19K20213 (Tomohiro I) and 18K18111 (Yoshimasa Takabatake), and the JST CREST Grant Number JPMJCR1402 including AIP challenge program (Keisuke Goto).

[8] A detailed description is found in the full version of this paper [14].

References

1. H. Bannai, M. Hirayama, D. Hucke, S. Inenaga, A. Jez, M. Lohrey, and C. P. Reh: *The smallest grammar problem revisited*. arXiv 1908.06428, 2019.
2. P. Bille, I. L. Gørtz, and N. Prezza: *Practical and effective Re-Pair compression*. arXiv 1704.08558, 2017.
3. P. Bille, I. L. Gørtz, and N. Prezza: *Space-efficient Re-Pair compression*, in Proc. DCC, 2017, pp. 171–180.
4. T. M. Chan, J. I. Munro, and V. Raman: *Selection and sorting in the "restore" model*. ACM Trans. Algorithms, 14(2) 2018, pp. 11:1–11:18.
5. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat: *The smallest grammar problem*. IEEE Trans. Information Theory, 51(7) 2005, pp. 2554–2576.
6. F. Claude and G. Navarro: *Fast and compact web graph representations*. TWEB, 4(4) 2010, pp. 16:1–16:31.
7. P. De Luca, V. M. Russiello, R. Ciro Sannino, and L. Valente: *A study for image compression using Re-Pair algorithm*. arXiv 1901.10744, 2019.
8. I. Furuya, T. Takagi, Y. Nakashima, S. Inenaga, H. Bannai, and T. Kida: *MR-RePair: Grammar compression based on maximal repeats*, in Proc. DCC, 2019, pp. 508–517.
9. M. Ganczorz: *Entropy lower bounds for dictionary compression*, in Proc. CPM, vol. 128 of LIPIcs, 2019, pp. 11:1–11:18.
10. M. Ganczorz and A. Jez: *Improvements on Re-Pair grammar compressor*, in Proc. DCC, 2017, pp. 181–190.
11. R. González, G. Navarro, and H. Ferrada: *Locally compressed suffix arrays*. ACM Journal of Experimental Algorithmics, 19(1) 2014.
12. J. C. Kieffer and E. Yang: *Grammar-based codes: A new class of universal lossless source codes*. IEEE Trans. Information Theory, 46(3) 2000, pp. 737–754.
13. D. E. Knuth: *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*, Addison-Wesley, 12th ed., 2009.
14. D. Köppl, T. I, I. Furuya, Y. Takabatake, K. Sakai, and K. Goto: *Re-pair in-place*. arXiv 1908.04933, 2019.
15. D. Köppl, T. I, I. Furuya, Y. Takabatake, K. Sakai, and K. Goto: *Re-pair in small space*, in Proc. DCC, 2020, p. 377.
16. N. J. Larsson and A. Moffat: *Offline dictionary-based compression*, in Proc. DCC, 1999, pp. 296–305.
17. M. Lohrey, S. Maneth, and R. Mennicke: *XML tree structure compression using RePair*. Inf. Syst., 38(8) 2013, pp. 1150–1167.
18. T. Masaki and T. Kida: *Online grammar transformation based on Re-Pair algorithm*, in Proc. DCC, 2016, pp. 349–358.
19. G. Navarro and L. M. S. Russo: *Re-Pair achieves high-order entropy*, in Proc. DCC, 2008, p. 537.
20. C. Ochoa and G. Navarro: *RePair and all irreducible grammars are upper bounded by high-order empirical entropy*. IEEE Trans. Information Theory, 65(5) 2019, pp. 3160–3164.
21. K. Sakai, T. Ohno, K. Goto, Y. Takabatake, T. I, and H. Sakamoto: *RePair in compressed space and time*, in Proc. DCC, 2019, pp. 518–527.
22. K. Sekine, H. Sasakawa, S. Yoshida, and T. Kida: *Adaptive dictionary sharing method for Re-Pair algorithm*, in Proc. DCC, 2014, p. 425.
23. Y. Tabei, H. Saigo, Y. Yamanishi, and S. J. Puglisi: *Scalable partial least squares regression on grammar-compressed data matrices*, in Proc. SIGKDD, 2016, pp. 1875–1884.
24. S. Vigna: *Broadword implementation of rank/select queries*, in Proc. WEA, vol. 5038 of LNCS, 2008, pp. 154–168.
25. J. W. J. Williams: *Algorithm 232 - heapsort*. Communications of the ACM, 7(6) 1964, pp. 347–348.
26. S. Yoshida and T. Kida: *Effective variable-length-to-fixed-length coding via a Re-Pair algorithm*, in Proc. DCC, 2013, p. 532.
27. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression*. IEEE Trans. Information Theory, 23(3) 1977, pp. 337–343.

Reducing Time and Space in Indexed String Matching by Characters Distance Text Sampling

Simone Faro and Francesco Pio Marino

Dipartimento di Matematica e Informatica,
Università di Catania, viale A.Doria n.6, 95125, Catania, Italia
faro@dmi.unict.it

Abstract. *Sampled string matching* is an efficient approach to the string matching problem which tries to overcome the prohibitive space requirements of indexed matching, on the one hand, and drastically reduce searching time for the online solutions, on the other hand. Sampled string matching dates back to 1991, however practical solutions to the problem only appeared more recently. They are able to speed up the online searching up to 9 times while using less than 5% of the text size. In this paper we take into account the problem of indexing sampled texts in order to reduce the space requirements of indexed matching and maintaining its very high practical and theoretical performances. Specifically we present a new efficient indexed string matching approach based on a characters distance sampling. The main idea is to sample the distances between consecutive occurrences of a given set of *pivot characters* and then to create a suffix array of the sampled text. From our experimental results it turns out that the newly presented approach is able to obtain a searching time gain up to 91%, when compared to the standard indexed approach, while using less than 15% of the space needed for the standard suffix array.

Keywords: string matching, ext processing, efficient searching, text indexing

1 Introduction

Searching for all occurrences of a pattern in a text is a fundamental problem in computer science with applications in many other fields, like natural language processing, information retrieval and computational biology. In literature such problem is called *String Matching*, and formally consists in finding all occurrences of a given pattern x, of length m, in a large text y, of length n, where both sequences are composed by characters drawn from an alphabet Σ of size σ.

Although data are memorized in different ways, textual data remain the main form to store information. This is particularly evident in literature and in linguistics where data are in the form of huge corpora and dictionaries. But this applies as well to computer science where large amounts of data are stored in linear files. And this is also the case, for instance, in molecular biology where biological molecules are often approximated as sequences of nucleotides or amino acids. Thus the need for more and more faster solutions to text searching problems.

Applications require two kinds of solutions: *online* and *offline* string matching. Solutions based on the first approach assume that the text is not preprocessed and thus they need to scan the text *online*, when searching. Their worst case time complexity is $\Theta(n)$, and was achieved for the first time by the well known Knuth-Morris-Pratt (KMP) algorithm [15], while the optimal average time complexity of the problem is $\Theta(\frac{n \log_\sigma m}{m})$ [22], achieved for example by the Backward-Dawg-Matching (BDM) algorithm [6]. Many string matching algorithms have been also developed to obtain

Simone Faro, Francesco Pio Marino: Reducing Time and Space in Indexed String Matching by Characters Distance Text Sampling, pp. 148–159.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9

sub-linear performance in practice [7]. Among them the Boyer-Moore-Horspool algorithm [2,12] deserves a special mention, since it has been particularly successful and has inspired much work.

Memory requirements of this class of algorithms are very low and generally limited to a precomputed table of size $O(m\sigma)$ or $O(\sigma^2)$ [7]. However their performances may stay poor in many practical cases, especially when used for processing huge input texts and short patterns.[1]

Solutions based on the second approach tries to drastically speed up searching by preprocessing the text and building a data structure that allows searching in time proportional to the length of the pattern. For this reason such kind of problem is known as *indexed searching*. Among the most efficient solutions to such problem we mention those based on suffix trees [1], which find all occurrences in $O(m + occ)$-worst case time, those based on suffix arrays [18], which solve the problem in $O(m + \log n + occ)$ [18], where occ is the number of occurrences of x in y, and those based on the FM-index [10] (Full-text index in Minute space), which is a compressed full-text substring index based on the Burrows-Wheeler transform allowing compression of the input text while still permitting fast substring queries. However, despite their optimal time performance[2], space requirements of full-index data structures, as suffix-trees and suffix-arrays, are from 4 to 20 times the size of the text, while the size of a compressed index, as the FM-Index, is typically less than the size of the text, but its construction may require almost the same space as that required by a full-index. Such space requirement is too large for many practical applications.

A different solution to the problem is to compress the input text and search online directly the compressed data in order to speed-up the searching process using reduced extra space. Such problem, known in literature as *compressed string matching*, has been widely investigated in the last few years. Although efficient solutions exist for searching on standard compressions schemes, as Ziv-Lempel [20] and Huffman [3], the best practical behaviour are achieved by ad-hoc schemes designed for allowing fast searching [17,19,14,21,11]. These latter solutions use less than 70% of text size extra space (achieving a compression rate over 30%) and are twice as fast in searching as standard online string matching algorithms. A drawback of such solutions is that most of them still require significant implementation efforts and a high time for each reported occurrence.

A more suitable solution to the problem is *sampled string matching*, introduced in 1991 da Vishkin [23], which consists in the construction of a succint sampled version of the text and in the application of any online string matching algorithm directly on the sampled sequence. The drawback of this approach is that any occurrence reported in the sampled-text may require to be verified in the original text. However a sampled-text approach may have a lot of good features: it may be easy to implement, may require little extra space and may allow fast searching. Additionally it may allow fast updates of the data structure.

Apart the theoretical result of Vishkin, a more practical solution to sampled string matching has been recently introduced by Claude *et al.* [5], based on an alphabet

[1] Search speed of an online string matching algorithm may depend on the length of the pattern. Typical search speed of a fast solution, on a modern laptop computer, goes from 1 GB/s (in the case of short patterns) to 5 GB/s (in the case of very long patterns) [4].

[2] Search speed of a fast offline solution do not depend on the length of the text and is typically under 1 millisecond per query.

reduction. Their algorithm has an extra space requirement which is only 14% of text size and is up to 5 times faster than standard online string matching on English texts. Thus it turns out to be one of the most effective and flexible solution for this kind of searching problems.

They also consider indexing the sampled text. They build a suffix array indexing the sampled positions of the text, and get a sampled suffix array. This approach is similar to the sparse suffix array [13] as both index a subset of the suffixes, but the different sampling properties induce rather different search algorithms and performance characteristics.

In this paper we present a new approach to the indexed string matching problem based on a different text sampling approach. The main idea behind our new text sampling approach is to sample the distances between consecutive occurrences of a given set of *pivot characters* and then to create a suffix array of the sampled text. For this reason we call this approach *characters distance sampling* (Cds). From our experimental results it turns out that our indexed solution based on such new sampling approach is able to obtain a searching time gain up to 91%, when compared to the standard indexed approach, while using less than 15% of the space needed for the standard suffix array.

The paper is organized as follows. In Section 2 we introduce in details the sampled string matching problem and present the first indexed approach to the problem proposed by Claude *et al.* [5]. Then, in Section 3, we present the new Cds approach to text sampling and propose an alternative indexed algorithm based on the suffix array. In Section 4 we present experimental results in order to compare our new solution against that proposed by Claude *et al.* and against the standard suffix array solution, in terms of both space and time.

2 Sampled String Matching

The task of the *sampled string matching* problem is to find all occurrences of a given pattern x, of length m, in a given text y, of length n, assuming that a fast and succint preprocessing of the text is allowed in order to build a data-structure, which is used to speed-up the searching phase. For its features we call such data structure a *partial-index* of the text.

In order to be of any practical and theoretical interest a partial-index of the text should:

(1) *be succint*: since it must be maintained together with the original text, it should require few additional spaces to be constructed;
(2) *be fast to build*: it should be constructed using few computational resources, also in terms of time. This should allow the data structure to be easily built online when a set of queries is required;
(3) *allow fast search*: it should drastically increase the searching time of the underlying string matching algorithm. This is one of the main features required by this kind of solutions;
(4) *allow fast update*: it should be possible to easily and quickly update the data structure if modifications have been applied on the original text. A desirable update procedure should be at least as fast as the modification procedure on the original text.

Sampled string matching has been introduced for the first time by Claude *et al.* in [5] where the author presented an online and an offline solution to the problem. More recently Faro *et al.* presented in an unpublished paper [13] an alternative solution for the online problem which turns out to be more efficient both in terms of space consumption and running times.

In this section we briefly describe the efficient indexed text-sampling approach proposed by Claude *et al.*. We will refer to this solution as the Occurrence-Text-Sampling approcah (OTS). To the best of our knowledge it is the only effective and flexible solution known in literature for the indexed sampled matching problem.

2.1 The Occurrence Text Sampling algorithm

Let y be the input text, of length n, and let x be the input pattern, of length m, both over an alphabet Σ of size σ. The main idea of their sampling approach is to select a subset of the alphabet, $\hat{\Sigma} \subset \Sigma$ (the sampled alphabet), and then to construct a partial-index as the subsequence of the text (the sampled text) $\hat{y}$, of length $\hat{n}$, containing all (and only) the characters of the sampled alphabet $\hat{\Sigma}$. More formally $\hat{y}[i] \in \hat{\Sigma}$, for all $1 \leq i \leq \hat{n}$.

During the searching phase of the algorithm a sampled version of the input pattern, $\hat{x}$, of length $\hat{m}$, is constructed and searched in the sampled text. Since $\hat{y}$ contains partial information, for each candidate position i returned by the search procedure on the sampled text, the algorithm has to verify the corresponding occurrence of x in the original text. For this reason a table ρ is maintained in order to map, at regular intervals, positions of the sampled text to their corresponding positions in the original text. The position mapping ρ has size $\lfloor \hat{n}/q \rfloor$, where q is the *interval factor*, and is such that $\rho[i] = j$ if character $y[j]$ corresponds to character $\hat{y}[q \times i]$. The value of $\rho[0]$ is set to 0. In their paper, on the basis of an accurate experimentation, the authors suggest to use values of q in the set $\{8, 16, 32\}$

Then, if the candidate occurrence position j is stored in the mapping table, i.e if $\rho[i] = j$ for some $1 \leq i \leq \lfloor \hat{n}/q \rfloor$, the algorithm directly checks the corresponding position in y for the whole occurrence of x. Otherwise, if the sampled pattern is found in a position r of $\hat{y}$, which is not mapped in ρ, the algorithm has to check the substring of the original text which goes from position $\rho[r/q] + (r \mod q) - \alpha + 1$ to position $\rho[r/q + 1] - (q - (r \mod q)) - \alpha + 1$, where α is the first position in x such that $x[\alpha] \in \hat{\Sigma}$.

Notice that, if the input pattern does not contain characters of the sampled alphabet, i.e. id $\bar{m} = 0$, the algorithm merely reduces to search for x in the original text y.

Example 1. Suppose $y =$ "abaacabdaacabcc" is a text of length 15 over the alphabet $\Sigma =$ {a,b,c,d}. Let $\hat{\Sigma} =$ {b,c,d} be the sampled alphabet, by omitting character "a". Thus the sampled text is $\hat{y} =$ "bcbdcbcc". If we map every $q = 2$ positions in the sampled text, the position mapping ρ is $\langle 5, 8, 12, 14 \rangle$. To search for the pattern $x =$ "acab" the algorithm constructs the sampled pattern $\hat{x} =$ "cb" and search for it in the sampled text, finding two occurrences at position 2 and 5, respectively. We note that $\hat{y}[2]$ is mapped and thus it suffices to verify for an occurrence starting at position 4, finding a match. However position $\hat{y}[5]$ is not mapped, thus we have to search in the substring $y[\rho(2) + 3 - 1..\rho(3)]$, finding no matches.

The above algorithm works well with most of the known pattern matching algorithms. However, since the sampled patterns tend to be short, the authors implemented the search phase using the Horspool algorithm, which has been found to be fast in such setting.

The real challenge in their algorithm is how to choose the best alphabet subset to sample. Based on some analytical results, supported by an experimental evaluation, they showed that it suffices in practice to sample the least frequent characters up to some limit.[3] Under this assumption their algorithm has an extra space requirement which is only 14% of text size and is up to 5 times faster than standard online string matching on English texts.

In [5] the authors also consider indexing the sampled text. Specifically they build a suffix array for the sampled text in order to index the sampled positions of the text. This approach is similar to the sparse suffix array [13] as both index a subset of the suffixes, but the different sampling properties induce rather different search algorithms and performance characteristics.

To optimize the entire process when constructing the suffix array their algorithm stores only suffixes starting with a sampled character. As a consequence, this approach can only be used for patterns which contain at least one character of the sampled alphabet.

The searching phase can be summarised as follows: the pattern is divided into two parts, the unsampled prefix, and the suffix starting with a sampled character. First of all the algorithm searches the sampled suffix of the pattern using the suffix array. Each candidate occurrence retrieved by this preliminary search will then be verified by comparing the unsampled prefix against the original text.

This approach turns out to perform very well for moderate to long patterns. Specifically according to their experimental evaluation it turns out that, when searching on an English text, the best performance are obtained when the number of removed characters from the original alphabet ranges between 13 and 16.

3 An Approach Based on Characters Distance Sampling

In this section we present a new efficient approach to the sampled string matching problem, introducing a new method[4] for the construction of the partial-index, which turns out to require limited additional space, still maintaining the same performance of the algorithm recently introduced by Claude *et al.* [5]. In the next subsections we illustrate in details our idea and describe the algorithms for the construction of the sampled text.

3.1 Characters Distance Sampling

Let y be the input text, of length n, and let x be the input pattern, of length m, both over an alphabet Σ of size σ. We assume that all strings can be treated as vectors

[3] According to their theoretical evaluation and their experimental results it turns out that, when searching on an English text, the best performance are obtained when the least 13 characters are removed from the original alphabet.

[4] The Charcaters Distance Sampling approach presented in this section has been previously introduced by the authors in [9] where the online string matching problem has been taken into account.

starting at position 1. Thus we refer to $x[i]$ as the i-th character of the string x, for $1 \leq i \leq m$, where m is the size of x.

We elect a set $C \subseteq \Sigma$ to be the *set of pivot characters*. Given this set of characters we sample the text y by taking into account the distances between consecutive positions of any character of C in y. More formally our sampling approach is based on the following definition of *position sampling* of a text.

Definition 2 (Position Sampling). *Let y be a text of length n, let $C \subseteq \Sigma$ be the set of pivot characters and let n_c be the number of occurrences of any $c \in C$ in the input text y.*

First we define the position function, *$\delta : \{1, .., n_c\} \rightarrow \{1, .., n\}$, where $\delta(i)$ is the position of the i-th occurrence of any character of C in y. Formally we have*

$$\begin{array}{ll} (i)\ 1 \leq \delta(i) < \delta(i+1) \leq n & \text{for each } 1 \leq i \leq n_c - 1 \\ (ii)\ y[\delta(i)] \in C & \text{for each } 1 \leq i \leq n_c \\ (iii)\ y[\delta(i)+1..\delta(i+1)-1] \text{ contains no character of } C & \text{for each } 0 \leq i \leq n_c \end{array}$$

where in (iii) we assume that $\delta(0) = 0$ and $\delta(n_c + 1) = n + 1$.

Then the position sampled version of y, indicated by $\dot{y}$, is a numeric sequence, of length n_c, defined as

$$\dot{y} = \langle \delta(1), \delta(2), .., \delta(n_c) \rangle. \tag{1}$$

Example 3. Suppose $y =$ "agaacgcagtata" is a DNA sequence of length 13, over the alphabet $\Sigma = \{$a,c,g,t$\}$. Let $C = \{$a$\}$ be the set of pivot characters. Thus the position sampled version of y is $\dot{y} = \langle 1, 3, 4, 8, 11, 13 \rangle$. Specifically the first occurrence of character c is at position 1 ($y[1] =$ a), its second occurrence is at position 3 ($y[3] =$ a), and so on.

Definition 4 (Characters Distance Sampling). *Let $C \subseteq \Sigma$ be the set of pivot characters, let $n_c \leq n$ be the number of occurrences of any pivot character in the text y and let δ be the position function of y. We define the* characters distance function *$\Delta(i) = \delta(i+1) - \delta(i)$, for $1 \leq i \leq n_c - 1$, as the distance between two consecutive occurrences of any pivot character in y.*

Then the characters-distance sampled version *of the text y is a numeric sequence, indicated by $\bar{y}$, of length $n_c - 1$ defined as*

$$\bar{y} = \langle \Delta(1), \Delta(2), .., \Delta(n_c - 1) \rangle = \langle \delta(2) - \delta(1), \delta(3) - \delta(2), .., \delta(n_c) - \delta(n_c - 1) \rangle \tag{2}$$

Plainly we have

$$\sum_{i=1}^{n_c - 1} \Delta(i) \leq n - 1.$$

Example 5. Let $y =$ "agaacgcagtata" be a text of length 13, over the alphabet $\Sigma = \{$a,c,g,t$\}$. Let $C = \{$a$\}$ be the pivot character. Thus the character distance sampling version of y is $\bar{y} = \langle 2, 1, 4, 3, 2 \rangle$. Specifically $\bar{y}[1] = \Delta(1) = \delta(2) - \delta(1) = 3 - 1 = 2$, while $\bar{y}[3] = \Delta(3) = \delta(4) - \delta(3) = 8 - 4 = 4$, and so on.

Definition 6 (Rank of a character). *Let x be a pattern of length m, and let $c \in \Sigma$. We define $\phi : \Sigma \rightarrow \{0..m\}$ as the function which associates any character of the text with the number of its occurrences in x. The rank of the character c is the position of c in the alphabet Σ, if we assume that all characters are sorted by their $\phi(c)$ values. More formally the rank of c is given by the cardinality of the set $\{k \in \Sigma \mid \phi(k) < \phi(c)\} + 1$*

3.2 String Matching with the Suffix Array of the Sampled Text

In this section we describe an approach to indexed searching which makes use of a suffix array constructed over the sampled version of the text.

We remember that a *suffix array* is a sorted array of all suffixes of a string. Such data structure has been introduced by Manber and Myers in 1990 [18] as a simple, space efficient alternative to suffix trees. It has been extensively studied in the last three decades and in 2016 Li, Li and Huo [16] gave the first in-place $O(n)$-time construction algorithm that is optimal both in time and space, where in-place means that the algorithm only needs $O(1)$ additional space beyond the input string and the output suffix array.

Formally, given a text y of length n, the suffix array s_y of y is defined to be an array of integers providing the starting positions of suffixes of y in lexicographical order. This means that $s_y[i]$ contains the starting positions of the i-th smallest suffix in y and thus for all $1 \leq i \leq n$, we have $y[s_y[i-1]..n] < y[s_y[i]..n]$.

The algorithm proposed in this section is divided into two phases: a first *preprocessing phase* which consists in the construction of a suffix array of the sampled version of the text and a *searching phase* which is used to search any pattern x of length m in y making use of the suffix array $s_{\bar{y}}$ and the sampled text $\dot{y}$. We notice that the preprocessing phase is performed only once for the construction of the partial index, while the searching phase can be run an indeterminate number of times. We notice also that the algorithm must maintain the original text y, the sampled version of the text $\dot{y}$ and the corresponding suffix array $s_{\bar{y}}$.

In this paper we do not go into the way for a correct selection of the set of pivot characters, and even we leave the details of an analysis about what is the best subset to be chosen. However in our experimental evaluation (see Section 4) we will show how it is enough to put a single character in the set of pivot characters to obtain very good and competitive results. We select such character on the basis of its *rank value*, where we remember that the rank of a character c corresponds to its position in the alphabet Σ when we assume that all characters are sorted by their frequencies inside the text (see Definition 6).

We are now ready to describe the preprocessing and the searching phase of our new proposed algorithm.

Let y be an input text of length n over an alphabet Σ of size σ and let $C \subseteq \Sigma$ be the set of pivot characters. During the preprocessing phase the algorithm builds and stores the position sampled text $\dot{y}$ of y. This requires $O(n)$-time and $O(n_c)$-space, where n_c is the number of occurrences of any pivot character in y. Subsequently a suffix array of $\bar{y}$ is constructed.

However when constructing the suffix array of $\bar{y}$, the algorithm takes into account only suffixes beginning with a pivot character in the original text, drastically reducing the space requirement for maintaining the whole index. Apart from this detail, all other features of the data structure remain unchanged.

Example 7. Let $y =$ "agaacgcagtata" be a text of length 13, over the alphabet $\Sigma = \{\text{a,c,g,t}\}$. Let $C = \{\text{a}\}$ be the pivot character. Thus the character distance sampling version of y is $\bar{y} = \langle 2, 1, 4, 3, 2 \rangle$.

$s_{\bar{y}}[0] = 1 \rightarrow \langle 1, 4, 3, 2\rangle$
$s_{\bar{y}}[1] = 0 \rightarrow \langle 2\rangle$
$s_{\bar{y}}[2] = 4 \rightarrow \langle 2, 1, 4, 3, 2\rangle$
$s_{\bar{y}}[3] = 3 \rightarrow \langle 3, 2\rangle$
$s_{\bar{y}}[4] = 2 \rightarrow \langle 4, 3, 2\rangle$

During the searching phase the algorithm uses the suffix array of the sampled text $s_{\bar{y}}$ as an index to quickly locate every occurrence of a sampled pattern $\bar{x}$ in $\bar{y}$. Each of these occurrences is treated as a candidate occurrence of x in y, and as such it will be verified by a comparison procedure.

The searching algorithm works as a standard search on a suffix array. It is based of the fact that finding every occurrence of the pattern $\bar{x}$ is equivalent to finding every suffix in $\bar{y}$ that begins with the $\bar{x}$. Thanks to the lexicographical ordering of the suffix array, all such suffixes are grouped together and can be found efficiently with a single binary search, which locates the starting position of the interval. All other occurrence are then grouped together close the first one.

Finding the first position of a sampled pattern $\bar{x}$ of length m_c in a suffix array $s_{\bar{y}}$ of length n_c takes $O(\log n_c)$-time [18] while finding the set set of all ρ occurrences of $\bar{x}$ in $\bar{y}$ takes $O(\rho)$-time. Since each occurrence must be verified in the original text we need $O(m\rho)$ additional time for the verification phase. The overall time complexity of the searching algorithm is then $O(\log(n_c) + m\rho)$.

4 Experimental Evaluation

In this section we report the results of an extensive evaluation of the new presented indexed algorithm based on Character Distance Sampling (CDS) in comparison with the standard searching algorithm based on the suffix array and the indexed algorithm based on the Occurrence Text Sampling (OTS) approach by Claude *et al.* [5].

The algorithms have been implemented in C, and have been tested using a variant of the SMART tool [8], properly tuned for testing string matching algorithms based on the indexed text-sampling approach, and executed locally on a MacBook Pro with 4 Cores, a 2 GHz Intel Core i7 processor, 16 GB RAM 1600 MHz DDR3, 256 KB of L2 Cache and 6 MB of Cache L3.[5]

Comparisons have been performed in terms of space consumption and searching times. For our tests, we used the English text of size 5 MB provided by the research tool SMART, available online for download.[6]

In the context of text-sampling string-matching space requirement is one of the most significant parameter to take into account. It indicates how much additional space, with regard to the size of the text, is required by a given solution to solve the problem.

Figure 1 shows the space consumption of the newly proposed text-sampling algorithms for different values of the rank r of the pivot character, whose value ranges from 2 to 20. Specifically it shows the size of the additional space (the size of the

[5] The SMART tool is available online for download at `http://www.dmi.unict.it/~faro/smart/` or at `https://github.com/smart-tool/smart`.

[6] Specifically, the text buffer is the concatenation of two different texts: The King James version of the bible (3.9 MB) and The CIA world fact book (2.4 MB). The first 5 MB of the resulting text buffer have been used in our experimental results.

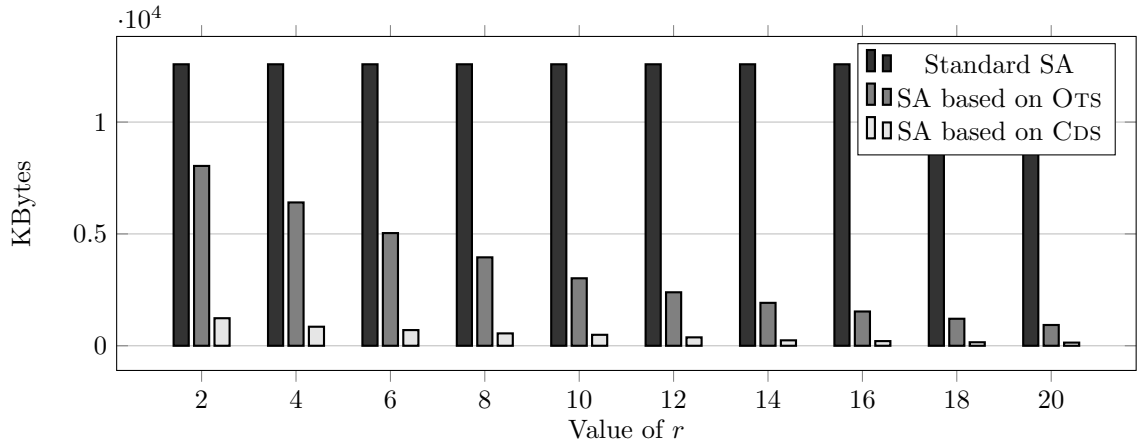


Figure 1. Size of the additional space (the size of the suffix array plus the size of the sampled text) consumed using the three compared approaches and specifically: the standard suffix array, the suffix array of a sampled text using an Ots approach and the suffix array of a sampled text using an Cds approach. Sizes are represented in KBytes. The x axis represents the rank of the pivot character in the case of the Cds approach, while represents the number of removed characters in the case of the Ots approach. We used a natural language text of size 5 Mb.

suffix array plus the size of the sampled text) consumed using the three compared approaches: the standard suffix array, the suffix array of a sampled text using an Ots approach and the suffix array of a sampled text using an Cds approach. Notice that sizes are represented in KBytes. The x axis represents the rank of the pivot character in the case of the Cds approach, while represents the number of removed characters in the case of the Ots approach. We used a natural language text of size 5 Mb.

As expected, the function which describes memory requirements follows a decreasing trend while the value of r decreases. Specifically the benefit in space consumption obtained by the algorithms based on character distance sampling ranges from 70% to 80% when compared with the Ots approach.

Figure 2 and Figure 3 show the experimental results obtained by comparing the three approaches to indexed searching in terms of running times, in a graphic and in a tabular representation, respectively. In the experimental evaluation, patterns of length m were randomly extracted from the text (thus the number of reported occurrences is always greater than 0), with m ranging over the set of values $\{8, 16, 32, 64, 128, 256\}$. In all cases, the sum over the running times (expressed in milliseconds) of 1000 runs has been reported.

From such experimental results it turns out that the indexed searching approach based on Ots reaches a speed-up between 61% and 74%, while our new proposed solution reaches a speed-up between 80% and 91%. In addition the best results are obtained in all cased by the approach based on character distance sampling. In general the behaviour of text-sampling algorithms follow an increasing trend for increasing rank values. Thus in most cases the better choice is to use the most frequent element as the pivot character.

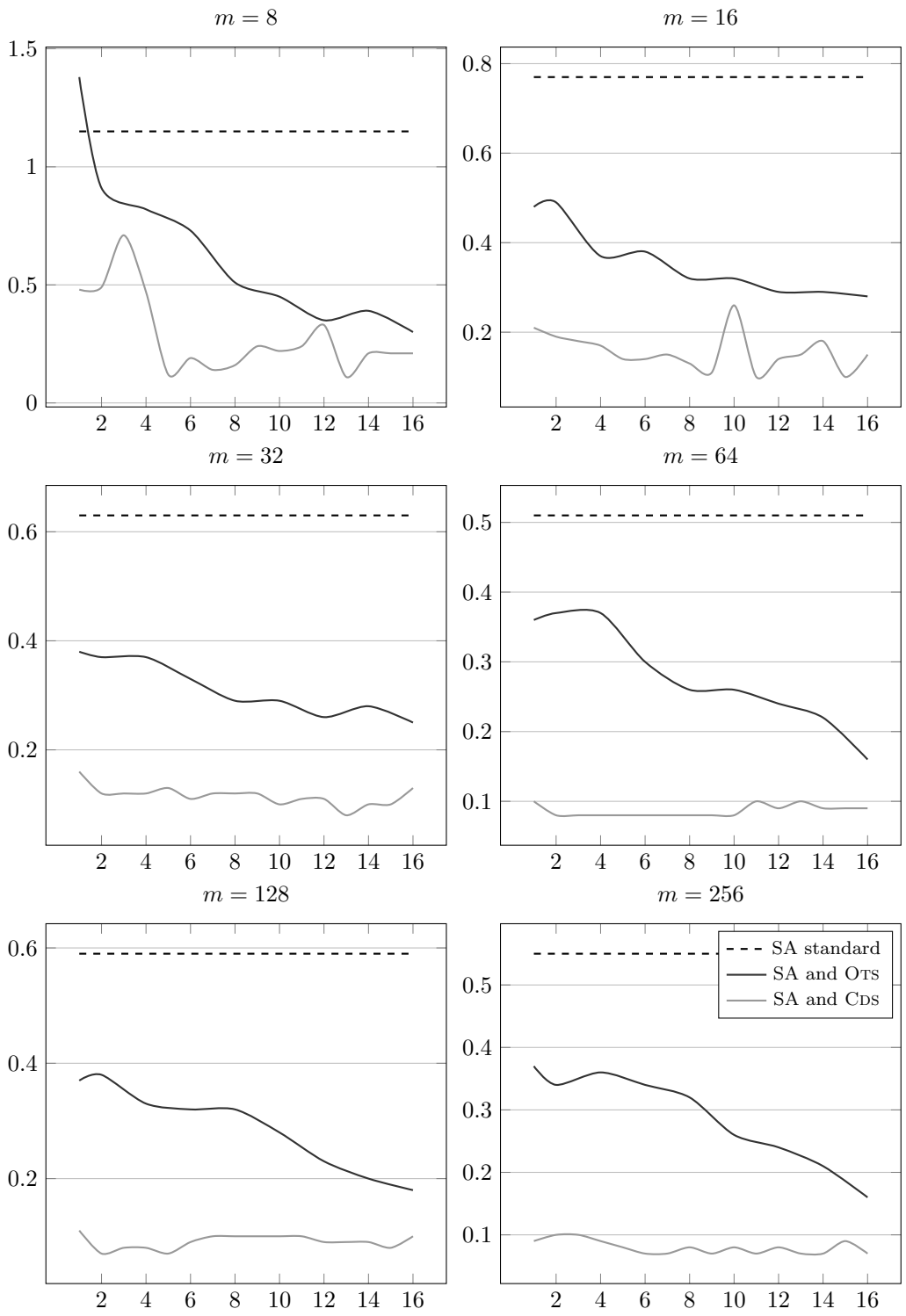


Figure 2. Running times of the text sampling algorithms in the case of long patterns ($8 \leq m \leq 256$). The dashed red line represent the Searching time of the standard Suffix array. Running times (in the y axis) are represented in thousands of seconds. The x axis represents the rank of the pivot character in the case of the new algorithm, while represents the number of removed characters in the case of the Ots algorithms. The searching time represented is the sum of all the 1000 tests executed for each length of the pattern. We are using a text of size 5 Mb.

$m = 8$

r	1	2	4	6	8	10	12	14	16
SA standard	1.15	1.15	1.15	1.15	1.15	1.15	1.15	1.15	1.15
SA and Ots	0.48	0.49	0.37	0.38	0.32	0.32	0.29	0.29	0.28
SA and Cds	0.48	0.49	0.47	0.19	0.16	0.22	0.33	0.21	0.21

$m = 16$

r	1	2	4	6	8	10	12	14	16
SA standard	0.77	0.77	0.77	0.77	0.77	0.77	0.77	0.77	0.77
SA and Ots	0.48	0.49	0.37	0.38	0.32	0.32	0.29	0.29	0.28
SA and Cds	0.21	0.19	0.17	0.14	0.13	0.26	0.14	0.18	0.15

$m = 32$

r	1	2	4	6	8	10	12	14	16
SA standard	0.63	0.63	0.63	0.63	0.63	0.63	0.63	0.63	0.63
SA and Ots	0.38	0.37	0.37	0.33	0.29	0.29	0.26	0.28	0.25
SA and Cds	0.16	0.12	0.12	0.11	0.12	0.10	0.11	0.10	0.13

$m = 64$

r	1	2	4	6	8	10	12	14	16
SA standard	0.51	0.51	0.51	0.51	0.51	0.51	0.51	0.51	0.51
SA and Ots	0.36	0.37	0.37	0.30	0.26	0.26	0.24	0.22	0.16
SA and Cds	0.10	0.08	0.08	0.08	0.08	0.08	0.10	0.09	0.09

$m = 128$

r	1	2	4	6	8	10	12	14	16
SA standard	0.59	0.59	0.59	0.59	0.59	0.59	0.59	0.59	0.59
SA and Ots	0.37	0.38	0.33	0.32	0.32	0.28	0.23	0.20	0.18
SA and Cds	0.11	0.07	0.08	0.09	0.10	0.10	0.09	0.09	0.10

$m = 256$

r	1	2	4	6	8	10	12	14	16
SA standard	0.55	0.55	0.55	0.55	0.55	0.55	0.55	0.55	0.55
SA and Ots	0.37	0.34	0.36	0.34	0.32	0.26	0.24	0.21	0.16
SA and Cds	0.09	0.1	0.09	0.07	0.08	0.08	0.08	0.07	0.07

Figure 3. Running times of the text sampling algorithms in the case of long patterns ($8 \leq m \leq 256$). The dashed red line represent the Searching time of the standard Suffix array. Running times (in the y axis) are represented in thousands of seconds. The x axis represents the rank of the pivot character in the case of the new algorithm, while represents the number of removed characters in the case of the Ots algorithms. The searching time represented is the sum of all the 1000 tests executed for each length of the pattern. We are using a text of size 5 Mb.

References

1. A. Apostolico: *The myriad virtues of suffix trees.* In: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, Vol. 12 of NATO Advanced Science Institutes, Series F, Springer-Verlag, pp. 85–96 (1985).
2. R.S. Boyer and J.S. Moore: *A fast string searching algorithm.* Commun. ACM 20(10), 762-772 (1977).
3. D. Cantone, S. Faro, and E. Giaquinta: *Adapting Boyer-Moore-like Algorithms for Searching Huffman Encoded Texts.* Int. J. Found. Comput. Sci. 23(2), pp. 343–356 (2012).
4. D. Cantone, S. Faro, A. Pavone: *Speeding Up String Matching by Weak Factor Recognition.* Stringology 2017, pp. 42–50 (2017).
5. F. Claude, G. Navarro, H. Peltola, L. Salmela, and J. Tarhio: *String matching with alphabet sampling.* Journal of Discrete Algorithms, vol. 11, pp. 37–50 (2012).
6. M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, W. Rytter: *Speeding up two string-matching algorithms.* Algorithmica 12 (4), pp. 247–267 (1994).
7. S. Faro and T. Lecroq: *The Exact Online String Matching Problem: a Review of the Most Recent Results.* ACM Computing Surveys (CSUR) vol. 45 (2), pp. 13 (2013).

8. S. Faro, T. Lecroq, S. Borzì, S. Di Mauro, and A. Maggio: *The String Matching Algorithms Research Tool.* In Proc. of Stringology, pages 99–111, 2016.
9. S. Faro, F.P. Marino, and A. Pavone: *Efficient Online String Matching Based on Characters Distance Text Sampling.* arXiv:1908.05930, 2018.
10. P. Ferragina and G. Manzini: *Indexing compressed text.* Journal of the ACM, 52 (4), pp. 552–581, 2005.
11. K. Fredriksson and S. Grabowski: *A general compression algorithm that supports fast searching.* Information Processing Letters, vol. 100 (6), pp. 226–232 (2006).
12. R. N. Horspool: *Practical fast searching in strings.* Software: Practice & Experience 10 (6), pp. 501–506 (1980).
13. J. Karkkainen and E. Ukkonen: *Sparse suffix trees.* In: Proc. 2nd Annual International Conference on Computing and Combinatorics (COCOON), LNCS 1090, pp. 219–230 (1996).
14. S. T. Klein and D. Shapira: *A new compression method for compressed matching.* In: Data Compression Conference, IEEE. pp. 400–409 (2000).
15. D. E. Knuth, J. H. Morris, and V. R. Pratt: *Fast pattern matching in strings.* SIAM J. Comput. 6 (2), pp. 323–350 (1977).
16. Z. Li, J. Li, and H. Huo: *Optimal In-Place Suffix Sorting.* Proceedings of the 25th International Symposium on String Processing and Information Retrieval (SPIRE). Lecture Notes in Computer Science. 11147. Springer. pp. 268–284 (2016).
17. U. Manber: *A text compression scheme that allows fast searching directly in the compressed file.* ACM Trans. Inform. Syst., 15(2), pp.124–136 (1997).
18. U. Manber and G. Myers: *Suffix arrays: A new method for online string searches.* SIAM J. Comput. 22 (5), pp. 935–948 (1993).
19. E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates: *Fast and flexible word searching on compressed text.* ACM Transactions on Information Systems (TOIS), 18(2), pp.113–139 (2000).
20. G. Navarro and J. Tarhio: *LZgrep: A Boyer-Moore string matching tool for Ziv-Lempel compressed text.* Software Practice & Experience, vol. 35, pp. 1107–1130 (2005).
21. Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa: *Speeding Up Pattern Matching by Text Compression.* CIAC 2000: pp. 306–315.
22. A. C. Yao: *The complexity of pattern matching for a random string.* SIAM J. Comput. 8 (3), pp. 368–387 (1979).
23. U. Vishkin: *Deterministic sampling – A new technique for fast pattern matching.* In Proc. of the ACM Symposium on Theory of Computing (STOC), pp.170-180 (1990).

Tune-up for the Dead-Zone Algorithm

Jorma Tarhio[1] and Bruce W. Watson[2]

[1] Department of Computer Science
Aalto University, Finland
jorma.tarhio@aalto.fi
[2] Department of Information Science
Stellenbosch University, South Africa
bruce@fastar.org

Abstract. We present a number of performance tuning techniques as applied to the Dead-Zone algorithm for exact single (keyword) pattern matching in strings in sequential processing. The tuning techniques presented here are focused on the algorithm skeleton as well as how the shifters are used, and include: removal of some redundant computation, and shifting using 2-grams, among others. Benchmarking results are given for the C implementation in a modern processor without penalties for misaligned memory access.

1 Introduction

String searching is a common task in any software which processes text. The task can be implemented either as an index search, like in web search engines, or as a local search—as in a web browser showing a loaded web page. Here we consider only the latter one where the text to be searched has not been processed beforehand. Formally, the *exact string matching problem* is defined as follows: given a pattern $P = p_0 \cdots p_{m-1}$ and a text $T = t_0 \cdots t_{n-1}$ both in an alphabet Σ, find all the occurrences (including overlapping ones) of P in T. So far, dozens of algorithms have been developed for this problem, see e.g. [5].

We present a number of performance tuning techniques as applied to the Dead-Zone algorithms for exact single pattern matching in strings in sequential processing. The original algorithm is actually a family of algorithms, accommodating numerous possible shifters in a way similar to what the Boyer-Moore family does. Because the Dead-Zone algorithm applies two-way shifting, it is possible to construct inputs for which the algorithm makes fewer comparisons than other comparison-based algorithms.

The tuning techniques presented here are focused on the algorithm skeleton as well as how the shifters are used, and include: removal of some redundant computation (surprisingly, not caught by the optimising compiler), and shifting using 2-grams, among others. Benchmarking results are given for the C implementation. The experiments show that tuning triples the speed of the algorithm.

The rest of the paper is organised as follows. Section 2 present principles of the Dead-Zone algorithms and introduces the base algorithm. Section 3 shows how we optimised the base algorithm. Section 4 gives the results of our practical experiments, and the discussion of Section 5 concludes the article.

2 Background

Here, we only give a brief introduction to the functioning of the Dead-Zone, while some other papers [3,12,16,17] provide a broader picture. Some of the performance details

Jorma Tarhio, Bruce W. Watson: Tune-up for the Dead-Zone Algorithm, pp. 160–167.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9

are discussed in [12], which also offers more pointers to various Dead-Zone versions as well as correctness proofs. In particular, that paper gives some simple recursive versions of Dead-Zone before presenting a loop-based (non-recursive) implementation which explicitly maintains a stack for efficiency. That loop-based version, known as *DZ(iter,sh)* in [12], is slightly improved and presented here as Algorithm DZ0—with an explanation below.

Algorithm DZ0 (Dead-Zone)

```
 1  lo ← 0; hi ← n − (m − 1)
 2  count ← 0
 3  push(0, max)
 4  while true do
 5     probe ← ⌊(lo + hi)/2⌋
 6     i ← 0
 7     while i < m and p_i = t_{probe+i} do i ← i + 1
 8     if i = m then count ← count + 1
 9     kdleft ← probe − shl[t_probe] + 1
10     kdright ← probe + shr[t_{probe+m−1}]
11     if lo < kdleft then
12        push(kdright, hi)
13        hi ← kdleft
14     else
15        lo ← kdright
16        if lo ≥ hi then
17           while top.first ≥ top.second do pop
18           if top.second = max then return count
19           else
20              lo ← top.first
21              hi ← top.second
22              pop
```

As mentioned earlier, rather than recursion, this version of Dead-Zone maintains a stack of 'live-zones'—substrings of the text T still to be considered for matches; each such substring is represented by its beginning index 'lo' (inclusive) and end index 'hi' (not inclusive) [12]. Line 3 pushes a sentinel live-zone onto the stack to make empty-stack detection more efficient, and initialises the *lo* and *hi* variables to indicate that the entire string is still a live-zone, keeping in mind that the upper bound is $n - (m - 1)$ because matches cannot occur in the last $m - 1$ symbols.

The outer while loop introduced in line 4 has no guard and is exited when the sentinel element of the stack is popped, indicating there are no more live-zones to consider.

In general, Dead-Zone algorithm variants are divide-and-conquer style—with a resemblance to Quick Sort. Lines 5–8 determines the mid-point/probe of the current live-zone (line 5), then uses a small inner loop (called a match loop) to make a match attempt at that position (lines 6 and 7), and notes any match by incrementing a counter (line 8); as with the SMART framework[1], only the number of matches is tracked as opposed to the positions of all matches.

Line 10 (we return to line 9 shortly) uses a (precomputed) lookup table *shr* giving a *right shift* used to split the current live-zone and compute the *new lo* of the right-hand portion. In this particular version, the lookup table is indexed by the character aligned

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

with the end of P, namely $t_{probe+m-1}$; this is known as the Horspool shifter [7], and its precomputation is not discussed here. In fact, any Boyer-Moore style shift function could have been used, and this is one of the optimisation opportunities discussed in the next section. Line 9 similarly uses a *symmetrical* left shift table *shl*[2] to give the *new hi* of the left-hand split of the current live-zone. Again, precomputation of that shift table is not discussed here.

Because shifters are symmetrical and *kdleft* is a lower bound pointing to the first dead position, one must be added to it in line 9. Alternatively, this addition could be incorporated into the table *shl*.

Lines 11–13 evaluate whether the newly-determined left-hand portion is empty (test, line 11)—if not, it needs to be explored, and the newly-determined right-hand portion is pushed onto the stack (line 12) for later consideration before proceeding with the left-hand portion (line 13).

Lines 14 onwards are for the case where the left-hand portion *is empty*, meaning that we proceed only with the newly-determined right-hand portion, starting with line 15. Line 16 onwards considers the possibility that this newly-determined right-hand portion is also empty, in which case elements are repeatedly popped from the stack (loop on line 17) until the top-of-stack contains a non-empty live-zone. If the top-of-stack is the sentinel pushed in line 3, the algorithm has fully explored T and it returns (line 18). If not, the top-of-stack contains the live-zone to use and *lo* and *hi* are appropriately updated and that element popped.

Execution then continues (in the live-zone just determined in lines 11–22) at the top of the loop in line 4.

3 Development

Our aim was to develop a faster version of the Dead-Zone algorithm. We tried several local changes to Algorithm DZ0 and evaluated experimentally how they affected the performance. As a result, we ended up suggesting three local optimisations to the Dead-Zone algorithm.

3.1 Elimination of Dead-Zones in the Stack

We noticed that Algorithm DZ0 may sometimes push dead-zones onto the stack. This can be eliminated by adding the test $kdright < hi$ to line 12 before pushing. After this change, the stack contains only live-zones and popping of dead-zones in line 17 can be removed. These changes make the algorithm a bit faster on average. Let us call the modified version Algorithm DZ1. The pseudocode of DZ1 given below shows the changes to Algorithm DZ0.

3.2 Shifting with 2-Grams

Shifting in Algorithms DZ0 and DZ1 is based on single characters according to Horspool's shift [7]. We tried several alternatives for Horspool's shift. It would be possible to apply a different strategy to left shift than to right shift but we decided to use the same strategy to the both directions in order to reduce the number of alternatives.

[2] Usually this is literally a mirror image of the right shift table, Horspool in this case. That is not a requirement and other left shifters may be used.

Algorithm DZ1 (Changes to DZ0)

```
11    if lo < kdleft then
12       if kdright < hi then push(kdright, hi)
13       hi ← kdleft
14    else
15       lo ← kdright
16       if lo ≥ hi then
17
18          if top.second = max then return count
```

Sunday's shift [14] is a natural choice for the Dead-Zone algorithm, because it is applied after exiting the match loop testing an alignment window $t_{probe} \cdots t_{probe+m-1}$ in the text against the pattern. The test characters of shift $t_{probe-1}$ and $t_{probe+m}$ are outside the alignment window, whereas the test character is the first/last character of the alignment window in Horspool's shift. Thus the maximal shift of Sunday is $m+1$ to the both directions, i.e. one more than Horspool's maximal shift. On average, DZ1 with Sunday's shift runs slightly faster than DZ1.

Shifting based on 2-grams is a better choice for making the algorithm more efficient. The original lookup tables *shl* and *shr* of DZ1 are replaced by new two-dimensional lookup tables *shl2* and *shr2*, respectively. We tried three 2-gram shifters, in which the locations of the test 2-grams are different as well as the preprocessing of the lookup tables.

First we tried the Zhu–Takaoka shift [18]. The original method consists of two shift functions like the Boyer–Moore algorithm [2]. We applied only the 2-gram shift based on the occurrence heuristic (a.k.a. the bad character heuristic). We denote this shifter by ZT. The test 2-grams are the first and last 2-gram of the alignment window. The maximal shift of ZT is m.

Next we tested the Berry–Ravindran (BR) shift[3] [1] which is an extension of Sunday's shift. The tested 2-grams $t_{j-2}t_{j-1}$ and $t_{j+m}t_{j+m+1}$ ($j = probe$) are outside the alignment window. The maximal shift of BR is $m+2$.

The third 2-gram shifter is BRX introduced by Kalsi et al. [9]. BRX is an intermediate approach of ZT and BR. The test 2-grams are $t_{j-1}t_j$ and $t_{j+m-1}t_{j+m}$, $j = probe$. The maximal shift of BRX is $m+1$. Figure 1 shows the locations of the test 2-grams of ZT, BRX, and BR in an alignment of a pattern.

In our experiments (see Section 4), the three 2-gram shifters gave a significant speed-up over Algorithm DZ1. BR was the slowest. ZT and BRX were almost equally good, but BRX was the winner in case of short DNA patterns. We selected BRX for further development.

[3] Mauch [11] (and related publications) was the first to apply the BR shift to the Dead-Zone algorithm.

```
 P:     acctcg
 T: acccgtatgactta
ZT:     xx  xx
BRX:   xx    xx
BR:   xx      xx
```

Figure 1. Locations of test 2-grams of ZT, BRX, and BR in an alignment.

Let us consider detailed conditions for the shift tables of BRX. The test 2-qram $y = y_1y_2$ of the right-hand shift is $t_{probe+m-1}t_{probe+m}$. If y is present in P, then

$$shr2[y] = m - \max(i \mid y = p_ip_{i+1}) - 1$$

Otherwise $shr2[y]$ is $m + 1$ if $y_2 \neq p_0$ and m if $y_2 = p_0$. The left-hand shift is symmetrical to the right-hand shift. The test 2-qram $x = x_1x_2$ is $t_{probe-1}t_{probe}$. If x is present in P, then

$$shl2[x] = \min(i \mid x = p_ip_{i+1}) + 1$$

Otherwise $shl2[x]$ is $m+1$ if $x_1 \neq p_{m-1}$ and m if $x_1 = p_{m-1}$. Based on these conditions, it is straightforward to program the preprocessing of *shr2* and *shl2*.

We modified the BRX shifter further. Instead of two-dimensional shift tables, we implemented the handling of 2-grams as 16-bit entities. This version is called Algorithm DZ2. Notation $q(x, h)$ refers to a h-gram starting at x. At the implementation level $q(x, 2)$ is `*((uint16_t*)x)`. The pseudocode of DZ2 given below shows the changes to Algorithm DZ1.

Algorithm DZ2 (Changes to DZ1)

```
 9    kdleft ← probe − shl2[q(t_{probe−1}, 2)] + 1
10    kdright ← probe + shr2[q(t_{probe+m−1}, 2)]
```

3.3 Guard Test

Guard test [7,13] is a widely used technique to speed-up string matching. The idea is to test certain pattern positions before entering a match loop. Guard test is a representative of a general optimisation technique called loop peeling, where a number of iterations are moved in front of the loop. As a result, the computation becomes faster because of fewer loop tests.

We decided to try such a guard test where the first q-gram *try* of an alignment in the text is compared with *prefix*, the first q-gram of the pattern. As a result, the match loop is entered more seldom. We tried values $q = 2$ and 4. We decided to apply the latter, because it performed better. Let us call the modified version Algorithm DZ3. The pseudocode of DZ3 given below shows the changes to Algorithm DZ2.

Algorithm DZ3 (Changes to DZ2)

```
5b    try ← q(t_{probe}, 4)
5c    if try = prefix then
6       i ← 4
7       while i < m and p_i = t_{probe+i} do i ← i + 1
8       if i = m then count ← count + 1
```

Algorithm DZ3 in the present form does not work for patterns shorter than four characters. However, it is trivial to add separate code for them if necessary.

Beyond the Dead-Zone algorithm, the guard test with a q-gram might improve the performance of some other algorithms as well. Testing of q-grams as entities has been earlier used by Faro and Külekci [4] and Khan [10]. If wider q-grams than four characters are applied to long patterns, separate code is necessary for short patterns.

With old processors, there is a performance penalty for reading q-grams at misaligned memory locations, i.e. the q-gram does not start at a word boundary. This penalty decrease the benefit of processing q-grams as entities for shifting or guarding. However, for newer processor microarchitectures of Intel starting from Sandy Bridge and Nehalem, there is no such penalty [6].

4 Experiments

The experiments were run on Intel Core i7-4578U with 4 MB L3 cache and 16 GB RAM; this CPU has a Haswell microarchitecture which is subsequent to Sandy Bridge and therefore has none of the misaligned access performance penalties mentioned above. Algorithms were written in the C programming language and compiled with gcc 5.4.0 using the O3 optimisation level. Testing was done in the framework of Hume and Sunday [8]. We used two texts: English (four concatenated copies of the KJV Bible, totaling 16.2 MB) and DNA (four concatenated copies of the genome of E. Coli, totaling 18.6 MB) for testing. The base texts were taken from the SMART corpus. Because of the irregular scanning order, the Dead-Zone algorithms benefit from cache more than other algorithms. This was noticeable for texts shorter than 6 MB in our test setting, and therefore we decided to use longer texts. Sets of patterns of lengths 5, 10, and 20 were randomly taken from the both texts. Each set contains 200 patterns. The running times of 200 patterns in Table 1 are averages of 100 runs excluding the preprocessing time. For the 2-gram shifters, preprocessing took about 10 ms per 200 patterns. We used Horspool's algorithm (Hor) [7] and Sbndm4b [15] as reference methods. Hor is a representative of classical algorithms and Sbndm4b is an example of a fairly efficient algorithm. The code of Hor was taken from the SMART repository.

Table 1. Running times (in seconds) of algorithms.

	English, m			DNA, m		
Alg.	5	10	20	5	10	20
DZ0	5.25	3.64	2.77	13.43	10.74	10.27
DZ1	4.99	3.56	2.73	12.76	10.26	9.83
DZ1s	4.39	3.20	2.47	12.38	10.99	10.55
DZ1br	3.67	2.19	1.43	8.96	6.62	5.30
DZ1zt	3.19	1.97	1.33	9.24	5.26	3.63
DZ1brx	2.98	1.87	1.26	6.97	4.81	3.57
DZ2	2.71	1.72	1.16	6.47	4.50	3.38
DZ3	2.12	1.41	0.99	4.09	3.02	2.37
Hor	4.17	2.41	1.49	10.30	7.39	7.10
Sbndm4b	1.18	0.42	0.30	1.50	0.63	0.45

For Algorithm DZ1 using Horspool's shift we tried four alternative shifters:

1. DZ1s: Sunday [14]
2. DZ1br: Berry–Ravindran [1]
3. DZ1zt: Zhu–Takaoka [18] (occurrence shift)
4. DZ1brx: Kalsi et al. [9]

Algorithm DZ1s ran slightly faster than DZ1 for English data. Although the average shift of DZ1s is longer than that of DZ1, DZ1s was slower than DZ1 for DNA patterns of 10 and 20 characters. All the 2-gram approaches DZ1br, DZ1zt, and DZ1brx were significantly faster than the single character shifters DZ1 and DZ1s.

Algorithm DZ1br was the slowest of the 2-gram shifters. This is obvious for DNA because the average shift of DZ1br is shorter than that of DZ1zt for $m > 7$. The reason for the poor performance of DZ1br for English data is partly due to appearances of common characters like space as p_0 or p_{m-1} which decreases the average length of shift (see justification in [9]). DZ1brx was slightly better than DZ1zt for English data, but the former was a clear winner in the case of short DNA patterns.

Algorithm DZ3 was clearly faster than Hor but left noticeably behind Sbndm4b. Comparison of Dead-Zone algorithms and efficient left-to-right algorithms is somewhat unfair in sequential processing, because the former algorithms contain more bookkeeping and they do not benefit from locality as much as the latter ones.

Table 2. Speed-ups of algorithms (Alg. DZ0 is one).

	English, m			DNA, m			
Alg.	5	10	20	5	10	20	Avg.
DZ0	1.00	1.00	1.00	1.00	1.00	1.00	1.00
DZ1	1.05	1.02	1.01	1.05	1.05	1.04	1.04
DZ1s	1.20	1.14	1.12	1.08	0.98	0.97	1.08
DZ1br	1.43	1.66	1.93	1.50	1.62	1.94	1.68
DZ1zt	1.64	1.85	2.08	1.45	2.04	2.83	1.98
DZ1brx	1.76	1.95	2.20	1.93	2.23	2.88	2.16
DZ2	1.94	2.11	2.38	2.08	2.38	3.03	2.32
DZ3	2.48	2.59	2.79	3.28	3.56	4.34	3.17

Table 2 shows speed-ups of the developed Dead-Zone versions against Algorithm DZ0. Algorithm DZ3 clearly tripled the speed of the original algorithm DZ0 on average. The gain was larger for DNA than for English.

5 Concluding Remarks

Although the Dead-Zone algorithm is clearly oriented to parallel processing, we managed to substantially improve its performance in sequential processing. Only some of the optimisations are unique to the Dead-Zone algorithm, and the others could be used to benefit many of the other well-known algorithms when they are not already used. In particular, multi-gram shifting is an interesting tradeoff of memory (for shift tables) against performance, while the guard test optimisation could be applied in most Boyer-Moore style algorithms. It is somewhat surprising that the guard test optimisation is not automatically part of gcc's O3 level, as such optimisations are well known in the compiler literature. In short, despite continuous compiler and algorithmic improvement, there remain interesting opportunities for skilled programmers to manually tune implementations.

Some optimisations may be sensitive to misaligned memory accesses—though this was not the case on the benchmarking system used for this paper. This indicates that future work could include optimisations on other architectures where misalignment gives a performance penalty, or on alternative architectures such as the Arm. Additional future work includes using this paper's optimisations on multiple-keyword Dead-Zone.

References

1. T. Berry and S. Ravindran: *A fast string matching algorithm and experimental results*, in Proceedings of the Prague Stringology Club Workshop 1999, Prague, Czech Republic, July 8-9, 1999, 1999, pp. 16–28.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) 1977, pp. 762–772.
3. J. W. Daykin, R. Groult, Y. Guesnet, T. Lecroq, A. Lefebvre, M. Léonard, L. Mouchard, É. Prieur-Gaston, and B. W. Watson: *Three strategies for the dead-zone string matching algorithm*, in Prague Stringology Conference 2018, Prague, Czech Republic, August 27-28, 2018, J. Holub and J. Zdárek, eds., Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2018, pp. 117–128.
4. S. Faro and M. O. Külekci: *Fast packed string matching for short patterns*, in Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, New Orleans, Louisiana, USA, January 7, 2013, 2013, pp. 113–121.
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. A. Fog: *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, tech. rep., Technical University of Denmark, www.agner.org/optimize/microarchitecture.pdf, 2020.
7. R. N. Horspool: *Practical fast searching in strings.* Softw., Pract. Exper., 10(6) 1980, pp. 501–506.
8. A. Hume and D. Sunday: *Fast string searching.* Softw., Pract. Exper., 21(11) 1991, pp. 1221–1248.
9. P. Kalsi, H. Peltola, and J. Tarhio: *Comparison of exact string matching algorithms for biological sequences*, in Bioinformatics Research and Development, Second International Conference, BIRD 2008, Vienna, Austria, July 7-9, 2008, Proceedings, 2008, pp. 417–426.
10. M. A. Khan: *A transformation for optimizing string-matching algorithms for long patterns.* Comput. J., 59(12) 2016, pp. 1749–1759.
11. M. Mauch: *An Investigation of Dead-Zone Pattern Matching Algorithms*, Master's thesis, Stellenbosch University, South Africa, 2016.
12. 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.
13. T. Raita: *On guards and symbol dependencies in substring search.* Softw., Pract. Exper., 29(11) 1999, pp. 931–941.
14. D. Sunday: *A very fast substring search algorithm.* Commun. ACM, 33(8) 1990, pp. 132–142.
15. B. Ďurian, J. Holub, H. Peltola, and J. Tarhio: *Improving practical exact string matching.* Inf. Process. Lett., 110(4) 2010, pp. 148–152.
16. B. W. Watson, L. G. Cleophas, and D. G. Kourie: *Using correctness-by-construction to derive dead-zone algorithms*, in Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1-3, 2014, J. Holub and J. Zdárek, eds., Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2014, pp. 84–95.
17. 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.
18. R. F. Zhu and T. Takaoka: *A technique for two-dimensional pattern matching.* Commun. ACM, 32(9) 1989, pp. 1110–1120.

Author Index

Proceedings of the Prague Stringology Conference 2020
Edited by Jan Holub and Jan Žďárek
Published by: Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science
Prague Stringology Club
Thákurova 9, Praha 6, 160 00, Czech Republic.
First edition.

ISBN 978-80-01-06749-9

URL: http://www.stringology.org/
E-mail: psc@stringology.org Phone: +420-2-2435-9811

Printed by powerprint s.r.o.
Brandejsovo nám. 1219/1, Praha 6 Suchdol, 165 00, Czech Republic