Proceedings of the Prague Stringology Conference 2009

Edited by Jan Holub and Jan Žďárek

Department of Computer Science
and Engineering

August 2009

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

Proceedings of the Prague Stringology Conference 2009
Edited by Jan Holub and Jan Žďárek
Published by: Prague Stringology Club
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo náměstí 13, Praha 2, 121 35, Czech Republic.
URL: http://www.stringology.org/
E-mail: psc@stringology.org Phone: +420-2-2435-7470 Fax: +420-2-2492-3325
Printed by Česká technika–Naklatelství ČVUT, Thákurova 550/1, Praha 6, 160 41, Czech Republic

ISBN 978-80-01-04403-2

Conference Organization

Program Committee

Amihood Amir	(Bar-Ilan University, Israel)
Gabriela Andrejková	(P. J. Šafárik University, Slovakia)
Maxime Crochemore	(Université de Marne la Vallée, France)
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)
Shmuel T. Klein	(Bar-Ilan University, Israel)
Thierry Lecroq	(Université de Rouen, France)
Bořivoj Melichar, *Co-chair*	(Czech Technical University in Prague, Czech Republic)
Yoan J. Pinzon	(King's College London, United Kingdom)
Marie-France Sagot	(Université Claude Bernard, Lyon, France)
William F. Smyth	(McMaster University, Canada)
Bruce W. Watson	(Technische Universiteit Eindhoven, Netherlands)

Organizing Committee

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

External Referees

Saïd Abdeddaïm	Ondřej Guth	Elise Prieur
Pavlos Antoniou	Jan Janoušek	Petr Procházka
Miroslav Balík	Derrick Kourie	Giuseppina Rindone
Manolis Christodoulakis	Jan Lahoda	German Tischler
Loek Cleophas	Arnaud Lefebvre	Wilson Soto
Jacqueline Daykin	Juan Mendivelso	Tinus Strauss
Simone Faro	Spiros Michalakopoulos	Michal Ziv-Ukelson
Mathieu Giraud	Ernest Ketcha Ngassam	
Szymon Grabowski	Solon Pissis	

Preface

The proceedings in your hands contains the papers presented in the Prague Stringology Conference 2009 (PSC'09) which was held at the Department of Computer Science and Engineering of the Czech Technical University in Prague, Czech Republic, on August 31–September 2. The conference focused on stringology and related topics. Stringology is a discipline concerned with algorithmic processing of strings and sequences.

The papers submitted were reviewed by the program committee and twenty three were selected for presentation at the conference, based on originality and quality. This volume contains not only these selected papers but also abstract of one invited talk devoted to the combination of text compression and string matching.

The Prague Stringology Conference has a long tradition. PSC'09 is the fourteenth event of the Prague Stringology Club. In the years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conferences (PSC's) in 2001–2006, 2008 preceded this conference. The proceedings of these workshops and conferences had been published by the Czech Technical University in Prague and are available on WWW pages of the Prague Stringology Club. Selected contributions were published in special issues of journals the Kybernetika, the Nordic Journal of Computing, the Journal of Automata, Languages and Combinatorics, and the International Journal of Foundations of Computer Science. The series of stringology conferences was interrupted in 2007 when the members of the Prague Stringology Club were honoured to organize Conference on Implementation and Application of Automata 2007 (CIAA 2007).

The Prague Stringology Club was founded in 1996 as a research group at the Department of Computer Science and Engineering of 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 finite 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.

I would like to thank all those who had submitted papers for PSC'09 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'09. Last, but not least, my thanks go to the members of the organizing committee for ensuring such a smooth running of the conference.

In Prague, Czech Republic
on August 2009

Jan Holub

Table of Contents

Invited Talk

Contributed Talks

Combining Text Compression and String Matching: The Miracle of Self-Indexing

Gonzalo Navarro*

Department of Computer Science, University of Chile.
Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl

This decade has witnessed the raise of what I consider the most important breakthrough of modern times in text compression and indexed string matching. *Self-indexing* is the mechanism by which a text is simultaneously compressed and indexed, so that the self-index occupies space close to that of the compressed text, provides random access to any part of it, and in addition supports efficient indexed pattern matching. Thus a self-index can replace the text by a compressed version with enhanced search functionalities. Self-indexing builds on a large base of *compressed data structures*, which is another fascinating algorithmic area that has appeared two decades ago with the aim of obtaining compact representations of classical data structures. Although they usually require more instructions than their classical counterparts to operate, they can benefit from the memory hierarchy. This is particularly noticeable when they can operate in main memory in cases where the classical structures require disk storage.

My aim in this talk is to present a thin "vertical" slice of this construction, so that there is time to visualize in sufficent detail a complete solution from the basics to the final result. I will start with a plain and a compressed solution to provide *rank* on bitmaps, a simple operation of counting the number of 1s up to a given position, with a surprising number of applications. I will then introduce *wavelet trees*, which constitute a sort of self-index for sequences, supporting operation *rank* for the alphabet symbols. Then I will explain the Burrows-Wheeler Transform and the FM-index concept, which coupled with wavelet trees offer a fully-functional self-index. Finally, I will show how this simple combination is able of achieving high-order compression of a text, and will give some insights on recent work around indexing highly repetitive sequence collections, such as DNA and protein databases, versioned data, and temporal text databases. I will conclude by posing some open challenges.

* Partially funded by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

Feature Extraction for Image Pattern Matching with Cellular Automata

Lynette van Zijl and Leendert Botha

Department of Computer Science
Stellenbosch University
South Africa
lvzijl@sun.ac.za, lbotha@cs.sun.ac.za

Abstract. It is shown that cellular automata can be used for feature extraction of images in image pattern matching systems. The problem under consideration is an image pattern matching problem of a single image against a database of LEGO bricks. The use of cellular automata is illustrated, and solves this classical content-based image retrieval problem in near realtime, with minimal memory usage.

Keywords: cellular automata, pattern matching, content-based image retrieval

1 Introduction

The use of cellular automata (CA) in image processing and graphical applications has received some attention over the past few years (for example, [3,6,10]). In this paper, CA are applied to the pattern matching of images in a content-based image retrieval (CBIR) system.

CBIR systems generally require the recognition of semantically equivalent subimages from a library of given images. For example, given a library of photographs, a requirement may be the retrieval of all photographs containing (any kind of) flower. The methods for solving this general problem, however, can be improved if specific sub-domains of images are considered. In addition, solutions for specific problems can then often be generalized to improve the general CBIR methods [4].

In this work, the specific domain of images is that of LEGO bricks. This seemingly frivolous domain contains many mathematically interesting aspects. For example, the image matching must be a semantically exact pattern matching, but the images themselves can differ in rotation, scale and color (for example, see figure 1). Moreover, a useful software implementation demands a realtime solution, which means that computationally expensive mathematical solutions are not appropriate. This work shows that the extraction of the semantic definition of a LEGO brick from a given image (the so-called feature extraction phase of this problem) can be implemented with CA. The CA solution allows for a direct parallel implementation, and is also implementable directly on the GPU – this implies that the use of the CA allows almost instantaneous feature extraction.

Section 2 contains the necessary background and definitions. The use of the CA for feature extraction is described in section 3. The results are analysed in section 4, and the conclusion is given in section 5.

2 Background and definitions

The relevant terminology and definitions for CA and CBIR are briefly summarized in this section.

Lynette van Zijl, Leendert Botha: Feature Extraction for Image Pattern Matching with Cellular Automata, pp. 3–14.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

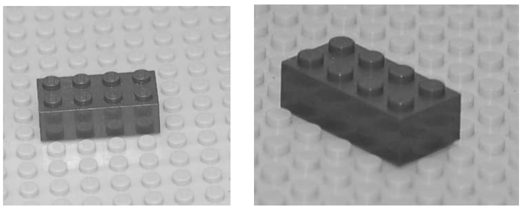


Figure 1. Two semantically equivalent LEGO bricks.

2.1 Cellular automata

We assume that the reader is familiar with CA as, for example, in [14]. We therefore summarize only the necessary definitions here.

A cellular automaton (CA) C is a k-dimensional array of automata. Each of the individual automata in the CA is said to occupy a *cell* in the CA. In the initial configuration of C, each automaton in C is in its initial state – this is typically refered to as time step $t = 0$. Each transition of C involves the *simultaneous* transitions of each of the individual automata. In addition, the individual automata are aware of the states of each of the other automata in the array, and the individual transitions may depend on the states of the other automata in the array. The global state of C thus evolves through time steps, where each time step describes the simultaneous changes in the individual automata.

In our case, CA are used to model images. Hence, only two-dimensional CA are considered, where each cell represents one pixel in the image plane. Furthermore, it is assumed that the individual automata in each cell are identical, and hence one transition function can be defined for the CA as a whole. Traditionally, each individual automaton is not dependent on all the other automata in the CA, but only on a subset of these automata. This subset is known as the *neighbourhood* of the automaton in the cell under consideration. We now formalize these intuitive concepts (see [8] for more detail):

Definition 1. *A 2D CA is a* 3*-tuple* $M = (A, N, f)$*, such that*

- A *is the finite nonempty state set,*
- $N = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ *is the neighbourhood vector consisting of vectors in* $\mathbb{Z}^2$*, and*
- $f : A^n \rightarrow A$ *is the transition rule.*

Given a configuration c of the cells in the CA at a certain time t, the configuration c' at time $t + 1$ for each cell $\boldsymbol{x}$ can be calculated as

$$c'(\boldsymbol{x}) = f(c(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)) .$$

In such a 2D CA, specific neighbourhoods can be defined. For example, the so-called Von Neumann neighbourhood for a cell $x_{i,j}$ is defined as $\langle x_{i-1,j}, x_{i,j-1}, x_{i,j+1}, x_{i+1,j} \rangle$.

2.2 Content-based image retrieval

Given an image pattern p, CBIR requires that p is compared to a set of images P to find a set of semantically equivalent matches Q. To define semantic equivalence, certain characteristics (features) of the images must correspond within given boundaries.

The set of images P is preprocessed off-line to obtain a so-called feature vector for each image, and this feature vector is stored with each image. The search pattern p requires (realtime) preprocessing to obtain its feature vector, and the matching process then becomes a comparison of the feature vector of p against all the feature vectors in P. A distance measure between feature vectors is used to return all images in P which are semantically closely related to the search image p.

The preprocessing of p in the case of the LEGO domain requires a number of steps:

– **Baseboard elimination**: Each p is assumed to be an image of a LEGO brick on a so-called baseboard, which is a flat LEGO surface with studs (see figure 1). It is assumed that the baseboard has a color contrasting with the color of the brick. The first step then is to eliminate the baseboard from p.
– **Edge detection**: The brick itself is identified in p by finding all the edges belonging to the brick.
– **Stud location**: The positions of the studs are located in p.
– **2D**: From the stud locations, the top surface of the brick is identified by finding the edges closest to the studs.
– **Geometry**: Given the stud locations, the arrangement of the studs in a geometric pattern defines the final semantics of the brick.

This work covers the preprocessing of the search image p, and it is shown how to accomplish this task by using CA.

3 Feature extraction with CA

The aim of the preprocessing phase is to construct a feature vector, and this process is described in detail in this section.

3.1 Background elimination

To be able to calculate an accurate feature vector for p, all the pixels that correspond to the background must be eliminated. As stated before, the background in this case always consists of a LEGO baseboard which has a color distinguishable from the color of the brick. As an initial step, the color of the pixels on the edge of the image is subtracted from all pixels which have approximately the same color.

In figure 2, after the initial color subtraction, the reader may note that the baseboard studs are not fully eliminated. This is due to the fact that the studs form shadows, which are not of the same color as the baseboard. To eliminate these shadows, a CA is used.

Let p_1 be the image obtained from p after the baseboard color subtraction. Define a Von Neumann-type neighbourhood $\boldsymbol{n}_x$ for each cell $x_{i,j}$, such that $\boldsymbol{n}_x = (\boldsymbol{x}_N, \boldsymbol{x}_S, \boldsymbol{x}_W, \boldsymbol{x}_E)$, where

$$\begin{aligned} \boldsymbol{x}_N &= \langle x_{i-\delta_L,j}, \ldots, x_{i-1,j} \rangle \\ \boldsymbol{x}_S &= \langle x_{i+\delta_L,j}, \ldots, x_{i+1,j} \rangle \\ \boldsymbol{x}_W &= \langle x_{i,j-\delta_L}, \ldots, x_{i,j-1} \rangle \\ \boldsymbol{x}_E &= \langle x_{i,j+\delta_L}, \ldots, x_{i,j+1} \rangle . \end{aligned}$$

That is, the Von Neumann neighbourhood is taken in the usual four directions up to a distance of δ_L from the current cell. Suppose that a background pixel in cell $x_{i,j}$ is indicated by $x_{i,j} = 0$. A CA C_L can now be defined, with transition function

$$c'(x_{i,j}) = \begin{cases} 0, \text{ if } \exists k_N, k_S, k_W, k_E : & x_{i-k_N,j} = 0 \ \& \\ & x_{i+k_S,j} = 0 \ \& \\ & x_{i,j-k_W} = 0 \ \& \\ & x_{i,j+k_E} = 0 \\ 1, \text{ otherwise,} & \end{cases}$$

where $1 \leq k_N, k_S, k_W, k_E \leq \delta_L$. That is, if there is any background pixel found within a distance of δ_L in all four directions from the current cell, then the cell is taken to be a background cell and will be eliminated. The number of iterations required to eliminate background pixels with this CA method is linearly dependent on the size of the original image p, and the relative size of the brick against the size of the background baseboard.

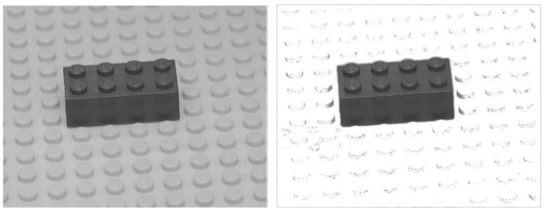


Figure 2. The original image on the left and the image after removing background pixels, based on their color. The small border around the picture indicates which pixels were used to determine the background color.

Some careful consideration will show that there is one instance where CA C_L will fail to remove all background pixels. This occurs under certain lighting conditions of p, when the background pixels form a straight line. In this scenario, there will be background pixels in only one or two directions from a given cell, and hence the line will not be removed. This is clear from the definition of the transition function of CA C_L above.

A second CA C_S can be constructed to eliminate the straight lines. This CA uses a smaller distance, δ_S, to look in all four directions. In contrast to C_L, a pixel is identified as background if *either* the horizontal or the vertical directions contain background pixels within the distance δ_S. Hence, let p_2 be the image obtained from p_1 after the background elimination described above. Again, define a Von Neumann-type neighbourhood $\boldsymbol{n}_x$ for each cell $x_{i,j}$, such that $\boldsymbol{n}_x = (\boldsymbol{x}_N^S, \boldsymbol{x}_S^S, \boldsymbol{x}_W^S, \boldsymbol{x}_E^S)$, where

$$\begin{aligned} \boldsymbol{x}_N^S &= \langle x_{i-\delta_S,j}, \ldots, x_{i-1,j} \rangle \\ \boldsymbol{x}_S^S &= \langle x_{i+\delta_S,j}, \ldots, x_{i+1,j} \rangle \\ \boldsymbol{x}_W^S &= \langle x_{i,j-\delta_S}, \ldots, x_{i,j-1} \rangle \\ \boldsymbol{x}_E^S &= \langle x_{i,j+\delta_S}, \ldots, x_{i,j+1} \rangle \, . \end{aligned}$$

That is, a Von Neumann neighbourhood in all four directions up to a distance of δ_S is used. Suppose that a background pixel in cell $x_{i,j}$ is indicated by $x_{i,j} = 0$. The transition function is then defined as

$$c'(x_{i,j}) = \begin{cases} 0, \text{ if } \exists k_N^S, k_S^S : x_{i-k_N^S,j} = 0 \ \& \ x_{i+k_S^S,j} = 0 \\ 0, \text{ if } \exists k_W^S, k_E^S : x_{i,j-k_W^S} = 0 \ \& \ x_{i,j+k_E^S} = 0 \\ 1, \text{ otherwise .} \end{cases}$$

Thus, a single subtraction of image pixels, followed by an application of CA C_1, followed by an application of CA C_2, yields the desired results, with all of the background pixels removed. The result is illustrated in figure 3.

Figure 3. The image after applying CAs C_L and C_S.

Once the background has been eliminated from the given image, one can proceed to find the edges of the brick itself.

3.2 Edge detection

The edge detection algorithm for the LEGO brick problem needs to isolate the inside and outside edges of the brick, as well as the pattern of studs on the top of the brick.

Our solution implements a CA-based method originally proposed by Popovici *et al* [10]. Let $\varphi(a, b)$ define a similarity measure between pixels a and b. The simplest example of such a similarity measure is the Euclidean distance in RGB-space [1], so that $\varphi(a, b) = \|a - b\|$. Hence, the value of $\varphi(a, b)$ decreases as the similarity between pixels a and b increases, so that $\varphi(a, a) = 0$.

Let ϵ be a specified lower threshold. Then define the CA C_e with the transition function as given below:

$$c'(x_{i,j}) = \begin{cases} 0, \text{ if} & \varphi(x_{i,j}, x_{i,j-1}) < \epsilon \;\&\; \varphi(x_{i,j}, x_{i,j+1}) < \epsilon \;\&\; \\ & \varphi(x_{i,j}, x_{i-1,j}) < \epsilon \;\&\; \varphi(x_{i,j}, x_{i+1,j}) < \epsilon \\ x_{i,j}, \text{ otherwise} . & \end{cases}$$

Again, the neighbourhood is clearly a Von Neumann neighbourhood, and in this case the distance is 1.

Sample output from C_e is shown in figure 4.

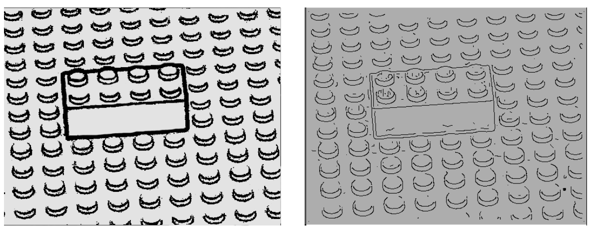


Figure 4. The edge detected images using cellular automaton C_e.

[1] Both Euclidean distance and vector angle were implemented as similarity measure, in both RGB space and YIQ space, in the software.

3.3 Feature extraction

The semantics of a LEGO brick is determined by its form, the number of studs and the arrangement of the studs[2]. For example, consider figure 5. Brick number 1 is a rectangular 2 by 4 brick. It has eight studs that are arranged in two rows of four studs each, in straight lines. There are also rounded bricks (brick number 3), macaroni bricks (brick number 4), and L-shaped bricks (brick number 6). Note that bricks number 2 and 3 have the same number of studs in the same arrangement, but their edges define the bricks to be semantically different. Also, brick number 2 and brick number 5 have the same number of studs (namely, four each), but in a different arrangement and hence these two bricks are also semantically different.

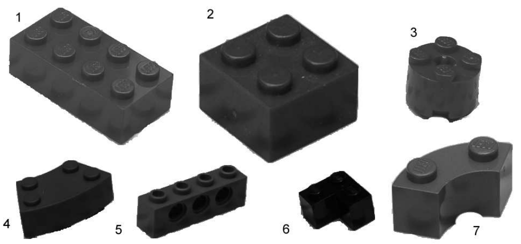


Figure 5. The semantic forms of LEGO bricks.

It is now necessary to find a feature vector that mathematically describes a LEGO brick, based on its form, number of studs, and stud arrangement. These characteristics are to be extracted and combined into a single feature vector for any given brick. These steps are discussed below.

Stud location The edges of a stud are difficult to find with standard shape-detection methods, as the edges have a distinctive halfmoon shape (see figure 6). A possible solution is to use template matching, where template shapes are moved around the image until a location is found which maximizes some match function. A popular match function is the squared error [13]:

$$\mathrm{SE}(x,y) = \sum_{\alpha=1}^{N} \sum_{\beta=1}^{N} \left(f(x-\alpha, y-\beta) - T(\alpha,\beta)\right)^2$$

where f is the image and T is the $N \times N$ template.

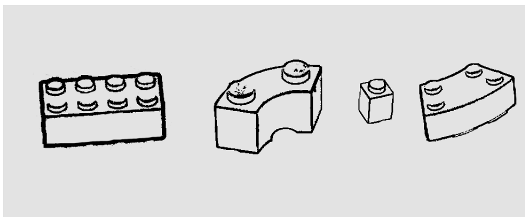


Figure 6. Four edge detected bricks showing similarity in the shape of the studs.

[2] In some user-defined cases, the color of the brick may also be used as an additional semantic feature.

The template used for the studs of the LEGO bricks is shown in figure 7. Note that templates of different size are provided, as scaling is not accurate in this specific case. The score of the matching function is scaled relative to the size of the template to prevent larger templates from getting higher scores.

Figure 7. The set of templates used in the template matching process.

Given the stud locations, the next step is to determine the stud formation.

Stud formation After the previous step, the center points of the locations of all the studs are available. The next step is to define the formation in which the studs occur. In other words, the number of rows and columns of the studs have to be extracted. For example, a brick with eight studs may have two rows of four studs each, or one row of eight studs.

Hence, it is necessary to find the minimal set of straight lines $\mathcal{L} = \{l_1, l_2, \ldots, l_n\}$ where each stud lies on exactly one l_i. Note that the stud location is point-specific, so that a point is deemed to lie on a line if it is within a given perpendicular small distance from the line.

Our algorithm is given below (see Algorithm 1), and is described in more detail in [1]. The output of Algorithm 1 gives the number of studs, and the number of lines needed to cover those studs, and these are used directly in the final feature vector.

Next, the form of the brick must be determined. A LEGO brick has a three-dimensional form, defined by both the inside and outside edges of the brick. The standard three-dimensional matching algorithms available in the literature would have been too computationally expensive in this case [13]. We therefore simplified the problem to a two dimensional problem, by the observation that all LEGO bricks are rectangular protrusions of the top surface of the brick [3]. Hence, it is only necessary to identify the top surface.

Identifying the top surface The top surface of the brick is identified by finding the edges surrounding the stud locations found in the previous step. This is a three step process: first, the edges of the studs themselves are removed by subtraction. This leaves random noise on the top surface, which is removed with a CA similar to the CA used to eliminate the background. Lastly, a CA is defined to flood in all directions, from the stud locations to the nearest edge.

The first step (removing the stud edges) is simply done by subtracting the matching template shape. This results in random noise, as the templates are not a perfect pixel-by-pixel match. The CA C_1 as defined previously, is used to remove this noise.

The flooding process to find the edges on the top surface of the brick, is again easily defined with a CA C_f. Initialize C_f so that there are four possible states in each cell: *background*, *edge*, *top surface* and *not top surface*. All the pixels where there were studs, are identified as top surface, and any cell that is not background, edge or top surface, is identified as *not top surface*.

[3] We only consider 'standard' LEGO bricks in this work. Other forms (such as sloped bricks, or bricks with a base larger than the top, such as cones) will be considered in subsequent work.

Algorithm 1 Determining the formation of the studs

```
procedure GET_FORMATION(Set of studs 𝒮 )
    ℒ ← ∅                                            ▷ Initialize the set holding the lines
    for i ← 1 to size(𝒮) do                          ▷ Add all possible lines
        for j ← i + 1 to size(𝒮) do
            ℒ ← ℒ + new Line{𝒮.get(i), 𝒮.get(j)}
        end for
    end for

    for each line l in ℒ do                           ▷ Determine how many studs covered by each line
        for each stud s in 𝒮 do
            if distance(l, s) < δ then                ▷ If s lies very close to l
                l.coveredStuds.add(s)                 ▷ Add s to set covered by l
            end if
        end for
    end for

    count ← 0                                         ▷ The cardinality of the covering set
    while size(𝒮) > 0 do
        l ← ℒ.removeMax()                             ▷ Remove line that covers most studs
        𝒮 ← 𝒮 − l.coveredStuds
        count+ = 1
    end while

    return count, size(𝒮)                            ▷ The formation is count by size(𝒮)/count
end procedure
```

The neighbourhood to be used in C_f is a Moore neighbourhood [4], with a specified distance ns. The transition function then considers each cell. If it is not a top surface cell, then the cell changes into a top surface cell if it is adjacent to any top surface cell in the Moore neighbour and it is neither edge nor background. That is, from the stud locations, the neighbours of each cell are considered. Count the number of neighbours that are not edge or background. If this number exceeds a given threshold, then the current cell is top surface. Formally, let th be the threshold and ns the neighbourhood size. Let a top surface cell be represented by 0, edges by 1, and background by 2. Then C_f has the transition function

$$c'(x_{i,j}) = \begin{cases} 0, \text{ if} & c(x_{i,j}) \neq 0 \;\&\; c(x_{i,j}) \neq 2 \;\&\; \\ & (\Sigma_{m,n} x_{m,n} = 0) > th, \\ & \text{ where } i - ns/2, j - ns/2 < m, n < i + ns/2, j + ns/2, \\ 1, \text{ otherwise} . & \end{cases}$$

An example of the stud edge, noise removal and flooding is shown in figure 8.

At this point, if color is to be used as a distinguising feature, a standard 256-bin histogram [11] is used to construct the color information for the brick.

It is now finally possible to encode the feature vector of a given LEGO brick, based on its number of studs, stud locations, form, and color. In our case, we used the Hu-set of invariant moments [7] to encode the feature vector with the information extracted from the image p.

[4] A Moore neighbourhood consists of all eight cells surrounding the current cell.

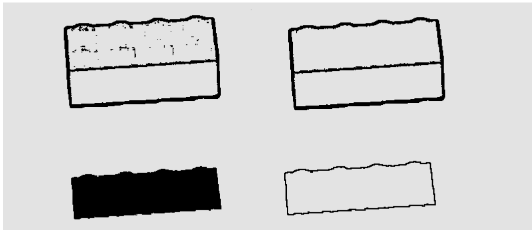


Figure 8. The output of the flooding process. The top left picture shows the edges after the studs have been removed. A CA is applied to remove the noise and the output is shown in the top right picture. The flooded region is shown on the bottom left and the boundary of this region, which is to be encoded into a feature vector, is shown on the bottom right.

Once the feature vector has been calculated for the search image p, that vector can be matched against all the pre-calculated images in the database. Our software can handle multiple search criteria on any of the elements in the feature vector, and hold a match score so that a set of best possible matches can be retrieved.

4 Analysis

This section illustrates some of the results in the final implemented CBIR system. More details, and comparative results with more traditional approaches, are discussed in [1]. In our initial experiments, bricks were correctly identified in almost 80% of our test cases.

An example of the shape-based retrieval is illustrated in figure 9. The bricks are shown in best match order (the lower the number, the better the match). In figure 9, note that an identical brick to the search image p was the best match, followed by two other curved bricks, while the rectangular bricks were the worst matches. Note that any shape is described by the Hu-set of invariant moments. In comparing two shapes, the Euclidean distance between the two shapes is calculated – the smaller the distance, the better the match. In figure 9 below, it therefore follows that the macaroni-shaped bricks are nearer to each other than to rectangular bricks.

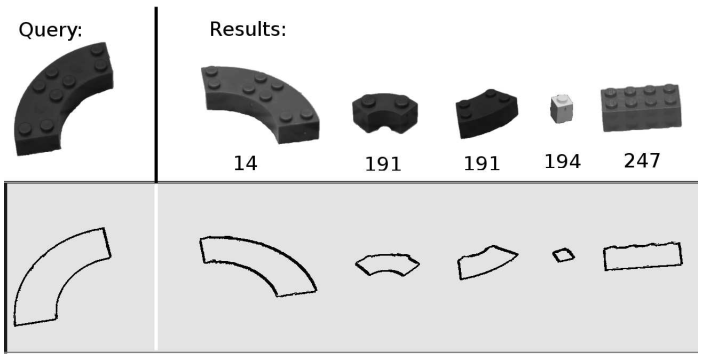


Figure 9. A sample shape retrieval query with match scores presented in thousands.

If an image is not of sufficient quality, the edge detection can result in discontinuous edges. This invalidates the flooding process. Recall that the flooding process terminates when an edge is encountered. Figure 10 shows an example of a brick for which the flooding process fails. Here, note that the brick does not have a continuous edge separating the top surface from the rest of the brick. Hence, the entire brick is flooded, resulting in an inaccurate feature vector.

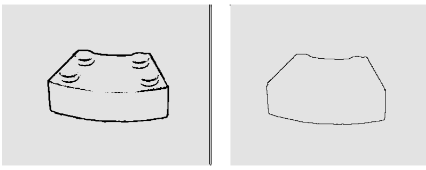


Figure 10. Shape extraction process fails for a brick that does not have a continuous edge separating the top surface from the rest of the brick.

To solve the problem, one can simply adjust the threshold of the edge detector CA (this is a parameter which can be set by the user in our software). As long as the image p is of sufficient quality, the increased threshold will always result in a continuous edge. Figure 11 illustrates a changed threshold and consequent successful edge detection.

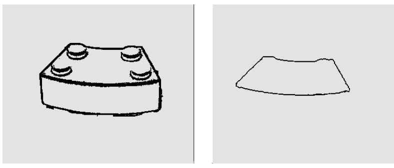


Figure 11. The same brick from figure 10, using a better threshold, and resulting in a correct identification of the top surface.

Almost all mismatches are due to an input image p where the edge detection fails. Failed or incorrect edge detection are due primarily either to a threshold that is not high enough for the picture quality (see below), or to distracting features which result in an incorrect edge detection. Figure 12 shows some examples of bricks that will not be correctly identified. The brick on the left is the same colour as the background, and hence is eliminated during the background elimination phase. The brick on the right results in false edges, due to the vertical stripes. This leads to a feature vector with an incorrect shape description for a 1×2 brick.

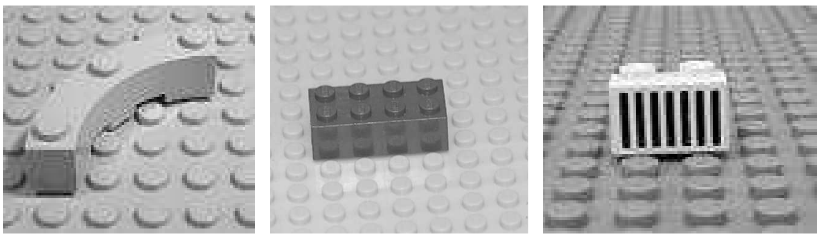


Figure 12. Bricks that cannot be recognized, due to background colour (left), lighting conditions (middle), and distracting features (right).

4.1 Comparison with existing systems

General content-based image retrieval systems cannot be directly compared to our system. Most CBIR system (such as the FIRE search engine [5]) classify images into broad groups and find matching topics. For example, given an image of a LEGO brick, the FIRE engine would return a wide variety of images of toys, with at best a few LEGO bricks.

We can compare at least one part of the image pre-processing with other algorithms, namely, the edge detection. There are many different edge detection algorithms and systems. In general, the more accurate the edge detection, the longer the algorithm takes to execute. For example, the well-known Canny [2] and SUSAN [12] edge detectors are extremely accurate, but too slow for real-time analysis. Other less accurate methods have other issues that make their use difficult in this domain. For example, the Marr-Hildreth algorithm [9] lacks in the localization of curved edges, which is essential in the LEGO images. It is also interesting to note that the more accurate edge detectors result in thin edges (see figure 13), while the rest of our algorithms such as flooding and template matching, work best with thick edges.

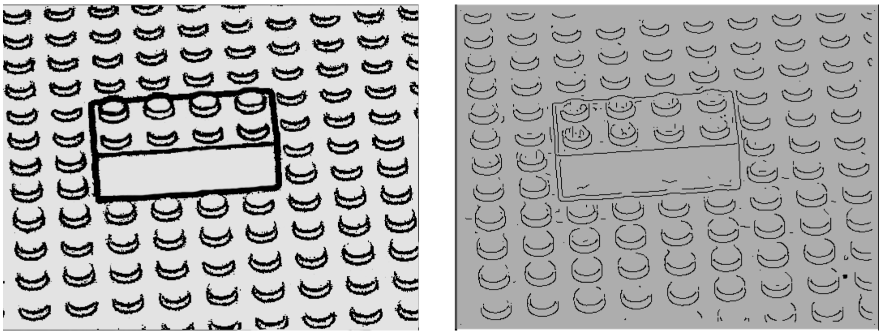


Figure 13. Results from he CA edge detector (left) versus the Canny edge detector (right).

5 Conclusion

We showed that cellular automata can be successfully applied for the realtime retrieval of LEGO brick images. The advantage of this approach is the limited memory use and fast execution time of a CA implementation.

We showed that it is possible to simplify the three-dimensional shape extraction problem to a two-dimensional case for the LEGO brick. We implemented a fully functional CBIR system based on the CA feature extraction, and illustrated the results.

For future work, we intend to extend this work to more general LEGO bricks. In particular, we want to consider those cases that are not simply protrusions of a brick with rectangular stud formations.

References

1. L. Botha: *A CBIR system for LEGO brick image retrieval*, tech. rep., Stellenbosch University, 2008.
2. J. Canny: *A computational approach to edge detection.* IEEE Trans. Pattern Anal. Mach. Intell., 8(6) 1986, pp. 679–698.
3. C. Chan, Y. Zhang, and Y. Gdong: *Cellular automata for edge detection of images*, in Proceedings of the Third International Conference on Machine Learning and Cybernetics, August 2004, pp. 3830–3834.

4. R. Datta, D. Joshi, J. Li, and J. Wang: *Image retrieval: ideas, influences, and trends of the new age*. ACM Computing Surveys, 40(2) April 2008.
5. T. Deselaers, D. Keysers, and H. Ney: *Features for image retrieval – a quantitative comparison*, in In DAGM 2004, Pattern Recognition, 26th DAGM Symposium, number 3175 in LNCS, 2004, pp. 228–236.
6. S. Druon, A. Crosnier, and L. Brigandat: *Efficient cellular automata for 2D/3D free-form modeling*. Journal of Winter School of Computer Graphics, 11(1) 2003, pp. 102–108.
7. K. Hu: *Visual pattern recognition by moment invariants*. IRE Transactions on Information Theory, IT-8 February 1962, pp. 179–187.
8. V. Lukkarila: *On undecidability of sensitivity of reversible cellular automata*, in AUTOMATA 2008, 2008, pp. 100–104.
9. D. Marr and E. Hildreth: *Theory of edge detection*. Proceedings of the Royal Society of London. Series B, Biological Sciences, 207(1167) 1980, pp. 187–217.
10. A. Popovici and D. Popovici: *Cellular automata in image processing*, in Proceedings of the 15th International Symposium on Mathematical Theory of Networks and Systems, University of Notre Dame, 2002.
11. S. Siggelkow: *Feature Histograms for Content-Based Image Retrieval*, PhD thesis, Albert-Ludwigs-Universität Freiburg, Fakultät für Angewandte Wissenschaften, Germany, Dec. 2002.
12. S. Smith and J. Brady: *Susan - a new approach to low level image processing*. International Journal of Computer Vision, 23 1997, pp. 45–78.
13. W. Snyder and H. Qi: *Machine Vision*, Cambridge University Press, New York, NY, USA, 2003.
14. S. Wolfram: *Cellular Automata and Complexity*, Westview Press, 1994.

On-line construction of a small automaton for a finite set of words

Maxime Crochemore[1] and Laura Giambruno[2]

[1] King's College London, London WC2R 2LS, UK, and Université Paris-Est, France
[2] Dipartimento di Matematica e Applicazioni, Universitàdi Palermo, Palermo, Italy
Maxime.Crochemore@kcl.ac.uk, lgiambr@math.unipa.it

Abstract. In this paper we describe a "light" algorithm for the on-line construction of a small automaton recognising a finite set of words. The algorithm runs in linear time. We carried out good experimental results on the suffixes of a text, showing how this automaton is small. For the suffixes of a text, we propose a modified construction that leads to an even smaller automaton.

1 Introduction

The aim of this paper is to design a "light" algorithm that builds a small automaton accepting a finite set of words and that works on-line in linear time. The study of algorithms for the construction of automata recognising finite languages is interesting for parsing natural text and for motif detection (see [4]). It is used also in many software like the intensively used BLAST [2]. In particular it is important to study algorithms with good time and space complexities since the dictionaries used for natural languages can contain a large number of words.

It is in general easy to construct an automaton recognising a given list of words. Initially the list can be represented by a trie (see [6]) and then, using an algorithm for tree minimisation (see [1], [9]), we can minimise the trie to get the minimal automaton of the finite set of words of the list. But this solution requires a large memory space to store the temporary large data structure.

Another solution was drafted by Revuz in his thesis ([11]) where he proposed a pseudo-minimisation algorithm that builds from set of words in lexicographic inverse order an automaton smaller than the trie, but that is not necessarily minimal. Anyway the solution is not completely experimentally tested and remains unpublished.

Other solutions were proposed recently by several authors (cf. [15], [13], [14], chapter 2 of [5], [10], [8], [7]). For instance Watson in [15] presented a semi-incremental algorithm for constructing minimal acyclic deterministic automata and Sgarbas et al. in [10] proposed an efficient algorithm to insert a word in a minimal acyclic deterministic automata in order to obtain yet a minimal automaton, but not so efficient on building the automaton for a set of words. In [8] Daciuk et al. also proposed an algorithm that constructs a minimal automaton for an ordered set of strings, by adding new strings one by one and minimizing the resulting automaton.

Here we propose an intermediate solution, similar to that one of Revuz, that is to build a rather small automaton with a light algorithm processing the list of words on-line in linear time on the length of the list, where the length of a list is the sum of the lengths of the elements in the list. The aim is not to get the corresponding minimal automaton but just a small enough structure. However, the minimal automaton can be later obtained with Revuz' algorithm [12] that works in linear time on the size of the acyclic automaton.

Maxime Crochemore, Laura Giambruno: On-line construction of a small automaton for a finite set of words, pp. 15–28.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

The algorithm works on lists satisfying the following condition: words are in right-to-left lexicographic order. Such hypothesis on the list is not limitative since list update is standard. Moreover with the light algorithm the automaton can possibly be built on demand and our solution avoids building a temporary large trie.

The advantages of our algorithm are simplicity, on-line construction and the fact that resulting automaton seems to be really close to minimal.

In particular, in this paper we show the results of experiments done on the list of suffixes of a text. For each set we consider the ratio between the number of states of the constructed automaton and that of the minimal automaton associated with the set. Such ratios happen to be fairly small. For the suffixes of a text we even propose a modified construction that results in an almost minimal automaton.

In Section 2, after some standard definitions, we define the iterative construction of the automaton for a list of words. In Section 3 we describe the on-line algorithm that builds the automaton and that works in linear time on the length of the list. We bring some examples of the non minimality of this automaton. In Section 4 we deal with sets that are suffixes of a given word. We carry out the experimental results and we show the modified construction. Conclusions are in Section 5.

2 The algorithm for a finite set of words

For definitions on automata we refer to [3] and to [9].

Let A be a finite alphabet. Let x in A^*, then we denote by $|x|$ the length of x, by $x[j]$ for $0 \leq j < |x|$ the letter of index j in x and by $x[j\,..\,k] = x[j] \cdots x[k]$. For any finite set X of words we will denote by $|X|$ the cardinality of X. Let u be a word in A^*, we denote by $S(u)$ the set of the proper suffixes of u together with u.

A *deterministic automaton* over A, $\mathcal{A} = (Q, i, T, \delta)$ consists of a finite set Q of *states*, of the initial state i, of a subset $T \subseteq Q$ of *final* states and of a *transition function* $\delta : Q \times A \longrightarrow Q$. For each p, q in Q, a in A such that $\delta(p, a) = q$, we call $(p, a, \delta(p, a))$ an *edge* of $\mathcal{A}$. An edge $e = (p, a, q)$ is also denoted $p \xrightarrow{a} q$. A *path* is a sequence of consecutive edges. A path is successful if its ending state is a final state. Given an automaton $\mathcal{A}$, we denote by $L(\mathcal{A})$ the language recognised by $\mathcal{A}$.

Let $X = (x_0, \ldots, x_m)$ be a list of words in A^* such that the list obtained reversing each word in X is sorted according to the lexicographic order. We will build an automaton recognising X with an algorithm that processes the list of words on-line. In order to do this we define inductively a sequence of $m + 1$ automata $\mathcal{A}_X^0, \ldots, \mathcal{A}_X^m$, such that, for each k, the automaton $\mathcal{A}_X^k$ recognises the language $\{x_0, \ldots, x_k\}$. In particular $\mathcal{A}_X^m$ will recognises X.

In the following we define $\mathcal{A}_X^0$ and then, for each $k \in \{1, \ldots, m\}$, we define the automaton $\mathcal{A}_X^k$ from the automaton $\mathcal{A}_X^{k-1}$. In these automata we will define a unique final state without any outgoing transition that we call q_{fin}. For each k, we consider the following functions over the set of states of $\mathcal{A}_X^k$ with values in N defined, for each state j in $\mathcal{A}_X^k$, as:

- Height: $H(j)$ is the maximal length of paths from j to a final state.
- Indegree: $Deg^-(j)$ is the number of edges ending at j.
- Paths toward final states: for $j \neq q_{fin}$, $PF(j)$ is the number of paths starting at j and ending at final states and $PF(q_{fin}) = 1$.

2.1 Definition of $\mathcal{A}_X^0$

Let $\mathcal{A}_X^0 = (Q_0, i_0, T_0, \delta_0)$ be the deterministic automaton having as states $Q_0 = \{0, \dots, |x_0|\}$, initial state $i_0 = 0$, final state $T_0 = \{|x_0|\}$ and transitions defined, for each i in $Q_0 \setminus \{|x_0|\}$, by $\delta_0(i, x_0[i]) = i+1$. We will denote by q_{fin} the final state $|x_0|$. For each k in $\{0, \dots, m\}$, q_{fin} will be the unique final state in $\mathcal{A}_X^k$ with no edge going out from it. It is easy to prove that $L(\mathcal{A}_X^0) = \{x_0\}$. In Figure 1 we can see $\mathcal{A}_X^0$ for $X = (\texttt{aaa}, \texttt{ba}, \texttt{aab})$.

2.2 Definition of $\mathcal{A}_X^k$ from $\mathcal{A}_X^{k-1}$

Assume $\mathcal{A}_X^{k-1} = (Q_{k-1}, i_{k-1}, T_{k-1}, \delta_{k-1})$ has been built and let us define $\mathcal{A}_X^k = (Q_k, i_k, T_k, \delta_k)$. We define $i_k = \{0\}$.

Let u be the longest prefix in common between x_k and the elements $\{x_0, \dots, x_{k-1}\}$. Let s be the longest suffix in common between x_k and x_{k-1}. If $|s| \geq |x_k| - |u|$ then we redefine s as $x_k[|u|+1 \,..\, |x_k|-1]$. Let us consider p the end state of the path c in $\mathcal{A}_X^{k-1}$ starting at 0 with label u. Let q be the state along the path from 0 with label x_{k-1} for which the sub-path from q to q_{fin} has label s.

Indegree-Control. The general idea of the construction of $\mathcal{A}_X^k$ from $\mathcal{A}_X^{k-1}$ would be to add a path from p to q. See Figure 1. Anyway in general we cannot do this since we would add others words other than x_k, as we can see in Figure 2. This depends on the fact that on the path c there are states r with $Deg^-(r) > 1$. Thus, before adding a path from p to q, we have to do a transformation of the automaton like in Figure 3.

Figure 1. The automata $\mathcal{A}_X^0$ (left) and $\mathcal{A}_X^1$ (right) for $X = (\texttt{aaa}, \texttt{ba}, \texttt{aab})$. Since u, the prefix in common between aaa and ba, is the empty word and since s, the suffix in common between aaa and ba is a, the automaton $\mathcal{A}_X^1$ is obtained from $\mathcal{A}_X^0$ by adding the edge $(0, \texttt{b}, 2)$.

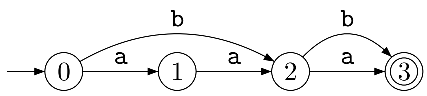


Figure 2. Incorrect construction of $\mathcal{A}_X^2$ for $X = (\texttt{aaa}, \texttt{ba}, \texttt{aab})$: in this case $u = \texttt{aa}$ and $s = \varepsilon$, but, since $Deg^-(2) > 1$, adding the edge $(2, \texttt{b}, 3)$ leads to an automaton accepting $\{\texttt{aaa}, \texttt{ba}, \texttt{aab}, \texttt{bb}\}$.

More formally we consider separately the case in which there is a state on c with indegree greater than 1 and the other case.

I CASE: In c there is a state with indegree greater than 1.

Let us call r the first state with $Deg^-(r) > 1$. Let us decompose the path c as $c : 0 \xrightarrow{u_0} r_0 \xrightarrow{x[\ell]} r \xrightarrow{u_1} p$. We construct the automaton $\mathcal{B}_X^{k-1} = (Q'_{k-1}, 0, T'_{k-1}, \delta'_{k-1})$ in the following way. In order to construct δ'_{k-1}:

- we delete the edge $r_0 \xrightarrow{x[\ell]} r$,

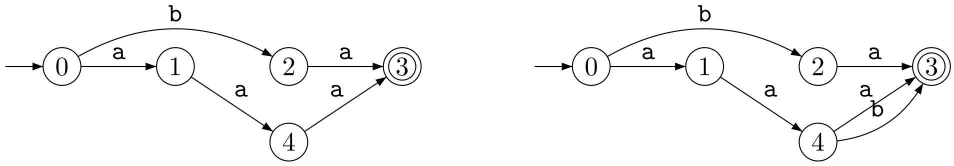


Figure 3. Automata $\mathcal{B}_X^1$ (left) and $\mathcal{A}_X^2$ (right) for $X = (\texttt{aaa},\texttt{ba},\texttt{aab})$. The automaton $\mathcal{B}_X^1$ is equivalent to $\mathcal{A}_X^1$ and it is obtained from $\mathcal{A}_X^1$ by doing a copy of the path from 0 to 4 with label aa. The automaton $\mathcal{A}_X^2$ is obtained by adding the edge $(4, \texttt{b}, 3)$

– we construct a path from r_0 with label $x[\ell]u_1$, let us call p' its ending state,
– we create for each edge going out from p with label a and ending at a state t, $p \xrightarrow{a} t$, the edge from p', $p' \xrightarrow{a} t$.

More formally we define $Q'_{k-1} = Q_{k-1} \cup \{|Q_{k-1}|, \ldots, |Q_{k-1}| + |u_1|\}$ and

$$\begin{cases} \delta'_{k-1}(i,a) = \delta_{k-1}(i,a), & \forall i \neq r_0, \forall a \in A; \\ \delta'_{k-1}(r_0, x_k[\ell]) = |Q_{k-1}|, & \\ \delta'_{k-1}(|Q_{k-1}| + i, x_k[\ell+i]) = |Q_{k-1}| + i + 1, & \forall i = 0, \ldots |u_1| - 1; \\ \delta'_{k-1}(|Q_{k-1}| + |u_1|, a) = \delta_{k-1}(p,a), & \forall a \in A. \end{cases}$$

We denote by p the state $|Q_{k-1}| + |u_1|$.

II CASE: the other case. We consider $\mathcal{B}_X^{k-1} = \mathcal{A}_X^{k-1}$.

We consider now the automaton $\mathcal{B}_X^{k-1}$. If x_k is the prefix of a word in $\{x_0, \ldots, x_{k-1}\}$ then we add p to the final states of $\mathcal{B}_X^{k-1}$, that is $T'_{k-1} = T_{k-1} \cup \{p\}$ and we define $\mathcal{A}_X^k = \mathcal{B}_X^{k-1}$.

Otherwise we proceed with the following control. We have the decomposition of x_k as $x_k = uws$, with $w \in A^+$.

Paths toward final states control. As before, the general idea is to add a path from p to q with label w, but there are some other controls that are required. In Figure 4 we see another situation in which we cannot add a path from p to q otherwise we would add words not in X. In this case, it depends on the fact that the number of paths from q towards final states is greater than one, that is $PF[q] > 1$.

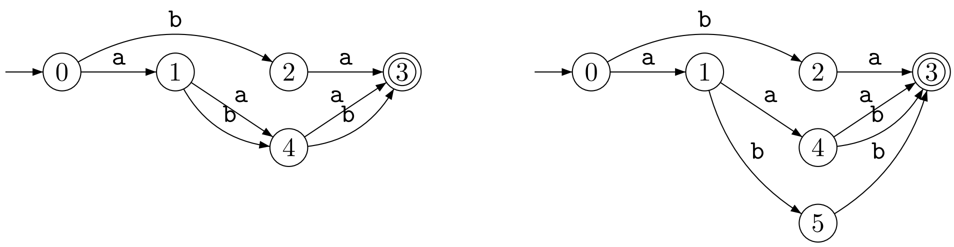


Figure 4. Incorrect construction of $\mathcal{A}_X^3$ (left) for $X = (\texttt{aaa},\texttt{ba},\texttt{aab},\texttt{abb})$: adding the edge $(1, \texttt{b}, 4)$ to $\mathcal{A}_X^2$ leads to an automaton accepting $\{\texttt{aaa},\texttt{ba},\texttt{aab},\texttt{abb},\texttt{aba}\}$. Right construction of $\mathcal{A}_X^3$ (right): it is obtained by adding the path from 1 to 3 with label bb. In particular 3 is the first state q' in the path from 4 to 3 with $PF[q'] = 1$

Thus, if $PF(q) > 1$ then we consider in the path d from q to q_{fin} with label s the first state q' such that $PF(q') = 1$, if it exists. In this case we call s_1 the label of the subpath of d from q to q' and let $s = s_1 s_2$. We call w the word ws_1, s the word s_2 and q the state q'. See Figure 4 for an example.

If there is no q' with $PF[q'] = 1$ in the path from q to q_{fin} with label s, then we define q as q_{fin} and w as ws. Otherwise we proceed with the Height control.

Height-Control. If $H(p) \leq H(q)$ then in general we cannot add a path from p to q because if there is a path from q to p then we will have infinitely many words recognised, as we can see in the example in Figure 5. We have to do another transformation as in Figure 6.

If $H(p) \leq H(q)$ then we consider in the path d, from q to q_{fin} with label s, the first state q' such that $H(p) > H(q')$. We call s_1 the label of the subpath of d from q to q'. Let $s = s_1 s_2$. We call w the word ws_1, s the word s_2 and q the state q'. In Figure 6 we have an example of the construction.

If $H(p) > H(q)$ then we go further.

Figure 5. We have the automaton $\mathcal{A}_X^0$ (left) for $X = (\texttt{aba},\texttt{abbba})$ and the incorrect construction of $\mathcal{A}_X^1$ (right): adding the edge $(2, \texttt{b}, 1)$ would lead to an automaton accepting the infinite language $\{\texttt{aba},\texttt{a(bb)}^*\texttt{a}\}$.

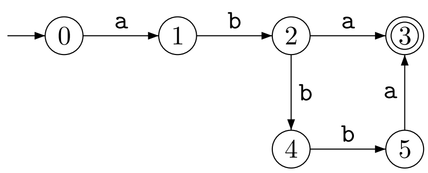


Figure 6. We have the right construction of $\mathcal{A}_X^1$ for $X = (\texttt{aba},\texttt{abbba})$.

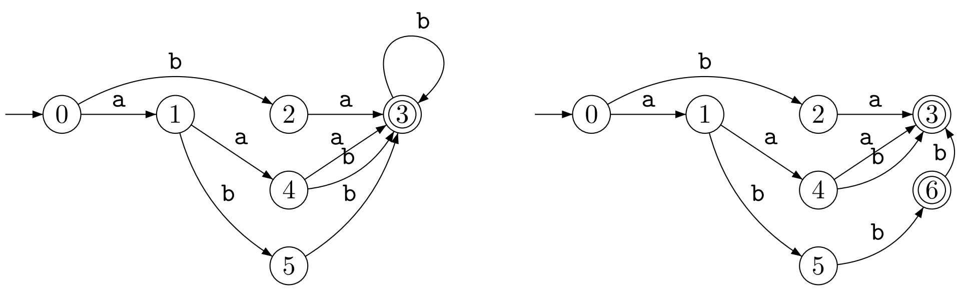


Figure 7. Incorrect construction of $\mathcal{A}_X^4$ (left) and the right construction of $\mathcal{A}_X^4$ (right) for $X = (\texttt{aaa},\texttt{ba},\texttt{aab},\texttt{abb},\texttt{abbb})$.

If there exists a word in $\{x_0, \ldots, x_{k-1}\}$ that is a prefix of x_k then, if $p \neq q_{fin}$ we add p to the set of final states, that is $T_k = T'_{k-1} \cup \{p\}$.

If $p = q_{fin}$ then, if we add a path from p to q_{fin} with label w then we would add also infinitely many words to the language recognised by the automaton, as in the example in Figure 7. In Figure 7 it is also reported the right construction of the automaton, as explained in the following.

When $p = q_{fin}$ we consider the following decomposition of c, the path from 0 with label u, $c : 0 \xrightarrow{u'} p' \xrightarrow{a} q_{fin}$. We delete the edge $p' \xrightarrow{a} q_{fin}$. Then we add an edge from p' to a new state p'' with label a and we add p'' to the set of final states. We call p the state p''. More formally we define $Q_k = Q'_{k-1} \cup \{|Q'_{k-1}|\}$ and

$$\begin{cases} \delta_k(i,a) = \delta'_{k-1}(i,a), \quad \forall i \neq p', \forall a \in A; \\ \delta_k(p',a) = |Q'_{k-1}|, \\ \delta_k(p',a) = \delta'_{k-1}(p',a), \forall b \in A, b \neq a; \end{cases}$$

We call p the state $|Q'_{k-1}|$.

Finally in all cases we add a path from p to q with label w, that is $Q_k = Q'_{k-1} \cup \{|Q'_{k-1}|+1, \ldots, |Q'_{k-1}|+|w|-3\}$ and

$$\begin{cases} \delta_k(i,a) = \delta'_{k-1}(i,a), & \forall i \neq p, \forall a \in A; \\ \delta_k(p,w[0]) = |Q'_{k-1}|, & \\ \delta_k(p,a) = \delta'_{k-1}(p,a), & \forall a \in A, a \neq w[0];| \\ \delta_k(|Q'_{k-1}|+i, w[i+1]) = |Q'_{k-1}|+i+1, & \forall i = 0, \ldots |w|-3; \\ \delta_k(|Q'_{k-1}|+|w|-3, w[|w|-1]) = q, & \forall a \in A. \end{cases}$$

We have proved the following:

Theorem 1. *For each $k \in \{0, \ldots, m\}$, the language recognised by the automaton $\mathcal{A}_X^k$ is $L(\mathcal{A}_X^k) = \{x_0, \ldots, x_k\}$.*

In order to prove it we make use of the following lemma:

Lemma 2. *Let $k \in \{0, \ldots, m\}$. For each state i of $\mathcal{A}_X^k$ with $Deg^-(i) > 1$, there exists a unique path starting from i and ending at the final state q_{fin}.*

3 Construction algorithm

Let $X = (x_0, \ldots, x_m)$ be a list of words in A^* ordered by right-to-left lexicographic order and let $\sum_{i=0,m} |x_i| = n$. Let us call $\mathcal{A}_X$ the automaton $\mathcal{A}_X^m$ recognising X. In order to build it on-line we have to go through all the automata $\mathcal{A}_X^k$, $0 \leq k \leq m$.

For the construction of $\mathcal{A}_X$ we consider a matrix of n lines and 3 columns where we will memorize the values of the three functions H, Deg^{-1} and PF for each state of the automaton. In the outline, when we write $\mathcal{A}$, we will consider the automaton $\mathcal{A}$ together with this matrix. The outline of the algorithm for computing $\mathcal{A}_X$ is the following:

```
Construction-A_X(X)
 1. (A, R) ← Construction-A_X^0 (X[0])
    ◁ denote by q_fin the final state of A, define PF[q_fin] = 1
 2. for k ← 1 to |X| − 1 do
 3.     (A, R) ← Add-word(A, X[k], X[k − 1], R)
 4. Return A
```

Let us consider now the function Construction-$\mathcal{A}_X$. In line 1 we have the function Construction-$\mathcal{A}_0$ that computes the automaton $\mathcal{A}$ recognising $X[0]$. The automaton $\mathcal{A}$ is constructed using lists of adjacency. Its states are the integer $\{0, \ldots, |X[0]|\}$, 0 is the initial state and $|X[0]|$ is the final state. Moreover the function CONSTRUCTION-$\mathcal{A}_0$ returns a list R containing the sequence of states of $\mathcal{A}$ taken in the order of the construction.

In lines 2-3, for each k from 1 to $|X|$, we add to the automaton $\mathcal{A}$ the word $X[k]$ using the procedure ADD-WORD below.

ADD-WORD$(\mathcal{A}, x, y, R)$
1. *compute* s *the suffix in common between* x *and* y
2. $(\mathcal{A}, j, p) \leftarrow$ INDEGREE-CONTROL $(\mathcal{A}, x)$
3. **if** $|x| = j$ **then**
4. $p \leftarrow final(\mathcal{A})$
5.   *redefine PF for the states in the path from* 0 *with label* x
6.   *define R as the list of states in the path from* 0 *with label* x
7. **else**
8.   **if** $|x| - |s| \leq j$ **then** $s \leftarrow s[j+1\,..\,|s|-1]$
9. $q \leftarrow R[|R| - |s|]$
10. $(\mathcal{A}, q, h) \leftarrow$ PF-CONTROL$(\mathcal{A}, q, s)$
11.   **if** $PF[q] \neq 1$ **then** $q_{fin} \leftarrow q$
12.   **else**
13. $s \leftarrow s[h\,..\,|s|-1]$
14. $(\mathcal{A}, q, h) \leftarrow$ HEIGHT-CONTROL$(\mathcal{A}, p, q, s)$
15.   **if** $x[0\,..\,j-1] \in X$ **then**
16.   *delete the last edge of the path* c *starting at* 0 *with label* $x[0\,..\,j-1]$
17.   *add an edge from* p_1 *, ending state of* c *, to a new state* p_2
18. $p_2 \leftarrow final(\mathcal{A})$
19. $(\mathcal{A}, R) \leftarrow$ ADD PATH$(\mathcal{A}, x[0\,..\,j-1], x[j\,..\,h-1], q)$
20. Return $(\mathcal{A}, R)$

Let us now see more in detail how the procedure ADD-WORD works. It has as input an automaton $\mathcal{A}$ and two words x and y and it returns the automaton obtained from $\mathcal{A}$, by adding the word x, and R, the sequence of states along the path corresponding to the added word x. In line 1 it computes s the suffix in common x and y. In line 2 it calls the INDEGREE-CONTROL function on $(\mathcal{A}, x)$.

INDEGREE-CONTROL$(\mathcal{A}, x)$
1. $p \leftarrow 0$, $j \leftarrow 0$, $InDegControl \leftarrow 0$
2. **while** $\delta(p, x[j]) \neq NIL$ **and** $j \neq |x|$
3. $p_1 \leftarrow p$
4. $p \leftarrow \delta(p, x[j])$
5.   **if** $InDegControl = 0$ **then**
6.   **if** $Deg^-[p] \neq 1$ **then**
7.   *create an edge from* p_1 *to a new state* p_2 with label $x[j]$
8.   *define* Deg^- *for* p_2 *and for* p
9. $InDegControl \leftarrow 1$
10.   **else**
11.   *create an edge from* p_2 *to a new state* q with label $x[j]$
12. $Deg^-[q] \leftarrow 1$
13. $p_2 \leftarrow q$
14. $j \leftarrow j + 1$
15. **if** $InDegControl = 1$ **then**
16.   *for each edge starting at* p*, with label* a *and ending state* q
17.   *create an edge from* p_2 *to* q *with label* a
18. $Deg^-[\delta(p_2, a)] \leftarrow Deg^-[\delta(p_2, a)] + 1$
19. $H[p_2] \leftarrow H[p]$
20. $p \leftarrow p_2$
21. Return $(\mathcal{A}, j, p)$

Such a function reads the word x in $\mathcal{A}$ until it is possible. Let us call u the longest prefix of x that is the label of an accessible path in $\mathcal{A}$, let p be the ending state of this path. If, in such a path, there is an edge $r1 \xrightarrow{a} r$ such that r has indegree greater than 1 then the function creates a path from $r1$ labelled by the remaining part of u, let p_2 be its ending state. It redefines also the function Deg^- for the states in the new path.

In this case, for each edge starting at p with label a and ending at a state p', it creates an edge starting at p_2 with label a and ending at p'. It calls p the state p_2 and it defines the height of p.

The Indegree-Control function returns $\mathcal{A}$, j and p, where j is the length of the longest prefix of x which is the label of an accessible path in $\mathcal{A}$ and p is the ending state of this path.

Let us come back to Add-word. In line 3 it controls if $x[0\,..\,j-1] = x$, that is if x is the prefix of an already seen word. In this case in lines 4-6 it puts p in final states, it redefines PF for the states on the path labelled x and it defines R as the list of states in the path from 0 with label x.

If $x[0\,..\,j-1] \neq x$, that is x is not the prefix of an already seen word, then we go to line 8. If $|x| - |s| \leq j$ then we redefine s. In line 9 we use R in order to find the state q such that there is a path from q to the final state q_{fin} with label s.

In line 10 the PF-Control function is called. It takes as argument the automaton $\mathcal{A}$, q and s. The function reads from q the word s until either it finds a state q' with $PF[q'] = 1$ or it ends reading s. If s' is the label of the path from q to q' then it returns the length of such a path h. In line 11 if $PF[q]$ is greater than 1 then we define q as q_{fin}. Otherwise we go to line 13 where we redefine s as $s[h\,..\,|s|]$.

In line 14 we have a call to the Height-control function. It takes as argument the automaton $\mathcal{A}$, p, q and the word s. Such a function reads in $\mathcal{A}$, starting at q, the word s until it finds a state q' with $H[p] > H[q']$. If s' is the label of the path from q to q' then it returns the length of such a path h.

```
PF-Control (A, q, s)
h ← 0
while PF[q] ≠ 1 and h ≠ |s|
      q ← δ(q, s[h])
      h ← h + 1
Return (A, q, h)
```

```
Height-Control (A, p, q, s)
h ← 0
while H[p] ≤ H[q] and h ≠ |s|
      q ← δ(q, s[h])
      h ← h + 1
Return (A, q, h)
```

In line 15 it controls if $x[0\,..\,j-1]$ is in X. In such a case it does the transformation as written in lines 16, 17 and 18. In line 19 we call the function Add-path on $(\mathcal{A}, x[0\,..\,j-1], x[j\,..\,h-1], q)$.

The function Add-path takes as argument $(\mathcal{A}, u, w, q)$ with u and w words and q state of $\mathcal{A}$. It returns the automaton $\mathcal{A}$ obtained by adding a path with label w from p, final state of the path in $\mathcal{A}$ from 0 with label u, to q. The function creates the path from p to q with label w and defines H, PF and Deg^- for the new states. It redefines H, PF and Deg^- for the states of the path from 0 with label u. Finally it puts all the states on the path from 0 to q_{fin} with label x in a list R. Then it returns the automaton $\mathcal{A}$ and the list R.

Time complexity

We define $\mathcal{A}_X^0$ using the list of adjacency. So we compute $\mathcal{A}_X^0$ with the associated matrix and R in $\mathcal{O}(|X[0]|)$. Let us analyze the time complexity of the other functions.

For each k, let us call u the longest prefix common to $X[k]$ and $\{X[0], \ldots, X[k-1]\}$. The INDEGREE-CONTROL function has time complexity $\mathcal{O}(|u|)$. Let us call x the word $X[k]$ and s the suffix in common between x and $X[k-1]$. The HEIGHT-CONTROL function works in $\mathcal{O}(h)$. The PF-CONTROL function works in $\mathcal{O}(h)$ also. Since $\mathcal{O}(h)$ are $\mathcal{O}(|s|)$ then the functions work in time $\mathcal{O}(|s|)$. The ADD PATH function works in time $\mathcal{O}(|x|)$.

Since the other instructions in ADD-WORD work in $\mathcal{O}(1)$ we get that the running time for executing ADD-WORD is $\mathcal{O}(|x|)$. And we get that the time complexity of CONSTRUCTION-$\mathcal{A}_X$ is $\mathcal{O}(|X|)$.

3.1 Non minimality of the automaton: example

Given X a finite language, the automaton $\mathcal{A}_X$ is not necessarily minimal. This can follow, for example, from the not necessary indegree control done while building an automaton.

In the example in Figure 1 we see the construction of $\mathcal{A}_X^0$ and $\mathcal{A}_X^1$ for $X =$ (aaa,ba,aab,bb). In order to construct $\mathcal{A}_X^2$ we have to do the indegree control as in Figure 3. In Figure 8 we have $\mathcal{A}_X^3$ that is not minimal since the states 2 and 4 are equivalent.

The non minimality follows here from the indegree control. In fact, in this case the indegree control would not be necessary since bb is also in X (see Figure 2). So the algorithm creates unnecessary states and the automaton $\mathcal{A}_X^3$ results to be non minimal.

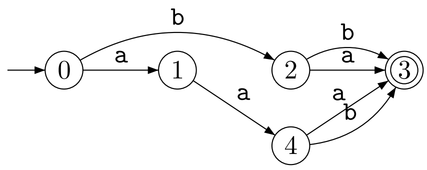


Figure 8. Non minimal automaton $\mathcal{A}_X^3$ for $X =$ (aaa,ba,aab,bb).

4 The algorithm for the set of suffixes of a given word

Let y in A^* and let us consider $S(y)$ sorted by decreasing order on the lengths of the elements in $S(y)$. For each $y \in A^*$, let us denote by $\mathcal{A}_y$ the automaton $\mathcal{A}_{S(y)}$ and by $\mathcal{M}_y$ the minimal automaton of $S(y)$. Given an automaton $\mathcal{A}$, let us denote by $\sharp\mathcal{A}$ the number of states of $\mathcal{A}$. For each y in A^*, in order to estimate the distance of $\mathcal{A}_y$ to its minimal automaton we consider the ratio $D(y) = \frac{\sharp\mathcal{A}_y}{\sharp\mathcal{M}_y}$.

We have done some experiments by generating all the words of a fixed length n. For each fixed length n we have considered D_n^{max} the greatest of $D(y)$ with y of length n.

n	D_n^{max}
10	1.83
15	2.41
20	3.04

In general the experimental results are good since D_n^{max} is not greater than 4 for words y with $|y| \leq 20$. Moreover the experiments done show that bad cases are linked with words that are powers of a short one with great exponent. So we thought that such words brought to automata far from being minimal (with great $D(y)$), or equivalently, that words with small entropy would have a great ratio $D(y)$.

Thus we have done experiments by generating 2000 words of a fixed length n with some constraints. For each of this experiment we have considered D_n, the greatest ratio among the $D(y)$. We report the results for different values of n in the following table. In the first column we have generated words such that either are not powers of the same word or that are powers of a word with an exponent less than a fixed number.

n	$exp < 3$	$exp < 2$	$exp < 1$
10	1.75	1.66	1.54
20	2.22	2.16	2.42
30	2.16	2.22	2.24
50	1.96	1.85	2.60
100	1.60	1.71	1.79

The experimental results are good in general even if they do not show clearly our conjecture. In the following we propose another approach.

4.1 Modified construction

Let y in A^* and $S(y) = [y_0 = y, \ldots, y_m]$ sorted by decreasing order on the lengths of the elements in $S(y)$. Let us denote by $\mathcal{A}_y^k$ the automaton $\mathcal{A}_{S(y)}^k$.

In case y_k is not a prefix of an already seen word, we consider the construction of the automaton $\mathcal{A}_y^k$ taking q in the path from 0 with label y and not in that one with label y_{k-1}. Let us note that in case of suffixes of a word we have that $y_k = uas$ with a in A and u, s defined as in Section 2. Moreover let us note that if there are two edges ending at p, state of $\mathcal{A}_y^k$, then they have the same label.

In this section we will propose a modification on the indegree control in order to avoid equivalent states as in the example in Figure 8. Before doing it we will note, with the following two propositions, that, in case of suffixes of a word, we do not have to execute the PF control and the Height control. In particular we prove that $PF(q) = 1$ and that $H(p) > H(q)$, with p and q as defined in Section 2.

Proposition 3. *Let y in A^* and y_k in $S(y)$ such that y_k is not a prefix of a word in $\{y_0, \ldots, y_{k-1}\}$. Then we have that $PF(q) = 1$.*

Proposition 4. *Let y in A^* and y_k in $S(y)$ such that y_k is not a prefix of a word in $\{y_0, \ldots, y_{k-1}\}$. Then we have that $H(p) > H(q)$.*

Let us consider the construction of $\mathcal{A}_y^k$. We have the following proposition:

Proposition 5. *Let y in A^* and y_k in $S(y)$. Let $y_k = uz$, with u such that there exists a path starting at 0 with label u in $\mathcal{A}_y^{k-1}$, let p its final state. If there exists a path from 0 to p with label v in $\mathcal{A}_y^{k-1}$ then, if $|v| < |u|$ then $vz \in \{y_{k+1}, \dots, y_m\}$, otherwise $vz \in \{y_0, \dots, y_{k-1}\}$.*

Let us associate with each state p of $\mathcal{A}_y^{k-1}$ the list $L(p)$ of the states q such that there exists an edge from q to p. We construct such list iteratively adding each time an element to the tail of the list.

With each state q in $L(p)$ we associate $V(q)$ the set of words such that there exists a path from 0 to q. Let us denote by p_h the state $L(p)[h]$.

Proposition 6. *Let y in A^* and let p be a state of $\mathcal{A}_y^k$ with $Deg^-(p) > 1$. Let $i < j < |L(p)|$. Then, for each u in $V(p_i)$ and for each v in $V(p_j)$, we have that $|u| > |v|$.*

We propose a new construction of $\mathcal{A}_y^k$ with the definition of $L(p)$, for each state p, and a different indegree control. Let u be the longest prefix in common between y_k and $\{y_0, \dots, y_{k-1}\}$ and p' the ending state of the path starting at 0 with label u.

The function reads the word y_k in $\mathcal{A}_y^{k-1}$ until it is possible. While the function reads y_k, the visiting state is called p and the state visited in the step before is called p_1. In particular we have that p_1 is in $L(p)$.

If the function finds a state p with Indegree greater than 1 and if p_1 is not equal to $L(p)[0]$ then, if a is the label of the edge from p_1 to p,

- it deletes all the edges starting at states in $L(p)$ that have a position in $L(p)$ greater or equal to that one of p_1.
- it creates a new state p_2 and it creates, for each state r in $L(p)$ that has a position greater or equal to that one of p_1, an edge from r to p_2 with label a
- it creates a path from p_2 with label the resting part of u. Let p be the end state of this path.
- it creates, for each edge, starting at p' an edge starting at p with the same ending state .

Time complexity

For each $\mathcal{A}_y^k$, for each state p in $\mathcal{A}_y^k$ we have that $Deg^-(p) \leq (k+1)$. In the indegree control, in the worst case, we have to visit completely the list for the state p with $Deg^-(p) > 1$ and such that $p_1 \neq L(p)[0]$. So for each k, in the worst case, the indegree control takes time $\mathcal{O}(|u| + k)$.

In total the contributions of the visit of the lists $L(p)$ for indegree controls take time $\mathcal{O}(\sum_{k=0,m} k) = \mathcal{O}(|S(y)|)$, so we have that in the worst case the algorithm works in $\mathcal{O}(|S(y)|)$.

5 Conclusion

The algorithm presented in the article builds a small automaton accepting a finite set of words. It has several advantages. It allows an extremely fast compiling of the set of words. With little modification, the method can handle efficiently updates of the automaton, and especially addition of new words. The condition imposed on the list of words is not a restriction because words can always be maintained sorted according to lexicographic order.

One open problem is to find a general upper bound for ratios D (ratio D is the quotient of the number of states of $\mathcal{A}_y$ and of the number of states of its minimal automaton).

Experiments leads us to conjecture that the ratios are bounded by a fixed number, after possibly a small change in the algorithm.

For the suffixes of a word y, we expect that an improved version of the algorithm actually builds the (minimal) suffix automaton of y.

The main open question is whether there exists an on-line construction for the minimal automaton accepting a finite set of words that runs in linear time on each word being inserted in the automaton.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman: *The design and analysis of computer algorithms*, Addison-Wesley Publishing Company, 1974.
2. S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman: *Basic local alignment search tool.* J. Mol. Biol., 215 1990, pp. 403–410.
3. J. Berstel and D. Perrin: *Theory of Codes*, Academic Press, 1985.
4. J. Clément, J.-P. Duval, G. Guaiana, D. Perrin, and G. Rindone: *Parsing with a finite dictionary.* Theoretical Computer Science, 340 2005, pp. 432–442.
5. C.Martin-Vide and V. Mitrana: *Grammars and Automata for String Processing: From Mathematics and Computer Science to Biology, and Back*, Taylor and Francis, 2003.
6. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on Strings*, Cambridge University Press, Cambridge, UK,, 2007.
7. J. Daciuk: *Comparison of construction algorithms for minimal, acyclic, deterministic, finite-state automata from sets of strings*, in Proceedings of CIAA 2002, vol. 2608 of LNCS, 2003, pp. 255–261.
8. J. Daciuk, S. Mihov, B. W. Watson, and R. E. Watson: *Incremental construction of minimal acyclic finite state automata.* Computational linguistics, 26(1) 2000, pp. 3–16.
9. J. E. Hopcroft and J. D. Ullman: *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley Publishing Company, 1979.
10. N. D. F. K. N. Sgarbas and G. K. Kokkinakis: *Optimal insertion in deterministic DAWGs.* Theoretical Computer Science, 301 2003, pp. 103–117.
11. D. Revuz: *Dictionnaires et lexiques: mthodes et algorithmes*, PhD thesis, Institut Blaise Pascal, Paris, France, LITP 91.44, 1991.
12. D. Revuz: *Minimization of acyclic deterministic automata in linear time.* Theoretical Computer Science, 92, number 1 1992, pp. 181–189.
13. B. W. Watson: *A taxonomy of algorithms for constructing minimal acyclic deterministic finite automata.* South African Computer Journal, 27 2001, pp. 12–17.
14. B. W. Watson: *A fast and simple algorithm for constructing minimal acyclic deterministic finite automata.* Journal of Universal Computer Science, 8, number 2 2002, pp. 363–367.
15. B. W. Watson: *A new algorithm for the construction of minimal acyclic DFAs.* Science of Computer Programming, 48 2003, pp. 81–97.

6 Appendix

Proof of Lemma 2

We will prove the lemma by induction on k. For $k=0$ it is easily true.

Let us suppose that it is true for $k-1$ and let us prove it for k. If i is a state of $\mathcal{A}_X^k$ with $Deg^-(i)>1$ then i is not contained in the path from 0 to p relative to x_k, by construction. So by the inductive hypothesis there is a unique path from i to q_{fin}.

Proof of Theorem 1

We will prove the theorem by induction on k. For $k=0$ it is easily true.

Let us suppose that it is true for $k-1$ and let us prove it for k. Let us prove that the automaton $\mathcal{B}_X^{k-1}$, obtained after the indegree control, recognises $\{x_0,\ldots,x_{k-1}\}$, that is $L(\mathcal{B}_X^{k-1})=L(\mathcal{A}_X^{k-1})$. Let us suppose to be in CASE I otherwise it is trivial.

Trivially we have that $L(\mathcal{B}_X^{k-1})\subseteq L(\mathcal{A}_X^{k-1})$. For the other inclusion, let d be a successful path in $\mathcal{A}_X^{k-1}$. If d does not contain the edge $r_0\xrightarrow{x[\ell]}r$ then the path d will be also in $\mathcal{B}_X^{k-1}$. If d contains the edge $r_0\xrightarrow{x[\ell]}r$ then d contains necessarily as subpath $r_0\xrightarrow{x[\ell]}r\xrightarrow{u_1}p$, in fact, since $Deg^-(r_0)>1$, by Lemma 2, there exists a unique path starting at r_0 and ending at q_{fin}. So there exists in $\mathcal{B}_X^{k-1}$ a successful path with the same label as d.

Let us prove now that the automaton $\mathcal{A}_k$ recognises $\{x_0,\ldots,x_k\}$. If x is the prefix of a word in $\{x_0,\ldots,x_{k-1}\}$ then we add p to the set of final states and since $Deg^-(p)=1$ we only add x_k to $L(\mathcal{A}_X^{k-1})=\{x_0,\ldots,x_{k-1}\}$.

Otherwise, if $p=q_{fin}$ then we transform $\mathcal{B}_X^{k-1}$ in an automaton recognising the same language.

In all cases the automaton $\mathcal{A}_k$ is obtained from $\mathcal{B}_X^{k-1}$ by adding a path from p to q with label w, as defined before.

By the 'indegree control', there exists a unique path in $\mathcal{B}_X^{k-1}$ from 0 to p with label u and, by the 'paths toward final states control' there exists a unique path in $\mathcal{B}_X^{k-1}$ from q to q_{fin} with label s. Moreover, since $H(p)>H(q)$, there are no paths from q to p, otherwise there would be a path from q to q_{fin} longer than every path from p to q_{fin}.

Thus we only add to $L(\mathcal{A}_X^{k-1})$ the word $x=uws$, that is the thesis.

Proof of Proposition 3

Let $y_k=uas$, with u and s as defined before. By contradiction, if $PF(q)>1$ then, there exists $i<k$ such that $y_i=u_1bs_1$ and $|s|>|s_1|$. Since y_k is a suffix of y_i we have that $s=ts_1$, for some word $t\neq\varepsilon$. Since $y_k=uats_1$ is a suffix of $y_i=u_1bs_1$ then there exists $z\neq\varepsilon$ such that uz is a suffix of u_1.

Since $y_i=u_1bs_1$ then there exists y_h with $h<i$ such that u_1 is a prefix of y_h. Let $y_h=u_1cs_2$, then we have that $|s_2|>|s_1|$ since $|y_h|>|y_i|$. Since uz is a suffix of u_1 there exists a suffix y_l of y_h with $y_l=uzcs_2$. Since $|s_2|>|s_1|$ we get $|y_l|=|uzcs_2|>|uzbs_1|=|y_k|$. So uz is a prefix in common between y_k and y_l, $l<k$, that is a contradiction since u was the greatest prefix in common between y_k and the words in $\{y_0,\ldots,y_{k-1}\}$.

Proof of Proposition 4

Let $y_k = uas$ with u and s defined as before. Since p is co-accessible, there exists a word uz in $\{y_0, \ldots, y_{k-1}\}$. By Prop. 3, there exists only one path from q to q_{fin} whose label is s.
If $H(p) \leq H(q)$ then $|s| \geq |z|$ and so $|y_k| = |uas| > |uz|$ that is a contradiction since uz in in $\{y_0, \ldots, y_{k-1}\}$.

Proof of Proposition 5

The state p is co-accessible and so let z' be the label of a path from p to a final state. Then uz' and vz' are in $\{y_0, \ldots, y_{k-1}\}$. If $|v| < |u|$ then v is a suffix of u and so vz is a suffix of uz and vz is in $\{y_{k+1}, \ldots, y_m\}$.

If $|v| \geq |u|$, then $|vz| \geq |uz|$. Thus uz is a suffix of vz and vz is in $\{y_0, \ldots, y_{k-1}\}$.

Proof of Proposition 6

The list $L(p)$ is iteratively constructed adding each time an element to the tail of $L(p)$. Then, for each $i < j < |L(p)|$, and for u in $V(p_i)$ and v in $V(p_j)$, u is the label of a path added during the construction of $\mathcal{A}_y^l$, v is the label of a path added during the construction of $\mathcal{A}_y^r$, with $l < r$. Since p is co-accessible in $\mathcal{A}_y^l$, we have that uaz in $S(y)$ and so vaz in $S(y)$, for some word z and some a in A. Since v is constructed in $\mathcal{A}_y^r$ with $r > l$ we get that $|v| < |u|$.

Adapting Boyer-Moore-Like Algorithms for Searching Huffman Encoded Texts

Domenico Cantone, Simone Faro, and Emanuele Giaquinta

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

Abstract. In this paper we propose an efficient approach to the compressed string matching problem on Huffman encoded texts, based on the Boyer-Moore strategy. Once a candidate valid shift has been located, a subsequent verification phase checks whether the shift is codeword aligned by taking advantage of the skeleton tree data structure. Our approach leads to algorithms with a sublinear behavior on the average, as shown by extensive experimentation.

Keywords: string matching, compression algorithms, Huffman coding, Boyer-Moore algorithm, text processing, information retrieval

1 Introduction

The *compressed string matching problem* is a variant of the classical string matching problem. It consists in searching for all the occurrences of a given pattern P in a text T stored in compressed form.

A straightforward solution is the so-called *decompress-and-search* strategy, which consists in decompressing the text and then using any classical string matching algorithm for searching. However, recent results show that in many cases searching directly in compressed texts can be more efficient.

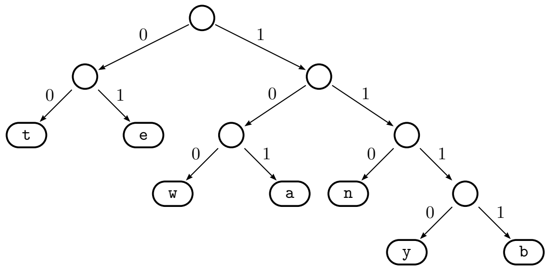
Here we are interested in the string matching problem on Huffman compressed texts. The Huffman data compression method [7] is an optimal statistical coding. More precisely, the Huffman algorithm computes an optimal *prefix code*, relative to given frequencies of the alphabet characters. A prefix code is a set of (binary) words containing no word which is a prefix of another word in the set. Thanks to such a property, decoding is particularly simple. Indeed, a binary prefix code can be represented by an ordered binary tree, whose leaves are labeled with the alphabet characters and whose edges are labeled by 0 (left edges) and 1 (right edges) in such a way that the codeword of an alphabet character is the word labeling the branch from the root to the leaf labeled by the same character.

Prefix code trees, as computed by the Huffman algorithm, are called *Huffman trees*. These are not unique, by any means. The usually preferred tree for a given set of frequencies, out of the various possible Huffman trees, is the one induced by *canonical Huffman codes* [14]. This tree has the property that, when scanning its leaves from left to right, the sequence of depths is nondecreasing.

When performing a search on the bitstream of a Huffman encoded text by a classical string matching algorithm, one faces the problem of *false matches*, i.e., occurrences of the encoded pattern in the encoded text which do not correspond to occurrences of the pattern in the original text. Indeed, the only valid occurrences of the pattern are those correctly aligned with codeword boundaries, or, otherwise said, valid matches

Domenico Cantone, Simone Faro, Emanuele Giaquinta: Adapting Boyer-Moore-Like Algorithms for Searching Huffman Encoded Texts, pp. 29–39.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

`t` : `00`
`e` : `01`
`w` : `100`
`a` : `101`
`n` : `110`
`y` : `1110`
`b` : `1111`



`twenty`	$\bar{0}0\bar{1}00\bar{0}1\bar{1}10\bar{0}0\bar{1}110$		
`ten`		$\bar{0}0\bar{0}1\bar{1}10$	
`ten`			$\bar{0}0\bar{0}1\bar{1}10$

Figure 1. A Huffman code for the set of symbols {`t`, `e`, `w`, `a`, `n`, `y`, `b`}. The binary string $\bar{0}0\bar{1}00\bar{0}1\bar{1}10\bar{0}0\bar{1}110$ is the encoding of the string `twenty`, where a "bar" indicates the starting bit of each codeword. Two occurrences of the binary string `ten` start at the 4-th and 10-th bit of the encoded version of the string `twenty`. Both of them are false matches.

must start on the first bit of a codeword. Consider, for example, the Huffman code presented in Figure 1. Note that there are two false occurrences of the string `ten` starting at the 4-th and at the 10-th bit, respectively, of the encoded string `twenty`. Thus a verification that the occurrences detected by the pattern matching algorithm are aligned on codeword boundaries is in order.

False matches could be avoided by using codes in which no codeword is a prefix or a suffix of any other codeword. However, such codes, which are called *affix* or *fix-free*, are extremely infrequent [5].

Klein and Shapira [11] showed that, for long enough patterns, the probability of finding a false match is often very low, independently of the algorithm. They then proposed a probabilistic algorithm which works on the assumption that Huffman codes tend to realign quickly after an error.

More recently, Shapira and Daptardar [15] proposed a modification of the Knuth-Morris-Pratt algorithm [12], in this paper referred to as Huffman-Kmp, which makes use of a data structure, called *skeleton tree* [9], suitably designed for efficient decoding of Huffman encoded sequences. The resulting algorithm is characterized by fast search times, if compared with the *decompress-and-search* method.

Algorithms based on the Boyer-Moore algorithm [2] have been considered unsuitable for searching Huffman encoded texts because the right to left scan does not allow one to determine the codeword boundaries in the compressed text, unless the text is decoded from left to right. In addition, Boyer-Moore-like algorithms are generally considered unsuitable for binary alphabets.

In this paper we present a new way to exploit skeleton trees for adapting Boyer-Moore-like algorithms to the compressed string matching problem in Huffman encoded texts. Specifically, we use skeleton trees to verify codeword alignments rather than for decoding. This allows us to skip up to 70% of bits during the processing of the encoded text. Futhermore, we make use of algorithms based on the Boyer-Moore strategy, suitably adapted for searching on binary strings by regarding texts and patterns as sequences of q-grams rather than as sequences of bits.

The paper is organized as follows. In Section 2 we introduce basic definitions and notations. In Section 3 we describe a strategy based on skeleton trees which is not specific to any algorithm and then in Section 4 we apply it to two string matching algorithms for binary strings. In Section 5 we present some experimental results and finally we draw our conclusions in Section 6.

2 Some Basic Definitions and Preliminaries

A string P of length $|P| = m \geq 0$ is represented as a finite array $P[0\,..\,m-1]$ of characters from a finite alphabet Σ. In particular, for $m = 0$ we obtain the empty string ε. By $P[i]$ we denote the $(i+1)$-st character of P, for $0 \leq i < m$. Likewise, by $P[i\,..\,j]$ we denote the substring of P contained between the $(i+1)$-st and the $(j+1)$-st characters of P, for $0 \leq i \leq j < m$. Moreover, for any $i, j \in \mathbb{Z}$, we put

$$P[i\,..\,j] = \begin{cases} \varepsilon & \text{if } i > j \\ P[\max(i,0)\,..\,\min(j,m-1)] & \text{if } i \leq j. \end{cases}$$

A substring of the form $P[0\,..\,i]$ is called a *prefix* of P and a substring of the form $P[i\,..\,m-1]$ is called a *suffix* of P, for $0 \leq i \leq m-1$. For any two strings P and Q, we write $Q \sqsupseteq P$ to indicate that Q is a suffix of P. Similarly, we write $Q \sqsubseteq P$ to indicate that Q is a prefix of P. In addition, we write $Q.P$ to denote the concatenation of Q and P. Also, if P is a string of length m and $P[i] = b$, for $i = 0, \ldots, m-1$, then we write $P = b^m$.

A *compression method* for a given text T over an alphabet Σ is characterized by a system $(\mathcal{E}, \mathcal{D})$ of two complementary functions,

- an *encoding function* $\mathcal{E} : \Sigma \to \{0,1\}^+$, and
- an inverse *decoding function* $\mathcal{D}$,

such that $\mathcal{D}(\mathcal{E}(c)) = c$, for each $c \in \Sigma$. The encoding function $\mathcal{E}$ is then recursively extended over strings of characters by putting

$$\begin{aligned} \mathcal{E}(\varepsilon) &= \varepsilon \\ \mathcal{E}(T[0\,..\,\ell]) &= \mathcal{E}(T[0\,..\,\ell-1]).\mathcal{E}(T[\ell]), \quad \text{for } 0 \leq \ell < |T|, \end{aligned}$$

so that $\mathcal{E}(T) = \mathcal{E}(T[0\,..\,|T|-1])$ is just a binary string, i.e., a string over the alphabet $\{0,1\}$.

For ease of notation, we usually write t in place of $\mathcal{E}(T)$ and, more generally, denote binary strings by lowercase letters.

Binary strings are conveniently stored in blocks of k bits, typically bytes ($k = 8$), half-words ($k = 16$), or words ($k = 32$), which can be processed at the cost of a single operation. If p is any binary string, we denote by B_p the vector of blocks whose concatenation gives p, for a given block size k, so that

$$p[i] = B_p[\lfloor i/k \rfloor][i \bmod k], \quad \text{for } i = 0, \ldots, |p|-1$$

(we assume that the last block, if not complete, is padded with 0's).

Thus, a genuine solution to the *compressed string matching problem* consists in finding *all* occurrences of a pattern P in a text T, over a common alphabet Σ, by operating directly on the block vectors B_t and B_p, representing respectively the binary strings $t = \mathcal{E}(T)$ and $p = \mathcal{E}(P)$ (again relative to a fixed block size k).

The algorithms for the compressed string matching problem in Huffman encoded texts, to be presented in Section 3, are based on a high-level model to process binary strings, adopted in [10,8,4], which we review next.

(A) *Patt*	0	1	2	3
0	11001011	00101100	10110000	
1	01100101	10010110	01011000	
2	00110010	11001011	00101100	
3	00011001	01100101	10010110	
4	00001100	10110010	11001011	00000000
5	00000110	01011001	01100101	10000000
6	00000011	00101100	10110010	11000000
7	00000001	10010110	01011001	01100000

(C) *Last*
2
2
2
2
3
3
3
3

(B) *Mask*	0	1	2	3
0	11111111	11111111	11111000	
1	01111111	11111111	11111100	
2	00111111	11111111	11111110	
3	00011111	11111111	11111111	
4	00001111	11111111	11111111	10000000
5	00000111	11111111	11111111	11000000
6	00000011	11111111	11111111	11100000
7	00000001	11111111	11111111	11110000

Figure 2. Let $P =$110010110010110010110. **(A)** The matrix *Patt*. **(B)** The matrix *Mask*. **(C)** The array *Last*. In the tables *Patt* and *Mask*, bits belonging to P are underlined. Blocks containing a factor of P of length 8 have a shaded background.

2.1 A High-Level Model for Matching on Binary Strings

Let us assume that the block size k is fixed, so that all references to both text and pattern will only be to entire blocks of k bits. We refer to a k-bit block as a *byte*, though larger values than $k = 8$ could be supported as well.

We first define a vector *Patt*, of size $k \times (\lceil m/k \rceil + 1)$, consisting in several copies of the pattern P stored in the form of a matrix B_p of bytes, where $p = \mathcal{E}(P)$ and $m = |p|$. More precisely, the i-th row of the matrix *Patt*, for $i = 1, \ldots, k$, contains a copy of p shifted by i position to the right, whose length in bytes is $m_i = \lceil (m+i)/k \rceil$. The i leftmost bits of the first byte remain undefined and are set to 0. Similarly, the rightmost $((k - ((m + i) \bmod k)) \bmod k)$ bits of the last byte are set to 0.

Observe that each factor of p of length k appears exactly once in the table *Patt*. For instance, the factor of length k starting at position j of p is stored in $Patt[k - (j \bmod k), \lceil j/k \rceil]$.

The vector *Patt* is paired with a matrix of bytes, *Mask*, of size $k \times (\lceil m/k \rceil + 1)$, containing binary masks of length k, to distinguish between significant and padding bits in *Patt*. In particular, a bit in the mask $Mask[i, h]$ is set to 1 if and only if the corresponding bit of $Patt[i, h]$ belongs to p.

Finally, we define a vector *Last*, of size k, where $Last[i]$ is the index of the last byte in the row $Patt[i]$, i.e., $Last[i] = m_i$, for $0 \leq i < k$.

The procedure Preprocess used to precompute the above vectors requires $\mathcal{O}(k \times \lceil m/k \rceil) = \mathcal{O}(m)$ time and $\mathcal{O}(m)$ extra-space. Figure 2 shows the tables *Patt*, *Mask*, and *Last* relative to the pattern $P =$110010110010110010110, for a block size $k = 8$.

When the pattern is aligned with the s-th bit of the text, a match is reported if

$$Patt[i, h] = B_t[j + h] \mathbin{\&} Mask[i, h]\,,$$

for $h = 0, 1, \ldots, Last[i]$, where

- B_t is the block representation of the text encoding $t = \mathcal{E}(T)$,
- $j = \lfloor s/k \rfloor$ is the starting byte position in t,
- $i = (s \bmod k)$, and
- "&" is the bitwise logic AND.

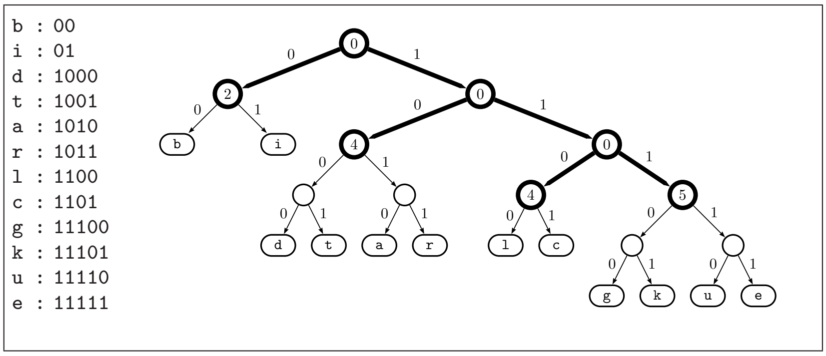


Figure 3. The Huffman tree induced by a Huffman code for the set of symbols $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{g}, \mathtt{i}, \mathtt{k}, \mathtt{l}, \mathtt{r}, \mathtt{t}, \mathtt{u}\}$. The skeleton tree is identified by bold lines.

3 Skeleton Tree Based Verification

The *skeleton tree* [9] is a pruned canonical Huffman tree, whose leaves correspond to minimal depth nodes in the Huffman tree which are roots of complete subtrees. It is useful to maintain at each leaf of a skeleton tree the common length of the codeword(s) sharing the prefix which labels the path from the root to it. A fast algorithm for building skeleton trees is described in [9]. Figure 3 shows a canonical Huffman tree and its corresponding skeleton tree, for the set of symbols $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{g}, \mathtt{i}, \mathtt{k}, \mathtt{l}, \mathtt{r}, \mathtt{t}, \mathtt{u}\}$, relative to suitable character frequencies.

Skeleton trees allow a faster Huffman decoding because, once the codeword length has been retrieved at its leaves, it is possible to read a burst of bits to complete the codeword, or just to skip them, if one is only interested in finding codeword boundaries.

Our approach consists in searching for the candidate occurrences of B_p in B_t, where we recall that B_p and B_t are respectively the block vectors associated to given Huffman encoded pattern and text, using Boyer-Moore-like algorithms and then taking advantage of the skeleton tree to verify whether the candidate matches are codeword aligned. In this way we obtain a substantial speedup, especially when the frequency of the pattern in the text is low or when the length of the pattern increases.

For every candidate valid shift s found by the binary pattern matching algorithm, one must verify whether s is codeword aligned. For this purpose, we maintain an offset ρ pointing at the start of the last window where a skeleton tree verification took place. The offset ρ is then updated, with the aid of the skeleton tree, to a minimal position $\rho^* \geq s$ which is codeword aligned. Only if $\rho^* = s$ the current window is codeword aligned and s is a valid shift. Plainly, the performance of the algorithm depends on the number of skeleton tree verifications and on the relative distance between candidate valid shifts.

Figure 4 shows the pseudocode for the procedure Sk-Align used to update ρ. In the pseudocode we assume that the starting value of ρ is codeword aligned and that a node x in the skeleton tree is a leaf if the corresponding key is nonzero, i.e., if $\mathtt{Key}(x) > 0$. If $\mathtt{Key}(x) = \ell > 0$ and c_x is the bit code which labels the path from the root to x, then all codewords c such that $c_x \sqsubseteq c$ have a length equal to ℓ. Thus, if we are interested only in the codeword boundaries, we can skip the $\ell - |c_x|$ following bits and restore the skeleton-tree verification from the first bit of the next codeword.

Sk-Align ($root$, t, ρ, b)
$x \leftarrow root$, $\ell \leftarrow 0$
while TRUE **do**
  $B = B_t[\lfloor \rho \,/\, k \rfloor] \ll (\rho \bmod k)$
  if $B < 2^{k-1}$ **then** $x \leftarrow$ Left(x) **else** $x \leftarrow$ Right(x)
  if Key(x) $\neq 0$ **then**
    $\rho \leftarrow \rho +$ Key(x) $- \ell$, $\ell \leftarrow 0$, $x \leftarrow root$
    if $\rho \geq b$ **then break**
  else $\rho \leftarrow \rho + 1$, $\ell \leftarrow \ell + 1$
return ρ

Figure 4. Procedure Sk-Align($root$, t, ρ, b) which computes the next codeword alignment starting from position ρ, where $root$ is the root of the skeleton tree, t is the encoded text in binary form, b is a codeword boundary, and k is the block size ($\ll$ denotes the left shift operator).

Consider as an example the search of the pattern $P =$ "`bit`" in the text $T =$ "`abigblackbugbitabigblackbear`". Suppose moreover that codewords are defined by the Huffman tree of Figure 3, so that $p = \mathcal{E}(P) =$"`00011001`".
A first candidate valid shift is encountered at position 12 in t, as shown below

$$\begin{array}{rl}
t & \texttt{101000011110}\underline{\texttt{00011001}}\texttt{01011011110100111101110000001100110100001}[\cdots] \\
p & \phantom{\texttt{101000011110}}\bar{\texttt{0}}\texttt{0}\bar{\texttt{0}}\texttt{1}\bar{\texttt{1}}\texttt{001} \\
\textit{verif.} & \bar{\texttt{1}}\texttt{0--}\bar{\texttt{0}}\texttt{-}\bar{\texttt{0}}\texttt{-}\bar{\texttt{1}}\texttt{11--}\bar{\texttt{0}}
\end{array}$$

The skeleton tree verification starts at position 0 and stops at position 13, skipping 6 bits over 14 (unprocessed bits are represented by the symbol "`-`"), showing that the occurrence at position 12 is not codeword aligned.

A second occurrence is found at the 45-th bit of t, as shown below

$$\begin{array}{rl}
t & [\cdots]\underline{\texttt{00011001}}\texttt{010110111101001111011100}\underline{\texttt{00011001}}\texttt{10100001111000}[\cdots] \\
p & \phantom{[\cdots]\texttt{00011001010110111101001111011100}}\bar{\texttt{0}}\texttt{0}\bar{\texttt{0}}\texttt{1}\bar{\texttt{1}}\texttt{001} \\
\textit{verif.} & \phantom{[\cdots]\texttt{0}}\bar{\texttt{0}}\texttt{-}\bar{\texttt{1}}\texttt{10-}\bar{\texttt{1}}\texttt{0--}\bar{\texttt{1}}\texttt{10-}\bar{\texttt{1}}\texttt{11--}\bar{\texttt{0}}\texttt{-}\bar{\texttt{1}}\texttt{11--}\bar{\texttt{1}}\texttt{11--}\bar{\texttt{0}}
\end{array}$$

The skeleton tree verification restarts from position 14 and finds a codeword alignment at position 45. Thus the occurrence is codeword aligned and the shift is valid. The verification skips 12 bits over 32.

Finally, a third candidate valid shift is found at the 65-th bit of t. This time, the skeleton tree verification skips 10 bits over 22.

$$\begin{array}{rl}
t & [\cdots]\underline{\texttt{00011001}}\texttt{101000011110}\underline{\texttt{00011001}}\texttt{010110111101001111110101011} \\
p & \phantom{[\cdots]\texttt{00011001101000011110}}\bar{\texttt{0}}\texttt{0}\bar{\texttt{0}}\texttt{1}\bar{\texttt{1}}\texttt{001} \\
\textit{verif.} & \bar{\texttt{0}}\texttt{-}\bar{\texttt{0}}\texttt{-}\bar{\texttt{1}}\texttt{0--}\bar{\texttt{1}}\texttt{0--}\bar{\texttt{0}}\texttt{-}\bar{\texttt{0}}\texttt{-}\bar{\texttt{1}}\texttt{11--}\bar{\texttt{0}}
\end{array}$$

The strategy presented above for verifying codeword alignment is general and not specific to any algorithm.

4 Adapting Two Boyer-Moore-Like Algorithms for Searching Huffman Encoded Texts

Next we deal with the problem of searching for all candidate valid shifts. For this purpose, we present two algorithms which are adaptations to the case of Huffman encoded texts, along the lines of the high-level model outlined in Section 2.1, of the Fed algorithm [8] and the Binary-Hash-Matching algorithm [4].

4.1 The Huffman-Hash-Matching Algorithm

Algorithms in the q-Hash family for exact pattern matching have been introduced in [13], by adapting the Wu and Manber multiple string matching algorithm [17] to the single string matching problem. Recently, variants of the q-Hash algorithms have been proposed for searching on binary strings [4].

The first algorithm which we present, called Huffman-Hash-Matching, associates directly each binary substring of length q with its numeric value in the range $[0, 2^q - 1]$, without using any *hash* function. To exploit the block structure of the text, the algorithm considers substrings of length $q = k$.

To begin with, a function $Hs : \{0, 1, \ldots, 2^k - 1\}, \rightarrow \{0, 1, \ldots, m\}$, defined by

$$Hs(B) = \min\Big(\{0 \leq u < m \mid p[m-u-k \,..\, m-u-1] \sqsupseteq B\} \cup \{m\}\Big)$$

for each byte $0 \leq B < 2^k$, is computed during the preprocessing phase. Observe that if $B = p[m-k \,..\, m-1]$, then $Hs[B] = 0$.

For example, in the case of the pattern $P =$ `110010110010110010110` presented in Figure 2, we have $Hs[$`01100101`$] = 2$, $Hs[$`11001011`$] = 1$, and $Hs[$`10010110`$] = 0$.

In contrast with algorithms in the q-Hash family, where the maximum shift is $m - q$, in this case maximum shifts can reach the value m. Since we do not use a hash function but rather map directly the binary substrings of the pattern, the shift table can be modified by taking into account the prefixes of the patterns $Patt[i]$ of length $k - i$, with $1 \leq i \leq k - 1$. Thus Hs can be conveniently computed by setting $Hs[B] = m-k+i$, where i is the minimum index such that $Patt[i][0] \sqsupseteq B$, if it exists; otherwise $Hs[B]$ is set to m.

The code of the Huffman-Hash-Matching algorithm is presented in Figure 5.

The preprocessing phase of the algorithm just consists in computing the function Hs defined above and requires $\mathcal{O}(m + k2^{k+1})$-time complexity and $\mathcal{O}(m + 2^k)$ extra space.

During the search phase, the algorithm reads, for each shift position s of the pattern in the text, the block $B = B_t[s+m-k \,..\, s+m-1]$ of k bits (line `9`). If $Hs(B) > 0$ then a shift of length $Hs(B)$ takes place (line `21`). Otherwise, if $Hs(B) = 0$, the pattern p is naively checked in the text block by block (lines `11-15`). The verification step is performed using the procedure Sk-Align described before (lines `16-19`).

After the test, an advancement of length *shift* takes place (line `20`), where

$$shift = \min\Big(\{0 < u < m \mid p[m-u-k \,..\, m-u-1] \sqsupset p[m-k \,..\, m-1]\} \cup \{m\}\Big).$$

Observe that if the block B has its $s\ell$ rightmost bits in the j-th block of t and the $(k - s\ell)$ leftmost bits in the block $B_t[j-1]$, then it is computed by performing the following bitwise operations (line `9`)

$$B = \Big(B_t[j] \gg (k - s\ell)\Big) \mid \Big(B_t[j-1] \ll (s\ell + 1)\Big)$$

The Huffman-Hash-Matching algorithm has an overall $\mathcal{O}(\lfloor m/k \rfloor n)$-time complexity and requires $\mathcal{O}(m + 2^k)$ extra space.

For blocks of length k, the size of the Hs table is 2^k, which seems reasonable for $k = 8$ or even 16. For greater values of k it is possible to adapt the algorithm to choose

```
Huffman-Hash-Matching (p, m, t, n)
root ← Build-Sk-Tree(φ)
(Patt, Last, Mask) ← Preprocess (p, m)
Hs ← compute-hash(Patt, Last, Mask, m)
ρ ← 0
i ← (k − (m mod k)) mod k
B ← Patt[i][Last[i]]
shift ← Hs[B], Hs[B] ← 0
gap ← i + 1,    j ← m − 1
while j < n do
    s ← j ≫ 3,    sℓ ← j & 7
    B ← (B_t[s] ≫ (k − sℓ))| (B_t[s − 1] ≪ (sℓ + 1))
  if Hs[B] = 0 then
    i ← (sℓ + gap) mod k
    h ← Last[i],    q ← s
    while h ≥ 0 and
          Patt[i, h] = (B_t[q] & Mask[i, h]) do
      h ← h − 1,    q ← q − 1
    if h < 0 then
      b ← (q + 1) × k + i
      ρ ← Sk-Align(root, t, ρ, b)
      if ρ = b then Print(b)
    j ← j + shift
  else j ← j + Hs[B]
```

```
Huffman-Fed (p, m, t, n)
root ← Build-Sk-Tree(φ)
(Patt, Last, Mask) ← Preprocess (p, m)
(δ, λ) ← compute-Fed(Patt, Last, m)
ρ ← 0
s = m/8
while s < n do
  for each i in λ[B_t[s]] do
    h ← Last[i]
    q ← s + 1
    while h ≥ 0 and
      Patt[i][h] = B_t[q] &Mask[i][h] do
        h ← h − 1
        q ← q − 1
    if h < 0 then
      b ← (q + 1) × 8 + i
      ρ ← Sk-Align(root, t, ρ, b)
      if ρ = b then Print(b)
  do
    s ← s + δ[B_t[s + 1]]
  while s < n and δ[B_t[s]] ≠ 1
```

Figure 5. The Huffman-Hash-Matching algorithm and the Huffman-Fed algorithm for the compressed string matching problem on Huffman encoded texts. Parameters p and t stand for the Huffman compressed version of the pattern and text, respectively.

the desired time/space tradeoff by introducing a new parameter $K \leq k$, representing the number of bits taken into account for computing the shift advancement. Roughly speaking, only the K rightmost bits of the current window of the text are taken into account, reducing the total size of the tables to 2^K, at the price of possibly getting shorter shift advancements of the pattern than the ones that would have been obtained if the full length of blocks had been taken into consideration.

4.2 The Huffman-Fed Algorithm

The Fed algorithm [8] (Fast matching with Encoded DNA sequences) is a string matching algorithm specifically designed for matching DNA sequences compressed with a fixed-length encoding, requiring two bits for each character of the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. It combines a multi-pattern version of the Quick-Search algorithm [16] and a simplified version of the Commentz-Walter algorithm [3]. However, its strategy is general enough to be adapted to different encodings, including the Huffman one.

The resulting algorithm, which we call Huffman-Fed, makes use of a shift table δ and a hash table λ, both of size 2^k.

More specifically, the shift table δ is defined as follows. For $0 \leq i < k$ and $c \in \Sigma$, we first define the Quick-Search shift table $qs[i][c]$, by putting

$$qs[i][c] = \min\Big(\{m_i - 2 + 1\} \cup \{m_i - 2 + 1 - k \mid Patt[i][k] = c \text{ and } 1 \leq k \leq m_i - 2\}\Big).$$

Then, we put $\delta[c] = \min\{qs[i][c], 0 \leq i < k\}$, for $c \in \Sigma$.

The algorithm maintains also, for each block $B \in \{0 \dots 2^k - 1\}$, a linked list λ which is used to find candidate patterns. In particular, for each block $B \in \{0, \dots, 2^k - 1\}$, the entry $\lambda[B]$ is a set of indexes, defined by

$$\lambda[B] = \{0 \leq i < k \mid P[i][Last[i] - 1] = B\}.$$

In practical cases, each set in the table can be implemented as a linked list.

The code of the HUFFMAN-FED algorithm is presented in Figure 5.

The preprocessing phase of the algorithm consists in computing the shift table δ and the hash table λ defined above and, as in the HUFFMAN-HASH-MATCHING algorithm, it requires $\mathcal{O}(m + k2^{k+1})$-time complexity and $\mathcal{O}(m + 2^k)$ extra space.

During the searching phase, the algorithm performs a fast loop using the shift table δ to locate a candidate alignment of the pattern (lines `18-20`). In particular, the algorithm checks whether $\delta[B_t[s]] \neq 1$ and, if this is the case, it advances the shift by $\delta[B_t[s + 1]]$ positions to the right.

If $\delta[B_t[s]] = 1$ then, by definition of δ, we have $B_t[s] = P[i, Last[i] - 1]$, for some $0 \leq i < k$. In such a case the last byte of the current window is used as an index in the hash table and all patterns $Patt[i]$, such that $i \in \lambda[B_t[s]]$, are checked naively against the window (line `7`). For each alignment i found, the pattern $Patt[i]$ is compared block by block with the text.

As in the HUFFMAN-HASH-MATCHING algorithm, one has also to verify that the window is codeword aligned (line `14-17`).

The HUFFMAN-FED algorithm has a $\mathcal{O}(\lceil m/k \rceil n)$-time complexity and requires $\mathcal{O}(m + 2^k)$ extra space.

5 Experimental Results

In this section we present experimental results which allow to compare, in terms of running times and percentage of processed bits, the following algorithms:

- the HUFFMAN-KMP algorithm (HKMP) [15];
- the HUFFMAN-HASH-MATCHING algorithm (HHM), presented in Section 4.1;
- the HUFFMAN-FED algorithm (HFED), presented in Section 4.2.

In addition, we also tested an algorithm based on the *decompress-and-search* method (D&S for short) that makes use of the 3-*Hash* algorithm [13] for classical exact pattern matching, which is considered among the most efficient algorithms for the problem.

All algorithms have been implemented in the `C` programming language and have been compiled with the `GNU C Compiler`, using the optimization options `-O2 -fno-guess-branch-probability`. The tests have been performed on a 1.5 GHz PowerPC G4 and running times have been measured with a hardware cycle counter, available on modern CPUs.

We used the following input files:

- the English King James version of the "Bible" (3 Mb),
- the English "CIA World Fact Book" (2 Mb), and
- the Spanish novel "Don Quixote" by Cervantes (2 Mb).

The first two files are from the Canterbury Corpus [1], whereas the third one is from the Project Gutenberg [6].

For each input file, we have generated sets of 100 patterns of fixed length m, for m ranging in the set {4, 8, 16, 32, 64, 128, 256}, randomly extracted from the text. For each set of patterns we reported the mean over the running times of the 100 runs. The tables also show the minimum ($l_{\min}$) and maximum ($l_{\max}$) length in bits of the compressed patterns. For each set of patterns we have also computed the average number of processed bits.

In the following tables, running times are expressed in milliseconds whereas the number of processed bits is expressed as percentage of the total number of bits.

Running times						Processed bits			
m	$[l_{\min}, l_{\max}]$	HKMP	HHM	HFED	D&S	m	HKMP	HHM	HFED
4	$[17, 31]$	188.64	134.79	146.34	502.82	4	0.75	0.81	0.95
8	$[36, 53]$	185.77	105.59	112.98	491.87	8	0.76	0.68	0.77
16	$[79, 102]$	185.99	76.04	81.25	489.03	16	0.75	0.45	0.53
32	$[164, 204]$	184.23	65.78	70.31	487.74	32	0.75	0.42	0.48
64	$[336, 378]$	185.36	64.71	68.91	489.27	64	0.75	0.38	0.42
128	$[694, 768]$	187.73	72.00	77.11	487.31	128	0.75	0.34	0.37
256	$[1383, 1545]$	184.09	61.45	65.77	488.46	256	0.76	0.34	0.36

Experimental results on the Huffman encoded version of the King James version of the Bible

Running times						Processed bits			
m	$[l_{\min}, l_{\max}]$	HKMP	HHM	HFED	D&S	m	HKMP	HHM	HFED
4	$[18, 29]$	96.35	64.13	74.23	296.69	4	0.67	0.74	0.98
8	$[38, 53]$	95.50	49.38	56.47	289.82	8	0.66	0.55	0.68
16	$[77, 108]$	95.23	39.26	45.03	287.78	16	0.66	0.43	0.50
32	$[162, 207]$	94.74	33.55	38.53	287.34	32	0.65	0.35	0.40
64	$[327, 392]$	94.99	34.21	39.39	287.85	64	0.65	0.35	0.38
128	$[662, 761]$	94.42	28.54	32.92	287.51	128	0.64	0.29	0.31
256	$[1347, 1610]$	94.39	29.67	34.18	287.21	256	0.65	0.30	0.32

Experimental results on the Huffman encoded version of the CIA World Fact Book

Running times						Processed bits			
m	$[l_{\min}, l_{\max}]$	HKMP	HHM	HFED	D&S	m	HKMP	HHM	HFED
4	$[18, 35]$	122.25	87.44	95.44	308.56	4	0.75	0.81	0.95
8	$[37, 60]$	119.33	73.69	79.74	302.38	8	0.76	0.68	0.77
16	$[83, 140]$	120.35	45.99	49.67	300.53	16	0.75	0.45	0.53
32	$[171, 216]$	120.12	45.97	49.70	299.81	32	0.75	0.42	0.48
64	$[348, 525]$	119.20	41.86	45.26	300.90	64	0.75	0.38	0.42
128	$[712, 1068]$	117.55	37.80	40.83	299.78	128	0.75	0.34	0.37
256	$[1439, 1773]$	124.10	38.43	41.45	300.36	256	0.76	0.34	0.36

Experimental results on the Huffman encoded version of “Don Quixote”

The experimental results show that the Huffman-Hash-Matching and Huffman-Fed algorithms always achieve the best running times. In addition, the Huffman-Hash-Matching algorithm always obtains better results than the Huffman-Fed algorithm. In particular the running time of both algorithms decreases as the length of the pattern increases, since, as is reasonable to expect, the frequency of the patterns, and thus the number of skeleton tree verifications, is inversely proportional to m.

As expected, the Huffman-Kmp algorithm maintains the same performance independently of the pattern frequency. The gain of our algorithms compared to Huffman-Kmp is at least around 20 % and grows as the pattern frequency decreases and the pattern length increases.

Observe that, with the exception of very short patterns, the percentage of bits processed by our newly presented algorithms is always lower than that of the Huffman-Kmp algorithm and, in many cases, the gain is almost 50 %.

6 Conclusions

We have presented a new efficient approach to the compressed string matching problem on Huffman encoded texts, based on the Boyer-Moore strategy. Codeword alignment takes advantage of the skeleton tree data structure, which allows to skip over a significant percentage of the bits. In particular, we have presented adaptations of the Binary-Hash-Matching and Fed algorithms for searching Huffman encoded texts. The experimental results show that our algorithms exhibit a sublinear behavior on the average and in most cases are able to skip more than 50 % of the total number of bits in the encoded text.

References

1. R. Arnold and T. Bell: *A corpus for the evaluation of lossless compression algorithms*, in DCC '97: Proceedings of the Conference on Data Compression, Washington, DC, USA, 1997, IEEE Computer Society, p. 201, `http://corpus.canterbury.ac.nz/`.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* 20(10) 1977, pp. 762–772.
3. B. Commentz-Walter: *A string matching algorithm fast on the average*, in Automata, Languages and Programming, 6th Colloquium, Graz, Austria, July 16-20, 1979, Proceedings, vol. 71 of Lecture Notes in Computer Science, 1979, pp. 118–132.
4. S. Faro and T. Lecroq: *Efficient pattern matching on binary strings*, in Current Trends in Theory and Practice of Computer Science (SOFSEM 09), 2009.
5. A. S. Fraenkel and S. T. Klein: *Bidirectional Huffman coding.* Comput. J., 33(4) 1990, pp. 296–307.
6. M. Hart: *Project Gutenberg*, `http://www.gutenberg.org/`.
7. D. A. Huffman: *A method for the construction of minimum redundancy codes.* Proc. I.R.E., 40 1951, pp. 1098–1101.
8. J. W. Kim, E. Kim, and K. Park: *Fast matching method for DNA sequences*, in Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, no. 4614 in Lecture Notes in Computer Science, 2007, pp. 271–281.
9. S. T. Klein: *Skeleton trees for the efficient decoding of Huffman encoded texts.* Inf. Retr., 3(1) 2000, pp. 7–23.
10. S. T. Klein and M. K. Ben-Nissan: *Accelerating Boyer Moore searches on binary texts*, in Implementation and Application of Automata, 12th International Conference, CIAA 2007, Praque, Czech Republic, July 16-18, 2007, Revised Selected Papers, J. Holub and J. Žďárek, eds., vol. 4783 of Lecture Notes in Computer Science, Springer-Verlag Berlin, 2007, pp. 130–143.
11. S. T. Klein and D. Shapira: *Pattern matching in Huffman encoded texts.* Inf. Process. Manage., 41(4) 2005, pp. 829–841.
12. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt: *Fast pattern matching in strings.* 6(1) 1977, pp. 323–350.
13. T. Lecroq: *Fast exact string matching algorithms.* Inf. Process. Lett., 102(6) 2007, pp. 229–235.
14. E. S. Schwartz and B. Kallick: *Generating a canonical prefix encoding.* Commun. ACM, 7(3) 1964, pp. 166–169.
15. D. Shapira and A. Daptardar: *Adapting the Knuth-Morris-Pratt algorithm for pattern matching in Huffman encoded texts.* Inf. Process. Manage., 42(2) 2006, pp. 429–439.
16. D. Sunday: *A very fast substring search algorithm.* Commun. ACM, 33(8) 1990, pp. 132–142.
17. S. Wu and U. Manber: *A fast algorithm for multi-pattern searching*, Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.

Finding Characteristic Substrings from Compressed Texts

Shunsuke Inenaga[1] and Hideo Bannai[2]

[1] Graduate School of Information Science and Electrical Engineering, Kyushu University, Japan
inenaga@c.csce.kyushu-u.ac.jp
[2] Department of Informatics, Kyushu University, Japan
bannai@inf.kyushu-u.ac.jp

Abstract. Text mining from large scaled data is of great importance in computer science. In this paper, we consider fundamental problems on text mining from compressed strings, i.e., computing a longest repeating substring, longest non-overlapping repeating substring, most frequent substring, and most frequent non-overlapping substring from a given compressed string. Also, we tackle the following novel problem: given a compressed text and compressed pattern, compute the representative of the equivalence class of the pattern w.r.t. the text. We present algorithms that solve the above problems in time polynomial in the size of input compressed strings. The compression scheme we consider is straight line program (SLP) which has exponential compression, and therefore our algorithms are more efficient than any algorithms that work on uncompressed strings.

1 Introduction

Text mining from large scaled data, e.g. biological and web data, is currently a very important topic in computer science [2]. The sheer size of the data makes the task difficult, and hence, it is convenient to store these data in a compressed form. The question is if it is possible to perform text mining operations on compressed strings.

In this paper, we consider the following fundamental text mining problems from compressed strings: given a compressed form $\mathcal{T}$ of a string T, compute (1) *a longest repeating substring* of T, (2) *a longest non-overlapping repeating substring* of T, (3) *a most frequent substring* of T, (4) *a most frequent non-overlapping substring* of T. We present algorithms to solve Problem 1 in $O(n^4 \log n)$ time and $O(n^3)$ space, Problem 2 in $O(n^6 \log n)$ time and $O(n^3)$ space, Problem 3 in $O(|\Sigma|^2 n^2)$ time and $O(n^2)$ space, and Problem 4 in $O(n^4 \log n)$ time and $O(n^3)$ space, where n is the size of $\mathcal{T}$ and Σ is the alphabet. We also consider the following problem: given compressed forms of two strings T and P, compute *the representative* of the string equivalence class [1] of P in T. We present an $O(n^4 \log n)$-time $O(n^3)$-space algorithm to solve this problem, where n denotes the total size of the two compressed representations. By computing the representative of the equivalence class, we can retrieve the left and right contexts of P in T. The equivalence class and its representative have played central roles in the discovery of characteristic expressions in classical Japanese poems [13], and in a blog spam detection algorithm [10]. To the best of our knowledge, our algorithms are the first to solve the above problems *without decompression*.

The text compression scheme we consider in this paper is *straight line program* (*SLP*). SLP is a context-free grammar in the Chomsky normal form and generates a single string. SLP is an abstract model of many kinds of text compression schemes, as

Shunsuke Inenaga, Hideo Bannai: Finding Characteristic Substrings from Compressed Texts, pp. 40–54.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

the resulting encoding of the LZ-family [14,15], run-length, multi-level pattern matching code [5], Sequitur [11] and so on, can quickly be transformed into SLPs [3,12][1]. The important property of SLP is that it allows *exponential* compression, i.e., the original (uncompressed) string length N can be exponentially large w.r.t. the corresponding SLP size n. Therefore, our algorithms are asymptotically faster than *any* approaches that treat uncompressed strings.

Related Work. Little is known for text mining from compressed strings. Gąsieniec et al. [3] stated that it is possible to compute a succinct representation of all squares that appear in a given compressed string of size n in $O(n^6 \log^5 N)$ time. Matsubara et al. [8] presented an $O(n^4)$-time $O(n^2)$-space algorithm to compute a succinct representation of all maximal palindromes from a given SLP-compressed string.

2 Preliminaries

2.1 Notations

For any set U of integers and an integer k, we denote $U \oplus k = \{i + k \mid i \in U\}$ and $U \ominus k = \{i - k \mid i \in U\}$.

Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. The length of a string T is denoted by $|T|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a string $T = XYZ$, X, Y and Z are called a *prefix*, *substring*, and *suffix* of T, respectively. The i-th character of a string T is denoted by $T[i]$ for $1 \leq i \leq |T|$, and the substring of a string T that begins at position i and ends at position j is denoted by $T[i : j]$ for $1 \leq i \leq j \leq |T|$. For convenience, let $T[i : j] = \varepsilon$ if $j < i$.

For any strings T and P, let $Occ(T, P)$ be the set of occurrences of P in T, i.e.,

$$Occ(T, P) = \{k > 0 \mid T[k : k + |P| - 1] = P\}.$$

For any two strings T and S, let $LCPref(T, S)$ and $LCSuf(T, S)$ denote the length of the *longest common prefix and suffix* of T and S, respectively.

For any string T, let $\overline{T}$ denote the reversed string of T, i.e., $\overline{T} = T[|T|] \cdots T[2]T[1]$.

A *period* of a string T is an integer p $(1 \leq p \leq |T|)$ such that $T[i] = T[i + p]$ for any $i = 1, 2, \ldots, |T| - p$.

For any two strings T and S, we define the set $OL(T, S)$ as follows:

$$OL(T, S) = \{k > 0 \mid T[|T| - k + 1 : |T|] = S[1 : k]\}$$

Namely, $k \in OL(T, S)$ iff the suffix of T of length k $(k > 0)$ matches the prefix of S of length k.

2.2 Text Compression by Straight Line Programs

In this paper, we treat strings described in terms of *straight line programs* (*SLPs*). A straight line program $\mathcal{T}$ is a sequence of assignments such that

$$X_1 = expr_1, X_2 = expr_2, \ldots, X_n = expr_n,$$

where each X_i is a variable and each $expr_i$ is an expression either

– $expr_i = a \quad (a \in \Sigma)$, or
– $expr_i = X_\ell X_r \quad (\ell, r < i)$.

[1] An important exception is compression schemes based on the Burrows-Wheeler transform.

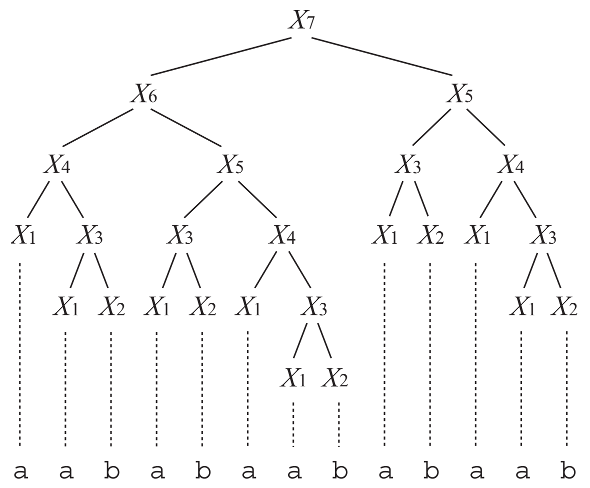


Figure 1. The derivation tree of an SLP that generates string `aababaababaab`.

Denote by T the string derived from the last variable X_n of the program $\mathcal{T}$. The *size* of the program $\mathcal{T}$ is the number n of assignments in $\mathcal{T}$. We remark that $|T| = O(2^n)$. Figure 1 shows the derivation tree of an SLP which derives string `aababaababaab`.

When it is not confusing, we identify a variable X_i with the string derived from X_i. Then, $|X_i|$ denotes the length of the string derived from X_i.

For any variable X_i of $\mathcal{T}$ with $1 \le i \le n$, we define $\overline{X_i}$ as follows:

$$\overline{X_i} = \begin{cases} a & \text{if } X_i = a \ \ (a \in \Sigma), \\ \overline{X_r}\ \overline{X_\ell} & \text{if } X_i = X_\ell X_r \ \ (\ell, r < i). \end{cases}$$

Let $\overline{\mathcal{T}}$ be the SLP consisting of variables $\overline{X_i}$ for $1 \le i \le n$. It is shown in [8] that SLP $\overline{\mathcal{T}}$ derives string $\overline{T}$ and $\overline{\mathcal{T}}$ can be easily computed from SLP $\mathcal{T}$ in $O(n)$ time.

Pattern Matching on SLP-compressed Strings. Here we briefly recall some existing results on pattern matching for SLP-compressed strings.

Let Y_j denote a variable of SLP $\mathcal{P}$ of size m that generates string P, for $1 \le j \le m$.

For any SLP variables $X_i = X_\ell X_r$ and Y_j, we define the set $Occ^{\triangle}(X_i, Y_j)$ of all occurrences of Y_j that cover or touch the boundary between X_ℓ and X_r, namely,

$$Occ^{\triangle}(X_i, Y_j) = \{s > 0 \mid X_i[s : s + |Y_j| - 1] = Y_j, |X_\ell| - |Y_j| + 1 \le s \le |X_\ell|\}.$$

Lemma 1 ([9]). *For any SLP variables X_i and Y_j, $Occ^{\triangle}(X_i, Y_j)$ forms a single arithmetic progression. Moreover, if $|Occ^{\triangle}(X_i, Y_j)| \ge 3$, then the common difference of the arithmetic progression is the smallest period of Y_j.*

Lemma 2 ([9]). *A membership query to $Occ(T, P)$ can be answered in $O(n)$ time, provided that $Occ^{\triangle}(X_i, Y_m)$ is already computed for every X_i.*

Theorem 3 ([7]). *$Occ^{\triangle}(X_i, Y_j)$ can be computed in a total of $O(n^2 m)$ time and $O(nm)$ space for every $1 \le i \le n$ and $1 \le j \le m$, using a DP table App of size $n \times m$ such that $App[i, j]$ stores an arithmetic progression for $Occ^{\triangle}(X_i, Y_j)$.*

Computing Overlaps of SLP-compressed Strings. The set of overlaps between two variables can be efficiently computed as follows (assume $n > m$).

Lemma 4 ([4]). *For any SLP variables X_i and Y_j, $OL(X_i, Y_j)$ can be represented by $O(n)$ arithmetic progressions.*

Theorem 5 ([4]). *$OL(X_i, Y_j)$ can be computed in total of $O(n^4 \log n)$ time and $O(n^3)$ space for every $1 \le i \le n$ and $1 \le j \le m$.*

2.3 The *FM* function

For any two SLP variables X_i, Y_j and any integer k with $1 \leq k \leq |X_i|$, we define function $FM(X_i, Y_j, k)$ which returns the length of the longest common prefix of $X_i[k : |X_i|]$ and Y_j, that is,

$$FM(X_i, Y_j, k) = LCPref(X_i[k : |X_i|], Y_j).$$

Lemma 6 ([4]). *For any SLP variables X_i, Y_j and integer k, $FM(X_i, Y_j, k)$ can be computed in $O(n \log n)$ time, provided that $OL(X_{i'}, Y_{j'})$ is already computed for any $1 \leq i' \leq i$ and $1 \leq j' \leq j$.*

3 Computing Repeating Substrings from Compressed Text

3.1 Problems

A string P is said to be a *repeating substring* of a string T if $|Occ(T, P)| \geq 2$. A *longest repeating substring* of T is a longest string P of T such that $|Occ(T, P)| \geq 2$. A *most frequent substring* of T is a string P such that $|Occ(T, P)| \geq |Occ(T, Q)|$ for any other string Q.

Any two occurrences $k_1, k_2 \in Occ(T, P)$ with $k_1 < k_2$ are said to *overlap* if $k_1 + |P| \geq k_2$. Otherwise, they are said to *non-overlap*. A *longest non-overlapping repeating substring* of T is a longest string P such that there exist at least two non-overlapping occurrences in $Occ(T, P)$. A *most frequent non-overlapping substring* of T is a string P such that it has the most non-overlapping occurrences in T.

In this section we consider the following problems.

Problem 1 (Computing longest repeating substring from SLP). Given an SLP $\mathcal{T}$ that derives a string T, compute two occurrences of a longest repeating substring P of T and its length $|P|$.

Problem 2 (Computing longest non-overlapping repeating substring from SLP). Given an SLP $\mathcal{T}$ that derives a string T, compute two non-overlapping occurrences of a longest non-overlapping repeating substring P of T and its length $|P|$.

Problem 3 (Computing most frequent substring from SLP). Given an SLP $\mathcal{T}$ that derives a string T, compute a most frequent substring P of T and a representation and the cardinality of $Occ(T, P)$.

Problem 4 (Computing most frequent non-overlapping substring from SLP). Given an SLP $\mathcal{T}$ that derives a string T, compute a most frequent non-overlapping substring P of T, and a representation and the number of non-overlapping occurrences of P in T.

By "representation" in Problems 3 and 4 we mean some succinct (polynomial-sized) representation of the sets. This is due to the fact that the cardinality of the sets can be exponentially large w.r.t. the input size.

In what follows, let n be the size of SLP $\mathcal{T}$ and let X_i denote each variable of $\mathcal{T}$ for $1 \leq i \leq n$.

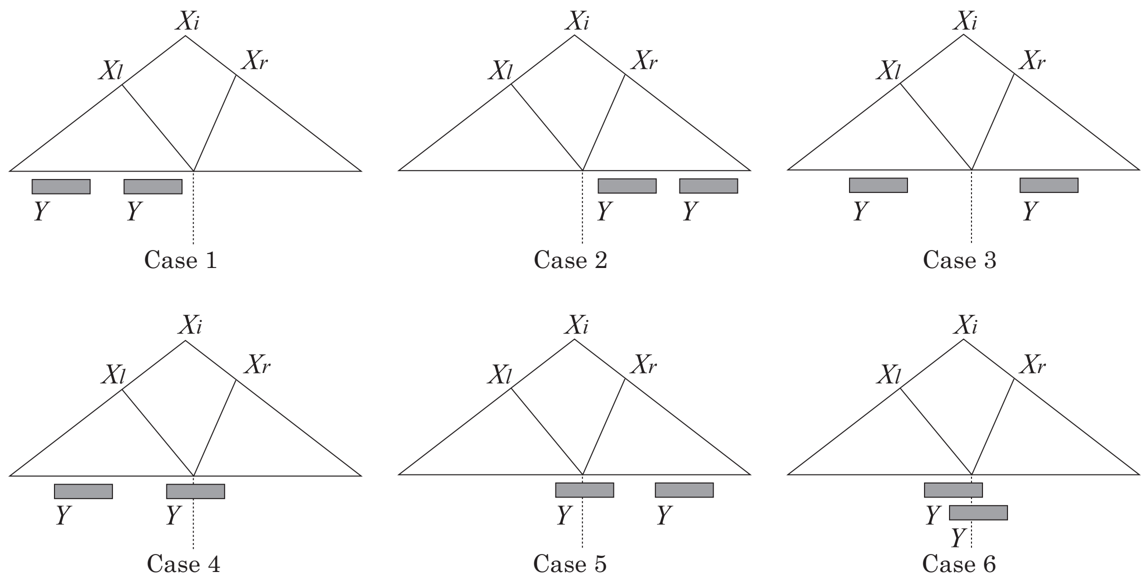


Figure 2. Illustration for Observation 7. The six cases for two distinct occurrences of a substring Y of X_i.

3.2 Solution to Problem 1

A key observation for solving Problem 1 is the following.

Observation 7. *For any SLP variable $X_i = X_\ell X_r$ and any string Y, assume that $|Occ(X_i, Y)| \geq 2$. Any two occurrences $k_1, k_2 \in Occ(X_i, Y)$ with $k_1 < k_2$ fall into one of the six following cases (see also Figure 2):*

1. $k_1, k_2 \in Occ(X_\ell, Y)$.
2. $k_1, k_2 \in Occ(X_r, Y)$.
3. $k_1 \in Occ(X_\ell, Y)$ *and* $k_2 \in Occ(X_r, Y)$.
4. $k_1 \in Occ(X_\ell, Y)$ *and* $k_2 \in Occ^\triangle(X_i, Y)$.
5. $k_1 \in Occ^\triangle(X_i, Y)$ *and* $k_2 \in Occ(X_r, Y)$.
6. $k_1, k_2 \in Occ^\triangle(X_i, Y)$.

Observation 7 implies that a longest repeating substring of T can be obtained by computing a longest repeating substring for every SLP variable X_i in each case. Case 1 and Case 2 are symmetric, and these two cases actually belong to one of the above cases with respect to variables X_ℓ and X_r, respectively. Since Case 4 and Case 5 are symmetric, we focus on Case 3, Case 4, and Case 6 in the sequel.

The following lemma is useful to deal with Case 3.

Lemma 8 ([8]). *For every pair of SLP variables X_i and X_j, we can compute the length of a longest common substring of X_i and X_j plus its occurrence position in X_i and X_j in $O(n^2 \log n)$ time, provided that $OL(X_{i'}, X_{j'})$ is already computed for any $1 \leq i' \leq i$ and $1 \leq j' \leq j$.*

Now we have the next lemma.

Lemma 9. *For every SLP variable X_i, two occurrences and the length of a longest repeating substring in Case 3 can be computed in $O(n^2 \log n)$ time.*

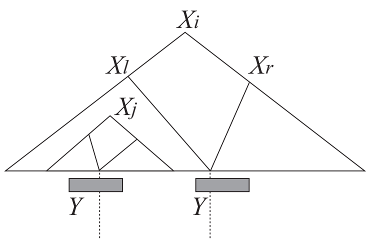


Figure 3. Illustration for Observation 10.

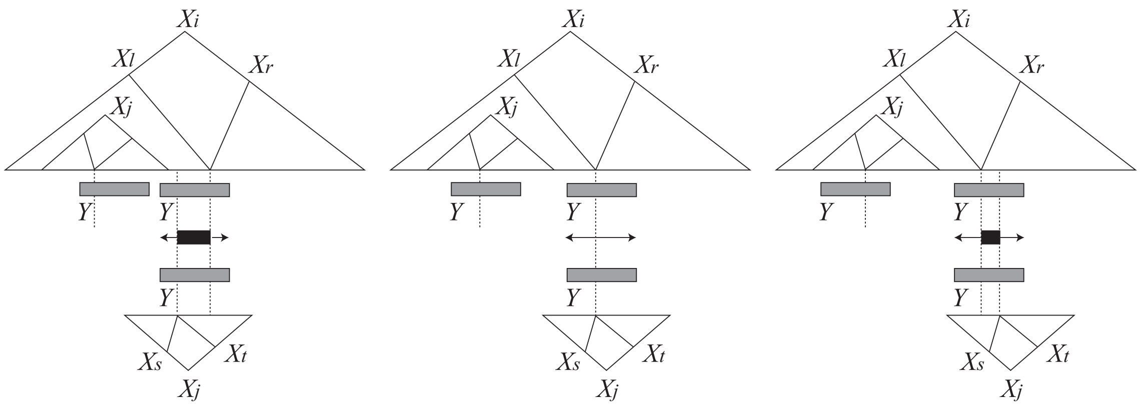


Figure 4. Illustration for Observation 11. The black rectangles of the left and right diagrams are an element of $OL(X_\ell, X_t)$ and $OL(X_s, X_r)$, respectively.

Proof. Note that a longest repeating substring of $X_i = X_\ell X_r$ in Case 3 is indeed a longest common substring of X_ℓ and X_r. Hence the lemma holds by Lemma 8. □

Next we consider Case 4. A key observation is the following:

Observation 10. *For the first occurrence of Y in Case 4 of Observation 7, there always exists a variable X_j such that X_j is a descendant of X_ℓ or X_ℓ itself, and the first occurrence of Y touches or covers the boundary of X_j (see also Figure 3).*

For any SLP variables $X_i = X_\ell X_r$ and $X_j = X_s X_t$ and any non-negative integer $z \in OL(X_\ell, X_t) \cup \{0\}$, let h_1 and h_2 be the maximum non-negative integers such that

$$X_i[|X_\ell| - z - h_1 + 1 : |X_\ell| + h_2] = X_j[|X_s| - h_1 + 1 : |X_s| + z + h_2].$$

That is, $h_1 = \mathit{LCSuf}(X_\ell[1 : |X_\ell| - z], X_s)$ and $h_2 = \mathit{LCPref}(X_r, X_t[z+1 : |X_t|])$. Let

$$\mathit{Ext}_{X_i,X_j}(z) = \begin{cases} z + h_1 + h_2 & \text{if } X_i = X_\ell X_r \text{ and } X_j = X_s X_t, \\ z & \text{if } X_i \text{ or } X_j \text{ is constant.} \end{cases}$$

For a set S of integers, we define $\mathit{Ext}_{X_i,X_j}(S) = \{\mathit{Ext}_{X_i,X_j}(z) \mid z \in S\}$. $\mathit{Ext}_{X_j,X_i}(z)$ and $\mathit{Ext}_{X_j,X_i}(S)$ are defined similarly.

Observation 11. *The length of a longest repeating substring of Case 4 is equal to the maximum element of*

$$\bigcup_{X_j \in A} (\mathit{Ext}_{X_i,X_j}(OL(X_\ell, X_t)) \cup \mathit{Ext}_{X_i,X_j}(0) \cup \mathit{Ext}_{X_j,X_i}(OL(X_s, X_r))) \qquad (1)$$

where $A = \{X_j = X_s X_t \mid X_j$ is a descendant of X_ℓ or $j = \ell\}$. (See also Figure 4.)

Now we have the following lemma.

Lemma 12. *For every SLP variable X_i, two occurrences and the length of a longest repeating substring in Case 4 can be computed in $O(n^3 \log n)$ time, provided that $OL(X_{i'}, X_{j'})$ and $Occ^{\triangle}(X_{i'}, X_{j'})$ are already computed for any $1 \leq i' \leq n$ and $1 \leq j' \leq n$.*

Proof. Let $X_i = X_\ell X_r$. It was proven by Lemma 4 of [8] that, for any variable $X_j = X_s X_t$ mentioned in Observation 10, $\max(Ext_{X_i,X_j}(OL(X_\ell, X_t)) \cup Ext_{X_i,X_j}(0) \cup Ext_{X_j,X_i}(OL(X_s, X_r)))$ can be computed in $O(n^2 \log n)$ time, provided that $OL(X_{i'}, X_{j'})$ is already computed for any $1 \leq i' \leq n$ and $1 \leq j' \leq n$. Recall Observation 11. Since the number of distinct descendants of any variable X_i is at most $n-1$, we can compute Equation (1) in $O(n^3 \log n)$ time. Let X_h be the variable that gives the maximum value of Equation (1). We can retrieve one position of $Occ(X_\ell, X_h)$ in $O(n^2)$ time from $Occ^{\triangle}(X_1, X_h), \ldots, Occ^{\triangle}(X_\ell, X_h)$. Then it is easy to compute two occurrences of the longest repeating substring in constant time. □

It is not difficult to see that Case 6 can be solved in a similar way to Case 4.

By Lemma 9, Lemma 12, Theorem 3, and Theorem 5, we obtain the main result of this subsection.

Theorem 13. *Problem 1 can be solved in $O(n^4 \log n)$ time and $O(n^3)$ space.*

3.3 Solution to Problem 2

Here we show how to find a longest non-overlapping repeating substring from a given SLP. The algorithm is based on the one proposed in Section 3.2. Below, we give our strategy to find a maximal non-overlapping repeating substring from the overlapping repeating substring found by the algorithm of Section 3.2.

An obvious fact is that Case 3 of Observation 7 only deals with a non-overlapping repeating substring. Hence we focus on Case 4. The other cases are solved similarly.

Lemma 14 ([6]). *Let $\mathcal{T}$ be any SLP of size n that generates string T. For any substring Y of T, it takes $O(n)$ time construct a new SLP of size $O(n)$ which generates the substring Y.*

Lemma 15. *Let k_1 and k_2 be any overlapping occurrences of string Y in string X such that $k_1 < k_2$. Let p be the smallest period of Y. Then, a longest non-overlapping repeating substring in $X[k_1 : k_2 + |Y| - 1]$ is $Y[1 : k_2 - 1 + pl]$, where $l = \lfloor(|Y| - k_2 + k_1)/2p\rfloor$.*

Proof. The length of the overlap is $|Y| - k_2 + k_1$. $Y[1 : p]$ appears in $\lfloor(|Y| - k_2 + k_1)/p\rfloor$ times in the overlap part $Y[k_2 : |Y| + k_1]$. Hence the lemma holds. □

Lemma 16. *For every SLP variable X_i, two occurrences and the length of a longest non-overlapping repeating substring in Case 4 of Observation 7 can be computed in $O(n^5 \log n)$ time and $O(n^3)$ space, provided that $OL(X_{i'}, X_{j'})$ is already computed for any $1 \leq i' \leq n$ and $1 \leq j' \leq n$.*

Proof. The proof is based on the proof of Lemma 8 of [8].

We consider $Ext_{X_i,X_j}(OL(X_\ell, X_t))$ of Observation 11. For each descendant X_j of X_i, it is sufficient to consider the leftmost occurrence γ of X_j in the derivation tree of X_i, since no other occurrences of X_j can correspond to longer non-overlapping repeating substring than the leftmost occurrence.

Let $\langle a, d, q\rangle$ denote any of the $O(n)$ arithmetic progressions in $OL(X_\ell, X_t)$, where a denotes the minimal element, d does the common difference and q does the number of elements of the progression. That is, $\langle a, d, q\rangle = \{a + (i-1)d \mid 1 \le i \le q\}$.

Assume $q > 1$ and $a < d$, as the case where $q = 1$ or $a = d$ is easier to show. Let $u = X_t[1 : a]$ and $v = X_t[a+1 : d]$.

Let e_1, e_2 be the largest integer such that $X_i[|X_\ell| - e_2 + 1 : |X_\ell| + e_1]$ is the longest substring of X_i that contains $X_i[|X_\ell| - d + 1 : |X_\ell|]$ and has a period d. Similarly, let e_3, e_4 be the largest integer such that $X_j[|X_s| - e_4 + 1 : |X_s| + e_3]$ is the longest substring of X_j that contains $X_j[|X_s|+1 : |X_s|+d]$ and has a period d. More formally,

$$\begin{aligned}
e_1 &= \mathit{LCPref}(X_r, (vu)^*) = \begin{cases} \mathit{FM}(X_t, X_r, a{+}1) & \text{if } \mathit{FM}(X_t, X_r, a{+}1) < d, \\ \mathit{FM}(X_r, X_r, d+1) + d & \text{otherwise,} \end{cases} \\
e_2 &= \mathit{LCSuf}(X_\ell, (vu)^*) = \mathit{FM}(\overline{X_\ell}, \overline{X_\ell}, d+1) + d, \\
e_3 &= \mathit{LCPref}(X_t, (uv)^*) = \mathit{FM}(X_t, X_t, d+1) + d, \\
e_4 &= \mathit{LCSuf}(X_s, (uv)^*) = \begin{cases} \mathit{FM}(\overline{X_\ell}, \overline{X_s}, a{+}1) & \text{if } \mathit{FM}(\overline{X_\ell}, \overline{X_s}, a{+}1) < d, \\ \mathit{FM}(\overline{X_s}, \overline{X_s}, d+1) + d & \text{otherwise,} \end{cases}
\end{aligned}$$

where $(vu)^*$ and $(uv)^*$ denote infinite repetitions of vu and uv, respectively.

Let $k \in \langle a, d, q\rangle$. We categorize $\mathit{Ext}_{X_i,X_j}(k)$ depending on the value of k, as follows.

(1) When $k < \min\{e_3 - e_1, e_2 - e_4\}$. If $k - d \in \langle a, d, q\rangle$, then we have $\mathit{Ext}_{X_i,X_j}(k) = \mathit{Ext}_{X_i,X_j}(k-d) + d$.
(2) When $k > \max\{e_3 - e_1, e_2 - e_4\}$. If $k + d \in \langle a, d, q\rangle$, then we have $\mathit{Ext}_{X_i,X_j}(k) = \mathit{Ext}_{X_i,X_j}(k+d) + d$.
(3) When $\min\{e_3 - e_1, e_2 - e_4\} < k < \max\{e_3 - e_1, e_2 - e_4\}$. In this case we have $\mathit{Ext}_{X_i,X_j}(k) = \min\{e_1 + e_2, e_3 + e_4\}$ for any k with $\min\{e_3 - e_1, e_2 - e_4\} < k < \max\{e_3 - e_1, e_2 - e_4\}$.
(4) When $k = e_3 - e_1$. In this case we have

$$\begin{aligned}
\mathit{Ext}_{X_i,X_j}(k) &= k + \min\{e_2 - k, e_4\} + \mathit{LCPref}(X_t[k+1 : |X_t|], X_r) \\
&= k + \min\{e_2 - k, e_4\} + \mathit{FM}(X_t, X_r, k+1).
\end{aligned}$$

(5) When $k = e_2 - e_4$. In this case we have

$$\begin{aligned}
\mathit{Ext}_{X_i,X_j}(k) &= k + \mathit{LCSuf}(X_\ell[1 : |X_\ell| - k], X_s) + \min\{e_1, e_3 - k\} \\
&= k + \mathit{FM}(\overline{X_\ell}, \overline{X_s}, k+1) + \min\{e_1, e_3 - k\}.
\end{aligned}$$

(6) When $k = e_3 - e_1 = e_2 - e_4$. In this case we have

$$\begin{aligned}
\mathit{Ext}_{X_i,X_j}(k) &= k + \mathit{LCSuf}(X_\ell[1 : |X_\ell| - k], X_s) + \mathit{LCPref}(X_t[k+1 : |X_t|], X_r) \\
&= k + \mathit{FM}(\overline{X_\ell}, \overline{X_s}, k+1) + \mathit{FM}(X_t, X_r, k+1).
\end{aligned}$$

Consider Case (1). For any $k - d, k \in \langle a, d, q\rangle$, if the occurrences of the substring that corresponds to $\mathit{Ext}_{X_i,X_j}(k-d)$ overlap, then the substring that corresponds to $\mathit{Ext}_{X_i,X_j}(k)$ also overlap. Note also that these substrings have the same ending position b in X_i, and have the same beginning position c in X_j. Since a membership query to the triple $\langle a, d, q\rangle$ can be answered in constant time, we can find the largest

k belonging to Case (1) in constant time. It is not difficult to see that d is the smallest period of $X_i[c+\gamma-1:b]$. By Lemma 15, we can compute the length of a longest non-overlapping repeating substring in $X_i[c+\gamma-1:b]$ in constant time, provided that e_1, e_2, e_3, e_4 are already computed. Similar arguments hold for Cases (2) and (3).

Consider Case (4). Let Z be the unique substring that corresponds to $Ext_{X_i,X_j}(k)$. Let x and y be the integers such that $x < y$ and $X_h[x : x+|Z|-1] = X_i[y : y+|Z|-1] = Z$. If $x+\gamma+|Z|-2 \geq y$, then we construct a new SLP $\mathcal{P}$ that generates string $P = X_i[x+\gamma-1 : y+|Z|-1]$ and compute its smallest period. It is clear that $|P| - \max(OL(P,P) - \{|P|\})$ is the smallest period of P. By Theorem 5 and Lemma 14, the length of a longest non-overlapping repeating substring in P can be computed in $O(n^4 \log n)$ time with $O(n^3)$ space. Similar arguments hold for Cases (5) and (6).

The values of e_1, e_2, e_3, e_4 can be computed by at most 6 calls of the FM function, each taking $O(n \log n)$ time.

Since there is $O(n)$ descendants of X_i, the total cost is $O(n^5 \log n)$ time and $O(n^3)$ space. □

The next theorem follows from Lemma 16.

Theorem 17. *Problem 2 can be solved in $O(n^6 \log n)$ time and $O(n^3)$ space.*

3.4 Solution to Problem 3

Consider Problem 3 of computing a substring that most frequently occurs in T.

Lemma 18. *For any non-empty strings T and P, $Occ(T, P[1:i]) \supseteq Occ(T,P)$ for any integer $0 \leq i \leq |P|$.*

The above monotonicity lemma implies that the empty string ε is always the solution for Problem 3. To make the problem more interesting, we consider the following version of the problem where the output is a substring of length at least 2.

Problem 5 (Computing most frequent substring of length at least 2 from SLP). Given an SLP $\mathcal{T}$ that derives a string T, compute a string P such that $|P| \geq 2$ and $|Occ(T,P)| \geq |Occ(T,Q)|$ for any other string Q with $|Q| \geq 2$.

Again, by the monotonicity lemma, it is sufficient only to consider a substring of length 2 as a solution to Problem 5.

The next lemma is fundamental for solving Problem 5.

Lemma 19. *For any SLP variables X_i and Y_j with $|Y_j| \geq 2$, $|Occ(X_i, Y_j)|$ can be computed in $O(n)$ time, with $O(mn^2)$-time $O(n^2)$-space preprocessing.*

Proof. Let D be a dynamic programming table of size $n \times n$ such that $D[i,j]$ represents how many times X_j appears in the derivation tree of X_i. After initializing all entries with 0, the value of each $D[i,j]$ is computed by the following recurrence:

$$D[i,j] = \begin{cases} 1 & \text{if } i = j, \\ D[\ell,j] + D[r,j] & \text{if } X_i = X_\ell X_r. \end{cases}$$

Then we obtain

$$|Occ(X_i,Y_j)| = \sum_{h=1}^{i} (D[i,h] \times |Occ^{\triangle}(X_h,Y_j) - \{|X_L| - |Y_j| + 1 \mid X_h = X_L X_R\}|).$$

We remove an occurrence of Y_j that touches the boundary of X_h from $Occ^{\triangle}(X_h, Y_j)$, since this occurrence covers the boundary of some other variable (recall we have assumed $|Y_j| \geq 2$).

In the preprocessing stage, we compute $Occ^{\triangle}(X_i, Y_j)$ for each $1 \leq i \leq n$ and $1 \leq j \leq m$. It takes $O(n^2 m)$ time and $O(nm)$ space by Theorem 3. Then we compute the D-table in $O(n^2)$ time and space. Hence the preprocessing cost is $O(n^2 m)$ time and $O(n^2)$ space, assuming $n \geq m$.

By Lemma 1, the value of $|Occ^{\triangle}(X_i, Y_m) - \{|X_\ell| - |Y_m| + 1\}|$ is computable in constant time. Thus we can compute $|Occ(T, P)|$ in $O(n)$ time and space. ☐

Theorem 20. *Problem 5 can be solved in $O(|\Sigma|^2 n^2)$ time and $O(n^2)$ space.*

Proof. Let n be the size of SLP $\mathcal{T}$ and X_i denote its variable for $1 \leq i \leq n$. For each pair of variables $X_h = a$ and $X_j = b$ such that $a, b \in \Sigma$, we construct a new SLP $\mathcal{S}_{h,j} : Y_{h,j} = X_h X_j, X_h = a, X_j = b$. Then for each $\mathcal{S}_{h,j}$, we compute $Occ^{\triangle}(X_i, Y_{h,j})$ for every variable X_i of $\mathcal{T}$. Then a string $Y_{h,j}$ for which $|Occ(X_n, Y_{h,j})|$ is maximum is a solution to Problem 5.

Since the size of each new SLP $\mathcal{S}_{h,j}$ is constant, we can compute a DP table *App* that correspond to $\{Occ^{\triangle}(X_i, Y_{h,j})\}_{i=1}^{n}$ in $O(n^2)$ time and $O(n)$ space for each new SLP $\mathcal{S}_{h,j}$ by Theorem 3.

Due to Lemma 19, $|Occ(X_n, Y_{h,j})|$ can be computed in $O(n)$ time with $O(n^2)$ time and space preprocessing. Note that we can use the same D-table of Lemma 19 to compute $|Occ(X_n, Y_{h,j})|$ for every $Y_{h,j}$. On the other hand, we can discard the *App* table after $|Occ(X_n, Y_{h,j})|$ has been computed. Hence the total space requirement is $O(n^2)$. Since there are $O(|\Sigma|^2)$ new SLPs, it takes a total of $O(|\Sigma|^2 n^2)$ time. ☐

3.5 Solution to Problem 4

Here we consider Problem 4 of computing a substring that has the most non-overlapping occurrences in a string T, when given a corresponding SLP $\mathcal{T}$.

For any string P to overlap itself, P has to be of length at least 2. Again, to make the problem more interesting, we consider the following problem.

Problem 6 (Computing most frequent non-overlapping substring of length at least 2 from SLP). Given an SLP $\mathcal{T}$ that derives a string T, compute a string P such that $|P| \geq 2$ and no other string Q with $|Q| \geq 2$ has more non-overlapping occurrences in T than P does.

The next lemma is a non-overlapping version of Lemma 18.

Lemma 21. *For any non-empty strings T and P, if there are two non-overlapping occurrences of P in T, then there are at least two non-overlapping occurrences of $P[1:i]$ in T for any integer $0 \leq i \leq |P|$.*

Hence it suffices to consider a substring of length 2 as a solution to Problem 6.

We are now ready to show the following theorem.

Theorem 22. *Problem 6 can be solved in $O(n^4 \log n)$ time and $O(n^3)$ space.*

Proof. By Lemma 21, we consider a substring of length 2 as a solution to Problem 6.

For any string $P = ab$ with $a \neq b$, the set of its non-overlapping occurrences in T is identical to $Occ(T, P)$, since P cannot overlap with itself. Thus this case can be solved in the same way to Theorem 5.

Now consider any string $P = aa$. Although we will only show how to compute the number of its non-overlapping occurrences in T, it is easy to extend our method to computing a representation of its non-overlapping occurrences in T without increasing asymptotic complexities. For any $a \in \Sigma$ and any variable X_i, let $\alpha_{X_i,a} = LCPref(X_i, a^*)$ and $\beta_{X_i,a} = LCSuf(X_i, a^*)$ where a^* denotes an infinite repetition of a. Then, for any variable X_h, the number $H(X_h, aa)$ of non-overlapping occurrences of aa in X_h can be computed by the following recurrence:

$$H(X_h, aa) = \begin{cases} |Occ^{\triangle}(X_h, aa)| & \text{if } |X_h| \leq 2, \\ H(X_\ell, aa) + H(X_r, aa) \\ -\lfloor \frac{\beta_{X_\ell,a}}{2} \rfloor + \lfloor \frac{\beta_{X_\ell,a}+\beta_{X_r,a}}{2} \rfloor - \lfloor \frac{\alpha_{X_r,a}}{2} \rfloor & \text{if } X_h = X_\ell X_r \text{ and } |X_h| > 2. \end{cases}$$

Consider the case where $|X_h| \leq 2$. For each variable X_h, $|Occ^{\triangle}(X_h, aa)|$ can be computed in total of $O(n^2)$ time and space, in the same way as mentioned in the proof of Theorem 20.

Now consider the other case. For any variable X_i, it holds that

$$\alpha_{X_i,a} = \begin{cases} 0 & \text{if } X_i[1] \neq a, \\ 1 + FM(X_i, X_i, 2) & \text{if } X_i[1] = a. \end{cases}$$

We can check whether $X_i[1] = a$ or not in $O(n)$ time, since the height of the derivation tree of X_i is at most $n + 1$. Therefore, we can compute $\alpha_{X_i,a}$ in $O(n \log n)$ time by Lemma 6. Similar arguments hold for computing $\beta_{X_i,a}$. The number of patterns of the form $P = aa$ is $O(|\Sigma|)$. Thus we need $O(|\Sigma| n \log n)$ time for this case.

To compute the FM function in $O(n \log n)$ time, we need to compute $OL(X_i, X_j)$ for any variables X_i and X_j, taking $O(n^4 \log n)$ time and $O(n^3)$ space due to Theorem 5. Overall, it takes $O(n^4 \log n)$ time and $O(n^3)$ space to solve Problem 6. □

4 Computing the Representative of a Given Pattern from Compressed Text

4.1 Problem

In this subsection, we begin with recalling the equivalence relations on strings introduced by Blumer et al. [1], and then state their properties.

We define two equivalence relations w.r.t. a string T based on Occ as follows. The equivalence relations $\equiv_L$ and $\equiv_R$ w.r.t. a string $T \in \Sigma^*$ are defined by:

$$Y \equiv_L Z \Leftrightarrow Occ(T, Y) = Occ(T, Z), \text{ and}$$
$$Y \equiv_R Z \Leftrightarrow Occ(T, Y) \oplus (|Y| - 1) = Occ(T, Z) \oplus (|Z| - 1),$$

where Y and Z are any strings in Σ^*. The equivalence classes of a string Y with respect to $\equiv_L$ and $\equiv_R$ are denoted by $[Y]_{\equiv_L}$ and $[Y]_{\equiv_R}$, respectively.

For any substring Y of T, let $\overrightarrow{Y}$ and $\overleftarrow{Y}$ denote the unique longest member of $[Y]_{\equiv_L}$ and $[Y]_{\equiv_R}$, respectively. Let $\overleftrightarrow{Y} = \alpha Y \beta$ such that $\alpha, \beta \in \Sigma^*$ are the strings satisfying $\overleftarrow{Y} = \alpha Y$ and $\overrightarrow{Y} = Y \beta$, respectively. Intuitively, $\overleftrightarrow{Y} = \alpha Y \beta$ means that:

– Every time Y occurs in T, it is preceded by α and followed by β.

– Strings α and β are longest possible.

We define another equivalence relation $\equiv$ w.r.t. T by $Y \equiv Z \Leftrightarrow \overleftrightarrow{Y} = \overleftrightarrow{Z}$, and the equivalence class of a string Y is denoted by $[Y]_\equiv$. String $\overleftrightarrow{Y}$ is called the *representative* of the equivalence class $[Y]_\equiv$.

The problem to be tackled in this section is the following.

Problem 7. Given two SLPs $\mathcal{T}$ and $\mathcal{P}$ that derive strings T and P respectively, compute an occurrence position and the length of the representative $\overleftrightarrow{P}$ w.r.t. T, if $|Occ(T, P)| \geq 1$.

4.2 Solution to Problem 7

Let m be the size of SLP $\mathcal{P}$, and let Y_j denote each variable of SLP $\mathcal{P}$ for $1 \leq j \leq m$. Assume without loss of generality that $m \leq n$.

Theorem 23. *Problem 7 can be solved in $O(n^4 \log n)$ time and $O(n^3)$ space.*

Proof. We only show how to compute the length and an occurrence position of $\overleftarrow{P}$, since those of $\overrightarrow{P}$ and $\overleftrightarrow{P}$ can be computed similarly.

We first compute $Occ^\triangle(X_i, Y_m)$ for each X_i according to Theorem 3.

If the length of $\overleftarrow{P}$ is known, then it is trivial that an occurrence position of $\overleftarrow{P}$ can be computed from an occurrence position of $P = Y_m$ in $T = X_n$. An occurrence position of Y_m in X_n can be retrieved in $O(n^2)$ time using the m-th column of the App table that correspond to $Occ^\triangle(X_1, Y_m), \ldots, Occ^\triangle(X_n, Y_m)$. Hence we concentrate on how to compute the length of $\overleftarrow{P}$ in the sequel.

For any variables X_i and X_h, and integers $1 \leq k_i \leq |X_i|$ and $1 \leq k_h \leq |X_h|$, let $LE_{X_i,X_h}(k_i.k_h)$ be the largest integer $p \geq 1$ such that $X_i[k_i - p : k_i - 1] = X_h[k_h - p : k_h - 1]$. If such p does not exist, then let $LE_{X_i,X_h}(k_i, k_h) = 0$.

Depending on the cardinality of $Occ^\triangle(X_i, Y_m)$, we have the following cases:

1. When $|Occ^\triangle(X_i, Y_m)| = 0$ for every variable X_i. Then clearly there is no answer to Problem 7.
2. When $|Occ^\triangle(X_i, Y_m)| = 1$ for some variable X_i and $|Occ^\triangle(X_{i'}, Y_m)| = 0$ for every $X_{i'} \neq X_i$. In this case, we have that $|Occ(T, P)| = 1$. Hence, by definition, we have $|\overleftarrow{P}| = k + |P| - 1$ where $\{k\} = Occ(T, P)$.
3. When $0 \leq |Occ^\triangle(X_i, Y_m)| \leq 1$ for any variable X_i and there are at least two variables such that $|Occ^\triangle(X_i, Y_m)| = 1$. For any variable X_i such that $|Occ^\triangle(X_i, Y_m)| = 1$, let $\{k_i\} = Occ^\triangle(X_i, Y_m)$. Let A and B be the sets of variable pairs such that

$$A = \{(X_i, X_h) \mid LE_{X_i,X_h}(k_i, k_h) < \min\{k_i, k_h\}\},$$
$$B = \{(X_i, X_h) \mid LE_{X_i,X_h}(k_i, k_h) = \min\{k_i, k_h\}\}.$$

   See also Figure 5.

   (a) When $\min\{LE_{X_i,X_h}(k_i, k_h) \mid (X_i, X_h) \in A\} \leq \min\{LE_{X_i,X_h}(k_i, k_h) \mid (X_i, X_h) \in B\}$. In this case, we have

$$|\overleftarrow{P}| = |\overleftarrow{Y_m}| = \min\{LE_{X_i,X_h}(k_i, k_h) \mid (X_i, X_h) \in A\} + |Y_m|.$$

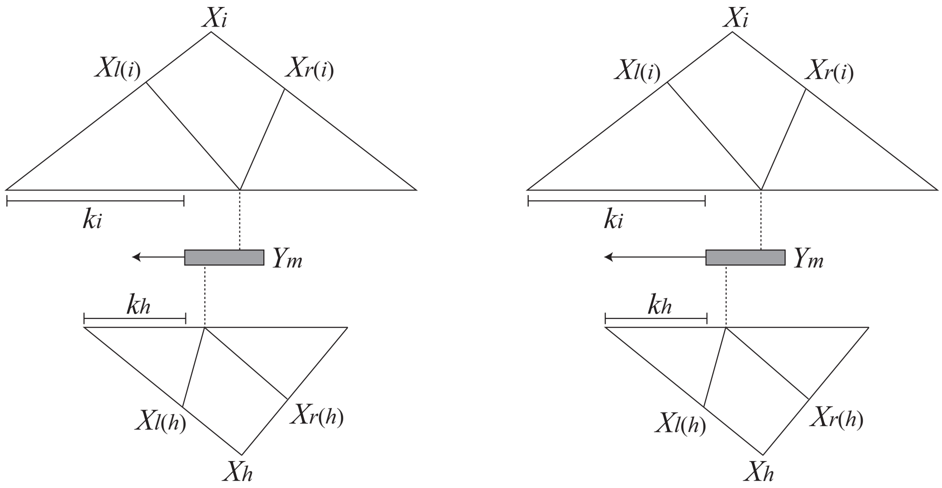


Figure 5. $LE_{X_i,X_h}(k_i,k_h) < \min\{k_i,k_h\}$ (left) and $LE_{X_i,X_h}(k_i,k_h) = \min\{k_i,k_h\}$ (right).

(b) When $\min\{LE_{X_i,X_h}(k_i,k_h) \mid (X_i,X_h) \in A\} > \min\{LE_{X_i,X_h}(k_i,k_h) \mid (X_i,X_h) \in B\}$.
For any pair of variables $X_i, X_h \in B$, assume w.l.o.g. that $k_i \geq k_h$. Let $X_h = X_{\ell(h)}X_{r(h)}$.
For any variable X_i such that $|Occ^{\triangle}(X_i,Y_m)| = 1$, we compute

$$G_i = Occ(X_i, X_{\ell(h)}) \cap (Occ^{\triangle}(X_i,Y_m) \ominus k_h).$$

Note that $|G_i| \leq 1$, since $|Occ^{\triangle}(X_i,Y_m)| = 1$. Let X_i, X_j be any variables such that $G_i = \{g_i\}$, $G_j = \{g_j\}$ and $g_i \leq g_j$. We compute $LE_{X_i,X_j}(g_i,g_j)$ until $LE_{X_i,X_j}(g_i,g_j) < \min\{g_i,g_j\}$ or X_i is a prefix of T (see Figure 6).
i. When $\min\{LE_{X_i,X_h}(Y_m,k_i,k_h) \mid (X_i,X_h) \in A\} \geq LE_{X_i,X_j}(g_i,g_j)$. In this case, $LE_{X_i,X_j}(g_i,g_j) + |Y_m|$ is a new candidate for $|\overleftarrow{Y_m}|$.
ii. When $\min\{LE_{X_i,X_h}(Y_m,k_i,k_h) \mid (X_i,X_h) \in A\} < LE_{X_i,X_j}(g_i,g_j)$. In this case, $LE_{X_i,X_j}(g_i,g_j) + |Y_m|$ is not a candidate for $|\overleftarrow{Y_m}|$.

4. When $0 \leq |Occ^{\triangle}(X_i,Y_m)| \leq 2$ for any $1 \leq i \leq n$. This case can be solved in a similar way to Case 3.
5. When $|Occ^{\triangle}(X_i,Y_m)| \geq 3$ for some $1 \leq i \leq n$. It follows from Lemma 1 that $Y_m = P = u^d v$ where $|u|$ is the smallest period of P, $d \geq 2$ is a positive integer, and v is a proper (possibly empty) prefix of u. It now holds that $|\overleftarrow{Y_m}| < |u| + |Y_m|$, since every occurrence of Y_m in $Occ^{\triangle}(X_i,Y_m)$ except for the first one is always preceded by u.
The length of $\overleftarrow{Y_m}$ can be computed as follows. See also Figure 7. For all variables $X_i = X_{\ell(i)}X_{r(i)}$ such that $|Occ^{\triangle}(X_i,Y_m)| \geq 1$, compute $FM(\overline{X_{\ell(i)}}, \overline{X_{\ell(i)}}, |u|+1)$. Then we have

$$|\overleftarrow{P}| = |\overleftarrow{Y_m}| = \min\{FM(\overline{X_{\ell(i)}}, \overline{X_{\ell(i)}}, |u|+1) \bmod |u| \mid |Occ^{\triangle}(X_i,Y_m)| \geq 1\} + |Y_m|.$$

Now we analyze the complexity. By Theorem 3, $Occ^{\triangle}(X_i,Y_m)$ can be computed in $O(n^3)$ time with $O(n^2)$ space. Moreover, the cardinality, and a membership query to each $Occ^{\triangle}(X_i,Y_m)$ is answered in constant time due to Lemma 1. Therefore, Cases 1 and 2 can be solved in constant time provided that $Occ^{\triangle}(X_i,Y_m)$ are pre-computed.

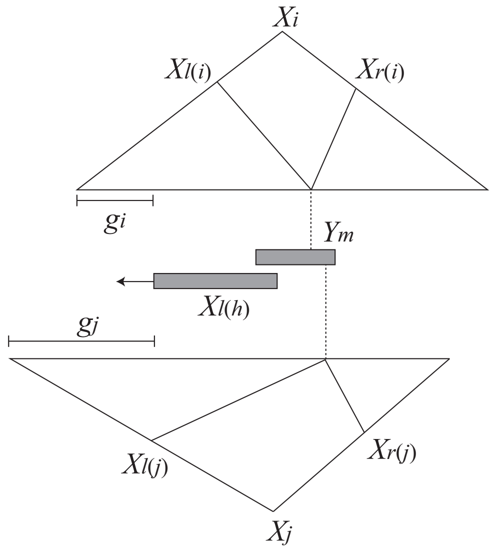


Figure 6. Compute $LE_{X_i,X_j}(g_i, g_j)$ until $LE_{X_i,X_j}(g_i, g_j) < \min\{g_i, g_j\}$ or X_i is a prefix of T.

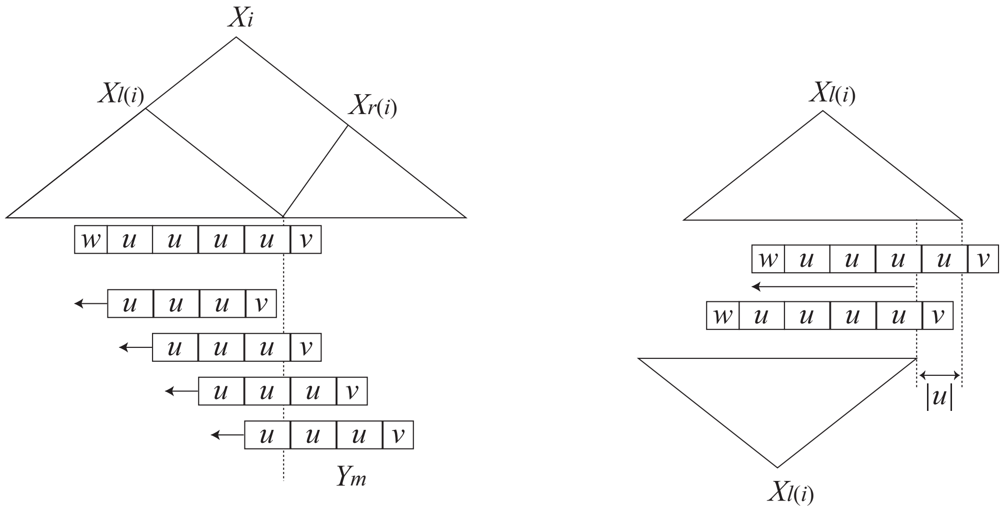


Figure 7. Illustration for Case 5. String w is a proper suffix of u such that every occurrence of $Y_m = u^d v$ is preceded by w (left). The length of w can be computed as $|w| = FM(\overline{X_{\ell(i)}}, \overline{X_{\ell(i)}}, |u| + 1) \bmod |u|$ (right).

Now consider Case 3. We compute the value of $LE_{X_i,X_h}(k_i, k_h)$ for all pairs of variables, whose number is $O(n^2)$. Since $\{k_i\} = Occ^{\triangle}(X_i, Y_m)$ and $\{k_h\} = Occ^{\triangle}(X_h, Y_m)$, it holds that

$$LE_{X_i,X_h}(k_i, k_h) = FM(\overline{X_i}, \overline{X_{\ell(h)}}, |X_i| - k_i - |X_{\ell(h)}| + k_j + 1) - |X_{\ell(h)}| + k_h. \quad (2)$$

It follows from Lemma 6 that Case 3a can be solved in $O(n^3 \log n)$ time. Now focus on Case 3b. For any variable X_i, G_i can be computed in $O(n)$ time, since $|Occ^{\triangle}(X_i, Y_m)| = 1$ and a membership query to $Occ(X_i, X_{\ell(h)})$ can be answered in $O(n)$ time due to Lemma 2. In each step of the loop, we compute the value of $LE_{X_i,X_j}(g_i, g_j) + |Y_m|$ for $O(n^2)$ pairs of variables. During this loop, the value of $LE_{X_i,X_j}(g_i, g_j) + |Y_m|$ is monotonically non-decreasing, and the size of G_i is monotonically non-increasing. Hence, the depth of the loop is bounded by n. Moreover, we have that

$$LE_{X_i,X_j}(g_i, g_j) = FM(\overline{X_j}, \overline{X_{\ell(i)}}, |X_i| - k_j - |X_{\ell(i)}| + k_i + 1) - |X_{\ell(i)}| + g_i. \quad (3)$$

By Equation (3) and Lemma 6, the total time cost for Case 3b is $O(n^4 \log n)$. Therefore, Case 3 is solvable in $O(n^4 \log n)$ time, and so is Case 4.

In Case 5 we call the *FM* function at most n times, and each call of the *FM* function takes $O(n \log n)$ time by Lemma 6. Hence Case 5 can be solved in $O(n^2 \log n)$ time.

Computing $OL(X_i, X_j)$ for each pair of variables X_i, X_j requires $O(n^4 \log n)$ time and $O(n^3)$ space due to Theorem 5. Overall, we conclude that Problem 7 can be solved in $O(n^4 \log n)$ time with $O(n^3)$ space. □

References

1. 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.
2. R. Feldman and J. Sanger: *The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*, Cambridge University Press, 2007.
3. L. Gąsieniec, M. Karpinski, W. Plandowski, and W. Rytter: *Efficient algorithms for Lempel-Ziv encoding*, in Proc. 5th Scandinavian Workshop on Algorithm Theory (SWAT'96), vol. 1097 of Lecture Notes in Computer Science, Springer-Verlag, 1996, pp. 392–403.
4. M. Karpinski, W. Rytter, and A. Shinohara: *An efficient pattern-matching algorithm for strings with short descriptions.* Nordic Journal of Computing, 4 1997, pp. 172–186.
5. J. Kieffer, E. Yang, G. Nelson, and P. Cosman: *Universal lossless compression via multilevel pattern matching.* IEEE Transactions on Information Theory, 46(4) 2000, pp. 1227–1245.
6. Y. Lifshits: *Solving classical string problems an compressed texts*, in Combinatorial and Algorithmic Foundations of Pattern and Association Discovery, no. 06201 in Dagstuhl Seminar Proceedings, 2006.
7. Y. Lifshits: *Processing compressed texts: A tractability border*, in Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM'07), vol. 4580 of Lecture Notes in Computer Science, Springer-Verlag, 2007, pp. 228–240.
8. W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto: *Efficient algorithms to compute compressed longest common substrings and compressed palindromes.* Theoretical Computer Science, 410(8–10) 2009, pp. 900–913.
9. M. Miyazaki, A. Shinohara, and M. Takeda: *An improved pattern matching algorithm for strings in terms of straight-line programs*, in Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM'97), vol. 1264 of Lecture Notes in Computer Science, Springer-Verlag, 1997, pp. 1–11.
10. K. Narisawa, H. Bannai, K. Hatano, and M. Takeda: *Unsupervised spam detection based on string alienness measures*, in Proc. 10th International Conference on Discovery Science (DS'07), vol. 4755 of Lecture Notes in Artificial Intelligence, 2007, pp. 161–172.
11. C. G. Nevill-Manning, I. H. Witten, and D. L. Maulsby: *Compression by induction of hierarchical grammars*, in Proc. Data Compression Conference '94 (DCC'94), IEEE Computer Society, 1994, pp. 244–253.
12. W. Rytter: *Grammar compression, LZ-encodings, and string algorithms with implicit input*, in Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP'04), vol. 3142 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 15–27.
13. M. Takeda, T. Matsumoto, T. Fukuda, and I. Nanri: *Discovering characteristic expressions in literary works.* Theoretical Computer Science, 292(2) 2003, pp. 525–546.
14. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, IT-23(3) 1977, pp. 337–349.
15. J. Ziv and A. Lempel: *Compression of individual sequences via variable-length coding.* IEEE Transactions on Information Theory, 24(5) 1978, pp. 530–536.

Delta Encoding in a Compressed Domain

Shmuel T. Klein and Moti Meir

Department of Computer Science
Bar Ilan University
Ramat Gan 52900, Israel
Tel: (972–3) 531 8865 Fax: (972–3) 736 0498
tomi@cs.biu.ac.il, moti.meir@gmail.com

Abstract. A delta compression algorithm is presented, working on an LZSS compressed reference file and an uncompressed version, and producing a delta file that can be used to reconstruct the version file directly in its compressed form. This has applications to accelerate data flow in network environments.

1 Introduction

This paper presents an algorithmic approach to work with highly correlated files in the compressed domain. The idea is to allow small changes to be reflected upon a reference file each time a newer file version becomes available. This ability is highly required by caching, versioning and additive backup systems.

The standard delta compression scheme [3,1,4,9,2] takes two files and outputs the difference between those files. Such a scheme tries to output a small amount of data which represents the difference between the two files in their uncompressed state. As a result, using the standard delta file scheme, one can reconstruct a version file by using a reference file and the delta file (see Figure 1.a). Our scheme, however, encodes a delta file that reflects the differences between the files in their *compressed state*. That is, using the delta file and the compressed reference file, the decoder outputs the *compressed* version file (See Figure 1.b), contrarily to standard delta schemes that would output a version file in uncompressed form. Our scheme uses an uncompressed version file and a compressed reference file as the inputs to the encoder, a scenario defined as *semi-compressed delta encoding* in [7], in contrast with the *fully-compressed* alternative treated in [6].

We provide a conceptual solution considering the fact that textual data is produced in uncompressed form and also by examination of the entire network route between the encoding part (usually a server) and the decoding part (usually a client), which includes intermediate network elements. The scheme considers the fact that most of the network's intermediate elements are indifferent to the data content and only wish to store and forward the most recent copy of the data. For such elements, having the ability to alter their cached copy without decompressing it first as needed when a regular delta encoding is used, presents a great advantage. Standard schemes producing uncompressed version files would require an encoding phase, that is, compressing the file in order to save storage space and network bandwidth. This imposes a penalty in terms of both CPU utilization and temporary storage space, which can be saved by directly dealing with a compressed output.

The proposed encoding algorithm has two inputs: the compressed reference file R_c, and the uncompressed version file V. The output of the encoder is a delta file Δ, which, together with R_c, is the input to the decoding algorithm that outputs the compressed version file V_c.

Shmuel T. Klein, Moti Meir: Delta Encoding in a Compressed Domain, pp. 55–64.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

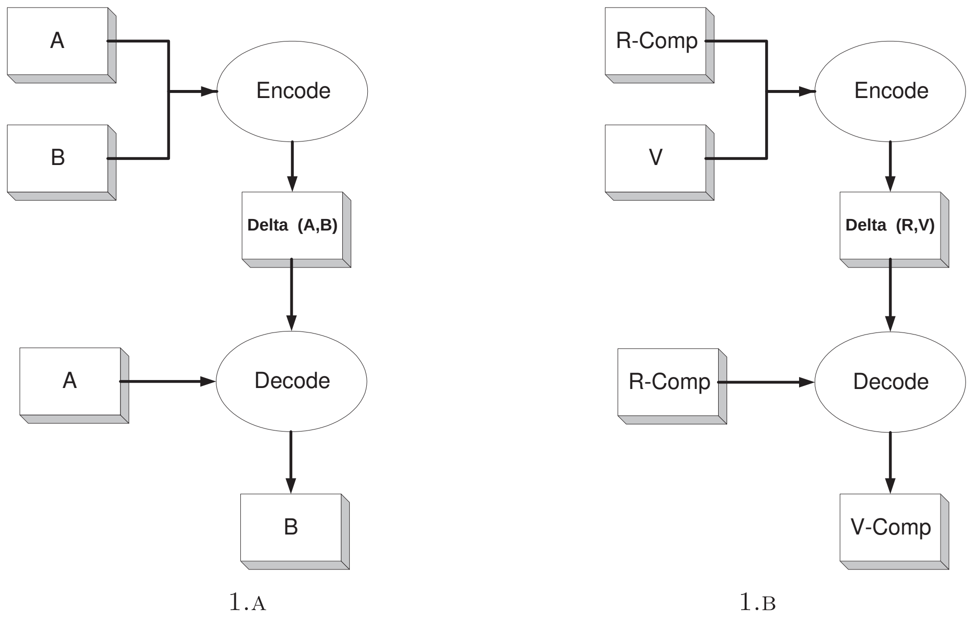


Figure 1. Schematic representation of delta encoding

The efficiency of our scheme is based on several assumptions regarding both version and reference files:

1. The files are highly correlated
2. The changes are local and sparse
3. The changes are very small compared to the size of V.

The algorithm is an extension of LZSS [5] encoding and uses an ordered hash table adapted from [10], to store previous occurrences of substrings. The parameters which control the encoding are:

– Window size — the length of the sliding window. This attribute defines the maximum valid size of the jumpbacks and thereby the maximum "memory" size of the hash table.
– Minimum Match — determines the minimum number of characters required to match, in order to create a back pointer to the previous text.
– Synch Chunk — the maximum number of characters that are affected by a single change. That is, the length of all changes are smaller than this number.

The output of the encoder is a set of `COPY`, `ADD`, `UPDATE` and `SPLIT` commands and a set of characters stored in Δ, acting as a set of control commands to a decoder for changing R_c. These commands are described in more detail in Section 4 below. The decoder uses Δ and R_c to build an updated compressed file equivalent to V_c. This is done without decoding R_c but rather by changing it while it is still compressed (see Figure 1.B)

Let $V_{cr}[i,j]$ be the substring of V_c with index in the real uncompressed form, that is, the indices i and j refer to the indices of V in their uncompressed form. For example, if the 100 first bytes of V are compressed to the first 20 elements of V_c, then we have $V_c[1,20] = Compress(V[1,100]) = V_{cr}[1,100]$. This provides a reverse mapping between the compressed domain and the uncompressed domain locations. We shall use the same notation when referring to just one character of V or R, that is $V_{cr}[50] = i$ is the location in V_c that corresponds to character $V[50]$.

2 The Encoding Algorithm

2.1 Overview

The algorithm encodes V and iteratively compares the results to R_c. During the processing, a sequence of characters and back pointers is generated, and checked for matches with R_c. If a mismatch is detected — denote the location of this *local mismatch point* by LMP — the original LZSS algorithm would output a file that will greatly differ from R_c. The result would be two encoded files, which were highly correlated when uncompressed, and when compared in the compressed domain loose their high resemblance. The current encoding fixes this problem by synchronizing both files. As a consequence there will be a *local synchronization point* (LSP) after which the output will match exactly the reference file, up to the next LMP. We assume here that the distance from LSP to LMP is at least Synch Chunk, otherwise the two changes are considered as one.

Several types of mismatches need to be addressed:

– one element is a back pointer while the other is a character;
– both elements are characters, but different ones;
– both elements are back pointers, but their copy length differs;
– both elements are back pointers, but their jump back length differs.

Maintaining a *Local Reconstructed Buffer* (LRB) in combination with the assumption of a limited change results in the ability to track the change. At each step, the algorithm checks whether a substitution, insertion or deletion of characters can explain the change, and continues according to the results by locating the LSPs in the uncompressed domain. When these points are found, the hash table is updated accordingly.

For example, consider the case of inserting a line of text into the version file. The LRB created by the reference instructions and the version data will match at a point in the version text which is beyond the inserted change. Since the change is assumed to be relatively small, the number of characters inserted is found by running a loop up to *synch chunk size*, trying to find this substring in the version file.

In order to compare the new version with the reference, we must be able to reconstruct the original substrings which might be represented by pointers in the reference file. This has led to the need of decoding the reference file in order to reconstruct the original data in the *change area* of V [7], which is the string starting one element prior to the actual change and ending at most Synch Chunk characters after the end of the change. The idea of running from right to left in R_c from elements prior to LMP, collecting the needed reference data characters, does not work well in most cases due to the fact that a back pointer in the encoded file can point to another back pointer, and so on, creating a chain. In addition to being expensive, this right to left decoding is not local. Since we want to maintain a local approach which greatly decreases memory needs, the semi compressed domain is exploited by using the reference characters and pointers to reconstruct the original data in the change area.

There is, however, one exception to the above. Since we use elements from R_c to reconstruct the reference data using the version data, care has to be taken in the case of self-pointers, i.e., when the copy *length* is larger than the *offset*. Indeed, the possibility of self-pointers is one of the major features of LZ77 schemes like LZSS, enabling the compression of variable-length repetitive strings; for example, a string of 50 a's can be compressed as a[offset = 1, length = 49]. The solution is to use the already reconstructed buffer as a reference for self pointers.

2.2 Substitution

This is the simplest case, e.g., a date field has been updated in the version file. The following steps are executed, referring to Figure 2:

1. find the actual change size by comparing characters from the LRB and V, bounded by the Synch Chunk parameter;
2. insert a new quarantine zone $[K, K+i]$ to the mismatch list;
3. output the relevant commands (split and update pointers) to Δ;
4. advance the R_c index to point to the location of the LSP, and also advance the version index to the same LSP.

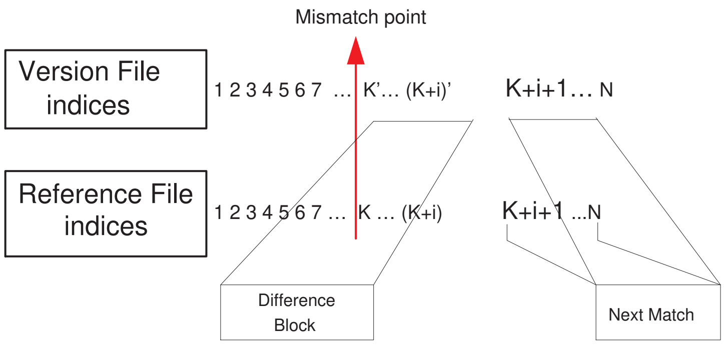


Figure 2. Schematic representation of substitution

The algorithm works because skipping over the change in both files and backwards updating V to the pre-change state, brings us to a point (LSP) where we just need to synchronize the hash tables. After this synchronization, the encoding process is the same, hence the output of the encoder are `COPY` and `UPDATE` commands up to the next mismatch. As we can see in Figure 2, jumping in both files as close as possible to index $K+i+1$ will bring us to the LSP. The reason that one does not always get exactly to $K+i+1$ is that the exact pointers from R_c are used and they are not split. However, breaking the back pointers to get to the exact spot in V is feasible and could in certain cases result in better compression.

2.3 Insertion

An inserted string may dramatically change the original LZSS output, as shown in Figure 3. V is scanned for an occurrence of the content of LRB inside the defined limited area, defined to start from $K-j$ up to $K+i$, where i is the Synch Chunk size and j is the maximum change size. The simplest way to deal with the insertion is to adjoin the inserted block to Δ with a single `ADD` and one `ADJUST` command. One could also consider using the hash table to add the newly inserted text as a sequence of pointers, which might improve compression.

In any case, a set of adjustment commands needs to be added to correct the back pointers of R_c. Each back pointer which points to the index in V prior to LMP has to be increased by the length of the insertion. If the increased length exceeds the maximum defined, the back pointer is split and the exceeding part is inserted as individual characters.

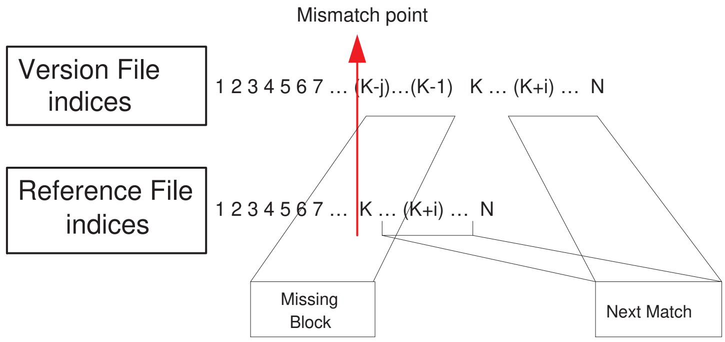


Figure 3. Schematic representation of insertion

2.4 Deletion

Deletion is similar to insertion, but here the LRB is scanned for an occurrence of a substring of V. The steps are, referring to Figure 4:

1. LRB $\longleftarrow$ $decode(R_c[K, K+i], V)$
2. search for $V[K+j, K+i]$ in LRB;
3. output to Δ a `COPY` command up to LMP; if LMP lies within a string compressed by a pointer, this pointer has to be split or some prefix of this string needs to be inserted by an `ADD` command;
4. output to Δ an `ADJUST` command for $[K, K+i]$ of R_c to reduce the relevant offsets by the length of the deleted string.

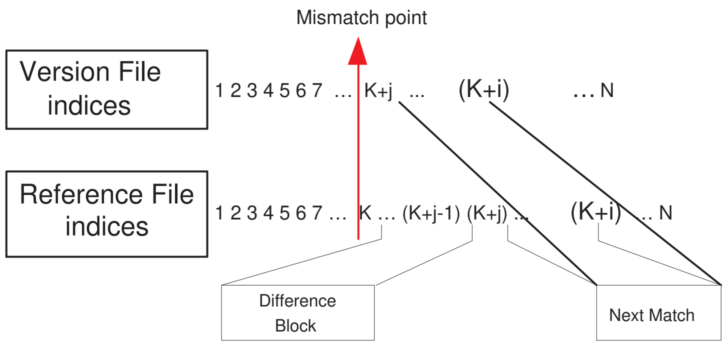


Figure 4. Schematic representation of deletion

3 The Decoding Algorithm

The descriptive nature of Δ is used to apply change commands to R_c, thereby transforming it into a file which is equivalent to V_c. The decoding process is linear in time and storage but is slightly more complex than the original LZSS decoding. This is due to the fact that pointers need to be adjusted along the way as data is inserted to or deleted from V_c. Moreover, the chain breaking mechanism might result in pointer splitting, as discussed above.

The algorithm is as follows, with the SPLITTO3 command explained below:

```
while Δ commands exist do:
    if COPY command
        copy substring of R_c to V_c
    if ADD command
        insert string into V_c
    if ADJUST pointers [i, j]
        do, up to window size from j + 1
            add the change size to the jump value of the pointers that
               point to a location preceding i;
            if the pointer is invalidated, issue a SPLITTO3 command
        end do
    end if
end while
```

As can be seen, the decoder runs in linear time and does not require additional memory. The algorithm performs a few more passes on parts of the data, but assuming the input is large relative to the window size, these passes are negligible. Further, if we consider a worst case scenario of needing to act upon many pointers pointing to the changed area, the pointers are bounded by the *offset* bits allocated for the pointer which bounds the entire procedure to part of the data. We assume that this is a small part compared to the input size. Again, if the changes are frequent in the input, the result will be a larger delta file and will require much work in the decoder. In such a case, it might be better to send the entire compressed version file and start fresh.

4 The Delta File

The delta file encapsulates commands which instruct the decoder how to convert its old compressed reference into a new compressed version. By applying the commands of the delta file, the decoding algorithm outputs the compressed version file with very low computational needs. The delta file is constructed such that the decoding complexity will be as small as possible and most of the computational effort is done by the encoder. This policy was chosen in order to be consistent with the *hop by hop* scheme to which this new compression suits. Reducing the complexity of the decoder in both time and storage allows small or computationally weak caching devices to be one of the hops along the way. Also, since the changes are the result of changes in the server side, the server can do most of the work for all the hops along the path to the clients. This will allow better utilization of the network nodes along the path between the server and the clients including the clients themselves.

Most of the previous work on delta files refers to an uncompressed domain, for example VCDIFF [8]. In our case, one needs in addition to the ADD, COPY and RUN commands of VCDIFF also means to split a pointer in order to insert changes. The new SPLITTO3 command breaks a pointer into three parts: the (possibly empty) prefix represents the pointer to the data portion which did not change; the middle part represents the change to be inserted and the (again possibly empty) suffix points to the representation of the remaining data. This methodology allows us to break the pointers of the compressed file and perform chain breaking with very little effort by the decoder.

In order to maintain this idea of lowering the computational demands of the decoder, we write all needed commands to Δ such that the decoder algorithm will not need to trace the restriction zones. This might lead to a larger delta file, but the size penalty is reasonable when the complexity needed by the hops is lower by doing most of the chain breaking at the encoder side.

Summarizing, the complete delta file command set consists of:

- `COPY` — similar to the copy of `VCDIFF`, it tells the decoder to copy a substring from the reference file to the decoded file.

- `ADDP` — Adds a pointer to V_c. This command is similar to `VCDIFF`'s `ADD` command, but adds a pointer, not characters.

- `ADDS` — Adds a string. Instructs the decoder to embed the command's parameter string into V_c in its current index. This command is similar to other delta file encoding `ADD` commands.

- `SPLITTO3` — the basic pointer breaking technique when a change which is in the middle of an area covered by a pointer is encountered. The decoder has to replace the changed pointer so that the result will be a reflection of the change in the compressed output. We split the pointer into three parts, the prefix part up to the change, the inserted part, which can be some string or a pointer to an early appearance, and a suffix part covering the remainder. The `SPLITTO3` command is also used when the encoder gets to a backpointer which points to a restricted zone. Since restricted zones need to be ignored as they represent invalid data, the command is used to break these pointers. The encoding algorithm stores each restricted zone and detects the first pointers that point to a restricted zone, then splits them. This way, we apply chain breaking, since, if we have P_1 pointing to a point in a restricted zone, and P_2 pointing to P_1, then by fixing P_1 we also take care of P_2 and the rest of the pointers that are connected to them. Also, this way we remove the decoder's need to trace pointers to restricted zones, hence simplifying the decoding algorithm.

- `ADJUSTJP` — Instructs the decoder to adjust all the offset sizes of the pointers, starting from a given start index up to a window size in R_c.

The size of the delta file is a major factor in the proposed scheme. Imagine a case where the compressed version is similar in size to the delta file. This overabundance of data is not needed since we could have sent V_c instead of the delta file. Sending V_c results is consuming less resources since there is no need for decoding along the path in each hop. Therefore, the rule is that if $|\Delta| \ll |V_c|$, send Δ. Otherwise, it is better to send V_c.

The following is a small sample of Δ in its textual form. In practice we used a binary code in order to encode the commands. Further, when dealing with very large files, in which the delta file is also large enough, Δ itself can be compressed in order to further reduce its size.

```
COPY [from = 0, to = 77]
SPLITTO3 [offset = 114, length = 113]
    PrefixPointer = [offset = 114, length = 103]
    Change = ['2'] // substitute one character
    SuffixPointer = [offset = 9, length = 9]
COPY [from = 81, to = 180]
```

The sample above represents a Δ of changing a single character in a file V consisting of 2864 1-byte numbers. In this example, a single number 1 has been changed into the number 2. The original file was compressed using LZSS to become R_c of size 180 bytes. The decoder algorithm manipulates R_c using the above Δ to output V_c.

5 Experiments

Table 1 summarizes the benefits of using the proposed algorithm. Real life HTML files have been used, e.g., www.cnn.com, with uncompressed size of 107 KByte, text files such as RFC's and a set of synthetic files, including many repetitions. The change type has been restricted in the tests to substitutions (S) and insertions (I), as deletions behave symmetrically to insertions. The results are compared to the binary uncompressed delta encoding scheme XDelta, which uses zlib to compress its delta file that requires a decompression phase before it can be used to reconstruct the uncompressed version. Our scheme takes into consideration that network elements are indifferent to the content of the data and thus prefer to maintain cached files in their compressed form and not to decompress them for running the delta encoding algorithm or recompress them after reconstruction. The results are compared to re-encoding the new version data, since this scenario represents a reference implementation of compression based transactions between a client and a server, for both types of regular recompression and the proposed delta encoding.

File	Number of Changes	Change type	Change size (bits)	GZip (bits)	CDDelta (bits)	Xdelta (bits)
CNN	0	No change	0	168472	67	1160
CNN	1	S	8	169944	145	1688
CNN	1	I	128	170024	265	1808
CNN	2	S	48	168496	255	1784
CNN	2	I	1608	168744	1912	2408
CNN	3	I	1872	168744	2254	2592
CNN	4	I	2144	168912	2652	2864
CNN	5	I	2408	169088	3042	3032
CNN	6	I	2672	169120	3384	3144
synthetic	1	S	8	1960	201	1712
synthetic	1	I	24	2136	217	1800
synthetic	1	S	72	2016	257	1728

Table 1: *Experimental results*

The first row of Table 1 describes the case of no change in the version file. Even though this case looks odd at first, it is a real life case. When considering the network elements that cache the data from a server, combined with the HTTP cache control commands, we see that many data objects, including textual HTML files, have an expiration time. This expiration time causes the caching element to ask the server if the data was modified (the `if-modified-since` HTTP attribute) and in some cases, depending on both the network element and the server type, it causes the network element to ask for a new version. In this typical case, our scheme will always output a 67 bit command which is to copy the entire reference file.

The encoder performance has been tested on an Intel's core 2 due processor 2.4 GHz, and achieved a throughput of 36 MByte/sec on one CPU. This performance is similar to our LZSS encoder implementation. A 3 bit field was used to represent the command, and 32 bits for an index in the file, such as the from field of the `COPY` command. When referring to offsets, as in the `ADJUSTP` command, 16 bits were used, since the window size was up to 64 Kbytes.

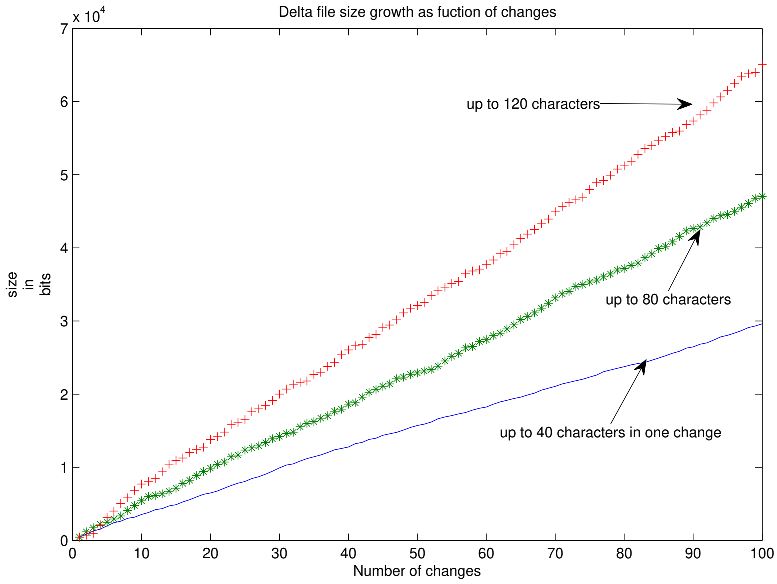


Figure 5. Size of delta file as function of the number of changes

From Table 1 we can see that on this data, a factor of about 46 is gained when compared to the compressed size of the file. The Delta file size increases when there are more changes. However, changes that were tested reflect a change of 7.4 lines with an average size of 45 characters per line. It must be noted that these results are without compression of the delta file and with no compression of the changed data relative to itself or relative to the entire file as a reference.

Figure 5 shows the size increase in bits of the delta file as a function of the number of changes, where each such change affects a randomly chosen set of 40, 80 and 120 characters. We can see that the size of the delta file increases linearly with the number of changes.

Figure 6 visualizes the data included in Table 1 concerning the size relation between a gzip compressed file (CNN) and the Delta file for our insertion tests. The x-axis gives the number of insertions, and the y -axis the size of the file on a logarithmic scale. We can see that the Delta file size increases with the number of inserted characters, while the insertion has only a negligible effect on the size of the new version gzip compressed file. The Delta file consists mostly of the inserted characters themselves.

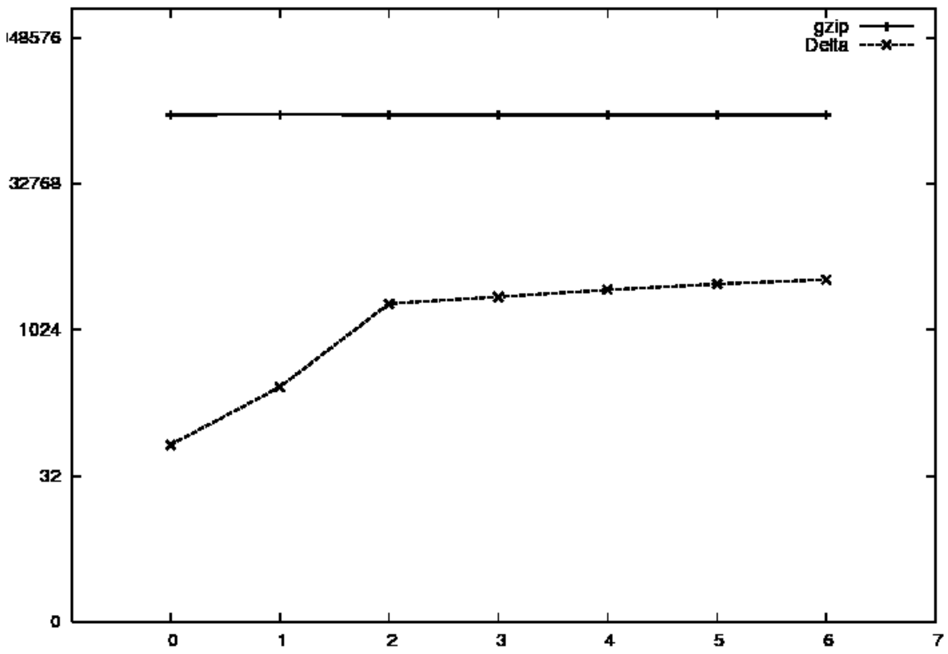


Figure 6. Comparing Gzip and Delta sizes

6 Conclusion

We focused on the semi-compressed delta encoding problem for LZSS encoded files, in an application in which the reconstructed version file is directly given in compressed form. This can greatly reduce network traffic and the CPU and storage requirements of the various network elements. The algorithm is based on a partial local reconstruction of a previous occurrence of the data, using the compressed reference and the new version.

References

1. M. Ajtai, R. Burns, R. Fagin, D. Long, and L. Stockmeyer: *Compactly encoding unstructured input with differential compression*. Journal of the ACM, 49(3) 2002, pp. 318–367.
2. B. B. C. Xiao and G. Chang: *Delta compression for fast wireless internet download*. IEEE GLOBECOM Proceedings, 2005, pp. 474–478.
3. M. Chan and T. Woo: *Cache-based compaction: A new technique for optimizing web transfer*. Proceedings of the IEEE Infocom '99 Conference, 1999, pp. 117–125.
4. J. Hunt, K. Vo, and W. Tichy: *Delta algorithms: An empirical analysis*. ACM Trans. on Software Eng. and Methodology, 7(2) 1998, pp. 192–214.
5. J. Storer and T. Syzmanski: *Data compression via textual substitution*. Journal of the ACM, 29 1982, pp. 928–951.
6. S. Klein, T. Serebro, and D. Shapira: *Modeling delta encoding of compressed files*. International Journal of the Foundations of Computer Science, 19 2008, pp. 137–146.
7. S. Klein and D. Shapira: *Compressed delta encoding for LZSS encoded files*. Proceedings of Data Compression Conference, DCC–07, 2007, pp. 113–122.
8. D. Korn, J. MacDonald, J. Mogul, and K. Vo: *The VCDIFF Generic Differencing and Compression Data Format*. RFC 3284 (Proposed Standard), June 2002.
9. G. Motta, J. Gustafson, and S. Chen: *Differencial compression of executable code*. Proceedings of Data Compression Conference, DCC–07, 2007, pp. 103–112.
10. R. Williams: *An extremly fast Ziv-Lempel data compression algorithm*. Proceedings of Data Compression Conference, DCC–91, 1991, pp. 362–371.

On Bijective Variants of the Burrows-Wheeler Transform

Manfred Kufleitner

Universität Stuttgart, FMI,
Universitätsstr. 38, 70569 Stuttgart, Germany
kufleitner@fmi.uni-stuttgart.de

Abstract. The sort transform (ST) is a modification of the Burrows-Wheeler transform (BWT). Both transformations map an arbitrary word of length n to a pair consisting of a word of length n and an index between 1 and n. The BWT sorts all rotation conjugates of the input word, whereas the ST of order k only uses the first k letters for sorting all such conjugates. If two conjugates start with the same prefix of length k, then the indices of the rotations are used for tie-breaking. Both transforms output the sequence of the last letters of the sorted list and the index of the input within the sorted list. In this paper, we discuss a bijective variant of the BWT (due to Scott), proving its correctness and relations to other results due to Gessel and Reutenauer (1993) and Crochemore, Désarménien, and Perrin (2005). Further, we present a novel bijective variant of the ST.

1 Introduction

The Burrows-Wheeler transform (BWT) is a widely used preprocessing technique in lossless data compression [5]. It brings every word into a form which is likely to be easier to compress [18]. Its compression performance is almost as good as PPM (prediction by partial matching) schemes [7] while its speed is comparable to that of Lempel-Ziv algorithms [13,14]. Therefore, BWT based compression schemes are a very reasonable trade-off between running time and compression ratio.

In the classic setting, the BWT maps a word of length n to a word of length n and an index (comprising $O(\log n)$ bits). Thus, the BWT is not bijective and hence, it is introducing new redundancies to the data, which is cumbersome and undesired in applications of data compression or cryptography. Instead of using an index, a very common technique is to assume that the input has a unique end-of-string symbol [3,18]. Even though this often simplifies proofs or allows speeding up the algorithms, the use of an end-of-string symbol introduces new redundancies (again $O(\log n)$ bits are required for coding the end-of-string symbol).

We discuss bijective versions of the BWT which are one-to-one correspondences between words of length n. In particular, no index and no end-of-string symbol is needed. Not only does bijectivity save a few bits, for example, it also increases data security when cryptographic procedures are involved; it is more natural and it can help us to understand the BWT even better. Moreover, the bijective variants give us new possibilities for enhancements; for example, in the bijective BWT different orders on the letters can be used for the two main stages.

Several variants of the BWT have been introduced [2,17]. An overview can be found in the textbook by Adjeroh, Bell, and Mukherjee [1]. One particularly important variant for this paper is the sort transform (ST), which is also known under the name Schindler transform [22]. In the original paper, the inverse of the ST is described only

Manfred Kufleitner: On Bijective Variants of the Burrows-Wheeler Transform, pp. 65–79.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

very briefly. More precise descriptions and improved algorithms for the inverse of the ST have been proposed recently [19,20,21]. As for the BWT, the ST also involves an index or an end-of-string symbol. In particular, the ST is not onto and it introduces new redundancies.

The bijective BWT was discovered and first described by Scott (2007), but his exposition of the algorithm was somewhat cryptic, and was not appreciated as such. In particular, the fact that this transform is based on the Lyndon factorization went unnoticed by Scott. Gil and Scott [12] provided an accessible description of the algorithm. Here, we give an alternative description, a proof of its correctness, and more importantly, draw connections between Scott's algorithm and other results in combinatorics on words. Further, this variation of the BWT is used to introduce techniques which are employed at the bijective sort transform, which makes the main contribution of this paper. The forward transform of the bijective ST is rather easy, but we have to be very careful with some details. Compared with the inverse of the bijective BWT, the inverse of the bijective ST is more involved.

Outline. The paper is organized as follows. In Section 2 we fix some notation and repeat basic facts about combinatorics on words. On our way to the bijective sort transform (Section 6) we investigate the BWT (Section 3), the bijective BWT (Section 4), and the sort transform (Section 5). We give full constructive proofs for the injectivity of the respective transforms. Each section ends with a running example which illustrates the respective concepts. Apart from basic combinatorics on words, the paper is completely self-contained.

2 Preliminaries

Throughout this paper we fix the finite non-empty alphabet Σ and assume that Σ is equipped with a linear order $\leq$. A *word* is a sequence $a_1 \cdots a_n$ of letters $a_i \in \Sigma$, $1 \leq i \leq n$. The set of all such sequences is denoted by Σ^*; it is the free monoid over Σ with concatenation as composition and with the empty word ε as neutral element. The set $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ consists of all non-empty words. For words u, v we write $u \leq v$ if $u = v$ or if u is lexicographically smaller than v with respect to the order $\leq$ on the letters. Let $w = a_1 \cdots a_n \in \Sigma^+$ be a non-empty word with letters $a_i \in \Sigma$. The *length* of w, denoted by $|w|$, is n. The empty word is the unique word of length 0. We can think of w as a labeled linear order: position i of w is labeled by $a_i \in \Sigma$ and in this case we write $\lambda_w(i) = a_i$, so each word w induces a labeling function λ_w. The first letter a_1 of w is denoted by $\mathrm{first}(w)$ while the last letter a_n is denoted by $\mathrm{last}(w)$. The *reversal* of a word w is $\overline{w} = a_n \cdots a_1$. We say that two words u, v are *conjugate* if $u = st$ and $v = ts$ for some words s, t, i.e., u and v are cyclic shifts of one another. The j-fold concatenation of w with itself is denoted by w^j. A word u is a *root* of w if $w = u^j$ for some $j \in \mathbb{N}$. A word w is *primitive* if $w = u^j$ implies $j = 1$ and hence $u = w$, i.e., w has only the trivial root w.

The *right-shift* of $w = a_1 \cdots a_n$ is $r(w) = a_n a_1 \cdots a_{n-1}$ and the i-fold right shift $r^i(w)$ is defined inductively by $r^0(w) = w$ and $r^{i+1}(w) = r(r^i(w))$. We have $r^i(w) = a_{n-i+1} \cdots a_n a_1 \cdots a_{n-i}$ for $0 \leq i < n$. The word $r^i(w)$ is also well-defined for $i \geq n$ and then $r^i(w) = r^j(w)$ where $j = i \bmod n$. We define the *ordered conjugacy class* of a word $w \in \Sigma^n$ as $[w] = (w_1, \ldots, w_n)$ where $w_i = r^{i-1}(w)$. It is convenient to think of $[w]$ as a cycle of length n with a pointer to a distinguished starting position. Every position i, $1 \leq i \leq n$, on this cycle is labeled by a_i. In particular, a_1 is a successor of a_n on this cycle since the position 1 is a successor of the position n. The mapping r moves the pointer to its predecessor. The (unordered) conjugacy class of w is the multiset $\{w_1, \ldots, w_n\}$. Whenever there is no confusion, then by abuse of notation we also write $[w]$ to denote the (unordered) conjugacy class of w. For instance, this is the case if w is in some way distinguished within its conjugacy class, which is true if w is a Lyndon word. A *Lyndon word* is a non-empty word which is the unique lexicographic minimal element within its conjugacy class. More formally, let $[w] = (w, w_2, \ldots, w_n)$, then $w \in \Sigma^+$ is a Lyndon word if $w < w_i$ for all $i \in \{2, \ldots, n\}$. Lyndon words have a lot of nice properties [15]. For instance, Lyndon words are primitive. Another interesting fact is the following.

Fact 1 (Chen, Fox, and Lyndon [6]). *Every word $w \in \Sigma^+$ has a unique factorization $w = v_s \cdots v_1$ such that $v_1 \leq \cdots \leq v_s$ is a non-decreasing sequence of Lyndon words.*

An alternative formulation of the above fact is that every word w has a unique factorization $w = v_s^{n_s} \cdots v_1^{n_1}$ where $n_i \geq 1$ for all i and where $v_1 < \cdots < v_s$ is a strictly increasing sequence of Lyndon words. The factorization of w as in Fact 1 is called the *Lyndon factorization* of w. It can be computed in linear time using Duval's algorithm [9].

Suppose we are given a multiset $V = \{v_1, \ldots, v_s\}$ of Lyndon words enumerated in non-decreasing order $v_1 \leq \cdots \leq v_s$. Now, V uniquely determines the word $w = v_s \cdots v_1$. Therefore, the Lyndon factorization induces a one-to-one correspondence between arbitrary words of length n and multisets of Lyndon words of total length n. Of course, by definition of Lyndon words, the multiset $\{v_1, \ldots, v_s\}$ of Lyndon words and the multiset $\{[v_1], \ldots, [v_s]\}$ of conjugacy classes of Lyndon words are also in one-to-one correspondence.

We extend the order $\leq$ on Σ as follows to non-empty words. Let $w^\omega = www \cdots$ be the infinite sequences obtained as the infinite power of w. For $u, v \in \Sigma^+$ we write $u \leq^\omega v$ if either $u^\omega = v^\omega$ or $u^\omega = paq$ and $v^\omega = pbr$ for $p \in \Sigma^*$, $a, b \in \Sigma$ with $a < b$, and infinite sequences q, r; phrased differently, $u \leq^\omega v$ means that the infinite sequences u^ω and v^ω satisfy $u^\omega \leq v^\omega$. If u and v have the same length, then $\leq^\omega$ coincides with the lexicographic order induced by the order on the letters. For arbitrary words, $\leq^\omega$ is only a preorder since for example $u \leq^\omega uu$ and $uu \leq^\omega u$. On the other hand, if $u \leq^\omega v$ and $v \leq^\omega u$ then $u^{|v|} = v^{|u|}$. Hence, by the periodicity lemma [10], there exists a common root $p \in \Sigma^+$ and $g, h \in \mathbb{N}$ such that $u = p^g$ and $v = p^h$. Also note that $b \leq ba$ whereas $ba \leq^\omega b$ for $a < b$.

Intuitively, the *context of order k* of w is the sequence of the first k letters of w. We want this notion to be well-defined even if $|w| < k$. To this end let $\mathrm{context}_k(w)$ be the prefix of length k of w^ω, i.e., $\mathrm{context}_k(w)$ consists of the first k letters on the cycle $[w]$. Note that our definition of a context of order k is left-right symmetric to the corresponding notion used in data compression. This is due to the fact that typical compression schemes are applying the BWT or the ST to the reversal of the input.

An important construction in this paper is the *standard permutation* π_w on the set of positions $\{1, \ldots, n\}$ induced by a word $w = a_1 \cdots a_n \in \Sigma^n$ [11]. The first step is to introduce a new order $\preceq$ on the positions of w by sorting the letters within w such that identical letters preserve their order. More formally, the linear order $\preceq$ on $\{1, \ldots, n\}$ is defined as follows: $i \preceq j$ if

$$a_i < a_j \qquad \text{or} \qquad a_i = a_j \text{ and } i \leq j.$$

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

(a) Conjugacy class $[w]$

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

(b) Lexicographically sorted

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

(c) Sorted by 2-order contexts

Figure 1. Computing the BWT and the ST of the word $w = bcbccbcbcabbaaba$

Let $j_1 \prec \cdots \prec j_n$ be the linearization of $\{1, \ldots, n\}$ according to this new order. Now, the standard permutation π_w is defined by $\pi_w(i) = j_i$.

Example 2. Consider the word $w = bcbccbcbcabbaaba$ over the ordered alphabet $a < b < c$. We have $|w| = 16$. Therefore, the positions in w are $\{1, \ldots, 16\}$. For instance, the label of position 6 is $\lambda_w(6) = b$. Its Lyndon factorization is $w = bcbcc \cdot bc \cdot bc \cdot abb \cdot aab \cdot a$. The context of order 7 of the prefix $bcbcc$ of length 5 is $bcbccbc$ and the context of order 7 of the factor bc is $bcbcbcb$. For computing the standard permutation we write w column-wise, add positions, and then sort the pairs lexicographically:

word w	w with positions	sorted
b	$(b, 1)$	$(a, 10)$
c	$(c, 2)$	$(a, 13)$
b	$(b, 3)$	$(a, 14)$
c	$(c, 4)$	$(a, 16)$
c	$(c, 5)$	$(b, 1)$
b	$(b, 6)$	$(b, 3)$
c	$(c, 7)$	$(b, 6)$
b	$(b, 8)$	$(b, 8)$
c	$(c, 9)$	$(b, 11)$
a	$(a, 10)$	$(b, 12)$
b	$(b, 11)$	$(b, 15)$
b	$(b, 12)$	$(c, 2)$
a	$(a, 13)$	$(c, 4)$
a	$(a, 14)$	$(c, 5)$
b	$(b, 15)$	$(c, 7)$
a	$(a, 16)$	$(c, 9)$

This yields the standard permutation

$$\pi_w = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 10 & 13 & 14 & 16 & 1 & 3 & 6 & 8 & 11 & 12 & 15 & 2 & 4 & 5 & 7 & 9 \end{pmatrix}.$$

The conjugacy class $[w]$ of w is depicted in Figure 1(a); the i-th word in $[w]$ is written in the i-th row. The last column of the matrix for $[w]$ is the reversal $\overline{w}$ of w.

3 The Burrows-Wheeler transform

The *Burrows-Wheeler transform* (BWT) maps words w of length n to pairs (L, i) where L is a word of length n and i is an index in $\{1, \ldots, n\}$. The word L is usually referred to as the Burrows-Wheeler transform of w. In particular, the BWT is not surjective. We will see below how the BWT works and that it is one-to-one. It follows that only a fraction of $1/n$ of all possible pairs (L, i) appears as an image under the BWT. For instance $(bacd, 1)$ where $a < b < c < d$ is not an image under the BWT.

For $w \in \Sigma^+$ we define $M(w) = (w_1, \dots, w_n)$ where $\{w_1, \dots, w_n\} = [w]$ and $w_1 \leq \dots \leq w_n$. Now, the Burrows-Wheeler transform of w consists of the word $\mathrm{BWT}(w) = \mathrm{last}(w_1) \cdots \mathrm{last}(w_n)$ and an index i such that $w = w_i$. Note that in contrast to the usual definition of the BWT, we are using right shifts; at this point this makes no difference but it unifies the presentation of succeeding transforms. At first glance, it is surprising that one can reconstruct $M(w)$ from $\mathrm{BWT}(w)$. Moreover, if we know the index i of w in the sorted list $M(w)$, then we can reconstruct w from $\mathrm{BWT}(w)$. One way of how to reconstruct $M(w)$ is presented in the following lemma. For later use, we prove a more general statement than needed for computing the inverse of the BWT.

Lemma 3. *Let $k \in \mathbb{N}$. Let $\bigcup_{i=1}^{s} [v_i] = \{w_1, \dots, w_n\} \subseteq \Sigma^+$ be a multiset built from conjugacy classes $[v_i]$. Let $M = (w_1, \dots, w_n)$ satisfy $\mathrm{context}_k(w_1) \leq \dots \leq \mathrm{context}_k(w_n)$ and let $L = \mathrm{last}(w_1) \cdots \mathrm{last}(w_n)$ be the sequence of the last symbols. Then*

$$\mathrm{context}_k(w_i) = \lambda_L \pi_L(i) \cdot \lambda_L \pi_L^2(i) \cdots \lambda_L \pi_L^k(i)$$

where π_L^t denotes the t-fold application of π_L and $\lambda_L \pi_L^t(i) = \lambda_L\big(\pi_L^t(i)\big)$.

Proof. By induction over the context length t, we prove that for all $i \in \{1, \dots, n\}$ we have $\mathrm{context}_t(w_i) = \lambda_L \pi_L(i) \cdots \lambda_L \pi_L^t(i)$. For $t = 0$ we have $\mathrm{context}_0(w_i) = \varepsilon$ and hence, the claim is trivially true. Let now $0 < t \leq k$. By the induction hypothesis, the $(t-1)$-order context of each w_i is $\lambda_L \pi_L(i) \cdots \lambda_L \pi_L^{t-1}(i)$. By applying one right-shift, we see that the t-order context of $r(w_i)$ is $\lambda_L(i) \cdot \lambda_L \pi_L^1(i) \cdots \lambda_L \pi_L^{t-1}(i)$.

The list M meets the sort order induced by k-order contexts. In particular, $(w_1, \dots, w_n)$ is sorted by $(t-1)$-order contexts. Let $(u_1, \dots, u_n)$ be a stable sort by t-order contexts of the right-shifts $(r(w_1), \dots, r(w_n))$. The construction of $(u_1, \dots, u_n)$ only requires a sorting of the first letters of $(r(w_1), \dots, r(w_n))$ such that identical letters preserve their order. The sequence of first letters of the words $r(w_1), \dots, r(w_n)$ is exactly L. By construction of π_L, it follows that $(u_1, \dots, u_n) = (w_{\pi_L(1)}, \dots, w_{\pi_L(n)})$. Since M is built from conjugacy classes, the multisets of elements occurring in $(w_1, \dots, w_n)$ and $(r(w_1), \dots, r(w_n))$ are identical. The same holds for the multisets induced by $(w_1, \dots, w_n)$ and $(u_1, \dots, u_n)$. Therefore, the sequences of t-order contexts induced by $(w_1, \dots w_n)$ and $(u_1, \dots, u_n)$ are identical. Moreover, we conclude

$$\mathrm{context}_t(w_i) = \mathrm{context}_t(u_i) = \mathrm{context}_t(w_{\pi_L(i)}) = \lambda_L \pi_L(i) \cdot \lambda_L \pi_L^2(i) \cdots \lambda_L \pi_L^t(i)$$

which completes the induction. We note that in general $u_i \neq w_i$ since the sort order of M beyond k-order contexts is arbitrary. Moreover, for $t = k+1$ the property $\mathrm{context}_t(w_i) = \mathrm{context}_t(u_i)$ does not need to hold (even though the multisets of $(k+1)$-order contexts coincide). □

Note that in Lemma 3 we do not require that all v_i have the same length. Applying the BWT to conjugacy classes of words with different lengths has also been used for the *Extended BWT* [17].

Corollary 4. *The BWT is invertible, i.e., given* $(\mathrm{BWT}(w), i)$ *where i is the index of w in $M(w)$ one can reconstruct the word w.*

Proof. We set $k = |w|$. Let $M = M(w)$ and $L = \mathrm{BWT}(w)$. Now, by Lemma 3 we see that

$$w = w_i = \mathrm{context}_k(w_i) = \lambda_L \pi_L^1(i) \cdots \lambda_L \pi_L^{|L|}(i).$$

In particular, $w = \lambda_L \pi_L^1(i) \cdots \lambda_L \pi_L^{|L|}(i)$ only depends on L and i. □

Remark 5. In the special case of the BWT it is possible to compute the i-th element w_i of $M(w)$ by using the inverse π_L^{-1} of the permutation π_L:

$$w_i = \lambda_L \pi_L^{-|w_i|+1}(i) \cdots \lambda_L \pi_L^{-1}(i) \lambda_L(i).$$

This justifies the usual way of computing the inverse of $(\mathrm{BWT}(w), i)$ from right to left (by using the restriction of π_L^{-1} to the cycle containing the element i). The motivation is that the (required cycle of the) inverse π_L^{-1} seems to be easier to compute than the standard permutation π_L.

Example 6. We compute the BWT of $w = bcbccbcbcabbaaba$ from Example 2. The lexicographically sorted list $M(w)$ can be found in Figure 1(b). This yields the transform $(\mathrm{BWT}(w), i) = (bacbbaaccacbbcbb, 10)$ where $L = \mathrm{BWT}(w)$ is the last column of the matrix $M(w)$ and w is the i-th row in $M(w)$. The standard permutation of L is

$$\pi_L = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 2 & 6 & 7 & 10 & 1 & 4 & 5 & 12 & 13 & 15 & 16 & 3 & 8 & 9 & 11 & 14 \end{pmatrix}.$$

Now, $\pi_L^1(10) \cdots \pi_L^{16}(10)$ gives us the following sequence of positions starting with $\pi_L(10) = 15$:

$$15 \overset{\pi_L}{\mapsto} 11 \overset{\pi_L}{\mapsto} 16 \overset{\pi_L}{\mapsto} 14 \overset{\pi_L}{\mapsto} 9 \overset{\pi_L}{\mapsto} 13 \overset{\pi_L}{\mapsto} 8 \overset{\pi_L}{\mapsto} 12 \overset{\pi_L}{\mapsto} 3 \overset{\pi_L}{\mapsto} 7 \overset{\pi_L}{\mapsto} 5 \overset{\pi_L}{\mapsto} 1 \overset{\pi_L}{\mapsto} 2 \overset{\pi_L}{\mapsto} 6 \overset{\pi_L}{\mapsto} 4 \overset{\pi_L}{\mapsto} 10.$$

Applying the labeling function λ_L to this sequence of positions yields

$$\begin{aligned} &\lambda_L(15)\lambda_L(11)\lambda_L(16)\lambda_L(14)\lambda_L(9)\lambda_L(13)\lambda_L(8)\lambda_L(12) \\ &\quad \cdot \lambda_L(3)\lambda_L(7)\lambda_L(5)\lambda_L(1)\lambda_L(2)\lambda_L(6)\lambda_L(4)\lambda_L(10) \\ &= bcbccbcbcabbaaba = w, \end{aligned}$$

i.e., we have successfully reconstructed the input w from $(\mathrm{BWT}(w), i)$.

4 The bijective Burrows-Wheeler transform

Now we are ready to give a comprehensive description of Scott's bijective variant of the BWT and to prove its correctness. It maps a word of length n to a word of length n — without any index or end-of-string symbol being involved. The key ingredient is the Lyndon factorization: Suppose we are computing the BWT of a Lyndon word v, then we do not need an index since we know that v is the first element of the list $M(v)$. This leads to the computation of a multi-word BWT of the Lyndon factors of the input.

The bijective BWT of a word w of length n is defined as follows. Let $w = v_s \cdots v_1$ with $v_s \geq \cdots \geq v_1$ be the Lyndon factorization of w. Let $\mathrm{LM}(w) = (u_1, \ldots, u_n)$ where $u_1 \leq^\omega \cdots \leq^\omega u_n$ and where the multiset $\{u_1, \ldots, u_n\} = \bigcup_{i=1}^{s} [v_i]$. Then, the bijective BWT of w is $\mathrm{BWTS}(w) = \mathrm{last}(u_1) \cdots \mathrm{last}(u_n)$. The S in BWTS is for *Scottified.* Note that if w is a power of a Lyndon word, then $\mathrm{BWTS}(w) = \mathrm{BWT}(w)$.

In some sense, the bijective BWT can be thought of as the composition of the Lyndon factorization [6] with the inverse of the Gessel-Reutenauer transform [11]. In particular, a first step towards a bijective BWT can be found in a 1993 article by Gessel and Reutenauer [11] (prior to the publication of the BWT [5]). The link between the Gessel-Reutenauer transform and the BWT was pointed out later by Crochemore et al. [8]. A similar approach as in the bijective BWT has been employed by Mantaci et al. [16]; instead of the Lyndon factorization they used a decomposition of the input into blocks of equal length. The output of this variant is a word and a sequence of indices (one for each block). In its current form, the bijective BWT has been proposed by Scott [23] in a newsgroup posting in 2007. Gil and Scott gave an accessible version of the transform, an independent proof of its correctness, and they tested its performance in data compression [12]. The outcome of these tests is that the bijective BWT beats the usual BWT on almost all files of the Calgary Corpus [4] by at least a few hundred bytes which exceeds the gain of just saving the rotation index.

Lemma 7. *Let* $w = v_s \cdots v_1$ *with* $v_s \geq \cdots \geq v_1$ *be the Lyndon factorization of* w*, let* $\mathrm{LM}(w) = (u_1, \ldots, u_n)$*, and let* $L = \mathrm{BWTS}(w)$*. Consider the cycle* C *of the permutation* π_L *which contains the element* 1 *and let* d *be the length of* C*. Then* $\lambda_L \pi_L^1(1) \cdots \lambda_L \pi_L^d(1) = v_1$*.*

Proof. By Lemma 3 we see that $\left(\lambda_L \pi_L^1(1) \cdots \lambda_L \pi_L^d(1)\right)^{|v_1|} = v_1^d$. Since v_1 is primitive it follows $\lambda_L \pi_L^1(1) \cdots \lambda_L \pi_L^d(1) = v_1^z$ for some $z \in \mathbb{N}$. In particular, the Lyndon factorization of w ends with v_1^z.

Let U be the subsequence of $\mathrm{LM}(w)$ which consists of those u_i which come from this last factor v_1^z. The sequence U contains each right-shift of v_1 exactly z times. Moreover, the sort-order within U depends only on $|v_1|$-order contexts.

The element $v_1 = u_1$ is the first element in U since v_1 is a Lyndon word. In particular, $\pi_L^0(1) = 1$ is the first occurrence of $r^0(v_1) = v_1$ within U. Suppose $\pi_L^j(1)$ is the first occurrence of $r^j(v_1)$ within U. Let $\pi_L^j(1) = i_1 < \cdots < i_z$ be the indices of all occurrences of $r^j(v_1)$ in U. By construction of π_L, we have $\pi_L(i_1) < \cdots < \pi_L(i_z)$ and therefore $\pi_L^{j+1}(1)$ is the first occurrence of $r^{j+1}(v_1)$ within U. Inductively, $\pi_L^j(1)$ always refers to the first occurrence of $r^j(v_1)$ within U (for all $j \in \mathbb{N}$). In particular it follows that $\pi_L^{|v_1|}(1) = 1$ and $z = 1$. □

Theorem 8. *The bijective BWT is invertible, i.e., given* $\mathrm{BWTS}(w)$ *one can reconstruct the word* w*.*

Proof. Let $L = \mathrm{BWTS}(w)$ and let $w = v_s \cdots v_1$ with $v_s \geq \cdots \geq v_1$ be the Lyndon factorization of w. Each permutation admits a cycle structure. We decompose the standard permutation π_L into cycles $C_1, \ldots, C_t$. Let i_j be the smallest element of the cycle C_j and let d_j be the length of C_j. We can assume that $1 = i_1 < \cdots < i_t$.

We claim that $t = s$, $d_j = |v_j|$, and $\lambda_L \pi_L^1(i_j) \cdots \lambda_L \pi_L^{d_j}(i_j) = v_j$. By Lemma 7 we have $\lambda_L \pi_L^1(i_1) \cdots \lambda_L \pi_L^{d_1}(i_1) = v_1$. Let π'_L denote the restriction of π_L to the set $C = C_2 \cup \cdots \cup C_t$, where by abuse of notation $C_2 \cup \cdots \cup C_t$ denotes the set of all elements occurring in $C_2, \ldots, C_t$. Let $L' = \mathrm{BWTS}(v_s \cdots v_2)$. The word L' can be obtained from L by removing all positions occurring in the cycle C_1. This yields a monotone bijection

$$\alpha : C \to \{1, \ldots, |L'|\}$$

such that $\lambda_L(i) = \lambda_{L'}\alpha(i)$ and $\alpha\pi_L(i) = \pi_{L'}\alpha(i)$ for all $i \in C$. In particular, $\pi_{L'}$ has the same cycle structure as π'_L and $1 = \alpha(i_2) < \cdots < \alpha(i_t)$ is the sequence of the minimal elements within the cycles. By induction on the number of Lyndon factors,

$$\begin{aligned} v_s \cdots v_2 &= \lambda_{L'}\pi^1_{L'}\alpha(i_t)\cdots\lambda_{L'}\pi^{d_t}_{L'}\alpha(i_t) \ \cdots \ \lambda_{L'}\pi^1_{L'}\alpha(i_2)\cdots\lambda_{L'}\pi^{d_2}_{L'}(i_2) \\ &= \lambda_{L'}\alpha\pi^1_L(i_t)\cdots\lambda_{L'}\alpha\pi^{d_t}_L(i_t) \ \cdots \ \lambda_{L'}\alpha\pi^1_L(i_2)\cdots\lambda_{L'}\alpha\pi^{d_2}_L(i_2) \\ &= \lambda_L\pi^1_L(i_t)\cdots\lambda_L\pi^{d_t}_L(i_t) \ \cdots \ \lambda_L\pi^1_L(i_2)\cdots\lambda_L\pi^{d_2}_L(i_2). \end{aligned}$$

Appending $\lambda_L\pi^1_L(i_1)\cdots\lambda_L\pi^{d_1}_L(i_1) = v_1$ to the last line allows us to reconstruct w by

$$w = \lambda_L\pi^1_L(i_t)\cdots\lambda_L\pi^{d_t}_L(i_t) \ \cdots \ \lambda_L\pi^1_L(i_1)\cdots\lambda_L\pi^{d_1}_L(i_1).$$

Moreover, $t = s$ and $d_j = |v_j|$. We note that this formula for w only depends on L and does not require any index to an element in $\mathrm{LM}(w)$. □

Example 9. We again consider the word $w = bcbccbcbcabbaaba$ from Example 2 and its Lyndon factorization $w = v_6 \cdots v_1$ where $v_6 = bcbcc$, $v_5 = bc$, $v_4 = bc$, $v_3 = abb$, $v_2 = aab$, and $v_1 = a$. The lists $([v_1], \ldots, [v_6])$ and $\mathrm{LM}(w)$ are:

	$([v_1], \ldots, [v_6])$
1	a
2	$a\ a\ b$
3	$b\ a\ a$
4	$a\ b\ a$
5	$a\ b\ b$
6	$b\ a\ b$
7	$b\ b\ a$
8	$b\ c$
9	$c\ b$
10	$b\ c$
11	$c\ b$
12	$b\ c\ b\ c\ c$
13	$c\ b\ c\ b\ c$
14	$c\ c\ b\ c\ b$
15	$b\ c\ c\ b\ c$
16	$c\ b\ c\ c\ b$

	$\mathrm{LM}(w)$
1	a
2	$a\ a\ b$
4	$a\ b\ a$
5	$a\ b\ b$
3	$b\ a\ a$
6	$b\ a\ b$
7	$b\ b\ a$
8	$b\ c$
10	$b\ c$
12	$b\ c\ b\ c\ c$
15	$b\ c\ c\ b\ c$
9	$c\ b$
11	$c\ b$
13	$c\ b\ c\ b\ c$
16	$c\ b\ c\ c\ b$
14	$c\ c\ b\ c\ b$

Hence, we obtain $L = \mathrm{BWTS}(w) = abababaccccbbcbb$ as the sequence of the last symbols of the words in $\mathrm{LM}(w)$. The standard permutation π_L induced by L is

$$\pi_L = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 1 & 3 & 5 & 7 & 2 & 4 & 6 & 12 & 13 & 15 & 16 & 8 & 9 & 10 & 11 & 14 \end{pmatrix}$$

The cycles of π_L arranged by their smallest elements are $C_1 = (1)$, $C_2 = (2, 3, 5)$, $C_3 = (4, 7, 6)$, $C_4 = (8, 12)$, $C_5 = (9, 13)$, and $C_6 = (10, 15, 11, 16, 14)$. Applying the labeling function λ_L to the cycle C_i (starting with the second element) yields the Lyndon factor v_i. With this procedure, we reconstructed $w = v_6 \cdots v_1$ from $L = \mathrm{BWTS}(w)$.

5 The sort transform

The *sort transform* (ST) is a BWT where we only sort the conjugates of the input up to a given depth k and then we are using the index of the conjugates as a tie-breaker. Depending on the depth k and the implementation details this can speed up compression (while at the same time slightly slowing down decompression).

In contrast to the usual presentation of the ST, we are using right shifts. This defines a slightly different version of the ST. The effect is that the order of the symbols occurring in some particular context is reversed. This makes sense, because in data compression the ST is applied to the reversal of a word. Hence, in the ST of the reversal of w the order of the symbols in some particular context is the same as in w. More formally, suppose $\overline{w} = x_0 c a_1 x_1 c a_2 x_2 \cdots c a_s x_s$ for $c \in \Sigma^+$ then in the sort transform of order $|c|$ of w, the order of the occurrences of the letters a_i is not changed. This property can enable better compression ratios on certain data.

While the standard permutation is induced by a sequence of letters (i.e., a word) we now generalize this concept to sequences of words. For a list of non-empty words $V = (v_1, \ldots, v_n)$ we now define the *k-order standard permutation* $\nu_{k,V}$ induced by V. As for the standard permutation, the first step is the construction of a new linear order $\preceq$ on $\{1, \ldots, n\}$. We define $i \preceq j$ by the condition

$$\mathrm{context}_k(v_i) < \mathrm{context}_k(v_j) \qquad \text{or} \qquad \mathrm{context}_k(v_i) = \mathrm{context}_k(v_j) \text{ and } i \leq j.$$

Let $j_1 \prec \cdots \prec j_n$ be the linearization of $\{1, \ldots, n\}$ according to this new order. The idea is that we sort the line numbers of $v_1, \ldots, v_n$ by first considering the k-order contexts and, if these are equal, then use the line numbers as tie-breaker. As before, the linearization according to $\preceq$ induces a permutation $\nu_{k,V}$ by setting $\nu_{k,V}(i) = j_i$. Now, $\nu_{k,V}(i)$ is the position of v_i if we are sorting V by k-order context such that the line numbers serve as tie-breaker. We set $M_k(v_1, \ldots, v_n) = (w_1, \ldots, w_n)$ where $w_i = v_{\nu_{k,V}(i)}$. Now, we are ready to define the sort transform of order k of a word w: Let $M_k([w]) = (w_1, \ldots, w_n)$; then $\mathrm{ST}_k(w) = \mathrm{last}(w_1) \cdots \mathrm{last}(w_n)$, i.e., we first sort all cyclic right-shifts of w by their k-order contexts (by using a stable sort method) and then we take the sequence of last symbols according to this new sort order as the image under ST_k. Since the tie-breaker relies on right-shifts, we have $\mathrm{ST}_0(w) = \overline{w}$, i.e., ST_0 is the reversal mapping. The k-order sort transform of w is the pair $(\mathrm{ST}_k(w), i)$ where i is the index of w in $M_k([w])$. As for the BWT, we see that the k-order sort transform is not bijective.

Next, we show that it is possible to reconstruct $M_k([w])$ from $\mathrm{ST}_k(w)$. Hence, it is possible to reconstruct w from the pair $(\mathrm{ST}_k(w), i)$ where i is the index of w in $M_k([w])$. The presentation of the back transform is as follows. First, we will introduce the *k-order context graph* G_k and we will show that it is possible to rebuild $M_k([w])$ from G_k. Then we will show how to construct G_k from $\mathrm{ST}_k(w)$. Again, the approach will be slightly more general than required at the moment; but we will be able to reuse it in the presentation of a bijective ST.

Let $V = ([u_1], \ldots, [u_s]) = (v_1, \ldots, v_n)$ be a list of words built from conjugacy classes $[u_i]$ of non-empty words u_i. Let $M = (w_1, \ldots, w_n)$ be an arbitrary permutation of the elements in V. We are now describing the edge-labeled directed graph $G_k(M)$ – the k-order context graph of M – which will be used later as a presentation tool for the inverses of the ST and the bijective ST. The vertices of $G_k(M)$ consist of all k-order contexts $\mathrm{context}_k(w)$ of words w occurring in M. We draw an edge (c_1, i, c_2) from context c_1 to context c_2 labeled by i if $c_1 = \mathrm{context}_k(w_i)$ and $c_2 = \mathrm{context}_k(r(w_i))$. Hence, every index $i \in \{1, \ldots, n\}$ of M defines a unique edge in $G_k(M)$. We can also think of $\mathrm{last}(w_i)$ as an additional implicit label of the edge (c_1, i, c_2), since $c_2 = \mathrm{context}_k(\mathrm{last}(w_i) c_1)$.

A *configuration* $(\mathcal{C}, c)$ of the k-order context graph $G_k(M)$ consists of a subset of the edges $\mathcal{C}$ and a vertex c. The idea is that (starting at context c) we are walking along the edges of $G_k(M)$ and whenever an edge is used, it is removed from the set of edges $\mathcal{C}$. We now define the transition

$$(\mathcal{C}_1, c_1) \xrightarrow{u} (\mathcal{C}_2, c_2)$$

from a configuration $(\mathcal{C}_1, c_1)$ to another configuration $(\mathcal{C}_2, c_2)$ with output $u \in \Sigma^*$ more formally. If there exists an edge in $\mathcal{C}_1$ starting at c_1 and if $(c_1, i, c_2) \in \mathcal{C}_1$ is the unique edge with the smallest label i starting at c_1, then we have the single-step transition

$$(\mathcal{C}_1, c_1) \xrightarrow{a} (\mathcal{C}_1 \setminus \{(c_1, i, c_2)\}, c_2) \qquad \text{where } a = \mathrm{last}(w_i)$$

If there is no edge in $\mathcal{C}_1$ starting at c_1, then the outcome of $(\mathcal{C}_1, c_1) \to$ is undefined. Inductively, we define $(\mathcal{C}_1, c_1) \xrightarrow{\varepsilon} (\mathcal{C}_1, c_1)$ and for $a \in \Sigma$ and $u \in \Sigma^*$ we have

$$(\mathcal{C}_1, c_1) \xrightarrow{au} (\mathcal{C}_2, c_2) \quad \text{if} \quad (\mathcal{C}_1, c_1) \xrightarrow{u} (\mathcal{C}', c') \text{ and } (\mathcal{C}', c') \xrightarrow{a} (\mathcal{C}_2, c_2)$$

for some configuration $(\mathcal{C}', c')$. Hence, the reversal $\overline{au}$ is the label along the path of length $|au|$ starting at configuration $(\mathcal{C}_1, c_1)$. In particular, if $(\mathcal{C}_1, c_1) \xrightarrow{u} (\mathcal{C}_2, c_2)$ holds, then it is possible to chase at least $|u|$ transitions starting at $(\mathcal{C}_1, c_1)$; vice versa, if we are chasing ℓ transitions then we obtain a word of length ℓ as a label. We note that successively taking the edge with the smallest label comes from the use of right-shifts. If we had used left-shifts we would have needed to chase largest edges for the following lemma to hold. The reverse labeling of the big-step transitions is motivated by the reconstruction procedure which will work from right to left.

Lemma 10. *Let $k \in \mathbb{N}$, $V = ([v_1], \ldots, [v_s])$, $c_i = \mathrm{context}_k(v_i)$, and $G = G_k(M_k(V))$. Let $\mathcal{C}_1$ consist of all edges of G. Then*

$$\begin{gathered}
(\mathcal{C}_1, c_1) \xrightarrow{v_1} (\mathcal{C}_2, c_1) \\
(\mathcal{C}_2, c_2) \xrightarrow{v_2} (\mathcal{C}_3, c_2) \\
\vdots \\
(\mathcal{C}_s, c_s) \xrightarrow{v_s} (\mathcal{C}_{s+1}, c_s).
\end{gathered}$$

Proof. Let $M_k(V) = (w_1, \ldots, w_n)$. Consider some index i, $1 \le i \le s$, and let $(u_1, \ldots, u_t) = ([v_1], \ldots, [v_{i-1}])$. Suppose that $\mathcal{C}_i$ consists of all edges of G except for those with labels $\nu_{k,V}(j)$ for $1 \le j \le t$. Let $q = |v_i|$. We write $v_i = a_1 \cdots a_q$ and $u_{t+j} = r^{j-1}(v_i)$, i.e., $[v_i] = (u_{t+1}, \ldots, u_{t+q})$. Starting with $(\mathcal{C}_{i,1}, c_{i,1}) = (\mathcal{C}_i, c_i)$, we show that the sequence of transitions

$$(\mathcal{C}_{i,1}, c_{i,1}) \xrightarrow{a_q} (\mathcal{C}_{i,2}, c_{i,2}) \xrightarrow{a_{q-1}} \cdots (\mathcal{C}_{i,q}, c_{i,q}) \xrightarrow{a_1} (\mathcal{C}_{i,q+1}, c_{i,q+1})$$

is defined. More precisely, we will see that the transition $(\mathcal{C}_{i,j}, c_{i,j}) \xrightarrow{a_{q+1-j}} (\mathcal{C}_{i,j+1}, c_{i,j+1})$ walks along the edge $\big(c_{i,j}, \nu_{k,V}(t+j), c_{i,j+1}\big)$ and hence indeed is labeled with the letter $a_{q+1-j} = \mathrm{last}(u_{t+j}) = \mathrm{last}(w_{\nu_{k,V}(t+j)})$. Consider the context $c_{i,j}$. By induction, we have $c_{i,j} = \mathrm{context}_k(u_{t+j})$ and no edge with label $\nu_{k,V}(\ell)$ for $1 \le \ell < t+j$ occurs in $\mathcal{C}_{i,j}$ while all other labels do occur. In particular, $(c_{i,j}, \nu_{k,V}(t+j), c_{i,j+1})$ for $c_{i,j+1} = \mathrm{context}_k(r(u_{t+j})) = \mathrm{context}_k(u_{t+j+1})$ is an edge in $\mathcal{C}_{i,j}$ (where $\mathrm{context}_k(r(u_{t+j})) = \mathrm{context}_k(u_{t+j+1})$ only holds for $j < q$; we will consider the case $j = q$ below). Suppose there were an edge $(c_{i,j}, z, c') \in \mathcal{C}_{i,j}$ with $z < \nu_{k,V}(t+j)$. Then $\mathrm{context}_k(w_z) = c_{i,j}$ and hence, w_z has the same k-order context as $w_{\nu_{k,V}(t+j)}$. But in this case, in the construction of $M_k(V)$ we used the index in V as a tie-breaker. It follows $\nu_{k,V}^{-1}(z) < t+1$ which contradicts the properties of $\mathcal{C}_{i,j}$. Hence, $(c_{i,j}, \nu_{k,V}(t+j), c_{i,j+1})$ is the edge with the smallest label starting at context $c_{i,j}$. Therefore, $\mathcal{C}_{i,j+1} = \mathcal{C}_{i,j} \setminus \{(c_{i,j}, \nu_{k,V}(t+j), c_{i,j+1})\}$ and $(\mathcal{C}_{i,j}, c_{i,j}) \xrightarrow{a_{q+1-j}} (\mathcal{C}_{i,j+1}, c_{i,j+1})$ indeed walks along the edge $(c_{i,j}, \nu_{k,V}(t+j), c_{i,j+1})$.

It remains to verify that $c_{i,1} = c_{i,q+1}$, but this is clear since $c_{i,1} = \mathrm{context}_k(u_{t+1}) = \mathrm{context}_k(r^q(u_{t+1})) = c_{i,q+1}$. □

Lemma 11. *Let $k \in \mathbb{N}$, $V = ([v_1], \ldots, [v_s])$, $M = M_k(V) = (w_1, \ldots, w_n)$, and $L = \mathrm{last}(w_1) \cdots \mathrm{last}(w_n)$. Then it is possible to reconstruct $G_k(M)$ from L.*

Proof. By Lemma 3 it is possible to reconstruct the contexts $c_i = \mathrm{context}_k(w_i)$. This gives the vertices of the graph $G_k(M)$. Write $L = a_1 \cdots a_n$. For each $i \in \{1, \ldots, n\}$ we draw an edge $(c_i, i, \mathrm{context}_k(a_i c_i))$. This yields the edges of $G_k(M)$. □

Corollary 12. *The k-order ST is invertible, i.e., given $(\mathrm{ST}_k(w), i)$ where i is the index of w in $M_k([w])$ one can reconstruct the word w.*

Proof. The construction of w consists of two phases. First, by Lemma 11 we can compute $G_k(M_k([w]))$. By Lemma 3 we can compute $c = \mathrm{context}_k(w)$ from $(\mathrm{ST}_k(w), i)$. In the second stage, we are using Lemma 10 for reconstructing w by chasing

$$(\mathcal{C}, c) \xrightarrow{w} (\emptyset, c)$$

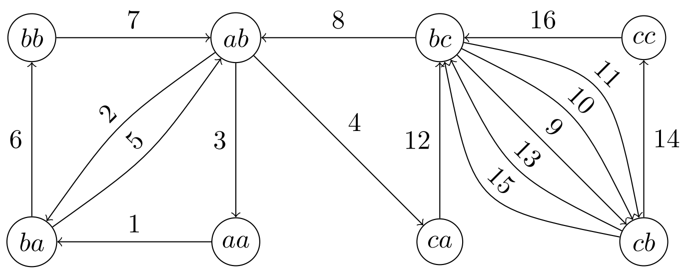
where $\mathcal{C}$ consists of all edges in $G_k(M_k([w]))$. □

Efficient implementations of the inverse transform rely on the fact that the k-order contexts of $M_k([w])$ are ordered. This allows the implementation of the k-order context graph G_k in a vectorized form [1,19,20,21].

Example 13. We compute the sort transform of order 2 of $w = bcbccbcbcabbaaba$ from Example 2. The list $M_2([w])$ is depicted in Figure 1(c). This yields the transform $(\mathrm{ST}_2(w), i) = (bbacabaacccbbcbb, 8)$ where $L = \mathrm{ST}_2(w)$ is the last column of the matrix $M_2([w])$ and w is the i-th element in $M_2([w])$. Next, we show how to reconstruct the input w from (L, i). The standard permutation induced by L is

$$\pi_L = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 3 & 5 & 7 & 8 & 1 & 2 & 6 & 12 & 13 & 15 & 16 & 4 & 9 & 10 & 11 & 14 \end{pmatrix}.$$

Note that π_L has four cycles $C_1 = (1, 3, 7, 6, 2, 5)$, $C_2 = (4, 8, 12)$, $C_3 = (9, 13)$, and $C_4 = (10, 15, 11, 16, 14)$. We obtain the context of order 2 of the j-th word by $c_j = \lambda_L \pi_L(j) \lambda_L \pi_L^2(j)$. In particular, $c_1 = aa$, $c_2 = c_3 = c_4 = ab$, $c_5 = c_6 = ba$, $c_7 = bb$, $c_8 = c_9 = c_{10} = c_{11} = bc$, $c_{12} = ca$, $c_{13} = c_{14} = c_{15} = cb$, and $c_{16} = cc$. With L and these contexts we can construct the graph $G = G_2(M_2([w]))$. The vertices of G are the contexts and the edge-labels represent positions in L. The graph G is depicted below:



We are starting at the context $c_i = c_8 = bc$ and then we are traversing G along the smallest edge-label amongst the unused edges. The sequence of the edge labels obtained this way is

$$(8, 2, 5, 3, 1, 6, 7, 4, 12, 9, 13, 10, 14, 16, 11, 15).$$

The labeling of this sequence of positions yields $\overline{w} = abaabbacbcbccbcb$. Since we are constructing the input from right to left, we obtain $w = bcbccbcbcabbaaba$.

6 The bijective sort transform

The bijective sort transform combines the Lyndon factorization with the ST. This yields a new algorithm which serves as a similar preprocessing step in data compression as the BWT. In a lot of applications, it can be used as a substitute for the ST. The proof of the bijectivity of the transform is slightly more technical than the analogous result for the bijective BWT. The main reason is that the bijective sort transform is less modular than the bijective BWT (which can be grouped into a 'Lyndon factorization part' and a 'Gessel-Reutenauer transform part' and which for example allows the use of different orders on the alphabet for the different parts).

For the description of the bijective ST and of its inverse, we rely on notions from Section 5. The bijective ST of a word w of length n is defined as follows. Let $w = v_s \cdots v_1$ with $v_s \geq \cdots \geq v_1$ be the Lyndon factorization of w. Let $M_k([v_1], \ldots, [v_s]) = (u_1, \ldots, u_n)$. Then the bijective ST of order k of w is $\mathrm{LST}_k(w) = \mathrm{last}(u_1) \cdots \mathrm{last}(u_n)$. That is, we are sorting the conjugacy classes of the Lyndon factors by k-order contexts and then take the sequence of the last letters. The letter L in LST_k is for *Lyndon*.

Theorem 14. *The bijective ST of order k is invertible, i.e., given $\mathrm{LST}_k(w)$ one can reconstruct the word w.*

Proof. Let $w = v_s \cdots v_1$ with $v_s \geq \cdots \geq v_1$ be the Lyndon factorization of w, let $c_i = \mathrm{context}_k(v_i)$, and let $L = \mathrm{LST}_k(w)$. By Lemma 11 we can rebuild the k-order context graph $G = G_k(M_k([v_1], \ldots, [v_s])) = (w_1, \ldots, w_n)$ from L. Let $\mathcal{C}_1$ consist of all edges in G. Then by Lemma 10 we see that

$$\begin{gathered}(\mathcal{C}_1, c_1) \xrightarrow{v_1} (\mathcal{C}_2, c_1) \\ \vdots \\ (\mathcal{C}_s, c_s) \xrightarrow{v_s} (\mathcal{C}_{s+1}, c_s).\end{gathered}$$

We cannot use this directly for the reconstruction of w since we do not know the Lyndon factors v_i and the contexts c_i.

The word v_1 is the first element in the list $M_k([v_1], \ldots, [v_s])$ because v_1 is lexicographically minimal and it appears as the first element in the list $([v_1], \ldots, [v_s])$. Therefore, by Lemma 3 we obtain $c_1 = \mathrm{context}_k(v_1) = \lambda_L \pi_L(1) \cdots \lambda_L \pi_L^k(1)$.

The reconstruction procedure works from right to left. Suppose we have already reconstructed $w' v_j \cdots v_1$ for $j \geq 0$ with w' being a (possibly empty) suffix of v_{j+1}. Moreover, suppose we have used the correct contexts $c_1, \ldots, c_{j+1}$. Consider the configuration $(\mathcal{C}', c')$ defined by

$$\begin{aligned}(\mathcal{C}_1, c_1) &\xrightarrow{v_1} (\mathcal{C}_2, c_1)\\ &\vdots\\ (\mathcal{C}_j, c_j) &\xrightarrow{v_j} (\mathcal{C}_{j+1}, c_j)\\ (\mathcal{C}_{j+1}, c_{j+1}) &\xrightarrow{w'} (\mathcal{C}', c')\end{aligned}$$

We assume that the following invariant holds: $\mathcal{C}_{j+1}$ contains no edges (c'', ℓ, c''') with $c'' < c_{j+1}$. We want to rebuild the next letter. We have to consider three cases. First, if $|w'| < |v_{j+1}|$ then

$$(\mathcal{C}', c') \xrightarrow{a} (\mathcal{C}'', c'')$$

yields the next letter a such that aw' is a suffix of v_{j+1}. Second, let $|w'| = |v_{j+1}|$ and suppose that there exists an edge $(c_{j+1}, \ell, c''') \in \mathcal{C}'$ starting at $c' = c_{j+1}$. Then there exists a word v' in $[v_{j+2}], \ldots, [v_s]$ such that $\mathrm{context}_k(v') = c_{j+1}$. If $\mathrm{context}_k(v_{j+2}) \neq c_{j+1}$ then from the invariant it follows that $\mathrm{context}_k(v_{j+2}) > c_{j+1} = \mathrm{context}_k(v')$. This is a contradiction, since v_{j+2} is minimal among the words in $[v_{j+2}], \ldots, [v_s]$. Hence, $\mathrm{context}_k(v_{j+2}) = c_{j+2} = c_{j+1}$ and the invariant still holds for $\mathcal{C}_{j+2} = \mathcal{C}'$. The last letter a of v_{j+2} is obtained by

$$(\mathcal{C}', c') = (\mathcal{C}_{j+2}, c_{j+2}) \xrightarrow{a} (\mathcal{C}'', c'').$$

The third case is $|w'| = |v_{j+1}|$ and there is no edge $(c_{j+1}, \ell, c''') \in \mathcal{C}'$ starting at $c' = c_{j+1}$. As before, v_{j+2} is minimal among the (remaining) words in $[v_{j+2}], \ldots, [v_s]$. By construction of G, the unique edge $(c'', \ell, c''') \in \mathcal{C}'$ with the minimal label ℓ has the property that $w_\ell = v_{j+2}$. In particular, $c'' = c_{j+2}$. Since v_{j+2} is minimal, the invariant for $\mathcal{C}_{j+2} = \mathcal{C}'$ is established. In this case, the last letter a of v_{j+2} is obtained by

$$(\mathcal{C}_{j+2}, c_{j+2}) \xrightarrow{a} (\mathcal{C}'', c''').$$

We note that we cannot distinguish between the first and the second case since we do not know the length of v_{j+1}, but in both cases, the computation of the next symbol is identical. In particular, in contrast to the bijective BWT we do not implicitly recover the Lyndon factorization of w. ☐

We note that the proof of Theorem 14 heavily relies on two design criteria. The first one is to consider $M_k([v_1], \ldots, [v_s])$ rather than $M_k([v_s], \ldots, [v_1])$, and the second is to use right-shifts rather than left-shifts. The proof of Theorem 14 yields the following algorithm for reconstructing w from $L = \mathrm{LST}_k(w)$:

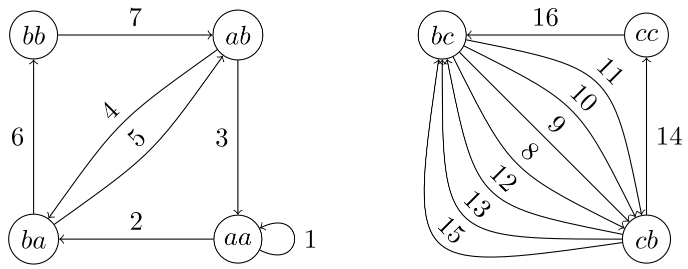
(1) Compute the k-order context graph $G = G_k$ and the k-order context c_1 of the last Lyndon factor of w.
(2) Start with the configuration $(\mathcal{C}, c)$ where $\mathcal{C}$ contains all edges of G and $c := c_1$.
(3) If there exists an outgoing edge starting at c in the set $\mathcal{C}$, then
   - Let (c, ℓ, c') be the edge with the minimal label ℓ starting at c.
   - Output $\lambda_L(\ell)$.
   - Set $\mathcal{C} := \mathcal{C} \setminus \{(c, \ell, c')\}$ and $c := c'$.
   - Continue with step (3).
(4) If there is no outgoing edge starting at c in the set $\mathcal{C}$, but $\mathcal{C} \neq \emptyset$, then
   - Let $(c', \ell, c'') \in \mathcal{C}$ be the edge with the minimal label ℓ.

– Output $\lambda_L(\ell)$.
– Set $\mathcal{C} := \mathcal{C} \setminus \{(c', \ell, c'')\}$ and $c := c''$.
– Continue with step (3).

(5) The algorithm terminates as soon as $\mathcal{C} = \emptyset$.

The sequence of the outputs is the reversal $\overline{w}$ of the word w.

Example 15. We consider the word $w = bcbccbcbcabbaaba$ from Example 2 and its Lyndon factorization $w = v_6 \cdots v_1$ where $v_6 = bcbcc$, $v_5 = bc$, $v_4 = bc$, $v_3 = abb$, $v_2 = aab$, and $v_1 = a$. For this particular word w the bijective Burrows-Wheeler transform and the bijective sort transform of order 2 coincide. From Example 9, we know $L = \mathrm{LST}_2(w) = \mathrm{BWTS}(w) = abababaccccbbcbb$ and the standard permutation π_L. As in Example 13 we can reconstruct the 2-order contexts $c_1, \ldots, c_{16}$ of $M_2([v_1], \ldots, [v_6])$: $c_1 = c_2 = aa$, $c_3 = c_4 = ab$, $c_5 = c_6 = ba$, $c_7 = bb$, $c_8 = c_9 = c_{10} = c_{11} = bc$, $c_{12} = c_{13} = c_{14} = c_{15} = cb$, and $c_{16} = cc$. With L and the 2-order contexts we can construct the graph $G = G_k(M_2([v_1], \ldots, [v_6]))$:



We are starting with the edge with label 1 and then we are traversing G along the smallest unused edges. If we end in a context with no outgoing unused edges, then we are continuing with the smallest unused edge. This gives the sequence $(1, 2, 5, 3)$ after which we end in context aa with no unused edges available. Then we continue with the sequences $(4, 6, 7)$ and $(8, 12, 9, 13, 10, 14, 16, 11, 15)$. The complete sequence of edge labels obtained this way is

$$(1, 2, 5, 3,\ 4, 6, 7,\ 8, 12, 9, 13, 10, 14, 16, 11, 15)$$

and the labeling of this sequence with λ_L yields $\overline{w} = abaabbacbcbccbcb$. As for the ST, we are reconstructing the input from right to left, and hence we get $w = bcbccbcbcabbaaba$.

7 Summary

We discussed two bijective variants of the Burrows-Wheeler transform (BWT). The first one is due to Scott. Roughly speaking, it is a combination of the Lyndon factorization and the Gessel-Reutenauer transform. The second variant is derived from the sort transform (ST); it is the main contribution of this paper. We gave full constructive proofs for the bijectivity of both transforms. As a by-product, we provided algorithms for the inverse of the BWT and the inverse of the ST. For the latter, we introduced an auxiliary graph structure — the k-order context graph. This graph yields an intermediate step in the computation of the inverse of the ST and the bijective ST. It can be seen as a generalization of the cycle decomposition of the standard permutation — which in turn can be used as an intermediate step in the computation of the inverse of the BWT and the bijective BWT.

Acknowledgments. The author would like to thank Yossi Gil and David A. Scott for many helpful discussions on this topic as well as Alexander Lauser, Antonio Restivo, and the anonymous referees for their numerous suggestions which improved the presentation of this paper.

References

1. D. Adjeroh, T. Bell, and A. Mukherjee: *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*, Springer Publishing Company, Incorporated, 2008.
2. Z. Arnavut and M. Arnavut: *Investigation of block-sorting of multiset permutations*. Int. J. Comput. Math., 81(10) 2004, pp. 1213–1222.
3. B. Balkenhol and S. Kurtz: *Universal data compression based on the Burrows-Wheeler transformation: Theory and practice*. IEEE Trans. Computers, 49(10) 2000, pp. 1043–1053.
4. T. Bell, I. H. Witten, and J. G. Cleary: *Modeling for text compression*. ACM Comput. Surv., 21(4) 1989, pp. 557–591.
5. M. Burrows and D. J. Wheeler: *A block-sorting lossless data compression algorithm*, Tech. Rep. 124, Digital SRC Research Report, 1994.
6. K. T. Chen, R. H. Fox, and R. C. Lyndon: *Free differential calculus, IV — The quotient groups of the lower central series*. Ann. Math., 68(1) 1958, pp. 81–95.
7. J. G. Cleary and I. H. Witten: *Data compression using adaptive coding and partial string matching*. IEEE Trans. Commun., 32(4) 1984, pp. 396–402.
8. M. Crochemore, J. Désarménien, and D. Perrin: *A note on the Burrows-Wheeler transformation*. Theor. Comput. Sci., 332(1-3) 2005, pp. 567–572.
9. J.-P. Duval: *Factorizing words over an ordered alphabet*. J. Algorithms, 4(4) 1983, pp. 363–381.
10. N. J. Fine and H. S. Wilf: *Uniqueness theorems for periodic functions*. Proc. Amer. Math. Soc., 16 1965, pp. 109–114.
11. I. M. Gessel and C. Reutenauer: *Counting permutations with given cycle structure and descent set*. J. Comb. Theory, Ser. A, 64(2) 1993, pp. 189–215.
12. J. Gil and D. A. Scott: *A bijective string sorting transform*, submitted.
13. A. Lempel and J. Ziv: *A universal algorithm for sequential data compression*. IEEE Trans. Inform. Theory, 23(3) 1977, pp. 337–343.
14. A. Lempel and J. Ziv: *Compression of individual sequences via variable-rate coding*. IEEE Trans. Inform. Theory, 24(5) 1978, pp. 530–536.
15. M. Lothaire, ed., *Combinatorics on Words*, Addison-Wesley, Reading, MA, 1983.
16. S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino: *An extension of the Burrows Wheeler transform and applications to sequence comparison and data compression*, in Combinatorial Pattern Matching, CPM 2005, Proceedings, vol. 3537 of LNCS, Springer, 2005, pp. 178–189.
17. 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.
18. G. Manzini: *An analysis of the Burrows-Wheeler transform*. Journal of the ACM, 48(3) 2001, pp. 407–430.
19. G. Nong and S. Zhang: *Unifying the Burrows-Wheeler and the Schindler transforms*, in Data Compression Conference, DCC 2006. Proceedings, IEEE Computer Society, 2006, p. 464.
20. G. Nong and S. Zhang: *Efficient algorithms for the inverse sort transform*. IEEE Trans. Computers, 56(11) 2007, pp. 1564–1574.
21. G. Nong, S. Zhang, and W. H. Chan: *Computing inverse ST in linear complexity*, in Combinatorial Pattern Matching, CPM 2008, Proceedings, vol. 5029 of LNCS, Springer, 2008, pp. 178–190.
22. M. Schindler: *A fast block-sorting algorithm for lossless data compression*, in Data Compression Conference, DCC 1997. Proceedings, IEEE Computer Society, 1997, p. 469.
23. D. A. Scott: *Personal communication*, 2009.

On the Usefulness of Backspace

Shmuel T. Klein[1] and Dana Shapira[2]

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

[2] Department of Computer Science
Ashkelon Acad. College
Ashkelon, Israel
shapird@ash-college.ac.il

Abstract. The usefulness of a backspace character in various applications of Information Retrieval Systems is investigated. While not being a character in the initial sense of the word, a backspace can be defined as being a part of an extended alphabet, thereby enabling the enhancement of various algorithms related to the processing of queries in Information retrieval. We bring examples of three different application areas.

1 Introduction

A large textual database can be made accessible by means of an Information Retrieval System (IRS), a set of procedures which process the given text to find the most relevant passages to a specific information request. This request is usually formulated according to some given rigid query syntax, but in fact the formulation of a query is an art in which the user has to find the right balance between a choice of query terms that may be too broad and others that could be too restrictive.

The present work is an extension of an earlier study of the *negation operator* as it appears in its various forms in Information Retrieval applications [7]. We restrict attention to the Boolean query model, as in Chang et al. [1], though several alternatives are available, like the classical vector space model [9], the probabilistic model [11], and others. The natural approach of most users to query formulation involves the choice of keywords that best describe their information needs. They often overlook the possibility, which sometimes could even be a necessity, of choosing also a *negative* set, that is, a set of keywords which should *not* appear in the vicinity of some others, thereby achieving improved precision. But the use of negation might in certain cases be tricky and is not always symmetrical to the use of positive terms.

We now turn to the usefulness of an element which is intrinsically negative, namely a *backspace* character. A backspace is not really a character in the classical sense, as it is not explicitly written or used to form any words, but keyboards contain a backspace key and standard codes like ASCII assign it a codeword, so programmers, rather than users, consider backspaces just as any other printable character. The purpose of this paper is to show that in spite of it not representing any concrete entity, a backspace can be a useful tool in various different applications related to the implementation of IR systems. The intention is not to present a comprehensive investigation of all the possible applications of backspaces, but rather to emphasize its usefulness by means of some specific examples in different areas of Information Retrieval. The next section deals with the processing of large numbers in an IRS and suggests a solution based on using backspaces as part of the elements that might be retrieved. In Section 3 we

Shmuel T. Klein, Dana Shapira: On the Usefulness of Backspace, pp. 80–89.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

consider compression aspects of the textual databases within an IRS, and show how a model including a backspace may lead to improved savings. As a last example, we show in Section 4 how the use of backspaces may lead to another time/space tradeoff in an application to the fast decoding of Huffman encoded texts.

2 Dealing with large numbers

2.1 Syntax definition

To enable the subsequent discussion, one has first to define a query language syntax. Most search engines allow simple queries, consisting just of a set of keywords, such as

$$A_1 \quad A_2 \ \ldots \ A_m, \tag{1}$$

which should retrieve all the documents in the underlying textual database in which all the terms A_i occur at least once. Often, some kind of stemming is automatically performed on all the terms of the text during the construction of the database, as well as online on the query terms [5]. *Negating* one or more keywords in the query means that one is interested in prohibiting the occurrence of the negated terms in the retrieved documents. A further extension of the query syntax accommodates also tools for *proximity* searches. The idea is that a user may wish to limit the location of possible occurrences of the query terms to be, if not adjacent, then at least quite close to each other. Many query languages support, in addition to the loose formulations of (1), also an *exact phrase* option. This should, however, be used with care, as one has to guess all possible occurrence patterns of the query terms, and failing to do so may yield reduced recall.

This leads to the following generalization. Consider a query containing only positive terms as consisting of m keywords and $m-1$ binary distance constraints, as in

$$A_1 \ \ (l_1 : u_1) \ \ A_2 \ \ (l_2 : u_2) \ \ \ldots \ \ A_{m-1} \ \ (l_{m-1} : u_{m-1}) \ \ A_m. \tag{2}$$

This is a conjunctive query, requiring all the keywords A_i to occur within the given metrical constraints specified by l_i, u_i, which are integers satisfying $-\infty < l_i \le u_i < \infty$ for $1 \le i < m$, with the couple $(l_i : u_i)$ imposing a lower and upper limit on the distance from an occurrence of A_i to one of A_{i+1}. The distance is measured in words, and usually restricts, in addition to the specific constraints imposed by the $(l_i : u_i)$ pairs, all the terms to appear within some predefined textual unit, like the same sentence. Negative distance means that A_i may follow A_{i+1} rather than precede it. In the presence of negated keywords, association of keywords to metrical constraints should be to the left, unless there is no such option, that is, all the keywords to the left of the leftmost non-negated one will be right associated (each query must have at least one non-negated keyword). An example of left association could be A (1:3) $-$B (1:5) C, meaning that we seek an occurrence of C following an occurrence of A by 1 to 5 words, but such that there is no occurrence of B in the range of 1 to 3 words after A. In the query $-$D (1:1) E, an occurrence of E should not be preceded immediately by an occurrence of D.

Such a query language is used for over thirty years at the *Responsa Retrieval Project* [4,2]. Even more extended features, mixing Boolean operators with proximity constraints between certain keywords can be found in the *word pattern* models for Boolean Information Retrieval $\mathcal{WP}$ and $\mathcal{AWP}$ [12].

2.2 The use of backspace for large numbers

The problem with large numbers in an IRS is that there are potentially too many of them. If we store 20,000 pages of a running text, including also the page numbers, at least all the numbers up to 20,000 will appear, which can be an increasingly large part of the dictionary. A real-life application of this problem is mentioned in [8]. A possible solution is to break long strings of digits into blocks of at most k digits each. For $k = 4$, the number `1234567` would thus be stored as 2 consecutive items: `1234` and `567`. We have thus bounded the number of possible numbers by 10^k, independently of the length of the text.

There is however a new problem, namely one of precision. If the query asks for the location of a number such as `5678`, the system would also retrieve certain occurrences of numbers having `5678` as substring, like `123456789`. We therefore need some indicator, telling if the string is surrounded by blanks or not. The two obvious solutions, of storing indices of all the blanks, or using special treatment on all queries involving numbers, have to be ruled out, the first because of the increased space requirements, the second on the basis of the additional processing time that would be required to check the vicinity of each occurrence of a number.

A possible solution to the problem can be based on the fact that we are not restricted to deal only with standard tokens such as characters or character strings. In fact, we need some mechanism to overcome our implicit assumption that all the words in the text, and therefore also in the query, are separated by blanks. Let us thus formalize the setup.

We assume that the words stored in the dictionary are full words as they appear in the text, *each followed by a blank*. For example, we might find there the words `house⊔`, `the⊔`, etc, where the ⊔ is used to visualize the terminating blank. Of course, these blanks are not actually stored, but they are conceptually present. A query asking for `House of Lords` will thus generate 3 accesses to the dictionary, with items `House⊔`, `of⊔` and `Lords⊔`, and one would check if there is an occurrence of these three items in the same sentence, having consecutive relative indices.

A similar treatment will also be given to numbers of up to k digits in length. For a longer number, the fact that there are no spaces within it will be stored by means of a *back-space* item, BS. For example, the number `1234567890` will be stored as a sequence of the following items: `1234⊔`, BS, `5678⊔`, BS, `90⊔`. The backspace will be treated like any other word: it will have a consecutive numbering, and all its occurrences will be referenced in the concordance. For example, consider the phrase:

`I declared an income of 1000000 on my last 10 1040 forms.`

When inverting the text to build the dictionary and the concordance, this will be parsed as `I declared an income of 1000 BS 000 on my last 10 1040 forms`, with the words numbered 1 to 14, respectively. In particular, the BS has index 7. Note the absence of a backspace between 10 and 1040, since these are indeed two separate numbers, and the space between them is a part of the original text.

Punctuation signs are traditionally attached to the preceding word, which should therefore loose its trailing blank. This can be implemented by considering each punctuation sign as if it were preceded by BS. The phrase `Mr. Jones` would thus be encoded by `Mr⊔ BS . Jones`.

It is true that having the backspaces numbered just like words may disrupt proximity searches which do not require adjacency. In the query `solve` $(-10 : 10)$

`differential`, the request is to find the query terms at a distance from up to 10 words from each other, without caring which term precedes the other. The occurrence of a backspace between the occurrences of these terms in the text may lead to the wrong numbering, and therefore cause retrieval of passages which would not have complied with the strict original definition of the distances, or it may imply the non-retrieval of other passages which should have been retrieved. However, the same is true already if large numbers are split, even when no backspaces are used. The current proposition is thus only valid if either:

– no proximity searches other than immediate adjacency are supported;
– or the software is adapted to deal with the correct numbering also in the presence of backspaces and number splits;
– or that strict adherence to the exact metrical constraints is not deemed critical. In most queries using large distances, one can hardly justify the use of $(-10 : 10)$ rather than $(-11 : 11)$ or $(-9 : 9)$, so even if backspaces or number splits effectively reduce the range of the query, this does not necessarily lead to a worse recall/precision tradeoff. In other words, the fact that a text passage does not strictly obey to the constraints imposed by the query does not yet mean that the passage is absolutely non-relevant.

Table 1 brings a few examples of queries including large numbers, and how they can be processed by means of the backspace item.

Searching for	Submit query	Comments
`234`	−BS `234`	the negated backspace to avoid retrieval of, e,g, `8888234`
`2000 1040`	−BS `2000 1040` −BS	
`12345678`	−BS `1234` BS `5678` −BS	
`1234567`	−BS `1234` BS `567`	Note that since the last part of the number has less than $k = 4$ digits, it is not necessary to add the −BS, which was used in the previous example to prevent the retrieval of `123456789` for example
`user@address.com`	`user @` BS `address .` BS `com`	Note the absence of BS preceding `@` and the dot, since these are punctuation signs

Table 1. Examples of the use of a backspace character

3 Compressing the text of an IR System

At the heart of any Information Retrieval system is the raw text, which is usually stored in some compressed form. A myriad of different text compression schemes has been suggested, but a full description is beyond the scope of this work, and the reader is referred to [14] for a description of many of these methods.

One class of compression techniques, often called *statistical*, uses variable length codewords to encode the different characters. Compression is achieved by assigning shorter codewords to characters occurring at higher frequency, and an optimal assignment of codewords lengths, once the character frequencies are known, is given by Huffman's algorithm [6].

But a Huffman code, applied on individual characters, does not achieve a good compression ratio, because adjacent characters are encoded as if they were independent, which is not the case for natural language texts. In order to exploit also inter-character correlations, one may extend the set of elements to be encoded, to include character pairs, triplets, etc., or even entire words. In the latter case, the Huffman tree could be huge, with a leaf for each of the different words in the text, but this overhead may be acceptable as the list of different words, also known as the *dictionary* of the Information Retrieval system, is needed anyway. The compression obtained by the word oriented Huffman code is excellent and competes with that of the best methods.

A text consists, however, not just of a sequence of juxtaposed words, but these words are separated by blanks and other punctuation signs, which have also to be encoded. One possibility is to use two, rather than a single Huffman code [8], one for the words, and another for the *non-words* separating them, keeping strict alternation to avoid having to encode an indicator of which code is being used. This method is referred to as the *Huffword* scheme [14].

But the overwhelming majority of the non-words are blanks, so their encoding would be wasteful. As alternative, one may use the idea of the backspace as above. A single Huffman code can be used, for which the set of elements to be encoded consists of:

1. the words including a trailing blank. This is the same idea as in the definition of the dictionary in the IR application of Section 1, in which the blank following a word is considered an integral part of the word itself;
2. punctuation signs, also including trailing blanks, but being preceded by a conceptual backspace to attach them to the word they follow;
3. the backspace character, to deal also with the exceptions of the attachment rules for punctuation signs.

Every text can be parsed into a sequence of such elements, and a *single* Huffman code can be built on the basis of the frequencies obtained from this parsing.

We tested the approach on two textual databases of different sizes and languages: the Bible (King James version) in English, and the French version of the European Union's JOC corpus, a collection of pairs of questions and answers on various topics used in the ARCADE evaluation project [13]. Table 2 reports on the basic statistics and on the compression results.

File	Size	Huffword	BShuff	# exceptions	gzip	bzip
English	3.1 MB	3.91	3.97	2006	3.28	4.41
French	7.1 MB	3.98	4.03	24430	3.27	4.63

Table 2. Comparing backspace based compression

The values give the compression ratio, which is the size of the original file divided by the size of the compressed file. We see that in both cases, the approach using a backspace (BShuff) is slightly better than what can be obtained by Huffword. The table also contains data for compression by gzip (with parameter -9 for maximal compression) and bzip2. Both Huffword and BShuff are preferable to gzip, but bzip is considerably better. It should however be noted that the comparison to the adaptive Lempel Ziv based gzip or Burrows-Wheeler based bzip is not necessarily meaningful: adaptive methods require the whole file to be decompressed sequentially and do not allow partial decoding of selected sub-parts, as can be obtained by the static Huffman based methods. In certain applications, the zip methods are thus not plausible alternatives to those treated here, even if their compression is better.

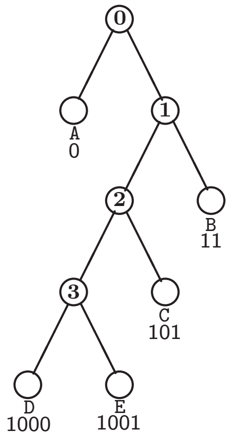
4 Blockwise decoding of Huffman codes

We now turn to a last example, also connected to the compression of large texts in an IRS, but concentrating on the decoding. Indeed, decoding might be of higher importance than encoding, since the latter is only done once, when the system is built, while efficient decoding is critical for getting a good response time each time a query is being submitted. However, the decoding of variable length codes, and in particular Huffman codes, can be slow, because the end of each codeword has to be detected by the decoding algorithm itself, and the implied manipulations of the encoded text at the bit level can have a negative impact on the decoding speed. But efficient decoding of k bits in every iteration, for $k > 1$, rather than only a single one, is made possible by using a set of m auxiliary tables, which are prepared in advance for every given prefix code. The method has first been mentioned in [3], and has since been reinvented several times, for example in [10].

4.1 Basic decoding scheme

The basic scheme is as follows. The number of entries in each table is 2^k, corresponding to the 2^k possible values of the k-bit patterns. Each entry is of the form (W, j), where W is a sequence of characters and j $(0 \leq j < m)$ is the index of the next table to be used. The idea is that entry i, $0 \leq i < 2^k$, of table number 0 contains, first, the longest possible decoded sequence W of characters from the k-bit block representing the integer i (W may be empty when there are codewords of more than k bits); usually some of the last bits of the block will not be decipherable, being the prefix P of more than one codeword; j will then be the index of the table corresponding to that prefix (if $P = \Lambda$, where Λ denotes the empty string, then $j = 0$). Table number j is constructed in a similar way except for the fact that entry i will contain the analysis of the bit pattern formed by the prefixing of P to the binary representation of i. We thus need a table for every possible proper prefix of the given codewords; the number of these prefixes is obviously equal to the number of internal nodes of the appropriate Huffman-tree (the root corresponding to the empty string and the leaves corresponding to the codewords), so that $m = N - 1$, where N is the size of the alphabet.

More formally, let P_j, $0 \leq j < N - 1$, be an enumeration of all the proper prefixes of the codewords (no special relationship needs to exist between j and P_j, except for the fact that $P_0 = \Lambda$). In table j corresponding to P_j, the i-th entry, $T(j, i)$, is defined as follows: let B be the bit-string composed of the juxtaposition of P_j to the left of the k-bit binary representation of i. Let W be the (possibly empty) longest sequence of characters that can be decoded from B, and P_ℓ the remaining undecipherable bits of B; then $T(j, i) = (W, \ell)$.



Entry	Pattern for Table 0	Table 0		Table 1		Table 2		Table 3	
		W	ℓ	W	ℓ	W	ℓ	W	ℓ
0	000	AAA	0	D	0	DA	0	DAA	0
1	001	AA	1	E	0	D	1	DA	1
2	010	A	2	CA	0	EA	0	D	2
3	011	AB	0	C	1	E	1	DB	0
4	100	—	3	BAA	0	CAA	0	EAA	0
5	101	C	0	BA	1	CA	1	EA	1
6	110	BA	0	B	2	C	2	E	2
7	111	B	1	BB	0	CB	0	EB	0

Figure 1. Huffman tree and partial decoding tables

As an example, consider the alphabet {A,B,C,D,E}, with codewords {0, 11, 101, 1000, 1001} respectively, and choose $k = 3$. There are 4 possible proper prefixes: Λ,1,10,100, hence 4 corresponding tables indexed 0,1,2,3 respectively, and these are given in Figure 1, along with the corresponding Huffman tree that has its internal nodes numbered accordingly. The column headed 'Pattern' contains for every entry the binary string which is decoded in Table 0; the binary strings which are decoded by Tables 1, 2 and 3 are obtained by prefixing '1', '10' or '100', respectively, to the strings in 'Pattern'. If the encoded text, which serves as input string to this decoding routine, consists of 100 101 110 000 101, we access sequentially Table 0 at entry 4, Table 3 at entry 5, Table 1 at entry 6, Table 2 at entry 0 and Table 0 at entry 5, yielding the output strings EA B DA C.

The general decoding routine is thus extremely simple. Let $M[f;t]$ denote the substring of the encoded string serving as input stream to the decoding, that starts at bit number f and extends up to and including bit number t; let j be the index of the currently used table and $T(j,\ell)$ the ℓ-th entry of table j:

$j \leftarrow 0$
for $\quad f \leftarrow 1 \quad$ to $\quad$ length of input $\quad$ do
$\qquad$ (output, j) $\leftarrow T(j, M[f; f+k-1])$
$\qquad f \leftarrow f + k$

The larger is k, the greater is the number of characters that can be decoded in a single iteration, thus transferring a substantial part of the decoding time to the preprocessing stage. The size of the tables, however, is $\Omega(N2^k)$, so it grows exponentially with k, and may become prohibitive for large alphabets and even moderately large k. For example, if $N = 30000$, k is chosen as 16 and every table entry requires 6 bytes, the tables, which should be stored in RAM, would need about 11 GB! Even if such amounts of memory were available, the number of cache misses and page faults would void a significant part of the benefits in time savings incurred by the reduced number of processing steps.

4.2 Using backspaces to get another time/space tradeoff

The number of tables, and thus the storage requirements, can be reduced, by conceptually modifying the text with the introduction of backspace characters at certain locations. Particularly in the case of a large alphabet, the blocksize k could be chosen smaller than the longest codeword, and tables would be constructed not for all the internal nodes, but only for those on levels that are multiples of k, that is for the root (level 0), and all the internal nodes on levels k, $2k$, $3k$, etc. There is an obvious gain in the number of tables, which comes at the price of a slower decoding pace: as before, the table entries consist first of the decoding W of a bit string B obtained by concatenating some prefix to the binary representation of the entry index. If B is not completely decipherable, the remainder P_j is used in the previous setting as index to the next table. For the new variant, if $|P_j|$, the length of the remainder, is smaller than k, then no corresponding table has been stored, so these $|P_j|$ bits have to be reread in the next iteration. This is enforced by adding, after the remainder, $|P_j|$ backspaces into the text.

Entry	Pattern for Table 0	Table 0 W	ℓ	b	Table 3 W	ℓ	b
0	000	AAA	0	0	DAA	0	0
1	001	AA	0	1	DA	0	1
2	010	A	0	2	D	0	2
3	011	AB	0	0	DB	0	0
4	100	—	3	0	EAA	0	0
5	101	C	0	0	EA	0	1
6	110	BA	0	0	E	0	2
7	111	B	0	1	EB	0	0

Figure 2. Reduced partial decoding tables

It should however be noted, that the modification of the text is only conceptual, and will manifest itself only in the modified decoding routine; the encoded text itself need not to be touched, so there is no loss in compression efficiency, only in decoding speed. The table entries for the new algorithm are thus extended to include a third component: a back skip b, indicating how many backspaces should have been introduced at that particular point, which is equivalent to the number of bits the pointer into the input string should be moved back. Using the above notations, $T(j, i)$ will consist of the triplet (W, ℓ, b), and the decoding routine is given by

$$
\begin{array}{l}
j \leftarrow 0 \qquad back \leftarrow 0 \\
\textsf{for} \quad f \leftarrow 1 \quad \textsf{to} \quad \textsf{length of input} \quad \textsf{do} \\
\qquad (\textsf{output}, j, back) \leftarrow T(j, M[f; f+k-1]) \\
\qquad f \leftarrow f + k - back
\end{array}
$$

As example, consider the same Huffman tree and the same input string as above with $k = 3$. Only two tables remain, Table 0 and Table 3, given in Figure 2. Decoding is now performed by six table accesses rather than only 5 with the original tables, using the sequence of blocks 100, 101, BS11, BS00, 001, BS01 to access tables 0, 3, 0, 0, 3, 0, respectively, where the backspace BS indicates that the preceding bit is read again, resulting in an overlap of the currently decoded block with the preceding one.

Figure 3 shows the input string and above it its parsing into codewords using the standard Huffman decoding. Below appear first the parsing into consecutive k-bit blocks using the original tables, then the parsing into partially overlapping k-bit blocks with the tables relying on the backspaces. Note that for simplicity, we do not deal here with the case that the last block may be shorter than k bits.

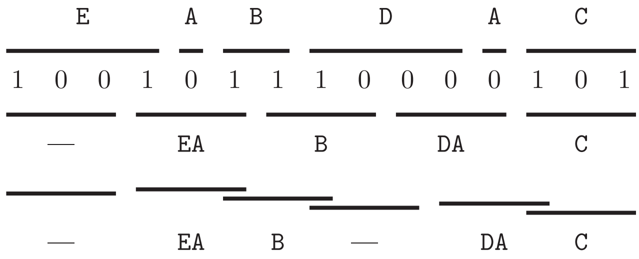


Figure 3. Example using original and reduced tables

To get some experimental results, we used the King James Bible (KJB) as above, and Wall Street Journal (WSJ) issues that appeared in 1989. Huffman codes were generated for large "alphabets", consisting of entire words. Some of the relevant statistics are given in Table 3, and the results are presented in Table 4. The row headed Bit corresponds to the regular bit per bit Huffman decoding. The next row brings the values of the Partial decoding Tables of [3] described in Section 4.1 above, and the row with title reduced corresponds to the variant with the backspaces.

For each of the test databases, the first column brings the maximal size k of the block of bits that is decoded as one unit. The next column, headed bpa is in fact the average value of k used during the decoding. It is the average number of decoded **b**its **p**er table **a**ccess, evaluated as the size of the compressed file in bits divided by the total number of such accesses. The next column brings timing results, in terms of the number of MB that can be processed per second on our test machine. The time for the variant with the full tables for WSJ could not be evaluated, due to exceeding RAM requirements. The last column, headed RAM, gives the size of the required auxiliary storage in MB. We see that the Reduced Tables saved 50 to 80 % of the space required by the partial decoding tables, while using the same k, reducing the decoding rate only by about 20 %.

	full size	compression ratio	number of words	average word length
KJB	3.1 MB	5.15 MB	11669	8.8 bit
WSJ	36.5 MB	5.05 MB	115136	11.2 bit

Table 3. Statistical data on test files

	KJB				WSJ			
	k	bpa	MB/sec	RAM	k	bpa	MB/sec	RAM
Bit	1	1	10.1	0.21	1	1	6,6	2.1
Tables	8	8	0.4	17	8	8	—	197
reduced	8	6.37	13.7	8.7	8	6.35	7.6	34.1

Table 4. Experimental comparison of decoding

5 Conclusion

We presented three examples of applications in various areas of Information Retrieval Systems, for which the inclusion of a backspace character in the alphabet may lead to improved performance. The main message we hoped to convey in this study, is that the definition of alphabets in the broadest sense as used in IR systems, does not have to be restricted to collections of classical items such as letters, words or strings, but may be extended to include also conceptual elements such as a backspace, which, even if not materialized as the other elements, may at times have helpful usages.

References

1. K. C.-C. Chang, H. Garcia-Molina, and A. Paepcke: *Predicate rewriting for translating Boolean queries in a heterogeneous information system.* ACM Trans. on Information Systems, 17(1) 1999, pp. 1–39.
2. Y. Choueka: *Responsa: A full-text retrieval system with linguistic processing for a 65-million word corpus of jewish heritage in Hebrew.* IEEE Data Eng. Bull., 14(4) 1989, pp. 22–31.
3. Y. Choueka, S. T. Klein, and Y. Perl: *Efficient variants of Huffman codes in high level languages*, in Proc. 8-th ACM-SIGIR Conference, Montreal, Canada, 1985, pp. 122–131.
4. A. S. Fraenkel: *All about the Responsa Retrieval Project you always wanted to know but were afraid to ask, Expanded Summary.* Jurimetrics J., 16 1976, pp. 149–156.
5. W. B. Frakes: *Stemming algorithms*, in Information Retrieval, Data Structures and Algorithms, W. B. Frakes and R. Baeza-Yates, eds., Prentice Hall, NJ, 1992, pp. 131–160.
6. D. Huffman: *A method for the construction of minimum redundancy codes*, in Proc. of the IRE, vol. 40, 1952, pp. 1098–1101.
7. S. T. Klein: *On the use of negation in Boolean IR queries.* Information Processing & Management, 45 2009, pp. 298–311.
8. A. Moffat and Z. J.: *Adding compression to a full-text retrieval system.* Software — Practice & Experience, 25(8) 1995, pp. 891–903.
9. G. Salton, A. Wong, and C. S. Yang: *A vector space model for automatic indexing.* Communications of the ACM, 18(11) 1975, pp. 613–620.
10. A. Sieminski: *Fast decoding of Huffman codes.* Information Processing Letters, 26 1998, pp. 237–241.
11. K. Sparck Jones, S. Walker, and S. E. Robertson: *A probabilistic model of information retrieval: development and comparative experiments – parts 1 and 2.* Information Processing & Management, 36(6) 2000, pp. 779–840.
12. C. Tryfonopoulos, M. Koubarakis, and Y. Drougas: *Filtering algorithms for information retrieval models with named attributes and proximity operators*, in Proc. SIGIR Conf., Sheffield, UK, 2004, pp. 313–320.
13. J. Véronis and P. Langlais: *Evaluation of parallel text alignment systems: The ARCADE project*, in Parallel Text Processing, J. Véronis, ed., Kluwer Academic Publishers, Dordrecht, 2000, pp. 369–388.
14. I. H. Witten, A. Moffat, and T. C. Bell: *Managing Gigabytes: Compressing and Indexing Documents and Images*, Van Nostrand Reinhold, New York, 1994.

An Efficient Algorithm for Approximate Pattern Matching with Swaps

Matteo Campanelli[1], Domenico Cantone[2], Simone Faro[2], and Emanuele Giaquinta[2]

[1] Università di Catania, Scuola Superiore di Catania
Via San Nullo 5/i, I-95123 Catania, Italy
[2] Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
macampanelli@ssc.unict.it, {cantone | faro | giaquinta}@dmi.unict.it

Abstract. The Pattern Matching problem with Swaps consists in finding all occurrences of a pattern P in a text T, when disjoint local swaps in the pattern are allowed. In the Approximate Pattern Matching problem with Swaps one seeks to compute, for every text location with a swapped match of P, the number of swaps necessary to obtain a match at the location.
In this paper, we present new efficient algorithms for the Approximate Swap Matching problem. In particular, we first present a $\mathcal{O}(nm^2)$ algorithm, where m is the length of the pattern and n is the length of the text, which is a variation of the Backward-Cross-Sampling algorithm, a recent solution to the swap matching problem. Subsequently, we propose an efficient implementation of our algorithm, based on the bit-parallelism technique. The latter solution achieves a $\mathcal{O}(mn)$-time and $\mathcal{O}(\sigma)$-space complexity, where σ is the dimension of the alphabet.
From an extensive comparison with some of the most recent and effective algorithms for the approximate swap matching problem, it turns out that our algorithms are very flexible and achieve very good results in practice.

Keywords: approximate pattern matching with swaps, nonstandard pattern matching, combinatorial algorithms on words, design and analysis of algorithms

1 Introduction

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

This problem is of relevance in practical applications such as text and music retrieval, data mining, network security, and many others. Following [6], we also mention a particularly important application of the swap matching problem in biological computing, specifically in the process of translation in molecular biology, with the genetic triplets (otherwise called *codons*). In such application one wants to detect the possible positions of the start and stop codons of an mRNA in a biological sequence and find hints as to where the flanking regions are relative to the translated mRNA region.

Matteo Campanelli, Domenico Cantone, Simone Faro, Emanuele Giaquinta: An Efficient Algorithm for Approximate Pattern Matching with Swaps, pp. 90–104.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

The swap matching problem was introduced in 1995 as one of the open problems in nonstandard string matching [12]. The first nontrivial result was reported by Amir *et al.* [1], who provided a $\mathcal{O}(nm^{\frac{1}{3}} \log m)$-time algorithm in the case of alphabet sets of size 2, showing also that the case of alphabets of size exceeding 2 can be reduced to that of size 2 with a $\mathcal{O}(\log^2 \sigma)$-time overhead (subsequently reduced to $\mathcal{O}(\log \sigma)$ in the journal version [2]). Amir *et al.* [4] studied some rather restrictive cases in which a $\mathcal{O}(m \log^2 m)$-time algorithm can be obtained. More recently, Amir *et al.* [3] solved the swap matching problem in $\mathcal{O}(n \log m \log \sigma)$-time. We observe that the above solutions are all based on the fast Fourier transform (FFT) technique.

In 2008 the first attempt to provide an efficient solution to the swap matching problem without using the FFT technique has been presented by Iliopoulos and Rahman in [11]. They introduced a new graph-theoretic approach to model the problem and devised an efficient algorithm, based on the bit-parallelism technique [7], which runs in $\mathcal{O}((n + m) \log m)$-time, provided that the pattern size is comparable to the word size in the target machine.

More recently, in 2009, Cantone and Faro [9] presented a first approach for solving the swap matching problem with short patterns in linear time. Their algorithm, named Cross-Sampling, though characterized by a $\mathcal{O}(nm)$ worst-case time complexity, admits an efficient bit-parallel implementation, named BP-Cross-Sampling, which achieves $\mathcal{O}(n)$ worst-case time and $\mathcal{O}(\sigma)$ space complexity in the case of short patterns fitting in few machine words.

In a subsequent paper [8] a more efficient algorithm, named Backward-Cross-Sampling and based on a similar structure as the one of the Cross-Sampling algorithm, has been proposed. The Backward-Cross-Sampling scans the text from right to left and has a $\mathcal{O}(nm^2)$-time complexity, whereas its bit-parallel implementation, named BP-Backward-Cross-Sampling, works in $\mathcal{O}(mn)$-time and $\mathcal{O}(\sigma)$-space complexity. However, despite their higher worst-case running times, in practice the algorithms Backward-Cross-Sampling and BP-Backward-Cross-Sampling show a better behavior than their predecessors Cross-Sampling and BP-Cross-Sampling, respectively.

In this paper we are interested in the approximate variant of the swap matching problem. The *Approximate Pattern Matching problem with Swaps* seeks to compute, for each text location j, the number of swaps necessary to convert the pattern to the substring of length m ending at text location j.

A straightforward solution to the approximate swap matching problem consists in searching for all occurrences (with swap) of the input pattern P, using any algorithm for the standard swap matching problem. Once a swap match is found, to get the number of swaps, it is sufficient to count the number of mismatches between the pattern and its swap occurrence in the text and then divide it by 2.

In [5] Amir *et al.* presented an algorithm that counts in time $\mathcal{O}(\log m \log \sigma)$ the number of swaps at every location containing a swapped matching, thus solving the approximate pattern matching problem with swaps in $\mathcal{O}(n \log m \log \sigma)$-time.

In [9] Cantone and Faro presented also an extension of the Cross-Sampling algorithm, named Approximate-Cross-Sampling, for the approximate swap matching problem. However, its bit-parallel implementation has a notably high space overhead, since it requires $(m \log(\lfloor m/2 \rfloor + 1) + m)$ bits, with m the length of the pattern.

In this paper we present a variant of the Backward-Cross-Sampling algorithm for the approximate swap matching problem, which works in $\mathcal{O}(nm^2)$-time and requires $\mathcal{O}(m)$-space. Its bit-parallel implementation, in contrast with the BP-Approximate-Cross-Sampling algorithm, does not add any space overhead and maintains a worst-case $\mathcal{O}(mn)$-time and $\mathcal{O}(\sigma)$-space complexity, when the pattern size is comparable to the word size in the target machine, and is very fast in practice.

The rest of the paper is organized as follows. In Section 2 we recall some preliminary definitions. Then in Section 3 we describe the Approximate-Cross-Sampling algorithm and its bit-parallel variant. In Section 4 we present a variant of the Backward-Cross-Sampling algorithm for the approximate swap matching problem and its straightforward bit-parallel implementation. Then we compare, in Section 5, our newly proposed algorithms against the most effective algorithms present in literature and, finally, we briefly draw our conclusions in Section 6.

2 Notions and Basic Definitions

Given a string P of length $m \geq 0$, we represent it as a finite array $P[0\mathinner{..}m-1]$ and write length$(P) = m$. In particular, for $m = 0$ we obtain the empty string ε. We denote by $P[i]$ the $(i+1)$-st character of P, for $0 \leq i <$ length(P), and by $P[i\mathinner{..}j]$ the substring of P contained between the $(i+1)$-st and the $(j+1)$-st characters of P, for $0 \leq i \leq j <$ length(P). A k-substring of a string S is a substring of S of length k. For any two strings P and P', we say that P' is a suffix of P if $P' = P[i\mathinner{..}\text{length}(P)-1]$, for some $0 \leq i <$ length(P). Similarly, we say that P' is a prefix of P if $P' = P[0\mathinner{..}i-1]$, for some $0 \leq i \leq$ length(P). We denote by P_i the nonempty prefix $P[0\mathinner{..}i]$ of P of length $i+1$, for $0 \leq i < m$, whereas, if $i < 0$, we agree that P_i is the empty string ε. Moreover, we say that P' is a proper prefix (suffix) of P if P' is a prefix (suffix) of P and $|P'| < |P|$. Finally, we write $P.P'$ to denote the concatenation of P and P'.

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

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

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

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

As already observed, if a pattern P of length m has a swap match ending at location j of a text T, then the number k of swaps needed to transform P into its swapped version $\pi(P) = T[j-m+1\mathinner{..}j]$ is equal to half the number of mismatches of P at location j. Thus the value of k lies between 0 and $\lfloor m/2 \rfloor$.

Definition 3. *Given a text T of length n and a pattern P of length m, P is said to* swap-match *(or to have a* swapped occurrence*)* at location j of T with k swaps *if there exists a swap permutation π of P such that $\pi(P)$ matches T at location j and $k = |\{i : P[i] \neq P[\pi(i)]\}|/2$. In such a case we write $P \propto_k T_j$.*

Definition 4 (Pattern Matching Problem with Swaps). *Given a text T of length n and a pattern P of length m, find all locations $j \in \{m-1, \ldots, n-1\}$ such that P swap-matches with T at location j, i.e., $P \propto T_j$.*

Definition 5 (Approximate Pattern Matching Problem with Swaps). *Given a text T of length n and a pattern P of length m, find all pairs (j, k), with $j \in \{m-1, \ldots, n-1\}$ and $0 \leq k \leq \lfloor m/2 \rfloor$, such that P has a swapped occurrence in T at location j with k swaps, i.e., $P \propto_k T_j$.*

The following elementary result will be used later.

Lemma 6 ([9]). *Let P and R be strings of length m over an alphabet Σ and suppose that there exists a swap permutation π such that $\pi(P) = R$. Then π is unique.*

Proof. Suppose, by way of contradiction, that there exist two different swap permutations π and π' such that $\pi(P) = \pi'(P) = R$. Then there must exist an index i such that $\pi(i) \neq \pi'(i)$. Without loss of generality, let us assume that $\pi(i) < \pi'(i)$ and suppose that i be the smallest index such that $\pi(i) \neq \pi'(i)$. Since $\pi(i), \pi'(i) \in \{i-1, i, i+1\}$, by Definition 1(b), it is enough to consider the following three cases:

Case 1: $\pi(i) = i - 1$ and $\pi'(i) = i$.
Then, by Definition 1(a), we have $\pi(i-1) = i$, so that $P[\pi(i-1)] = P[i] = P[\pi'(i)] = P[\pi(i)]$, thus violating Definition 1(c).
Case 2: $\pi(i) = i$ and $\pi'(i) = i + 1$.
Since by Definition 1(a) we have $\pi'(i+1) = i$, then $P[\pi'(i+1)] = P[i] = P[\pi(i)] = P[\pi'(i)]$, thus again violating Definition 1(c).
Case 3: $\pi(i) = i - 1$ and $\pi'(i) = i + 1$.
By Definition 1(c) we have $\pi(i-1) = \pi'(i+1) = i$. Thus $\pi'(i-1) \neq i = \pi(i-1)$, contradicting the minimality of i. □

Corollary 7. *Given a text T of length n and a pattern P of length m, if $P \propto T_j$, for a given position $j \in \{m-1, \ldots, n-1\}$, then there exists a unique swapped occurrence of P in T ending at position j.* □

3 The Approximate-Cross-Sampling Algorithm

The Approximate-Cross-Sampling algorithm [9] computes the swap occurrences of all prefixes of a pattern P (of length m) in continuously increasing prefixes of a text T (of length n), using a dynamic programming approach. Additionally, for each occurrence of P in T, the algorithm computes also the number of swaps necessary to convert the pattern in its swapped occurrence.

In particular, during its $(j+1)$-st iteration, for $j = 0, 1, \ldots, n-1$, it is established whether $P_i \propto_k T_j$, for each $i = 0, 1, \ldots, m-1$, by exploiting information gathered during previous iterations as follows.

Let us put

$$\bar{\mathcal{S}}_j =_{\mathrm{Def}} \{(i,k) \mid 0 \leq i \leq m-1 \text{ and } P_i \propto_k T_j\}$$
$$\bar{\lambda}_j =_{\mathrm{Def}} \begin{cases} \{(0,0)\} & \text{if } P[0] = T[j] \\ \emptyset & \text{otherwise}, \end{cases}$$

for $0 \leq j \leq n-1$, and

$$\bar{\mathcal{S}}'_j =_{\mathrm{Def}} \{(i,k) \mid 0 \leq i < m-1 \text{ and } (P_{i-1} \propto_k T_{j-1} \vee i = 0) \text{ and } P[i] = T[j+1]\},$$

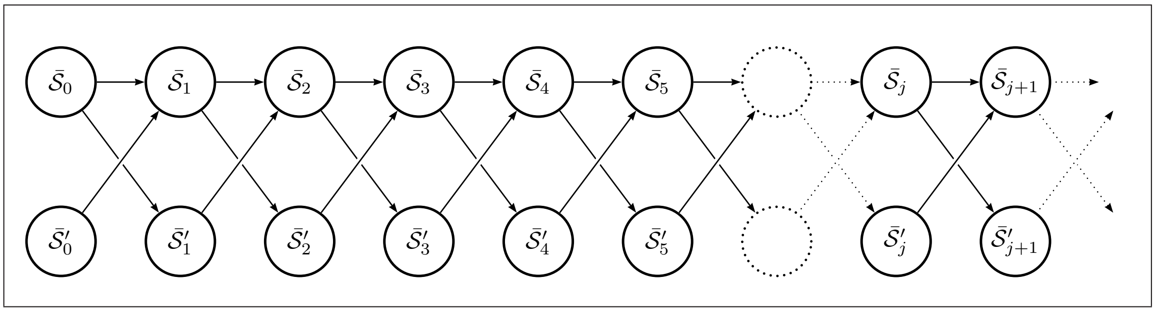


Figure 1. A graphic representation of the iterative fashion for computing sets $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$ for increasing values of j.

for $1 \leq j < n-1$. Then the following recurrences hold:

$$\begin{aligned} \bar{\mathcal{S}}_{j+1} &= \{(i,k) \mid i \leq m-1 \text{ and } ((i-1,k) \in \bar{\mathcal{S}}_j \text{ and } P[i] = T[j+1]) \text{ or} \\ &\qquad\qquad\qquad\qquad\quad ((i-1,k-1) \in \bar{\mathcal{S}}'_j \text{ and } P[i] = T[j])\ \} \cup \bar{\lambda}_{j+1} \\ \bar{\mathcal{S}}'_{j+1} &= \{(i,k) \mid i < m-1 \text{ and } (i-1,k) \in \bar{\mathcal{S}}_j \text{ and } P[i] = T[j+2]\} \cup \bar{\lambda}_{j+2}. \end{aligned} \tag{1}$$

where the base cases are given by $\mathcal{S}_0 = \bar{\lambda}_0$ and $\mathcal{S}'_0 = \bar{\lambda}_1$.

Such relations allow one to compute the sets $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$ in an iterative fashion, where $\bar{\mathcal{S}}_{j+1}$ is computed in terms of both $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$, whereas $\bar{\mathcal{S}}'_{j+1}$ needs only $\bar{\mathcal{S}}_j$ for its computation. The resulting dependency graph has a doubly crossed structure, from which the name of the algorithm in Fig. 2(A), Approximate-Cross-Sampling, for the swap matching problem. Plainly, the time complexity of the Approximate-Cross-Sampling algorithm is $\mathcal{O}(nm)$.

In [9], a bit-parallel implementation of the Approximate-Cross-Sampling algorithm, called BP-Approximate-Cross-Sampling, has been presented.

The BP-Approximate-Cross-Sampling algorithm[1] uses a representation of the sets $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$ as lists of qm bits, $\bar{D}_j$ and $\bar{D}'_j$ respectively, where m is the length of the pattern and $q = \log(\lfloor m/2 \rfloor + 1) + 1$. If $(i,k) \in \bar{\mathcal{S}}_j$, where $0 \leq i < m$ and $0 \leq k \leq \lfloor m/2 \rfloor$, then the rightmost bit of the i-th block of $\bar{D}_j$ is set to 1 and the leftmost $q-1$ bits of the i-th block correspond to the value k (we need exactly q bits to represent a value between 0 and $\lfloor m/2 \rfloor$). The same considerations hold for the sets $\bar{\mathcal{S}}'_j$. Notice that if $mq \leq w$, each list fits completely in a single computer word, whereas if $mq > w$ one needs $\lceil mq/w \rceil$ computer words to represent each of the sets $\bar{\mathcal{S}}_j$ and $\bar{\mathcal{S}}'_j$.

For each character c of the alphabet Σ, the algorithm maintains a bit mask $M[c]$, where the rightmost bit of the i-th block is set to 1 if $P[i] = c$, and a bit mask $B[c]$, whose i-th block have all its bits set to 1 if $P[i] = c$.

The algorithm also maintains two bit vectors, $\bar{D}$ and $\bar{D}'$, whose configurations during the computation are respectively denoted by $\bar{D}_j$ and $\bar{D}'_j$, as the location j advances over the input text. For convenience, we introduce also the bit vectors $\bar{D}_{-1}$ and $\bar{D}'_{-1}$, which are both set to 0^{qm}. While scanning the text from left to right, the algorithm computes for each position $j \geq 0$ the bit vector $\bar{D}_j$ in terms of $\bar{D}_{j-1}$ and $\bar{D}'_{j-1}$, by performing the following bitwise operations (in brackets the corresponding operations on the set $\bar{\mathcal{S}}_j$ represented by $\bar{D}_j$):

[1] Here we provide some minor corrections to the code of the BP-Approximate-Cross-Sampling algorithm presented in [9].

(A) APPROXIMATE-CROSS-SAMPLING (P, m, T, n)

$\bar{S}_0^0 \leftarrow \bar{S}_0^1 \leftarrow \emptyset$
if $P[0] = T[0]$ **then** $\bar{S}_0^0 \leftarrow \{(0,0)\}$
if $P[0] = T[1]$ **then** $\bar{S}_0^1 \leftarrow \{(0,0)\}$
for $j \leftarrow 1$ **to** $n-1$ **do**
  $\bar{S}_j^0 \leftarrow \bar{S}_j^1 \leftarrow \emptyset$
  for $(i,k) \in \bar{S}_{j-1}^0$ **do**
    if $i < m-1$ **then**
      if $P[i+1] = T[j]$
        then $\bar{S}_j^0 \leftarrow \bar{S}_j^0 \cup \{(i+1,k)\}$
      if $j < n-1$ and $P[i+1] = T[j+1]$
        then $\bar{S}_j^1 \leftarrow \bar{S}_j^1 \cup \{(i+1,k)\}$
    else Output$((j-1,k))$
  for $(i,k) \in \bar{S}_{j-1}^1$ **do**
    if $i < m-1$ and $P[i+1] = T[j-1]$
      then $\bar{S}_j^0 \leftarrow \bar{S}_j^0 \cup \{(i+1,k+1)\}$
  if $P[0] = T[j]$ **then** $\bar{S}_j^0 \leftarrow \bar{S}_j^0 \cup \{(0,0)\}$
  if $j < n-1$ and $P[0] = T[j+1]$
    then $\bar{S}_j^1 \leftarrow \bar{S}_j^1 \cup \{(0,0)\}$
for $(i,k) \in \bar{S}_{n-1}^0$ **do**
  if $i = m-1$ **then** Output$(n-1,k)$

(B) BP-APPROXIMATE-CROSS-SAMPLING (P, m, T, n)

$q \leftarrow \log(\lfloor m/2 \rfloor + 1) + 1$
$F \leftarrow 0^{qm-1}1$
$G \leftarrow 0^{q(m-1)}1^q$
for $c \in \Sigma$ **do**
  $M[c] \leftarrow 0^{qm}$
  $B[c] \leftarrow 0^{qm}$
for $i \leftarrow 0$ to $m-1$ **do**
  $M[P[i]] \leftarrow M[P[i]] \mid F$
  $B[P[i]] \leftarrow B[P[i]] \mid G$
  $F \leftarrow (F \ll q)$
  $G \leftarrow (G \ll q)$
$F \leftarrow 0^{q-1}10^{q(m-1)}$
$\bar{D} \leftarrow \bar{D}' \leftarrow 0^{qm}$
for $j \leftarrow 0$ to $n-1$ **do**
  $H^0 \leftarrow (\bar{D} \ll q) \mid 1$
  $H^1 \leftarrow (\bar{D}' \ll q)\ \&\ B[T[j-1]]$
  $H^2 \leftarrow (\bar{D}' \ll q)\ \&\ M[T[j-1]]$
  $\bar{D} \leftarrow (H^0\ \&\ B[T[j]]) \mid H^1$
  $\bar{D} \leftarrow \bar{D}\ +\ (H^2 \ll 1)$
  $\bar{D}' \leftarrow (H^0\ \&\ B[T[j+1]])\ \&\ \sim \bar{D}$
  if $(\bar{D}\ \&\ F) \neq 0^{qm}$ **then**
    $k \leftarrow (\bar{D} \gg (q(m-1)+1))$
    Output(j,k)

Figure 2. (A) The APPROXIMATE-CROSS-SAMPLING algorithm for the approximate swap matching problem. **(B)** Its bit-parallel variant BP-APPROXIMATE-CROSS-SAMPLING.

$$
\begin{array}{ll}
\bar{D}_j \leftarrow \bar{D}_{j-1} \ll q & [\,\bar{\mathcal{S}}_j = \{(i,k) : (i-1,k) \in \bar{\mathcal{S}}_{j-1}\}\,] \\
\bar{D}_j \leftarrow \bar{D}_j \mid 1 & [\,\bar{\mathcal{S}}_j = \bar{\mathcal{S}}_j \cup \{(0,0)\}\,] \\
\bar{D}_j \leftarrow \bar{D}_j\ \&\ B[T[j]] & [\,\bar{\mathcal{S}}_j = \bar{\mathcal{S}}_j \setminus \{(i,k) : P[i] \neq T[j]\}\,] \\
\bar{D}_j \leftarrow \bar{D}_j \mid H^1 & [\,\bar{\mathcal{S}}_j = \bar{\mathcal{S}}_j \cup K\,] \\
\bar{D}_j \leftarrow \bar{D}_j\ +\ (H^2 \ll 1) & [\,\forall\ (i,k) \in K \text{ change } (i,k) \text{ with } (i,k+1) \text{ in } \bar{\mathcal{S}}_j\,]
\end{array}
$$

where $H^1 = ((\bar{D}'_{j-1} \ll q)\ \&\ B[T[j-1]])$, $H^2 = ((\bar{D}'_{j-1} \ll q)\ \&\ M[T[j-1]])$, and $K = \{(i,k) : (i-1,k) \in \bar{\mathcal{S}}'_{j-1} \wedge P[i] = T[j-1]\}$.

Similarly, the bit vector $\bar{D}'_j$ is computed in the j-th iteration of the algorithm in terms of $\bar{D}_{j-1}$, by performing the following bitwise operations (in brackets the corresponding operations on the set $\bar{\mathcal{S}}'_j$ represented by $\bar{D}'_j$):

$$
\begin{array}{ll}
\bar{D}'_j \leftarrow \bar{D}_{j-1} \ll q & [\,\bar{\mathcal{S}}'_j = \{(i,k) : (i-1,k) \in \bar{\mathcal{S}}_{j-1}\}\,] \\
\bar{D}'_j \leftarrow \bar{D}'_j \mid 1 & [\,\bar{\mathcal{S}}'_j = \bar{\mathcal{S}}'_j \cup \{(0,0)\}\,] \\
\bar{D}'_j \leftarrow \bar{D}'_j\ \&\ B[T[j+1]] & [\,\bar{\mathcal{S}}'_j = \bar{\mathcal{S}}'_j \setminus \{(i,k) : P[i] \neq T[j+1]\}\,] \\
\bar{D}'_j \leftarrow \bar{D}'_j\ \&\ \sim \bar{D}_j & [\,\bar{\mathcal{S}}'_j = \bar{\mathcal{S}}'_j \setminus \{(i,k) : (i,k) \in \bar{\mathcal{S}}_j\}\,].
\end{array}
$$

During the j-th iteration, if the rightmost bit of the $(m-1)$-st block of $\bar{D}_j$ is set to 1, i.e. if $(\bar{D}_j\ \&\ 10^{q(m-1)}) \neq 0^m$, a swap match is reported at position j. The total number of swaps is contained in the $q-1$ leftmost bits of the $(m-1)$-st block of $\bar{D}_j$, which can be retrieved by performing a bitwise shift on $\bar{D}_j$ of $(q(m-1)+1)$ positions to the right.

The code of the BP-APPROXIMATE-CROSS-SAMPLING algorithm is shown in Fig. 2(B). It achieves a $\mathcal{O}(\lceil mn \log m/w \rceil)$ worst-case time complexity and requires $\mathcal{O}(\sigma \lceil m \log m/w \rceil)$ extra space, where σ is the size of the alphabet. If $m(\log(\lfloor m/2 \rfloor + 1) + 1) \leq c_1 w$, where c_1 is a small integer constant, then the algorithm requires $\mathcal{O}(n)$-time and $\mathcal{O}(\sigma)$ extra space.

4 New Algorithms for the Approximate Swap Matching Problem

In this section we present a new practical algorithm for solving the swap matching problem, called Approximate-BCS (Approximate Backward Cross Sampling), which is characterized by a $\mathcal{O}(mn^2)$-time and $\mathcal{O}(m)$-space complexity, where m and n are the length of the pattern and text, respectively.

Our algorithm is an extension of the Backward-Cross-Sampling algorithm [8], for the standard swap matching problem. It inherits from the Approximate-Cross-Sampling algorithm the same doubly crossed structure in its iterative computation, but searches for all occurrences of the pattern in the text by scanning characters backwards, from right to left.

Later, in Section 4.2, we present an efficient implementation based on bit parallelism of the Approximate-BCS algorithm, which achieves a $\mathcal{O}(mn)$-time and $\mathcal{O}(\sigma)$-space complexity, when the pattern fits within few computer words, i.e., if $m \leq c_1 w$, for some small constant c_1.

4.1 The Approximate-BCS Algorithm

The Approximate-BCS algorithm searches for all the swap occurrences of a pattern P (of length m) in a text T (of length n) using right-to-left scans in windows of size m, as in the Backward DAWG Matching (BDM) algorithm for the exact single pattern matching problem [10]. In addition, for each occurrence of P in T, the algorithm counts the number of swaps necessary to convert the pattern in its swapped occurrence.

The BDM algorithm processes the pattern by constructing a *directed acyclic word graph* (DAWG) of the reversed pattern. The text is processed in windows of size m which are searched for the longest prefix of the pattern from right to left by means of the DAWG. At the end of each search phase, either a longest prefix or a match is found. If no match is found, the window is shifted to the start position of the longest prefix, otherwise it is shifted to the start position of the second longest prefix.

As in the BDM algorithm, the Approximate-BCS algorithm processes the text in windows of size m. Each attempt is identified by the last position, j, of the current window of the text. The window is searched for the longest prefix of the pattern which has a swapped occurrence ending at position j of the text. At the end of each attempt, a new value of j is computed by performing a safe shift to the right of the current window in such a way to left-align it with the longest prefix matched in the previous attempt.

To this end, if we put

$$\mathcal{S}_j^h =_{\text{Def}} \{h-1 \leq i \leq m-1 \mid P[i-h+1\,..\,i] \propto T_j\},$$

$$\mathcal{W}_j^h =_{\text{Def}} \{h \leq i < m-1 \mid P[i-h+2\,..\,i] \propto T_j \text{ and } P[i-h+1] = T[j-h]\},$$

for $0 \leq j < n$ and $0 \leq h \leq m$, then the following recurrences hold:

$$\begin{aligned}\mathcal{S}_j^{h+1} &= \{h-1 \leq i \leq m-1 \mid (i \in \mathcal{S}_j^h \text{ and } P[i-h] = T[j-h]) \text{ or} \\ &\qquad (i \in \mathcal{W}_j^h \text{ and } P[i-h] = T[j-h+1])\} \\ \mathcal{W}_j^{h+1} &= \{h \leq i \leq m-1 \mid i \in \mathcal{S}_j^h \text{ and } P[i-h] = T[j-h-1]\}.\end{aligned} \tag{2}$$

where the base cases are given by

$$\mathcal{S}_j^0 = \{i \mid 0 \leq i < m\} \quad \text{and} \quad \mathcal{W}_j^0 = \{0 \leq i < m-1 \mid P[i+1] = T[j]\}.$$

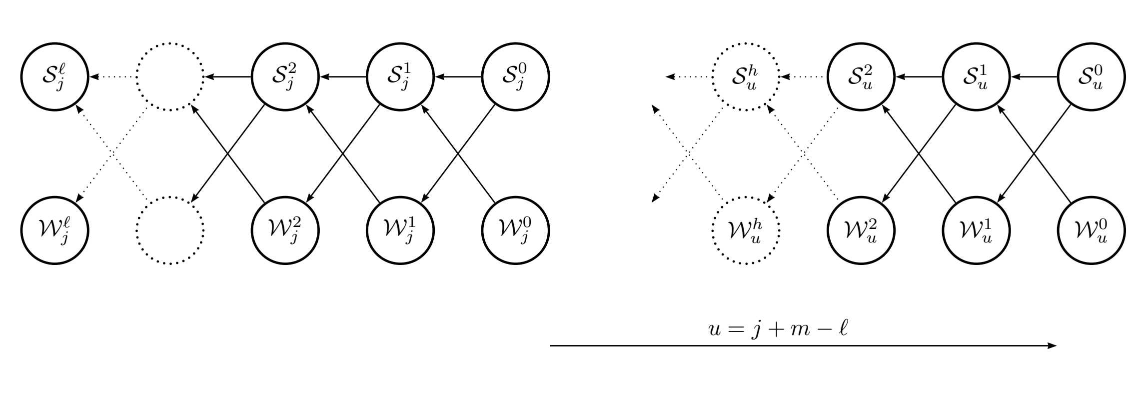


Figure 3. A graphic representation of the iterative fashion for computing the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$ for increasing values of h. A first attempt starts at position j of the text and stops with $h = \ell$. The subsequent attempt starts at position $u = j + m - l$.

Such relations allow one to compute the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$ in an iterative fashion, where $\mathcal{S}_j^{h+1}$ is computed in terms of both $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$, whereas $\mathcal{W}_j^{h+1}$ needs only $\mathcal{S}_j^h$ for its computation. The resulting dependency graph has a doubly crossed structure as shown in Figure 3.

Plainly the set S_j^h includes all the values i such that the h-substring of P ending at position i has a swapped occurrence ending at position j in T. Thus, if $(h-1) \in S_j^h$, then there is a swapped occurrence of the prefix of length h of P. Hence, it follows that P has a swapped occurrence ending at position j if and only if $(m-1) \in S_j^m$.

Observe however that the only prefix of length m is the pattern P itself. Thus $(m-1) \in S_j^m$ if and only if $S_j^m \neq \emptyset$.

The following result follows immediately from (2).

Lemma 8. *Let P and T be a pattern of length m and a text of length n, respectively. Moreover let $m-1 \leq j \leq n-1$ and $0 \leq i < m$. If $i \in S_j^\gamma$, then it follows that $i \in (S_j^h \cup W_j^h)$, for $1 \leq h \leq \gamma$.* □

Lemma 9. *Let P and T be a pattern of length m and a text of length n, respectively. Then, for every $m-1 \leq j \leq n-1$ and $0 \leq i < m$ such that $i \in (S_j^\gamma \cap W_j^{\gamma-1} \cap S_j^{\gamma-1})$, we have $P[i-\gamma+1] = P[i-\gamma+2]$.*

Proof. From $i \in (S_j^\gamma \cap S_j^{\gamma-1})$ it follows that $P[i-\gamma+1] = T[j-\gamma+1]$. Also, from $i \in W_j^{\gamma-1}$ it follows that $P[i-\gamma+2] = T[j-\gamma+1]$. Thus $P[i-\gamma+1] = P[i-\gamma+2]$. □

The following lemma will be used.

Lemma 10. *Let P and T be a pattern of length m and a text of length n, respectively. Moreover let $m-1 \leq j \leq n-1$ and $0 \leq i < m$. Then, if $i \in S_j^\gamma$, there is a swap between characters $P[i-\gamma+1]$ and $P[i-\gamma+2]$ if and only if $i \in (S_j^\gamma \setminus S_j^{\gamma-1})$.*

Proof. Before entering into details we remember that, by Definition 1, a swap can take place between characters $P[i-\gamma+1]$ and $P[i-\gamma+2]$ if and only if $P[i-\gamma+1] = T[j-\gamma+2]$, $P[i-\gamma+2] = T[j-\gamma+1]$ and $P[i-\gamma+1] \neq P[i-\gamma+2]$.

Now, suppose that $i \in S_j^\gamma$ and that there is a swap between characters $P[i-\gamma+1]$ and $P[i-\gamma+2]$. We proceed by contradiction to prove that $i \notin S_j^{\gamma-1}$. Thus, we have

(i) $i \in S_j^{\gamma}$	(by hypothesis)
(ii) $P[i-\gamma+2] = T[j-\gamma+1] \neq P[i-\gamma+1]$	(by hypothesis)
(iii) $i \in S_j^{\gamma-1}$	(by contradiction)
(iv) $i \notin W_j^{\gamma-1}$	(by (ii), (iii), and Lemma 9)
(v) $P[i-\gamma+1] = T[j-\gamma+1]$	(by (i) and (iv))

obtaining a contradiction between (ii) and (v).

Next, suppose that $i \in (S_j^{\gamma} \setminus S_j^{\gamma-1})$. We prove that there is a swap between characters $P[i-\gamma+1]$ and $P[i-\gamma+2]$. We have

(i) $i \in S_j^{\gamma}$ and $i \notin S_j^{\gamma-1}$	(by hypothesis)
(ii) $i \in W_j^{\gamma-1}$	(by (i) and Lemma 8)
(iii) $i \in S_j^{\gamma-2}$	(by (ii) and (2))
(iv) $P[i-\gamma+1] = T[j-\gamma+2]$	(by (i) and (ii))
(v) $P[i-\gamma+2] = T[j-\gamma+1]$	(by (ii))
(vi) $P[i-\gamma+2] \neq T[j-\gamma+2] = P[i-\gamma+1]$	(by (i) and (iii)).

□

The following corollary is an immediate consequence of Lemmas 10 and 8.

Corollary 11. *Let P and T be strings of length m and n, respectively, over a common alphabet Σ. Then, for $m-1 \leq j \leq n-1$, P has a swapped occurrence in T at location j with k swaps, i.e., $P \propto_k T_j$, if and only if*

$$(m-1) \in S_j^m \quad \textit{and} \quad |\Delta_j| = k\,,$$

where $\Delta_j = \{1 \leq h < m \ : \ (m-1) \ \in \ (S_j^{h+1} \setminus S_j^h)\}$. □

In consideration of the preceding corollary, the Approximate-BCS algorithm maintains a counter which is incremented every time $(m-1) \in (S_j^{h+1} \setminus S_j^h)$, for any $1 < h \leq m$, in order to count the swaps for an occurrence ending at a given position j of the text.

For any attempt at position j of the text, let us denote by ℓ the length of the longest prefix matched in the current attempt. Then the algorithm starts its computation with $j = m-1$ and $\ell = 0$. During each attempt, the window of the text is scanned from right to left, for $h = 1, \ldots, m$. If, for a given value of h, the algorithm discovers that $(h-1) \in \mathcal{S}_j^h$, then ℓ is set to the value h.

The algorithm is not able to remember the characters read in previous iterations. Thus, an attempt ends successfully when h reaches the value m (a match is found), or unsuccessfully when both sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$ are empty. In any case, at the end of each attempt, the start position of the window, i.e., position $j-m+1$ in the text, can be shifted to the start position of the longest proper prefix detected during the backward scan. Thus the window is advanced $m-\ell$ positions to the right. Observe that since $\ell < m$, we plainly have that $m-\ell > 0$.

Moreover, in order to avoid accessing the text character at position $j-h+1 = n$, when $j = n-1$ and $h = 0$, the algorithm benefits of the introduction of a sentinel character at the end of the text.

The code of the Approximate-BCS algorithm is shown in Figure 4(A). Its time complexity is $\mathcal{O}(nm^2)$ in the worst case and requires $\mathcal{O}(m)$ extra space to represent the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$.

4.2 The Approximate-BPBCS Algorithm

In [8], an efficient bit-parallel implementation of the Backward-Cross-Sampling algorithm, called BP-Backward-Cross-Sampling, has also been presented. In this section we illustrate a practical bit-parallel implementation of the Approximate-BCS algorithm, named Approximate-BPBCS, along the same lines of the BP-Backward-Cross-Sampling algorithm.

In the Approximate-BPBCS algorithm, the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$ are represented as lists of m bits, D_j^h and C_j^h respectively, where m is the length of the pattern.

The $(i-h+1)$-th bit of D_j^h is set to 1 if $i \in \mathcal{S}_j$, i.e., if $P[i-h+1\,..\,i] \propto T_j$, whereas the $(i-h+1)$-th bit of C_j^h is set to 1 if $i \in \mathcal{W}_j^h$, i.e., if $P[i-h+2\,..\,i] \propto T_j$ and $P[i-h+1] = T[j-h]$. All remaining bits are set to 0. Notice that if $m \leq w$, each bit vector fits in a single computer word, whereas if $m > w$ we need $\lceil m/w \rceil$ computer words to represent each of the sets $\mathcal{S}_j^h$ and $\mathcal{W}_j^h$.

For each character c of the alphabet Σ, the algorithm maintains a bit mask $M[c]$ whose i-th bit is set to 1 if $P[i] = c$.

As in the Approximate-BCS algorithm, the text is processed in windows of size m, identified by their last position j, and the first attempt starts at position $j = m-1$. For any searching attempt at location j of the text, the bit vectors D_j^1 and C_j^1 are initialized to $M[T[j]] \mid (M[T[j+1]]\&(M[T[j]] \ll 1))$ and $M[T[j-1]]$, respectively, according to the recurrences (2) and relative base cases. Then the current window of the text, i.e., $T[j-m+1\,..\,j]$, is scanned from right to left, by reading character $T[j-h+1]$, for increasing values of h. Namely, for each value of $h > 1$, the bit vector D_j^{h+1} is computed in terms of D_j^h and C_j^h, by performing the following bitwise operations:

$$
\begin{array}{ll}
(a) & D_j^{h+1} \leftarrow (D_j^h \ll 1) \;\&\; M[T[j-h]] \\
(b) & D_j^{h+1} \leftarrow D_j^{h+1} \mid ((C_j^h \ll 1) \;\&\; M[T[j-h+1]])\,.
\end{array}
$$

Concerning (a), by a left shift of D_j^h, all elements of $\mathcal{S}_j^h$ are added to the set $\mathcal{S}_j^{h+1}$. Then, by performing a bitwise and with the mask $M[T[j-h]]$, all elements i such that $P[i-h] \neq T[j-h]$ are removed from $\mathcal{S}_j^{h+1}$. Similarly, the bit operations in (b) have the effect to add to $\mathcal{S}_j^{h+1}$ all elements i in $\mathcal{W}_j^h$ such that $P[i-h] = T[j-h+1]$. Formally, we have:

$$
\begin{array}{ll}
(a') & \mathcal{S}_j^{h+1} \leftarrow \mathcal{S}_j^h \setminus \{i \in \mathcal{S}_j^h \;:\; P[i-h] \neq T[j-h]\} \\
(b') & \mathcal{S}_j^{h+1} \leftarrow \mathcal{S}_j^{h+1} \cup \mathcal{W}_j^h \setminus \{i \in \mathcal{W}_j^h \;:\; P[i-h] \neq T[j-h+1]\}\,.
\end{array}
$$

Similarly, the bit vector C_j^{h+1} is computed in terms of D_j^h, by performing the following bitwise operations

$$
(c) \quad C_j^{h+1} \leftarrow (D_j^h \ll 1) \;\&\; M[T[j-h-1]]
$$

which have the effect to add to the set $\mathcal{W}_j^{h+1}$ all elements of the set $\mathcal{S}_j^h$ (by shifting D_j^h to the left by one position) and to remove all elements i such $P[i] \neq T[j-h-1]$ holds (by a bitwise and with the mask $M[T[j-h-1]]$), or, more formally:

$$
(c') \quad \mathcal{W}_j^{h+1} \leftarrow \mathcal{S}_j^h \;\setminus\; \{i \in \mathcal{S}_j^h \;:\; P[i-h] \neq T[j-h-1]\}\,.
$$

In order to count the number of swaps, observe that the $(i-h+1)$-th bit of D_j^h is set to 1 if $i \in S_j^h$. Thus, the condition $(m-1) \in (S_j^{h+1} \setminus S_j^h)$ can be implemented by the following bitwise condition:

$$
(d) \quad ((D^{h+1} \;\&\; \sim (D^h \ll 1)) \;\&\; (1 \ll h)) \neq 0\,.
$$

```
(A) Approximate-BCS (P, m, T, n)
T[n] ← P[0]
j ← m − 1
while j < n do
    h ← 0
    S_j^0 ← {i | 0 ≤ i < m}
    W_j^0 ← {0 ≤ i < m − 1 | P[i + 1] = T[j]}
    c ← 0
    while h < m and S_j^h ∪ W_j^h ≠ ∅ do
        if (h − 1) ∈ S_j^h then ℓ ← h
        for each i ∈ S_j^h do
            if i ≥ h  and  P[i − h] = T[j − h]
                then S_j^{h+1} ← S_j^{h+1} ∪ {i}
            if i > h  and  P[i − h] = T[j − h − 1]
                then W_j^{h+1} ← W_j^{h+1} ∪ {i}
        for each i ∈ W_j^h do
            if i ≥ h and P[i − h] = T[j − h + 1]
                then S_j^{h+1} ← S_j^{h+1} ∪ {i}
        if m − 1 ∈ S_j^{h+1} and m − 1 ∉ S_j^h
            then c ← c + 1
        h ← h + 1
    if (h − 1) ∈ S_j^h then Output(j,c)
    j ← j + m − ℓ
```

```
(B) Approximate-BPBCS (P, m, T, n)
F ← 10^{m−1}
for c ∈ Σ do M[c] ← 0^m
for i ← 0 to m − 1 do
    M[P[i]] ← M[P[i]] | F
    F ← F ≫ 1
T[n] ← P[0]
j ← m − 1
F ← 10^{m−1}
while j < n do
    h ← 1, ℓ ← 0
    D ← M[T[j]]
    D ← D | (M[T[j + 1]]&(M[T[j]] ≪ 1))
    C ← M[T[j − 1]]
    c ← 0
    while h < m and (D | C) ≠ 0 do
        D′ ← D ≪ 1
        if F&D ≠ 0 then ℓ ← h
        H ← (C ≪ 1) & M[T[j − h + 1]]
        C ← D′ & M[T[j − h − 1]]
        D ← D′ & M[T[j − h]]
        D ← D | H
        if (D & ∼ D′) & (1 ≪ h) ≠ 0
            then c ← c + 1
        h ← h + 1
    if D ≠ 0 then Output(j,c)
    j ← j + m − ℓ
```

Figure 4. (A) The Approximate-BCS algorithm for the approximate swap matching problem. **(B)** Its bit-parallel variant Approximate-BPBCS.

As in the Approximate-BCS algorithm, an attempt ends when $h = m$ or $(D_j^h|C_j^h) = 0$. If $h = m$ and $D_j^h \neq 0$, a swap match at position j of the text is reported. In any case, if $h < m$ is the largest value such that $D_j^h \neq 0$, then a prefix of the pattern, of length $\ell = h$, which has a swapped occurrence ending at position j of the text, has been found. Thus, a safe shift of $m - \ell$ position to the right can take place.

In practice, two vectors only are enough to implement the sets D_j^h and C_j^h, for $h = 0, 1, \ldots, m$, as one can transform the vector D_j^h into the vector D_j^{h+1} and the vector C_j^h into the vector C_j^{h+1}, during the h-th iteration of the algorithm at a given location j of the text.

The counter for taking note of the number of swaps requires $\log(\lfloor m/2 \rfloor + 1)$ bits to be implemented. This compares favorably with the BP-Approximate-Cross-Sampling algorithm which uses instead m counters of $\log(\lfloor m/2 \rfloor + 1)$ bits, one for each prefix of the pattern.

The resulting Approximate-BPBCS algorithm is shown in Fig. 4(B). It achieves a $\mathcal{O}(\lceil nm^2/w \rceil)$ worst-case time complexity and requires $\mathcal{O}(\sigma\lceil m/w \rceil + \log(\lfloor m/2 \rfloor + 1))$ extra space, where σ is the alphabet size. If the pattern fits in few machine words, then the algorithm finds all swapped matches and their corresponding counts in $\mathcal{O}(nm)$ time and $\mathcal{O}(\sigma)$ extra space.

5 Experimental Results

Next we report and comment experimental results relative to an extensive comparison under various conditions of the following approximate swap matching algorithms:

- Approximate-Cross-Sampling (ACS)
- BP-Approximate-Cross-Sampling (BPACS)
- Approximate-BCS (ABCS)
- Approximate-BPBCS (BPABCS)
- Iliopoulos-Rahman algorithm with a naive check of the swaps (IR&C)
- BP-Backward-Cross-Sampling algorithm with a naive check of the swaps (BPBCS&C)

We have chosen to include in our comparison also the algorithms IR&C and BP-BCS&C, since the algorithms IR and BPBCS turned out, in [8], to be the most efficient solutions for the swap matching problem. Instead, the Naive algorithm and algorithms based on the FFT technique have not been taken into consideration, as their overhead is quite high, resulting in poor performances.

All algorithms have been implemented in the C programming language and were used to search for the same strings in large fixed text buffers on a PC with AMD Turion 64 X2 processor with mobile technology TL-60 of 2 GHz and a RAM memory of 4 GB. In particular, all algorithms have been tested on six Randσ problems, for $\sigma = 4, 8, 16, 32, 64$ and 128, on a genome, on a protein sequence, and on a natural language text buffer, with patterns of length $m =$ 4, 8, 12, 16, 20, 24, 28, 32.

In the following tables, running times are expressed in hundredths of seconds and the best results have been bold-faced.

Running Times for Random Problems

In the case of random texts, all algorithms have been tested on six Randσ problems. Each Randσ problem consists in searching a set of 100 random patterns for any given length value in a 4 Mb random text over a common alphabet of size σ, with a uniform character distribution.

Running times for a Rand4 problem

m	4	8	12	16	20	24	28	32
ACS	5.916	5.768	5.835	5.860	5.753	5.739	5.571	5.604
ABCS	17.132	10.681	8.504	7.278	6.322	6.096	5.778	5.341
BPACS	0.817	0.794	0.752	0.800	0.784	0.799	0.818	0.747
BPABCS	0.573	0.341	**0.255**	**0.204**	**0.177**	**0.159**	**0.141**	**0.129**
IR&C	**0.275**	**0.275**	0.275	0.276	0.275	0.279	0.276	0.282
BPBCS&C	0.614	0.358	0.262	0.212	0.182	0.161	0.145	0.132

Running times for a Rand8 problem

m	4	8	12	16	20	24	28	32
ACS	4.769	4.756	4.762	4.786	4.761	4.808	4.765	4.796
ABCS	11.675	7.273	5.632	4.736	4.167	3.782	3.511	3.305
BPACS	0.832	0.830	0.828	0.831	0.830	0.829	0.827	0.827
BPABCS	0.413	**0.229**	**0.175**	**0.145**	**0.127**	**0.114**	**0.104**	**0.096**
IR&C	**0.282**	0.279	0.279	0.277	0.280	0.279	0.283	0.285
BPBCS&C	0.388	0.249	0.193	0.157	0.141	0.121	0.111	0.101

Running times for a Rand16 problem

m	4	8	12	16	20	24	28	32
ACS	5.210	5.291	5.162	5.282	5.198	5.201	5.202	5.131
ABCS	10.200	6.314	5.297	4.554	3.932	3.511	3.448	3.140
BPACS	0.786	0.807	0.780	0.783	0.812	0.806	0.743	0.721
BPABCS	0.346	**0.198**	**0.144**	**0.118**	**0.103**	**0.093**	**0.086**	**0.081**
IR&C	**0.275**	0.274	0.279	0.274	0.275	0.277	0.279	0.274
BPBCS&C	0.330	0.211	0.155	0.126	0.110	0.099	0.091	0.085

Running times for a Rand32 problem

m	4	8	12	16	20	24	28	32
ACS	5.285	5.080	5.228	5.262	5.175	5.190	5.216	5.296
ABCS	9.414	5.831	4.437	3.955	3.521	3.232	2.954	2.890
BPACS	0.776	0.746	0.796	0.775	0.834	0.807	0.791	0.796
BPABCS	0.294	**0.184**	**0.138**	**0.113**	**0.097**	**0.086**	**0.078**	**0.073**
IR&C	**0.275**	0.276	0.276	0.276	0.279	0.275	0.275	0.277
BPBCS&C	0.285	0.191	0.146	0.119	0.103	0.091	0.083	0.077

Running times for a Rand64 problem

m	4	8	12	16	20	24	28	32
ACS	5.101	5.108	5.254	5.174	5.155	5.098	5.095	5.262
ABCS	8.857	5.350	4.165	3.502	3.273	2.972	2.717	2.692
BPACS	0.838	0.808	0.769	0.714	0.835	0.806	0.807	0.766
BPABCS	0.267	**0.162**	**0.127**	**0.108**	**0.095**	**0.086**	**0.078**	**0.073**
IR&C	0.272	0.276	0.275	0.280	0.281	0.283	0.279	0.279
BPBCS&C	**0.255**	0.165	0.130	0.111	0.098	0.089	0.082	0.076

Running times for a Rand128 problem

m	4	8	12	16	20	24	28	32
ACS	5.070	5.052	5.091	4.996	5.088	4.940	4.968	5.216
ABCS	8.672	5.288	3.994	3.289	2.941	2.778	2.660	2.523
BPACS	0.833	0.836	0.836	0.836	0.835	0.835	0.836	0.833
BPABCS	0.248	**0.148**	**0.115**	**0.098**	**0.087**	**0.080**	**0.075**	**0.071**
IR&C	0.352	0.354	0.354	0.353	0.353	0.353	0.354	0.333
BPBCS&C	**0.230**	0.151	0.117	0.099	0.090	0.082	0.077	0.072

The experimental results show that the BPABCS algorithm obtains the best run-time performance in most cases. In particular, for very short patterns and small alphabets, our algorithm is second only to the IR&C algorithm. In the case of very short patterns and large alphabets, our algorithm is second only to the BPBCS&C algorithm. In addition we notice that the algorithms IR&C, ACS, and BPACS show a linear behavior, whereas the algorithms ABCS and BPABCS are characterized by a decreasing trend.

Running Times for Real World Problems

The tests on real world problems have been performed on a genome sequence and on a natural language text buffer. The genome we used for the tests is a sequence of 4,638,690 base pairs of *Escherichia coli* taken from the file E.coli of the Large Canterbury Corpus.[1] The tests on the protein sequence have been performed using a 2.4 Mb file containing a protein sequence from the human genome with 22 different characters. The experiments on the natural language text buffer have been done with the file world192.txt (The CIA World Fact Book) of the Large Canterbury Corpus. The file contains 2,473,400 characters drawn from an alphabet of 93 different characters.

[1] http://www.data-compression.info/Corpora/CanterburyCorpus/

Running times for a genome segence ($\sigma = 4$)

m	4	8	12	16	20	24	28	32
ACS	5.629	5.643	5.654	5.636	5.644	5.640	5.647	6.043
ABCS	18.018	11.261	8.805	7.523	6.700	6.117	5.710	5.359
BPACS	0.950	0.914	0.917	0.766	0.874	0.934	0.935	0.843
BPABCS	0.647	**0.318**	**0.266**	**0.232**	**0.195**	**0.174**	**0.160**	0.147
IR&C	**0.262**	0.287	0.314	0.311	0.311	0.311	0.310	0.311
BPBCS&C	0.678	0.367	0.290	0.233	0.204	0.176	**0.160**	**0.146**

Running times for a protein sequence ($\sigma = 22$)

m	4	8	12	16	20	24	28	32
ACS	3.777	3.784	3.671	3.729	3.766	3.703	3.716	3.741
ABCS	7.045	4.557	3.734	3.162	2.806	2.661	2.600	2.351
BPACS	0.565	0.581	0.561	0.563	0.584	0.580	0.534	0.519
BPABCS	0.249	**0.142**	**0.103**	**0.084**	**0.074**	**0.066**	**0.061**	**0.058**
IR&C	0.388	0.390	0.391	0.389	0.391	0.391	0.396	0.389
BPBCS&C	**0.241**	0.145	0.107	0.087	0.075	0.068	0.062	**0.058**

Running times for a natural language text buffer ($\sigma = 93$)

m	4	8	12	16	20	24	28	32
ACS	3.170	2.757	2.748	2.756	2.761	2.745	2.746	2.754
ABCS	6.175	4.054	3.164	2.705	2.306	2.288	2.042	1.866
BPACS	0.492	0.497	0.492	0.491	0.492	0.491	0.494	0.493
BPABCS	0.194	**0.114**	**0.086**	**0.071**	**0.062**	**0.056**	**0.051**	**0.049**
IR&C	0.171	0.165	0.164	0.168	0.165	0.165	0.165	0.167
BPBCS&C	**0.164**	0.126	0.094	0.076	0.070	0.059	0.056	0.055

From the above experimental results, it turns out that the BPABCS algorithm obtains in most cases the best results and, in the case of very short patterns, is second to IR&C (for the genome sequence) and to BPBCS&C (for the protein sequence and the natual language text buffer).

6 Conclusions

In this paper we have presented new efficient algorithms for the Approximate Swap Matching problem. In particular, we have devised an extension of the Backward-Cross-Sampling general algorithm, named Approximate-BCS, and of its bit-parallel implementation BP-Backward-Cross-Sampling, named Approximate-BPBCS.

The Approximate-BCS algorithm achieves a $\mathcal{O}(nm^2)$-time complexity and requires $\mathcal{O}(nm)$ additional space, whereas the Approximate-BPBCS algorithm achieves a $\mathcal{O}(\lceil nm^2/w \rceil)$ worst-case time complexity and, when the pattern fits in few machine words, finds all swapped matches and their corresponding counts in $\mathcal{O}(nm)$-time.

In contrast with the BP-Approximate-Cross-Sampling algorithm, the Approximate-BPBCS algorithm requires $\mathcal{O}(\sigma\lceil m/w \rceil + \log(\lfloor m/2 \rfloor + 1))$ extra space and is thus preferable to the former in the case of longer patterns.

From an extensive experimentation, it turns out that the Approximate-BPBCS algorithm is very fast in practice and obtains the best results in most cases, being second only to algorithms based on a naive check of the number of swaps in the case of very short patterns.

References

1. A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein: *Pattern matching with swaps*, in IEEE Symposium on Foundations of Computer Science, 1997, pp. 144–153.
2. A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein: *Pattern matching with swaps*. Journal of Algorithms, 37(2) 2000, pp. 247–266.
3. A. Amir, R. Cole, R. Hariharan, M. Lewenstein, and E. Porat: *Overlap matching*. Inf. Comput., 181(1) 2003, pp. 57–74.
4. A. Amir, G. M. Landau, M. Lewenstein, and N. Lewenstein: *Efficient special cases of pattern matching with swaps*. Information Processing Letters, 68(3) 1998, pp. 125–132.
5. A. Amir, M. Lewenstein, and E. Porat: *Approximate swapped matching*. Inf. Process. Lett., 83(1) 2002, pp. 33–39.
6. P. Antoniou, C. Iliopoulos, I. Jayasekera, and M. Rahman: *Implementation of a swap matching algorithm using a graph theoretic model*, in Bioinformatics Research and Development, Second International Conference, BIRD 2008, vol. 13 of Communications in Computer and Information Science, Springer, 2008, pp. 446–455.
7. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching*. Commun. ACM, 35(10) 1992, pp. 74–82.
8. M. Campanelli, D. Cantone, and S. Faro: *A new algorithm for efficient pattern matching with swaps*, in IWOCA 2009: 20th International Workshop on Combinatorial Algorithms, Lecture Notes in Computer Science, Springer, 2009.
9. D. Cantone and S. Faro: *Pattern matching with swaps for short patterns in linear time*, in SOFSEM 2009: Theory and Practice of Computer Science, 35th Conference on Current Trends in Theory and Practice of Computer Science, vol. 5404 of Lecture Notes in Computer Science, Springer, 2009, pp. 255–266.
10. M. Crochemore and W. Rytter: *Text algorithms*, Oxford University Press, 1994.
11. C. S. Iliopoulos and M. S. Rahman: *A new model to solve the swap matching problem and efficient algorithms for short patterns*, in SOFSEM 2008, vol. 4910 of Lecture Notes in Computer Science, Springer, 2008, pp. 316–327.
12. S. Muthukrishnan: *New results and open problems related to non-standard stringology*, in Combinatorial Pattern Matching, 6th Annual Symposium, CPM 95, vol. 937 of Lecture Notes in Computer Science, Springer, 1995, pp. 298–317.

Searching for Jumbled Patterns in Strings

Ferdinando Cicalese[1], Gabriele Fici[1], and Zsuzsanna Lipták[2]

[1] Dipartimento di Informatica ed Applicazioni, University of Salerno, Italy
{cicalese,fici}@dia.unisa.it
[2] AG Genominformatik, Technische Fakultät, Bielefeld University, Germany
zsuzsa@cebitec.uni-bielefeld.de

Abstract. The Parikh vector of a string s over a finite ordered alphabet $\Sigma = \{a_1, \ldots, a_\sigma\}$ is defined as the vector of multiplicities of the characters, i.e. $p(s) = (p_1, \ldots, p_\sigma)$, where $p_i = |\{j \mid s_j = a_i\}|$. Parikh vector q occurs in s if s has a substring t with $p(t) = q$. The problem of searching for a query q in a text s of length n can be solved simply and optimally with a sliding window approach in $O(n)$ time. We present two new algorithms for the case where the text is fixed and many queries arrive over time. The first algorithm finds all occurrences of a given Parikh vector in a text (over a fixed alphabet of size $\sigma \geq 2$) and appears to have a sub-linear expected time complexity. The second algorithm only decides whether a given Parikh vector appears in a binary text; it iteratively constructs a linear size data structure which then allows answering queries in constant time, for many queries even during the construction phase.

Keywords: Parikh vectors, permuted strings, pattern matching, string algorithms, average case analysis

1 Introduction

Parikh vectors of strings count the multiplicity of the characters. They have been reintroduced many times by many different names (compomer [5], composition [3], Parikh vector [14], permuted string [7], permuted pattern [9], and others). They are very natural objects to study, if for nothing else because of the many different applications they appear in; for instance, in computational biology, they have been applied for alignment [3], SNP discovery [5], repeated pattern discovery [9], and, most naturally, in interpretation of mass spectrometry data [4]. Parikh vectors can be seen as a generalization of strings, where we view two strings as equivalent if one can be turned into the other by permuting its characters; in other words, if the two strings have the same Parikh vector.

The problem we are interested in here is answering the question whether a query Parikh vector q appears in a given text s (decision version), or where it occurs (occurrence version). An occurrence of q is defined as an occurrence of a substring t of s with Parikh vector q. The problem can be viewed as an approximate pattern matching problem: We are looking for an occurrence of a jumbled version of a query string t, i.e. for the occurrence of a substring t' which has the same Parikh vector. In the following, let n be the length of the text s, m the length of the query q (defined as the length of a string t with Parikh vector q), and σ the size of the alphabet.

The above problem (both decision and occurrence versions) can be solved with a simple sliding window based algorithm, in $O(n)$ time and $O(\sigma)$ additional storage space. This is worst case optimal with respect to the case of one query. However, when we expect to search for many queries in the same string, the above approach leads to $O(Kn)$ runtime for K queries. To the best of our knowledge, no faster approach is known. This is in stark contrast to the classical exact pattern matching problem:

Ferdinando Cicalese, Gabriele Fici, Zsuzsanna Lipták: Searching for Jumbled Patterns in Strings, pp. 105–117.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

There, for one query, any naive approach leads to $O(nm)$ runtime, while quite involved ideas for preprocessing and searching are necessary to achieve an improved runtime of $O(n+m)$, as do the Knuth-Morris-Pratt [12], Boyer-Moore [6] and Boyer-Moore-type algorithms (see, e.g., [2,10]). However, when many queries are expected, the text can be preprocessed to produce a data structure of size linear in n, such as a suffix tree, suffix array, or suffix automaton, which then allows to answer individual queries in time linear in the length of the pattern.

In this paper, we present two new algorithms which perform significantly better than the naive window algorithm, in the case where many queries arrive. In the course of both algorithms, a data structure of size $O(n)$ is constructed, which is subsequently used for fast searching.

1. For general alphabets: We present the Jumping algorithm (Sect. 3) which uses $O(n)$ space to answer occurrence queries in time $O\left(\sigma J \log_2\left(\frac{n}{J}+m\right)\right)$, where J denotes the number of iterations of the main loop of the algorithm. We argue that the expected value of J for the case of random strings and patterns is $O(n/\sqrt{\sigma m})$, yielding an expected runtime of $O\left(\frac{\sqrt{\sigma}\log_2 m}{\sqrt{m}}n\right)$. Our simulations on random strings and real biological strings indicate that this is indeed the performance of the algorithm in practice. This is a significant improvement over the naive algorithm w.r.t. expected runtime, both for a single query and repeated queries over one string.
2. For binary alphabets: After a data structure of size $O(n)$ has been constructed, we answer decision queries in $O(1)$ time (Interval Algorithm, Sect. 4).

The Jumping algorithm is reminiscent of the Boyer-Moore-like approaches to the classical string matching problem [6,2,10]. This analogy is used both in its presentation and in the analysis of the number of iterations performed by the algorithm. We approximate the behavior of the algorithm with a probabilistic automaton, as it is done in [15] to estimate the expected running time of Boyer-Moore on random strings.

A straightforward implementation of the Interval Algorithm requires $\Theta(n^2)$ time for the preprocessing. Instead we present it employing lazy computation of the data structure, and thus the runtime is improved such that a query can be answered either in $O(1)$ or $\Theta(n)$ time, depending on whether the respective entries in the data structure have already been computed. For $K=\omega(n)$ queries, we require $\Theta(K+n^2)$ time (with either implementation), thus always outperforming the naive algorithm, which has $\Theta(Kn)$ runtime. We conjecture that there is no algorithm that can answer any $\Omega(n)$ queries in $o(n^2)$ time.

Related work: An efficient algorithm for computing all Parikh fingerprints of substrings of a given string was developed in [1]. Parikh fingerprints are Boolean vectors where the k'th entry is 1 if and only if a_k appears in the string. The algorithm involves storing a data point for each Parikh fingerprint, of which there are at most $O(n\sigma)$ many. This approach was adapted in [9] for Parikh vectors and applied to identifying all repeated Parikh vectors within a given length range; using it to search for queries of arbitrary length would imply using $\Omega(P(s))$ space, where $P(s)$ denotes the number of different Parikh vectors of substrings of s. This is not desirable, since there are strings with quadratic $P(s)$ [8].

The authors of [7] present an algorithm for finding all occurrences of a Parikh vector in a runlength encoded text. The algorithm's time complexity is $O(n' + \sigma)$, where n' is the length of the runlength encoding of s. Obviously, if the string is not runlength encoded, a preprocessing phase of time $O(n)$ has to be added. However, this may still be feasible if many queries are expected. To the best of our knowledge, this is the only algorithm that has been presented for the problem we are treating here.

2 Notation and Problem Statement

Let $\Sigma = \{a_1, \ldots, a_\sigma\}$ be a finite ordered alphabet. For a string $s \in \Sigma^*$, $s = s_1 \cdots s_n$, we define the Parikh vector $p(s) = (p_1, \ldots, p_\sigma)$ by $p_i := |\{j \mid s_j = a_i\}|$, for $i = 1, \ldots, \sigma$. A Parikh vector p *occurs* in string s if there are positions $i \leq j$ such that $p(s_i \cdots s_j) = p$. We refer to the pair (i, j) as an occurrence of p in s. By convention, we say that the empty string ϵ occurs in each string once. For a Parikh vector $p \in \mathbb{N}^\sigma$, where $\mathbb{N}$ denotes the set of non-negative integers, let $|p| := \sum_i p_i$ denote the *length* of p, namely the length of any string t with $p(t) = p$. Further, by $s[i, j] = s_i \cdots s_j$ we denote the substring of s from i to j, for $1 \leq i \leq j \leq n$.

For two Parikh vectors $p, q \in \mathbb{N}^\sigma$, we define $p \leq q$ and $p + q$ component-wise: $p \leq q$ if and only if $p_i \leq q_i$ for all $i = 1, \ldots, \sigma$, and $p + q = u$ where $u_i = p_i + q_i$ for $i = 1, \ldots, \sigma$. Similarly, for $p \leq q$, we set $q - p = v$ where $v_i = q_i - p_i$ for $i = 1, \ldots, \sigma$.

We want to solve the following problem:

Problem Statement: Let $s \in \Sigma^*$ be given. For a Parikh vector $q \in \mathbb{N}^\sigma$,

1. Decide whether q occurs in s (decision problem);
2. Find all occurrences of q in s (occurrence problem).

In the following, let $|s| = n$ and $|q| = m$. Assume that K many queries arrive over time.

For $K = 1$, both the decision version and the occurrence version can be solved optimally with the following simple algorithm: Move a sliding window of size $|q|$ along string s. This way, we encounter all substrings, and thus all Parikh vectors, of length $|q|$. We maintain the Parikh vector c of the current substring and a counter r which equals the number of indices i such that $c_i \neq q_i$. Each sliding step now costs either 0 or 2 update operations of c, depending on whether the new character entering the window is the same or different from the one that falls out. Whenever we change the value of an entry c_i, we check whether $c_i = q_i$ and increment or decrement r accordingly.

This algorithm solves both the decision and occurrence problems and has running time $\Theta(n)$, using additional storage space $\Theta(\sigma)$. In other words, for one query, it is optimal (save maybe for the additional storage of $\Theta(\sigma)$).

Obviously, one can precompute all sub-Parikh vectors of s, store them (sorted, e.g. lexicographically) and do binary search when a query arrives. Preprocessing time is $\Theta(n^2 \log n)$, because the number of Parikh vectors of s is at most $\binom{n}{2} = O(n^2)$, and there are nontrivial strings with quadratic number of Parikh vectors over arbitrary alphabets [8]. (Now and in the following, we denote the binary logarithm by log, the natural logarithm by ln, and otherwise explicitly state the base.) Moreover, simulations reported there have shown that protein strings have quadratically many sub-Parikh vectors, a result relevant for mass spectrometry applications. Query time is $O(\log n)$ for the decision problem and $O(\log n + M)$ for the occurrence problem for a query with M occurrences. However, the storage space of $\Theta(n^2)$ is inacceptable in many applications.

For small queries, the problem can be solved exhaustively with a linear size indexing structure such as a suffix tree (size $O(n)$). We can search up to length $m = |q|$ (of the substrings); whenever we find a match, we traverse the subtree below and report the leaf numbers, yielding the occurrences of that substring. Total running time is $O(\sigma^m)$ for searching the tree down to level m, and $O(M)$ total time for the enumeration of the leaves in the individual subtrees, where M is the number of occurrences of q in s. If m is small, namely $m = o(\log_\sigma n)$, then the query time is $o(n) + O(M)$. The suffix tree can be constructed in a preprocessing step in time $O(n)$, so altogether we get time $O(n)$, since $M = O(n)$ for any query q.

3 The Jumping Algorithm

In this section, we introduce our algorithm for general alphabets. Let $s = s_1 \cdots s_n \in \Sigma^*$ be given, and let $pr(i)$ denote the Parikh vector of the prefix of s of length i, for $i = 0, \dots, n$, where $pr(0) = p(\epsilon) = (0, \dots, 0)$. We make the following observations:

Observation 1. *Consider Parikh vector $p \in \mathbb{N}^\sigma$, $p \neq (0, \dots, 0)$.*

1. *For any $1 \leq i \leq j \leq n$, $p = pr(j) - pr(i-1)$ if and only if p occurs in s at position (i, j).*
2. *If an occurrence of p ends in position j, then $pr(j) \geq p$.*

The algorithm moves two pointers L and R along the text, pointing at these potential positions $i - 1$ and j. Instead of moving linearly, however, the pointers are updated in jumps, alternating between updates of R and L, in such a manner that many positions are skipped. Moreover, because of the way we update the pointers, after any update it suffices to check whether $R - L = |q|$ to confirm that an occurrence has been found.

We use the following rules for updating the two pointers, illustrated in Fig. 1:

1. the *first fit rule* for updating R, and
2. the *good suffix rule* for updating L.

First fit rule: Assume that the left pointer is pointing at position L, i.e. no unreported occurrence starts before $L + 1$. Notice that, if there is an occurrence of q ending at any position $j > L$, it must hold that $pr(L) + q \leq pr(j)$. In other words, we must fit both $pr(L)$ and q at position j. We define a function FIRSTFIT as the first potential position where an occurence of a Parikh vector p can end:

$$\textsc{firstfit}(p) := \min\{j \mid pr(j) \geq p\}, \tag{1}$$

and set $\textsc{firstfit}(p) = \infty$ if no such j exists. We will update R to the first position where $pr(L)$ and q can fit:

$$R \leftarrow \textsc{firstfit}(pr(L) + q). \tag{2}$$

Good suffix rule: Now assume that R has just been updated. Thus, $p(s[L{+}1, R]) = pr(R) - pr(L) \geq q$. If equality holds, then we have found an occurrence of q in position $(L+1, R)$, and L can be incremented by 1. Otherwise $pr(R) - pr(L) > q$, which implies that, interspersed between the characters that belong to q, there are some "superfluous" characters. Now the first position where an occurrence of q can start is at the beginning of a *contiguous* sequence of characters ending in R which all belong to q. In other words, we need the beginning of the longest suffix of $s[L+1, R]$ with Parikh vector $\leq q$, i.e. the smallest position i such that $pr(R) - pr(i) \leq q$. We find this position by setting

$$L \leftarrow \textsc{firstfit}(pr(R) - q). \tag{3}$$

Note that this rule can also be interpreted as a *bad character rule:* $pr(R) - q = pr(L) + (pr(R) - pr(L)) - q$ contains all those superfluous characters between $L+1$ and R that we have to fit before a possible next occurrence of q. Below we give the pseudo-code of the algorithm.

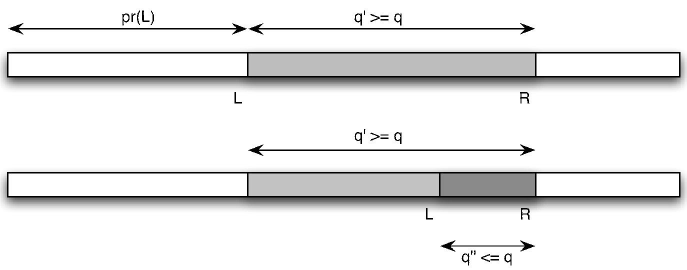


Figure 1. The situation after the update of R (above) and after the update of L (below). R is placed at the first fit of $pr(L) + q$, thus q' is a super-Parikh vector of q. Then L is placed at the beginning of the longest good suffix ending in R, so q'' is a sub-Parikh vector of q.

Algorithm *Jumping Algorithm*
Input: query Parikh vector q
Output: A set Occ containing all beginning positions of occurrences of q in s

set $m \leftarrow |q|; Occ \leftarrow \emptyset;\ L \leftarrow 0$;
while $L < n - m$
  do $R \leftarrow \textsc{firstfit}(pr(L) + q)$;
    if $R - L = m$
      then add $L + 1$ to Occ;
        $L \leftarrow L + 1$;
      else $L \leftarrow \textsc{firstfit}(pr(R) - q)$;
        if $R - L = m$
          then add $L + 1$ to Occ;
            $L \leftarrow L + 1$;
return Occ;

It remains to see how to compute the FIRSTFIT and pr functions.

3.1 How to compute FIRSTFIT and *pr*

In order to compute $\textsc{firstfit}(p)$ for some Parikh vector p, we need to know the prefix vectors of s. However, storing all prefix vectors of s would require $O(\sigma n)$ storage space, which may be too much. Instead, the algorithm uses an "inverted prefix vector table" I containing the increment positions of the prefix vectors: for each character $a_k \in \Sigma$, and each value j up to $p(s)_k$, the position in s of the j'th occurrence of character a_k. In other words, $I[k][j] = \min\{i \mid pr(i)_k \geq j\}$ for $j \geq 1$, and $I[k][0] = 0$. Thus we have

$$\textsc{firstfit}(p) = \max_{k=1,\ldots,\sigma} \{I[k][p_k]\}. \tag{4}$$

Moreover, we can compute the prefix vectors $pr(i)$ from table I: For $k = 1, \ldots, \sigma$,

$$pr(j)_k = \begin{cases} 0 & \text{if } j < I[k][1] \\ \max\{i \mid I[k][i] \leq j\} & \text{otherwise.} \end{cases} \tag{5}$$

The obvious way to find these values is to do binary search for j in each row of I. However, this would take time $\Theta(\sigma \log n)$; a better way is to use information already acquired during the run of the algorithm. As we shall see later (Lemma 3), it always holds that $L \leq R$. Thus, for computing $pr(R)_k$, it suffices to search for R between $pr(L)_k$ and $pr(L)_k + (R - L)$. This search takes time proportional to $\log(R - L)$. Moreover, after each update of L, we have $L \geq R - m$, so when computing $pr(L)_k$, we can restrict the search for L to between $pr(R)_k - m$ and $pr(R)_k$, in time $O(\log m)$. For more details, see Section 3.4.

Example 2. Let $\Sigma = \{a, b, c\}$ and $s = cabcccaaabccbaacca$. The prefix vectors of s are given below. Note that the algorithm does not actually compute these.

pos.		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
s		c	a	b	c	c	c	a	a	a	b	c	c	b	a	a	c	c	a
# a's	0	0	1	1	1	1	1	2	3	4	4	4	4	4	5	6	6	6	7
# b's	0	0	0	1	1	1	1	1	1	1	2	2	2	3	3	3	3	3	3
# c's	0	1	1	1	2	3	4	4	4	4	4	5	6	6	6	6	7	8	8

The inverted prefix table I:

	0	1	2	3	4	5	6	7	8
a	0	2	7	8	9	14	15	18	
b	0	3	10	13					
c	0	1	4	5	6	11	12	16	17

Query $q = (3, 1, 2)$ has 4 occurrences, beginning in positions $5, 6, 7, 13$, since $(3, 1, 2) = pr(10) - pr(4) = pr(11) - pr(5) = pr(12) - pr(6) = pr(18) - pr(12)$. The values of L and R are given below:

k, see Sec. 3.3	1	2	3	4	5	6	7
L	0	4	5	6	7	10	12
R	8	10	11	12	14	18	18
occ. found?	–	yes	yes	yes	–	–	yes

3.2 Preprocessing

Table I can be computed in one pass over s (where we take the liberty of identifying character $a_k \in \Sigma$ with its index k). The variables c_k count the number of occurrences of character a_k seen so far, and are initialized to 0.

Algorithm *Preprocess s*
1. **for** $i = 1$ **to** n
2. $\quad c_{s_i} = c_{s_i} + 1;$
3. $\quad I[s_i][c_{s_i}] = i;$

Table I requires $O(n)$ storage space (with constant 1). Moreover, the string s can be discarded, so we have zero additional storage.

3.3 Correctness

We have to show that (1) if the algorithm reports an occurrence, then it is correct, and (2) if there is an occurrence, then the algorithm will find it. We first need the following lemma:

Lemma 3. *The following algorithm invariants hold:*

1. *After each update of R, we have $pr(R) - pr(L) \geq q$.*
2. *After each update of L, we have $pr(R) - pr(L) \leq q$.*
3. *$L \leq R$.*

Proof. *1.* follows directly from the definition of FIRSTFIT and the update rule for R. For *2.*, if an occurrence was found at (i, j), then before the update we have $L = i - 1$ and $R = j$. Now L is incremented by 1, so $L = i$ and $pr(R) - pr(L) = q - e_{s_i} < q$, where e_k is the k'th unity vector. Otherwise, $L \leftarrow \text{FIRSTFIT}(pr(R) - q)$, and again the claim follows directly from the definition of FIRSTFIT. For *3.*, if an occurrence was found, then L is incremented by 1, and $R - L = m - 1 \geq 0$. Otherwise, $L = \text{FIRSTFIT}(pr(R) - q) = \min\{\ell \mid pr(\ell) \geq pr(R) - q\} \leq R$. □

Proof of (1): If the algorithm reports an index i, then $(i, i+m-1)$ is an occurrence of q: An index i is added to Occ whenever $R - L = m$. If the last update was that of R, then we have $pr(R) - pr(L) \geq q$ by Lemma 3, and together with $R - L = m = |q|$, this implies $pr(R) - pr(L) = q$, thus $(L + 1, R) = (i, i + m - 1)$ is an occurrence of q. If the last update was L, then $pr(R) - pr(L) \leq q$, and it follows analogously that $pr(R) - pr(L) = q$.

Proof of (2): All occurrences of q are reported: Let's assume otherwise. Then there is a minimal i and $j = i + m - 1$ such that $p(s[i, j]) = q$ but i is not reported by the algorithm. By Observation 1, we have $pr(j) - pr(i - 1) = q$.

Let's refer to the values of L and R as two sequences $(L_k)_{k=1,2,...}$ and $(R_k)_{k=1,2,...}$. So we have $L_1 = 0$, and for all $k \geq 1$, $R_k = \text{FIRSTFIT}(pr(L_k) + q)$, and $L_{k+1} = L_k + 1$ if $R_k - L_k = m$ and $L_{k+1} = \text{FIRSTFIT}(pr(R_k) - q)$ otherwise. In particular, $L_{k+1} > L_k$ for all k.

First observe that if for some k, $L_k = i - 1$, then R will be updated to j in the next step, and we are done. This is because $R_k = \text{FIRSTFIT}(pr(L_k) + q) = \text{FIRSTFIT}(pr(i - 1) + q) = \text{FIRSTFIT}(pr(j)) = j$. Similarly, if for some k, $R_k = j$, then we have $L_{k+1} = i - 1$.

So there must be a k such that $L_k < i-1 < L_{k+1}$. Now look at R_k. Since there is an occurrence of q after L_k ending in j, this implies that $R_k = \text{FIRSTFIT}(pr(L_k)+q) \leq j$. However, we cannot have $R_k = j$, so it follows that $R_k < j$. On the other hand, $i-1 < L_{k+1} \leq R_k$ by our assumption and by Lemma 3. So R_k is pointing to a position somewhere between $i-1$ and j, i.e. to a position within our occurrence of q. Denote the remaining part of q to the right of R_k by q': $q' = pr(j) - pr(R_k)$. Since $R_k = \text{FIRSTFIT}(pr(L_k) + q)$, all characters of q must fit between L_k and R_k, so the Parikh vector $p = pr(i) - pr(L_k)$ is a super-Parikh vector of q'. If $p = q'$, then there is an occurrence of q at $(L_k + 1, R_k)$, and by minimality of (i, j), this occurrence was correctly identified by the algorithm. Thus, $L_{k+1} = L_k + 1 \leq i - 1$, contradicting our choice of k. It follows that $p > q'$ and we have to find the longest good suffix of the substring ending in R_k for the next update L_{k+1} of L. But $s[i, R_k]$ is a good suffix because its Parikh vector is a sub-Parikh vector of q, so $L_{k+1} = \text{FIRSTFIT}(pr(R_k) - q) \leq i - 1$, again in contradiction to $L_{k+1} > i - 1$.

We illustrate the proof in Fig. 2.

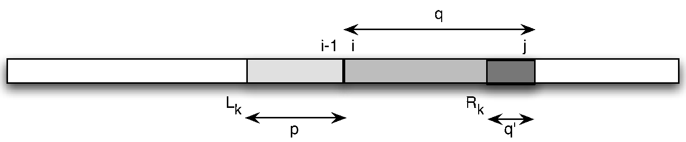


Figure 2. Illustration for proof of correctness.

3.4 Algorithm Analysis

Let $\mathbb{A}(s, q)$ denote the running time of Jumping Algorithm over a text s and a Parikh vector q. Let $J = J(s, q)$ be the number of iterations performed in the `while` loop in line 2, i.e., the number of jumps performed by the algorithm, for the input (s, q). Further, for each $i = 1, \ldots, J$, let $\hat{L}_i$, $\hat{R}_i$ denote the value of L and R, respectively, after the i'th execution of line 3 of the algorithm[1].

In order to calculate the running time of the algorithm on the given input we need to evaluate the number of iterations it performs, the running time of the functions FIRSTFIT and the time needed to compute the Parikh vectors $pr(\cdot)$ necessary in lines 3 and 7.

It is easy to see that computing FIRSTFIT takes $O(\sigma)$ time.

The computation of $pr(\hat{L}_i)$ in line 3 takes $O(\sigma \log m)$: For each $k = 1, \ldots, \sigma$, the component $pr(\hat{L}_i)_k$ can be determined by binary search over the list $I[k][pr(\hat{R}_{i-1})_k - m], I[k][pr(\hat{R}_{i-1})_k - m + 1], \ldots, I[k][pr(\hat{R}_{i-1})_k]$. By $\hat{L}_i \geq \hat{R}_{i-1} - m$, the claim follows.

The computation of $pr(\hat{R}_i)$ in line 7 takes $O(\sigma \log(\hat{R}_i - \hat{R}_{i-1} + m))$. Simply observe that in the prefix ending at position $\hat{R}_i$ there can be at most $\hat{R}_i - \hat{L}_i$ more occurrences of the k'th character than there are in the prefix ending at position $\hat{L}_i$. Therefore, as before, we can determine $pr(\hat{R}_i)_k$ by binary search over the list $I[k][pr(\hat{L}_i)_k], I[k][pr(\hat{L}_i)_k + 1], \ldots, I[k][pr(\hat{L}_i)_k + \hat{R}_i - \hat{L}_i]$. Using the fact that $\hat{L}_i \geq \hat{R}_{i-1} - m$, the desired bound follows.

[1] The $\hat{L}_i$ and $\hat{R}_i$ coincide with the L_k and R_k from Section 3.3 almost but not completely: When an occurrence is found after the update of L, then the corresponding pair L_k, R_k is skipped here. The reason is that now we are only considering those updates that carry a computational cost.

The last three observations imply

$$\mathbb{A}(s,q) = O\left(\sigma J \log m + \sigma \sum_{i=1}^{J} \log(\hat{R}_i - \hat{R}_{i-1} + m)\right).$$

Note that this is an overestimate, since line 7 is only executed if no occurrence was found after the current update of R (line 4). Standard algebraic manipulations using Jensen's inequality (see, e.g. [11]) yield $\sum_{i=1}^{J} \log(\hat{R}_i - \hat{R}_{i-1} + m) \leq J \log\left(\frac{n}{J} + m\right)$. Therefore we obtain

$$\mathbb{A}(s,q) = O\left(\sigma J \log\left(\frac{n}{J} + m\right)\right). \tag{6}$$

The worst case running time of the Jumping Algorithm is superlinear, since there exist strings s and Parikh vectors q such that $J = \Theta(n)$: For instance, on the string $s = ababab \cdots ab$ and $q = (2,0)$, the algorithm will execute $n/2$ jumps.

This sharply contrasts with the experimental evaluation we present later. The Jumping Algorithm appears to have in practice a sublinear behavior. In the rest of this section we sketch an average case analysis of the running time of the Jumping Algorithm leading to the conclusion that its expected running time is sublinear.

We assume that the string s is given as a sequence of i.i.d. random variables uniformly distributed over the alphabet Σ. According to Knuth *et al.* [12] "It might be argued that the average case taken over random strings is of little interest, since a user rarely searches for a random string. However, this model is a reasonable approximation when we consider those pieces of text that do not contain the pattern [...]". The experimental results we provide will show that this is indeed the case.

To simplify the presentation, let us fix the Parikh vector q as being perfectly balanced, i.e., $q = (\frac{m}{\sigma}, \ldots, \frac{m}{\sigma})$. Let E_i denote the expected number ℓ such that $\textsc{firstfit}(pr(i) + q) = i + \ell$. Because of the assumption on the string, we have that E_i is independent of i, so we can write $E_i = E_{m,\sigma}$. In particular, we have

$$E_{m,\sigma} \approx m + \begin{cases} m2^{-m}\binom{m}{m/2} & \text{if } \sigma = 2, \\ \sqrt{2m\sigma \ln \frac{\sigma}{\sqrt{2\pi}}} & \text{otherwise.} \end{cases} \tag{7}$$

This result can be found in [13] where the author studied a variant of the well known coupon collector problem in which the collector has to accumulate a certain number of copies of each coupon. It should not be hard to see that by identifying the characters with the coupon types, the random string with the sequence of coupons obtained, and the query Parikh vector with the number of copies we require for each coupon type, the expected time when the collection is finished is the same as our $E_{m,\sigma}$.

We shall now follow the approach taken by Schaback in the average case analysis of the Boyer-Moore algorithm [15]. We build a probabilistic automaton which simulates the behavior of the Jumping Algorithm. We also assume that each new reference to a position in the string is done by generating the character again. See [15] for how this assumption does not affect the result.

The automaton $\mathcal{A}(n,m,\sigma)$ moves the pointers L and R along the string as follows: with probability $\zeta = \zeta(m,\sigma)$ the pointer L is moved forward by one position (this corresponds to the case of a match); with probability $(1-\zeta)$ the pointer R is moved forward to the closest position to L such that $pr(R) - pr(L) \geq q$; in this case also L is updated and set to $R - m$ (this corresponds to the case of no match; in fact, we are upper bounding the Jumping Algorithm's behavior, since it always updates L to a position at most m away from R).

Let $\mathbb{E}[\mathcal{A}(n,m,\sigma)]$ denote the expected number of jumps of $\mathcal{A}(n,m,\sigma)$. We have $\mathbb{E}[\mathcal{A}(n,m,\sigma)] = \frac{n}{\zeta+(1-\zeta)(E_{m,\sigma}-m)}$. If we take ζ to be the probability that a random string of size m over an alphabet of size σ has Parikh vector q, we get $\zeta \approx \sqrt{\frac{\sigma^\sigma}{(2\pi m)^{\sigma-1}}}$, where we use Stirling approximation for the multinomial $\binom{m}{\frac{m}{\sigma},\ldots,\frac{m}{\sigma}}$. Note that due to the magnitude of ζ, for large values of m, we have

$$\mathbb{E}[\mathcal{A}(n,m,\sigma)] \approx n/(E_{m,\sigma} - m). \tag{8}$$

Recalling (6) and using (7) and (8) as an approximation of the number $\mathbb{E}[J]$ of jumps performed by the Jumping Algorithm, over a random instance, we get that the average case complexity of the Jumping Algorithm can be estimated as $O\left(n\sqrt{\frac{2\pi}{m}}\log m\right)$ in the case of a binary alphabet, and $O\left(\frac{\sigma n}{\sqrt{2\sigma m \ln\left(\sigma/\sqrt{2\pi}\right)}}\log m\right)$, for $\sigma \geq 3$. Summarizing, according to the above approximations, we would expect the algorithm's running time to be $O\left(\frac{n\log m}{\sqrt{m}}\right)$, with the constant in the order of $\sqrt{\frac{\sigma}{2\ln\sigma}}$.

We conclude this section by remarking once more that the above estimate obtained by the approximating probabilistic automaton appears to be confirmed by the experiments.

3.5 Simulations

We implemented the Jumping Algorithm in C++ in order to study the number of jumps J. We ran it on random strings of different lengths and over different alphabet sizes. The underlying probability model is an i.i.d. model with uniform distribution. We sampled random query vectors with length between $\log n$ $(= \log_2 n)$ and $\sqrt{n}$, where n is the length of the string. Our queries were of one of two types:

1. Quasi-balanced Parikh vectors: Of the form $(q_1,\ldots,q_\sigma)$ with $q_i \in (x-\epsilon, x+\epsilon)$, and x running from $\log n/\sigma$ to $\sqrt{n}/\sigma$. For simplicity, we fixed $\epsilon = 10$ in all our experiments, and sampled uniformly at random from all quasi-balanced vectors around each x.
2. Random Parikh vectors with fixed length m. These were sampled uniformly at random from the space of all Parikh vectors with length m.

The rationale for using quasi-balanced queries is that those are clearly worst-case for the number of jumps J, since J depends on the shift length, which in turn depends on $\textsc{firstfit}(pr(L)+q)$. Since we are searching in a random string with uniform character distribution, we can expect to have minimal $\textsc{firstfit}(pr(L)+q)$ if q is close to balanced, i.e. if all entries q_i are roughly the same. This is confirmed by our experimental results which show that J decreases dramatically if the queries are not balanced (Fig. 4, right).

We ran experiments on random strings over different alphabet sizes, and observe that our average case analysis agrees well with the simulation results for random strings and random quasi-balanced query vectors. Plots for $n = 10^5$ and $n = 10^6$ with alphabet sizes $\sigma = 2, 4, 16$ resp. $\sigma = 4, 16$ are shown in Fig. 3.

To see how our algorithm behaves on non-random strings, we downloaded human DNA sequences from GenBank (`http://www.ncbi.nlm.nih.gov/Genbank/`) and ran the Jumping Algorithm with random quasi-balanced queries on them. We found that the algorithm performs 2 to 10 times fewer jumps on these DNA strings than on random strings of the same length, with the gain increasing as n increases. We show the results on a DNA sequence of 1 million bp (from Chromosome 11) in comparison with the average over 10 random strings of the same length (Fig. 4, left).

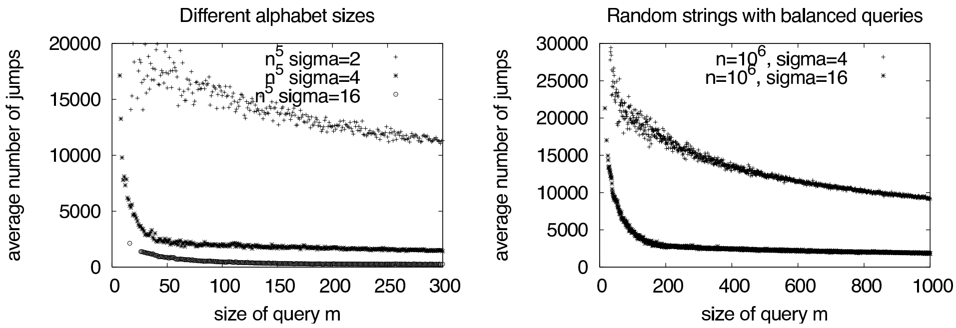


Figure 3. Number of jumps for different alphabet sizes for random strings of size 100 000 (left) and 1 000 000 (right). All queries are randomly generated quasi-balanced Parikh vectors (cf. text). Data averaged over 10 strings and all random queries of same length.

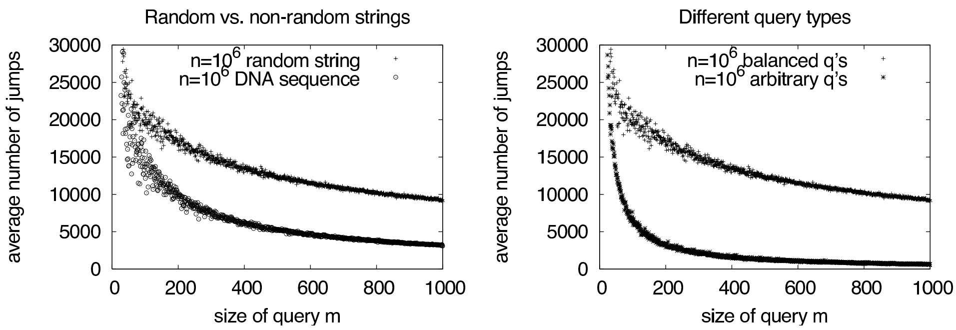


Figure 4. Number of jumps in random vs. nonrandom strings: Random strings over an alphabet of size 4 vs. a DNA sequence, all of length 1 000 0000, random quasi-balanced query vectors. Data averaged over 10 random strings and all queries with the same length (left). Comparison of quasi-balanced vs. arbitrary query vectors over random strings, alphabet size 4, length 1 000 000, 10 strings. The data shown are averaged over all queries with same length m (right).

4 Often Constant Query Time for Binary Alphabets

We now describe our algorithm for binary alphabets. It uses a data structure of size $O(n)$ which it constructs in a lazy manner, only computing those entries that are needed for the current query, and storing them for future queries. Once the data structure has been completely constructed, all queries can be answered in constant time. During the construction phase, answering queries may take either $O(1)$ or $O(n)$ time. Only decision queries are answered.

The algorithm makes use of the following property of binary strings:

Lemma 4. *Let* $s \in \{a,b\}^*$ *with* $|s| = n$. *Fix* $1 \leq m \leq n$. *If the Parikh vectors* $(x_1, m - x_1)$ *and* $(x_2, m - x_2)$ *both occur in* s, *then so does* $(y, m - y)$ *for any* $x_1 \leq y \leq x_2$.

Proof. Consider a sliding window of fixed size m moving along the string and let (p_1, p_2) be the Parikh vector of the current substring. When the window is shifted by one, the Parikh vector either remains unchanged (if the character falling out is the same as the character coming in), or it becomes $(p_1 + 1, p_2 - 1)$ resp. $(p_1 - 1, p_2 + 1)$ (if they are different). Thus the Parikh vectors of substrings of s of length m build a set of the form $\{(x, m - x) \mid x = \min(m), \min(m) + 1, \ldots, \max(m)\}$ for appropriate $\min(m)$ and $\max(m)$. In other words, they build an interval. □

So all we need in order to decide whether a query $q = (q_1, q_2)$ with $|q| = m$ has an occurrence in s is to check whether $\min(m) \leq q_1 \leq \max(m)$. We would like to have a table with $\min(m)$ and $\max(m)$ for all $1 \leq m \leq n$; however, computing the complete table takes $O(n^2)$ time. Notice though that, for any individual query q, we only need the values for $|q|$. So when a query q arrives with $|q| = m$, we look up whether $\min(m)$ and $\max(m)$ have already been computed. If so, we answer the query in constant time. Otherwise, we compute the entries for m by moving a sliding window of size m over s and collecting the minimum and maximum number of a's.

Analysis: All queries take time either $O(1)$ or $O(n)$, and after n queries of the latter kind, the table is completely constructed and all subsequent queries can be answered in $O(1)$ time. If we assume that the query lengths are uniformly distributed, then we can view this as another coupon collector problem (see Section 3.4), where the coupon collector has to collect one copy of each n coupons, namely the different lengths m. Then the expected number of queries needed before having seen all m and thus before having completed the table is $nH_n \approx n \ln n$. The algorithm will have taken $O(n^2)$ time to answer these $n \ln n$ queries, because it spends linear time only on queries with new length m, and $O(1)$ on queries with length that it has seen before; now it can answer all further queries in constant time.

The assumption of the uniform length distribution may not be very realistic; however, even if it does not hold, we never take more time than $O(n^2 + K)$ for K many queries. Since any one query may take at most $O(n)$ time, our algorithm never performs worse than the naive algorithm. Moreover, for those queries where the table entries have to be computed, we can even run the naive algorithm itself and report all occurrences, as well. For all others, we only give decision answers, but in constant time.

Finally, the table can of course be computed completely in a preprocessing step in $O(n^2)$ time, thus always guaranteeing constant query time. The overall running time is $\Theta(K + n^2)$. As long as the number of queries is $K = \omega(n)$, this variant, too, outperforms the naive algorithm, whose running time is $\Theta(Kn)$.

5 Conclusion and Open Problems

Our simulations appear to confirm that in practice the performance of the Jumping Algorithm is well predicted by the expected $O(\frac{\sqrt{\sigma}\log m}{\sqrt{m}} n)$ time of the probabilistic analysis we proposed. A more precise analysis is needed, however. Our approach seems unlikely to lead to any refined average case analysis since that would imply improved results for the intricate variant of the coupon collector problem of [13].

Moreover, in order to better simulate DNA or other biological data, more realistic random string models than uniform i.i.d. should also be analysed, such as first or higher order Markov chains.

Another open problem is whether the Interval Algorithm can be improved by constructing in subquadratic time the data structure it uses (in a preprocessing step). In fact, we conjecture that this is not possible, and that no algorithm can answer arbitrary $\Omega(n)$ many queries in $o(n^2)$ time. However, proving such an upper bound has so far proven elusive.

Acknowledgements: Thanks to Travis Gagie for the pointer to some recent literature on the coupon collector problem, and for pointing out the similarities of the Jumping Algorithm with Boyer-Moore. We thank Rosa Caiazzo for generously giving us of her time. Part of this work was done while F. C. and Zs. L. were funded by a Sofja Kovalevskaja grant of the Alexander von Humboldt Foundation and the German Federal Ministry of Education and Research.

References

1. A. Amir, A. Apostolico, G. M. Landau, and G. Satta: *Efficient text fingerprinting via Parikh mapping*. J. Discrete Algorithms, 1(5-6) 2003, pp. 409–421.
2. A. Apostolico and R. Giancarlo: *The Boyer-Moore-Galil string searching strategies revisited*. SIAM J. Comput., 15(1) 1986, pp. 98–105.
3. G. Benson: *Composition alignment*, in Proc. of the 3rd International Workshop on Algorithms in Bioinformatics (WABI'03), 2003, pp. 447–461.
4. S. Böcker: *Sequencing from compomers: Using mass spectrometry for DNA de novo sequencing of 200+ nt*. Journal of Computational Biology, 11(6) 2004, pp. 1110–1134.
5. S. Böcker: *Simulating multiplexed SNP discovery rates using base-specific cleavage and mass spectrometry*. Bioinformatics, 23(2) 2007, pp. 5–12.
6. R. S. Boyer and J. S. Moore: *A fast string searching algorithm*. Commun. ACM, 20(10) 1977, pp. 762–772.
7. A. Butman, R. Eres, and G. M. Landau: *Scaled and permuted string matching*. Inf. Process. Lett., 92(6) 2004, pp. 293–297.
8. M. Cieliebak, T. Erlebach, Zs. Lipták, J. Stoye, and E. Welzl: *Algorithmic complexity of protein identification: combinatorics of weighted strings*. Discrete Applied Mathematics, 137(1) 2004, pp. 27–46.
9. R. Eres, G. M. Landau, and L. Parida: *Permutation pattern discovery in biosequences*. Journal of Computational Biology, 11(6) 2004, pp. 1050–1060.
10. R. N. Horspool: *Practical fast searching in strings*. Softw., Pract. Exper., 10(6) 1980, pp. 501–506.
11. S. Jukna: *Extremal Combinatorics*, Springer, 1998.
12. D. E. Knuth, J. H. Morris Jr., and V. R. Pratt: *Fast pattern matching in strings*. SIAM J. Comput., 6(2) 1977, pp. 323–350.
13. R. May: *Coupon collecting with quotas*. Electr. J. Comb., 15 2008.
14. A. Salomaa: *Counting (scattered) subwords*. Bulletin of the European Association for Theoretical Computer Science (EATCS), 81 2003, pp. 165–179.
15. R. Schaback: *On the expected sublinearity of the Boyer-Moore algorithm*. SIAM J. Comput., 17(4) 1988, pp. 648–658.

Filter Based Fast Matching of Long Patterns by Using SIMD Instructions

M. Oğuzhan Külekci

TÜBİTAK-UEKAE
National Research Institute of Electronics & Cryptology
41470 Gebze, Kocaeli, Turkey
kulekci@uekae.tubitak.gov.tr

Abstract. SIMD instructions exist in many recent microprocessors supporting parallel execution of some operations on multiple data simultaneously via a set of special instructions working on limited number of special registers. Although the usage of SIMD is explored deeply in multimedia processing, implementation of encryption/decryption algorithms, and on some scientific calculations, it has not been much addressed in pattern matching. This study introduces a filter based exact pattern matching algorithm for searching long strings benefiting from SIMD instructions of Intel's SSE (streaming SIMD extensions) technology. The proposed algorithm has worst, best, and average time complexities of $O(n \cdot m)$, $O(n/m)$, and $O(n/m + n \cdot m/2^{16})$ respectively, while searching an m bytes pattern on a text of n bytes. Experiments on small, medium, and large alphabet text files are conducted to compare the performance of the new algorithm with other alternatives, which are known to be very fast on long string search operations. In all cases the proposed algorithm is the clear winner on the average. When compared with the nearest successor, the matching speed is improved in orders of magnitude on small alphabet sequences. The performance is 40 % better on medium alphabets, and 50 % on natural language text.

Keywords: pattern matching, filtering, SIMD, SSE

1 Introduction

Searching for exact or approximate matches of given pattern(s) on a text file is one of the fundamental problems in computer science. Numerous algorithms focusing on some aspects of the general problem have been developed during the last three decades, some of which can be found in [4,5]. Although the main problem is well studied, recent advances in genomics research, new developments in processor architectures, and the accelerated growth of information on the Internet introduces new challenges in the area.

This study focuses on exact matching of long patterns on random sequences via a filtering methodology. Instead of checking the occurrence of the pattern(s) on all over the text, filtering methods first detects the portions of the text, on which the observation of the pattern is probable with a fast heuristic, and then performs a full verification on those positions reported by the filtering phase. Thus, a filter based string matching algorithm is actually composed of two parts, as filtering and the verification. The first part aims to detect possible match positions on the text without a deep investigation, and the verification process is checking the real existence of the pattern on those detected positions.

Some of the previous filter based pattern matching algorithms may be listed as follows. The algorithm of Wu&Manber [18] combines bit-parallelism with a fast 2-gram hashing heuristic filter. Later on, their algorithm is implemented as the *agrep* [17]

M. Oğuzhan Külekci: Filter Based Fast Matching of Long Patterns by Using SIMD Instructions, pp. 118–128.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

approximate match utility program, which is known to be very powerful especially on approximate and multiple pattern matching. The average optimal (AOSO) and fast average optimal (FAOSO) variants of the original shift-or [2] algorithm defined by Fredriksson&Grabowski [7] may also be viewed from a filtering perspective since they include a verification procedure.

The bit-parallel algorithms [14,15,9,7] that suffer from the computer word size limitation[1] in general can also be used in a filtering framework for searching patterns longer than the computers word size. In such cases, the part of the pattern, which is selected to be less than the word length, is searched on the text by the bit-parallel algorithms, and rest of the pattern is verified on match positions. Moreover, character overloading for searching long patterns or multiple patterns with bit-parallel techniques has been proposed previously [6,15,2] also.

More recently, Lecroq [13] has offered one of the most effective representative of filtering algorithms. The simplicity and average speed of the Lecroq's *new* algorithm makes it a strong candidate in all practical cases including search on small alphabets.

The power of a filtering algorithm may be measured by two metrics: i) the distinguishing power of the proposed filtering method, ii) the computation speed of the filtering function. If the filter is not very selective, then the average number of calls to the verification procedure grows, which in turn degrades the performance. On the other side, if the distinguishing power is good, but the computation of the filter is expensive, then the speed again falls as it will consume more time to calculate the filter value, although the recall of verification is small. This study aims to benefit from the intrinsic SIMD instructions of the modern processors for fast calculation of a distinguishing filter.

SIMD instructions let simultaneous execution of some operands on multiple data by the help of a limited number of special registers. Figure 1 sketches the operation on 128 bit SSE registers, x, y, z. In the example, each register is divided into 4 integers of 32 bit each, and the given operation $\ominus$ is performed and stored between the corresponding data. Note that instead of using 4 integer portions, several other type definitions exist on SSE intrinsics, such as viewing the 128 bit as 16 bytes, or 4 floats also.

The original idea of SIMD was to speed up multimedia procedures, such as audio/video/image processing issues. It is also used in cryptographic applications and on some scientific computations. A good review of SIMD may be found in [8]. Despite the fact that it has not been explored deeply in pattern matching, this study shows that it serves as a good basis especially for filtering techniques.

$$\begin{array}{|c|c|c|c|c|}
\hline x_0 & \ominus & y_0 & \Rightarrow & z_0 \\
\hline x_1 & \ominus & y_1 & \Rightarrow & z_1 \\
\hline x_2 & \ominus & y_2 & \Rightarrow & z_2 \\
\hline x_3 & \ominus & y_3 & \Rightarrow & z_3 \\
\hline
\end{array}$$

Figure 1. The sketch of a sample SIMD instruction.

The algorithm introduced in this study, which will be referred as *SSEF*, uses Intel streaming SIMD extensions (SSE [11]) technology. SSEF finds exact occurrences of patterns longer than 32 bytes on random sequences. Experimental results indicated

[1] Külekci [12] has proposed a bit-parallel algorithm which is not restricted with the computer word size limitation.

that on the average it is approximately 6 times faster than Lecroq's new algorithm, and 15% better than the backward oracle and suffix oracle methods, which are mainly the best choices for long patterns until now.

2 Preliminaries and Basics

Let string S of k characters be shown as $S = s_0 s_1 s_2 \cdots s_{k-1}$. Assuming each character is represented by a single byte, $S[i \ldots j]$ shows the byte array $[s_i s_{i+1} s_{i+2} \cdots s_j]$, where $0 \leq i \leq j < k$. The individual bits of byte s_i are denoted by $s_i = b_0^i b_1^i b_2^i b_3^i b_4^i b_5^i b_6^i b_7^i$, where b_0^i is referred as $sign(s_i)$. In chunks of 16 bytes, same string is represented by $S = C^0 C^1 C^2 \cdots C^{\lfloor (k-1)/16 \rfloor}$, where $C^i = s_{i \cdot 16} s_{i \cdot 16+1} s_{i \cdot 16+2} \cdots s_{i \cdot 16+15}$, for $0 \leq i \leq \lfloor (k-1)/16 \rfloor$. The last block $C^{\lfloor (k-1)/16 \rfloor}$ is not complete if $k \neq 0 \ mod \ 16$. In that case, the remaining bytes of the block are set to zero as $s_j = 0$ for $k - 1 < j$.

Given text T and pattern P of lengths n and m bytes, the number of 16-byte blocks in T and P are denoted by $N = \lceil n/16 \rceil$ and $M = \lceil m/16 \rceil$ respectively. The individual bytes of text T are accessed by t_i, $0 \leq i < n$, and similarly the 16-byte blocks are addressed by D^i, $0 \leq i < N$. The byte and block symbols for pattern P are p_i, $0 \leq i < m$, and Q^i, $0 \leq i < M$ respectively. Figure 2 demonstrates the defined structure.

D^0	D^1		D^{N-1}
$t_0 t_1 \ldots t_{15}$	$t_{16} t_{17} \ldots t_{31}$		$t_{(N-1) \cdot 16} t_{(N-1) \cdot 16+1} \cdots t_{n-1}$

a) The representation of text T.

Q^0	Q^1	....	Q^{M-1}
$p_0 p_1 \ldots p_{15}$	$p_{16} p_{17} \ldots p_{31}$	....	$p_{(M-1) \cdot 16} p_{(M-1) \cdot 16+1} \cdots p_{m-1}$

a) The representation of pattern P.

Figure 2.

The proposed filtering algorithm is designed to be effective on long patterns, where the lower limit for m is 32 ($32 \leq m$). Although it is possible to adapt the algorithm for lesser lengths, the performance gets worse under 32. The number L is defined as $L = \lfloor m/16 \rfloor - 1$, which is the zero-based address of the last 16-byte block of Q whose individual bytes are totally composed of pattern bytes without any padding. For example, if $m = 42$, the 16-byte blocks of the pattern will be $Q = Q^0 Q^1 Q^2$, where the last 6 bytes of Q^2 are padded with zero. The L value for $m = 42$ is $L = 1$, which indicates the last whole block of the pattern is Q^1. Actually, if length of the pattern is a multiple of 16, there is no remainder in the last 16-byte block, and thus, $L = M - 1$. In the other case, L should point to the block preceding the last one as the last one is not a complete block, making $L = M - 2$.

The basic idea of the proposed algorithm is to compute a filter on block $D^{z \cdot L + L}$, where $0 \leq z < \lfloor N/L \rfloor$, to explore if it is appropriate to observe pattern P beginning from any byte inside the prior blocks $D^{z \cdot L}$ to $D^{z \cdot L + (L-1)}$. If the filter value indicates some of the alignments are possible, then those fitting ones are compared with the text byte by byte.

Figure 3 demonstrates this basic idea by assuming $i = z \cdot L$. Note that as $m \geq 32$, and $L = \lfloor m/16 \rfloor - 1$, the pattern fills the bytes in D^{i+L} always.

Block No	D^i	D^{i+1}		....	D^{i+L-1}		D^{i+L}		
Bytes of T	$t_{i \cdot 16}$	$t_{(i+1)\cdot 16}$	...	....	$t_{(i+L-1)\cdot 16}$	...	$t_{(i+L)\cdot 16}$	...	$t_{(i+1)\cdot 16+15}$
P aligned to $t_{i \cdot 16}$	p_0	p_{16}	...	....	$p_{(L-1)\cdot 16}$	...	$p_{L \cdot 16}$	...	$p_{L\cdot 16+15}$
P aligned to $t_{i \cdot 16+1}$	p_0	p_{15}	...	....	$p_{(L-1)\cdot 16-1}$	...	$p_{L\cdot 16-1}$	...	$p_{L\cdot 16+14}$
	....								
P aligned to $t_{i \cdot 16+15}$	.... p_0	p_1	...	....	$p_{(L-1)\cdot 16-15}$	...	$p_{L\cdot 16-15}$	...	$p_{L\cdot 16}$
	....								
P aligned to $t_{(i+L)\cdot 16-1}$						p_0	p_1	...	p_{16}

Figure 3. Appropriate alignments of pattern P according to the filter value computed from D^{i+L},for any $i = z \cdot L$

3 The SSEF Exact Pattern Matching Algorithm

3.1 Preprocessing

The preprocessing stage of the algorithm consist of compiling the possible filter values of the pattern according to the alignments shown in figure 3. Formally, the filter values for $P[(L \cdot 16) \ldots (L \cdot 16 + 15)]$, $P[(L \cdot 16 - 1) \ldots (L \cdot 16 + 14)]$, $\ldots$, $P[1 \ldots 16]$ are computed and stored in a linked list, which will be referred as $FList$ from now on. The pseudo-code of the preprocessing procedure is depicted in Algorithm 1.

Algorithm 1 PreProcess($P = p_0p_1p_2 \cdots p_{m-1}$,$K$)

1: **for** $i = 0$ to 65535 **do**
2: $\quad FList[i] = \emptyset$;
3: **end for**
4: $L = \lfloor m/16 \rfloor - 1$
5: **for** $i = 0$ to $L \cdot 16 - 1$ **do**
6: $\quad r = L \cdot 16 - i$;
7: $\quad f = sign(p_i << K) \cdot 2^{15} + sign(p_{i+1} << K) \cdot 2^{14} + \cdots + sign(p_{i+15} << K)$
8: $\quad FList[f] = FList[f] \cup i$;
9: **end for**
10: **return** L;

The corresponding filter of a 16 bytes sequence is the 16 bits formed by concatenating the sign bits of each byte after shifting by K bits as shown in line 7 of Algorithm 1. The reason for shifting is to generate a distinguishing filter. For example, when the search is to be performed on an English text, the sign bits of bytes are generally 0 as in the standard ascii table the printable characters of the language reside in first 128, where the sign bits are always 0. If we do not include a shift operation, then the filter f value will be 0 in all cases, and while passing over the text verification will be called at each byte. On the other hand, if the text we are searching on is composed of uniformly distributed random 256 bytes, then there is obviously no need for shifting.

Hence, the K value is to be decided depending on the alphabet size and character distribution of the text. K should be set to a value that the most informative bit of the byte must become the sign bit after shift operation. Thus, detection of the most informative bit among the 8 bits of a byte is required for best filtering. This is actually the position on which the distribution of the bits among the whole text is close to their expected values. Note that this requires an additional pass over the whole text, which is not good in practice. A more practical approach may be to consider just the alphabet, and assume the distribution of characters is uniform on the given text. In that case, we are left with just the $|\Sigma|$ bytes, and it is more convenient to decide on the bit position. As an example, let's consider pattern matching on an ascii coded plain DNA sequence, where the alphabet is '`a`','`t`','`c`','`g`' having ascii codes 01100001, 01110100, 01100011, and 01100111 respectively. The first three bits and the fifth bit are all same. Since the number of 1s and 0s are equal on the sixth and seventh positions from the remaining bits, one of them, say 6^{th}, may be used as the distinguishing bit. Thus, while searching on a DNA sequence, setting $K = 5$ to move this bit to the sign bit position would be a good choice when only the alphabet is considered.

3.2 Main algorithm

The pseudo code given in Algorithm 2 depicts the skeleton of the $SSEF$. After the preprocessing stage, the main loop investigates 16-byte blocks of text T in steps of L. If the filter f computed on D^i, where $i = z \cdot L + L$, and $0 \leq z < \lfloor N/L \rfloor$, is not empty, then the appropriate positions listed in $FList[f]$ are verified accordingly.

Algorithm 2 SSEF($P = p_0p_1p_2 \cdots p_{m-1}$, $T = t_0t_1t_2 \cdots t_{m-1}$)

1: Set $K = a$, $0 \leq a < 8$, according to the alphabet;
2: $i = L$ =PreProcess(P,K);
3: **while** $i < N$ **do**
4: $\quad f = sign(t_{i \cdot 16} << K) \cdot 2^{15} + sign(t_{i \cdot 16+1} << K) \cdot 2^{14} + \cdots + sign(t_{i \cdot 16+15} << K)$
5: $\quad$ **for all** $j \in FList[f]$ **do**
6: $\quad\quad$ **if** $P = [t_{(i-L)\cdot 16+j} \ldots t_{(i-L)\cdot 16+j+m-1}]$ **then**
7: $\quad\quad\quad$ pattern detected at $t_{(i-L)\cdot 16+j}$;
8: $\quad\quad$ **end if**
9: $\quad$ **end for**
10: $\quad i = i + L$;
11: **end while**

$Flist[f]$ contains a linked list of integers marking the beginning of the pattern. While investigating the filter on D^i, if $FList[f]$ contains number j, where $0 \leq j < 16 \cdot L$, the pattern potentially begins at $t_{(i-L)\cdot 16+j}$. In that case, a complete verification is to be performed between P and $[t_{(i-L)\cdot 16+j} \ldots t_{(i-L)\cdot 16+j+m-1}]$.

Calculating the corresponding filter of D^i via SSE intrinsics The computation of the filter f of D^i in line 4 of pseudo code given in Alg. 2 is performed by 2 SSE2 intrinsic functions as

$tmp128 = \_mm\_slli\_epi64(D^i, K);$
$f = \_mm\_movemask\_epi8(tmp128);$

First instruction shifts the corresponding 16 bytes of the text D^i by K bits and stores the result in a temporary 128 bit register aiming not to destruct D^i itself.

Second, the instruction `mm_movemask_epi8` returns a 16 bit mask composed of the sign bits of the individual 16 bytes forming the 128 bit value. Figure 4 demonstrates this function.

$$D^j$$
$$\left| b_0^{16\cdot j}\ b_1^{16\cdot j} \dots b_7^{16\cdot j} \middle| b_0^{16\cdot j+1}\ b_1^{16\cdot j+1} \dots b_7^{16\cdot j+1} \middle| \ \dots\dots \ \middle| b_0^{16\cdot j+15}\ b_1^{16\cdot j+15} \dots b_7^{16\cdot j+15} \right|$$
$$\Downarrow$$
$$\texttt{mm\_movemask\_epi8}$$
$$\Downarrow$$
$$\left| b_0^{16\cdot j} b_0^{16\cdot j+1} b_0^{16\cdot j+2} \dots b_0^{16\cdot j+15} \right|$$

Figure 4. The `mm_movemask_epi8` SSE instruction as the filter.

4 Complexity Analysis

The preprocessing stage of the $SSEF$ algorithm requires an additional space to store the 65536 items of $FList$ linked list. On a 32 bit machine, assuming each node consist of an integer and a next pointer, this makes up a total of 512 KB (= 65536 × 8 byte) memory requirement.

The first loop in Algorithm 1 just initializes the $FList$ list, and the second for loop is run $L \cdot 16$ times during the preprocessing. Thus, time complexity of preprocessing is $O(L \cdot 16)$ that approximates to $O(m)$.

$SSEF$ algorithm investigates the N 16-byte block text T in steps of L blocks. Total number of filtering operations is exactly $\lfloor N/L \rfloor$. At each attempt, maximum number of verification requests is $L \cdot 16$, since the filter gives information about that number of appropriate alignments of the pattern. This situation can also be viewed from figure 3. On the other hand, if the computed filter is empty, then there is obviously no need for verification. The verification cost is assumed to be $O(m)$ with the brute-force checking of the pattern.

From these facts, the best case complexity is $O(\lfloor N/L \rfloor)$, and worst case complexity is $O(\lfloor N/L \rfloor \cdot (L \cdot 16) \cdot m)$. Remembering the definitions of N and L as $N = \lceil n/16 \rceil$, and $L = \lfloor m/16 \rfloor - 1$, the best/worst time complexities approximately converges to $O(n/m)$ and $O(n.m)$ respectively, which are equivalent to standard Boyer-Moore [3] algorithm.

There are at most $L \cdot 16$ distinct filter values for any given pattern among the possible 65536 values. Hence, the probability that the filter computed on D^{i+L} hits to a non-empty set is $L \cdot 16/65536$. This indicates that verification will be requested for $\lfloor N/L \rfloor \times (L \cdot 16/65536)$ times during the whole execution, assuming characters of the text is randomly uniform distributed. The average case complexity, being sum of the filter computation time and verification computation time, is then $O(\lfloor N/L \rfloor + \lfloor N/L \rfloor \times (L \cdot 16/65536) \times m)$, which converges to $O(n/m + n \cdot m/65536)$.

5 Implementation and Experimental Results

The SSEF algorithm is implemented on 64 bit Intel Xeon processor with 3 GB of memory. All of the algorithms included in tests are compiled with *GNU C* compiler `gcc 4.1.2` with full optimization turned on by `-O3` flag.

The SSE instructions used in the study require the source data to be 16-byte aligned for best performance. The cost of misalignment is very high [16,10,11], and special attention was paid to make sure that the text is properly aligned. For that purpose the input text is loaded to the memory ensuring that it is 16-byte aligned by using *union* aggregate with $\_\_m128i$ data type introduced by SSE intrinsics as shown in figure 5.

```
typedef union{
        __m128i* data16;
        unsigned char* data;
} TEXT;
```

Figure 5. The `TEXT` data type defined for 16-byte alignment of data.

The performance of SSEF algorithm is compared with:

- Lecroq's *q-hash* algorithm, which is one of the best filtering algorithms [13], with ranks $q = 3$ (*3-hash*) and $q = 8$ (*8-hash*).
- The quick search (QS) of Sunday, which is a fast implementation of standard Boyer-Moore [3].
- The BLIM of Külekci [12], as this bit-parallel algorithm is not limited with the computer word size, and thus can be run on long patterns also.
- Fast variants of backward oracle and suffix oracle matching [1]. BOM2 and BSOM2 are especially fast on long patterns.

	$\|\Sigma\| = 256$ 2-bit encoded DNA sequence							$\|\Sigma\| = 128$ English text						
Len.	BLIM	3-hash	8-hash	QS	BOM2	BSOM2	SSEF	BLIM	3-hash	8-hash	QS	BOM2	BSOM2	SSEF
32	11,06	10,69	11,05	10,53	9,46	9,47	12,03	10,38	9,44	9,75	10,05	9,65	10,04	10,88
96	12,16	11,11	10,85	11,35	9,03	9,12	9,69	11,24	9,80	9,57	10,36	8,51	8,56	8,74
160	13,20	15,14	14,64	16,06	6,60	6,88	6,64	12,50	13,31	12,85	12,56	8,08	8,03	5,95
224	14,18	14,33	13,94	15,29	5,09	5,28	4,66	13,72	12,29	12,11	12,20	6,30	6,23	4,21
288	15,14	13,87	13,57	14,43	4,12	4,23	3,58	14,89	11,57	11,66	12,03	5,11	5,06	3,21
352	16,16	13,32	13,09	13,95	3,45	3,59	2,91	16,06	11,12	11,26	11,66	4,46	4,41	2,58
416	17,14	12,81	12,66	13,28	2,96	3,05	2,44	17,22	10,43	10,67	11,32	3,82	3,75	2,17
480	18,07	12,30	12,20	12,84	2,56	2,62	2,16	18,43	10,00	10,33	11,19	3,50	3,41	1,80
544	19,14	11,87	11,85	12,55	2,22	2,24	1,90	19,53	9,38	9,85	11,05	3,03	3,02	1,62
608	20,26	11,58	11,53	12,16	1,93	1,96	1,68	20,34	9,07	9,64	10,62	2,85	2,79	1,45
672	21,01	11,33	11,23	11,86	1,69	1,67	1,54	21,26	8,65	9,36	10,39	2,60	2,55	1,34
736	22,06	11,14	11,07	11,72	1,52	1,50	1,36	22,16	8,52	9,16	10,00	2,48	2,36	1,21
800	23,00	11,00	10,89	11,53	1,37	1,41	1,21	23,22	8,28	9,04	9,91	2,28	2,26	1,13
864	23,99	10,78	10,78	11,30	1,31	1,29	1,15	24,01	7,96	8,78	9,51	2,14	2,08	1,06
928	25,07	10,76	10,74	11,40	1,20	1,24	1,09	25,05	7,75	8,67	9,28	2,00	1,99	1,01
992	26,22	10,66	10,67	11,22	1,19	1,20	0,98	25,92	7,46	8,54	8,99	1,94	1,91	0,90
1056	27,41	10,62	10,63	11,08	1,16	1,18	0,96	27,16	7,26	8,54	8,97	1,78	1,81	0,92
1248	30,82	10,48	10,53	10,89	1,10	1,15	0,92	30,44	7,07	8,39	8,37	1,72	1,70	0,84
1440	34,19	10,40	10,51	10,51	1,11	1,16	0,82	33,75	6,75	8,20	7,93	1,60	1,64	0,80
1632	38,05	10,42	10,40	10,64	1,16	1,17	0,84	37,20	6,53	8,21	7,67	1,57	1,60	0,76
1824	41,85	10,49	10,44	10,56	1,22	1,21	0,82	41,35	6,57	8,21	7,72	1,57	1,60	0,75
2000	47,09	10,40	10,46	10,99	1,28	1,25	0,81	45,84	6,26	8,08	7,62	1,57	1,60	0,76
Avg.	27,79	11,27	11,2	11,67	2,33	2,37	2,14	27,4	8,22	9,16	9,36	2,96	2,96	1,94

Table 1. Experimental comparison of algorithms on large alphabets.

Benchmarks are conducted on various text files having small ($\Sigma = \{2, 4\}$), medium ($\Sigma = \{16, 20\}$), and large ($\Sigma = \{128, 256\}$) alphabets. In practice, small alphabets mimic the nucleic acid sequences, and middle alphabets correspond to biological sequences with larger blocks such as amino acids or proteins. Large alphabets represent the case for natural languages, and series of random bytes such as the compressed files.

The summary of the data sets[2] used in the experiments are given in Table 2. The distribution of characters are randomly uniform on all data sets except the 5^{th} one, which is a natural language text. Remembering the discussion in section 3, it is enough to consider the character codings of the alphabet while deciding on the value of bit shift amount K on test files except the English text. On natural language text file, the experiment is repeated for all possible K values as $K = 0, \ldots, 7$. It is observed that the performances are compatible for $K \in \{3, 4, 5, 7\}$, and significantly worse on $K \in \{1, 2\}$. Obviously, selecting $K = 0$ is the worst since it does not include any distinguishing power on the set of printable ascii characters.

	$\|\Sigma\|$	Data set	Size	K-bit shift
1	2	Uniformly distributed random sequence of two characters ('a' and 'b').	30 MB	6
2	4	Plain ASCII coded DNA sequence from Manzini's DNA corpus	21.6 MB	5
3	16	Uniformly distributed random sequence of 16 characters ('a' ... 'p').	30 MB	7
4	20	Uniformly distributed random sequence of 20 characters ('a' ... 't').	30 MB	7
5	128	English text from `enwik8` corpus.	20 MB	7
6	256	2-bit encoded DNA sequence from Manzini's DNA corpus.	22.7 MB	0

Table 2. Test files used in the experiments.

Patterns of length 32 to 2000 are randomly selected from the input text, and searched via the included algorithms. 100 samples are taken for each length, and each sample is matched 10 times on the text. The mean user times are recorded by `getrusage` function.

Tables 1 and 3 compare the timings of BLIM, 3-hash, 8-hash, QS, BOM2, BSOM2 and SSEF for various pattern lengths in *milliseconds*. Experimental results indicate that the SSEF algorithm is the clear winner on all tested alphabet sizes and followed by the BOM2 and BSOM2 algorithms, which are actually known to be the fastest ones on long pattern matching. The performance of BOM2 and BSOM2 are quite good, but with the increasing length of the patterns, the SSEF becomes more dominant. The performances of BOM2/BSOM2 and SSEF improves with the increased length, where Lec3 and Lec8 are not very much effected with the length.

Table 4 summarizes the average measured speeds of the algorithms in mega byte per seconds on tested alphabet sizes. Based on the overall speeds depicted in this table, the performance gain is maximum on small alphabets. SSEF is 3.62 and 2.47 times faster than its nearest successor on binary alphabet and plain text DNA sequences respectively. When medium size alphabets are concerned, it is 40 % faster than the following best. On natural language text, the performance of the BOM2/BSOM2 degrades a little bit since the underlying data is not uniform now, and thus, SSEF is 50 % more speedy in this case. When timings on 256-byte alphabets are investigated, 10 % improvement is observed according to the next best BOM2 algorithm.

SSEF is approximately more than 5 times faster than the *q-hash* family, which is one of the best representative of *filter-then-search* algorithms. Note that the speed

[2] Manzini's DNA compression benchmark corpus can be downloaded from `http://web.unipmn.it/manzini/dnacorpus`.
The enwik8.txt file is the subject of the Hutter Prize compression competition, and can be downloaded from `http://prize.hutter1.net`

	$\vert\Sigma\vert = 4$ Plain ascii DNA sequence							$\vert\Sigma\vert = 2$ Randomly uniform sequence of 2 characters						
Len.	BLIM	3-hash	8-hash	QS	BOM2	BSOM2	SSEF	BLIM	3-hash	8-hash	QS	BOM2	BSOM2	SSEF
32	15,19	10,95	10,54	49,48	16,20	20,56	11,87	37,09	52,06	14,62	212,37	39,52	52,17	16,17
96	12,30	11,04	10,40	50,27	10,72	11,71	9,45	19,12	51,98	14,54	218,18	21,27	25,40	13,06
160	13,44	12,54	14,06	51,37	15,77	16,30	6,53	20,22	51,67	19,46	224,04	31,59	35,95	8,98
224	14,59	12,60	13,42	53,10	12,22	12,53	4,56	21,38	51,88	18,45	223,00	23,60	26,61	6,32
288	15,68	12,69	12,99	50,60	10,06	10,21	3,46	22,51	50,74	18,02	213,45	19,04	21,51	4,87
352	16,84	12,64	12,65	52,89	8,69	8,81	2,85	23,65	52,40	17,47	225,81	16,20	18,06	4,00
416	17,94	12,67	12,03	50,04	7,53	7,66	2,36	24,68	51,86	16,77	218,05	14,13	15,63	3,36
480	19,04	12,74	11,62	49,88	6,75	6,83	2,03	25,92	51,53	16,27	222,09	12,61	13,87	2,91
544	20,24	12,72	11,25	49,96	6,11	6,12	1,82	26,96	51,48	15,90	218,66	11,43	12,47	2,60
608	21,31	12,57	10,92	52,87	5,56	5,65	1,61	28,09	51,17	15,45	221,21	10,50	11,38	2,36
672	22,45	12,56	10,70	51,64	5,12	5,14	1,46	29,18	49,81	15,20	216,25	9,77	10,50	2,21
736	23,50	12,42	10,47	51,07	4,76	4,78	1,35	30,32	51,23	14,90	215,31	9,16	9,75	1,95
800	24,70	12,48	10,16	51,76	4,44	4,45	1,22	31,45	52,17	14,62	220,85	8,59	9,06	1,81
864	25,87	12,21	9,98	51,40	4,13	4,20	1,15	32,46	49,82	14,58	216,94	8,19	8,55	1,62
928	26,87	12,44	9,98	51,37	3,88	3,91	1,11	33,67	50,32	14,42	219,99	7,76	8,08	1,55
992	28,04	12,25	9,80	49,86	3,68	3,70	1,02	34,88	49,84	14,44	204,53	7,38	7,62	1,46
1056	29,42	12,33	9,74	49,15	3,46	3,50	1,00	36,31	50,32	14,32	217,54	7,07	7,31	1,30
1248	33,10	12,22	9,56	49,44	2,95	3,04	0,92	39,67	49,15	14,07	210,31	6,14	6,23	1,19
1440	36,82	12,26	9,35	48,10	2,59	2,68	0,88	43,54	51,81	14,07	215,02	5,41	5,51	1,10
1632	40,67	12,18	9,16	53,39	2,40	2,46	0,83	47,15	50,50	14,15	228,67	4,82	4,89	1,00
1824	44,68	12,18	9,18	51,01	2,27	2,27	0,84	51,21	51,56	14,27	223,11	4,42	4,44	0,96
2000	50,42	12,17	8,96	50,21	2,18	2,37	0,78	57,04	51,08	14,04	214,72	4,16	4,04	0,93
Avg.	29,62	12,27	10,31	50,81	5,23	5,48	2,12	36,80	51,01	15,04	218,52	10,47	11,64	2,89

a) Benchmarks on small alphabet sequences.

	$\vert\Sigma\vert = 20$ Randomly uniform sequence of 20 characters							$\vert\Sigma\vert = 16$ Randomly uniform sequence of 16 characters						
Len.	BLIM	3-hash	8-hash	QS	BOM2	BSOM2	SSEF	BLIM	3-hash	8-hash	QS	BOM2	BSOM2	SSEF
32	14,89	14,40	14,60	15,50	13,28	13,50	15,76	15,00	14,46	14,62	16,75	13,32	13,68	16,40
96	15,87	15,30	14,48	15,12	12,32	12,47	12,79	16,00	15,34	14,48	16,15	12,48	12,71	13,06
160	17,04	19,67	19,37	15,21	10,67	11,10	8,78	17,03	18,89	19,44	15,96	11,79	12,13	9,01
224	18,16	19,23	18,50	15,22	8,33	8,66	6,16	18,16	18,61	18,55	16,12	9,26	9,44	6,32
288	19,29	18,90	18,02	15,22	6,96	7,21	4,76	19,34	18,48	17,96	16,18	7,57	7,74	4,91
352	20,42	18,70	17,38	15,20	6,04	6,21	3,86	20,45	18,30	17,40	16,06	6,39	6,54	3,99
416	21,55	18,55	16,83	15,30	5,30	5,48	3,31	21,54	18,30	16,81	16,13	5,53	5,70	3,40
480	22,64	18,34	16,20	15,26	4,69	4,92	2,89	22,67	18,17	16,28	15,94	4,84	5,05	2,92
544	23,78	18,10	15,78	15,20	4,21	4,40	2,58	23,86	18,02	15,89	16,07	4,34	4,53	2,58
608	24,89	17,96	15,39	15,38	3,83	4,06	2,33	24,95	17,90	15,43	16,11	3,98	4,15	2,33
672	26,04	17,70	15,14	15,20	3,54	3,71	2,13	26,10	17,76	15,11	16,04	3,64	3,80	2,15
736	27,19	17,68	14,86	15,22	3,23	3,43	1,94	27,19	17,54	14,87	15,97	3,38	3,53	1,92
800	28,26	17,50	14,56	15,22	3,01	3,14	1,77	28,22	17,54	14,64	16,14	3,14	3,27	1,74
864	29,45	17,32	14,52	15,18	2,74	2,96	1,64	29,50	17,42	14,48	16,06	2,97	3,06	1,64
928	30,49	17,18	14,49	15,18	2,57	2,98	1,49	30,61	17,29	14,38	16,16	2,84	2,89	1,53
992	31,73	17,14	14,31	15,14	2,41	2,65	1,38	31,80	17,26	14,30	16,16	2,69	2,81	1,47
1056	33,08	17,05	14,23	15,26	2,21	2,37	1,32	33,04	17,23	14,31	16,02	2,54	2,68	1,34
1248	36,59	16,84	14,17	15,18	1,97	2,08	1,11	36,78	16,96	14,13	16,10	2,30	2,35	1,17
1440	40,34	16,70	14,06	15,26	1,82	1,86	1,01	40,50	16,85	13,99	16,23	2,11	2,15	1,08
1632	44,08	16,56	14,04	15,24	1,71	1,81	0,96	44,04	16,86	14,04	16,08	2,00	2,01	1,00
1824	48,05	16,45	14,08	15,26	1,73	1,76	0,92	48,04	16,54	14,05	16,13	1,92	1,94	1,00
2000	53,94	16,39	14,09	15,20	1,68	1,80	0,94	53,16	16,51	14,14	16,20	1,93	2,02	0,92
Avg.	33,22	17,23	15,04	15,27	3,84	4,00	2,81	33,06	17,20	15,00	16,11	4,13	4,24	2,90

b) Benchmarks on medium alphabet sequences.

Table 3. Experimental comparison of algorithms on small and medium alphabets.

$\|\Sigma\|$	BLIM	3-hash	8-hash	QS	BOM2	BSOM2	SSEF
2	815,24	588,07	1994,18	137,29	2865,06	2577,82	10382,87
4	729,20	1760,55	2094,16	425,14	4126,17	3939,90	10201,31
16	907,38	1744,17	1999,72	1862,76	7269,64	7074,85	10347,06
20	903,09	1741,58	1994,93	1965,23	7809,45	7507,04	10659,09
128	729,85	2433,05	2182,84	2135,84	6750,49	6748,78	10307,95
256	816,72	2015,00	2026,76	1945,14	9749,29	9591,72	10585,84

Table 4. Average speed of the tested algorithms in MB/sec for each $|\Sigma|$ alphabet size.

of the proposed algorithm is not much effected with the size or distribution of the alphabet unlike its nearest competitors BOM2 and BSOM2.

6 Conclusion

This study introduced a filter-then-search type pattern matching algorithm for long patterns benefiting from computers intrinsic SIMD instructions. Using SIMD intrinsics has not been much addressed in pattern matching, and this study is an initial exploration of designing algorithms according to that technology, which is developing very fast.

The proposed SSEF algorithm is implemented on Intel's SSE (version 2) technology. Experimental benchmarks showed that on every alphabet sizes it is faster than all competitors included in this study. Considering the orders of magnitude performance gain on small and medium alphabet sizes, SSEF becomes a strong alternative for exact matching of long patterns on biological sequences.

The best and worst case time complexities being $O(n/m)$ and $O(n{\cdot}m)$ respectively are identical with the classical Boyer-Moore type algorithms. The main improvement comes with the average case complexity of $O(n \cdot m/2^{16})$. Note that the performance of the algorithm is independent of the alphabet size (assuming $|\Sigma| > 1$), and conducted experiments proves this empirically also.

References

1. C. Allauzen, M. Crochemore, and M. Raffinot: *Factor oracle: A new structure for pattern matching*, in Proceedings of SOFSEM'99, vol. 1725 of LNCS, Springer Verlag, 1999, pp. 291–306.
2. R. Baeza-Yates and G. Gonnet: *A new approach to text searching*. Communications of the ACM, 35(10) 1992, pp. 74–82.
3. R. Boyer and J. Moore: *A fast string searching algorithm*. Communications of the ACM, 20(10) 1977, pp. 762–772.
4. C. Charras and T. Lecroq: *Handbook of exact string matching algorithms*, King's Collage Publications, 2004.
5. M. Crochemore and W. Rytter: *Jewels of stringology*, World Scientific Publishing, 2003.
6. K. Fredriksson: *Faster string matching with super–alphabets*, in Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE'2002), LNCS 2476, Springer–Verlag, 2002, pp. 44–57.
7. K. Fredriksson and S. Grabowski: *Practical and optimal string matching*, in Proceedings of the 12th International Symposium on String Processing and Information Retrieval (SPIRE'2005), LNCS 3772, Springer–Verlag, 2005, pp. 374–385.

8. M. Hassaballah, S. Omran, and Y. Mahdy: *A review of SIMD multimedia extensions and their usage in scientific and engineering applications*. The Computer Journal, 51(6) November 2008, pp. 630–649.
9. J. Holub and B.Durian: *Fast variants of bit parallel approach to suffix automata*. Unpublished Lecture, University of Haifa, April 2005.
10. I. Hurbain and G. Silber: *An empirical study of some x86 simd integer extensions*, in Proceedings of CPC'2006, 12th International Workshop on Compilers for Parallel Computers, Spain, January 9–11 2006.
11. Intel Corporation, *Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual*, 2001.
12. M. O. Külekci: *A method to overcome computer word size limitation in bit-parallel pattern matching*, in Proceedings of 19th International Symposium on Algorithms and Computation, ISAAC'2008, S.-H. Hong, H. Nagamochi, and T. Fukunaga, eds., vol. 5369 of Lecture Notes in Computer Science, Gold Coast, Australia, December 2008, Springer Verlag, pp. 496–506.
13. T. Lecroq: *Fast exact string matching algorithms*. Information Processing Letters, 102(6) 2007, pp. 229–235.
14. G. Navarro and M. Raffinot: *Fast and flexible string matching by combining bit-parallelism and suffix automata*. ACM Journal of Experimental Algorithms, 5(4) 2000, pp. 1–36.
15. H. Peltola and J. Tarhio: *Alternative algorithms for bit-parallel string matching*, in LNCS 2857, Proceedings of SPIRE'2003, 2003, pp. 80–94.
16. A. Shahbahrami, B. Juurlink, and S. Vassiliadis: *Performance impact of misaligned accesses in simd extensions*, in Proceeedings of ASAP'05, IEEE Conference on Application Spesific Systems Architecture Procesors, Washington,DC, USA, 2005, pp. 393–398.
17. S. Wu and U. Manber: *Agrep – a fast approximate pattern-matching tool*, in Proceedings of USENIX Winter 1992 Technical Conference, 1992, pp. 153–162.
18. S. Wu and U. Manber: *Fast text searching allowing errors*. Communications of the ACM, 35(10) 1992, pp. 83–91.

Validation and Decomposition of Partially Occluded Images with Holes

Julien Allali[1], Pavlos Antoniou[2], Costas S. Iliopoulos[2], Pascal Ferraro[1], and Manal Mohamed[2]

[1] LaBRI, University of Bordeaux I, UMR5800, 33405 Talence, France
[2] Dept. of Computer Science, King's College London, London WC2R 2LS, England, UK

Abstract. A partially occluded image consists of a set of objects where some may be partially occluded by others. Validating occluded images distinguishes whether a given image can be covered by the members of a finite set of objects, where both the image and the object range over identical alphabet. The algorithm presented here validates a one-dimensional image x of length n, over a given set of objects all of equal length and each composed of two parts separated by a transparent hole.

Keywords: valid image, approximate valid image, image decomposition

1 Introduction

In recent studies of repetitive structures of strings, generalized notions of periods have been introduced [2]. Here we present practical methods to study the following type of regularity: we want to "*cover*" a string using a set of "*objects*".These objects may "*occlude*" each other and may be separated by a hole.

Validating partially occluded images is a classical problem in computer vision and its computational complexity is exponential. An input image is valid, if it can be composed from a members of a finite set of objects, with some of the appearing objects being partially occluded by other ones. This problem is also typical in pattern recognition and computer graphics. There is a great number of artificial intelligence and neural net solutions to this problem.

Validating occluded one dimensional images has been a well studied problem in algorithm design. Iliopoulos and Simpson [6] focused on the theoretical aspect of the problem and produced a sequential on-line algorithm for validating occluded one-dimensional images. Furthermore, different aspects of this problem have been studied and solved by Iliopoulos and Reid. In [5], the authors provided a linear time solution to the problem in the presence of errors, in [4] they presented an optimal $\mathcal{O}(\log\log n)$-time algorithm using parallel computation and in [3] solved the problem for discrete two-dimensional partially occluded images in linear time.

In this paper, we move a step forward, based on the above analyses and we extend the previous work by considering the validity of a family of images, that we call *valid images with holes*. In this context, given a set of objects $s_1, \ldots, s_k$, each composed of two parts separated by a small transparent hole, an image x of length n is a valid image with hole, if x is iteratively obtained from a string $z = \#^n$ by substituting substrings of z by some objects s_i , for some $i \in \{1..k\}$ and a special "background" symbol #. We focus on designing an on-line algorithm for testing images in one dimension for validity, with restricted set of objects, e.g., objects of the same length, that are consisting of two parts separated by a hole of small size.

Julien Allali, Pavlos Antoniou, Costas S. Iliopoulos, Pascal Ferraro, Manal Mohamed: Validation and Decomposition of Partially Occluded Images with Holes, pp. 129–136.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

The paper is organized as follows. In Section 2, we introduce basic definitions and notations used in this paper. In Section 3, we present the principles for validating images in one dimension. In Section 4, the main validating algorithm is presented with its time complexity analysis. Finally, we state our conclusions in Section 5.

2 Background

An *alphabet* Σ is a set of elements that are called letters, characters or symbols. A *string* x is a sequence of zero or more letters from Σ, that is $x[1]x[2]\cdots x[n]$ with $x[i] \in \Sigma$, $1 \leq i \leq n$. The *length* of x, denoted by $|x|$, is the total number of letters in x. The string of length zero is the empty string ε. The string xy is a *concatenation* of two strings x and y.

A string y is a *substring* of x, if and only if, there exist two strings u and v such that $x = uyv$. A string u is a *prefix* (respectively *suffix*) of x, if and only if, there exists a string v over such that $x = uv$ (respectively $x = vu$). If $v \neq \varepsilon$ then u is a *proper prefix* (respectively *proper suffix*) of x.

Additionally, $\mathit{prefix}_p(x)$ denotes the first p letters of x and $\mathit{suffix}_p(x)$ denotes the last p letters of x. Given two strings $x = x[1]x[2]\cdots x[n]$ and $y = y[1]y[2]\cdots y[m]$, such that $x[n-i+1]\cdots x[n] = y[1]\cdots y[i]$ for some $i \geq 1$ (that is such that x has a suffix equal to a prefix of y), the string $x[1] \ \cdots \ x[n]y[i+1]\cdots y[m]$ is called a *superposition* of x and y with i overlap. A string w of x is called *a cover* of x if and only if an extension of x can be constructed by concatenations and superposition of w.

Valid Image over set of Objects:

Definition 1. Let x be a string of length n over an alphabet Σ and let the dictionary $\mathcal{O} = \{s_1, \ldots, s_m\}$ be a set of strings called the *objects* also over Σ. Then x is called a *valid image* if and only if $x = z_i$ for some $i \geq 0$, where

$$\begin{aligned} z_0 &= \#^n \\ z_{i+1} &= \mathrm{prefix}_p(z_i)\ s_l\ \mathrm{suffix}_q(z_i)\,. \end{aligned} \tag{1}$$

for some $s_l \in \mathcal{O}$ and $p, q \in \{0, \ldots, n-1\}$ such that $p + |s_m| + q = n$. □

Equation (1) is called the *substitution rule* and the sequence $z_0, z_1, \ldots, z_i$ is called the *generating sequence* of x. The number of distinct generating sequences was proved to be exponential [6].

An example of such generating sequences for a specific string is as follows. Let $\mathcal{O} = \{s_1 = abc, s_2 = acde, s_3 = ade, s_4 = dc, s_5 = abd\}$. Then $x = ababab acdedcdcade$ is a valid image over $\mathcal{O}$ with generating sequence:

$$z_0 = \#^{17},$$

$$z_1 = \underline{abc}\#^{14},$$

$$z_2 = abc\#^{11}\underline{ade},$$

$$z_3 = ab\underline{abc}\#^9 ade,$$

$$z_4 = abab\underline{abc}\#^7 ade,$$

$$z_5 = ababab\underline{acde}\#^4 ade,$$

$$
\begin{aligned}
z_6 &= abababacde\underline{dc}\#^2ade,\\
z_7 &= abababacdedc\underline{dc}ade.
\end{aligned}
$$

Note that the generating sequence of x is not unique. The following sequence:

$$
\begin{aligned}
z_0 &= \#^{17},\\
z_1 &= \underline{abd}\#^{14},\\
z_2 &= ab\underline{abc}\#^{12},\\
z_3 &= abab\underline{abc}\#^{10},\\
z_4 &= abababc\#^7\underline{ade},\\
z_5 &= abababc\#^3\underline{dc}\#^2ade,\\
z_6 &= abababc\#^3dc\underline{dc}ade,\\
z_7 &= ababab\underline{acde}dcdcade.
\end{aligned}
$$

also generates x as a valid image over $\mathcal{O}$.

Valid Image over Set of Objects with Hole:
Let x be a string of length n over an alphabet Σ and let the dictionary $\mathcal{O} = \{s_1, \ldots, s_k\}$ be a set of strings called the objects, *where each object s_i is composed of two strings s_i^l and s_i^r* separated *by a hole of length h. Then x is called a* valid image *if and only if $x = z_i$ for some $0 \leq i$, where*

$$
\begin{aligned}
z_0 &= \#^n\\
z_{i+1} &= \mathrm{prefix}_p(z_i)\ s_m\ \mathrm{suffix}_q(z_i)\,. \qquad (2)
\end{aligned}
$$

for some $s_m \in \mathcal{O}$ and $p, q \in \{0, \ldots, n-1\}$ such that $p + |s_m| + q = n$.

Figure 1, presents the notion of finding the objects comprising an image. If the image is observed from above, one can see some of the objects are partially occluded by others but can see some of the covered ones *through* the hole. We are trying to decompose what the eye sees to its sources. In this example of Figure 1 the valid image of the objects is composed of the following elements:
Image = *prefix*(s_2^l) s_1^l *suffix*(s_3^l) *substring*(s_4^l) *prefix*(s_2^r) s_1^r *suffix*(s_3^r) *suffix*(s_4^r)

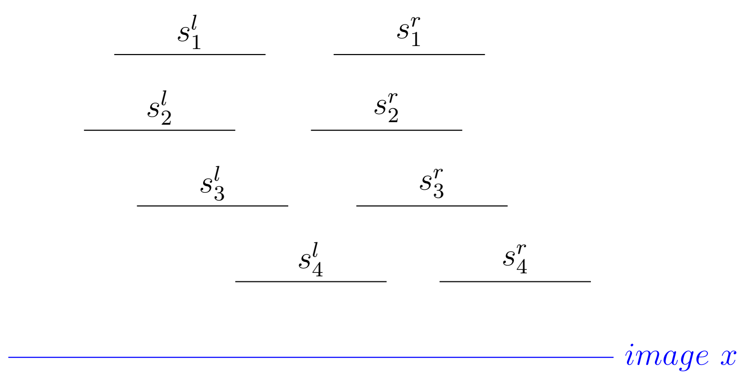


Figure 1. Image consisting from objects separated by a hole of same length.

In this paper, we consider the problem of validation of an image over a set of objects with holes. Each object $s_i \in \mathcal{O}$ consists of a *left part* (*head*) and a *right part*(*tail*) separated by a transparent hole of length h. We denote the left part of s_i as s_i^l and the the right part as s_i^r. For simplicity, we require that $|s_i^l| = |s_i^r|$ and $h \ll |s_i^l|$, for each $s_i \in \mathcal{O}$.

The definition of a valid image implies that constituent objects are contained within the image x. That is, there is no s_i for all $i \in \{1, \ldots, k\}$ that is 'cut' at $x[1]$ or $x[n]$.

This leads to the following facts:

If x is a valid image over $\mathcal{O} = \{s_1, s_2, \ldots, s_k\}$, then for some $i \in \{1, \ldots, k\}$,

Fact 1: there exists a suffix $\bar{s}_i^r$ of s_i^r that is also a suffix of x.

Fact 2: there exists a prefix $\hat{s}_i^l$ of s_i^l that is also a prefix of x.

Fact 3: there is no suffix of a left part s_i^l that occurs in x ending at position ℓ, where $\ell > n - h - |s_i^r|$.

Fact 4: there is no prefix of a right part s_i^r that occurs in x at position ℓ', where $\ell' < |s_i^l| + h$.

3 Validation of Images with Objects of Equal Length

In this section, we start by defining what part of a valid image one should expect to see within the hole *i.e.* between the left and the right parts of an object. Subsequently, we proceed and present the main mechanism for validating one-dimensional images over a set of objects with holes.

Given a set of objects $\mathcal{O}$, a string b of length h is a *binding* if it is a concatenation of the following three (possibly empty) parts:

Part 1: is a sequence of suffixes of left/right parts of objects in $\mathcal{O}$, where the leading (first) suffix is a suffix of a left part of an object.

Part 2: is a substring of a left/right part of an object.

Part 3: is a sequence of prefixes of left/right parts of objects in $\mathcal{O}$, where the leading (last) prefix is a prefix of a right part of an object.

Note that any substring of a left or a right part of an object is also a binding if it is of length h. A binding b is *satisfied*, if and only if, the length of the part of the valid image following the binding is big enough to insure that each object from $\mathcal{O}$ appears within the hole is totally occluded by the image.

Theorem 2. *Let x be a string over Σ. Let $\mathcal{O} = \{s_1, s_2, \ldots, s_k\}$ be a set of objects all of the same length and each composed of left part s_i^l and right part s_i^r separated by a hole of length h. The string x is a valid image over $\mathcal{O}$ if and only if*

$$x = \hat{s}_i^l y \qquad with \qquad i \in \{1..k\}, \tag{3}$$

or

$$x = y\bar{s}_i^r \qquad with \qquad i \in \{1..k\}, \tag{4}$$

or

$$x = y\widetilde{s}_i w \qquad with \qquad i \in \{1..k\}, \tag{5}$$

or

$$x = y\bar{s}_i^l b\hat{s}_i^r z \qquad with \qquad i \in \{1..k\}, \tag{6}$$

where $\hat{s}_i^l$, $\bar{s}_i^r$ *and* $\widetilde{s}_i$ *denote a prefix of the left part* s_i^l, *suffix of the right part* s_i^r *and a substring of either parts of* s_i *respectively,* y *and* w *are valid images and* b *is a satisfied binding.*

The above theorem provides the main mechanism for validating images over a set of objects with holes and all of equal length. Equations (3) and (4) are restatements of Facts 1 and 2. Equations (5) and (6) state what one should expect at the decomposition of two valid sub-images.

If an image x is of the form (5), and $s_i = u\widetilde{s}_i v$ for some strings u, v and a non-empty substring $\widetilde{s}_i$ of either the left or the right part of s_i, then x is a valid image, since x can be generated by the sequence:

$z_0 = \#^n$, $z_0 = \#^p s_i \#^q$, where $p = |y| - |u|$,
followed by an application of the generating sequence of y on the first $|y|$ symbols of z_1 and the generating sequence of w on the last $|w|$ symbols of z_1.

If an image x is of the form (6), and s_i^l, and s_i^r are both the left and the right part of s_i separated by a hole of length h, then x is a valid image, since x can be generated as:

$z_{i+1} = \mathit{prefix}(z_i) s_i^l b s_i^r \mathit{suffix}(z_i)$,
where b is the part of z_i appearing in the hole separating the left and the right part of s_i, followed by an application of the generating sequence of y on the first $|y| = |\mathit{prefix}(z_i)| + |s_i^l| - |\bar{s}_i^l|$ symbols of z_1 and the generating sequence of w on the last $|w|$ symbols of z_1.

4 An On-Line Algorithm

Here we present the algorithm for validating an image over a set of objects with holes and of equal length. Algorithm 1 presents the main commands of the algorithm in the form of pseudocode.

Algorithm 1 On-line Image Validation ALgorithm

Input: image $x[1\ldots,n]$, the set of objects $\mathcal{O} = \{s_1, s_2 \ldots s_k\}$ all of equal length.
Output: T if and only if x is a valid image, F otherwise.

initialization
valid$[0,\ldots,n] \leftarrow [T, F,\ldots,$ F$]$
p_valid$[0] \leftarrow 1$
$last\_prefix \leftarrow last\_valid \leftarrow 0$
begin
for $i = 1$ to N **do**
  do
  p_valid$[i] \leftarrow last_valid$
  case study
  (1) $\hat{s}^l_j = x[\ell..i]$ is the longest **prefix** of some s^l_j
  if valid$[\ell-1] =$ T OR prefix$[\ell-1] =$ T **then**
    prefix$[\max\{last_prefix+1,\ell\}\ldots i] \leftarrow$ T
  end if
  if $x[p\_valid[\ell-1]+1\ldots\ell-1]$ is a substring of some $s_j \in \mathcal{O}$ **then**
    prefix$[\max\{last_prefix+1,\ell\}\ldots i] \leftarrow$ T
  end if
  if p_valid$[\ell-1] \geq \ell - |s_j| - 1$ **then**
    Return "Invalid Image"
  end if
  $last\_prefix \leftarrow i$
  (2) $\hat{s}^r_j = x[\ell..i]$ is the longest **prefix** of some s^r_j.
  if prefix$[\ell-1] =$ T and $first\_prefix \leq \ell - h + |s^r_j|$ **then**
    prefix$[\max\{last_prefix+1,\ell\}\ldots i] \leftarrow$ T
  end if
  if $\hat{s}^r_j = s^r_j$ **then**
    valid$[i] \leftarrow$ T
    $last\_valid \leftarrow i$
  end if
  if $l\_suffix[j][\ell-h-1] =$ T and $x[\ell-h..\ell-1]$ is a satisfied binding **then**
    prefix$[\max\{last_prefix+1,\ell\}\ldots i] \leftarrow$ T
  end if
  if $\hat{s}_j = s_j$ **then**
    valid$[i] \leftarrow$ T
    $last\_valid \leftarrow i$
  end if
  (3) $\bar{s}^l_j = x[\ell..i]$ is the largest **suffix** of some s^l_j.
  $l\_suffix[j][i] \leftarrow$ T
  (4) $\bar{s}^r_j = x[\ell..i]$ is the largest **suffix** of some s^r_j.
  if p_valid$[i] \geq \ell - 1$ **then**
    valid$[i] \leftarrow$ T
    $last\_valid \leftarrow i$
  end if
end for

The algorithm is based on Facts 1-4 as well as the following principles:

(a) The occurrence of a proper prefix of either a left or a right part of an object in a valid image must be followed by a prefix (not necessarily proper) of a left or a right part of an object.
(b) If the occurrence of a proper prefix of either a left or a right part of an object is followed by an occurrence of a proper suffix of either a left or a right part of an object, then the image is not valid. In a valid image, the occurrence of a proper suffix of an object is always preceded by the suffix of either a left or a right part of an object.
(c) The occurrence of a suffix of either a left or a right part of an object can be followed by either a prefix or a substring or a suffix.

(d) If an occurrence of a suffix of a left part of an object is not followed by either an occurrence of a prefix of its corresponding right part in a distance h or an occurrence of a prefix of a left part of an object in a distance at most h, then the image is not valid. In both cases a satisfied binding should separate the two parts.
(e) The occurrence of a substring in a valid image may be preceded by and followed by valid images.

Preprocessing Stage
In this stage we preprocess the set of objects. We compute the suffix tree of the set of the left and right parts of all objects in $\mathcal{O}$ [7,9,8]. This data structure will allow us to perform a constant time on-line checks whether a suffix, or a substring of s_j^l/s_j^r occurs in any position of x. We will also build the Aho-Corasick automaton [1] for the set of the left and right parts of all objects in $\mathcal{O}$ that will allow us to compute the largest prefixes of s_j^l/s_j^r occurring in x.

Main Algorithm
At the beginning of step i the algorithm has already determined whether $x[1..j]$ is a valid image or not, for all $j \in \{1..i-1\}$. Moreover, the algorithm should determine by the end of the current step whether $x[1..j]$ is valid or not for $j \in \{1..i\}$. This is achieved by examining the suffixes of $x[1..i]$. There are six possible cases: A suffix of $x[1..i]$ can be either a prefix of a left part, a prefix of a right part, a suffix of a left part, a suffix of a right part, a substring, a binding or a complete part of an object s_j for some $j \in \{1..k\}$. Otherwise, the string is not a valid image (Theorem 2).

Let $\hat{s}_j^l = x[\ell..i]$ be the longest prefix of a left part of an object in $\mathcal{O}$ that is also a suffix of $x[1..i]$. A prefix of a left part of an object is preceded by either a valid image, or a proper prefix of left/right part an object or a substring of an object.

- If *valid*$[\ell-1]$ is marked *TRUE*, then $x[1..\ell-1]$ is a valid image and position ℓ could be the beginning of a valid sub-image, thus we mark *prefix*$[i]$ = *TRUE*, *first-prefix* $= \ell$ and *last-prefix* $= i$.
- If *prefix*$[\ell-1]$ is marked *TRUE*, then we have a chain of prefixes, thus we mark *prefix*$[i]$ = *TRUE* and *last-prefix* $= i$.
- If there is no prefix of a left/right part of an object or a valid image preceding $\hat{s}_j^l$, then $x[1..i]$ is valid if and only if $x[\textit{previous-valid}[\ell-1]+1..\ell-1]$ is a substring of left/right part of an object or $x[\textit{previous-valid}[\ell-1]+1..i]$ is a prefix of a satisfied binding. If $x[\textit{previous-valid}[\ell-1]+1..\ell-1]$ is a substring then ℓ is the start of a valid image.

Let $\hat{s}_j^r = x[\ell..i]$ be the longest prefix of a right part of an object in $\mathcal{O}$ that is also a suffix of $x[1..i]$. Similarly, a prefix of an object is preceded by either a proper prefix of left/right part an object or a substring of an object.

- If *prefix*$[\ell-1]$ is marked *TRUE* and *first-prefix* $\leq \ell - h + |s_j^r|$, then we have a chain of prefixes thus we mark *prefix*$[i]$ = *TRUE* and *last-prefix* $= i$. If $\hat{s}_j^r = s_j^r$ (a complete left part), then $x[1..i]$ is a valid image and we mark the relevant array as *TRUE*.
- If *l-suffix*$[j][\ell-h-1]$ is marked *TRUE* and $x[\ell-h..\ell-1]$ is a satisfied binding then we have a prefix of a valid image (Eq. (6)), thus we mark *prefix*$[i]$ = *TRUE* and *last-prefix* $= i$. If $\hat{s}_j^r = s_j^r$ (a complete left part), then $x[1..i]$ is a valid image and we mark the relevant array as *TRUE*.

Let $\bar{s}^l_j = x[\ell..i]$ be the longest suffix of a left part of an object in $\mathcal{O}$ that is also a suffix of $x[1..i]$. If *valid*$[\ell-1]$ then *l-suffix*$[j][i]$ is marked *TRUE*.

Finally, let $\bar{s}^r_j = x[\ell..i]$ be the longest suffix of a right part of an object in $\mathcal{O}$ that is also a suffix of $x[1..i]$. Note that, in a valid image, a suffix $\bar{s}^r_j$ is always preceded by a valid image.

– If *previous-valid*$[\ell-1] \geq \ell-1$, then $x[1..i]$ is valid.
– If there is no valid image preceding $\bar{s}^r_j$, then $x[1..i]$ is valid if and only if the length of $i-$*previous-valid*$[\ell-1] < |s_j|$.

Theorem 3. *Algorithm 1 validates an image x over a set $\mathcal{O}$ of objects of equal length and all and each composed of two parts separated by a hole in linear $O(|x|+|\mathcal{O}|)$ time.*

Proof. The construction of the Aho-Corasick automaton and the suffix tree of the dictionary $\mathcal{O}$ both require $O(|\mathcal{O})|$ time.

At Stage i, finding the largest suffix that is a prefix of some part of an object requires constant time. At Stage $i-1$, we have traced on the Aho-Corasick automaton the largest prefix of a part of an object that is a suffix of $x[1..i-1]$; on Stage i, we can either extend this prefix with one symbol, $x[i]$, or we can follow the failure link that lead to the largest such prefix. Each of the other lines of Algorithm 1 requires constant time and thus the bound on the running time follows.

5 Conclusions

We have presented an on-line algorithm that determines whether a given image is valid or not over a given set of objects with holes where each object composed of two parts separated by a transparent hole. We have solved the problem for a restricted set of objects. *I.e.* objects of the same lengths and presented a linear time algorithm. As future work, the algorithm may be modified in the same way as the original validation algorithm by [6], in order to deal with a set of objects of different lengths. Another interesting problem is the computation of the depth of an object in an image, *i.e.* the number of rules applied after the placement of an object in an image.

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search.* Commun. ACM, 18(6) 1975, pp. 333–340.
2. C. S. Iliopoulos and L. Mouchard: *Quasiperiodicity and string covering*. Theoretical Computer Science, 218(1) 1999, pp. 205–216.
3. C. S. Iliopoulos and J. F. Reid: *Validating and decomposing partially occluded two-dimensional images*, in Proc. Prague Stringology Club Workshop (PSCW'98), J. Holub and M. Šimánek, eds., 1998, pp. 83–94.
4. C. S. Iliopoulos and J. F. Reid: *Optimal parallel analysis and decomposition of partially occluded strings*. Parallel Computing, 26(4) 2000, pp. 483–494.
5. C. S. Iliopoulos and J. F. Reid: *Decomposition of partially occluded strings in the presence of errors*. International Journal of Pattern Recognition and Artificial Intelligence, 15(7) 2001, pp. 1129–1142.
6. C. S. Iliopoulos and J. Simpson: *On line validation and analysis of partially occluded images*. Journal of Automata, Languages and Combinatorics, 6(3) 2001, pp. 291–303.
7. E. M. McCreight: *A space-economical suffix tree construction algorithm.* J. ACM, 23(2) 1976, pp. 262–272.
8. E. Ukkonen: *On-line construction of suffix trees*. Algorithmica, 14(3) 1995, pp. 249–260.
9. P. Weiner: *Linear pattern matching algorithms*, in SWAT '73: Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973), Washington, DC, USA, 1973, IEEE Computer Society, pp. 1–11.

Compressing Bi-Level Images by Block Matching on a Tree Architecture

Sergio De Agostino

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

Abstract. A work-optimal $O(\log M \log n)$ time parallel implementation of lossless image compression by block matching of bi-level images is shown on a full binary tree architecture under some realistic assumptions, where n is the size of the image and M is the maximum size of the match. Decompression on this architecture is also possible with the same parallel computational complexity. Such implementations have no scalability issues.

Keywords: lossless compression, sliding dictionary, bi-level image, tree architecture

1 Introduction

Storer suggested that fast encoders are possible for two-dimensional lossless compression by showing a square greedy matching heuristic for bi-level images, which can be implemented by a simple hashing scheme [6]. Rectangle matching improves the compression performance, but it is slower since it requires $O(M \log M)$ time for a single match, where M is the size of the match [7]. Therefore, the sequential time to compress an image of size n by rectangle matching is $\Omega(n \log M)$.

The technique is a two-dimensional extension of LZ1 compression [5]. Simple and practical heuristics exist to implement LZ1 compression by means of hashing techniques [2], [9], [10]. The hashing technique used for the two-dimensional extension is even simpler.

Among the different ways of reading an image, we assume that the rectangle matching compression heuristic is scanning an $m \times m'$ image row by row (*raster* scan). A 64 K table with one position for each possible 4×4 subarray is the only data structure used. All-zero and all-one rectangles are handled differently. The encoding scheme is to precede each item with a flag field indicating whether there is a monochromatic rectangle, a match, or raw data. When there is a match, the 4×4 subarray in the current position is hashed to yield a pointer to a copy. This pointer is used for the current rectangle greedy match and then replaced in the hash table by a pointer to the current position. As mentioned above, the procedure for computing the largest rectangle match with left upper corners in positions (i, j) and (k, h) takes $O(M \log M)$ time, where M is the size of the match. This procedure can be used for computing the largest monochromatic rectangle in a given position (i, j) as well. If the 4×4 subarray in position (i, j) is monochromatic, then we compute the largest monochromatic rectangle in that position. Otherwise, we compute the largest rectangle match in the position provided by the hash table and update the table with the current position. If the subarray is not hashed to a pointer, then it is left uncompressed and added to the hash table with its current position. The positions covered

Sergio De Agostino: Compressing Bi-Level Images by Block Matching on a Tree Architecture, pp. 137–145.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

by matches are skipped in the linear scan of the image and the sequential time to compress an image of size n by rectangle matching is $\Omega(n \log M)$. We want to point out that besides the proper matches we call a match every rectangle of the parsing of the image produced by the heuristic. We also call pointer the encoding of a match.

The analysis of the running time of these algorithms involve a so called *waste factor*, defined as the average number of matches covering the same pixel. In [7], it is conjectured that the waste factor is less than 2 on realistic image data. Therefore, the square greedy matching heuristic takes linear time while the rectangle greedy matching heuristic takes $O(n \log M)$ time. On the other hand, the decoding algorithms are both linear.

Parallel coding and decoding algorithms were shown in [3] requiring $O(\log M \log n)$ time and $O(n/\log n)$ processors on the PRAM EREW, mesh of trees, pyramidal, and multigrid architectures. The parallel encoder and decoder on the pyramid and the multigrid require some realistic assumptions. Under the same realistic assumptions, we show in this paper how to implement such encoder/decoder with the same parallel complexity on a full binary tree architecture. In section 2, we explain the block matching heuristic. In section 3, we describe scalable algoritms for coding and decoding bi-level images compressed by block matching on an exclusive read, exclusive write shared memory parallel machine. In section 4, we show how such parallel implementations can be run on a tree architecture. Conclusions and future work are given in section 5.

2 The Block Matching Heuristic

The compression heuristic scans an image row by row. We denote with $p_{i,j}$ the pixel in position (i, j). The procedure for finding the largest rectangle with left upper corner (i, j) that matches a rectangle with left upper corner (k, h) is described in figure 1.

$w = k$;

$r = i$;

$width = m$;

$length = 0$;

$side1 = side2 = area = 0$;

repeat

    Let $p_{r,j} \cdots p_{r,j+\ell-1}$ be the longest match in (w, h) with $\ell \le width$;

    $length = length + 1$;

    $width = \ell$;

    $r = r + 1$;

    $w = w + 1$;

    if $(length * width > area)$ {

        $area = length * width$;

        $side1 = length$;

        $side2 = width$;

    }

until $area \ge width * (i - k + 1)$ or $w = i + 1$

Figure 1. Computing the largest rectangle match in (i, j) and (k, h).

At the first step, the procedure computes the longest possible width for a rectangle match in (i, j) with respect to the position (k, h). The rectangle $1 \times \ell$ computed at the first step is the current rectangle match and the sizes of its sides are stored in *side*1 and *side*2. In order to check whether there is a better match than the current one, the longest one-dimensional match on the next row and column j, not exceeding the current width, is computed with respect to the row next to the current copy and to column h. Its length is stored in the temporary variable *width* and the temporary variable *length* is increased by one. If the rectangle R whose sides have size *width* and *length* is greater than the current match, the current match is replaced by R. We iterate this operation on each row until the area of the current match is greater or equal to the area of the longest feasible *width*-wide rectangle, since no further improvement would be possible at that point. For example, in figure 2 we apply the procedure to find the largest rectangle match between position $(0, 0)$ and $(6, 6)$.

0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	step 1
0	0	1	1	1	0	0	1	0	0	0	0	0	1	0	step 2
1	0	1	1	1	0	0	1	0	1	0	0	1	1	1	step 3
0	1	1	0	1	1	0	0	0	0	0	0	0	1	1	step 4
0	1	1	0	1	0	0	1	0	0	0	0	0	0	1	step 5
0	0	0	0	1	1	0	1	1	0	0	0	1	1	1	step 6
0	0	1	0	1	1	0	0	1	0	1	1	1	1	1	step 1
0	0	1	0	1	1	0	0	1	1	0	0	1	1	1	step 2
0	0	1	0	1	1	1	0	0	0	0	0	1	1	1	step 3
0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	step 4
0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	step 5
0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	step 6
0	0	0	0	1	1	0	1	1	0	0	0	1	1	1	

Figure 2. The largest match in (0,0) and (6,6) is computed at step 5.

A one-dimensional match of width 6 is found at step 1. Then, at step 2 a better match is obtained which is 2×4. At step 3 and step 4 the current match is still 2×4 since the longest match on row 3 and 9 has width 2. At step 5, another match of width 2 provides a better rectangle match which is 5×2. At step 6, the procedure stops since the longest match has width 1 and the rectangle match can cover at most 7 rows. It follows that 5×2 is the greedy match since a rectangle of width 1 cannot have a larger area. Obviously, this procedure can be used for computing the largest monochromatic rectangle in a given position (i, j) as well.

As mentioned in the introduction, the procedure for computing the largest rectangle match takes $\mathrm{O}(M \log M)$ time, where M is the size of the match. The positions covered by matches are skipped in the linear scan of the image and the sequential time to compress an image of size n by rectangle matching is $\Omega(n \log M)$. The analysis of the running time of this algorithm involve a so called *waste factor*, defined as the average number of matches covering the same pixel. In [7], it is conjectured that the waste factor is less than 2 on realistic image data. Therefore, the square greedy matching heuristic takes linear time while the rectangle greedy matching heuristic takes $O(n \log M)$ time. On the other hand, the decoding algorithms are both linear.

3 A Massively Parallel Block Matching Algorithm

Coding and decoding algorithms are shown in [3] on the PRAM EREW, mesh of trees, pyramidal, and multigrid architectures, requiring $O(\log M \log n)$ time and $O(n/\log n)$ processors. The pyramid and multigrid implementations need some realistic assumptions. Under the same realistic assumptions, we show in the next section how to implement such encoder/decoder with the same parallel complexity on a full binary tree architecture. In this section, we present the PRAM EREW encoder/decoder. These algorithms can be implemented in $O(\alpha \log M)$ time with $O(n/\alpha)$ processors for any integer square value $\alpha \in \Omega(\log n)$.

To achieve sublinear time we partition an $m \times m'$ image I in $x \times y$ rectangular areas, where x and y are $\Theta(\alpha^{1/2})$. In parallel for each area, one processor applies the sequential parsing algorithm so that in $O(\alpha \log M)$ time each area will be parsed in rectangles, some of which are monochromatic. Before encoding we wish to compute larger monochromatic rectangles.

3.1 Computing the Monochromatic Rectangles

We compute larger monochromatic rectangles by merging adjacent monochromatic areas without considering those monochromatic rectangles properly contained in some area. Such limitation has no relevant effect on the compression ratio.

We denote with $A_{i,j}$ for $1 \leq i \leq \lceil m/x \rceil$ and $1 \leq j \leq \lceil m'/y \rceil$ the areas into which the image is partitioned. In parallel for $1 \leq i \leq \lceil m/x \rceil$, if i is odd, a processor merges areas $A_{2i-1,j}$ and $A_{2i,j}$ provided they are monochromatic and have the same color. The same is done horizontally for $A_{i,2j-1}$ and $A_{i,2j}$. At the k-th step, if areas $A_{(i-1)2^{k-1}+1,j}$, $A_{(i-1)2^{k-1}+2,j}, \ldots, A_{i2^{k-1},j}$, with i odd, were merged, then they will merge with areas $A_{i2^{k-1}+1,j}$, $A_{i2^{k-1}+2,j}, \ldots, A_{(i+1)2^{k-1},j}$, if they are monochromatic with the same color. The same is done horizontally for $A_{i,(j-1)2^{k-1}+1}$, $A_{i,(j-1)2^{k-1}+2}, \ldots, A_{i,j2^{k-1}}$, with j odd, and $A_{i,j2^{k-1}+1}$, $A_{i,j2^{k-1}+2}, \ldots, A_{i,(j+1)2^{k-1}}$. After $O(\log M)$ steps, the procedure is completed and each step takes $O(\alpha)$ time and $O(n/\alpha)$ processors since there is one processor for each area. Therefore, the image parsing phase is realized in $O(\alpha \log M)$ time with $O(n/\alpha)$ processors on an exclusive read, exclusive write shared memory machine.

3.2 The Parallel Encoder

We derive the sequence of pointers from the image parsing computed above. In $O(\alpha)$ time with $O(n/\alpha)$ processors we can identify every upper left corner of a match (proper, monochromatic, or raw) by assigning a different segment of length $\Theta(\alpha)$ on a row to each processor. Each processor detects the upper left corners on its segment. Then, by parallel prefix we obtain a sequence of pointers decodable by the decompressor paired with the sequential heuristic. However, the decoding of such sequence seems hard to parallelize. In order to design a parallel decoder, it is more suitable to produce the sequence of pointers by a raster scan of each of the areas into which the image was originally partitioned, where the areas are ordered by a raster scan themselves. Again we can easily derive the sequence of pointers in O(α) time with O(n/α) processors by detecting in each area every upper left corner of a match and producing the sequence of pointers by parallel prefix.

As mentioned in the introduction, the encoding scheme for the pointers uses a flag field indicating whether there is a monochromatic rectangle (0 for the white ones and 10 for the black ones), a proper match (110), or raw data (111). For the feasibility of the parallel decoder, we want to indicate the end of the encoding of the sequence of matches with the upper left corner in a specific area. Therefore, we change the encoding scheme by associating the flag field 1110 to the raw match so that we can indicate with 1111 the end of the sequence of pointers corresponding to a given area. Moreover, since some areas could be entirely covered by a monochromatic match 1111 is followed by the index associated with the next area by the raster scan. The pointer of a monochromatic match has fields for the width and the length while the pointer of a proper match also has fields for the coordinates of the left upper corner of the copy in the window. In order to save bits, the value stored in any of these fields is the binary value of the field plus 1 (so, we employ the zero value). Also, the range for these values is determined by α but for the width and length of monochromatic matches sharing the upper left corner with one of the areas $A_{i,j}$ (in this case, the range is determined by the width and length of the image). This coding technique is more redundant than others previously designed, but its compression effectiveness is still better than the one of the square greedy matching technique.

3.3 The Parallel Decoder

The parallel decoder has three phases. Observe that at each position of the binary string encoding the image, we read a substring of bits that is either 1111 (recall that the k bits following 1111 provide the area index, where k is the number of bits used to encode it) or can be interpreted as a flag field of a pointer. Then, in the first phase we reduce the binary string to a doubly-linked structure and apply the Euler tour technique in order to identify for each area the corresponding pointers. The reduction works as follows: link each position p of the string to the position next to the end of the substring starting in position p that either is equal to 1111 followed by k bits or can be interpreted as a pointer. For those suffixes of the string which can be interpreted as pointers, their first positions are linked to a special node denoting the end of the coding. For those suffixes of the string which cannot be interpreted as pointers, their first positions are not linked to anything. The linked structure is a forest with one tree rooted in the special node denoting the end of the coding and the other trees rooted in the first position of a suffix of the encoding string not interpretable as a pointer. The first position of the binary string is a leaf of the tree rooted in the special node. The sequence of pointers encoding the image is given by the path from the first position to the root. In order to compute such path we need the children to be doubly-linked to the parent. Then, we need to reserve space for each node to store the links to the children. Each node has at most five children since there are only four different pointer sizes (white, black, raw, or proper match). So, for each position p of the binary sequence we set aside five locations $[p, 1], \ldots, [p, 5]$, initially set to zero. When a link is added from position p' to p, depending on whether the substring starting in position p' is 1111 or can be interpreted as a pointer to a raw, white, black or proper match, the value p' is overwritten on location $[p, 1]$, $[p, 2]$, $[p, 3]$ $[p, 4]$ or $[p, 5]$, respectively. The linking for the substrings starting with 1111 is done first, since only afterwards we know exactly which substrings can be interpreted as pointers (recall that encoding the width and length of a monochromatic match sharing the left upper corner with one of the areas $A_{i,j}$ depends on the width and length of the whole image). Then, by means of the well-known Euler technique [8] we can linearize the linked structure and apply list ranking to obtain the path from the first position of the sequence to the root of its tree in $O(\alpha)$ time with $O(n/\alpha)$ processors on an exclusive read, exclusive write shared memory machine [1], [4], since the row image size is greater than the size of the encoding binary string. Then, still in $O(\alpha)$ time with $O(n/\alpha)$ processors we can identify the positions on the path corresponding to 1111.

In the second phase of the parallel decoder a different processor decodes the sequence of pointers corresponding to a different area. As far as the pointers to monochromatic matches are considered, each processor decompresses either a match contained in an area or the portion of the match corresponding to the left upper area. Therefore, after the second phase an area might not be decompressed. Obviously, the second phase requires $O(\alpha)$ time and $O(n/\alpha)$ processors.

The third phase completes the decoding. In the previous subsection, we denoted with $A_{i,j}$ for $1 \leq i \leq \lceil m/x \rceil$ and $1 \leq j \leq \lceil m'/y \rceil$ the areas into which the image was partitioned by the encoder. At the first step of the third phase, one processor for each area $A_{2i-1,j}$ decodes $A_{2i,j}$, if $A_{2i-1,j}$ is the upper left portion of a monochromatic match and the length field of the corresponding pointer informs that $A_{2i,j}$ is part of the match. The same is done horizontally for $A_{i,2j-1}$ and $A_{i,2j}$ (using the width field of its pointer) if it is known already by the decoder that $A_{i,2j-1}$ is part of a monochromatic match. Similarly at the k-th step, one processor for each of the areas $A_{(i-1)2^{k-1}+1,j}$, $A_{(i-1)2^{k-1}+2,j}, \ldots, A_{i2^{k-1},j}$, with i odd, decodes the areas $A_{i2^{k-1}+1,j}$, $A_{i2^{k-1}+2,j}, \ldots, A_{(i+1)2^{k-1},j}$, respectively. The same is done horizontally for $A_{i,(j-1)2^{k-1}+1}$, $A_{i,(j-1)2^{k-1}+2}, \ldots, A_{i,j2^{k-1}}$, with j odd, and $A_{i,j2^{k-1}+1}$, $A_{i,j2^{k-1}+2}, \ldots, A_{i,(j+1)2^{k-1}}$. After $O(\log M)$ steps the image is entirely decompressed. Each step takes $O(\alpha)$ time and $O(n/\alpha)$ processors since there is one processor for each area. Therefore, the decoder is realized in $O(\alpha \log M)$ time with $O(n/\alpha)$ processors.

4 The Tree Architecture Implementations

We implement the parallel BLOCK MATCHING encoder and decoder on a full binary tree architecture. We extend the $m \times m'$ image I with dummy rows and columns so that I is partitioned into $x \times y$ areas $A_{i,j}$ for $1 \leq i, j \leq 2^h$, where x and y are $\Theta(\alpha^{1/2})$, $n = mm'$ is the size of the image and $h = \min\{k : 2^k \geq \max\{m/x, m'/y\}\}$. We store these areas into the leaves of the tree according to a one-dimensional layout which allows an easy way of merging the monochromatic ones at the upper levels. Let $\mu = 2^h$. The number of leaves is 2^{2h} and the leaves are numbered from 1 to 2^{2h} from left to right. It follows that the tree has height $2h$. Therefore, the height of the tree is $O(\log n)$ and the number of nodes is $O(n/\alpha)$. Such layout is described by the recursive procedure of figure 3. The initial value for i, j and ℓ is 1 and ℓ is a global variable.

In parallel for each area, each leaf processor applies the sequential parsing algorithm so that in $O(\alpha \log M)$ time each area is parsed into rectangles, some of which are monochromatic. Again, before encoding we wish to compute larger monochromatic rectangles.

STORE$(I,\, \mu,\, i,\, j)$

if $\mu > 1$

STORE$(I,\, \mu/2,\, i,\, j)$

STORE$(I,\, \mu/2,\, i + \mu/2,\, j)$

STORE$(I,\, \mu/2,\, i,\, j + \mu/2)$

STORE$(I,\, \mu/2,\, i + \mu/2,\, j + \mu/2)$

else store $A_{i,j}$ into leaf ℓ; $\ell = \ell + 1$

Figure 3. Storing the image into the leaves of the tree.

4.1 Computing the Monochromatic Rectangles

After the compression heuristic has been executed on each area, we have to show how the procedure to compute larger monochromatic rectangles can be implemented on a full binary tree architecture with the same number of processors without slowing it down. This is possible by making some realistic assumptions. Let ℓ_R and w_R be the length and the width of a monochromatic match R, respectively. We define $s_R = \max\{\ell_R, w_R\}$. We make a first assumption that the number of monochromatic matches R with $s_R \geq 2^k\lceil\log^{1/2} n\rceil$ is $\mathrm{O}(n/(2^{2k}\log n))$ for $1 \leq k \leq h-1$. While computing larger monochromatic rectangles, we store in each leaf the partial results on the monochromatic rectangles covering the corresponding area (it is enough to store for each rectangle the indices of the areas at the upper left and lower right corners). If i is odd, it follows from the procedure of figure 3 that the processors storing areas $A_{2i-1,j}$ and $A_{2i,j}$ are siblings. Such processors merge $A_{2i-1,j}$ and $A_{2i,j}$ provided they are monochromatic and have the same color by broadcasting the information through their parent. It also follows from such procedure that the same can be done horizontally for $A_{i,2j-1}$ and $A_{i,2j}$ by broadcasting the information through the processors at level $2h-2$. At the k-th step, if areas $A_{(i-1)2^{k-1}+1,j}$, $A_{(i-1)2^{k-1}+2,j}, \ldots, A_{i2^{k-1},j}$, with i odd, were merged for $w_1 \leq j \leq w_2$, the processor storing area $A_{(i-1)2^{k-1}+1,w_1}$ will broadcast to the processors storing the areas $A_{i2^{k-1}+1,j}$, $A_{i2^{k-1}+2,j}, \ldots, A_{(i+1)2^{k-1},j}$ to merge with the above areas for $w_1 \leq j \leq w_2$, if they are monochromatic with the same color. The broadcasting will involve processors up to level $2h-2k+1$. The same is done horizontally, that is, if $A_{i,(j-1)2^{k-1}+1}$, $A_{i,(j-1)2^{k-1}+2}, \ldots, A_{i,j2^{k-1}}$, with j odd, were merged for $\ell_1 \leq i \leq \ell_2$, the processor storing area $A_{\ell_1,(j-1)2^{k-1}+1}$ will broadcast to the processors storing the areas $A_{i,j2^{k-1}+1}$, $A_{i,j2^{k-1}+2}, \ldots, A_{i,(j+1)2^{k-1}}$ to merge with the above areas for $\ell_1 \leq i \leq \ell_2$, if they are monochromatic with the same color. The broadcasting will involve processors up to level $2h-2k$.

After $\mathrm{O}(\log M)$ steps, the procedure is completed. If the waste factor is less than 2, as conjectured in [7], we can make a second assumption that each pixel is covered by a constant small number of monochromatic matches. It follows from this second assumption that the information about the monochromatic matches is distributed among the processors at the same level in a way very close to uniform. Then, it follows from the first assumption that the amount of information each processor of the tree must broadcast is constant. Therefore, each step takes $O(\alpha)$ time and the image parsing phase is realized with $O(\alpha \log M)$ time and $O(n/\alpha)$ processors.

4.2 The Parallel Encoder

The sequence of pointers is trivially produced by the processors which are leaves of the tree. For the monochromatic rectangles, the pointer is written in the leaf storing the area at the upper left corner. Differently from the shared memory machine decoder, the order of the pointers is the one of the leaves. Since some areas could be entirely covered by a monochromatic match, the subsequence of pointers corresponding to a given area is ended with 1111 followed by the index of the leaf storing the next area to decode. We define a variable for each leaf. This variable is set to 1 if the leaf stores at least a pointer, 0 otherwise. Then, the index of the next area to decode is computed for each leaf by parallel suffix computation. Moreover, with the possibility of a parallel output the sequence can be put together by parallel prefix. This is, obviously, realized in $O(\alpha)$ time with $O(n/\alpha)$ processors.

4.3 The Parallel Decoder

We store each encoding of an area in a leaf of the tree. The storing procedure reads the encoding binary string from left to right. When it finds the substring 1111, this denotes the end of the encoding of an area and the next k bits provide the leaf index of the next area where k is the number of bits used to encode it. At this point, each processor at the leaf level completes the second phase of the decoder described in subsection 3.3. Then, the third and last phase of the shared memory machine decoder has the same parallel computational complexity on the tree architecture with the same realistic assumptions we made for the coding phase. In conclusion, the decoder takes $O(\alpha \log M)$ time on a full binary tree architecture with $O(n/\alpha)$ processors.

5 Conclusions

Parallel coding and decoding algorithms for lossless image compression by block matching were shown requiring $O(\log M \log n)$ time and $O(n/\log n)$ processors on a full binary tree architecture, where n is the size of the image and M is the size of the match. The parallel coding algorithm is work-optimal since the sequential time required by the coding is $\Omega(n \log M)$. On the other hand, the parallel decoding algorithm is not work-optimal since the sequential decompression time is linear. These implementations have the same performance of the shared memory machine algorithms under some realistic assumptions and if we do not consider the input/output process. In [3], with such assumptions these algorithms had been realized on a multigrid which is the simplest architecture among the ones with small diameter and large bisection width. Such realistic assumptions are that each pixel is covered by a small constant number of monochromatic rectangles of the image parsing and that the increasing of the dimensions of the monochromatic rectangles is proportional to the decreasing of the number of monochromatic rectangles with such dimensions. We have shown in this paper that with the assumptions made for the multigrid we can relax on the requirement of an architecture with large bisection width and design compression and decompression on a two-dimensional architecture such as a full binary tree. The communication cost might be low enough to realize efficient implementations on one of the available parallel machines since the algorithms are scalable.

References

1. R. P. Brent: *The parallel evaluation of general arithmetic expressions.* Journal of the ACM, 21 1974, pp. 201–206.
2. R. P. Brent: *A linear algorithm for data compression.* Australian Computer Journal, 19 1987, pp. 64–68.
3. L. Cinque and S. DeAgostino: *Lossless image compression by block matching on practical massively parallel architectures*, in Proceedings Prague Stringology Conference, 2008, pp. 26–34.
4. R. Cole and U. Vishkin: *Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time.* SIAM Journal on Computing, 17 1988, pp. 148–152.
5. A. Lempel and J. Ziv: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23 1977, pp. 337–343.
6. J. A. Storer: *Lossless image compression using generalized LZ1-type methods*, in Proceedings IEEE Data Compression Conference, 1996, pp. 290–299.
7. J. A. Storer and H. Helfgott: *Lossless image compression by block matching.* The Computer Journal, 40 1997, pp. 137–145.
8. R. E. Tarjan and U. Vishkin: *An efficient parallel biconnectivity algorithm.* SIAM Journal on Computing, 14 1985, pp. 862–874.
9. J. R. Waterworth: *Data compression system.* US Patent 4 701 745, 1987.
10. D. A. Whiting, G. A. George, and G. E. Ivey: *Data compression apparatus and method.* US Patent 5016009, 1991.

Taxonomies of Regular Tree Algorithms

Loek Cleophas[1] and Kees Hemerik[2]

[1] FASTAR/Espresso Research Group, Department of Computer Science,
University of Pretoria, 0002 Pretoria, Republic of South Africa,
http://www.fastar.org

[2] Software Engineering & Technology Group, Department of Mathematics and Computer Science,
Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands,
http://www.win.tue.nl/set

loek@loekcleophas.com, c.hemerik@tue.nl

Abstract. Algorithms for acceptance, pattern matching and parsing of regular trees and the tree automata used in these algorithms have many applications, including instruction selection in compilers, implementation of term rewriting systems, and model checking. Many such tree algorithms and constructions for such tree automata appear in the literature, but some deficiencies existed, including: inaccessibility of theory and algorithms; difficulty of comparing algorithms due to variations in presentation style and level of formality; and lack of reference to the theory in many publications. An algorithm taxonomy is an effective means of bringing order to such a field. We report on two taxonomies of regular tree algorithms that we have constructed to deal with the deficiencies. The complete work has been presented in the PhD thesis of the first author.

Keywords: tree acceptance, tree pattern matching, tree automata, algorithm taxonomies

1 Introduction

We consider the field of *regular tree languages* for *ordered, ranked trees*.[1] This field has a rich theory, with many generalizations from the field of regular string languages, and many relations between the two [9,10,12,14]. Parts of the theory have broad applicability in areas as diverse as instruction selection in compilers, implementation of term rewriting systems, and model checking.

We focus on algorithmic solutions to three related problems in the field, i.e. *tree acceptance, tree pattern matching* and *tree parsing*. Many such algorithms appear in the literature, but unfortunately some deficiencies exist, including:

1. Inaccessibility of the theory and algorithms, as they are scattered over the literature and few or no (algorithm oriented) overview publications exist.
2. Difficulty of comparing the algorithms due to differences in presentation style and level of formality.
3. Lack of reference to the theory and of correctness arguments in publications of practical algorithms.

A *taxonomy*—in a technical sense made more precise below—is an effective means of bringing order to such a subject. A taxonomy is a systematic classification of problems and solutions in a particular (algorithmic) problem domain. We have constructed two such taxonomies, one for tree acceptance algorithms and one for tree pattern matching ones.

[1] An example of such a language as defined by a regular tree grammar can be found in Section 4.

Loek Cleophas, Kees Hemerik: Taxonomies of Regular Tree Algorithms, pp. 146–159.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

A few more practical deficiencies existed as well: no large and coherent collection of implementations of the algorithms existed; and for practical applications it was difficult to choose between algorithms. We therefore designed, implemented, and benchmarked a highly coherent *toolkit* of most of these algorithms as well. Taxonomies also form a good starting point for the construction of such algorithmic toolkits.

In the past, taxonomies and/or toolkits of this kind have been constructed for e.g. sorting [3,11], garbage collection [17], string pattern matching, finite automata construction and minimization [21,22].

In this paper we focus on one of our taxonomies, and comment only briefly on the other one and on the toolkit. The complete work has been presented in the PhD thesis of the first author [9]. For more details we refer to this thesis and to recent shorter publications [5,6,18].

Section 2 gives a brief introduction to taxonomies as we consider them. In Section 3 we outline the structure of our taxonomy of algorithms for tree acceptance and briefly compare it to the one for tree pattern matching. Afterwards we focus on the one for tree acceptance. Definitions of tree and tree grammar related notions are given in Section 4. The main branches of the taxonomy for tree acceptance are discussed in Sections 5–8. Section 9 briefly discusses some other parts of the work, namely the toolkit and accompanying graphical user interface and the benchmarking experiments performed with them. We end the paper with some concluding remarks in Section 10.

2 Taxonomies

In our technical sense a taxonomy is a means of ordering a set of algorithmic problems and their solutions. Each node of the taxonomy graph is a pair consisting of (a specification of) a problem and an algorithm solving the problem. For each (problem, algorithm) pair the set of essential details is determined. In general, there are two kinds of details: *problem details*, which restrict the problem, and *algorithm details*, which restrict the algorithm (e.g. by making it more deterministic). The root of the taxonomy graph contains a high-level algorithm of which the correctness is easily shown. A branch in the graph corresponds to addition of a detail in a correctness preserving way. Hence, the correctness of each algorithm follows from the details on its root path and the correctness of the root.

Construction of an algorithm taxonomy is a bottom-up process. A literature survey of the problem domain is performed to gather algorithms. The algorithms are rephrased in a common presentation style and analyzed to determine their essential details. When two algorithms differ only in a few details, abstracting over those details yields a common ancestor. Repeating this abstraction process leads to the main structure of a taxonomy graph. Considering new combinations of details may lead to discovery of new algorithms. Eventually the taxonomy may be presented in a top-down manner.

Several taxonomies of this kind appear in the literature. Broy and Darlington each constructed one of sorting algorithms [3,11]. Jonkers [17] constructed a taxonomy of garbage collection algorithms and also developed a general theory about algorithm taxonomies. Watson [21] applied the method to construct taxonomies for string pattern algorithms and for the construction and minimization of finite automata. Both in subject and in style our work is closest to Watson's.

3 Overview of the Taxonomies of Regular Tree Algorithms

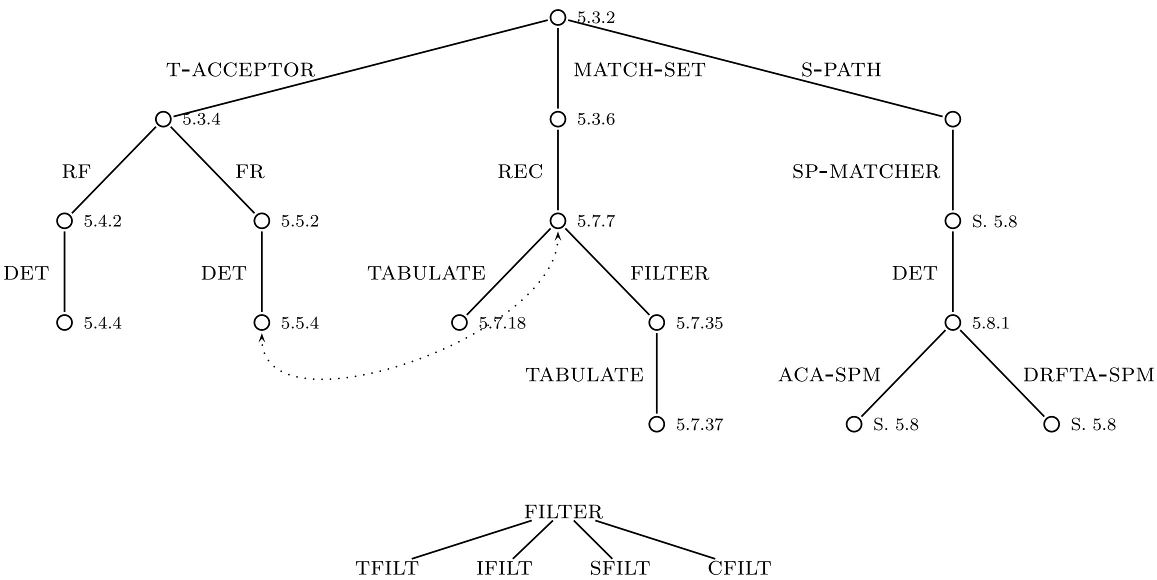


Figure 1. Tree acceptance taxonomy. Each node is labeled with its corresponding algorithm or section (S.) number in [9]. Constructions for tree acceptors used in algorithms of branch (T-ACCEPTOR) are not depicted. The bottom part of the figure shows the four possible filters that can be used for detail FILTER.

The tree acceptance (aka language membership, membership) problem as we consider it is the following: Given a regular tree grammar and a subject tree, determine whether the tree is an element of the language defined by the grammar. Figure 1 depicts the taxonomy of algorithms we have constructed for this problem. The edge labels correspond to details, explained in Table 1.

In the taxonomy graph, three main subgraphs can be distinguished. The first subgraph (detail T-ACCEPTOR and below) contains all algorithms based on the correspondence between regular tree grammars and finite tree automata. For every regular tree grammar an undirected finite tree automaton can be constructed, which accepts exactly the trees generated by the grammar. By adding more detail, viz. a direction (detail FR: frontier-to-root or detail RF: root-to-frontier) or determinacy (detail DET) more specific constructions are obtained. The acceptance algorithms from this part of the taxonomy are described in more detail in Section 5, while the tree automata constructions used in them are discussed in Section 6.

The second subgraph (detail MATCH-SET and below) contains all algorithms based on suitably chosen generalizations of the relation $S \overset{*}{\Rightarrow} t$ (where $\overset{*}{\Rightarrow}$ indicates derivation in zero or more steps (see Section 4), S is the start symbol of the grammar and t is the subject tree). For each subtree of t, they compute a set of *items* from which t may be derived, a so-called *match set*. Tree t is accepted if and only if its match set contains S. The algorithms in this subgraph of the taxonomy differ in the item set used and in how the match sets are computed. This part of the taxonomy is described in more detail in Section 7.

T-ACCEPTOR	Use a tree automaton accepting the language of a regular tree grammar to solve the language membership problem.
RF	Consider the transition relations of the tree automaton used in an algorithm to be directed in a root-to-frontier or top-down direction.
FR	Consider the transition relations of the tree automaton used in an algorithm to be directed in a frontier-to-root or bottom-up direction.
DET	Use a deterministic version of an automaton.
MATCH-SET	Use an item set and a match set function to solve the tree acceptance/language membership problem. Such an item set is derived from the productions of the regular tree grammar and the match set function indicates from which of these items a tree is derivable.
REC	Compute match set values recursively, i.e. compute the match set values for a tree from the match set values computed for its direct subtrees.
FILTER	Use a filtering function in the computation of match set function values. Before computing the match set for a tree, such a filtering function is applied to the match sets of its direct subtrees.
TABULATE	Use a tabulated version of the match set function (and of the filter functions, if filtering is used), in which a bijection is used to identify match sets by integers.
S-PATH	Uniquely decompose production right hand sides into stringpaths. Based on matching stringpaths, production right hand sides and nonterminals deriving the subject tree can be uniquely determined and tree acceptance can thus be solved.
SP-MATCHER	Use an automaton as a pattern matcher for a set of stringpaths in a root-to-frontier or top-down subject tree traversal.
ACA-SPM	Use an (optimal) Aho-Corasick automaton as a stringpath matcher and define transition and output functions in terms of that automaton.
DRFTA-SPM	Use a deterministic root-to-frontier tree automaton as a stringpath matcher and define transition and output functions in terms of that automaton.

Table 1.

The third subgraph (detail SP-MATCHER and below) contains algorithms based on the decomposition of items into so-called *stringpaths* and subsequent use of string matching techniques. Based on stringpath matches found, matches of items and hence essentially the match sets mentioned previously are computed for each subtree of t. Section 8 gives a brief explanation of this taxonomy part.

As our focus in this paper is on the tree acceptance taxonomy and the algorithms and constructions included in it, we do not formally define the tree pattern matching problem. Figure 2 shows the taxonomy of tree pattern matching algorithms. Although we do not explicitly give the meaning of the details used, it should be clear that the taxonomies for tree acceptance and tree pattern matching have much in common. Techniques such as the subset construction, match sets, and stringpaths are used in both. This is not surprising: the two problems are closely related, and some kinds of tree acceptors can be turned into tree pattern matchers (or vice versa) with little effort. The same phenomenon can be observed in acceptors and pattern matchers for string languages.

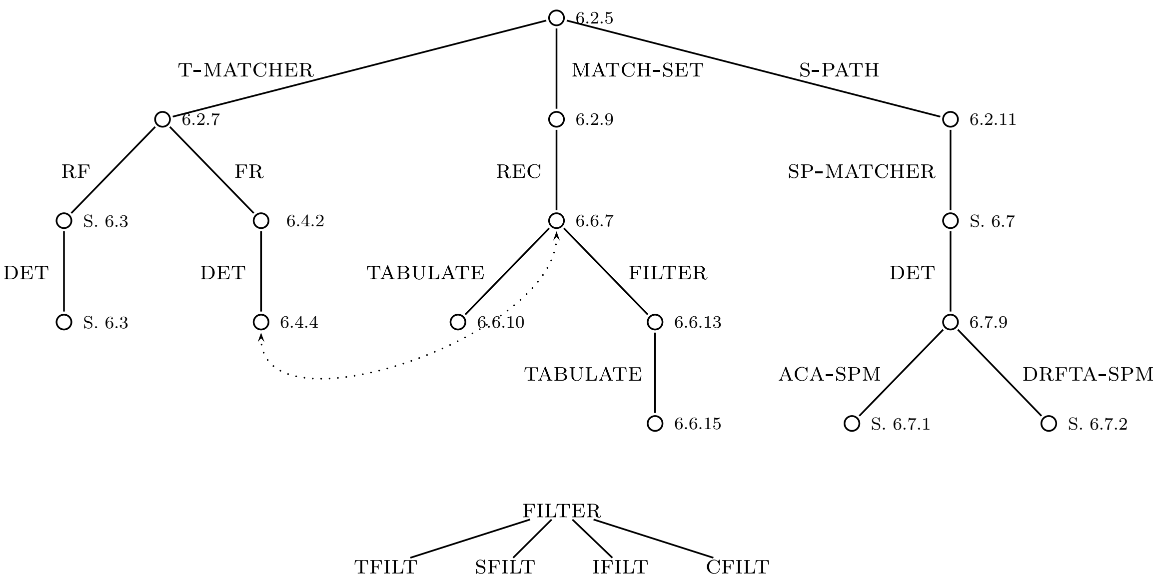


Figure 2. Tree pattern matching taxonomy. Each node is labeled with its corresponding algorithm or section (S.) number in [9]. Constructions for tree pattern matchers used in algorithms of branch (T-MATCHER) are not depicted. The bottom part of the figure shows the four possible filters that can be used for detail FILTER.

4 Notation and definitions

We use $\mathbb{B}$ and $\mathbb{N}$ to denote the booleans and the natural numbers. We use notation $\langle \mathbf{Set}\ a : R(a) : E(a) \rangle$ for the set of expressions $E(a)$ for which a satisfies range predicate $R(a)$.

Many of the other notations and definitions we use are related to regular tree language theory and to a large extent generalizations of familiar ones from regular string language theory. To aid readers unfamiliar with this theory, we briefly introduce the concepts needed in the rest of this paper. Readers may want to consult e.g. [9,10,12,14] for more detail.

Let Σ be an *alphabet*, and $r \in \Sigma \mapsto \mathbb{N}$. Pair (Σ, r) is a *ranked alphabet*, r is a *ranking function*, and for all $a \in \Sigma$, $r(a)$ is called the *rank* or *arity* of a. (The ranking function indicates the number of child nodes a node labeled by a particular symbol will have.) We use Σ_n for $0 \leq n$ to indicate the subset of Σ of symbols with arity n.

Given a ranked alphabet (Σ, r), the set of *ordered, ranked trees* over this alphabet, set $Tr(\Sigma, r)$, is the smallest set satisyfing

1. $\Sigma_0 \subseteq Tr(\Sigma, r)$, and
2. $a(t_1, \ldots, t_n) \in Tr(\Sigma, r)$ for all $t_1, \ldots, t_n \in Tr(\Sigma, r), a \in \Sigma$ such that $r(a) = n \neq 0$.

As a running example, we assume (Σ, r) to be $\{(a, 2), (b, 1), (c, 0), (d, 0)\}$, i.e. consisting of symbols a, b, c and d with rank $2, 1, 0$ and 0. Trees in $Tr(\Sigma, r)$ include for example c, $a(b(c), d)$ and $a(a(b(c), c), d)$.

A *regular tree grammar* (RTG) G is a 5-tuple $(N, \Sigma, r, Prods, S)$ where N and Σ are disjoint alphabets (the *nonterminals* and *terminals*), $(N \cup \Sigma, r)$ is a ranked alphabet in which all nonterminals have rank 0, $Prods \subseteq N \times Tr(N \cup \Sigma, r)$ is the finite set of *productions*, and $S \in N$ (the *start symbol*). We use LHS and RHS for left hand side and right hand side (of a production), and use $RHS(Prods)$ for the set of production RHSs.

Given a grammar G, we use $\Rightarrow$ for a derivation step, in which a nonterminal is replaced by a corresponding production RHS. The reflexive and transitive closure of $\Rightarrow$ is denoted by $\stackrel{*}{\Rightarrow}$. The subset of $Tr(\Sigma, r)$ derivable from S is denoted $\mathcal{L}(G)$. For technical reasons, we introduce the *augmented* grammar G' for a grammar G, defined by $G' = (N \cup \{S'\}, \Sigma, r \cup \{(S', 0)\}, Prods \cup \{S' \mapsto S\}, S')$ where S' is a fresh symbol.

In this paper, we assume an example grammar $G_1 = (N, \Sigma, r, Prods, S)$ with $N = \{S, B\}$, r and Σ as before, and with $Prods$ defined as $\{S \mapsto a(B, d),\ \ S \mapsto a(b(c), B),\ \ S \mapsto c,\ \ B \mapsto b(B),\ \ B \mapsto S,\ \ B \mapsto d\}$. We assume G to be the corresponding augmented grammar.

5 Algorithms based on Tree Automata

The first subgraph of the taxonomy deals with algorithms for tree acceptance that are based on correspondences between regular tree grammars and finite tree automata. The theoretical basis for this correspondence is well-known and generalizes a similar correspondence between regular string grammars and finite string automata. To ease understanding we briefly outline how the generalization works.

It is well known that the theory of regular tree languages generalizes that of regular string languages [9,10,12,14]. This is not surprising: any string $a_0 \cdots a_{n-1}$ can be seen as a special kind of regular tree, viz. one consisting of n unary nodes, each labeled with a symbol a_i of rank 1, closed by a nullary node marked with a symbol of rank 0. Notions from finite automata for strings can be generalized to the tree case as well, although this requires a particular view of such automata. Suppose that a particular string automaton goes through a state sequence $q_0, \ldots, q_n$ when presented the string $a_0 \cdots a_{n-1}$. This means that for each $i : 0 \leq i < n$ the pair of states (q_i, q_{i+1}) must be in the transition relation of symbol a_i. We can summarize the transition sequence by the following alternation of states and symbols: $q_0 a_0 \cdots a_{n-1} q_n$. In other words, the positions in the string have been consistently annotated with states $q_0, \ldots, q_n$. The language accepted by the automaton can be defined as the set of strings that can be consistently annotated in this way, such that q_0 and q_n are initial and final states.

This view can easily be generalized to ordered, ranked trees: each node is annotated with a state, and for each node labeled with a symbol a of rank n, the state q_0 assigned to that node and the states $q_1, \ldots, q_n$ of the n direct subnodes should be such that the tuple $(q_0, (q_1, \ldots, q_n))$ is in the transition relation of symbol a. Note that this simplifies to $(q_0, ())$ for symbols of rank 0. (Hence, taking a frontier-to-root or bottom-up view on tree automata, no equivalent for a string automaton's initial states is needed; no equivalent for a string automaton's final states is needed when taking a root-to-frontier or top-down view.) A tree is *accepted* by a finite tree automaton if and only if it can be consistently annotated such that the state assigned to the root is a so-called *root accepting* state. This motivates the following definition:

Definition 1. *A* (finite) tree automaton (TA) M *is a 5-tuple* $(Q, \Sigma, r, R, Q_{ra})$ *such that* Q *is a finite set, the* state set*;* (Σ, r) *is a ranked alphabet;* $R = \{R_a | a \in \Sigma\} \cup R_\varepsilon$ *is the set of transition relations (where* $R_\varepsilon \subseteq Q \times Q$ *and* $R_a \subseteq Q \times Q^n$*, for all* $a \in \Sigma$ *with* $r(a) = n$*); and* $Q_{ra} \subseteq Q$ *is the set of* root accepting states*.*

Many important theorems carry over from regular string grammars and automata to the tree case as well. In particular:

Theorem 2. *For every regular tree grammar G there exists a tree automaton M such that $\mathcal{L}(G) = \mathcal{L}(M)$.*

This theorem justifies the following algorithm as a solution for tree acceptance:

Algorithm 3 (T-ACCEPTOR)

$[\![$ **const** $G = (N', \Sigma, r', Prods', S')$: augmented RTG;
    $t : Tr(\Sigma, r)$;
  var $b : \mathbb{B}$
$|$ **let** $M = (Q, \Sigma, r, R, Q_{ra})$ be a TA such that $\mathcal{L}(M) = \mathcal{L}(G)$;
  $b :\ = t \in \mathcal{L}(M)$
  $\{\ b \equiv t \in \mathcal{L}(G)\ \}$
$]\!]$

This abstract and rather trivial algorithm forms the root of the part of the taxonomy graph containing all algorithms based on tree automata. Note that it does not specify *how* $t \in \mathcal{L}(M)$ is determined. It could consider all state assignments to t respecting the transition relations R, and determine whether an accepting one exists.

To obtain more specific and more practical algorithms, the automata and hence the state assignments can be considered as directed ones (detail FR: frontier-to-root aka bottom-up or detail RF: root-to-frontier aka top-down). This results in (the use of) an *ε-nondeterministic frontier-to-root* TA (εNFRTA) and *ε-nondeterministic root-to-frontier* TA (εNRFTA).

Restricting the directed automata to the case without ε-transitions, we obtain the ε-less TA and (ε-less) NRFTA and NFRTA. As with string automata, ε-transitions can be removed by a straightforward transformation. The use of the resulting automata slightly simplifies the acceptance algorithms.

5.1 FR: Frontier-to-Root Tree Acceptors

For (ε)NFRTAs, a recursive acceptance function $RSt \in Tr(\Sigma, r) \mapsto \mathcal{P}(Q)$ can be defined. This function yields the states assigned to a tree's root node based on those assigned to that node's child nodes. A subject tree t is then accepted if and only if at least one accepting state occurs in state set $RSt(t)$.

Restricting the directed R_a of the (ε-less) NFRTA to be single-valued functions, we obtain the deterministic DFRTA. A subset construction $\text{SUBSET}_{\text{FR}}$ can be given, similar to that for string automata, to obtain a DFRTA for an (ε)NFRTA. The use of a DFRTA leads to the straightforward Algorithm 4 given below.

Algorithm 4 (T-ACCEPTOR, FR, DET)

$[\![$ **const** $G = (N', \Sigma, r', Prods', S')$: augmented RTG;
    $t : Tr(\Sigma, r)$;
  var $b : \mathbb{B}$
| **let** $M = (Q, \Sigma, r, R, Q_{ra})$ be a DFRTA such that $\mathcal{L}(M) = \mathcal{L}(G)$;
  $b :\ = Traverse(t) \in Q_{ra}$
  $\{\ b \equiv t \in \mathcal{L}(G)\ \}$

  func $Traverse(st : Tr(\Sigma, r)) : Q =$
  $[\![$
  | **let** $a = st(\varepsilon)$;
    $\{\ st = a(st_1, \ldots, st_n)$ where $n = r(a)\ \}$
    $Traverse :\ = R_a(\,Traverse(st_1), \ldots, Traverse(st_n))$
  $]\!]\{$ Post: $\{Traverse\} = RSt(st)\ \}$
$]\!]$

5.2 RF: Root-to-Frontier Tree Acceptors

For root-to-frontier automata, we can define a root-to-frontier acceptance function $Accept \in Tr(\Sigma, r) \times Q \mapsto \mathbb{B}$ indicating whether an accepting computation starting from some state exists for a tree. In the resulting Algorithm (T-ACCEPTOR, RF) (not given here), the value of this function is computed by possibly many root-to-frontier subject tree traversals (starting from each of the root accepting states).

As with FRTAs, RFTAs can be restricted to ε-less ones and further to deterministic ones. Since DRFTAs are known to be less powerful than other TA kinds, algorithms using DRFTAs cannot solve the acceptance problem for each input grammar. We refer the reader to [9] for more information on algorithms using RFTAs to directly solve the tree acceptance problem.

In Section 8 we briefly discuss how DRFTAs can be used for so-called stringpath matching. Since there is a one-to-one correspondence between a tree and its set of stringpaths, DRFTAs can thus be used to solve the tree acceptance problem, albeit indirectly.

6 Construction of tree automata

Nowhere in Section 5 did we specify *how* the tree automata M, which are used in Algorithm (T-ACCEPTOR) and derived algorithms, are to be constructed. Such constructions can be considered separately, as we do in this section.

Algorithm (T-ACCEPTOR) and derived ones use TAs M such that $\mathcal{L}(M) = \mathcal{L}(G)$. Depending on the algorithm, the acceptor may need to be undirected or directed RF or FR, and directed ones may need to be nondeterministic or deterministic. The constructions differ in a number of aspects:

- Which *item set* is used to construct states: one containing *all* subtrees of production RHSs, or one just containing all nonterminals as well as the *proper* subtrees among RHSs,
- whether ε-*transitions* are present or not—the latter indicated by label REM-ε ,

– whether automata are *undirected*, *root-to-frontier* (aka top-down) or *frontier-to-root* (aka bottom-up), and
– whether ε-less directed automata are *deterministic* or not.

By combining choices for these aspects, twenty four constructions for tree acceptors can be obtained. Roughly half are treated in [9, Chapter 6], seeming most interesting because they occur in the literature or because they lead to ones that do.

For each construction in our taxonomy, the discussion in [9, Chapter 6] defines the state set, root accepting state set and transition relation are defined; and usually gives an example and a discussion of correctness and of related constructions and literature. Presenting all of the constructions in such a similar, uniform and precise way facilitates understanding and comparing the different constructions.

To further simplify understanding and comparison, the constructions are identified by sequences of detail labels. For example, the first construction, Construction (TGA-TA:ALL-SUB), is a basic construction for undirected TAs. Its state set corresponds to all subtrees of production RHSs, while its transitions encode the relations between (tuples of) such states, based on the relation between a tree and its direct subtrees and the relation between a production LHS and RHS.

We cannot present the constructions here in detail, but restrict ourselves to describing them briefly and showing how constructions from the literature are included. We emphasize that our taxonomy presents all of them together and relates all of them for the first time.

– The basic Construction (TGA-TA:ALL-SUB) described above does not explicitly appear in the literature, but its FR and RF versions appear in van Dinther's 1987 work [20].
– Applying REM-ε results in Construction (TGA-TA:ALL-SUB:REM-ε) for automata isomorphic to those constructed by Ferdinand et al. (1994) [13]. This detail makes states corresponding to certain full RHSs unreachable and therefor useless.
– To prevent such states from occurring, a state set containing only nonterminals and proper subtrees of RHSs can be used instead. Of the resulting Construction (TGA-TA:PROPER-N:REM-ε),
  • an undirected version appears in Ferdinand, Seidl and Wilhelm's 1994 paper [13] and later in Wilhelm & Maurer [23]. Somewhat surprisingly, the construction in its general form apparently did not occur in the literature before 1994.

  It is well known however that every RTG can easily be transformed into one with productions of the form $A \mapsto a(A_1, \ldots, A_n)$ only (by introducing fresh nonterminals and productions). For such RTGs,
  • an FR directed version already appeared in Gecseg and Steinby's [14, Lemma 3.4] in 1984.

  It is also straightforward to transform any RTG into one with productions of the form given above and of the form $A \mapsto B$ (i.e. additionally allowing unit productions). For such RTGs,
  • an FR directed version of Construction (TGA-TA:PROPER-N:REM-ε) already appears in Brainerd's 1960s work [2] and again in [20], and
  • an RF directed version appears in Comon et al. 's online work [10].
– Constructions (TGA-TA:ALL-SUB:REM-ε:RF:SUBSET$_{\text{RF}}$) and (TGA-TA:PROPER-N:-REM-ε:RF:SUBSET$_{\text{RF}}$), which are derived constructions resulting in DRFTAs, do not appear in the literature, probably due to the restricted power of such automata.

For a specific subclass of RTGs for which DRFTAs can be constructed, a variant resulting in tree *parsers* based on such DRFTAs is presented in [20].

- A construction for DFRTAs which uses all RHSs for state set construction—i.e. Construction (TGA-TA:ALL-SUB:REM-ε:FR:SUBSET$_{\text{FR}}$)—appears in [15]. The encompassed subset construction constructs the reachable subsets only, with an explicit sink state for the empty set. The presentation mostly disregards the automata view and uses the recursive match set view of Section 7. It was inspired by and gives a more formal version of the initial construction presented in Chase's 1987 paper [4].
- A construction for DFRTAs which uses only nonterminals and proper subtrees of RHSs—Construction (TGA-TA:PROPER-N:REM-ε:FR:SUBSET$_{\text{FR}}$)—appears in [13, Section 6] and in [23, Sections 11.6–11.7].

7 Algorithms based on Match Sets

In this section we consider the second subgraph of the taxonomy. Algorithms in this part solve the tree acceptance problem, i.e. $S \stackrel{*}{\Rightarrow} t$, by suitably chosen generalizations of relation $\stackrel{*}{\Rightarrow}$. First, from the tree grammar a set of *Items* is constructed, e.g. the set of subtrees of right hand sides of productions of the grammar. Then, for the subject tree t, a so-called match set $MS(t)$ is computed, the set of all $p \in Items$ for which $p \stackrel{*}{\Rightarrow} t$ holds. Tree t is accepted if and only if $S \in MS(t)$.

Algorithms in this part of the taxonomy differ in the set *Items* used and in how function *MS* is computed. The first algorithm, Algorithm (MATCH-SET), does not specify *how* to compute function *MS*.

Function *MS* can effectively be computed recursively over a subject tree, i.e. by a scheme of the form $MS(a(t_1,\ldots,t_n)) = \mathcal{F}(MS(t_1),\ldots,MS(t_n))$. Function $\mathcal{F}$ composes and filters items for $MS(a(t_1,\ldots,t_n))$ from those in the match sets $MS(t_1),\ldots,MS(t_n)$ computed for the n direct subtrees of $a(t_1,\ldots,t_n)$. For symbols a of rank n and trees $t_1,\ldots,t_n$, the value of $\mathcal{F}(MS(t_1),\ldots,MS(t_n))$ is defined to be

$$Cl(Comp_a(\,Filt_{a,1}(MS(t_1))\,,\ldots,\,Filt_{a,n}(MS(t_n))\,))$$

where:

- The $Filt_{a,i}$ are filter functions, filtering items from the respective match sets based e.g. on the values of a and i. Filtering is based on certain elements of children's match sets never contributing to the parent's match set. Such a child match set element may thus be safely disregarded for the computation of the parent's match set. Note that the identity function is among these filter functions.
- The $Comp_a$ are composition functions, which result in those subtrees of RHSs that are compositions of the subnodes' (filtered) match set elements and the symbol a.
- *Cl* is a closure function, adding e.g. nonterminal LHSs corresponding to complete RHSs that are in the composite match set.

The resulting algorithms are Algorithm (MATCH-SET, REC) (not using filter functions) and Algorithms (MATCH-SET, REC, FILTER) (with different instantiations of filter functions).

As an example of recursive match set computation, assume that we want to compute $MS(a(b(c),d))$ and that we use the identity function as a filter function (i.e. no filtering is applied). Furthermore, assume that $MS(b(c)) = \{b(c), b(B), B\}$

and $MS(d) = \{d, B\}$ have already been computed. Based on this, $MS(a(b(c), d))$ will contain $a(b(c), d)$ and $a(B, d)$ by composition with a, and B and S by the closure function, since $S \Rightarrow a(b(c), d)$ and $B \stackrel{*}{\Rightarrow} S$. No other elements are included in $MS(a(b(c), d))$.

It is straightforward to show that match sets and relations between them, as computed by Algorithm (MATCH-SET, REC) with particular item sets, correspond to states and transition relations of DFRTAs obtained by particular automata constructions as in Section 6. Recursive match set computation and the use of a DFRTA as an acceptor are simply two views on one approach [9, Chapter 5]. This correspondence is indicated by the dotted line in Figure 1.

To improve computation efficiency, values of MS cq. the acceptance function of the DFRTA are usually *tabulated* to prevent recomputation. Such tabulation uses a bijection between states (elements of $\mathcal{P}(Items)$) and integers for indexing the tables. The tabulation starts with symbols of rank 0, creating a state for each of them, and continues by computing the composition of symbols with match sets represented by existing states, for as long as new states are encountered, i.e. the computation is performed for the reachable part of state set $\mathcal{P}(Items)$ only. Such reachability-based tabulation is essentially straightforward, but somewhat intricate for trees/n-ary relations, even more so in the presence of filtering. We therefore do not present an example here; see e.g [9, Chapter 5] or [15] instead.

In practice, the size of the RTGs used leads to large but usually sparse tables: e.g for instruction selection, an RTG may well have hundreds of productions and lead to tables of over 100 MB. Filtering is therefore used to reduce storage space. For example, given match set $MS(b(c))$ above, $b(B)$ can be filtered, as it does not occur as a subtree of any *Item* in G. Different item categories can be filtered out (and may lead to different space savings, depending on the grammar):

– Filtering trees not occurring as proper subtrees (such as $b(B)$); filter TFILT, originally by Turner [19].
– Filtering trees not occurring as the ith child tree of a node labeled a; filter CFILT, originally by Chase [4,15].
– One of two new filter functions. Our research in taxonomizing the existing algorithms and filter functions lead us to describe these new ones, which can be seen as simplifications of Chase's filter functions yet somewhat surprisingly had not been described before:
  - Filtering trees based on index i only, i.e. not occurring as the ith child tree of any node; filter IFILT.
  - Filtering trees based on symbol a only, i.e. not occurring as a tree of a node labeled a at any child position; filter SFILT.

Even more surprisingly given their non-appearance in the literature, these two filters turn out to outperform Chase's filter on both text book example RTGs and instruction selection RTGs for e.g. the Intel X86 and Sun SPARC families: the index filter results in lower memory use, while the symbol filter results in slightly faster tabulation time than with Chase's filter. The experimental results have been described in detail in [5,9,18].

8 Algorithms using stringpath matching

The third subgraph (detail SP-MATCHER and below) of the taxonomy in Figure 1 contains algorithms for tree acceptance that are derived from algorithms for tree *pattern matching*. We only briefly sketch the main ideas.

The tree pattern matchers that we use in these tree acceptors reduce tree pattern matching to string pattern matching, using a technique first described in [16]. Each tree can be fully characterized by a set of stringpaths, and a tree pattern matches at a certain position in a tree if and only if all its stringpaths do. By traversing the subject tree and using a multiple string pattern matcher (e.g. [1]), matches of stringpaths can be detected. In [8] (originally presented at this conference as [7]) and [9] we discuss such algorithms in more detail and show that a certain DRFTA construction leads to DRFTAs—i.e. deterministic RF tree automata—that are also usable for stringpath matching. With a little extra bookkeeping, a tree pattern matcher of this kind can be turned into a tree acceptor.

9 Other Parts of the Work

Our work on regular tree algorithms has resulted in two taxonomies and a toolkit of algorithms. In this paper, we have mainly reported on one of the taxonomies, although it was pointed out in Section 3 how similar the tree pattern matching and tree acceptance algorithms and taxonomies are. In this section we present some remarks on the rest of the work. We refer the interested reader to [5,9] for more information.

As mentioned in Section 1, taxonomies form a good starting point for the construction of highly coherent algorithm toolkits. Based on the taxonomies of tree acceptance and tree matching algorithms, such an (experimental) toolkit was developed as part of our research. The toolkit contains most of the concrete algorithms and automata constructions from the taxonomies, as well as a number of fundamental algorithms and data structures—such as alphabets, trees, regular tree grammars, simple grammar transformations—and some extensions of tree acceptance algorithms to tree parsing and rudimentary instruction selection. The design of the toolkit was guided by the two taxonomies: the hierarchy of the taxonomies determines the class and interface hierarchies of the toolkit, and the abstract algorithms lead to straightforward method implementations. The toolkit, called FOREST FIRE, is implemented in Java and accompanied by a graphical user interface (GUI) called FIRE WOOD. This GUI supports input, output, creation and manipulation of data structures from the toolkit and was used to interactively experiment with and get insight into algorithms. More details on the toolkit and GUI can be found in [5,18]. The toolkit and GUI, including source code, example input files and brief manuals, are available for non-commercial purposes via `http://www.fastar.org`.

10 Concluding Remarks

The two taxonomies we constructed cover many algorithms and automata constructions for tree acceptance and tree pattern matching, which appeared in the literature in the past forty years. As for earlier taxonomies, their construction required a lot of time and effort to study original papers and distill the published algorithms' essential details (more so than in usual scientific research, which is typically limited to studying one or a few existing publications and building on those). Abstraction and sequentially adding details to obtain algorithms were essential and powerful means to clearly describe the algorithms and to make their correctness more apparent.

The uniform presentation in the taxonomies improves accessibility and shows algorithm relations: comparing algorithms previously presented in different styles has become easier and consultation of the original papers is often no longer necessary.

The taxonomies also lead to new and rediscovered algorithms: for example, two new filters were discovered which, though conceptually simple, are practically relevant. Furthermore, Turner's filter was more or less rediscovered. Our initial literature search, although apparently quite extensive, did not find Turner's paper—likely because it was not referred to by any other literature in the same field. As a result, we came up with the rather basic filter independently, before eventually finding it in the literature.

The uniform presentation simplified and guided the high-level design of our toolkit of regular tree algorithms, although the choice of representations for basic data structures still took some time and effort. Experiments with the toolkit provided some interesting results, including the fact that the new filters outperformed Chase's more complex but frequently used filter in many cases.

The results from our research thus are both theoretical and practical, ranging from formal definitions and algorithm taxonomies to a toolkit and experimental results. A form of symbiosis occurred between the theoretical and the practical: the taxonomies were helpful in constructing the toolkit, while the experiments with the toolkit in turn lead to a better understanding of the theoretical definitions and algorithm descriptions, thus helping to simplify the taxonomies.

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search*. Communications of the ACM, 18 1975, pp. 333–340.
2. W. S. Brainerd: *Tree generating regular systems*. Information and Control, 14 February 1969, pp. 217–231.
3. M. Broy: *Program construction by transformations: a family tree of sorting programs*, in Computer Program Synthesis Methodologies, A. W. Biermann and G. Guiho, eds., Reidel, 1983, pp. 1–49.
4. D. R. Chase: *An improvement to bottom-up tree pattern matching*, in Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, ACM, 1987, pp. 168–177.
5. L. Cleophas: *Forest FIRE and FIRE Wood: Tools for tree automata and tree algorithms*, in Post-proceedings of the 7th International Workshop on Finite-State Methods and Natural Language Processing (FSMNLP 2008), IOS Press, 2009.
6. L. Cleophas and K. Hemerik: *Forest FIRE: A taxonomy-based toolkit of tree automata and regular tree algorithms*, in Proceedings of the 14th International Conference on Implementation and Application of Automata (CIAA 2009), Springer, 2009.
7. L. Cleophas, K. Hemerik, and G. Zwaan: *A missing link in root-to-frontier tree pattern matching*, in Proceedings of the Prague Stringology Conference (PSC) 2005, August 2005.
8. L. Cleophas, K. Hemerik, and G. Zwaan: *Two related algorithms for root-to-frontier tree pattern matching*. International Journal of Foundations of Computer Science, 17(6) December 2006, pp. 1253–1272.
9. L. G. W. A. Cleophas: *Tree Algorithms: Two Taxonomies and a Toolkit*, PhD thesis, Dept. of Mathematics and Computer Science, Eindhoven University of Technology, April 2008, `http://alexandria.tue.nl/extra2/200810270.pdf`.

10. 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/`.
11. J. Darlington: *A synthesis of several sorting algorithms.* Acta Informatica, 11 1978, pp. 1–30.
12. J. Engelfriet: *Tree Automata and Tree Grammars*, Lecture Notes DAIMI FN-10, Aarhus University, April 1975.
13. C. Ferdinand, H. Seidl, and R. Wilhelm: *Tree automata for code selection.* Acta Informatica, 31 1994, pp. 741–760.
14. F. Gécseg and M. Steinby: *Tree Automata*, Akadémiai Kiadó, Budapest, 1984.
15. C. Hemerik and J. P. Katoen: *Bottom-up tree acceptors.* Science of Computer Programming, 13(1) 1989, pp. 51–72.
16. C. M. Hoffmann and M. J. O'Donnell: *Pattern matching in trees.* Journal of the ACM, 29(1) January 1982, pp. 68–95.
17. H. B. M. Jonkers: *Abstraction, specification and implementation techniques, with an application to garbage collection.* Mathematical Centre Tracts, 166 1983.
18. R. Strolenberg: *ForestFIRE & FIREWood, A Toolkit & GUI for Tree Algorithms*, Master's thesis, Dept. of Mathematics and Computer Science, Eindhoven University of Technology, June 2007, `http://alexandria.tue.nl/extra1/afstversl/wsk-i/strolenberg2007.pdf`.
19. P. K. Turner: *Up-down parsing with prefix grammars.* SIGPLAN Notices, 21(12) December 1986, pp. 167–174.
20. Y. van Dinther: *De systematische afleiding van acceptoren en ontleders voor boomgrammatica's*, Master's thesis, Faculteit Wiskunde en Informatica, Technische Universiteit Eindhoven, August 1987, (In Dutch).
21. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Dept. of Mathematics and Computing Science, Technische Universiteit Eindhoven, September 1995, `http://www.fastar.org/publications/PhD_Watson.pdf`.
22. B. W. Watson and G. Zwaan: *A taxonomy of sublinear multiple keyword pattern matching algorithms.* Science of Computer Programming, 27(2) 1996, pp. 85–118.
23. R. Wilhelm and D. Mauer: *Compiler Design*, Addison-Wesley, 1995.

String Suffix Automata and Subtree Pushdown Automata*

Jan Janoušek

Department of Computer Science
Faculty of Information Technologies
Czech Technical University in Prague
Zikova 1905/4, 166 36 Prague 6, Czech Republic
`Jan.Janousek@fit.cvut.cz`

Abstract. String suffix automata accept all suffixes of a given string and belong to the fundamental stringology principles. Extending their transitions by specific pushdown operations results in new subtree pushdown automata, which accept all subtrees of a given subject tree in prefix notation and are analogous to the suffix automata in their properties. The deterministic subtree pushdown automaton accepts an input subtree in time linear to the number of nodes of the subtree and its total size is linear to the number of nodes of the given subject tree.

Keywords: tree, subtree, string suffix automata, tree pattern matching, pushdown automata

1 Introduction

The theory of formal string (or word) languages [1,10,17] and the theory of formal tree languages [4,5,9] are important parts of the theory of formal languages [16]. The most famous models of computation of the theory of tree languages are various kinds of tree automata [4,5,9]. Trees can also be seen as strings, for example in their prefix (also called preorder) or postfix (also called postorder) notation. [11] shows that the deterministic pushdown automaton (PDA) is an appropriate model of computation for labelled ordered ranked trees in postfix notation and that the trees in postfix notation acceptable by deterministic PDA form a proper superclass of the class of regular tree languages, which are accepted by finite tree automata. In the further text we will omit word "string" when referencing to string languages or string automata.

Tree pattern matching is often declared to be analogous to the problem of string pattern matching [4]. One of the basic approaches used for string pattern matching can be represented by finite automata constructed for the text, which means that the text is preprocessed. Examples of these automata are suffix automata [6]. Given a text of size n, the suffix automaton can be constructed for the text in time linear in n. The constructed suffix automaton represents a complete index of the text for all possible suffixes and can find all occurrences of a string suffix and their positions in the text. The main advantage of this kind of finite automata is that the deterministic suffix automaton performs the search phase in time linear in the size of the input subtree and not depending on n.

This paper presents a new kind of acyclic PDAs for trees in prefix notation, which is analogous to string suffix automata and their properties: *subtree PDAs* accept all

* This research has been partially supported by the Ministry of Education, Youth and Sports under research program MSMT 6840770014, and by the Czech Science Foundation as project No. 201/09/0807.

subtrees of the tree. The basic idea of the subtree PDAs has been presented in [13]. This paper deals with the subtree PDAs in more details. [12] contains the detailed description of the subtree PDAs, related formal theorems, lemmas, and their proofs, many of which are skipped in this paper. Moreover, [12] describes an extension of the subtree PDAs – *tree pattern PDAs*, which accept all tree patterns matching the tree and are analogous to string factor automata in their basic properties.

By analogy with the string suffix automaton, the subtree PDA represents a complete index of the tree for all possible subtrees. Given a tree of size n, the main advantage of the deterministic subtree PDA is again that the search phase is performed in time linear in the size of the input subtree and not depending on n. We note that this cannot be achieved by any standard tree automaton because the standard deterministic tree automaton runs always on the subject tree, which means the searching by tree automata can be linear in n at the best.

Moreover, the presented subtree PDAs have the following two other properties. First, they are input-driven PDAs [20], which means that each pushdown operation is determined only by the input symbol. Input-driven PDAs can always be determinised [20]. Second, their pushdown symbol alphabets contain just one pushdown symbol and therefore their pushdown store can be implemented by a single integer counter. This means that the presented PDAs can be transformed to counter automata [3,19], which is a weaker and simpler model of computation than the PDA.

The rest of the paper is organised as follows. Basic definitions are given in section 2. Some properties of subtrees in prefix notation are discussed in the third section. The fourth section deals with the subtree PDA. The last section is the conclusion.

2 Basic notions

2.1 Ranked alphabet, tree, prefix notation

We define notions on trees similarly as they are defined in [1,4,5,9].

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

Based on concepts from graph theory (see [1]), a labelled, ordered, ranked tree over a ranked alphabet $\mathcal{A}$ can be defined as follows:

An *ordered directed graph* G is a pair (N, R), where N is a set of nodes and R is a set of linearly ordered lists of edges such that each element of R is of the form $((f, g_1), (f, g_2), \ldots, (f, g_n))$, where $f, g_1, g_2, \ldots, g_n \in N$, $n \geq 0$. This element would indicate that, for node f, there are n edges leaving f, the first entering node g_1, the second entering node 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 node f_n if there is an edge which leaves node f_{i-1} and enters node f_i for $1 \leq i \leq n$. A *cycle* is a path $(f_0, f_1, \ldots, f_n)$, where $f_0 = f_n$. An ordered *dag* (dag stands for Directed Acyclic Graph) is an ordered directed graph that has no cycle. A *labelling* of an ordered graph $G = (A, R)$ is a mapping of A into a set of labels. In the examples we use a_f for a short declaration of node f labelled by symbol a.

Given a node f, its *out-degree* is the number of distinct pairs $(f, g) \in R$, where $g \in A$. By analogy, the *in-degree* of the node f is the number of distinct pairs $(g, f) \in R$, where $g \in A$.

A *labelled, ordered, ranked and rooted tree* t over a ranked alphabet $\mathcal{A}$ is an ordered dag $t = (N, R)$ with a special node $r \in A$ called the *root* such that
(1) r has in-degree 0,
(2) all other nodes of t have in-degree 1,
(3) there is just one path from the root r to every $f \in N$, where $f \neq r$,
(4) every node $f \in N$ is labelled by a symbol $a \in \mathcal{A}$ and out-degree of a_f is $Arity(a)$.

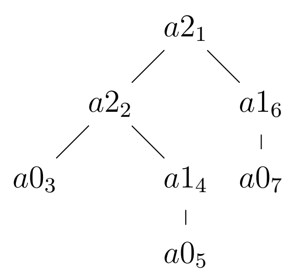
Nodes labelled by nullary symbols (constants) are called *leaves*.

Prefix notation pref(t) of a labelled, ordered, ranked and rooted tree t is obtained by applying the following *Step* recursively, beginning at the root of t:
Step: Let this application of *Step* be to node a_f. If a_f is a leaf, list a and halt. If a_f is not a leaf, let its direct descendants be $a_{f_1}, a_{f_2}, \ldots, a_{f_n}$. Then list a and subsequently apply *Step* to $a_{f_1}, a_{f_2}, \ldots, a_{f_n}$ in that order.

Example 1. Consider a ranked alphabet $\mathcal{A} = \{a2, a1, a0\}$. Consider a tree t_1 over $\mathcal{A}$ $t_1 = (\{a2_1, a2_2, a0_3, a1_4, a0_5, a1_6, a0_7\}, R)$, where R is a set of the following ordered sequences of pairs:

$$\begin{array}{l}((a2_1, a2_2), (a2_1, a1_6)),\\ ((a2_2, a0_3), (a2_2, a1_4)),\\ ((a1_4, a0_5)),\\ ((a1_6, a0_7))\end{array}$$

Tree t_1 in prefix notation is string $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$. Trees can be represented graphically and tree t_1 is illustrated in Fig. 1. □



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

Figure 1. Tree t_1 from Example 1 and its prefix notation

The height of a tree t, denoted by *Height(t)*, is defined as the maximal length of a path from the root of t to a leaf of t.

2.2 Alphabet, language, pushdown automaton

We define notions from the theory of string languages similarly as they are defined in [1,10].

Let an *alphabet* be a finite nonempty set of symbols. A *language* over an alphabet $\mathcal{A}$ is a set of strings over $\mathcal{A}$. Symbol $\mathcal{A}^*$ denotes the set of all strings over $\mathcal{A}$ including the empty string, denoted by ε. Set $\mathcal{A}^+$ is defined as $\mathcal{A}^+ = \mathcal{A}^* \setminus \{\varepsilon\}$. Similarly for string $x \in \mathcal{A}^*$, symbol x^m, $m \geq 0$, denotes the m-fold concatenation of x with $x^0 = \varepsilon$. Set x^* is defined as $x^* = \{x^m : m \geq 0\}$ and $x^+ = x^* \setminus \{\varepsilon\} = \{x^m : m \geq 1\}$.

A *nondeterministic finite automaton* (NFA) is a five-tuple $FM = (Q, \mathcal{A}, \delta, q_0, F)$, where Q is a finite set of *states*, $\mathcal{A}$ is an *input alphabet*, δ is a mapping from $Q \times \mathcal{A}$ into a set of finite subsets of Q, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is the set of final (accepting) states. A finite automaton FM is *deterministic* (DFA) if $\delta(q, a)$ has no more than one member for any $q \in Q$ and $a \in \mathcal{A}$. We note that the mapping δ is often illustrated by its transition diagram.

Every NFA can be transformed to an equivalent DFA [1,10]. The transformation constructs the states of the DFA as subsets of states of the NFA and selects only such accessible states (ie subsets). These subsets are called *d-subsets.* In spite of the fact that d-subsets are standard sets, they are often written in square brackets ([ ]) instead of in braces ({ }).

An (extended) *nondeterministic pushdown automaton* (nondeterministic PDA) is a seven-tuple $M = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$, where Q is a finite set of *states*, $\mathcal{A}$ is an *input alphabet*, G is a *pushdown store alphabet*, δ is a mapping from $Q \times (\mathcal{A} \cup \{\varepsilon\}) \times G^*$ into a set of finite subsets of $Q \times G^*$, $q_0 \in Q$ is an initial state, $Z_0 \in G$ is the initial pushdown symbol, and $F \subseteq Q$ is the set of final (accepting) states. Triplet $(q, w, x) \in Q \times \mathcal{A}^* \times G^*$ denotes the configuration of a pushdown automaton. In this paper we will write the top of the pushdown store x on its right hand side. The initial configuration of a pushdown automaton is a triplet (q_0, w, Z_0) for the input string $w \in \mathcal{A}^*$.

The relation $\vdash_M \subset (Q \times \mathcal{A}^* \times G^*) \times (Q \times \mathcal{A}^* \times G^*)$ is a *transition* of a pushdown automaton M. It holds that $(q, aw, \alpha\beta) \vdash_M (p, w, \gamma\beta)$ if $(p, \gamma) \in \delta(q, a, \alpha)$. The k-th power, transitive closure, and transitive and reflexive closure of the relation $\vdash_M$ is denoted $\vdash_M^k$, $\vdash_M^+$, $\vdash_M^*$, respectively. A pushdown automaton M is *deterministic* pushdown automaton (deterministic PDA), if it holds:

1. $|\delta(q, a, \gamma)| \leq 1$ for all $q \in Q$, $a \in \mathcal{A} \cup \{\varepsilon\}$, $\gamma \in G^*$.
2. If $\delta(q, a, \alpha) \neq \emptyset$, $\delta(q, a, \beta) \neq \emptyset$ and $\alpha \neq \beta$ then α is not a suffix of β and β is not a suffix of α.
3. If $\delta(q, a, \alpha) \neq \emptyset$, $\delta(q, \varepsilon, \beta) \neq \emptyset$, then α is not a suffix of β and β is not a suffix of α.

A pushdown automaton is *input-driven* if each of its pushdown operations is determined only by the input symbol.
A language L accepted by a pushdown automaton M is defined in two distinct ways:

1. *Accepting by final state:*

$$L(M) = \{x : \delta(q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \gamma) \wedge x \in \mathcal{A}^* \wedge \gamma \in G^* \wedge q \in F\}.$$

2. *Accepting by empty pushdown store:*

$$L_\varepsilon(M) = \{x : (q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \varepsilon) \wedge x \in \mathcal{A}^* \wedge q \in Q\}.$$

If PDA accepts the language by empty pushdown store then the set F of final states is the empty set. The subtree PDAs accept the languages by empty pushdown store.

For more details see [1,10].

2.3 Example of string suffix automaton

Example 2. Given the prefix notation $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ of tree t_1 from Example 1, the corresponding nondeterministic suffix automaton is $FM_{nsuf}(pref(t_1)) = (\{0,1,2,3,4,5,6,7\}, \mathcal{A}, \delta_n, 0, \{7\}))$, where its transition diagram is illustrated in Fig. 2. (For the construction of the nondeterministic suffix automaton see [14].)

After the standard transformation of a nondeterministic suffix automaton to a deterministic one [10], the deterministic suffix automaton for $pref(t_1)$ is $FM_{dsuf}(pref(t_1)) = (\{[0],[1,2],[2],[3],[4],[5],[6],[7],[3,5,7],[4,6],[5,7]\}, \mathcal{A}, \delta_d, 0, \{[7],[3,5,7],[5,7]\}))$, where its transition diagram is illustrated in Fig. 3.

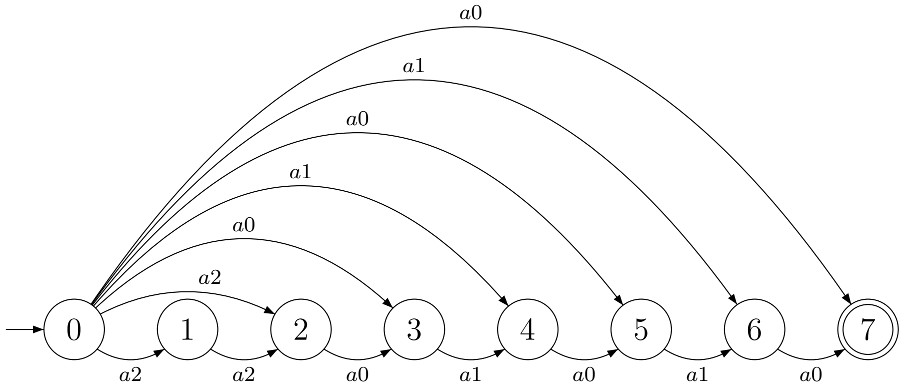


Figure 2. Transition diagram of nondeterministic suffix automaton for string $a2\ a2\ a0\ a1\ a0\ a1\ a0$

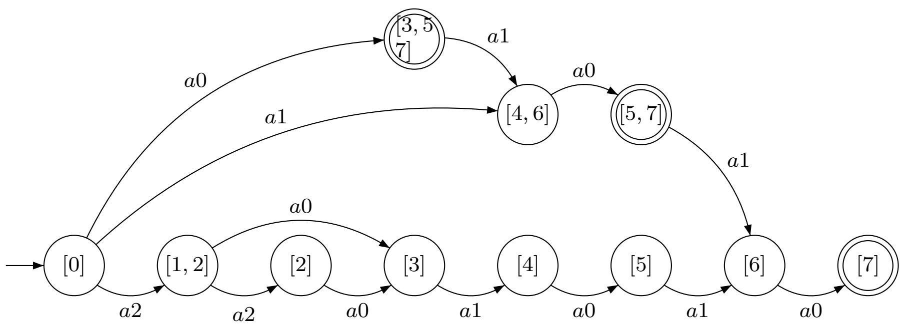


Figure 3. Transition diagram of deterministic suffix automaton for string $a2\ a2\ a0\ a1\ a0\ a1\ a0$

3 Properties of subtrees in prefix notation

In this section we describe some general properties of the prefix notation of a tree and of its subtrees. These properties are important for the construction of subtree PDA, which is described in the next section.

Example 3. Consider tree t_1 in prefix notation $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 1, which is illustrated in Fig. 1. Tree t_1 contains only subtrees shown in Fig. 4.

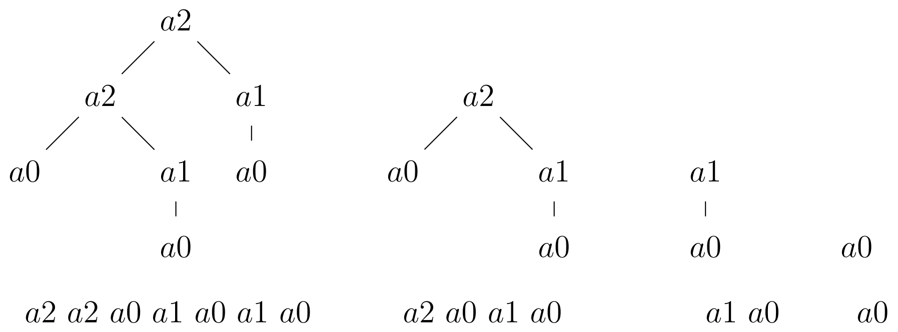


Figure 4. All subtrees of tree t_1 from Example 1, and their prefix notations

Generally, it holds for any tree that each of its subtrees in prefix notation is a substring of the tree in prefix notation.

Theorem 4. *Given a tree t and its prefix notation $pref(t)$, all subtrees of t in prefix notation are substrings of $pref(t)$.*

Proof. In [12]. □

However, not every substring of a tree in prefix notation is a prefix notation of its subtree. This can be easily seen from the fact that for a given tree with n nodes there can be $\mathcal{O}(n^2)$ distinct substrings, but there are just n subtrees – each node of the tree is the root of just one subtree. Just those substrings which themselves are trees in prefix notation are those which are the subtrees in prefix notation. This property is formalised by the following definition and theorem.

Definition 5. *Let $w = a_1a_2 \cdots a_m$, $m \geq 1$, be a string over a ranked alphabet $\mathcal{A}$. Then, the* arity checksum $ac(w) = arity(a_1) + arity(a_2) + \cdots + arity(a_m) - m + 1 = \sum_{i=1}^{m} arity(a_i) - m + 1$.

Theorem 6. *Let $pref(t)$ and w be a tree t in prefix notation and a substring of $pref(t)$, respectively. Then, w is the prefix notation of a subtree of t, if and only if $ac(w) = 0$, and $ac(w_1) \geq 1$ for each w_1 , where $w = w_1x$, $x \neq \varepsilon$.*

Proof. In [12]. □

We note that in subtree PDAs the arity checksum is computed by pushdown operations, where the contents of the pushdown store represents the corresponding arity checksum. For example, an empty pushdown store means that the corresponding arity checksum is equal to 0.

4 Subtree pushdown automaton

This section deals with the subtree PDA for trees in prefix notation: algorithms and theorems are given and the subtree PDA and its construction are demonstrated on an example.

Definition 7. *Let t and $pref(t)$ be a tree and its prefix notation, respectively. A* subtree pushdown automaton *for $pref(t)$ accepts all subtrees of t in prefix notation.*

First, we start with a PDA which accepts the whole subject tree in prefix notation. The construction of the PDA accepting a tree in prefix notation by the empty pushdown store is described by Alg. 1. The constructed PDA is deterministic.

Algorithm 1. Construction of a PDA accepting a tree t in prefix notation $pref(t)$.
Input: A tree t over a ranked alphabet $\mathcal{A}$; prefix notation $pref(t) = a_1 a_2 \cdots a_n$, $n \geq 1$.
Output: PDA $M_p(t) = (\{0, 1, 2, \ldots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$.
Method:

1. For each state i, where $1 \leq i \leq n$, create a new transition
$\delta(i-1, a_i, S) = (i, S^{Arity(a_i)})$, where $S^0 = \varepsilon$. □

Example 8. A PDA accepting tree t_1 in prefix notation $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 1, which has been constructed by Alg. 1, is deterministic PDA $M_p(t_1) = (\{0, 1, 2, 3, 4, 5, 6, 7\}, \mathcal{A}, \{S\}, \delta_1, 0, S, \emptyset))$, where the mapping δ_1 is a set of the following transitions:

$$\begin{array}{l}
\delta_1(0, a2, S) = (1, SS) \\
\delta_1(1, a2, S) = (2, SS) \\
\delta_1(2, a0, S) = (3, \varepsilon) \\
\delta_1(3, a1, S) = (4, S) \\
\delta_1(4, a0, S) = (5, \varepsilon) \\
\delta_1(5, a1, S) = (6, S) \\
\delta_1(6, a0, S) = (7, \varepsilon)
\end{array}$$

The transition diagram of deterministic PDA $M_p(t_1)$ is illustrated in Fig. 5. In this figure for each transition rule $\delta_1(p, a, \alpha) = (q, \beta)$ from δ the edge leading from state p to state q is labelled by the triple of the form $a|\alpha \mapsto \beta$.

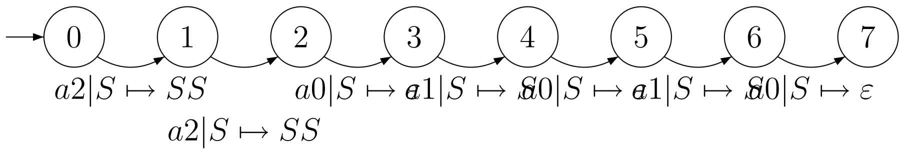


Figure 5. Transition diagram of deterministic PDA $M_p(t_1)$ accepting tree t_1 in prefix notation $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 8

Fig. 6 shows the sequence of transitions (trace) performed by deterministic PDA $M_p(t_1)$ for tree t_1 in prefix notation. □

It holds that every input-driven PDA that has the same pushdown operations as they are defined for the above deterministic PDA $M_p(t)$ for tree t in prefix notation behaves such that the contents of its pushdown store corresponds to the arity checksum. This is described by the following theorem. We note that such pushdown operations correspond to the pushdown operations of the standard top-down parsing algorithm for a context-free grammar with rules of the form

$$S \to a\ S^{arity(a)}.$$

For principles of the standard top–down (LL) parsing algorithm see [1].

State	Input	Pushdown Store
0	$a2\ a2\ a0\ a1\ a0\ a1\ a0$	S
1	$a2\ a0\ a1\ a0\ a1\ a0$	$S\ S$
2	$a0\ a1\ a0\ a1\ a0$	$S\ S\ S$
3	$a1\ a0\ a1\ a0$	$S\ S$
4	$a0\ a1\ a0$	$S\ S$
5	$a1\ a0$	S
6	$a0$	S
7	ε	ε
accept		

Figure 6. Trace of deterministic PDA $M_p(t_1)$ from Example 8 for tree t_1 in prefix notation $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$

Theorem 9. *Let $M = (\{Q, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$ be an input-driven PDA of which each transition from δ is of the form $\delta(q_1, a, S) = (q_2, S^i)$, where $i = arity(a)$. Then, if $(q_3, w, S) \vdash_M^+ (q_4, \varepsilon, S^j)$, then $j = ac(w)$.*

Proof. In [12]. □

The correctness of the deterministic PDA constructed by Alg. 1, which accepts trees in prefix notation, is described by the following lemma.

Lemma 10. *Given a tree t and its prefix notation $pref(t)$, the PDA $M_p(t) = (\{0, 1, 2, \ldots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$, where $n \geq 0$, constructed by Alg. 1 accepts $pref(t)$.*

Proof. In [12]. □

We present the construction of the deterministic subtree PDA for trees in prefix notation. The construction consists of two steps. First, a nondeterministic subtree PDA is constructed by Alg. 2. This nondeterministic subtree PDA is an extension of the PDA accepting tree in prefix notation, which is constructed by Alg. 1. Second, the constructed nondeterministic subtree PDA is transformed to the equivalent deterministic subtree PDA. Although a nondeterministic PDA cannot generally be determinised, the constructed nondeterministic subtree PDA is an input-driven PDA and therefore can be determinised [20].

Algorithm 2. Construction of a nondeterministic subtree PDA for a tree t in prefix notation $pref(t)$.
Input: A tree t over a ranked alphabet $\mathcal{A}$; prefix notation $pref(t) = a_1 a_2 \cdots a_n$, $n \geq 1$.
Output: Nondeterministic subtree PDA $M_{nps}(t) = (\{0, 1, 2, \ldots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$.
Method:

1. Create PDA $M_{nps}(t)$ as PDA $M_p(t)$ by Alg. 1.
2. For each state i, where $2 \leq i \leq n$, create a new transition $\delta(0, a_i, S) = (i, S^{Arity(a_i)})$, where $S^0 = \varepsilon$. □

Example 11. A subtree PDA for tree t_1 in prefix notation $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 1, which has been constructed by Alg. 2, is nondeterministic PDA $M_{nps}(t_1) = (\{0, 1, 2, 3, 4, 5, 6, 7\}, \mathcal{A}, \{S\}, \delta_2, 0, S, \emptyset))$, where mapping δ_2 is a set of the following transitions:

$$\begin{array}{ll}
\delta_2(0,a2,S)=(1,SS) & \\
\delta_2(1,a2,S)=(2,SS) & \delta_2(0,a2,S)=(2,SS)\\
\delta_2(2,a0,S)=(3,\varepsilon) & \delta_2(0,a0,S)=(3,\varepsilon)\\
\delta_2(3,a1,S)=(4,S) & \delta_2(0,a1,S)=(4,S)\\
\delta_2(4,a0,S)=(5,\varepsilon) & \delta_2(0,a0,S)=(5,\varepsilon)\\
\delta_2(5,a1,S)=(6,S) & \delta_2(0,a1,S)=(6,S)\\
\delta_2(6,a0,S)=(7,\varepsilon) & \delta_2(0,a0,S)=(7,\varepsilon)
\end{array}$$

The transition diagram of nondeterministic PDA $M_{nps}(t_1)$ is illustrated in Fig. 7. Again, in this figure for each transition rule $\delta_2(p,a,\alpha)=(q,\beta)$ from δ_2 the edge leading from state p to state q is labelled by the triple of the form $a|\alpha \mapsto \beta$.

A comparison of Figs. 7 and 2 shows that the states and the transitions of nondeterministic subtree PDA $M_{nps}(t_1)$ correspond to the states and the transitions, respectively, of the nondeterministic string suffix automaton for *pref*(t_1); the transitions of the subtree PDA are extended by pushdown operations so that it holds that the number of symbols S in the pushdown store is equal to the corresponding arity checksum. □

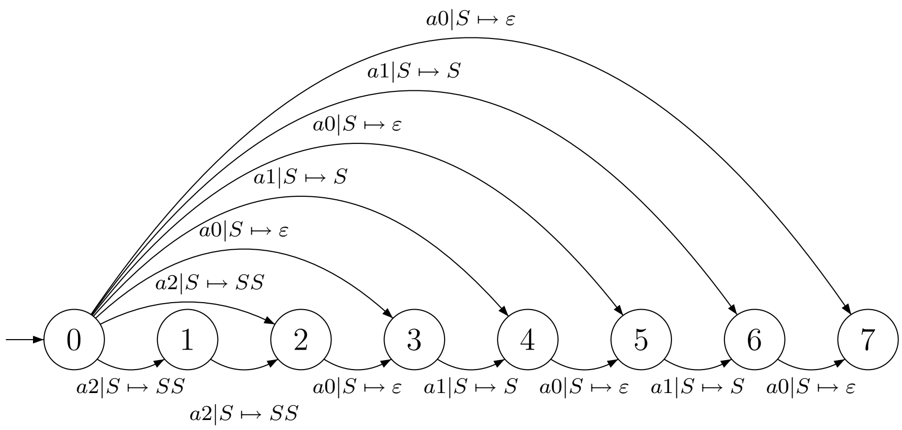


Figure 7. Transition diagram of nondeterministic subtree PDA $M_{nps}(t_1)$ for tree t_1 in prefix notation *pref*$(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 11

Theorem 12. *Given a tree t and its prefix notation pref(t), the PDA $M_{nps}(t)$ constructed by Alg. 2 is a subtree PDA for pref(t).*

Proof. In [12]. □

It is known that each nondeterministic input-driven PDA can be transformed to an equivalent deterministic input-driven PDA [20]. To construct deterministic subtree or tree pattern PDAs from their nondeterministic versions we use the transformation described by Alg. 3. This transformation is a simple extension of the well known transformation of a nondeterministic finite automaton to an equivalent deterministic one [10]. Again, the states of the resulting deterministic PDA correspond to subsets of the states of the original nondeterministic PDA, and these subsets are again called d-subsets. Moreover, the original nondeterministic PDA is assumed to be acyclic with a specific order of states, and Alg. 3 precomputes the possible contents of the pushdown store in particular states of the deterministic PDA according to pushdown operations and selects only those transitions and accessible states of the deterministic PDA for which the pushdown operations are possible. The assumption that the PDA is acyclic results in a finite number of possible contents of the pushdown store. Furthermore, the assumption of the specific order of states allows us to compute these contents of the pushdown store easily in a one-pass way.

Algorithm 3. Transformation of an input-driven nondeterministic PDA to an equivalent deterministic PDA.
Input: Acyclic input-driven nondeterministic PDA $M_{nx}(t) = (\{0, 1, 2, \ldots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$, where the ordering of its states is such that if $\delta(p, a, \alpha) = (q, \beta)$, then $p < q$.
Output: Equivalent deterministic PDA $M_{dx}(t) = (Q', \mathcal{A}, \{S\}, \delta', q_I, S, \emptyset)$.
Method:

1. Let $cpds(q')$, where $q' \in Q'$, denote a set of strings over $\{S\}$. (The abbreviation *cpds* stands for Contents of the PushDown Store.)
2. Initially, $Q' = \{[0]\}$, $q_I = [0]$, $cpds([0]) = \{S\}$ and $[0]$ is an unmarked state.
3. (a) Select an unmarked state q' from Q' such that q' contains the smallest possible state $q \in Q$, where $0 \leq q \leq n$.
   (b) For each input symbol $a \in \mathcal{A}$:
      i. Add transition $\delta'(q', a, \alpha) = (q'', \beta)$, where $q'' = \{q : \ \delta(p, a, \alpha) = (q, \beta)$ for all $p \in q'\}$. If q'' is not in Q' then add q'' to Q' and create $cpds(q'') = \emptyset$. Add ω, where $\delta(q', a, \gamma) \vdash_{M_{dx}(t)} (q'', \varepsilon, \omega)$ and $\gamma \in cpds(q')$, to $cpds(q'')$.
   (c) Set the state q' as marked.
4. Repeat step 3 until all states in Q' are marked. □

The deterministic subtree automaton for a tree in prefix notation is demonstrated by the following example. The PDA reads an input subtree in prefix notation and the accepting state corresponds to the rightmost leaves of all occurrences of the input subtree in the subject tree.

Example 13. The deterministic subtree PDA for tree t_1 in prefix notation $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 1, which has been constructed by Alg. 3 from nondeterministic subtree PDA $M_{nps}(t_1)$ from Example 11, is deterministic PDA $M_{dps}(t_1) = (\{[0], [1, 2], [2], [3], [4], [5], [6], [7], [3, 5, 7], [4, 6], [5, 7]\}, \mathcal{A}, \{S\}, \delta_3,\ [0], S, \emptyset))$, where mapping δ_3 is a set of the following transitions:

$$
\begin{array}{ll}
\delta_3([0], a2, S) \quad = ([1, 2], SS) & \delta_3([0], a0, S) \quad = ([3, 5, 7], \varepsilon) \\
\delta_3([1, 2], a2, S) = ([2], SS) & \delta_3([0], a1, S) \quad = ([4, 6], S) \\
\delta_3([2], a0, S) \quad = ([3], \varepsilon) & \delta_3([1, 2], a0, S) = ([3], \varepsilon) \\
\delta_3([3], a1, S) \quad = ([4], S) & \delta_3([4, 6], a0, S) = ([5, 7], \varepsilon) \\
\delta_3([4], a0, S) \quad = ([5], \varepsilon) & \\
\delta_3([5], a1, S) \quad = ([6], S) & \\
\delta_3([6], a0, S) \quad = ([7], \varepsilon) &
\end{array}
$$

We note that there are no transitions leading from states $[3, 5, 7]$, $[5, 7]$ and $[7]$, because the pushdown store in these state is always empty and therefore no transition is possible from these states due to the pushdown operations. This means that the deterministic subtree PDA $M_{dps}(t_1)$ has fewer transitions than the deterministic string suffix automaton constructed for $pref(t_1)$ [6,14,18], as can be seen by comparing Figs. 3 and 8.

The transition diagram of deterministic PDA $M_{dps}(t_1)$ is illustrated in Fig. 8. Again, in this figure for each transition rule $\delta_3(p, a, \alpha) = (q, \beta)$ from δ_3 the edge leading from state p to state q is labelled by the triple of the form $a|\alpha \mapsto \beta$.

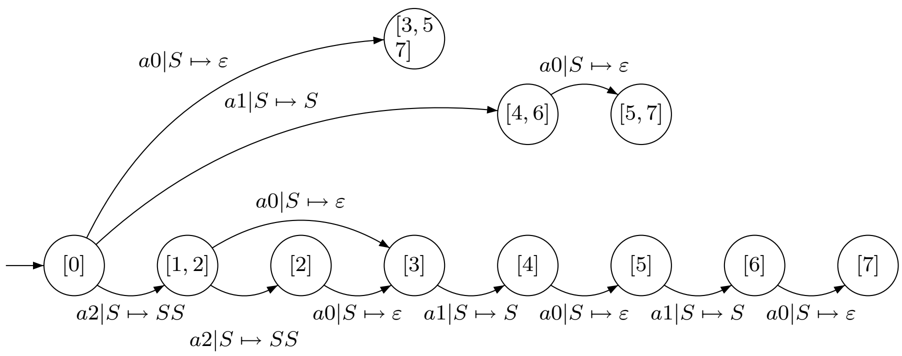


Figure 8. Transition diagram of deterministic subtree PDA $M_{dps}(t_1)$ for tree in prefix notation $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 13

Fig. 9 shows the sequence of transitions (trace) performed by deterministic subtree PDA $M_{dps}(t_1)$ for an input subtree st in prefix notation $pref(st) = a_1a_0$. The accepting state is $[5, 7]$, which means there are two occurrences of the input subtree st in tree t_1 and their rightmost leaves are nodes $a0_5$ and $a0_7$. □

State	Input	Pushdown Store
[0]	$a1\ a0$	S
[4, 6]	$a0$	S
[5, 7]	ε	ε
accept		

Figure 9. Trace of deterministic subtree PDA $M_{dps}(t_1)$ from Example 13 for an input subtree st in prefix notation $pref(st) = a1a0$

Theorem 14. *Given an acyclic input-driven nondeterministic PDA* $M_{nx}(t) = (Q, \mathcal{A}, \{S\}, \delta, q_0, S, \emptyset)$*, the deterministic PDA* $M_{dx}(t) = (Q', \mathcal{A}, \{S\}, \delta', \{q_0\}, S, \emptyset)$ *constructed by Alg. 3 is equivalent to PDA* $M_{nx}(t)$*.*

Proof. In [12]. □

We note that trees with the structure $pref(t) = (a1)^{n-1}a0$ represent strings. Such a tree is illustrated in Fig. 10. It can be simply shown that the deterministic subtree PDAs for such trees have the same number of states and transitions as the deterministic suffix automata constructed for $pref(t)$ and accept the same language.

It is obvious that the number of distinct subtrees in a tree can be at most the number of nodes of the tree.

Lemma 15. *Given a tree t with n nodes, the number of distinct subtrees of tree t is equal or smaller than n.*

Proof. In [12]. □

At the end of this section we discuss the total size of the constructed deterministic subtree PDA, which cannot be greater than the total size of the deterministic suffix automaton constructed for $pref(t)$ [6,7]. We recall that the deterministic subtree PDA can have even fewer states and transitions than the corresponding deterministic string suffix automaton as certain states and transitions need not be accessible due to pushdown operations.

$$\begin{array}{c} a1 \\ | \\ a1 \\ | \\ a1 \\ | \\ \vdots \\ | \\ a0 \end{array}$$

$$pref(t_2) = (a1)^{n-1}a0$$

Figure 10. A tree t_2, which represents a string, and its prefix notation

Theorem 16. *Given a tree t with n nodes and its prefix notation $pref(t)$, the deterministic subtree PDA $M_{dps}(t)$ constructed by Algs. 2 and 3 has just one pushdown symbol, fewer than $N \leq 2n + 1$ states and at most $N + n - 1 \leq 3n$ transitions.*

Proof. The deterministic subtree PDA in question may have only states and transitions which correspond to the states and the transitions, respectively, of the deterministic suffix automaton constructed for $pref(t)$. Therefore, the largest possible numbers of states and transitions of the deterministic subtree PDA are the same as those of the deterministic suffix automaton. The numbers of states and transitions of the deterministic suffix automaton are proved in Theorems 6.1 and 6.2 in [7] or in Theorem 5.3.5 in [18]. We note that these proofs are based on the following principle: Given a substring u, the d-subset of the state in which the deterministic suffix automaton is after reading u is called the *terminator set* of u [18]. It holds for any two substrings u_1 and u_2 that their terminator sets cannot overlap; in other words, the terminator sets of a deterministic suffix automaton correspond to a tree structure. It has been proved that this tree structure is such that the above-mentioned numbers of states and transitions hold. □

5 Conclusion

We have described a new kind of pushdown automata: subtree PDAs for trees in prefix notation. These pushdown automata are in their properties analogous to suffix automata, which are widely used in stringology. The presented subtree PDAs represent a complete index of the subject tree with n nodes for all possible subtrees and the deterministic version allows to find all occurrences of input subtrees of size m in time linear in m and not depending on n.

Regarding specific tree algorithms whose model of computation is the standard deterministic pushdown automaton, recently we have introduced principles of other three new algorithms. First, a new and simple method for constructing subtree pattern matchers as deterministic pushdown automata directly from given subtrees without constructing finite tree automata as an intermediate product [8,13]. Second, tree pattern pushdown automata, which represent a complete index of the tree for all tree patterns matching the tree and the search phase of all occurrences of a tree pattern of size m is performed in time linear in m and not depending on the size of the tree [12,13]. These automata representing indexes of trees for all tree patterns are analogous in their properties to the string factor automata [6,7] and are an extension of the subtree PDA presented in this paper. Third, a method for finding all repeats of connected subgraphs in trees with the use of subtree or tree pattern PDA [15,13]. More details on these results and related information can also be found on [2].

I would like to thank to Bořivoj Melichar and anonymous referees – their comments have contributed to improving the text significantly.

References

1. A. V. Aho and J. D. Ullman: *The theory of parsing, translation, and compiling*, Prentice-Hall Englewood Cliffs, N.J.,, 1972.
2. *Arbology www pages*: Available on: `http://www.arbology.org`, July 2009.
3. J. Berstel: *Transductions and Context-Free Languages*, Teubner Studienbucher, Stuttgart, 1979.
4. L. Cleophas: *Tree Algorithms. Two Taxonomies and a Toolkit.*, PhD thesis, Technische Universiteit Eindhoven, Eindhoven, 2008.
5. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi: *Tree automata techniques and applications.* Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007, release October, 12th 2007.
6. M. Crochemore and C. Hancart: *Automata for matching patterns*, in Handbook of Formal Languages, G. Rozenberg and A. Salomaa, eds., vol. 2 Linear Modeling: Background and Application, Springer-Verlag, Berlin, 1997, ch. 9, pp. 399–462.
7. M. Crochemore and W. Rytter: *Jewels of Stringology*, World Scientific, New Jersey, 1994.
8. T. Flouri, J. Janoušek, and B. Melichar: *Tree pattern matching by deterministic pushdown automata.* accepted for WAPL 2009 conference, 2009.
9. F. Gecseg and M. Steinby: *Tree languages*, in Handbook of Formal Languages, G. Rozenberg and A. Salomaa, eds., vol. 3 Beyond Words. Handbook of Formal Languages, Springer-Verlag, Berlin, 1997, pp. 1–68.
10. J. E. Hopcroft, R. Motwani, and J. D. Ullman: *Introduction to automata theory, languages, and computation*, Addison-Wesley, Boston, 2nd ed., 2001.
11. J. Janoušek and B. Melichar: *On regular tree languages and deterministic pushdown automata.* accepted for publication in Acta Informatica, Springer, 2009.
12. J. Janoušek and B. Melichar: *Subtree and tree pattern pushdown automata for trees in prefix notation.* submitted for publication, 2009.
13. *London stringology days 2009 conference presentations*: Available on: `http://www.dcs.kcl.ac.uk/events/LSD&LAW09/`, King's College London, London, February 2009.
14. B. Melichar, J. Holub, and T. Polcar: *Text searching algorithms.* Available on: `http://stringology.org/athens/`, 2005, release November 2005.
15. B. Melichar and J. Janoušek: *Repeats in trees by subtree and tree pattern pushdown automata.* draft, 2009.
16. G. Rozenberg and A. Salomaa, eds., *Handbook of Formal Languages*, Springer-Verlag, Berlin, 1997.
17. G. Rozenberg and A. Salomaa, eds., *Vol. 1: Word, Language, Grammar, Handbook of Formal Languages*, Springer-Verlag, Berlin, 1997.
18. B. Smyth: *Computing Patterns in Strings*, Addison-Wesley-Pearson Education Limited, Essex, England, 2003.
19. L. G. Valiant and M. Paterson: *Deterministic one-counter automata*, in Automaten theorie und Formale Sprachen, 1973, pp. 104–115.
20. K. Wagner and G. Wechsung: *Computational Complexity*, Springer-Verlag, Berlin, 2001.

On Minimizing Deterministic Tree Automata

Loek Cleophas, Derrick G. Kourie, Tinus Strauss, and Bruce W. Watson

FASTAR Research Group, Department of Computer Science, University of Pretoria,
0002 Pretoria, Republic of South Africa, http://www.fastar.org
loek@loekcleophas.com, dkourie@cs.up.ac.za, tstrauss@cs.up.ac.za, bruce@fastar.org

Abstract. We present two algorithms for minimizing deterministic frontier-to-root tree automata (DFRTAs) and compare them with their string counterparts. The presentation is incremental, starting out from definitions of minimality of automata and state equivalence, in the style of earlier algorithm taxonomies by the authors. The first algorithm is the classical one, initially presented by Brainerd in the 1960s and presented (sometimes imprecisely) in standard texts on tree language theory ever since. The second algorithm is completely new. This algorithm, essentially representing the generalization to ranked trees of the string algorithm presented by Watson and Daciuk, incrementally minimizes a DFRTA. As a result, intermediate results of the algorithm can be used to reduce the initial automaton's size. This makes the algorithm useful in situations where running time is restricted (for example, in real-time applications). We also briefly sketch how a concurrent specification of the algorithm in CSP can be obtained from an existing specification for the DFA case.

Keywords: deterministic frontier-to-root tree automata, deterministic bottom-up tree automata, minimization, minimality

1 Introduction

Minimization of deterministic finite *string* automata (DFAs) has been studied since the late 1950s. Many applications of such minimization arose, and as a result many algorithms were published, often with vastly differing presentation styles and levels of formality [12]. For the case of deterministic frontier-to-root (aka bottom-up) tree automata (DFRTAs), minimization was considered less frequently, likely due to fewer applications being considered at the time. Minimization for DFRTAs was first discussed in the late 1960s by Brainerd [1,2], who presented a textual procedure for minimization that is essentially the generalization to trees of a classical DFA minimization approach. Later standard references either do not discuss minimization at all or present an approach similar to Brainerd's. Later standard references either do not discuss minimization at all [8], have a discussion similar to Brainerd's [9], or give a somewhat imprecise algorithm [6]. As pointed out by Carrasco, Daciuk and Forcada [3], discussions of an implementation of such a minimization algorithm are hard to find. Their paper presents such a discussion for the case of deterministic bottom-up tree automata *over unranked trees*. Carrasco et al. also presented an algorithm for *incremental construction* of minimal deterministic bottom-up tree automata over unranked trees [4].

For the string case, Watson presented an extensive taxonomy of minimization algorithms [12, Chapter 7]. A concurrent specification of an incremental minimization algorithm for the string case was recently presented by Strauss et al. [11], offering possibilities for exploiting parallelism on systems or networks of systems with multiple CPU cores.

Loek Cleophas, Derrick G. Kourie, Tinus Strauss, Bruce W. Watson: On Minimizing Deterministic Tree Automata, pp. 173–182.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

Both in the string and the tree case, minimization is based on the notion of language equivalence between states; either in the form of that equivalence relation, its complement (i.e. the distinguishability relation), or of the state set partition induced by the equivalence relation. In the classical approach, the algorithms start off with a set of possibly equivalent state pairs. This is then refined iteratively by removing state pairs that are definitely not equivalent, until the greatest fixed-point is reached. The resulting set defines a partitioning of the states set into equivalence classes which correspond to the state set of the minimal automaton equivalent to the original one. Put differently, the algorithms compute the greatest fixed point from the top i.e. from the unsafe side [13], meaning that intermediate results of the algorithm cannot be used to reduce the initial DFA.

Watson in [12] and most recently Watson & Daciuk in [13] present an incremental approach to DFA minimization. Their approach results in an algorithm that starts out with a singleton partition for each of the states of the initial DFA and refines this partition by iteratively merging partitions that are shown to be equivalent. The greatest fixed-point reached corresponds to the state set of the minimal automaton equivalent to the original one. Such an algorithm thus computes the greatest fixed point from below i.e. from the safe side. Clearly, intermediate results from such an algorithm can already be used to reduce the original DFA.

In this paper, we focus on minimization of DFRTAs. We present both an algorithm using the classical approach and a new algorithm using the incremental approach to minimization. The latter is the first description of such an algorithm for the tree case. The former is presented more precisely than in most existing literature (with the exception of [3], although that work considers the case of *unranked* deterministic bottom-up tree automata). Furthermore, its inclusion allows one easily to compare and contrast the two approaches (as is the case for DFA minimization algorithms in the taxonomy of such algorithms in [12, Ch. 7]).

We also briefly consider the generalization to the DFRTA case of an existing concurrent specification in CSP of the incremental DFA minimization algorithm. This elegant generalization further increases the parallelism in the specification.

The rest of this paper is structured as follows:

- Section 2 discusses some preliminaries on DFAs, trees, and DFRTAs needed in the remainder of the paper.
- Equivalence of states and minimality of DFAs and DFRTAs are discussed in Section 3.
- Section 4 discusses the classical approach to minimization of DFAs and DFRTAs.
- The incremental approach to minimization and our resulting new incremental minimization algorithm for DFRTAs is discussed in Section 5.
- Section 6 presents a concurrent specification for the new algorithm in CSP, based on an existing specification of incremental minimization for the string case.
- Finally, Section 7 presents some concluding remarks and suggestions for future work.

2 Preliminaries

Since our discussion of minimization of tree automata frequently refers to that for the case of string automata, we recall some definitions related to *deterministic finite (string) automata*.

Definition 1. *A* DFA *M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ such that Q is a finite set, the* state set*; Σ is an* alphabet *(a finite set of symbols); $\delta \in Q \times \Sigma \nrightarrow Q$ is the transition function; $q_0 \in Q$ is the* initial *or* start state*; and $F \subseteq Q$ is the set of* final *or* accepting states*.*

We extend transition function δ to its transitive closure δ^*, defined inductively by $\delta^*(q, \varepsilon) = q$ and $\delta^*(q, aw) = \delta^*(\delta(q, a), w)$. For every state $q \in Q$ of a DFA, its *right language* (*left language*) is defined as $\overrightarrow{\mathcal{L}} = \{w \in \Sigma^* | \delta^*(q, w) \in F\}$ ($\overleftarrow{\mathcal{L}} = \{w \in \Sigma^* | \delta^*(q_0, w) = q\}$).

We call a DFA *complete* if and only if its transition function δ is a total function. A DFA with a partial transition function can be transformed into a complete one by adding a single *absorption state* or *sink state*, usually denoted $\perp$, and having undefined transitions lead to this state. A state $q \in Q$ is called *unreachable* if and only if its left language is empty. A DFA without unreachable states is called *reduced*. For simplicity, we assume DFAs to be both reduced and complete from here on.

Many of the notations and definitions we use are related to regular tree language theory. To a large extent they are straightforward generalizations of ones familiar from regular string language theory. Readers may want to consult e.g. [5,6,8,9] for more detail.

Let Σ be an alphabet, and $r \in \Sigma \to \mathbb{N}$. Pair (Σ, r) is a *ranked alphabet*, r is a *ranking function*, and for all $a \in \Sigma$, $r(a)$ is called the *rank* or *arity* of a. We use Σ_n for $0 \leq n$ to indicate the subset of Σ of symbols with arity n. In algorithms and predicates, we will use n to indicate the rank or arity of a symbol a.

Given a ranked alphabet (Σ, r), the set of *ordered, ranked trees* over this alphabet, set $Tr(\Sigma, r)$, is the smallest set satisfying $a \in Tr(\Sigma, r)$ for all $a \in \Sigma_0$ and $a(t_1, \ldots, t_n) \in Tr(\Sigma, r)$ for all $t_1, \ldots, t_n \in Tr(\Sigma, r), a \in \Sigma$ such that $r(a) = n \neq 0$. Such trees can trivially be presented as rooted, connected, directed, acyclic graphs in which each node has at most one incoming edge. Nodes labeled by symbols of rank 0 are called *leaf nodes* or *leaves*; the sequence of leafs of a tree is called its *frontier*.

In one common view on processing of a tree by tree automata (TAs), each tree node is annotated with a state. For each node labeled by a symbol a (of rank n), state q_0 and states $q_1, \cdots, q_n$ may be assigned to that node and its direct subnodes respectively if the tuple $(q_0, (q_1, \cdots q_n))$ is in the transition relation of symbol a. Note that this simplifies to $(q_0, ())$ for $n = 0$. A tree is *accepted* by a TA if and only if it can be consistently annotated such that the state assigned to the root is a so-called *root accepting* or final state.

By considering transitions of TAs to be directed Frontier-to-Root, we obtain the nondeterministic εNFRTAs; the deterministic DFRTAs are obtained by further restricting the automata to have no ε-transitions and by restricting the transition relations to be (partial) functions, i.e. for every state tuple and symbol yielding (at most) one state. This motivates the following definition:

Definition 2. *A* DFRTA *is a 5-tuple $(Q, \Sigma, r, R, Q_{ra})$ such that Q is a finite set, the* state set*; (Σ, r) is a ranked alphabet; $R = \{R_a | a \in \Sigma\}$ is the set of transition functions (where for all $a \in \Sigma$ with $r(a) = n$ we have $R_a \in Q^n \nrightarrow Q$); and $Q_{ra} \subseteq Q$ is the set of* root accepting *or* final states*.*

Compared to DFAs on strings, two main differences appear: DFRTAs have no start states, and each transition on a symbol (of rank or arity n) relates an n-tuple of states to a state, instead of relating a single state to a state. Note that we will sometimes refer to Q_{ra} as F.

Just as the extension, δ^*, of a DFA's transition function δ (δ yields the state reached after processing a single symbol) yields the state reached after processing a string, for a DFRTA we can define a function RSt yielding the state reached after processing a tree, i.e. the state assigned to the root node of such a tree. It is defined inductively by $RSt(a) = R_a()$ for $a \in \Sigma_0$ and $RSt(a(t_1, \dots, t_n)) = R_a(RSt(t_1), \dots, RSt(t_n))$ for $t_1, \dots, t_n \in Tr(\Sigma, r), a \in \Sigma$ such that $r(a) = n \neq 0$.

Using this definition, we can define the language accepted at state q in a DFRTA M as $\mathcal{L}^{\downarrow}_M(q) = \{t \in Tr(\Sigma, r) | RSt(t) = q\}$. The language accepted by the DFRTA then is simply the language accepted at either of its final or root accepting states: $\mathcal{L}_M = \bigcup_{q \in Q_{ra}} \mathcal{L}^{\downarrow}_M(q)$. If M is clear from the context, we simply write $\mathcal{L}$ instead of $\mathcal{L}_M$. Note that $\mathcal{L}^{\downarrow}_M(q)$ is analogous to the left language of a DFA state [7]; for obvious reasons, we will therefore call it the *down language* of state q. Likewise, we define the *up language* of a state, a notion similar to the right language of a DFA state [7]: $\mathcal{L}^{\uparrow}_M(q) = \{t \in Tr(\Sigma', r') | RSt(t \cdot_{\#} s) \in F \textit{ for all } s \in \mathcal{L}^{\downarrow}_M(q)\}$ where t has a single leaf labeled $\#$ (and Σ' and r' are obtained by extending Σ and r with this symbol of arity 0) and $t \cdot_{\#} s$ denotes the tree obtained from t by substituting tree s for this leaf.

We call a DFRTA *complete*, similar to the notion of completeness for DFAs, if and only if the R_a are total functions; a DFRTA that is not complete can always be made complete by adding a sink state and transitions to it, as is the case for DFAs. A DFRTA state q is *unreachable* if and only if it can never be assigned to the root of any tree by a computation of the DFRTA—i.e. if and only if its down language $\mathcal{L}^{\downarrow}_M$ is empty. Like DFAs, DFRTAs are *reduced* if and only if they contain no unreachable states. From here on, we assume DFRTAs to be both reduced and complete.

Finally, the size of a DFA or DFRTA is defined as $|Q|$, i.e. the size of its state set.

3 Equivalence and minimality

We use *Equiv* as a predicate on two DFA states, defined for all $p, q \in Q$ by

$$Equiv(p, q) \equiv (\overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q)).$$

and as a predicate on two DFRTA states, defined for all $p, q \in Q$ by

$$Equiv(p, q) \equiv (\mathcal{L}^{\uparrow}(p) = \mathcal{L}^{\uparrow}(q)).$$

Thus DFA (DFRTA) states are equivalent if and only if their right languages (up languages) are the same. Using an inductive definition of $\overrightarrow{\mathcal{L}}$, $Equiv(p, q)$ for DFA states can easily be defined recursively [13] as

$$(p \in F \equiv q \in F) \wedge \left\langle \forall\, a : a \in \Sigma : Equiv(\delta(p, a), \delta(q, a)) \right\rangle .$$

Similarly, $Equiv(p, q)$ can be defined recursively for DFRTA states. Before doing so, we introduce two abbreviations:

- $\overrightarrow{\rho}_{i:s}$ is used to abbreviate $(\rho_1, \dots, \rho_{i-1}, s, \rho_{i+1}, \dots, \rho_n)$ (given $\overrightarrow{\rho} = (\rho_1, \dots, \rho_n)$);
- Predicate $P(a, i, \overrightarrow{\rho})$ is defined as $a \in \Sigma \wedge 1 \leq i \leq n \wedge \overrightarrow{\rho} \in Q^n$.

Using these abbreviations $Equiv(p, q)$ can be defined recursively for DFRTA states as

$$(p \in Q_{ra} \equiv q \in Q_{ra}) \wedge \left\langle \forall\, a, i, \vec{\rho} \colon P(a, i, \vec{\rho}) : Equiv(R_a(\vec{\rho}_{i:p}), R_a(\vec{\rho}_{i:q})) \right\rangle$$

(proof omitted and similar to the string case, albeit slightly more complicated due to the generalisation from string right languages to tree up languages). In other words, two DFRTA states are equal if and only if they are both final or non-final, and for each alphabet symbol and each two state tuples that are identical except for the appearance of p and q in corresponding positions, the transitions from these two tuples on the symbol lead to equivalent states.

The usual definition of minimality of a finite automaton (whether on strings or on trees), is that no language equivalent automaton with fewer states exists. Using the definition of up language for DFRTA states (respectively right language for DFA states), minimality can also be written as a predicate

$$\left\langle \forall\, p, q \in Q : p \neq q : \neg Equiv(p, q) \right\rangle .$$

For any two states p, q (such that $p \neq q$), if $Equiv(p, q)$ holds, they can be merged, i.e. one of them can be eliminated in favor of the other (while redirecting in-transitions to the eliminated state to the equivalent remaining one). Eventually, the resulting automaton will be the minimal one recognizing the same language as the original one. (Note that this minimal DFA or DFRTA is unique up to isomorphism). We do not address this reduction step in this paper, but focus on the computation of *Equiv* in two essentially different ways.

4 The classical minimization approach

In the classical approach, the computation of *Equiv* starts out from two initial partitions, corresponding to F and $Q\backslash F$. This is refined iteratively until the greatest fixed-point is reached. The resulting partitioning corresponds to the state set of the minimal automaton equivalent to the original one. Put differently, the algorithms compute the greatest fixed point from the top i.e. from the unsafe side [13].

Classically, minimization algorithms may in fact be based on computing the distinguishability relation between states instead of the equivalence relation, or on computing the partition induced on states by the equivalence relation, or some combination of the three. One variant of the classical minimization approach *for the case of* DFA*s* is presented in [12, Algorithm 7.18]. It uses layerwise computation of *Equiv* (called E there) and of its negation $\neg Equiv$ (called D there, and not included in our presentation). We slightly adapt that algorithm to our notation here:

Algorithm 3 (Layerwise computation of *Equiv* for DFAs)

$H := (F \times F) \cup ((Q \backslash F) \times (Q \backslash F));$
$H_{old} := Q \times Q;$
$\{$ invariant: $H \supseteq Equiv \}$
do $H \neq H_{old} \rightarrow$
  $\{ H \neq H_{old} \}$
  $H_{old} := H;$
  for $(p,q) : (p,q) \in H_{old} \rightarrow$
    as $\langle \exists\, a : a \in \Sigma : (\delta(p,a), \delta(q,a)) \notin H_{old} \rangle \rightarrow H := H \backslash (p,q)$ **sa**
  rof
od$\{ H = Equiv \}$

As pointed out by Watson, without the computation of $\neg Equiv$ (i.e. D), this is essentially Wood's algorithm for computing minimal DFAs [14, p. 132], with Wood stating it is based on Moore's 1950s work [10].

A version of this algorithm *for the case of* DFRTA*s* is presented below. It essentially corresponds to the approach in [6, Section 1.5].[1] The resulting equivalence relation *Equiv* (called P there) is then used to determine the induced equivalence classes and construct the corresponding minimal DFRTA.

Algorithm 4 (Layerwise computation of *Equiv* for DFRTAs)

$H := (F \times F) \cup ((Q \backslash F) \times (Q \backslash F));$
$H_{old} := Q \times Q;$
$\{$ invariant: $H \supseteq Equiv \}$
do $H \neq H_{old} \rightarrow$
  $\{ H \neq H_{old} \}$
  $H_{old} := H;$
  for $(p,q) : (p,q) \in H_{old} \rightarrow$
    as $\left\langle \exists\, a, i, \vec{\rho} : P(a,i,\vec{\rho}) : (R_a(\vec{\rho}_{i:p}), R_a(\vec{\rho}_{i:q})) \notin H_{old} \right\rangle \rightarrow H := H \backslash (p,q)$ **sa**
  rof
od$\{ H = Equiv \}$

Even though this algorithm may look rather complicated compared to the one for DFAs, there is only one essential difference: instead of considering for each symbol a and state p and q where their out-transition on this symbol leads, one has to consider this for states p and q within contexts: their occurrence at the same position in two otherwise equal state tuples (cf. the definition of *Equiv* for DFRTA states as given at the end of Section 3).

The classical minimization approach for DFRTAs has been known for decades, with its first description appearing in Brainerd's 1967 PhD thesis [1]. That description is not as explicitly algorithmic as the one given here or in [6, Section 1.5], and in fact, an algorithmic presentation worked out in detail to an implementation level—albeit for the case of DFRTAs on *unranked* trees—did not appear until 2007 [3].

[1] Note however, that the quantification used in [6, Section 1.5] is somewhat imprecise, as it leaves i unbounded.

5 An incremental minimization algorithm

Watson in [12] and most recently Watson & Daciuk in [13] presented an incremental approach to DFA minimization. Their approach results in an algorithm that starts out with a singleton partition for each of the states of the initial DFA and refines this partition by iteratively merging partitions that are shown to be equivalent. The greatest fixed-point reached corresponds to the state set of the minimal automaton equivalent to the original one. Such an algorithm thus computes the fixed point from below i.e. from the safe side. Clearly, intermediate results from such an algorithm can already be used to reduce the original DFA. We provide a first version of this incremental approach for DFRTAs.

From the problem of deciding the structural equivalence of two types, it is known that equivalence of two states can be computed recursively by turning the mutually recursive set of equivalences *Equiv* into a functional program. For cyclic automata, a direct translation from definition to functional program might lead to non-termination. Thus, in addition to two states, the functional program for compute equivalence also takes a third parameter. An invocation $equiv(p, q, \emptyset)$ returns, via the local variable eq, the truth value of $Equiv(p, q)$. The third parameter, S, is used during recursion to capture pairs of states that are assumed to be equivalent until shown otherwise.

The recursion depth can be bounded by the larger of $|Q| - 2$ and 0 without affecting the result [12, Section 7.3.3], and we add a parameter k to function *equiv* to do so. For efficiency reasons, parameter S is made a global variable. We assume that it is initialized to $\emptyset$. When $S = \emptyset$, an invocation $equiv(p, q, (|Q| - 2)\ \mathbf{max}\ 0)$ returns $Equiv(p, q)$; after such an invocation returns, $S = \emptyset$.

Algorithm 5 (Pointwise computation of $Equiv(p, q)$ for DFAs)

---

func $equiv(p, q, k) =$
$[\![$ **if** $k = 0 \rightarrow eq := (p \in F \equiv q \in F)$
  $[\!]$ $k \neq 0 \wedge \{p, q\} \in S \rightarrow eq := true$
  $[\!]$ $k \neq 0 \wedge \{p, q\} \notin S \rightarrow$
    $eq := (p \in F \equiv q \in F)$;
    $S := S \cup \{\{p, q\}\}$;
    for $a : a \in \Sigma \rightarrow$
      $eq := eq \wedge equiv(\delta(p, a), \delta(q, a), k - 1)$
    rof;
    $S := S \backslash \{\{p, q\}\}$
  fi;
  return eq
$]\!]\{\ equiv(p, q, k) \equiv Equiv(p, q)\ \}$

Function *equiv* can be used to compute relation *Equiv*. To do so, we maintain set G (H) consisting of pairs of states known to be distinguishable i.e. belonging to $\neg Equiv$ (equivalent i.e. belonging to *Equiv*). To initialize both sets, we note that final states are never equivalent to non-final ones, and that a state is always equivalent to itself. Since *Equiv* is an equivalence relation, we ensure that H is transitive at each step of the algorithm. Finally, we have a global variable S as used by function *equiv*:

Algorithm 6 (Incremental computation of *Equiv*)

$S, G, H := \emptyset, ((Q\backslash F) \times F) \cup (F \times (Q\backslash F)), \{(q,q) | q \in Q\};$
$\{$ invariant: $G \subseteq \neg Equiv \wedge H \subseteq Equiv$ $\}$
do $(G \cup H) \neq Q \times Q \rightarrow$
  let $p, q : (p,q) \in ((Q \times Q)\backslash(G \cup H));$
  if $equiv(p, q, (|Q| - 2)$ **max** $0) \rightarrow$
    $H := H \cup \{(p,q),(q,p)\};$
    $H := H^+$
  $[\!]$ $\neg equiv(p, q, (|Q| - 2)$ **max** $0) \rightarrow$
    $G := G \cup \{(p,q),(q,p)\};$
  fi
od$\{$ $H = Equiv$ $\}$

The repetition in this algorithm can be interrupted and the partially computed H can be safely used to merge states, leading to a not necessarily minimal but potentially smaller automaton than the original one.

The algorithm is not DFA-specific and as a result can be applied for the DFRTA-case, provided function *equiv* is suitably chosen. Looking at function *equiv* for the DFA case, we see that the update to *eq* in the loop is performed for every out-transition of p and q. For the DFRTA case, the equivalent is to perform the update for every out-transition *involving* p and q, with such out-transitions involving tuples of states that are identical except for an appearance of p and q respectively at the same position:

Algorithm 7 (Pointwise computation of *Equiv*(p,q) for DFRTAs)

func $equiv(p, q, k) =$
$[\![$ **if** $k = 0 \rightarrow eq := (p \in F \equiv q \in F)$
  $[\!]$ $k \neq 0 \wedge \{p,q\} \in S \rightarrow eq := true$
  $[\!]$ $k \neq 0 \wedge \{p,q\} \notin S \rightarrow$
    $eq := (p \in F \equiv q \in F);$
    $S := S \cup \{\{p,q\}\};$
    for $a, i, \vec{\rho} : P(a, i, \vec{\rho}) \rightarrow$
      $eq := eq \wedge equiv(R_a(\vec{\rho}_{i:p}), R_a(\vec{\rho}_{i:q}), k-1)$
    rof;
    $S := S\backslash\{\{p,q\}\}$
  fi;
  return eq
$]\!]\{$ $equiv(p,q,k) \equiv Equiv(p,q)$ $\}$

Watson and Daciuk in [13] considered different ways to improve both the theoretical and the practical running time of the algorithm for the DFA case. Furthermore, they showed the resulting efficient implementation to be competitive to implementations of classical minimization algorithms, even though the basic incremental algorithm is known to have a worse theoretical running time complexity. We expect similar results to hold for the DFRTA case and plan to consider these as future work.

6 A CSP specification for incremental DFRTA minimization

In [11, Section 5], Strauss et al. presented a concurrent version of the incremental minimization algorithm for DFAs in the form of a CSP specification. The crucial part of that specification w.r.t. the difference between DFAs and DFRTAs is presented below.[2] It corresponds to the **for**-loop in function *equiv* of Algorithm 5.

$$\begin{aligned}
FanOut_{pq}(S,k) = {|||}_{a\in\Sigma} \\
&\text{(if } (\{\delta(p,a),\delta(q,a)\} \notin S) \text{ then} \\
&\quad (\ Equiv_{\delta(p,a),\delta(q,a)}(S\cup\{(p,q)\},k-1)\ \triangle && (1)\\
&\quad (to_{\delta(p,a),\delta(q,a)}?eq_a \rightarrow (EqSet := EqSet \cup \{eq_a\})\ ) \\
&\text{else } (EqSet := EqSet \cup \{true\})\ ) && (2)
\end{aligned}$$

We refer to [11, Section 5] for details on this and other parts of the specification. Here, we focus on adapting this particular part to the case of DFRTAs. All the other parts of the specification stay the same, just as all the other parts of the sequential algorithm stay the same when generalizing from DFAs to DFRTAs (compare Algorithm 5 to Algorithm 7).

To generalize the specification of $FanOut_{pq}(S,k)$, we merely need to generalize the range of the interleaving operator $|||$ from $a \in \Sigma$ to $P(a,i,\overrightarrow{\rho})$ and replace the $\delta(p,a)$ and $\delta(q,a)$ by $R_a(\overrightarrow{\rho}_{i:p})$ and $R_a(\overrightarrow{\rho}_{i:q})$.

The generalization of the CSP specification from DFAs to DFRTAs is thus rather elegant. As hinted at in [11], a significant advantage of a CSP specification such as the foregoing, is maximally to expose opportunities for parallelization. Expressing the DFRTA minimization algorithm in the suggested CSP format indicates that these opportunities will increase drastically if there are a large number of state tuples (which in turn depends on the ranks of the symbols and the number of states). How one exploits these opportunities will clearly depend on the available hardware configuration. The CSP specification is provided in anticipation of a continuation in the current surge in the chip industry towards increasingly large multi-core processors. Thus, while in some senses the CSP specification is a theoretical result, we believe that it is sufficiently generic to serve as a useful reference point in experimenting with parallel implementations of the DFRTA minimization algorithm.

7 Conclusion

This paper has high-lighted once again that many results from the field of regular string languages generalize to that of regular tree languages. It showed, by way of three minimization algorithms, how this generalization becomes quite transparent and elegant if suitable notation is used.

The first algorithm that was generalized to the DFRTA case was already known, but has been presented here in a style which highlights how the generalization occurs.

The second algorithm generalized to the DFRTA case gives a completely new result, being namely a generalization to ranked trees of the string algorithm presented by Watson and Daciuk, incrementally minimizing a DFRTA. As a result, intermediate results of the algorithm can be used to reduce the initial automaton's size. This makes

[2] The specification has been slightly adapted to the notation used in the current paper.

the algorithm useful in situations where running time is restricted (for example, in real-time applications). The new incremental minimization algorithm for DFRTAs can be further improved, similar to the improvements made for the DFA case in [13, Section 6]. We expect such improvements to lead to better performance in practice, similar to the DFA case. To verify this and to be able to compare the (improved) new algorithm and the one using a classical approach to minimization, both need to be implemented and benchmarked.

In the third instance, we also briefly described how an existing concurrent specification of the incremental DFA minimization algorithm in CSP gives rise to one for the DFRTA case. Once again, the generalization was facilitated by relying on suitably defined notation. While implementations of the concurrent specification could be investigated to see whether the parallelization is efficient in practice on currently available hardware, we consider that its principal value lies in serving as a reference point in deriving parallel implementations on the anticipated massively parallel machines of the future.

References

1. W. S. Brainerd: *Tree Generating Systems and Tree Automata*, PhD thesis, Purdue University, June 1967.
2. W. S. Brainerd: *The minimalization of tree automata*. Information and Control, 13(5) November 1968, pp. 484–491.
3. R. C. Carrasco, J. Daciuk, and M. L. Forcada: *An implementation of deterministic tree automata minimization*, in CIAA, J. Holub and J. Zdárek, eds., vol. 4783 of Lecture Notes in Computer Science, Springer, 2007, pp. 122–129.
4. R. C. Carrasco, J. Daciuk, and M. L. Forcada: *Incremental construction of minimal tree automata*. Algorithmica, 2008.
5. L. G. W. A. Cleophas: *Tree Algorithms: Two Taxonomies and a Toolkit*, PhD thesis, Dept. of Mathematics and Computer Science, Eindhoven University of Technology, April 2008, `http://alexandria.tue.nl/extra2/200810270.pdf`.
6. 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/`.
7. J. Daciuk and R. C. Carrasco: *Perfect hashing with pseudo-minimal bottom-up deterministic tree automata*, in Intelligent Information Systems XVI, Proceedings of the International IIS'08 Conference held in Zakopane, Poland, June 16-18, 2008, M. A. Klopotek, A. Przepiorkowski, S. T. Wierzchon, and K. Trojanowski, eds., Academic Publishing House Exit, Warszawa, 2008, pp. 229–238.
8. J. Engelfriet: *Tree Automata and Tree Grammars*, Lecture Notes DAIMI FN-10, Aarhus University, April 1975.
9. F. Gécseg and M. Steinby: *Tree Automata*, Akadémiai Kiadó, Budapest, 1984.
10. E. F. Moore: *Gedanken experiments on sequential machines*, in Automata Studies, C. E. Shan and J. McCarthy, eds., Princeton University Press, Princeton, NJ, 1956.
11. T. Strauss, D. G. Kourie, and B. W. Watson: *A concurrent specification of an incremental DFA minimisation algorithm*, in Proceedings of the Prague Stringology Conference 2008, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2008, pp. 218–226.
12. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Dept. of Mathematics and Computing Science, Technische Universiteit Eindhoven, September 1995, `http://www.fastar.org/publications/PhD_Watson.pdf`.
13. B. W. Watson and J. Daciuk: *An efficient incremental DFA minimization algorithm*. Natural Language Engineering, 9(1) 2003, pp. 49–64.
14. D. Wood: *Theory of Computation*, Harper & Row, New York, 1987.

Constant-memory Iterative Generation of Special Strings Representing Binary Trees

Sebastian Smyczyński

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

Abstract. The shapes of binary trees can be encoded as permutations having a very special property. These permutations are tree permutations, or equivalently they avoid subwords of the type 231. The generation of binary trees in natural order corresponds to the generation of these special permutations in the lexicographic order. In this paper we use a stringologic approach to the generation of these special permutations: decompositions of essential parts into the subwords having staircase shapes. A given permutation differs from the next one with respect to its tail called here the working suffix. Some new properties of such working suffixes are discovered in the paper and used to design effective algorithms transforming one tree permutation into its successor or predecessor in the lexicographic order. The algorithms use a constant amount of additional memory and they look only at those elements of the permutation which belong to the working suffix. The best-case, average-case and worst-case time complexities of the algorithms are $O(1)$, $O(1)$, and $O(n)$ respectively. The advantages of our stringologic approach are constant time and iterative generation, while other known algorithms are usually recursive or not constant-memory ones.

Keywords: tree permutations, stack-sortable permutations, 231-avoiding permutations, enumeration of binary trees

1 Introduction

The generation in natural order of the shapes of binary trees with n nodes corresponds to the lexicographic generation of all special permutations of elements $1, 2, \ldots, n$. This is easy when done recursively and more technical when done iteratively. Our goal is to do it iteratively and at the same time with small time and small space (constant-memory). The natural order of trees as well as its corresponding tree permutations are defined in [3]. It's worth mentioning that the natural order of binary trees is also called an A-order of binary trees [6] and tree permutations are often referred to as stack-sortable permutations or 231-avoiding permutations [13].

In this paper we explore stringologic approach and consider carefully the structure of subwords of special permutations. We introduce the notion of the working suffix of the permutation and reveal its staircase structure. The working suffix is a concatenation of descending staircases, with bottom points of staircases strictly increasing.

We say that the permutation $p = (p_1, p_2, \ldots, p_n)$ of the integer numbers $1, 2, \ldots, n$ is 231-*avoiding* (or that p avoids the pattern 231) if there are no such indices $1 \leq i_2 < i_3 < i_1 \leq n$ such that $p_{i_1} < p_{i_2} < p_{i_3}$.

In other words the subsequence (a, b, c) matches the pattern 231 iff

$$a \; < \; b \; > \; c \; < \; a.$$

A permutation p avoids the pattern 231 if there is no subsequence of p which matches pattern 231 (see Figure 1).

Sebastian Smyczyński: Constant-memory Iterative Generation of Special Strings Representing Binary Trees, pp. 183–191.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

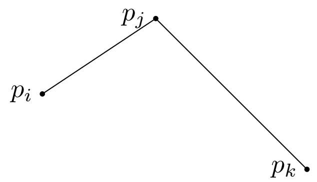


Figure 1. Subsequence (p_i, p_j, p_k) matching the pattern 231. Permutation p is 231-avoiding if there is no subsequence of p matching the pattern 231.

Example 1. The permutation

$$(4, 1, 2, 5, 3, 6, 7)$$

is not a 231-avoiding since the subsequence $4, 5, 3$ matches (in the sense of order) pattern 231, while the permutation

$$(4, 1, 2, 3, 5, 6, 7)$$

avoids the pattern 231.

A *binary tree* T is either a *null tree* or it consists of a node called the *root* and two binary trees denoted $left(T)$ and $right(T)$. Let $|T|$ denote the *size* of T. In the former case, the *size* of T is zero; in the latter case, $|T| = 1 + |left(T)| + |right(T)|$. The *natural order* [3] of binary trees follows the recursive definition:
We say that $T_1 \prec T_2$ if

1. $|T_1| < |T_2|$, or
2. $|T_1| = |T_2|$ and $left(T_1) \prec left(T_2)$, or
3. $|T_1| = |T_2|$ and $left(T_1) = left(T_2)$ and $right(T_1) \prec right(T_2)$,

This order is related to the order relation given by D. E. Knuth in [4, Sec. 2.3.1, excercise 25] specialized to unlabeled binary trees, and is also known as A-order of binary trees [6].

Let T be a binary tree on n nodes. We can represent the tree T as a sequence of the integer numbers $1, 2, \ldots, n$ first labeling the nodes with their position's number as they appear in the inorder traversal of the tree and then listing those labels as they appear in the preorder traversal of the tree. We shall call such a representation *preorder-inorder representation* and the corresponding sequence *tree permutation*.

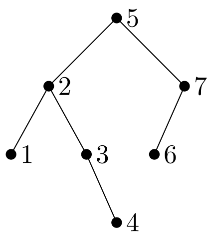


Figure 2. A binary tree T and the string $(5, 2, 1, 3, 4, 7, 6) = preorder(inorder(T))$ representing its shape. $(5, 2, 1, 3, 4, 7, 6)$ is the tree permutation of T.

Interestingly, the natural order of binary trees is preserved by the lexicographic order on their preorder-inorder representation [1] (see Figure 3).

Lemma 2. *For the binary trees $T_1 \prec T_2$ iff the tree permutation of T_1 is lexicographically smaller than tree permutation of T_2.*

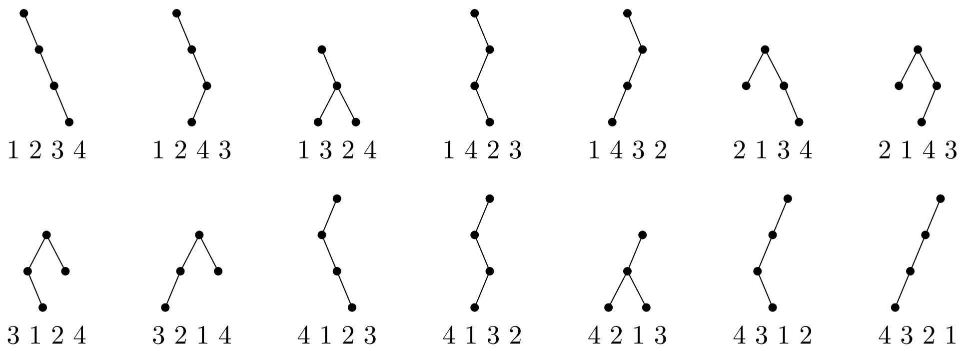


Figure 3. Binary trees for $n = 4$ listed in the lexicographic order on their preorder-inorder representation

The most important property of tree permutations which is employed in this paper is their equivalence with 231-avoiding permutations [1].

Lemma 3. *A permutation p is a tree permutation iff it is* 231*-avoiding.*

Further on we will refer to this property as *the basic property*.

Let $p = (p_1, p_2, \ldots, p_n)$ be a tree permutation. The suffix of p which makes p different from its successor in the lexicographic order is called *working suffix*.

Example 4. For tree permutations for $n = 4$ the working suffixes are underlined as follows:

$$\begin{array}{ccccccc} 1\,2\,\underline{3\,4} & 1\,\underline{2\,4\,3} & 1\,\underline{3\,2\,4} & 1\,4\,\underline{2\,3} & \underline{1\,4\,3\,2} & 2\,1\,\underline{3\,4} & \underline{2\,1\,4\,3} \\ 3\,\underline{1\,2\,4} & \underline{3\,2\,1\,4} & 4\,1\,\underline{2\,3} & 4\,\underline{1\,3\,2} & 4\,\underline{2\,1\,3} & 4\,3\,\underline{1\,2} & 4\,3\,2\,1 \end{array}$$

The *working suffix* for permutation 1234 is 34 since the next permutation is 1243. The *working suffix* for 4321 is empty, since there is no successor for 4321.

We shall call a decreasing sequence of consecutive numbers a *descending stairs sequence* and refer to its first (largest) element as *the top* of the sequence and last (smallest) element as *the bottom* of the sequence. A single number always forms *trivial descending stairs sequence*.

2 Working Suffix Properties

In this section we elaborate on *the basic property* of tree permutations and present a few properties of the working suffix. Those properties in consequence let us construct effective algorithms transforming a given tree permutation into its successor or predecessor in the lexicographic order.

Lemma 5. *Let p be a tree permutation and let i be the index of the first position of its working suffix. If tree permutation q is the successor of p in the lexicographic order, then $q_i = p_i + 1$.*

Proof. Since the suffix $q_i, q_{i+1}, \ldots, q_n$ is a permutation of the working suffix $p_i, p_{i+1}, \ldots, p_n$ then there exists an index $k > i$ such that $q_k = p_i$.

Suppose for the sake of contradiction that $q_i > p_i + 1$. There exists an index j such that $p_j = p_i + 1$. Due to *the basic property* $j > i$ since otherwise we would have three indices $j < i < k$ for which $q_k = p_i < q_j = p_i + 1 < q_i$.

Having such an index $j > i$ for which $p_j = p_i + 1$ we can construct a permutation r by exchanging elements p_j and p_i and then sorting the suffix starting at index $i+1$ in ascending order. The permutation $r = (p_1, p_2, \ldots, p_{i-1}, p_i + 1, r_{i+1}, \ldots, r_n)$ is a valid tree permutation with $r_i = p_i + 1 > p_i$ and $r_{i+1} < r_{i+2} < \cdots < r_n$. Hence $p \prec r$ and $r \prec q$ which contradicts with q being the successor of p. □

Lemma 6. *Let p be a tree permutation and let i be the index of the first position of its working suffix, then there exist no such indices j, k such that $i < j < k$ and $p_k = p_j + 1$.*

Proof. The proof is similar to that of Lemma 5.

Let $q = (q_1, q_2, \ldots, q_{i-1}, q_i, q_{i+1}, \ldots, q_n)$ be the successor of p. Since the working suffix of p starts at index i therefore $q = (p_1, p_2, \ldots, p_{i-1}, q_i, q_{i+1}, \ldots, q_n)$, with $q_i = p_i + 1$.

Suppose for the sake of contradiction that there exist indices $i < j < k$ such that $p_k = p_j + 1$. Then we can construct a new permutation r from p by exchanging elements p_j with p_k and sorting the suffix starting at index $j+1$ in ascending order. The permutation $r = (p_1, p_2, \ldots, p_{j-1}, p_k, r_{k+1}, r_{k+2}, \ldots, r_n)$ with $r_j = p_k > p_j$ and $r_{k+1} < r_{k+2} < \cdots < r_n$ is a valid tree permutation. Hence we have $p \prec r$ and also $r \prec q$ which contradicts with q being the successor of p. □

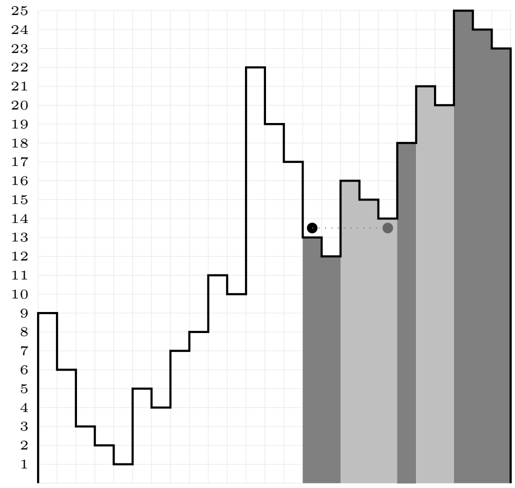
Lemma 7. *Let $p = (p_1, p_2, \ldots, p_n)$ be a tree permutation and i be the starting index of its working suffix. For any index $i \leq k < n$, $p_k > p_{k+1}$ implies that $p_k = p_{k+1} + 1$.*

Proof. Let $q = (q_1, q_2, \ldots, q_{i-1}, q_i, q_{i+1}, \ldots, q_n)$ be the successor of p. Since the working suffix of p starts at index i, therefore $q = (p_1, p_2, \ldots, p_{i-1}, q_i, q_{i+1}, \ldots, q_n)$, with $q_i = p_i + 1$.

Assume that there exists an index k such that $i \leq k < n$ and $p_k > p_{k+1}$. Suppose for the sake of contradiction that $p_k > p_{k+1} + 1$. Let j be an index for which $p_j = p_{k+1} + 1$. Due to *the basic property* $j > k + 1$. We obtain a contradiction since we have indices $i < k + 1 < j$ for which $p_j = p_{k+1} + 1$ which is impossible with respect to Lemma 6. □

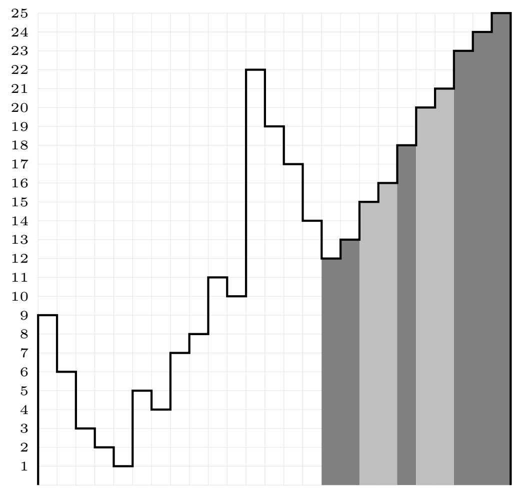
Theorem 8. *For any tree permutation $p = (p_1, p_2, \ldots, p_n)$ its (not empty) working suffix starting at index i forms a staircase of descending stairs sequences (possibly of length 1) for which the top element p_i of the first descending stairs sequence is equal to $p_j - 1$, where j is the index of the bottom step of the following second descending stairs sequence. Furthermore for no other indices $i < k < l$, $p_k = p_l - 1$.*

This theorem is the direct consequence of the previously proven lemmas 5,6,7. See Figure 4 for graphic interpretation of this theorem.



$$p = (9, 6, 3, 2, 1, 5, 4, 7, 8, 11, 10, 22, 19, 17, \mathbf{13}, 12, 16, 15, \mathbf{14}, 18, 21, 20, 25, 24, 23)$$

Figure 4. The tree permutation p represented in a graphic form with shaded parts corresponding to the special staircase structure of the working suffix. The black dot appears above the top of the first sequence of descending stairs and the gray one at the bottom of the following sequence of descending stairs.



$$q = (9, 6, 3, 2, 1, 5, 4, 7, 8, 11, 10, 22, 19, 17, \mathbf{14}, 12, \mathbf{13}, 15, 16, 18, 20, 21, 23, 24, 25)$$

Figure 5. Graphic representation of the tree permutation q, which is the successor of the tree permutation p from Figure 4.

3 The Algorithm

The lemmas 5, 6 presented in the previous section provide us enough information about the structure of the working suffix to design the algorithm transforming given tree permutation p into its successor. The algorithm consists of two steps:

1. Finding the first pair of indices $i < j$ starting from the end of p such that $p_j = p_i + 1$. From lemmas 5 and 6 we know that index i must be then the starting position of the working suffix of p.

2. Transforming the found working suffix by exchanging the elements p_i and p_j and then sorting the suffix starting from position $i+1$ in ascending order.

The theorem 8 on the other hand gives us an exact way how to implement the above mentioned steps effectively by exploiting the staircase structure of the working suffix.

The algorithm $Next(p)$ presented in this section is in fact a direct implementation of the theorem 8.

Algorithm 1: Next(P)

Step 1. *Find the working suffix.*

Let lbs, cbs and cs denote respectively the index of the bottom of the last seen descending stairs sequence, the index of the bottom of the current descending stairs sequence, and the current step index.

```
lbs := n; cbs := n; cs := n;
repeat
    cs := cs − 1;
    If we processed the whole permutation then there is no next one
    if cs < 1 then
        return false;
    end
    Check if we reach the bottom of the previous descending stairs sequence
    if P[cs] < P[cs + 1] then
        lbs := cbs;
        cbs := cs;
    end
until P[lbs] = P[cs] + 1 ;
```

Step 2. *Changing the working suffix to form the successor of P. The cs points at the first element of the working suffix with $P[lbs] = P[cs] + 1$.*

Exchange the elements $P[cs]$ and $P[lbs]$, and then sort the suffix starting at position $cs + 1$ in ascending order (by reversing the stairs).

```
swap(P[cs], P[lbs]);
cs := cs + 1;
while cs < n do
    Let es denote the end of the current descending stairs sequence
    es = cs + 1;
    while es ≤ n and P[es] < P[cs] do
        es := es + 1;
    end
    Change the current descending stairs sequence into increasing sequence
    reverse(P, cs, es − 1);
    cs := es;
end
return true; The permutation P has been transformed to its successor.
```

The work done by the algorithm $Next(p)$ can simply be reverted. During the execution of the algorithm $Next(p)$ the staircase of descending stairs sequences is changed into a staircase of ascending stairs sequences. The element which was at the starting position of the working suffix is the top of the first of ascending stairs sequences, and the element which is placed at the starting position of the working suffix after the execution of the algorithm delimits the new staircase of ascending stairs sequences and simply allows us to find the starting index of the working suffix.

The algorithm $Prev(p)$ transforms the given tree permutation p into its predecessor in the lexicographic order.

Algorithm 2: Prev(P)

Step 1. *Find the working suffix of P predecessor.*

Let cts and cs denote respectively the index of the top of the current ascending stairs sequence, and the current step index.

$cts := n$; $cs := n$;
repeat
  $cs := cs - 1$;
  If we processed whole permutation then there is no previous one
  if $cs < 1$ **then**
    return false;
  end
  Check if we reach the top of the preceding ascending stairs sequence
  if $P[cs] < P[cs+1] - 1$ **then**
    $cts := cs$;
  end
until $P[cs] = P[cts] + 1$;

Step 2. *Changing the working suffix to form the predecessor of P. The cs points at the first element of the working suffix with $P[cs] = P[cts] + 1$.*

Exchange the elements $P[cs]$ and $P[cts]$, and then reverse each ascending stairs sequence in the rest of the working suffix starting at position $cs + 1$.

swap($P[cs]$, $P[cts]$);
$cs := cs + 1$;
while $cs < n$ **do**
  Let es denote the end of the current ascending stairs sequence
  $es = cs + 1$;
  while $es \leq n$ *and* $P[es] = P[es-1] + 1$ **do**
    $es := es + 1$;
  end
  Change the current ascending stairs sequence into descending sequence
  reverse(P, cs, $es - 1$);
  $cs := es$;
end
return true; *The permutation P has been transformed to its predecessor.*

4 Time Complexity

Let us recall that the number of binary trees with n nodes or equivalently the number of the tree permutations of length n is given by the Catalan number [4]

$$C_n = \binom{2n}{n}/(n+1).$$

Let W_n be the sum of lengths of the working suffixes for all tree permutations. Since the algorithm $Next(p)$ performs work proportional to the length of the working suffix of the given permutation p, therefore its average-case time-complexity in enumerating all C_n tree permutations is $O(\frac{W_n}{C_n})$.

Each tree permutation p can be represented as $p = p_1p'p''$ where p' is itself a tree permutation of length $p_1 - 1$ and p'' is a tree permutation of length $n - p_1$ translated by having p_1 added to each element [3] (see Figure 6).

$$p = 8\ \overbrace{4\ \ 1\ \ 3\ \ 2\ \ 6\ \ 5\ \ 7}^{p'}\ \ \overbrace{12\ \ 11\ \ 9\ \ 10}^{\overbrace{4\ \ 3\ \ 1\ \ 2}^{p''}\oplus 8}$$

Figure 6. Example of the decomposition of tree permutation $(8, 4, 1, 3, 2, 6, 5, 7, 12, 11, 9, 10)$.

Using this recursive property of the tree permutations we can formulate the recurrence formula for W_n. If we fix p_1 then p'' is a tree permutation of length $n - p_1$ and p' of length $p_1 - 1$. There are exactly C_{p_1-1} permutations of length $p_1 - 1$. So each working suffix of p'' appears in p exactly C_{p_1-1} times. Therefore the summarized length of all working suffixes of p starting in p'' is equal to $C_{p_1-1}W_{n-p_1}$.

When the working suffix of p starts in p' then its length is equal to the length of the working suffix of p' plus length of p'' which is equal to $n - p_1$. Since the working suffix of p starts in p' each time this permutation is going to be changed (except the last change which will be connected with changing the value of p_1) then the summarized length of working suffixes starting at p' is equal to $W_{p_1-1} + (n - p_1)(C_{i-1} - 1)$.

Now summarizing those values for each possible value of p_1 and adding $n(n - 1)$ for the summarized length of all working suffixes which changes the whole string, we obtain the following recurrence equation:

$$W_n = \sum_{i=1}^{n} \Big(C_{i-1}W_{n-i} + W_{i-1} + (n - i)(C_{i-1} - 1) \Big) + n(n - 1)$$

Solving this recurrence we obtain $W_n = C_{n+1} - n - 1$.

Since the Catalan numbers satisfy the recursive relation $C_1 = 1, C_{n+1} = \frac{2(2n+1)}{n+2}C_n$, we may conclude that the average-case time-complexity of the algorithm $Next(p)$ is constant since $O(\frac{W_n}{C_n}) = O(1)$.

This proves the following theorem:

Theorem 9. (Main Result) *For a tree permutation p we can compute the next tree permutation in the lexicographic order in constant amortized time using only a constant amount of additional memory.*

The algorithm $Prev(p)$ presented in the previous section was obtained by a simple modification of the $Next(p)$ algorithm and similarly performs work proportional to the length of the working suffix of the given permutation p, therefore we obtain a similar result:

Theorem 10. *For a tree permutation p we can compute the previous tree permutation in the lexicographic order in constant amortized time using only a constant amount of additional memory.*

Acknowledgements

The author thanks W. Rytter for the scientific pressure.

References

1. M. C. Er: *On enumerating tree permutations in natural order.* International Journal of Computer Mathematics, 22 1987, pp. 105–115.
2. T. Feil, K. Hutson, and R. M. Kretchmar: *Tree traversals and permutations.* Congressus Numerantium, 172 2005, pp. 201–211.
3. G. D. Knott: *A numbering system for binary trees.* Commun. ACM, 20(2) 1977, pp. 113–115.
4. D. E. Knuth: *The Art of Computer Programming, Volume I: Fundamental Algorithms, 3rd Edition*, Addison-Wesley, 1997.
5. D. E. Knuth: *The Art of Computer Programming, Volume 4: Generating All Trees*, Addison Wesley, 2006.
6. J. Pallo and R. Racca: *A note on generating binary trees in A-order and B-order.* International Journal of Computer Mathematics, 18 1985, pp. 27–39.
7. A. Proskurowski: *On the generation of binary trees.* J. ACM, 27(1) 1980, pp. 1–2.
8. D. Rotem: *Stack sortable permutations.* Discrete Mathematics, 33(2) 1981, pp. 185–196.
9. D. Rotem and Y. L. Varol: *Generation of binary trees from ballot sequences.* J. ACM, 25(3) 1978, pp. 396–404.
10. M. H. Solomon and R. A. Finkel: *A note on enumerating binary trees.* J. ACM, 27(1) 1980, pp. 3–5.
11. R. P. Stanley: *Enumerative Combinatorics, Vol. 2*, vol. 62 of Cambridge Studies in Advanced Mathematics, Cambridge University Press, 1999.
12. J. West: *Generating trees and forbidden subsequences*, in Proceedings of the 6th conference on Formal power series and algebraic combinatorics, Amsterdam, The Netherlands, The Netherlands, 1996, Elsevier Science Publishers B. V., pp. 363–374.
13. J. West: *Permutations with restricted subsequences and stack-sortable permutations*, PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1999.

An input sensitive online algorithm for LCS computation

Heikki Hyyrö

Department of Computer and Information Sciences, University of Tampere, Finland.
heikki.hyyro@cs.uta.fi

Abstract. We consider the classic problem of computing (the length of) the longest common subsequence (LCS) between two strings A and B with lengths m and n, respectively. There are several input sensitive algorithms for this problem, such as the $\mathcal{O}(\sigma n + \min\{Lm, L(n-L)\})$ algorithms by Rick [15] and Goeman and Clausen [5] and the $\mathcal{O}(\sigma n + \min\{\sigma d, Lm\})$ algorithms by Chin and Poon [4] and Rick [15]. Here L is the length of the LCS and d is the number of dominant matches between A and B, and σ is the alphabet size. These algorithms require $O(\sigma n)$ time preprocessing for both A and B. We propose a new fairly simple $\mathcal{O}(\sigma m + \min\{Lm, L(n-L)\})$ time algorithm that works in online manner: It needs to preprocess only A, and it can process B one character at a time, without knowing the whole string B beforehand. The algorithm also adapts well to the linear space[1] scheme of Hirschberg [6] for recovering the LCS, which was not as easy with the above-mentioned algorithms. In addition, our scheme fits well into the context of incremental string comparison [12,10]. The original algorithm of Landau et al. [12] for this problem uses $\mathcal{O}(\sigma m + Lm)$ space. By using our scheme instead, the space usage becomes $\mathcal{O}(\sigma m + \min\{Lm, L(n-L)\})$.

Keywords: string algorithms, longest common subsequences, incremental string comparison

1 Introduction

We use the following conventions and notation in this paper. Σ is a finite alphabet of size σ. Strings are composed of a finite (possibly length-zero) sequence of characters from Σ. The length of a string A is denoted by $|A|$. When $1 \le i \le |A|$, A_i denotes the ith character of A. The notation $A_{i..h}$, where $i \le h$, denotes the substring of A that begins at character A_i and ends at character A_h. Hence $A = A_{1..|A|}$. String A is a subsequence of string B if B can be transformed into A by deleting zero or more characters from it. That is, the characters of A must appear in B in the same order as in A, but they do not need to appear consecutively.

The length of a longest common subsequence (LCS) between two strings is a classic measure of string similarity. Given two strings A and B, we denote the set of their longest common subsequences by $LCS(A,B)$. The length of each longest common subsequence is denoted by $LLCS(A,B)$. For example if A = "string" and B = "writing", then $LCS(A,B) = \{$"ring", "ting"$\}$ and $LLCS(A,B) = 4$. Throughout this paper we use the traditional conventions that m denotes the length of string A, n denotes the length of string B, and $m \le n$.

The problem of LCS/LLCS computation has been studied extensively (see e.g. [3]). Wagner and Fischer [17] have given a basic $\mathcal{O}(mn)$ time LCS algorithm based on dynamic programming. In terms of theoretical results, it has been proven that the time complexity of the LCS problem has a general lower bound of $\Omega(n \log m)$ [8], and

[1] Here, as well as in [5] and [16], the alphabet size σ is assumed to be a constant.

Heikki Hyyrö: An input sensitive online algorithm for LCS computation, pp. 192–202.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

that the quadratic $\mathcal{O}(mn)$ worst case time complexity of the basic dynamic programming algorithm cannot be improved by any algorithm that is based on individual "equal/nonequal" comparisons between characters [1]. Currently the theoretically fastest algorithm in the worst case is the $\mathcal{O}(mn/\log n)$ "Four Russians" algorithm of Masek and Paterson [13].

There are several input sensitive algorithms for the LCS problem whose running times depend on the properties of the input strings. For example the algorithm of Hunt and Szymanski [9] has a running time $O(r \log n)$, where r is the number of matches $A_i = B_j$ over all $i = 1, \ldots, m$ and $j = 1, \ldots, n$. In similar fashion both Chin and Poon [4] and Rick [15] have proposed $\mathcal{O}(\sigma n + \min\{\sigma d, Lm\})$ time algorithms, where d is the number of so-called dominant matches and $L = LLCS(A, B)$. Algorithms whose running time depends on L are typically more efficient than basic dynamic programming either when L is low, like for example the $\mathcal{O}(Ln + n \log n)$ algorithm of Hirschberg [7], or when L is high, like for example the $\mathcal{O}(n(m - L))$ algorithm of Wu et al. [18], but not in both cases simultaneously. Exceptions to this rule are the $\mathcal{O}(\sigma n + \min\{Lm, L(n-L)\})$ algorithms by Rick [15] and later by [5]. Rick's algorithm was the first algorithm that is efficient with both low and high values of L, and it has also been found to be very efficient in practice [3].

The input sensitive algorithms typically rely on a preprocessing phase that is possibly costly. For example the term σn in the two algorithms of Rick [15] and the algorihms of Goeman and Clausen [5] and Chin and Poon [4] comes from the preprocessing phase. It is furher often the case that the preprocessing needs to be done for both A and B before the actual computation. This is true for example in the case of each of the four above mentioned algorithms. This may be significant for example within the setting of one-against-many type of comparison, e.g. when comparing a single pattern string A against each string B in some string database. In such a setting it would be desirable that the preprocessing phase would not need to be repeated for each different string B, that is, if it would be enough to only preprocess the string A once before the comparisons.

In addition to preprocessing only A, a further sometimes desirable property is that the LCS algorithm should work in online manner. By this we mean that the algorithm is able to process the string B one character at a time, without relying on knowledge about the yet unprocessed characters. That is, the algorithm can first read B_1 and compute $LLCS(A, B_1)$, then read B_2 and update the previous solution to correspond to $LLCS(A, B_{1..2})$, and so on until $LLCS(A, B_{1..n})$. This is useful for example if we wish to generate the set of all strings B for which it is true that $LLCS(A, B) \geq \alpha$, where α is some threshold. Such a setting is feasible for example within the context of indexed approximate matching, as proposed by Myers [14][2]. For a pattern (piece) A, the method of Myers generates a set of interesting strings by performing a depth-first search (DFS) over a conceptional trie that contains all possible strings. During the DFS, A is always compared to $B_{1..j}$, the string that corresponds to the current node in the trie. Maintaining this information in an efficient manner essentially requires an online algorithm: when stepping from the node of $B_{1..j}$ to one of its child nodes, which corresponds to some $B_{1..j+1}$, the comparison information about $LLCS(A, B_{1..j})$ should be updated to correspond to $LLCS(A, B_{1..j+1})$.

[2] Myers considered edit distance, but a similar scheme can be used with LCS.

Typically the space complexities of LCS algorithms coincide with their time complexities if we wish to construct a string from the set $LCS(A, B)$[3]. The divide-and-conquer scheme of Hirschberg [6] is a classic method to save space. It can be used with several LCS algorithms in such manner that the value $LLCS(A, B)$ and a string from $LCS(A, B)$ can be computed in linear space while the original asymptotic time complexity of the LCS algorithm is preserved. This space saving scheme is not simple to use with Rick's algorithm. Goeman and Clausen [5] proposed their own $\mathcal{O}(\sigma n + \min\{Lm, L(n-L)\})$ variant of Rick's algorithm and showed how to modify the algorithm to use $\mathcal{O}(\sigma n)$ space, ie. linear space when the alphabet size σ is constant. Their linear space scheme, however, changed the time complexity to $\mathcal{O}(\sigma n + \min\{Lm, m \log m + L(n-L)\})$. Finally, Rick [16] proposed another linear space variant that was able to maintain the $\mathcal{O}(\sigma n + \min\{Lm, L(n-L)\})$ time complexity of his original algorithm.

In this paper we propose an LCS algorithm that has $\mathcal{O}(\sigma m + \min\{Lm, L(n-L)\})$ time complexity, same as the algorithms by Rick [15] and Goeman and Clausen [5]. Our algorithm, however, has some advantages. First of all we find it a bit more simple than the previous two, which may be an important consideration in practice. Perhaps the most significant difference is that our algorithm needs to preprocess only the string A, and it furhermore works in online manner. As discussed above, there are situations where these properties are important. The proposed algorithm is also straight-forward to use within the linear-space divide-and-conquer scheme of Hirschberg while preserving the $\mathcal{O}(\sigma m + \min\{Lm, L(n-L)\})$ time complexity. And as last we mention that the underlying principle behind our algorithm can also be used quite directly within the setting of incremental string comparison [12,10].

2 Preliminaries

2.1 Dynamic programming

The basic $\mathcal{O}(mn)$ dynamic programming solution for the LCS problem is based on filling an $(m+1) \times (n+1)$ dynamic programming matrix D in such manner, that eventually each cell $D[i, j]$ holds the value $D[i, j] = LLCS(A_{1..i}, B_{1..j})$. This can be done using the well-known rules that are shown in Recurrence 1.

Recurrence 1.

$$\text{When } 0 \le i \le m \text{ and } 0 \le j \le n:$$
$$D[i, j] = \begin{cases} 0, \text{ if } i = 0 \text{ or } j = 0, \\ D[i-1, j-1] + 1, \text{ if } A_i = B_j, \text{ and otherwise} \\ \max\{D[i-1, j], D[i, j-1]\}. \end{cases}$$

In the end, the desired LCS length $LLCS(A, B)$ is found in the cell $D[m, n]$. The matrix D is usually filled either in column- or rowwise order. If we are interested only in the value $LLCS(A, B) = D[m, n]$, for example a rowwise filling process needs to store only the currently filled row i and the previous row $i-1$, which means that only linear space is needed. A string in $LCS(A, B)$ can be constructed by backtracking along legal values from the cell $D[m, n]$ to the cell $D[0, 0]$ in the filled matrix D. Any such legal path from $D[m, n]$ to $D[0, 0]$ represents a string in $LCS(A, B)$. The

[3] If only the value $LLCS(A, B)$ is required, most algorithms can be modified to use much less space.

characters of the string are determined in reverse order by the diagonal steps along the path from $D[i,j]$ to $D[i-1,j-1]$ (meaning that $A_i = B_j$ is included in the subsequence). This backtracking process needs $\mathcal{O}(mn)$ space to recover a string in $LCS(A,B)$ as it needs to store the whole matrix D.

2.2 Linear space construction of a longest common subsequence

Hirschberg [6] proposed a divide-and-conquer scheme that can construct a string in $LCS(A,B)$ in linear space. Let $\overleftarrow{A}$ and $\overleftarrow{B}$ denote the reverse strings of A and B. That is, $\overleftarrow{A}_i = A_{m-i+1}$ and $\overleftarrow{B}_j = B_{n-j+1}$ for $1 \le i \le m$ and $1 \le j \le n$. Also let $\overleftarrow{D}$ denote dynamic programming matrix that has been filled using strings $\overleftarrow{A}$ and $\overleftarrow{B}$ instead of A and B. The method finds the middle point of a backtracking path, and then proceeds recursively. The first step is to compute the the values $D[\lfloor \frac{m}{2} \rfloor, j]$ and $\overleftarrow{D}[\lceil \frac{m}{2} \rceil, j]$ for $j = 1, \ldots, n$, where $\lfloor \frac{m}{2} \rfloor$ is a chosen middle row. This information can be computed in $\mathcal{O}(m+n)$ space with rowwise filling order. It can be shown that a backtracking path goes through those cells $D[\lfloor \frac{m}{2} \rfloor, k]$ for which the sum $D[\lfloor \frac{m}{2} \rfloor, k] + \overleftarrow{D}[\lceil \frac{m}{2} \rceil, n-k+1]$ is maximal. Finding one such k takes linear time. After that the recursion proceeds to find the midpoints in the two submatrices that correspond to comparing the string $A_{1..\lfloor \frac{m}{2} \rfloor}$ with $B_{1..k}$ and the string $A_{\lfloor \frac{m}{2} \rfloor+1..m}$ with $B_{k+1..n}$, respectively. The subsequence can be constructed during this process (see [6]). The total work is directly proportional to the total number of cells filled in the dynamic programming matrices. And this number is $\approx \Sigma_{h=0}^{\log_2 m} \frac{1}{2^h} mn < 2mn = \mathcal{O}(mn)$, ie. the total work is at most roughly twice as much as in the basic dynamic programming algorithm.

2.3 Incremental encoding of the dynamic programming matrix

Lemma 1 states well-known properties of adjacent values in D.

Lemma 1. *Let D be a dynamic programming matrix that contains the values $D[i,j] = LLCS(A_{1..i}, B_{1..j})$ for $0 \le i \le m$ and $0 \le j \le n$. Then the following three properties hold for $1 \le i \le m$ and $1 \le j \le n$:*

1. *D[i,j] = D[i-1,j] or D[i,j] = D[i-1,j] + 1*
2. *D[i,j] = D[i,j-1] or D[i,j] = D[i,j-1] + 1*
3. *D[i,j] = D[i-1,j-1] or D[i,j] = D[i-1,j-1] + 1*

Hunt and Szymanski [9] gave an $\mathcal{O}(M \log L + n \log \sigma)$ algorithm that uses these properties. Here M denotes the number of match points between A and B, ie. $M = |\{(i,j) \mid A_i = B_j, 1 \le i \le m, 1 \le j \le n\}|$. Two relevant variants of the algorithm of Hunt And Szymanski are the $\mathcal{O}(\sigma n + Lm)$ algorithm of Apostolico and Guerra [2] and the $\mathcal{O}(M + Lm + n \log \sigma)$ algorithm of Kuo and Cross [11].

All these algorithms represent the dynamic programming matrix D in an incremental manner. When we move from the cell $D[i,j-1]$ to the cell $D[i,j]$, Lemma 1 states that the value of the current cell either remains the same or grows by one. Let us define $R_i[k]$ as the smallest column j where $D[i,j] = k$. Such a column j exists for $k = 0, \ldots, D[i,n]$. It is convenient to define also special sentinel values $R_i[k] = n+1$ for $k > D[i,n]$. Now when $0 \le k \le D[i,n]$, the values $R_i[k]$ represent the values $D[i,j]$ according to the relationship $D[i,j] = k$ for $j = R_i[k], \ldots, R_i[k+1]-1$. In addition, the equality $D[i, R_i[k]] = k = D[i, R_i[k]-1] + 1$ holds when $1 \le k \le D[i,n]$. Due to this latter rule, we may view the values $R_i[k]$ as *increment points*, although this is awkward in the case of the first point $R_i[0] = 0$, which does not have a previous value "$D[i, -1] = -1$" to increment. The left side of Fig. 1 shows an example of increment points.

The values $R_i[k]$ may be computed according to Recurrence 2.

Recurrence 2. (Based on Hunt and Szymanski [9])

$$\text{When } 0 \leq i \leq m:$$

$$R_i[k] = \begin{cases} 0, \text{ if } k = 0, \\ n+1, \text{ if } i = 0 \text{ and } k > 0, \text{ and otherwise} \\ \min\{\ j \mid (A_i = B_j \text{ and } R_{i-1}[k-1] < j < R_{i-1}[k]) \text{ or } j = R_{i-1}[k]\}. \end{cases}$$

Several LCS algorithms use either a *MatchList* or a *NextMatch*[4] auxiliary data structure (see e.g. [12]).

MatchList is a vector of length m, where the entry *MatchList*$[i]$ points to a linked list that contains in sorted order indices j where $A_i = B_j$. This data structure takes overall $\mathcal{O}(m+n)$ space and can be constructed in $\mathcal{O}(m \log \sigma)$ time.

NextMatch is an $n \times \sigma$ matrix. For a given character $a \in \Sigma$, the entry *NextMatch*$[i, a]$ gives smallest k that is larger than i and where $B_k = a$. It is convenient to use $n+1$ as a sentinel value if such k does not exist. So more formally *NextMatch*$[i, a] = \min\{k \mid (k > i \text{ and } B_k = a) \text{ or } k = n+1\}$. *NextMatch* can be constructed in $\mathcal{O}(\sigma n)$ time and space.

The algorithm of Hunt and Szymanski [9] uses the *MatchList* data structure and stores the increment points $R_i[k]$ of row i consecutively and in sorted order in an array. Note that one row has at most $\mathcal{O}(L)$ increment points. The algorithm processes row i after row $i-1$. In order to compute the values $R_i[k]$, the list *MatchList*$[i]$ is processed sequentially. At each match column $j \in$ *MatchList*$[i]$, the algorithm uses an $\mathcal{O}(\log L)$ binary search in the array of the values $R_{i-1}[k]$ to check if the condition $R_{i-1}[k-1] < j \leq R_{i-1}[k]$ of Recurrence 2 holds for some k, and then updates the increment points accordingly. This process takes overall $\mathcal{O}(M \log L + n \log \sigma)$ time, which includes preprocessing the *MatchList* data structure.

The algorithm of Kuo and Cross [11] is quite similar to the algorithm of Hunt and Szymanski. The difference is that the condition $R_{i-1}[k-1] < j \leq R_{i-1}[k]$ of Recurrence 2 is checked at each step while going through the list *MatchList*$[i]$ and a sorted list of values $R_{i-1}[k]$ in parallel. The overall number of steps in this process is limited by the total number of match points and increment points. Since the former number equals M and the latter number is at most Lm, the overall time is $\mathcal{O}(M + Lm + n \log \sigma)$ when also preprocessing *MatchList* is included.

The algorithm of Apostolico and Guerra [2] uses the *NextMatch* matrix. When processing row i, the values $R_{i-1}[k]$ are considered in increasing order. For given $R_{i-1}[k-1]$ and $R_{i-1}[k]$, the existence of j that fulfills the conditions $A_i = B_j$ and $R_{i-1}[k-1] < j \leq R_{i-1}[k]$ of Recurrence 2 can be checked in $\mathcal{O}(1)$ time by consulting the value *NextMatch*$[R_{i-1}[k-1], A_i]$[5]. Now the number of steps is limited only by the number of increment points, and the overall time, including constructing *NextMatch*, is $\mathcal{O}(\sigma n + Lm)$.

---

[4] The *NextMatch* data structure is sometimes called *Closest*.

[5] It can be seen from Recurrence 2 that the condition $R_{i-1}[k-1] < j \leq R_{i-1}[k]$ needs to be checked only once for each pair $R_{i-1}[k-1]$ and $R_{i-1}[k]$.

3 An algorithm using block encoding of the increment points

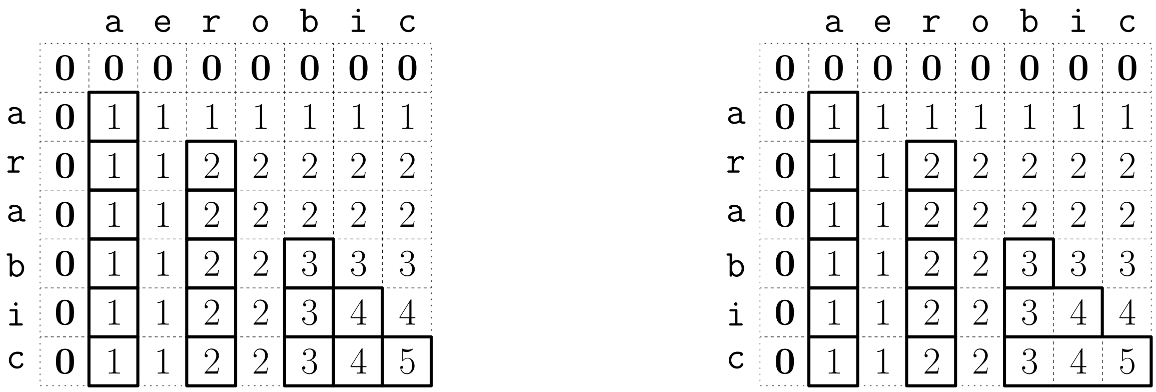


Figure 1. An example with $A =$ "arabic" and $B =$ "aerobic". The left side shows each cell $D[i, j]$, which has an increment point $R_i[k]$, in bold. The right side shows the cells $D[i, j]$ within a block β_i enclosed in a bold rectangle.

We propose to store the increment points $R_i[k]$ using a kind of block encoding. The idea is in some sense similar to the well-known run-length encoding used in data compression. We will define β_i as a list that stores the values $R_i[k]$ of row i using block encoding. Each block item in β_i is a pair of integers (s, e). This value tells that there exists some k for which $R_i[k] = s$ and $R_i[k + h] = s + h$ for $h = 0, \ldots, e - s$. That is, there is a block of consecutive increment points starting from column $s = R_i[k]$ and ending at column $e = R_i[k + e - s]$. We also require that each block (s, e) is maximal: if $k > 0$, then $R_i[k - 1] < s - 1$, and if $k + e - s < D[i, n]$, then $R_i[k + e - s + 1] > e$. The right side of Fig. 1 shows an example.

It is convenient to define that each list β_i has in its end a special sentinel block $(n + 1, n + 1)$. The sentinel delimits the end of row i and does not correspond to any real increment point. The sentinel blocks also never change, and they for example cannot be merged with another block if column n contains an increment point. A constructive definition of the list β_i is as follows:

1. The initial case $k = 0$: Set the pair $(R_i[0], 0) = (0, 0)$ as the first item in β_i.
2. The general case $1 \le k \le D[i, n]$: Let (s, e) be the last item in the list β_i before processing $R_i[k]$.
   a) If $R_i[k] = e + 1$, replace the item (s, e) with $(s, e + 1)$ in the list β_i.
   b) If $R_i[k] > e + 1$, insert the new item $(R_i[k], R_i[k])$ to the end of the list β_i.
3. The sentinel corresponding to $k = D[i, n] + 1$: Insert the pair $(n + 1, n + 1)$ to the end of the list β_i.

From here on we will use the notation (s^i_q, e^i_q) to denote the qth item in the list β_i. Using this notation, a list β_i with r items may be expressed as $\beta_i = ((s^i_1, e^i_1), \ldots, (s^i_r, e^i_r))$. Note that a complete list β_i always has at least two blocks. In addition we will use the notation (s^i_ℓ, e^i_ℓ) to denote the last block in the current, possibly only partially completed list β_i.

Initially at row 0 we know that $\beta_0 = ((0, 0), (n + 1, n + 1))$. Algorithm 1 describes how the list β_i can be constructed from the list β_{i-1}.

Algorithm 1. Assume that we are given the list $\beta_{i-1} = ((s_1^{i-1}, e_1^{i-1}), \ldots, (s_r^{i-1}, e_r^{i-1}))$ that contains r items and represents all increment points $R_{i-1}[k]$ of row $i-1$. Then the list β_i that represents all increment points $R_i[k]$ of row i can be formed correctly by using the following steps:

1. Initially set the list β_i to be empty.
2. The first case $q = 1$:
   Insert the item $(s_1^{i-1}, e_1^{i-1}) = (0, e_1^{i-1})$ to β_i.
3. For $q = 2, \ldots, r$:
   Set $j = \mathit{NextMatch}[e_{q-1}^{i-1}, A_i]$. There are the following subcases:
   a) If $e_{q-1}^{i-1} < j < s_q^{i-1}$, then:
      i) If $j > e_\ell^i$, insert the block (j, j) to the end of β_i.
      ii) If $j = e_\ell^i + 1$, replace (s_ℓ^i, e_ℓ^i) with $(s_\ell^i, e_\ell^i + 1)$ in the list β_i.
      iii) After processing case i or ii, insert the block $(\min\{n+1, s_q^{i-1}+1\}, e_q^{i-1})$ to the end of β_i if $\min\{n+1, s_q^{i-1}+1\} \le e_q^{i-1}$.
   b) If $j \ge s_q^{i-1}$ and $q < r$, insert the block (s_q^{i-1}, e_q^{i-1}) to the end of β_i.

Theorem 2. *Algorithm 1 builds the list β_i correctly.*

Proof. It is not difficult to show that Algorithm 1 follows the principles of Recurrence 2. Fig. 2 illustrates the process.

The case $q = 1$ can be seen to be correct. When $q > 1$ and Algorithm 1 begins processing the block (s_q^{i-1}, e_q^{i-1}), it can be shown that within the column interval $(e_\ell^i, \ldots, e_{q-1}^{i-1})$ and (s_q^{i-1}, e_q^{i-1}), β_i should contain increment points in the column $j = \mathit{NextMatch}[e_{q-1}^{i-1}, A_i]$ and in the columns $j = s_q^{i-1}+1, \ldots, e_q^{i-1}$ (if $s_q^{i-1}+1 \le e_q^{i-1}$). Figs. 2$b$, 2$c$ and 2d, correspond, respectively, to the case 3b, the subcase i of the case 3a, and the subcase ii of the case 3a in Algorithm 1.

Note that when processing the next q, the newly created block (s_ℓ^i, e_ℓ^i) may be appended to contain one more increment point in the subcase ii of the case 3a in Algorithm 1. This is reflected in Figs. 2b - 2d in how the value in column $e_{q-1}^{i-1}+1$ remains undecided.

We omit a more detailed proof from this version of the paper. □

Theorem 3. *The time complexity of Algorithm 1 is $\mathcal{O}(\min\{Lm, L(n-L)\})$.*

Proof. Fig. 3 illustrates the proof. Since the time complexity of Algorithm 1 is directly proportional to the total number of blocks that it processes, we will find an upper bound for the number of the blocks.

We begin by considering some column interval $j = u, \ldots, v$ on row i, where $0 \le u \le v \le n$. Let $\#_i(u, v)$ denote the number of increment points and $\overline{\#}_i(u, v)$ denote the number of non-increment points within these columns. Clearly $\#_i(u, v) + \overline{\#}_i(u, v) = v - u + 1$, since each column j either does or does not contain an increment point $R_i[k]$ for some k.

Since each maximal block of consecutive increment points contains at least one increment point, the number of blocks that appear, even partially, within columns $j = u, \ldots, v$ is bounded by $\#_i(u, v)$. On the other hand, each maximal block is followed by a non-increment point or the end of the considered region. Hence the number of blocks within columns $j = u, \ldots, v$ is bounded also by $\overline{\#}_i(u, v) + 1$.

Let us first analyse the first $z = m - L$ rows. Row i has $\#_i(0, n) = D[i, n]$ increment points, and from Lemma 1 we know that $D[i, n] \le i$. Hence β_i contains at most i

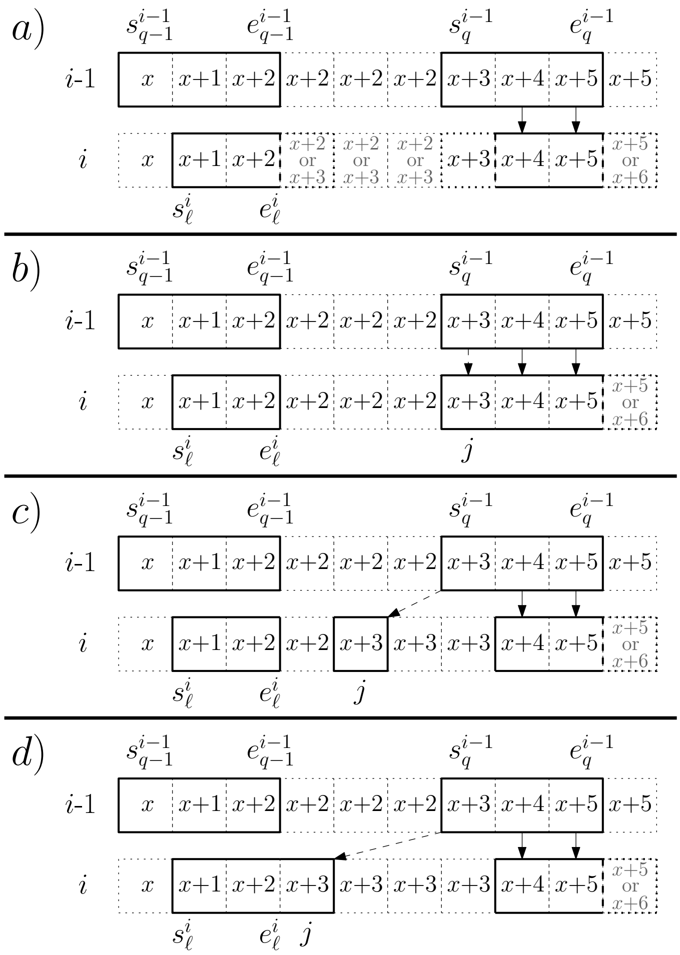


Figure 2. Figures a) - d) illustrate processing the block (s_q^{i-1}, e_q^{i-1}). Here x denotes the value $D[i{-}1, s_q^{i-1}]$, and the bold rectangles enclose blocks of consecutive increment points. The solid arrows show increment blocks that must be inherited to row i in columns $s_q^{i-1}+1, \ldots, e_q^{i-1}$. The dashed arrow shows where the increment point at s_q^{i-1} moves.

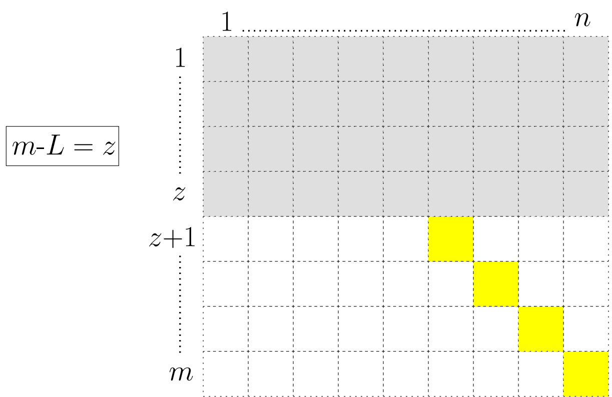


Figure 3. The figure illustrates the time complexity analysis of Algorithm 1.

blocks. This means that the lists β_i for rows $i = 0, \ldots, z$ contain at most a total of $\Sigma_{i=0}^{z} i = \frac{z(z+1)}{2}$ blocks.

Let us then consider the rows $z+i$ for $i = 1, \ldots, L$. Each such row contains at most $\#_{z+i}(0, n) \leq z + i$ increment points. On the other hand, at least i of the increment points must be located within the first $n-L+i$ columns, ie. $\#_i(0, n-L+i) \geq i$. This is because the condition $LLCS(A, B) = L$ requires that $LLCS(A_{1..m-L+i}, B_{1..n-L+i}) = D[z + i, n - L + i] \geq i$ (this is not difficult to prove by using Lemma 1). Now $\overline{\#}_i(0, n - L + i) = n - L + i + 1 - \#_i(0, n - L + i) \leq n - L + 1$, so columns $j = 0, \ldots, n-L+i$ of row $z+i$ contain at most $n-L+1$ blocks. If $i < L$, the remaining columns $j = n - L + i + 1, \ldots, n$ of row i contain at most $\#_{z+i}(n - L + i + 1, n) = \#_{z+i}(0, n) - \#_{z+i}(0, n - L + i) \leq z$ blocks. Hence the total number of blocks on row $z + i$ is bounded by $n - L + 1 + z$.

Based on the preceding discussion, the total number of blocks on all rows $i = 0, \ldots, m$ is at most $\frac{z(z+1)}{2} + L(n - L + 1 + z)$. We note that $\frac{z(z+1)}{2} \leq m(m - L) = \mathcal{O}(m(n - L))$, and that $L(n - L + 1 + z) = L(n - L + 1 + m - L) \leq L(2n - 2L + 1) = \mathcal{O}(L(n - L)) = \mathcal{O}(m(n - L))$.

On the other hand $\#_i(0, n) \leq L$ for all $i = 0, \ldots, m$, and so the total number of blocks has also the bound $\mathcal{O}(Lm)$.

By combining the previous two bounds, we have that the total number of blocks is bounded by $\mathcal{O}(\min\{Lm, m(n - L)\}$. If $n - L < L$, then $\frac{m}{2} \leq \frac{n}{2} < L$, ie. $m = \mathcal{O}(L)$. This implies that $\mathcal{O}(\min\{Lm, m(n - L)\} = \mathcal{O}(\min\{Lm, L(n - L)\}$, since the choice $m(n - L)$ is smaller than Lm only when $m = \mathcal{O}(L)$.

Hence we have reached the conclusion that the asymptotic time complexity of Algorithm 1 may be stated in the form $\mathcal{O}(\min\{Lm, L(n - L)\}$. When also preprocessing of *NextMatch* is taken into account, the time complexity becomes $\mathcal{O}(\sigma n + \min\{Lm, L(n - L)\}$. □

3.1 Constructing a longest common subsequence in $\mathcal{O}(\sigma n)$ space

We briefly sketch how to use Algorithm 1 in the divide-and-conquer scheme discussed in Section 2.2 in order to construct a string from the set $LCS(A, B)$ using $\mathcal{O}(\sigma n + \min\{Lm, L(n - L)\}$ time and $\mathcal{O}(\sigma n)$ space. The required space is determined by *NextMatch* and is linear for constant σ.

Let $\overleftarrow{\beta}_i$ denote the block list for row i of $\overleftarrow{D}$ that corresponds to the reverse strings $\overleftarrow{A}$ and $\overleftarrow{B}$. Algorithm 1 can produce both middle-row lists $\beta_{\lfloor \frac{m}{2} \rfloor}$ and $\overleftarrow{\beta}_{\lceil \frac{m}{2} \rceil}$ in $\mathcal{O}(\min\{Lm, m(n-L)\}$ time. A column k where the sum $D[\lfloor \frac{m}{2} \rfloor, k] + \overleftarrow{D}[\lceil \frac{m}{2} \rceil, n-k+1]$ is maximal can be found in $\mathcal{O}(L)$ time by merging the size-$\mathcal{O}(L)$ lists $\beta_{\lfloor \frac{m}{2} \rfloor}$ and $\overleftarrow{\beta}_{\lceil \frac{m}{2} \rceil}$ in such manner that the other list is processed in reverse order. Overall we may state that the process has an upper bound of $c \min\{Lm, m(n - L)\}$ operations for some constant c.

Then the divide and conquer scheme does the same process for the string-pairs $(A_{1..\lfloor \frac{m}{2} \rfloor}, B_{1..k})$ and $(A_{\lfloor \frac{m}{2} \rfloor+1..m}, B_{k+1..n})$. Handling these takes at most $c(\min\{L_1 \frac{m}{2}, \frac{m}{2}(k - 1 - L_1)\} + c(\min\{L_2 \frac{m}{2}, \frac{m}{2}(n - k + 1 - L_2)\}$ operations, where $L_1 = LLCS((A_{1..\lceil \frac{m}{2} \rceil-1}, B_{1..k-1})$, $L_2 = LLCS(A_{\lceil \frac{m}{2} \rceil+1..m}, B_{k+1..n})$, and $L_1 + L_2 = L$. The sum of minimal choices in these two min-clauses is obviously never larger than a sum of two arbitrary choices. Therefore the overall value is limited above by both $c(L_1 \frac{m}{2} + L_2 \frac{m}{2}) = cL\frac{m}{2} = \frac{1}{2}cLm$ and $c(\frac{m}{2}(k - 1 - L_1) + \frac{m}{2}(n - k + 1 - L_2)) = c(\frac{m}{2}(k - 1 - L_1 + n - k + 1 - L_2)) = \frac{1}{2}cm(n - L)$, which results in the upper bound

$\frac{1}{2}c\min\{Lm, m(n-L)\}$ for the operations done in the second stage. By continuing the same analysis, the result is that the overall number of operations done during the divide-and-conquer scheme has an asymptotic limit $\Sigma_{h=0}^{\log_2 m}\frac{1}{2^h}c\min\{Lm, m(n-L)\} < 2c\min\{Lm, m(n-L)\} = \mathcal{O}(\min\{Lm, m(n-L)\}) = \mathcal{O}(\min\{Lm, L(n-L)\}$. By employing simple index-readjustments, each stage can use the same *NextMatch* built for the complete strings A and B (and a sorresponding $\overleftarrow{NextMatch}$ for $\overleftarrow{A}$ and $\overleftarrow{B}$). Hence the preprocessing needs to be done only once, using $\mathcal{O}(\sigma n)$ time and space.

3.2 A remark on incremental string comparison

Landau et al. [12] proposed an algorithm (that Ishida et al. [10] later extended), which can handle an incremental version of the LCS problem: After computing $LLCS(A, B)$, we should next be able to compute either $LLCS(A, Bb)$ or $LLCS(A, bB)$, ie. a character b may be added to either end of B. The algorithm uses *NextMatch* and maintains all increment points of D that corresponds to comparing the current A and B. We do not go into more details in this paper but just briefly note that our block encoding can be used also in this setting with very few modifications. Then the overall space usage becomes $\mathcal{O}(\sigma n + \min\{Lm, L(n-L)\})$ instead of $\mathcal{O}(\sigma n + Lm)$, the time for computing $LLCS(A, bB)$ after $LLCS(A, B)$ remains the same, and the time for computing $LLCS(A, Bb)$ after $LLCS(A, B)$ becomes $\mathcal{O}(\min\{L, (n-L)\})$ instead of the $\mathcal{O}(L)$ time of the original scheme [10].

References

1. A. V. Aho, D. S. Hirschberg, and J. D. Ullman: *Bounds on the complexity of the longest common subsequence problem*. Journal of the ACM, 23(1) 1976, pp. 1–12.
2. A. Apostolico and C. Guerra: *The longest common subsequence problem revisited*. Algorithmica, 2 1987, pp. 316–336.
3. L. Bergroth, H. Hakonen, and T. Raita: *A survey of longest common subsequence algorithms*, in Proc. 7th String Processing and Information Retrieval (SPIRE 2000), 2000, pp. 39–48.
4. F. Y. L. Chin and C. K. Poon: *A fast algorithm for computing longest common subsequences of small alphabet size*. Journal of Information Processing, 13(4) 1990, pp. 463–469.
5. H. Goeman and C. Clausen: *A new practical linear space algorithm for the longest common subsequence problem*. Kybernetika, 38(1) 2002, pp. 45–66.
6. D. S. Hirschberg: *A linear space algorithm for computing maximal common subsequences*. Communications of the ACM, 18(6) 1975, pp. 341–343.
7. D. S. Hirschberg: *Algorithms for the longest common subsequence problem*. Journal of the ACM, 24(4) 1977, pp. 664–675.
8. D. S. Hirschberg: *An information-theoretic lower bound for the longest common subsequence problem*. Information Processing Letters, 7(1) 1978, pp. 40–41.
9. J. W. Hunt and T. G. Szymanski: *A fast algorithm for computing longest subsequences*. Communications of the ACM, 20(5) 1977, pp. 350–353.
10. Y. Ishida, S. Inenaga, A. Shinohara, and M. Takeda: *Fully incremental lcs computation*, in Proc. 15th Fundamentals of Computation Theory (FCT 2005), vol. 3623 of Lecture Notes in Computer Science, 2005, pp. 563–574.
11. S. Kuo and G. R. Cross: *An improved algorithm to find the length of the longest common subsequence of two strings*. ACM SIGIR Forum, 23(3-4) 1989, pp. 89–99.
12. G. M. Landau, E. W. Myers, and M. Ziv-Ukelson: *Two algorithms for lcs consecutive suffix alignment*. Journal of Computer and System Sciences, 73(7) 2007, pp. 1095–1117.
13. W. J. Masek and M. Paterson: *A faster algorithm for computing string edit distances*. Journal of Computer and System Sciences, 20 1980, pp. 18–31.
14. E. W. Myers: *A sublinear algorithm for approximate keyword searching*. Algorithmica, 12(4) 1994, pp. 345–374.

15. C. Rick: *A new flexible algorithm for the longest common subsequence problem*, in Proc. 6th Combinatorial Pattern Matching (CPM'95), vol. 937 of Lecture Notes in Computer Science, 1995, pp. 340–351.
16. C. Rick: *Simple and fast linear space computation of longest common subsequences.* Information Processing Letters, 75(6) 2000, pp. 275–281.
17. R. A. Wagner and M. J. Fischer: *The string-to-string correction problem.* Journal of the ACM, 21(1) 1974, pp. 168–173.
18. S. Wu, U. Manber, G. Myers, and W. Miller: *An* $\mathcal{O}(NP)$ *sequence comparison algorithm.* Information Processing Letters, 35 1990, pp. 317–323.

Bit-parallel algorithms for computing all the runs in a string

Kazunori Hirashima[1], Hideo Bannai[1], Wataru Matsubara[2], Akira Ishino[2,3], and Ayumi Shinohara[2]

[1] Department of Informatics, Kyushu University,
744 Motooka, Nishiku, Fukuoka 819-0395 Japan.
{kazunori.hirashima,bannai}@inf.kyushu-u.ac.jp
[2] Graduate School of Information Science, Tohoku University,
Aramaki aza Aoba 6-6-05, Aoba-ku, Sendai 980-8579, Japan
{matsubara@shino., ishino@, ayumi@}ecei.tohoku.ac.jp
[3] Presently at Google Japan Inc.

Abstract. We present three bit-parallel algorithms for computing all the runs in a string. The algorithms are very efficient especially for computing all runs of short binary strings, allowing us to run the algorithm for all binary strings of length up to 47 in a few days, using a PC with the help of GPGPU. We also present some related statistics found from the results.

1 Introduction

Repetitions in strings is an important element in the analysis and processing of strings. It was shown in [8] that when considering *maximal repetitions*, or *runs*, the maximum number of runs $\rho(n)$ in a string of length n is $O(n)$. The result leads to a linear time algorithm for computing all the runs in a string. Although no bounds for the constant factor was given, it was conjectured that $\rho(n) < n$.

Recently, there has been steady progress towards proving this conjecture [12,13,3]. The currently known best upper bound[1] is $\rho(n) \leq 1.029n$, obtained by calculations based on the proof technique of [3]. On the other hand, it was shown in [6] that the value $\alpha = \lim_{n\to\infty} \rho(n)/n$ exists, but is never reached. A lower bound on α was first presented in [5], where it was shown that $\alpha \geq \frac{3}{1+\sqrt{5}} \approx 0.927$. Although it was conjectured that this bound is optimal [4], a counter example was shown in [10], giving a new lower bound of 0.944565. The currently known best lower bound is $(11z^2+7z-6)/(11z^2+8z-6) \approx 0.94457571235$, where z is the real root of $z^3 = z+1$. This bound was conjectured for a new series of words in [9], and proved independently for a different series of words in [14]. Whether or not the original conjecture $\rho(n) < n$ of [8] holds, or more importantly, the exact constant $\lim_{n\to\infty} \rho(n)/n$ is still not known. On a related note, the average number of runs in a word of a given length has been completely characterized in [11].

In order to better understand the combinatorial properties of runs in strings, an exhaustive calculation of runs in short strings could be very useful. In previous work [7], the maximum number of runs function was shown for binary strings of length up to 31, together with an example of a string which achieves the maximum number of runs for each length. In this paper, we present several algorithms for computing all the runs in short binary strings using *bit-parallel* techniques.

[1] Presented on a website http://www.csd.uwo.ca/faculty/ilie/runs.html

Kazunori Hirashima, Hideo Bannai, Wataru Matsubara, Akira Ishino, Ayumi Shinohara : Bit-parallel algorithms for computing all the runs in a string, pp. 203–213.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

In [2], a fast suffix array based algorithm for calculating all the runs in a string was presented. However, as the algorithm relies on a Lempel-Ziv factorization, our algorithm is much simpler and efficient for binary strings whose length fits in a computer word. The simple algorithm also allows us to implement a very efficient massively parallelized version using General Purpose Graphics Processor Unit Programming (GPGPU). We successfully compute the maximal number of runs, as well as several other related statistics, for binary strings of length up to 47.

2 Preliminaries

Let Σ be a finite set of symbols, called an *alphabet*. Strings x, y and z are said to be a *prefix*, *substring*, and *suffix* of the string $w = xyz$, respectively. The length of a string w is denoted by $|w|$. The i-th symbol of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of w that begins at position i and ends at position j is denoted by $w[i : j]$ for $1 \leq i \leq j \leq |w|$. A string w has period p if $w[i] = w[i + p]$ for $1 \leq i \leq |w| - p$.

A string u is a *run* of period p if it has period $p \leq |u|/2$. A substring $u = w[i : j]$ of w is a *run in w* if it is a run of some period p and neither $w[i-1 : j]$ nor $w[i : j+1]$ is a run of period p, that means the run is maximal. We denote the run $u = w[i : j]$ in w by the pair $\langle i, j\rangle$ of its begin position i and end position j.

For example, the string `aabaabaaaacaacac` contains the following 7 runs: $\langle 1, 2\rangle = \mathtt{a}^2$, $\langle 4, 5\rangle = \mathtt{a}^2$, $\langle 7, 10\rangle = \mathtt{a}^4$, $\langle 12, 13\rangle = \mathtt{a}^2$, $\langle 13, 16\rangle = (\mathtt{ac})^2$, $\langle 1, 8\rangle = (\mathtt{aab})^{\frac{8}{3}}$, and $\langle 9, 15\rangle = (\mathtt{aac})^{\frac{7}{3}}$. A run in w is called a *prefix run* if it is also a prefix of w. Among the above 7 runs, the prefix runs are $\langle 1, 2\rangle$ and $\langle 1, 8\rangle$.

3 Algorithms

From the definition of run in a string, we have only to consider the periods of length at most $|w|/2$ in order to count all runs in w. In the next subsections, we introduce 3 bit-parallel algorithms for counting all runs in w.

We will use the bitwise operations AND, OR, NOT, XOR, SHIFT_RIGHT, and SHIFT_LEFT, denoted by `&`, `|`, $\sim$, `^`, `>>`, and `<<`, respectively, as in the C language.

3.1 Counting prefix runs

Let us begin by counting all prefix runs in a given string. Table 1 shows the continuations of each period in the string w = `aabaabaaaacaacac`, which has two prefix runs $\langle 1, 2\rangle$ and $\langle 1, 8\rangle$. In the table, the value at row p and column j is 1 if and only if p is a period of prefix $w[1 : j]$. The cell (p, j) is shadowed if $2p < j$, and is said to be in the *active area*. In each row, the first (leftmost) position where the period discontinued is emphasized by displaying **0** in bold face. If its position (p, j) is in the active area, it implies that the prefix $u = w[1 : j - 1]$ becomes a *run* of period p since $p \leq |u|/2$. Moreover, if any period continues to the end (rightmost), it means that the whole string w itself is a (prefix) run. In the example, $\langle 1, 16\rangle$ is not a prefix run since no period continues to the end.

We will efficiently compute the table by representing a column as a bit vector named *alive*, and keep tracking it by clever bit operations in the spirit of *bit parallelism* [1,16]. We first represent the occurrences of each character in the string w as

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
period	a	a	b	a	a	b	a	a	a	a	c	a	a	c	a	c
1	1	1	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	1	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	1	1	1	1	1	1	1	**0**	0	0	0	0	0	0	0
4	1	1	1	1	1	**0**	0	0	0	0	0	0	0	0	0	0
5	1	1	1	1	1	**0**	0	0	0	0	0	0	0	0	0	0
6	1	1	1	1	1	1	1	1	**0**	0	0	0	0	0	0	0
7	1	1	1	1	1	1	1	1	1	**0**	0	0	0	0	0	0
8	1	1	1	1	1	1	1	1	1	**0**	0	0	0	0	0	0

Table 1. Continuations of each period in the string `aabaabaaaacaacac`.

$w[1] =$ `a`		
	1 1 1 1 1 1 1 1	alive
$w[2] =$ `a`	& *1 1 1 1 1 1 1 1*	$bitmask_2$
	1 1 1 1 1 1 1 1	alive
$w[3] =$ `b`	& *1 1 1 1 1 1* 0 0	$bitmask_3$
	1 1 1 1 1 1 **0 0**	alive
$w[4] =$ `a`	& *1 1 1 1 1* 1 1 0	$bitmask_4$
	1 1 1 1 1 1 0 **0**	alive
$w[5] =$ `a`	& *1 1 1 1* 1 1 0 1	$bitmask_5$
	1 1 1 1 1 1 0 0	alive
$w[6] =$ `b`	& *1 1 1* 0 0 1 0 0	$bitmask_6$
	1 1 1 **0 0** 1 0 0	alive
⋮	⋮	⋮

Occurrences of each character in w.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
occ	a	a	b	a	a	b	a	a	a	a	c	a	a	c	a	c
occ[a]	1	1	0	1	1	0	1	1	1	1	0	1	1	0	1	0
occ[b]	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
occ[c]	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1

Figure 1. Bit operations to compute the period continuations, for the string $w =$ `aabaabaaaacaacac` (left). The representation here is rotated 90° compared with Table 1 to show the bit representation horizontally. The active area is shadowed in the same way. The italic *1*'s in the prefix part of the bitmasks are always 1 regardless of the string w. Remark that the essential part of the bitmask can be directly derived from the occurrence table (right) of each character in w. For example, in $bitmask_5$, the essential part 1101 equals to $occ[\mathtt{a}][1:4]$ since $w[5] = \mathtt{a}$, and 00100 in $bitmask_6$ is equal to $occ[\mathtt{b}][1:5]$ since $w[6] = \mathtt{b}$.

a bit vector $occ[]$, as shown in Fig. 1 (right), where $occ[c][i] = 1$ if $w[i] = c$, and 0 otherwise, for any $c \in \Sigma$. The bit vector will be used to generate bitmasks to compute the next *alive* by a logical AND operation demonstrated in Fig. 1 (left). The desired $bitmask_i$ for $i \geq 2$ is obtained by $occ[c][1 : i-1]$ where $c = w[i]$, and filling sufficient numbers of preceding *1*'s. The initial value of *alive* is $\sim 0 = 11\ldots1$. When the length of w is at most twice the word size of the computer, each bit vectors *alive* and *bitmask* fit in a single register, and can be processed very efficiently. Whenever the value of *alive* becomes 0, (in the current example, at $w[11]$) we can immediately quit the computation since no bit in *alive* can turn from 0 into 1 by AND operations.

We now turn our attention to count other runs that are not a prefix of the given string w. In principle, we would use the above procedure at each starting positions for $2 \leq i \leq |w|-1$. However, a little care must be taken to avoid duplicated counting. Let us consider the string $v = w[3:16] =$ `baabaaaacaacac` which is a suffix of w starting at position 3. Although $\langle 1, 6\rangle$ is a prefix run in v, it does *not* immediately imply that $\langle 3, 8\rangle$ is a run in w, since it is properly included in the run $\langle 1, 8\rangle$ in w. How to avoid duplicated counting of runs *effectively* is the main subject of this algorithm, as well as the subsequent two algorithms. To solve this problem, we focus on the fact that the character a $= w[2]$, that is the left neighbor of the starting position 3, appears in $v = w[3:16]$ at 2, 3, 5, 6, 7, and 8. (The occurrences at 10, 11 and 13 are not important for the purpose.) Even if v has a prefix run of period either $p = 2, 3, 5, 6, 7,$ or 8, it *never* becomes a run in w since it continues to the left at least one position. Therefore we have only to consider the periods either 1 or 4 (up to 8). In our bit vector implementation, we have only to initialize *alive* = 00001001. The bit vector can be easily obtained by the complement of reversal of occ[a]$[3:10]$ = 01101111. Since the reversal operation is required at each starting positions, we compute the bit vectors both $occ[c]$ and its reversal $occ\_reversal[c]$ for each $c \in \Sigma$ in the pre-processing phase, given string w.

Algorithm 1: counting prefix runs at each starting position

```
Input: w, length: string to count runs, and its length.
Result: number of runs in w.
// construct bit vectors representing the occurrences of each character
foreach c ∈ Σ do
    occ[c] := 0 ;                                   // occurrence bit vector
    occ_reversal[c] := 0 ;                          // reversal of occurrence bit vector
end
for i := 1 to length do
    c := w[i];
    occ[c] := occ[c] | (1 << length − i − 1);
    occ_reversal[c] := occ_reversal[c] | (1 << i);
end
// now count all prefix runs at each beginning position
count := 0;
for begPos := 1 to length − 1 do
    activeArea := 0;
    restLength := length − begPos;
    alive := (1 << (restLength/2)) − 1;
    if begPos > 0 then
        leftChar := w[begPos − 1];
        alive := alive & ((∼occ_reversal[leftChar]) >> begPos);
    end
    for i := 1 to restLength do
        nextChar := w[begPos + i];
        bitmask := ((occ[nextChar] >> (restLength − i)) | (∼0) << i);
        lastAlive := alive;
        alive := alive & bitmask;
        if (lastAlive ^ alive) & activeArea ≠ 0 then
            count++ ;                               // some bit in alive is changed in active area
        end
        if alive = 0 then break ;                   // all runs ended
        if i mod 2 = 1 then
            activeArea := (activeArea << 1) | 1 ;   // widen active area by one
        end
    end
    if alive ≠ 0 then
        count++ ;                                   // the run is continued to the rightmost position
    end
end
return count
```

The full description of the algorithm is in Algorithm 1. The correctness of the algorithm can be verified based on the above mentioned facts. If the length n of the given string is at most the word size, the running time is $O(n^2)$ with $O(|\Sigma|)$ space.

The time complexity for general n is $O(n^3/m)$, where m is the length of the machine word.

3.2 Efficient algorithms for binary strings

In this section, we take another approach to efficiently count the number of runs for binary strings, given as bit vectors.

We assume that the length of the string does not exceed the word size. For a binary string $w \in \{\mathtt{0}, \mathtt{1}\}^+$ of length n and an integer $p \leq n/2$, we denote by $\alpha(w, p)$ the bit vector whose i-th bit is defined by

$$\alpha(w,p)[i] = \begin{cases} 1 & (w[i-p] = w[i]), \\ 0 & (i \leq p \text{ or } w[i-p] \neq w[i]). \end{cases}$$

For instance, Table 2 shows $\alpha(w, p)$ for $w = \mathtt{1111010101001001}$, who has 5 runs $\langle 1, 4\rangle$, $\langle 11, 12\rangle$, $\langle 14, 15\rangle$, $\langle 4, 11\rangle$, and $\langle 9, 16\rangle$.

p \ w	1	1	1	1	0	1	0	1	0	1	0	0	1	0	0	1
1	*0*	1	1	1	0	0	0	0	0	0	0	1	0	0	1	0
2	*0*	*0*	1	1	0	1	1	1	1	1	1	0	0	1	0	0
3	*0*	*0*	*0*	1	0	1	0	0	0	0	0	1	1	1	1	1
4	*0*	*0*	*0*	*0*	0	1	0	1	1	1	1	0	0	0	1	0
5	*0*	*0*	*0*	*0*	*0*	1	0	1	0	0	0	1	1	1	0	0
6	*0*	*0*	*0*	*0*	*0*	*0*	0	1	0	1	1	0	0	0	1	1
7	*0*	*0*	*0*	*0*	*0*	*0*	*0*	1	0	1	0	1	1	1	0	0
8	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	0	1	0	0	0	0	1	1

Table 2. $\alpha(w, p)$ for $w = \mathtt{1111010101001001}$. In each p-th row, consecutive 1's of length at least p is shadowed. The leading p elements at each p-th row are always *0* regardless of w, and shown in italic form.

We note that $\alpha(w, p)$ can be implemented efficiently by the bit operations

$$(w\,\hat{}\,((\sim w)\ \mathtt{>>}\ p))\ \mathtt{\&}\ (1^n\ \mathtt{>>}\ p)$$

Notice that for any bit vector x, x `&` $(1^n$ `>>` $p)$ sets the p leading bits of x to 0. It is easy to see that the following property holds.

Lemma 1. *For any binary string w of length n, the following two conditions are equivalent.*

1. *$\langle s, t\rangle$ is a run of period p in w.*
2. *$s+2p \leq t+1$, and $\alpha(w,p)[i] = 1$ for every $s+p \leq i \leq t$, and $\alpha(w,p)[s+p-1] = 0$. Moreover, if $t < n$ then $\alpha(w,p)[t+1] = 0$.*

Since a run must be at least as long as twice its period, $s + 2p \leq t + 1$ holds for any run $\langle s, t\rangle$. Therefore, the lemma states that each run of period p in w corresponds to a stretch of consecutive 1's with length at least p in $\alpha(w, p)$. The problem now is how this can be counted efficiently for each period.

Notice that for any bit vector x, the operation x `&` $(x$ `>>` $1)$ reduces the length of each stretch of consecutive 1's in x by one. Therefore, we can detect stretches of consecutive 1's of length at least p in $\alpha(w,p)$ by counting the number of stretches of 1's in $x = \alpha(w,p)$ `&` $(\alpha(w,p)$ `>>` $1)$ `&` $\cdots$ `&` $(\alpha(w,p)$ `>>` $(p-1))$. It is not difficult to see that calculating x can be done with $O(\log p)$ operations, as shown by `selfAND` in Algorithm 2. Further, the number c of stretches of 1's in a bit vector can be computed in $O(c)$ steps as shown by `oneRuns` in Algorithm 2. Details and alternate implementations for these operations can be found in [15].

However, as with counting prefix runs, we must be careful not to count the same run multiple times. This could happen, for example, when a run is longer than 4 times its minimal period p, and a run would be detected for period p and period $2p$. For example, a run `abababab` with period 2 (`ab`) will also be counted as a run at period 4 (`abab`). Below, we consider two methods for removing such duplicates in the process.

Algorithm 2: Common subroutines

```
selfAND(v,k): calculate v = v&(v>>1)&···&(v>>(k − 1)) (with fewer steps)
begin
    while k > 1 do  s = k >> 1; v & = (v >> s); k −= s;
    return v;
end
oneRuns(v): count the number of stretch of 1's in v
begin
    c:= 0;
    while v do
        v & = (v | (v − 1)) + 1 ;          // remove rightmost stretch of consecutive 1's
        c ++;
    end
    return c;
end
```

Removing duplicate runs by position The first approach utilizes the fact that runs with different minimal periods cannot begin and end at the same positions. Therefore, for each beginning position of a run, we use a bit vector to mark its end position. This way, we can check if we have already considered the run via a different period. This can be implemented efficiently using bit operations as shown in Algorithm 3. The time complexity is $O(n^3/m)$, where m is the length of the machine word. Note that `lsb(x)` computes the least significant set bit of `x`, and can be computed in $O(1)$ for a machine word, or $O(n/m)$ time for general bit strings.

Removing duplicate runs by sieve The second approach to eliminate duplicate counting is based on the following observation.

Lemma 2. *Let $\langle s,t\rangle$ be a run of period p in w. For any k with $2kp \le t-s+1$, the run $\langle s,t\rangle$ is also a run of period kp in w.*

For example, consider again the runs in $w =$ `1111010101001001` (see Table 2). The run $\langle 1,4\rangle =$ `1111` is a run of period both 1 and 2. The run $\langle 4,11\rangle =$ `10101010` is a run of period both 2 and 4.

Therefore, if we count each run only at its minimum period, we can avoid duplications. Our strategy is somewhat similar to the *Sieve of Eratosthenes* to generate prime numbers. From the smallest period $p = 1$ to the maximum possible period $p = |w|/2$ in this order, if a run of period p is found in x, we will sieve out all runs of period kp satisfying the length condition in Lemma 2. The sieving procedure can be implemented by tricky bit operations, shown in Algorithm 4. The time complexity is $O(n^3/m)$, where m is the length of the machine word.

Algorithm 3: Removing duplicate runs by position

```
Input: w, length: bit vector to count runs, and its length.
Result: number of runs in w.
runEndsByBegPos[length − 1] ;                       // array of bitvectors (initialized to 0)
for period := 1 to length/2 do
    v:= (w ^ ((∼w) >> period)) & (1^length >> period) ;          // calculate α(w, period)
    x:= selfAND(v, period) ;
    while x ≠ 0 do
        begPos:= lsb(x) ;                                  // position index of rightmost 1
        y:= x + (1 << begPos) ;                   // if x =...0111100 then y = ...1000000
        x:= x & y ;                                // clear rightmost consecutive 1's in x
        y:= y & (−y) ;                                   // clear all but rightmost 1 in y
        y:= y << ((period − 1) << 1) ;                // convert to actual position in w
        if (runEndsByBegPos[begPos] & y) = 0 then
            // a run starting at begPos doesn't already end here
            count ++;
            runEndsByBegPos[begPos] = runEndsByBegPos[begPos] | y;
        end
    end
end
return count
```

Algorithm 4: Removing duplicates by Sieve

```
Input: w, length: bit vector to count runs, and its length.
Result: number of runs in w.
pvec[length/2 + 1] ;                                         // array to store bitvectors
for period := 1 to length/2 do
    pvec[period] := (w^((∼w) >> period)) & (1^length >> period) ;   // calculate α(w, period)
end
for period := 1 to length/2 do
    x := selfAND(pvec[period], period);
    count := count + oneRuns(x) ;                         // number of runs of this period.
    for p := 2 ∗ period to length/2 step period do
        x := x & (x >> period);
        if x = 0 then break;
        pvec[p] := pvec[p]^x;
    end
end
return count
```

4 Computational Experiments

4.1 Running time

Table 3 compares the running times of the three algorithms. All experiments were conducted on an Apple Mac Pro (Early 2008) with 3.2 GHz dual core Xeons and 18 GB of memory, running MacOSX 10.5 Leopard, using only one thread. Programs were compiled with the Intel C++ compiler 11.0. The algorithms were run on all binary strings of length n for $n = 20, \ldots, 30$. However, only strings ending with 0 are considered, since a complementary binary string will always have runs in the same position. That is, all runs for 2^{n-1} strings are calculated for each n.

For the sieve approach, we further developed a GPGPU version and measured its performance on the same computer. The video card used for GPGPU was NVIDIA GeForce 8800 GT, and the GPGPU environment used was CUDA[2]. Although GPUs

[2] http://www.nvidia.com/object/cuda_home.html

contain many processing units, there are very strict limitations to the resources that each processing unit may use. The simple bit-parallel algorithm presented in this paper only requires a small amount of resources, and is an ideal example for efficient processing on GPUs. We note that the sieve approach using GPGPU was developed to deal with 64 bits, and requires some overhead compared to the other three algorithms that were developed for only 32 bits.

n	20	21	22	23	24	25	26	27	28	29	30
prefix	0.32	0.69	1.49	3.13	8.02	15.6	32.4	66.4	150.2	296.5	625.4
position	0.09	0.18	0.36	0.73	1.49	3.0	6.2	12.6	25.6	52.1	106.0
sieve	0.10	0.18	0.37	0.75	1.50	3.0	6.0	11.9	23.9	48.1	96.7
GPGPU	0.01	0.02	0.04	0.08	0.18	0.4	0.7	1.4	3.0	5.9	12.2

Table 3. Running times in seconds of each algorithm for calculating the runs in all binary strings (excepting complementary strings) of length n.

4.2 Results

Using the GPGPU implementation, we computed the maximum number of runs function $\rho(n)$ for binary strings of length up to $n = 47$, as shown in Table 4. It has been known that $\rho(14) = \rho(13) + 2$. However, as noted in [5], it is not known whether this is an asymptotic property of $\rho(n)$, that is, if there exists infinitely many n for which $\rho(n + 1) = \rho(n) + 2$. To the best of our knowledge this is the second example satisfying this property, namely, $\rho(42) = \rho(41) + 2$, provided that $\rho(n)$ is achieved by binary words.

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
$\rho(n)$	1	1	2	2	3	4	5	5	6	7	8	8	10	10	11	12	13	14	15	15	16	17	18	19	20	21	22

n	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$\rho(n)$	23	24	25	26	27	27	28	29	30	30	31	32	33	35	35	36	37	38	38

Table 4. The maximum number of runs function $\rho(n)$ for binary strings calculated for n up to 47.

Figure 2 plots the maximum number of runs function obtained by our exhaustive computation, the conjectured upper bound ($y = x$) and the current best asymptotic lower bound ($y = 0.94457571235x$).

Although the problem of finding the maximum number of runs function is still difficult, we have found the following empirical characteristics in the distribution of the number of runs in binary strings, which could give insight in further analyses. Let $f(n, r)$ denote the number of binary strings of length n with r runs. Table 5 shows the values of $f(n, r)$ for $n = 2, \ldots, 42$ and $r = 1, \ldots, 4$.

- $f(n, 1) = 20$ for $n \geq 7$.
- for $n \geq 7$,

$$f(n,2) = \begin{cases} 36n - 190 & \text{if } n \text{ is even,} \\ 36n - 186 & \text{if } n \text{ is odd.} \end{cases}$$

Furthermore, $f(n, 2) = f(n - 2, 2) + 72$ for $n \geq 9$.

n	$f(n,1)$	$f(n,2)$	$f(n,3)$	$f(n,4)$
2	2	0	0	0
3	6	0	0	0
4	14	2	0	0
5	18	14	0	0
6	18	38	8	0
7	20	66	38	4
8	20	98	102	34
9	20	138	202	130
10	20	170	376	306
11	20	210	596	682
12	20	242	880	1314
13	20	282	1220	2296
14	20	314	1622	3736
15	20	354	2080	5686
16	20	386	2598	8260
17	20	426	3174	11562
18	20	458	3808	15642
19	20	498	4502	20626
20	20	530	5252	26574
21	20	570	6064	33590
22	20	602	6930	41754
23	20	642	7860	51184
24	20	674	8842	61898
25	20	714	9890	74070
26	20	746	10988	87732
27	20	786	12154	103000
28	20	818	13368	119922
29	20	858	14652	138664
30	20	890	15982	159216
31	20	930	17384	181764
32	20	962	18830	206308
33	20	1002	20350	233012
34	20	1034	21912	261896
35	20	1074	23550	293138
36	20	1106	25228	326696
37	20	1146	26984	362804
38	20	1178	28778	401434
39	20	1218	30652	442762
40	20	1250	32562	486776
41	20	1290	34554	533702
42	20	1322	36580	583470

Table 5. Number of binary strings of length n with r runs.

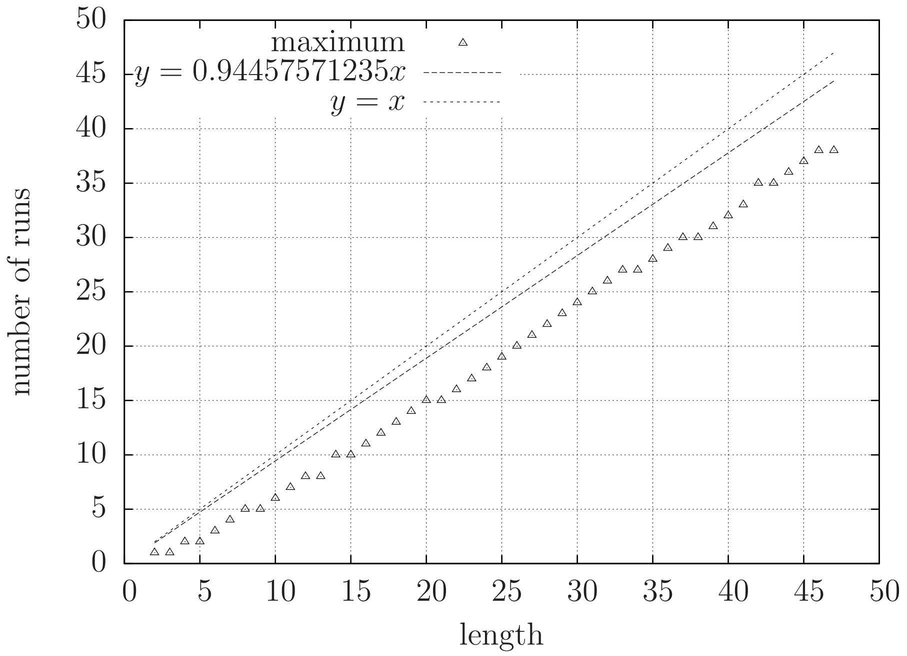


Figure 2. The maximum number of runs in a binary string obtained from exhaustive calculations.

– for $n \geq 12$,

$$f(n,3) = \begin{cases} (117n^2 - 1558n + 5368)/4 & \text{if } n \text{ is even,} \\ (117n^2 - 1556n + 5335)/4 & \text{if } n \text{ is odd.} \end{cases}$$

Furthermore, $f(n,3) = 2f(n-2,3) - f(n-4,3) + 234$ for $n \geq 16$.

We can see that $f(n,1) = 20$ will hold for any $n > 7$, since a binary string with only one run can only be one of $(01)^{n/2}$, $x0^{n-4}y$ for $x, y \in \{00, 01, 10\}$, or their bitwise complements.

5 Discussion and Conclusion

We presented 3 bit-parallel algorithms for computing all the runs in short strings. The two latter algorithms specialized for binary strings are very efficient, while the first algorithm can be used for strings with larger alphabet size at the cost of some efficiency. Through exhaustive computations, the algorithms have enabled us to obtain various statistics concerning runs in strings up to a certain length.

Although it seems that many researchers believe it to be true, it is still unknown whether $\rho(n)$ can always be achieved by a binary string.

References

1. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching*. Comm. ACM, 35(10) 1992, pp. 74–82.
2. G. Chen, S. Puglisi, and W. Smyth: *Fast and practical algorithms for computing all the runs in a string*, in Proc. CPM 2007, vol. 4580 of LNCS, 2007, pp. 307–315.

3. M. Crochemore and L. Ilie: *Maximal repetitions in strings.* J. Comput. Syst. Sci., 74 2008, pp. 796–807.
4. F. Franěk, R. Simpson, and W. Smyth: *The maximum number of runs in a string*, in Proc. 14th Australasian Workshop on Combinatorial Algorithms (AWOCA2003), 2003, pp. 26–35.
5. F. Franěk and Q. Yang: *An asymptotic lower bound for the maximal-number-of-runs function*, in Proc. Prague Stringology Conference (PSC'06), 2006, pp. 3–8.
6. M. Giraud: *Not so many runs in strings*, in Proc. LATA 2008, 2008, pp. 245–252.
7. R. Kolpakov and G. Kucherov: *Maximal repetitions in words or how to find all squares in linear time*, Tech. Rep. Rapport Interne LORIA 98-R-227, Laboratoire Lorrain de Recherche en Informatique et ses Applications, 1998.
8. R. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in Proc. 40th Annual Symposium on Foundations of Computer Science (FOCS'99), 1999, pp. 596–604.
9. W. Matsubara, K. Kusano, H. Bannai, and A. Shinohara: *A series of run-rich strings*, in Proc. LATA 2009, vol. 5457 of LNCS, 2009, pp. 578–587.
10. W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and A. Shinohara: *New lower bounds for the maximum number of runs in a string*, in Proc. Prague Stringology Conference (PSC'08), 2008, pp. 140–145.
11. S. J. Puglisi and J. Simpson: *The expected number of runs in a word.* Australasian Journal of Combinatorics, 42 2008, pp. 45–54.
12. W. Rytter: *The number of runs in a string: Improved analysis of the linear upper bound*, in Proc. 23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS 2006), vol. 3884 of LNCS, 2006, pp. 184–195.
13. W. Rytter: *The number of runs in a string.* Inf. Comput., 205(9) 2007, pp. 1459–1469.
14. J. Simpson: *Modified padovan words and the maximum number of runs in a word.* Australasian Journal of Combinatorics, to appear.
15. H. S. Warren: *Hacker's Delight*, Addison-Wesley Professional, 2002.
16. S. Wu and U. Manber: *Fast text searching allowing errors.* Comm. ACM, 35(10) October 1992, pp. 83–91.

Crochemore's repetitions algorithm revisited – computing runs

Frantisek Franek* and Mei Jiang

Department of Computing & Software
Faculty of Engineering
McMaster University
Hamilton, Ontario
Canada L8S 4K1
franek@mcmaster.ca jiangm5@mcmaster.ca

Abstract. Crochemore's repetitions algorithm introduced in 1981 was the first $O(n \log n)$ algorithm for computing repetitions. Since then, several linear-time worst-case algorithms for computing runs have been introduced. They all follow a similar strategy: first compute the suffix tree or array, then use the suffix tree or array to compute the Lempel-Ziv factorization, then using the Lempel-Ziv factorization compute all the runs. It is conceivable that in practice an extension of Crochemore's repetitions algorithm may outperform the linear-time algorithms, or at least for certain classes of strings. The nature of Crochemore's algorithm lends itself naturally to parallelization, while the linear-time algorithms are not easily conducive to parallelization. For all these reasons it is interesting to explore ways to extend the original Crochemore's repetitions algorithm to compute runs. We present three variants of extending the repetitions algorithm to compute runs: two with a worsen complexity of $O(n(\log n)^2)$, and one with the same complexity as the original algorithm. The three variants are tested for speed of performance and their memory requirements are analyzed. The third variant is tested and analyzed for various memory-saving alterations. The purpose of this research is to identify the best extension of Crochemore's algorithm for further study, comparison with other algorithms, and parallel implementation.

Keywords: repetition, run, string, periodicity, suffix tree, suffix array

1 Introduction

An important structural characteristic of a string over an alphabet is its periodicity. Repetitions (tandem repeats) have always been in the focus of the research into periodicities. The concept of runs that captures maximal fractional repetitions which themselves are not repetitions was introduced by Main [12] as a more succinct notion in comparison to repetitions. The term run was coined by Iliopoulos et al. [8]. It was shown by Crochemore in 1981 that there could be $O(n \log n)$ repetitions in a string of length n and an $O(n \log n)$ time worst-case algorithm was presented [3] (a variant is also described in Chapter 9 of [4]), while Kolpakov and Kucherov proved in 2000 that the number of runs was $O(n)$ [9].

Since then, several linear-time worst-case algorithms have been introduced, all based on linear algorithms for computing suffix trees or suffix arrays. Main [12] showed how to compute the leftmost occurrences of runs from the Lempel-Ziv factorization in linear time, Weiner [14] showed how to compute Lempel-Ziv factorization from a suffix tree in linear time. Finally, in 1997 Farach [6] demonstrated a linear construction

* Supported in part by a research grant from the Natural Sciences and Engineering Research Council of Canada.

of suffix tree. In 2000, Kolpakov and Kucherov [9] showed how to compute all the runs from the leftmost occurrences in linear time. Suffix trees are complicated data structures and Farach construction was not practical to implement for sufficiently large n, so such a linear algorithm for computing runs was more of a theoretical result than a practical algorithm.

In 1993, Manber and Myers [13] introduced suffix arrays as a simpler data structure than suffix trees, but with many similar capabilities. Since then, many researchers showed how to use suffix arrays for most of the tasks suffix trees were used without worsening the time complexity. In 2004, Abouelhoda et al. [1] showed how to compute in linear time the Lempel-Ziv factorization from the extended suffix array. In 2003, several linear time algorithms for computing suffix arrays were introduced (e.g. [10,11]). This paved the way for practical linear-time algorithms to compute runs. Currently,there are several implementations(e.g. Johannes Fischer's, Universität Tübingen, or Kucherov's, CNRS Lille) and the latest, CPS, is described and analyzed in [2].

Though suffix arrays are much simpler data structures than suffix trees, these linear time algorithms for computing runs are rather involved and complex. In comparison, Crochemore's algorithm is simpler and mathematically elegant. It is thus natural to compare their performances. The strategy of Crochemore's algorithm relies on repeated reffinements of classes of equivalence, a process that can be easily parallelized, as each reffinement of a class is independent of the other classes and their reffinements, and so can be performed simultaneously by different processors. The linear algorithms for computing runs are on the other hand not very conducive to parallelization (the major reason is that all linear suffix array constructions rely on recursion). For these reasons we decided to extend the original Crochemore's algorithm based on the most memory efficient implementation by Franek et.al. [4]. In this report we discuss and analyze three possible extensions of [4] for computing runs and their performance testing: two variants with time-complexity of $O(n(\log n)^2)$ and one variant with time-complexity of $O(n \log n)$. Two diffierent menthods to save memory for the third variant are tested and analyzed. The purpose of this study was to identify the best extension of Crochemore's repetitions algorithm to compute runs for comaprison with other runs algorithm and for parallel implementation.

2 Basic notions

Repeat is a collection of repeating substrings of a given string. *Repetition*, or *tandem repeat*, consists of two or more adjacent identical substrings. It is natural to code repetitions as a triple (s, p, e), where s is the *start* or *starting position* of the repetition, p is its *period*, i.e. the length of the repeating substring, and e is its *exponent* (or *power*) indicating how many times the repeating substring is repeated. The repeating substring is referred to as the *generator* of the repetition. More precisely:

Definition 1. *(s, p, e) is a repetition in a string $x[0..n-1]$ if*
$x[s..(s+p-1)] = x[(s+p)..(s+2p-1)] = \cdots = x[(s+(e-1)p)..(s+ep-1)]$.
A repetition (s, p, e) is maximal *if it cannot be extended to the left nor to the right, i.e. (s, p, e) is a repetition in x and $x[(s-p+1)..(s-1)] = x[s..(s+p-1)]$ and*
$x[(s+(e-1)p)..(s+ep-1)] = x[(s+ep)..(s+(e+1)p-1)]$.

In order to make the coding of repetitions more space efficient, the repetitions with generators that are themselves repetitions are not listed; for instance, `aaaa` should be

reported as (0,1,4) just once, there is no need to report (1,2,2) as it is subsumed in (0,1,4).
Thus we require that generator of a repetition be *irreducible*, i.e. not a repetition.

Consider a string `abababa`, there are maximal repetitions (0,2,3) and (1,2,3). But, in fact, it can be viewed as a fractional repetition (0,2,3+$\frac{1}{2}$). This is an idea of a run coded into a quadruple (s, p, e, t), where s, p, and e are the same as for repetitions, while t is the *tail* indicating the length of the last incomplete repeat. For instance, for the above string we can only report one run (0,2,3,1) and it characterizes all the repetitions implicitly. The notion of runs is thus more succinct and more space efficient in comparison with the notion of repetitions. More precisely:

Definition 2. *$x[s..(s+ep+t)]$ is a* run *in a string $x[0..n-1]$ if $x[s..(s+p-1)] = x[(s+p)..(s+2p-1)] = \cdots = x[(s+(e-1)p)..(s+ep-1)]$ and $x[(s+(e-1)p)..(s+(e-1)p+t)] = x[(s+ep)..(s+ep+t)]$, where $0 \leq s < n$ is the* start *or the* starting position *of the run, $1 \leq p < n$ is the* period *of the run, $e \geq 2$ is the* exponent *(or* power*) of the run, and $0 \leq t < p$ is the* tail *of the run. Moreover, it is required that either $s = 0$ or that $x[s-1] \neq x[s+2p-1]$ (in simple terms it means that it cannot be extended to the left) and that $x[s+(ep)+t+1] \neq x[s+(e+1)p+t+1]$ (in simple terms it means that the tail cannot be extended to the right). It is also required, that the generator be irreducible.*

3 A brief description of Crochemore's algorithm

Let $x[0..n-1]$ be a string. We deffine an equivalence $\approx_p$ on positions $\{0, \ldots, n-1\}$ by $i \approx_p j$ if $x[i..i+p-1] = x[j..j+p-1]$. In Fig. 1, the classes of $\approx_p$, $p = 1..8$ are illustrated. For technical reasons, a sentinel symbol \$ is used to denote the end of the input string; it is considered to be the lexicographically smallest character. If $i, i+p$ are in the same class of $\approx_p$ (as illustrated by 5,8 in the class $\{0, 3, 5, 8, 11\}$ on level 3, or 0,5 in class $\{0, 5, 8\}$ on level 5, in Fig. 1) then there is a tandem repeat of period p (thus $x[5..7] = x[8..10] =$`aba` and $x[0..4] = x[5..9] =$`abaab`). Thus the computation of the classes and identiffication of repeats of the same "gap" as the level (period) being computed lay in the heart of Crochemore's algorithm. A naive approach following the scheme of Fig. 1 would lead to an $O(n^2)$ algorithm, as there are potentially $\leq n$ classes on each level and there can be potentially $\leq \frac{n}{2}$ levels.

The first level is computed directly by a simple left-to-right scan of the input string - of course we are assuming that the input alphabet is $\{0, \ldots, n-1\}$, if it is not, in $O(n \log n)$ the alphabet of the input string can be transformed to it.

Each follow-up level is computed from the previous level by refinement of the classes of the previous level (in Fig. 1 indicated by arrows). Once a class decreases to a singleton (as $\{15\}$ on level 1 , or $\{14\}$ on level 2), it is not refined any further. After a level p is computed, the equivalent positions with "gap" are identified, extended to maximum, and reported. Note that the levels do not need to be saved, all we need is a previous level to compute the new level (which will become the previous level in the next round). When all classes reach its final singleton stage, the algorithm terminates.

How to compute next level from the previous level – refinement of a class by class. Consider a refinement of a class $\mathcal{C}$ on level L by a class $\mathcal{D}$ on level L: take $i, j \in \mathcal{C}$, if $i+1, j+1 \in \mathcal{D}$, then we leave them together, otherwise we must separate them. For instance, let us refine a class $\mathcal{C} = \{0, 2, 3, 5, 7, 8, 10, 11, 13\}$ by a class

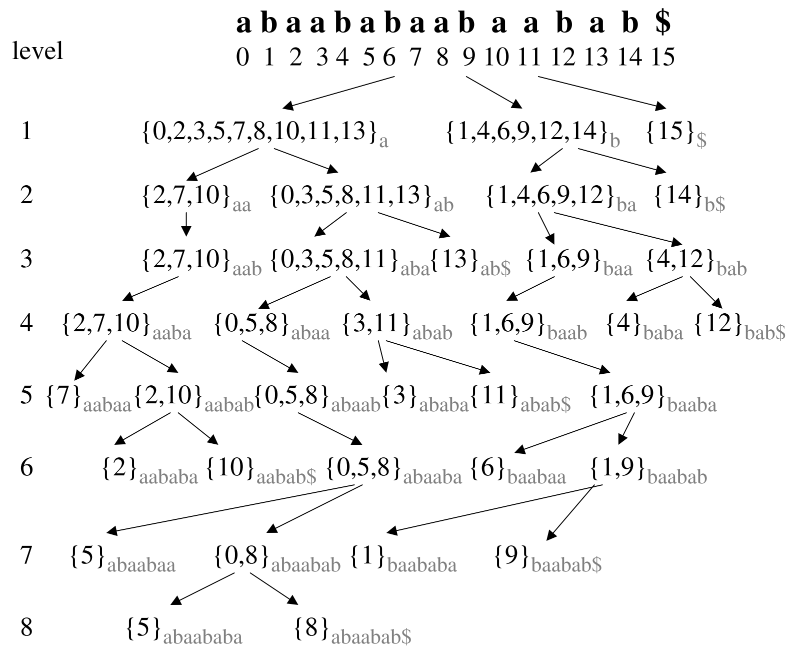


Figure 1. Classes of equivalence and their refinements for a string `abaababaabaabab`

$\mathcal{D} = \{1, 4, 6, 9, 12, 14\}$ on level 1. 0 and 2 must be separated as 1,3 are not both in $\mathcal{D}$, 0 and 3 will be in the same class, since 1,4 are both in $\mathcal{D}$. In fact $\mathcal{C}$ will be refined into two classes, one consisting of $\mathcal{D}$ shifted one position to the left ($\{0, 3, 5, 8, 11, 13\}$), and the ones that were separated ($\{2, 7, 10\}$). If we use all classes for refinement, we end up with the next level.

A major trick is not to use all classes for refinement. For each "family" of classes (classes that were formed as a refinementof a class on the previous level – for instance classes $\{2, 7, 10\}$ and $\{0, 3, 5, 8, 11, 13\}$ on level 2 form a family as they are a refinement of the class $\{0, 2, 3, 5, 7, 8, 10, 11, 13\}$ on level 1). In each family we identify the largest class and call all the others small. By using only small classes for refinement, $O(n \log n)$ complexity is achieved as each element belongs only to $O(\log n)$ small classes.

Many linked lists are needed to be maintained to keep track of classes, families, the largest classes in families, and gaps. Care must be taken to avoid traversing any of these structure lest the $O(n \log n)$ complexity be compromised. It was estimated that an implementation of Crochemore's algorithm requires about $20 * n$ machine words. FSX03 [4] managed to trim it down to $14 * n$ using memory multiplexing and virtualization without sacrificing either the complexity or much of the performance.

4 Extending Crochemore's algorithm to compute runs

One of the features of Crochemore's algorithm is that

(a) repetitions are reported level by level, i.e. all repetitions of the same period are reported together, and
(b) there is no order of repetition reporting with respect to the starting positions of the repetitions (this is a byproduct of the process of refinement),

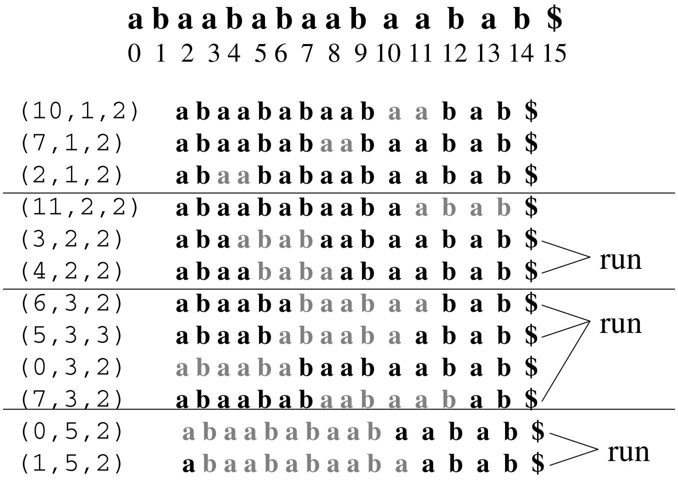


Figure 2. Reporting repetitions for string `abaababaabaabab`

and thus the repetitions must be "collected" and "joined" into runs. For instance, for a string $x =$`abaababaabaabab`, the order of repetitions as reported by the algorithm FSX03 ([4]) is shown in Fig. 2; it also shows some of the repetitions that have to be joined into runs.

The first aspect of Crochemore's algorithm (see (a) above) is good for computing runs, for all candidates of joining must have the same period. The second aspect (see (b) above) is detrimental, for it is needed to check for joining two repetitions with "neigbouring" starts.

4.1 Variant A

In this variant all repetitions for a level are collected, joined into runs, and reported. The high level logic:

1. Collect the runs in a binary search tree based on the starting position. There is no need to record the period, as all the repetitions and all the runs dealt with are of the same period.
2. When a new repetition is reported, find if it should be inserted in the tree as a new run, or if it should be joined with an existing run.
3. When all repetitions of the period had been reported, traverse the tree and report all runs (if depth first traversal is used, the runs will be reported in order of their starting positions).

The rules for joining:

1. Descend the tree as if searching for a place to insert the newly reported repetition.
2. For every run encountered, check if the repetition should be joined with it.
   (a) If the repetition is a substring of the run, ignore the repetition and terminate the search.
   (b) If the run is a substring of the repetition, replace the run with the repetition and terminate the search.
   (c) If the run's starting position is to the left of the starting position of the repetition, if the run and the repetition have an overlap of size $\geq p$, the run's tail

must be updated to accommodate the repetition (i.e. the run is extended to the right). On the other hand, if the overlap is of size $< p$ or empty, continue search.

(d) If the run's starting position is to the right of the starting position of the repetition, if the repetition and the run have an overlap of size $\geq p$, the run's starting position must be updated to accommodate the repetition (i.e. the run is extended to the left). On the other hand, if the overlap is of size $< p$ or empty, continue search.

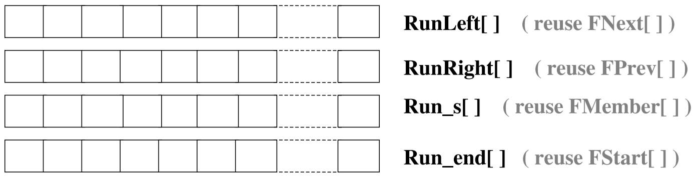


Figure 3. Data structures for Variant A

For technical reasons and to lower memory requirements, the runs are recorded in the search tree as pairs (s, d) where s is the starting position of the run, while d is the end position of the run (let us remark again that we do not need to store the period p). Note that we can easily compute the exponent: $e = (d{-}s{+}1) \ / \ p$, and the tail $t = (d{-}s{+}1) \ \% \ p$.

To avoid dynamic memory allocation and the corresponding deterioration of performance, the search tree is emulated by 4 integer arrays of size n, named `RunLeft[]` (emulating pointers to the left children), `RunRight[]` (emulating pointers to the right children), `Run_s[]` (emulating storing of the starting position in the node), and `Run_d[]` (emulating storing of the endposition in the node), see Fig. 3. Since the four arrays, `FNext[]`, `FPrev[]`, `FMember[]`, and `FStart[]`, are used in the underlying Crochemore's algorithm only for class refinement, and at the time of repetition reporting they can be used safely (as long as they are properly "cleaned" after the use), we do not need any extra memory.

Thus the variant A does not need any extra memory as each search tree is "destroyed" after the runs have been reported, however there is an extra penalty of traversing a branch of the search tree for each repetition reporting, i.e. extra $O(\log n)$ steps, leading to the complexity of $O(n(\log n)^2)$.

4.2 Variant B

In this variant all repetitions for all levels are collected, joined into runs, and reported together at the end.

The basic principles are the same as for variant A. However, for each level we build a separate search tree and keep it till the repetitions of all levels (periods) have been reported. We cannot use any of the data structures from the underlying Crochemore's algorithm as we did for variant A, so the memory requirement grows by additional $4 * n$ machine words. The time-complexity is the same as for variant A, i.e. $O(n(\log n)^2)$.

How do we know that all the runs can fit into the search trees with a total of n nodes? We do not know, for it is just a conjecture that the maximum number of

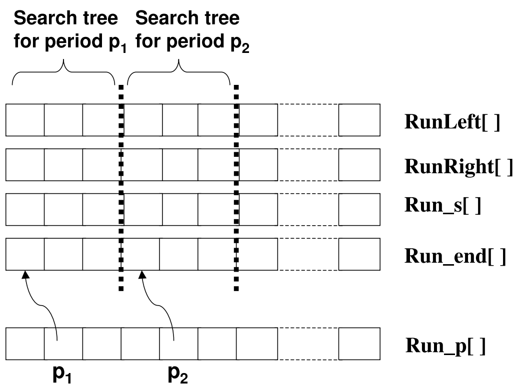


Figure 4. Data structures for Variant B

runs $< n$. However, if we run out of the space (there is a safeguard), we will have found a counterexample to the conjecture on the maximum number of runs (see e.g. [5]).

4.3 Variant C

As in Variant B, all repetitions for all levels are collected, joined into runs, and reported together at the end. However, this variant differs from B in the data structure used.

The repetitions are collected in a simple data structure consisting of an array `Buckets[]`. In the bucket `Buckets[`s`]` we store a simple singly-linked list of all repetitions that start at position s. To avoid as much as possible dynamic allocation, so-called "allocation-from-arena" technique is used for the linked lists (`Buckets[]` is allocated with the other structures) and $3 * n$ words is allocated in chunks as needed. The memory requirement for collecting and storing all the repetitions is $\leq 4n * \log n$ words, however an expected memory requirement is $4n$ words as the expected number of repetitions is n ($3n$ for the links, n for the buckets).

After all repetitions had been reported and collected, `Buckets[]` is traversed from left to right and all repetitions are joined into runs - we call this phase "sweep". In another traversal,the runs can bereported. During the sweep, everything to the left of the current index are runs, while everything to the right and including the current

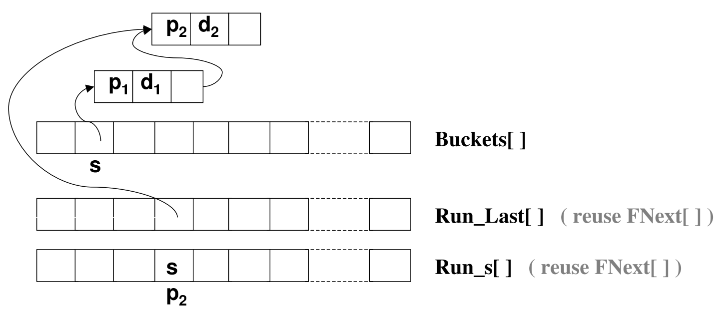


Figure 5. Data structures for Variant C

index are repetitions. For the joining business, we need for each period to remember the rightmost run with that period, that is the role of the array `RunLast[]` (we can reuse `FNext[]`). Thus when traversing the linked list in the bucket `Buckets[`i`]` and currently dealing with a repetition with period p_2, `RunLast[`p_2`]` points to the last run of period p_2 so we can decide if the current repetition is to be "promoted" to a run (with a zero tail), or joined with the run. Since the starting position of the last run of period p_2 is not stored in the run, we need one more array `Run_s[]` in which we store the starting position (we can reuse `FPrev[]`).

Since storing a repetition in `Buckets[]` takes a constant time, and there are $O(n \log n)$ repetitions, and since the joining business is also constant time, the overall time complexity is $O(n \log n) + O(n \log n)$, i.e. $O(n \log n)$.

5 Experimental results

Implementations of the three variants were compared as to their performance. The testing was rather informal, just to give indications how the three variants compare. Hardware: Sony VAIO laptop with Intel Core-2 Duo CPU T5800 @ 2.00 GHz, 4 GB of RAM
Software: Windows Vista Home Premium SP1. The code was written in C++ and was compiled using the GNU `g++` compiler.
Each run was repeated five times, the minimum numbers are recorded in the table given in Fig. 6 (`random2.txt` is a file of random strings on a binary alphabet, while `random21.txt` is a file of random strings on an alphabet of size 21).

Data Set	File Name	String Length	Time (seconds)		
			Variant A	Variant B	Variant C
DNA	dna.dna4	510976	105.87	110.15	3.12
English	bible.txt	4047392	63.27	62.65	23.90
Fibonacci	fibo.txt	305260	173.30	177.00	2.39
Periodic	fss.txt	304118	159.61	168.78	2.44
Protein	p1Mb.txt	1048576	47.93	53.23	5.15
Protein	p2Mb.txt	2097152	189.20	189.98	11.42
Random	random2.txt	510703	193.01	189.28	4.42
Random	random21.txt	510703	7.69	7.46	1.89

Figure 6. Comparing speed performance of variants A, B, and C

The table given in Fig. 7 records the performance averaged per a character of input:

The results allow for a quick conclusion:

1. Overall, variant C is significantly faster than variants A and B. In fact by 3643%!
2. Even though variant A requires less additional memory, speed-wise does not do much better than B.
3. The speed of variants A and B is not proportional to the string's length. Rather, it mostly depends on the type of the string. It works better on strings with large alphabet size and low periodicity. This is intuitively clear, as for high periodicity strings the height of the search trees are large.

6 Memory-saving modifications of Variant C

In the first modification, C1, repetitions are collected for a round of K levels, then a sweep is executed and the resulting runs are reported, and the bucket memory is then

Data Set	File Name	Name	# of runs	Time (µsec / letter)		
				Variant A	Variant B	Variant C
DNA	dna.dna4	510976	130368	207.18	215.57	6.11
English	bible.txt	4047392	63690	15.63	15.48	5.91
Fibonacci	fibo.txt	305260	233193	567.70	579.83	7.82
Periodic	fss.txt	304118	281912	524.84	554.98	8.01
Protein	p1Mb.txt	1048576	69605	45.71	50.76	4.91
Protein	p2Mb.txt	2097152	139929	90.22	90.59	5.45
Random	random2.txt	510703	210122	377.93	370.62	8.64
Random	random21.txt	510703	24389	15.06	14.60	3.70
Overall average				**230.53**	**236.55**	**6.32**

Figure 7. Comparing speed performance of variants A, B, and C per character of input

reused in the next "batch" of repetitions. For our experiments, we used $K = 100$, so we refer to this variant as C1-100.

In the second modification, C2, we consolidate repetitions with small periods ($\leq K$) into runs when putting them to the buckets (this saves memory since there are fewer runs than repetitions). For a repetition with periond $p \leq K$ and start s, we check p buckets to the left and to the right of s; for $p > K$, we check K buckets to the left and to the right of s. This guarantes that all reptitions up to period K have been consolidated into runs before the final sweep, while repetitions of periods $> K$ are partially consolidated. Thus the final sweep ignores the repetitions with periods $\leq K$. Beside saving memory, the final sweep is a bit shorter, while putting repetitions into the buckets is a bit longer. For our experiemts, we used $K = 10$, so we refer to this variant as C2-10.

The table given in Fig. 8 show comparisons of C, C1-100, and C2-10 for the speed of performance on the same datasets as the tests among the variants A, B, and C in tables in Fig. 6 and Fig. 7.

Data Set	File Name	File size (bytes)	# of runs	Time (seconds)		
				C	C1-100	C2-10
DNA	dna.dna4	510976	130368	3.02	3.04	2.87
English	bible.txt	4047392	63690	20.29	20.36	20.53
Fibonacci	fibo.txt	305260	233193	2.75	6.60	2.76
Periodic	fss.txt	304118	281912	2.65	5.34	3.02
Protein	p1Mb.txt	1048576	69605	4.47	4.52	4.42
Protein	p2Mb.txt	2097152	139929	10.21	10.56	10.29
Random	random2.txt	510703	210122	4.15	4.16	4.01
Random	random21.txt	510703	24389	1.59	1.65	1.57

Figure 8. Comparing speed performance of the variants C, C1-100, and C2-10

As expected, C is the fastest, however the differences are insignificant, except somehow significant results for `fibo.txt` and `fss.txt`.

The table given in Fig. 9 show comparisons of memory usage of C, C1-100, and C2-10.

Only on `fibo.txt` and `fss.txt` C1-100 and C2-10 exhibit memory savings, for all other data sets, the memory requirements are the same corresponding to the string's length (i.e. only 1 arena segment is allocated).

For the next set of tests we used large strings with large number of runs. The strings were obtained from W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and

Data set	File name	File size (bytes)	Alphabet size	# of runs	Memory (blocks)		
					C	C1-100	C2-10
DNA	dna.dna4	510976	5	130368	510976	510976	510976
English	bible.txt	4047392	63	63690	4047392	4047392	4047392
Fibonacci	fibo.txt	305260	2	233193	2747340	1221040	610520
Periodic	fss.txt	304118	2	281912	1824708	912354	608236
Protein	p1Mb.txt	1048576	23	69605	1048576	1048576	1048576
Protein	p2Mb.txt	2097152	23	139929	2097152	2097152	2097152
Random	random2.txt	510703	2	210122	510703	510703	510703
Random	random21.txt	510703	21	24389	510703	510703	510703

Figure 9. Comparing memory usage of the variants C, C1-100, and C2-10

A. Shinohara's website dedicated to "Lower Bounds for the Maximum Number of Runs in a String" at URL `http://www.shino.ecei.tohoku.ac.jp/runs/`.

The table in Fig. 10 indicates the time performance C, C1-100, and C2-10 on these run-rich large strings, while the table in Fig. 11 gives the memory usage.

File name	File size (bytes)	Alphabet size	# of runs	Time (seconds)		
				C	C1-100	C2-10
60064.txt	60064	2	56714	0.34	0.51	0.34
79568.txt	79568	2	75136	0.50	0.72	0.56
105405.txt	105405	2	99541	0.70	1.05	0.79
139632.txt	139632	2	131869	1.06	1.59	1.10
176583.txt	176583	2	166772	1.71	2.58	1.43
184973.txt	184973	2	174697	1.63	2.78	1.46

Figure 10. Comparing speed of C, C1-100, and C2-10 on large run-rich strings

File name	File size (bytes)	Alphabet size	# of runs	Time (seconds)		
				C	C1-100	C2-10
60064.txt	60064	2	56714	240256	180192	120128
79568.txt	79568	2	75136	318272	238704	159136
105405.txt	105405	2	99541	527025	316215	210810
139632.txt	139632	2	131869	698160	418896	279264
176583.txt	176583	2	166772	882915	529749	353166
184973.txt	184973	2	174697	924865	554919	369946

Figure 11. Memory usage of C, C1-100, and C2-10 on large run-rich strings

As expected, for strings with many short runs and a few long runs, C2-10 exhibits significant memory savings, with little performance degradation.

7 Conclusion and further research

We extended Crochemore's repetitions algorithm to compute runs. Of the three variants, variant C is by far more efficient time-wise, but requiring $O(n \log n)$ additional memory. However, its performance warrantied further investigation into further reduction of memory requirements. The preliminary experiments indicate that C2-K is the most efficient version and so it is the one that should be the used as the basis for parallelization. Let us remark that variant C (and any of its modifications) could be used as an extension of any repetitions algorithm that reports repetitions of the same period together.

References

1. M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch: *Replacing suffix trees with enhanced suffix arrays*, J. Discr. Algorithms 2 (2004), pp. 53–86
2. G. Chen, S.J. Puglisi, and W.F. Smyth: *Fast & practical algorithms for computing all the runs in a string*, Proc. 18th Annual Symposium on Combinatorial Pattern Matching (2007), pp. 307–315
3. M. Crochemore: *An optimal algorithm for computing the repetitions in a word*, Inform. Process. Lett. 5 (5) 1981, pp. 297–315
4. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on Strings*, Cambridge University Press 2007
5. M. Crochemore and L. Ilie: *Maximal repetitions in strings*, Journal of Computer and System Sciences 74-5 (2008), pp. 796–807
6. M. Farach: *Optimal suffix tree construction with large alphabets*, 38th IEEE Symp. Found. Computer Science (1997), pp. 137–143
7. F. Franek, W.F. Smyth, and X. Xiao: *A note on Crochemore's repetitions algorithm, a fast space-efficient approach*, Nordic J. Computing 10-1 (2003), pp. 21–28
8. C. Iliopoulos, D. Moore, and W.F. Smyth: *A characterization of the squares in a Fibonacci string*, Theoret. Comput. Sci.,172 (1997), pp. 281-291
9. R. Kolpakov and G. Kucherov: *On maximal repetitions in words*, J. of Discrete Algorithms, (1) 2000, pp. 159–186
10. J. Kärkkäinen and P. Sanders: *Simple linear work suffix array construction*, Proc. 30th Internat. Colloq. Automata, Languages & Programming (2003), pp. 943–955
11. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays*, Proc. 14th Annual Symp. Combinatorial Pattern Matching, R. Baeza-Yates, E. Chàvez, and M. Crochemore (eds.), LNCS 2676, Springer-Verlag (2003), pp. 200–210
12. M.G. Main: Detecting leftmost maximal periodicities, Discrete Applied Math., (25) 1989, pp. 145–153
13. U. Manber and G. Myers: *Suffix arrays: A new method for on-line string searches*, SIAM J. Comput. 22 (1993), pp. 935–938
14. P. Weiner: *Linear pattern matching algorithms*, Proc. 14th Annual IEEE Symp. Switching & Automata Theory (1973), pp. 1–11

Reducing Repetitions

Peter Leupold*

Department of Mathematics, Faculty of Science
Kyoto Sangyo University
Kyoto 603-8555, Japan
leupold@cc.kyoto-su.ac.jp

Abstract. For a given word w, all the square-free words that can be reached by successive application of rewriting rules $uu \to u$ constitute w's duplication root. One word can have several such roots. We provide upper and lower bounds on the maximal number of duplication roots of words of length n that show that this number is at least exponential in n.

Keywords: repetitions in strings, DNA operations, duplication

1 Repetitions and Duplication

A mutation, $\rightarrowtail$ which occurs frequently in DNA strands, is the duplication of a factor inside a strand [19]. The result is called a tandem repeat, and the detection of these repeats has received a great deal of attention in bioinformatics [1,20]. The reconstruction of possible duplication histories of a gene is used in the investigation of the evolution of a species [24]. Thus duplicating factors and deleting halves of squares is an interesting algorithmic problem with some motivation from bioinformatics, although squares do not need to be exact there. A very similar reduction was also introduced in the context of data compression by Ilie et al. [10,11]. They, however conserve information about each reduction step in the resulting string such that the operation can also be undone again. In this way the original word can always be reconstructed, which is essential for data compression. We will present their approach in more detail in Section 3 and establish some relations between the two reductions.

So far, the interpretation of duplication as an operation on a string has mainly inspired work in Formal Languages, most prominently the duplication closure. Dassow et al. introduced the duplication closure of a word and showed that the languages generated are always regular over two letters [7]. Wang then proved that this is not the case over three or more letters [23]. These results had actually been discovered before in the context of copy systems [8], [3]. It remains an open problem, whether such duplication closures are always context-free or not. Later on, length bounds for the duplicated factor were introduced [17], [15], and also the closure of language classes under the duplication operations was investigated [12]. Finally, also a special type of codes robust against duplications was investigated [16].

Besides considering duplication as a generative operation elongating strings, also the effects of the inverse operation on words have been the object of investigations [15]. Here duplications are undone, i.e. one half of them is deleted leaving behind only the other half of the square. In this way words are reduced to square-free words, which are in some sense primitive under this notion; this is why we call the set of all

* This work was done while Peter Leupold was funded as a post-doctoral fellow by the Japanese Society for the Promotion of Science under grant number P07810.

Peter Leupold: Reducing Repetitions, pp. 225–236.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

square-free words reachable from a given word w the duplication root of w in analogy to concepts like the primitive root or the periodicity root of words. Duplication roots of languages were studied already in earlier work by the present author [14].

Here we will focus on duplication roots of single words. Mainly the following question is addressed: how many different duplication roots can a word have? We establish an exponential lower bound for this number as well as an upper bound. Besides any possible applications, this study of how repetitions in a sequence can be nested follows important lines of study in Combinatorics of Words, where repetitions have been in the center of attention from the very start in the work of Thue [22].

2 Definitions

We assume the reader to be familiar with fundamental concepts from Formal Language Theory such as alphabet, word, and language, which can be found in many standard textbooks like the one by Harrison [9]. The length of a finite word w is the number of not necessarily distinct symbols it consists of and is written $|w|$. The number of occurrences of a certain letter a in w is $|w|_a$. The i-th symbol we denote by $w[i]$. The notation $w[i \ldots j]$ is used to refer to the part of a word starting at the i-th position and ending at the j-th position.

A word u is a *prefix* of w if there exists an $i \leq |w|$ such that $u = w[1 \ldots i]$; if $i < |w|$, then the prefix is called *proper*. The set of all prefixes is $\mathsf{pref}(w)$. Suffixes are the corresponding concept reading from the back of the word to the front and they are denoted by suff. We define the *letter sequence* $\mathsf{seq}(u)$ of a word u as follows: any word u can be uniquely factorized as $u = x_1^{i_1} x_2^{i_2} \cdots x_\ell^{i_\ell}$ for some integers $\ell \geq 0$ and $i_1, i_2, \ldots, i_\ell \geq 1$ and for letters $x_1, x_2, \ldots, x_\ell$ such that always $x_j \neq x_{j+1}$; then $\mathsf{seq}(u) := x_1 x_2 \cdots x_\ell$. Intuitively speaking, every block of several adjacent occurrences of the same letter is reduced to just one occurrence.

We call a word w *square-free* iff it does not contain any non-empty factor of the form u^2, where exponents of words refer to iterated catenation, and thus u^i is the i-fold catenation of the word u with itself. A word w has a positive integer k as a *period*, if for all i, j such that $i \equiv j (\mathrm{mod} k)$ we have $w[i] = w[j]$, if both $w[i]$ and $w[j]$ are defined.

For applying duplications to words we use string-rewriting systems. In our notation we mostly follow Book and Otto [2] and define such a *string-rewriting system* R on Σ to be a subset of $\Sigma^* \times \Sigma^*$. Its single-step reduction relation is defined as $u \rightarrow_R v$ iff there exists $(\ell, r) \in R$ such that for some u_1, u_2 we have $u = u_1 \ell u_2$ and $v = u_1 r u_2$. We also write simpler just $\rightarrow$, if it is clear which is the underlying rewriting system. By $\xrightarrow{*}$ we denote the relation's reflexive and transitive closure, which is called the *reduction relation* or *rewrite relation*. The inverse of a single-step reduction relation $\rightarrow$ is $\rightarrow^{-1} := \{(r, \ell) : (\ell, r) \in R\}$. Further notation that will be used is $IRR(R)$ for the set of words irreducible for a string-rewriting system R. With this we come to the definition of duplications.

The string-rewriting system we use here is the *duplication relation* defined as $u \heartsuit v :\Leftrightarrow \exists z [z \in \Sigma^+ \wedge u = u_1 z u_2 \wedge v = u_1 z z u_2]$. Notice how the symbol $\heartsuit$ nicely visualizes the operation going from one origin to two equal halves. If we have length bounds $|z| \leq k$ or $|z| = k$ on the factors to be duplicated we write $\heartsuit^{\leq k}$ or $\heartsuit k$ respectively; the relations are called *bounded* and *uniformly bounded duplication* respectively.

$\heartsuit^*$ is the reflexive and transitive closure of the relation $\heartsuit$. The *duplication closure* of a word w is then $w^\heartsuit := \{u : w\heartsuit^* u\}$. The languages $w^{\heartsuit \leq k}$ and $w^{\heartsuit k}$ are defined analogously. Because our main topic is the reduction of squares, we will mainly use the inverse of $\heartsuit$ and will denote it by $\rightarrowtail := \heartsuit^{-1}$; the notations for length-bounded versions and iterated applications are used accordingly. Notice that for $\rightarrowtail_{\leq k}$ the length bound does not refer to the length of the rules' left sides, but rather to half that length. This makes sense, because otherwise for all even k we would have $\rightarrowtail_{\leq k} = \rightarrowtail_{\leq k+1}$, and because this way the relations $\rightarrowtail_{\leq k}$ and $\heartsuit^{\leq k}$ correspond. We will use a similar convention when talking about squares. Thus we will say that a square u^2 is of length $|u|$; in this case u will be called the *base* of this square.

With this we have all the prerequisites for defining the central notion of this work, the duplication root.

Definition 1. *The* duplication root *of a non-empty word w is*

$$\sqrt[\heartsuit]{w} := IRR(\rightarrowtail) \cap \{u : w \rightarrowtail^* u\}.$$

As usual, this notion is extended in the canonical way from words to languages such that

$$\sqrt[\heartsuit]{L} := \bigcup_{w \in L} \sqrt[\heartsuit]{w}.$$

The roots $\sqrt[\heartsuit \leq k]{w}$ and $\sqrt[\heartsuit k]{w}$ are defined in completely analogous ways, and also these are extended to entire languages in the canonical way. When we want to contrast the duplication (root) without length bound to the bounded variants we will at times call it *general duplication (root)*.

When talking about the elements of a word's duplication root, we will also call them simply roots; no confusion should arise. Similarly, where we say "the number of roots" we mean the root's cardinality. Though not completely correct, these formulations are more compact and in many cases easier to understand.

Finally, notice that all words in a duplication root are square-free, and over an alphabet of two letters only the seven square-free words $\{\lambda, a, b, ab, ba, aba, bab\}$ exist. They are uniquely determined by their first letter, the last letter, and the set of letters occurring in them. Thus most problems about duplication roots are trivial unless we have at least three letters. Therefore, unless otherwise stated, we will suppose an alphabet of size at least three in what follows. First off, we illustrate this definition with an example that also shows that duplication roots are in general not unique, i.e., the set $\sqrt[\heartsuit]{w}$ can contain more than one element as we will see further on.

Example 2. By undoing duplications, i.e., by applying rules from $\rightarrowtail$, we obtain from the word $w = abcbabcbc$ the words in the set $\{abc, abcbc, abcbabc\}$; in a first step either the prefix $(abcb)^2$ or the suffix $(bc)^2$ can be reduced, only the former case results in a word with another square, which can be reduced to abc.

Thus we have the root $\sqrt[\heartsuit]{abcbabcbc} = \{abc, abcbabc\}$. Exhaustive search of all shorter words shows that this is a shortest possible example of a word with more than one root over three letters.

Other examples with cardinalities of the root greater than two are the words $w_3 = babacabacbcabacb$ where

$$\sqrt[\heartsuit]{w_3} = \{bacabacb, bacbcabacb, bacb\},$$

and $w_5 = ababcbabcacbabcabacbabcab$ where

$$\sqrt[\heartsuit]{w_5} = \{abcbabcabacbabcab, abcbabcab, abcacbabcab, abcabacbabcab, abcab\},$$

As the examples have finite length, the bounded duplication root is in general not unique either. The uniformly bounded duplication root, however, is known to be unique over any alphabet [15].

As already stated in the Introduction, so far research on duplication has mainly focused on its language theoretic properties. We recall the most important results on these from [14].

Theorem 3. *The closure properties of the classes of regular and context-free languages under the three duplication roots are as follows:*

	$\sqrt[\heartsuit k]{L}$	$\sqrt[\heartsuit<k]{L}$	$\sqrt[\heartsuit]{L}$
REG	*Y*	*Y*	*N*
CF	*?*	*?*	*N*

The symbol Y stands for closure, N stands for non-closure, and ? means that the problem is open.

Here our focus is different. We will look in more detail at the duplication roots of single words. One interesting question is how ambiguous it can be in relation to a word's length.

Before we take a closer look at this question, however, we will now recall a notion that is very closely related to our reduction.

3 The Relation to Repetition Complexity

In an effort to define a new measure for the complexity of words, Ilie et al. [10,11] defined a reduction relation very similar to undoing duplications, which however remembers the steps it takes, and in this way the original word can be restored from the reduced one. For the definition let $D = \{0, 1, \ldots 9\}$ be the set of decimal digits, and Σ be an alphabet disjoint from D. The alphabet for the reduction relation is $T := \Sigma \cup D \cup \{\langle, \rangle, \hat{}\}$. For a positive integer n let $\mathsf{dec}\, n$ denote its decimal representation. Then the reduction relation $\Rightarrow$ is defined by $u \Rightarrow v$ iff $u = u_1 x^n u_2$, $v = u_1 \langle x \rangle \hat{}\langle \mathsf{dec}\, n \rangle u_2$ for some $u_1, u_2 \in T^*$, $x = \Sigma^+$, $n > 2$. Finally, let h be the morphism erasing all symbols except the letters from Σ.

We illustrate in a simple example the different way of operation of the two relations.

Example 4. For the word $ababcbc$ there are two irreducible forms under $\Rightarrow$, namely $\langle ab \rangle^{\langle 2 \rangle} cbc$ and $aba \langle bc \rangle^{\langle 2 \rangle}$. Under $\rightarrowtail$, however, the images of both words under h are further reducible to a common normal form: both $ababcbc \rightarrowtail abcbc \rightarrowtail abc$ and $ababcbc \rightarrowtail ababc \rightarrowtail abc$ are possible reductions leading to abc. Notice how the brackets block the further reduction of $abab$ in $aba \langle bc \rangle \hat{}\langle 2 \rangle$ and of $bcbc$ in $\langle ab \rangle \hat{}\langle 2 \rangle cbc$.

There are two main differences between the two relations.

1. A reduction $u^n \Rightarrow \langle u \rangle \hat{}\langle n \rangle$ is done in a single step while the reduction $u^n \rightarrowtail^* u$ will always take $n - 1$ steps.

2. If $w \Rightarrow^* u$ then $w \rightarrowtail^* h(u)$, but the reverse does not hold, see Example 4.

Despite these differences, the similarities are evident, and $\Rightarrow^*$ can be embedded in $\rightarrowtail^*$. We state a further relation.

Proposition 5. *For a word w, if $\sqrt[\heartsuit]{w} \subseteq \{h(u) : w \Rightarrow^* u\}$ then $|\sqrt[\heartsuit]{w}| = 1$.*

Proof. Let p and q be two different words in $\sqrt[\heartsuit]{w}$. Then there exist words u, p', q' such that $w \rightarrowtail^* u$, $u \rightarrowtail p' \rightarrowtail^* p$, $u \rightarrowtail q' \rightarrowtail^* q$, but no reductions $p' \rightarrowtail^* q$ or $q' \rightarrowtail^* p$ exist. Intuitively this means that the paths to p and q divide in the point u, which thus is a greatest lower bound of $\{p, q\}$ in the set $w^{\rightarrowtail}$ with $\rightarrowtail^*$ as partial order. The two unduplications in $u \rightarrowtail p'$ and $u \rightarrowtail q'$ must overlap, otherwise there would be a word v such that $p' \rightarrowtail v$ and $q' \rightarrowtail v$. Let the two factors that are reduced be u_p^2 and u_q^2, where $|u_p| > |u_q|$ without loss of generality; notice that $|u_p| = |u_q|$ would result in $p' = q'$.

The overlap of the unduplications must be greater than $|u_q|$. Otherwise there is a w' such that the unduplications are applied to a factor $u_p w' u_q$ or $u_q w' u_p$ and the effect can be seen as the deletion of u_p and u_q; both would be possible consecutively. Further, the maximal repetition of u_p were it is reduced must be less than u_p^3, otherwise the factor u_q would still be present after deletion of one u_p. This means that a reduction under $\Rightarrow$ can only result in $\langle u_p \rangle\,\hat{}\,\langle 2 \rangle$, no higher exponent, and no factor u_p can follow on either side.

There can be no factor u_q^2 directly preceding or following $\langle u_p \rangle$ on the side of the overlap. Otherwise, again derivations $p' \rightarrowtail v$ and $q' \rightarrowtail v$ would have been possible. This means that the square u_q^2 in $h(p')$ cannot be reduced, neither can an equivalent reduction leading to the same result be done. Analogous reasoning holds for the case that first u_q^2 is reduced to $\langle u_q \rangle\,\hat{}\,\langle 2 \rangle$, and thus $\{h(u) : w \Rightarrow^* u\}$ cannot contain any square-free word.

□

Intuitively this means that if $\Rightarrow$ can reduce a word to a square-free one, then the overlaps of its repetitive factors must be so minor that they do not lead to ambiguous duplication roots either. Already Example 4 shows that the converse of Proposition 5 does not hold.

From the proof of Proposition 5 we can extract an important property of the relation $\rightarrowtail^*$ that characterizes the situation, when two strings derived from the same word can become incomparable.

Definition 6. *Let w be a word. We will call two squares p^2 and q^2 a pair of* critical squares *in w, if w has a factor u such that*

1. $p^2 \in \mathsf{pref}(u)$,
2. $q^2 \in \mathsf{suff}(u)$,
3. $|u| \leq 2(\max(|p|, |q|)) + \min(|p|, |q|) - 1$.

Without further proof we state the following.

Lemma 7. *Let w, p, and q be words such that $w \rightarrowtail p$ and $w \rightarrowtail q$.*
If $\{v : p \rightarrowtail^* v\} \cap \{v : q \rightarrowtail^* v\} = \emptyset$, then w contains a pair of critical squares.

4 The Number of Duplication Roots

A decisive question for any algorithmic problem related to duplication is the one about the possible number of duplication roots of a word with respect to its length. To find an exact bound seems to be a very intricate problem, and so we try to find good upper and lower bounds on this number. More formally, we try to find bounds for the function defined as

$$\mathsf{duproots}(n) := \max\{|\sqrt[\heartsuit]{w}| : |w| = n\}.$$

The function $\mathsf{duproots}$ is monotonically growing. For any word w, duplicate one of its letters to obtain a word w' of length $|w|+1$. Clearly $w' \rightarrowtail w$ and thus $\sqrt[\heartsuit]{w} \subseteq \sqrt[\heartsuit]{w'}$. Consequently we have $\mathsf{duproots}(n) \leq \mathsf{duproots}(n+1)$ for all $n > 0$. Therefore writing $|w| = n$ in the definition is equivalent to writing $|w| \leq n$.

Because it has often turned out to be very useful to consider problems about duplications with a length restriction, we also define the function

$$\mathsf{bduproots}{\leq}\ k(n) := \max\{|\sqrt[\heartsuit \leq k]{w}| : |w| = n\}.$$

By definition we have $\mathsf{bduproots}_{\leq k} \leq \mathsf{duproots}$ and $\mathsf{bduproots}_{\leq k} \leq \mathsf{bduproots}_{\leq k+1}$ for all $k > 0$. We now try to characterize the growth of the function $\mathsf{duproots}$ more exactly.

4.1 Bounding from Above

Obviously, rules from $\rightarrowtail$ can only be applied on square factors. Thus the number of squares is the number of possible distinct rule applications in a string. However, when we are interested in rule applications with distinct result and thus with potentially distinct roots, the number of runs captures this more exactly.

Recall that a *run* is a maximal repetition of exponent at least two in a string. It is known that the number of runs in a string of length n is linearly bounded by n [13]. A great deal of work has been done to determine the constant c such that $c \cdot n$ is the exact bound. The most recent results indicate that c lies between 1.6 [6] and 0.94 [18]. The following fact shows how this number plays a role for the number of possible reductions via $\rightarrowtail$ and thus for the number of duplication roots.

Fact 8. *Let w be a word with period k. Then all applications of rules from $\rightarrowtail_k$ will result in the same word, i.e. $\{u : w \rightarrowtail_k u\}$ is a singleton set.*

As a consequence of this, the number of distinct descendants of w with respect to $\rightarrowtail$ is equal to the number of runs in w. In this way. the number of runs seems to play an important role for the computation of the maximal number of duplication roots.

To obtain a first approximation on this number, let us state the following: the number of runs in a string of length n is bounded linearly by the string's length. Reducing one square leaves the word's length in general in the order of n, thus also the number of runs is again in the order of n.

On the other hand, every reduction via $\rightarrowtail$ removes at least one letter, thus there can be at most $n-1$ steps in the reduction of a word of length n. More precisely, observe that deleting one half of a square cannot remove all copies of a letter from a given string. Thus all roots of a word over three letters have at least three letters themselves. Overall, there are up to $n-3$ times up to n choices for reducing squares, and the number of different reduction paths lies in $\mathcal{O}(n^n)$. Using the upper bound on the number of runs we see that

$$\mathsf{duproots}(n) \leq (1.6n)^{n-3}.$$

Of course, this gives a very rough upper bound. Most importantly, it disregards the fact that many reductions starting in different points will converge again at some point. Obviously, two rule applications in factors that do not overlap can be applied in either order with identical result. Further, not all of the strings reachable during a reduction will reach the maximum number of runs.

We recall a result from [14].

Lemma 9. *If for two words $u, v \in \Sigma^*$ we have $\mathsf{seq}(u) = \mathsf{seq}(v)$, then also $\sqrt[\heartsuit]{u} = \sqrt[\heartsuit]{v} = \sqrt[\heartsuit]{\mathsf{seq}(u)}$.*

This means that we can first do all the possible reductions of the form $x^2 \to x$ for single letters x. So for possible splits to different duplication roots we can assume that at least two letters are deleted in every step. Actually, also the fact that u from Example 4 is the shortest possible word with at least two distinct duplication roots shows that we need only consider applications of rules $u^2 \to u$ with $|u| \geq 2$. This lowers our upper bound to $(1.6n)^{\frac{n-3}{2}}$.

The improvement is not substantial, however. In the initial approach, in some sense all possible paths from w to words in $\sqrt[\heartsuit]{w}$ in the Hasse diagram of the partial order $[w^{\rightarrowtail}, \rightarrowtail^*]$ are counted. The improved version counts only the paths starting from $\mathsf{seq}(w)$ as depicted in Figure 1. The optimal case, however would be to count only one path per element of $\sqrt[\heartsuit]{w}$. We can take another step into this direction for the partial order $[w^{\rightarrowtail_{\leq k}}, \rightarrowtail_{\leq k}{}^*]$. As exemplary value for k we choose 30, the reason for this will become evident in the next section.

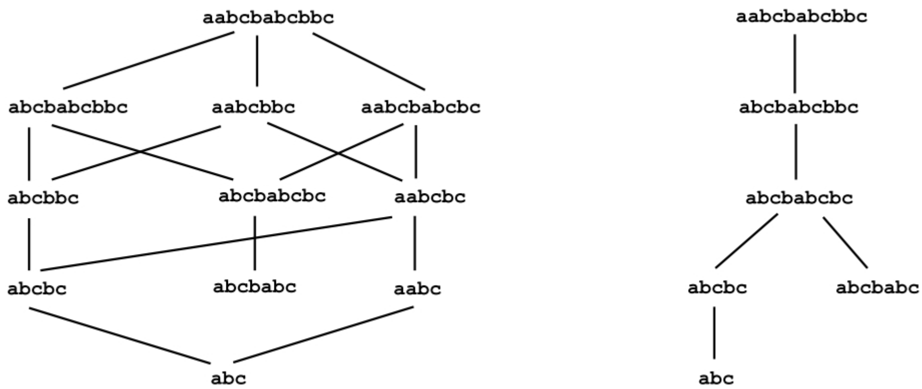


Figure 1. 10 versus 2 paths for the word *aabcbabcbbc*, by first reducing one-letter squares from left to right. The direction of reductions is top to bottom.

Lemma 7 characterizes the words, from which it may not be possible to rejoin outgoing paths. They need to have a critical overlap. The involved squares' bases cannot be longer than 30. Further, one must be shorter than the other, but of length at least two. So for a given square, there are at most 29 such candidates. They can overlap on either side, which gives 56 possible combinations. The shorter square must have more than one half of its length inside the other, and at least one letter must be outside the other. So for a square of length m we have $m-1$ possible positions. The overall number of possible configurations is therefore $2\sum_{2\le i\le 29} i-1 = 2\sum_{1\le i\le 28} i = 812$.

For calculating the number of possible roots of a word w, we now employ the following tactics. Again, we first compute $\mathsf{seq}(w)$. Then we do not follow all possible paths from $\mathsf{seq}(w)$, but rather select one random square. If it does not form part of a pair of critical squares, then we simply reduce it and proceed further with the next square. Otherwise, for all critical pairs we follow also the paths resulting from reducing the possible partners in these pairs. As we have seen, a square of length 30 can form part of at most 812 critical pairs. The length of the paths is subject to the same bound as for $\rightarrowtail$, and thus we have to follow at most $812^{\frac{n-3}{2}}$ paths, which is the upper bound on $\mathsf{bduproots}_{\le 30}$.

Clearly, especially the first bound of $(1.6n)^{\frac{n-3}{2}}$ is very far off the real value. Indeed, all roots of a word w are shorter than w unless w is square-free. Let us label w's letters from the start to the end. We can look at a rule $uu \to u$ like the deletion of one copy of u, so its labels disappear. Thus every word in $\sqrt[\heartsuit]{w}$ corresponds to a subset of the set of $|w|$ labels. There are only $2^{|w|}$ such subsets, which gives us a much better upper bound, also independent of the alphabet size. We still have given the construction of our bound, because we feel that it bears potential for improvement even beyond $2^{|w|}$. Intuitively, the linear bound on the number of runs in a string means that they must be distributed rather evenly over the string's length. Further, results like the Theorem of Fine and Wilf suggest that one run can only form a very limited number of pairs of critical squares, so that even in the case of unbounded duplication we should be able to get an average constant bound like the one of 812 for $\rightarrowtail_{\le 30}$. By careful analysis of the possibilities, it should be possible to lower the bound even beyond $2^{|w|}$.

4.2 Bounding from Below

The upper bound on the number of duplication roots is very high and raises the question how far from the real number it is. By an example we now establish a lower bound for this number, which is also exponential. Thus it shows that the upper bound is not too bad.

Example 10. We construct an example of a sequence of words w_n, which are simply powers of a word w, namely $w_n := w^n$. The number of roots increases exponentially in n. This is a modification of a construction used earlier to present a simple language with infinite duplication root [14]. We start the construction of w from the word $u = abcbabcbc$; in Example 2 we have seen that the root of u consists of the two words $u_1 = abc$ and $u_2 = abcbabc$. The basic idea is to concatenate copies of u; in every factor there is the choice of u_1 or u_2 and thus every additional copy of u doubles the number of roots. However, simple concatenation of u would allow further reductions. Therefore we need to modify and separate the different copies of u in ways that prevent the creation of further squares.

The first measure we take is permuting the letters. Let ρ be the morphism, which simply renames letters according to the scheme $a \to b \to c \to a$. Then $\rho(u)$ has the two roots $\rho(u_1)$ and $\rho(u_2)$; similarly, $\rho(\rho(u))$ has the two roots $\rho(\rho(u_1))$ and $\rho(\rho(u_2))$.

We will now use this ambiguity to construct the word w. This word over the four-letter alphabet $\{a, b, c, d\}$ is

$$w = ud\rho(u)d\rho(\rho(u))d = abcbabcbc \cdot d \cdot bcacbcaca \cdot d \cdot cabacabab \cdot d.$$

Thus the duplication root of w contains among others the three words

$$\begin{aligned} w_a &= abc \cdot d \cdot bca \cdot d \cdot cabacab \cdot d \\ w_b &= abc \cdot d \cdot bcacbca \cdot d \cdot cab \cdot d \\ w_c &= abcbabc \cdot d \cdot bca \cdot d \cdot cab \cdot d, \end{aligned}$$

which are square-free. We now need to recall that a morphism h is called square-free, iff $h(v)$ is square-free for all square-free words v. Crochemore has shown that a uniform morphism h is square-free iff it is square-free for all square-free words of length 3 [5]. Here uniform means that all images of single letters have the same length, which is given in our case.

The morphism we define now is $\varphi(x) := w_x$ for all $x \in \{a, b, c\}$. Thus to establish the square-freeness of φ, we need to check this property for the images of all square-free words up to length 3. These are

$$\begin{aligned} \varphi(aba) &= abcdbcadcabacabdabcdbcacbcadcabdabcdbcadcabacabd \\ \varphi(abc) &= abcdbcadcabacabdabcdbcacbcadcabdabcbabcdbcadcabd \\ \varphi(aca) &= abcdbcadcabacabdabcbabcdbcadcabdabcdbcadcabacabd \\ \varphi(acb) &= abcdbcadcabacabdabcbabcdbcadcabdabcdbcacbcadcabd \\ \varphi(bab) &= abcdbcacbcadcabdabcdbcadcabacabdabcdbcacbcadcabd \\ \varphi(bac) &= abcdbcacbcadcabdabcdbcadcabacabdabcbabcdbcadcabd \\ \varphi(bca) &= abcdbcacbcadcabdabcbabcdbcadcabdabcdbcadcabacabd \\ \varphi(bcb) &= abcdbcacbcadcabdabcbabcdbcadcabdabcdbcacbcadcabd \\ \varphi(cac) &= abcbabcdbcadcabdabcdbcadcabacabdabcbabcdbcadcabd \\ \varphi(cab) &= abcbabcdbcadcabdabcdbcadcabacabdabcdbcacbcadcabd \\ \varphi(cba) &= abcbabcdbcadcabdabcdbcacbcadcabdabcdbcadcabacabd \\ \varphi(cbc) &= abcbabcdbcadcabdabcdbcacbcadcabdabcbabcdbcadcabd, \end{aligned}$$

where, of course, the images of all words shorter than three are contained in them. All the twelve words listed here are indeed square-free as an eager reader can check, and thus φ is square-free.

Now let t be an infinite square-free word over the letters a, b and c. Such a word exists [22]. Then all the words in $\varphi(\mathsf{pref}(t))$ are square-free, too. From the construction of φ we know that for any word z of length i we can reach $\varphi(z)$ from w^i by undoing duplications. Therefore $\varphi(\mathsf{pref}(t)) \subseteq \sqrt[\heartsuit]{w^+}$. For two distinct square-free words t_1 and t_2, also $\varphi(t_1) \neq \varphi(t_2)$. Finally, notice that for all positive $i \leq n$ we have $w^n \rightarrowtail^* w^i$.

This means that all square-free words that are not longer than n lead to a different duplication root of w_n. Therefore $\mathsf{bduproots}_{\leq 30} \leq s$, where $s(n)$ is the number of ternary square-free words of length up to n. This function's value is not known, however, it was first bounded to $6 \cdot 1.032^n \leq s(n) \leq 6 \cdot 1.379^n$ by Brandenburg [4]. A better lower bound was found by Sun $s(n) \geq 110^{\frac{n}{42}}$ [21]. w itself is of length $3|u| + 3 = 30$. So we see that $\mathsf{bduproots}_{\leq 30}(n) \geq \frac{1}{30} 110^{\frac{n}{42}}$.

Example 10 leaves room for improvement in several respects.

- The word w is over a four-letter alphabet. The letter d is used to separate the different blocks that introduce the ambiguities and only use the alphabet $\{a, b, c\}$. The question is whether this function can also be fulfilled by an appropriate word over $\{a, b, c\}$; computer experiments with candidate words have always led to unwanted squares with some of the adjoining factors.

– The ambiguity of u that we use is only two-fold. Using the words w_3 and w_5 from Example 2, it might be possible to pack more choices into less room and thus improve the initial constant of $\frac{1}{5}$ with similar constructions. However, this would not change the magnitude. On the other hand, the resulting morphism would not be uniform, which would complicate the establishment of its square-freeness.

Summarizing this section up to this point, we have the following bounds for the function $\mathsf{duproots}$.

Proposition 11. $\frac{1}{30}110^{\frac{n}{42}} \leq \mathsf{duproots}(n) \leq 2^n$ *for all* $n > 0$.

Because Example 10 uses only rules from $\succ\!\!\rightarrow_{\leq 30}$, its bound holds also for $\mathsf{bduproots}_{\leq 30}$. So for this function we get a much sharper characterization of its growth.

Proposition 12. $\frac{1}{30}110^{\frac{n}{42}} \leq \mathsf{bduproots}_{\leq 30}(n) \leq \max\{812^{\frac{n-3}{2}}, 2^n\}$ *for all* $n > 0$.

While this upper bound is still enormous, we have at least achieved a bounding between two exponential functions. So for this case the bounds are much tighter, though still rather loose. For ternary alphabet, the upper bound $6 \cdot 1.379^n$ by Brandenburg can replace 2^n in both Propositions.

4.3 Computing the Number of Duplication Roots

Proposition 11 shows that the straight-forward approach to computing the function $\mathsf{duproots}$ will lead to exponential runtime. But it seems reasonable to assume that it is not necessary to actually compute the set $\sqrt[\heartsuit]{w}$ to determine its size. Example 4 suggests that it suffices to identify the number of critical overlaps in the original word. In this case, even linear time might suffice. However, it remains to show that no new critical pairs can come up during a reduction, or at least that their number can be foreseen by looking a the original word.

5 Open Problems

The first and most evident problem is, of course, a better characterization of the function $\mathsf{duproots}$. We conjecture that in some way a bounding will be possible in a way similar to that for $\mathsf{bduproots}$, and thus also $\mathsf{duproots}$ can be bounded from above and below by exponential functions. For this a refined analysis of pairs of critical squares might be the key, just as for a linear time algorithm to actually compute $\mathsf{duproots}$.

Besides this, three more algorithmic problems related to the duplication root of a word suggest themselves.

(i) Duplication Root: For a given word w, find one of its duplication roots.
(ii) Minimal Duplication Root: For a given word w, find one of the shortest of its duplication roots.
(iii) Complete Duplication Root: For a given word w, find all of its duplication roots.

To solve Problem (i) we can follow any reduction. As seen above, these can take up to n steps. In each step one square must be detected and reduced. Therefore a runtime of $\mathcal{O}(n^2 \log n)$ can be expected. An interesting question is whether Problem (ii) can be solved faster than Problem (iii), i.e. do we basically have to enumerate the entire root to know which is its smallest element, or can, for example, a greedy strategy eliminate many candidate reductions early on. No exact results on the complexity of any of the three problems are known.

Another interesting field is finding restrictions on the general duplication which are on the one hand motivated from practical considerations like possible tandem repeats in DNA, and on the other hand make the problems described here more tractable. Since tandem repeats cannot occur at arbitrary factors of a DNA strand, there might be less than exponentially many possible duplication histories for DNA strands.

Acknowledgments. The words w_3 and w_5 from Example 2 were found by Szilárd Zsolt Fazekas, the fact that w is the shortest example of a word with ambiguous root was established by Artiom Alhazov with a computer. The observation that the number of subsets of a set with $|w|$ elements bounds $|\sqrt[\heartsuit]{w}|$ was first made by Masami Ito.

References

1. D. A. Benson: *Tandem repeat finder: A program to analyze DNA sequences*. Nucleic Acids Research, 27(2) 1999, pp. 573–580.
2. R. Book and F. Otto: *String-Rewriting Systems*, Springer, Berlin, 1993.
3. D. P. Bovet and S. Varricchio: *On the regularity of languages on a binary alphabet generated by copying systems*. Information Processing Letters, 44(3) 1992, pp. 119–123.
4. F.-J. Brandenburg: *Uniformly growing k-th power-free homomorphisms*. Theor. Comput. Sci., 23 1983, pp. 69–82.
5. M. Crochemore: *Sharp characterizations of squarefree morphisms*. Theoretical Computer Science, 18 1982, pp. 221–226.
6. M. Crochemore and L. Ilie: *Maximal repetitions in strings*. Journal of Computer and System Sciences, 74(5) 2008, pp. 796–807.
7. J. Dassow, V. Mitrana, and G. Păun: *On the regularity of duplication closure*. Bulletin of the EATCS, 69 1999, pp. 133–136.
8. A. Ehrenfeucht and G. Rozenberg: *On the separating power of EOL systems*. RAIRO Informatique Thorique, 17(1) 1983, pp. 13–22.
9. M. A. Harrison: *Introduction to Formal Language Theory*, Addison-Wesley, Reading, Massachusetts, 1978.
10. L. Ilie, S. Yu, and K. Zhang: *Repetition complexity of words*, in COCOON, O. H. Ibarra and L. Zhang, eds., vol. 2387 of Lecture Notes in Computer Science, Springer, 2002, pp. 320–329.
11. L. Ilie, S. Yu, and K. Zhang: *Word complexity and repetitions in words*. Int. J. Found. Comput. Sci., 15(1) 2004, pp. 41–55.
12. M. Ito, P. Leupold, and K. Shikishima-Tsuji: *Closure of language classes under bounded duplication*, in Developments in Language Theory, O. H. Ibarra and Z. Dang, eds., vol. 4036 of Lecture Notes in Computer Science, Springer, 2006, pp. 238–247.
13. R. M. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in FOCS, 1999, pp. 596–604.
14. P. Leupold: *Duplication roots*, in Developments in Language Theory, T. Harju, J. Karhumäki, and A. Lepistö, eds., vol. 4588 of Lecture Notes in Computer Science, Springer, 2007, pp. 290–299.
15. P. Leupold, C. Martín-Vide, and V. Mitrana: *Uniformly bounded duplication languages*. Discrete Applied Mathematics, 146(3) 2005, pp. 301–310.
16. P. Leupold and V. Mitrana: *Uniformly bounded duplication codes*. Theor. Inform. Appl., 41(4) 2007, pp. 411–424.
17. P. Leupold, V. Mitrana, and J. M. Sempere: *Formal languages arising from gene repeated duplication*, in Aspects of Molecular Computing, N. Jonoska, G. Paun, and G. Rozenberg, eds., vol. 2950 of Lecture Notes in Computer Science, Springer, 2004, pp. 297–308.

18. W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and A. Shinohara: *New lower bounds for the maximum number of runs in a string*, in Proceedings of the Prague Stringology Conference 2008, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2008, pp. 140–145.
19. E. Pennisi: *MOLECULAR EVOLUTION: Genome Duplications: The Stuff of Evolution?* Science, 294(5551) 2001, pp. 2458–2460.
20. D. Sokol, G. Benson, and J. Tojeira: *Tandem repeats over the edit distance.* Bioinformatics, 23(2) 2007, pp. 30–35.
21. X. Sun: *New lower bound on the number of ternary square-free words.* Journal of Integer Sequences, 6(3) 2003, pp. 1–8.
22. A. Thue: *Über die gegenseitige Lage gleicher Teile verschiedener Zeichenreihen.* Norske Videnskabers Selskabs Skrifter Mat.-Nat. Kl. (Kristiania), 1 1912, pp. 1–67.
23. M.-W. Wang: *On the irregularity of the duplication closure.* Bull. EATCS, 70 2000, pp. 162–163.
24. I. Wapinski, A. Pfeffer, N. Friedman, and A. Regev: *Natural history and evolutionary principles of gene duplication in fungi.* Nature, 449 2007, pp. 54–61.

Asymptotic Behaviour of the Maximal Number of Squares in Standard Sturmian Words

(Extended Abstract)

Marcin Piątkowski[2*] and Wojciech Rytter[1,2**]

[1] Department of Mathematics, Computer Science and Mechanics,
University of Warsaw, Warsaw, Poland
rytter@mimuw.edu.pl

[2] Faculty of Mathematics and Informatics,
Nicolaus Copernicus University, Toruń, Poland
martinp@mat.umk.pl

Abstract. Denote by $sq(w)$ the number of distinct squares in a string w and let $\mathcal{S}$ be the class of standard Sturmian words. They are generalizations of Fibonacci words and are important in combinatorics on words. For Fibonacci words the asymptotic behaviour of the number of runs and the number of squares is the same. We show that for Sturmian words the situation is quite different. The tight bound $\frac{8}{10}|w|$ for the number of runs was given in [3]. In this paper we show that the tight bound for the maximal number of squares is $\frac{9}{10}|w|$. We use the results of [11] where exact (but not closed) complicated formulas were given for $sq(w)$ for $w \in \mathcal{S}$ and we show:

(1) for all $w \in \mathcal{S} \quad sq(w) \leq \frac{9}{10}\,|w| + 4$,

(2) there is an infinite sequence of words $w_k \in \mathcal{S}$ such that

$$\lim_{k\to\infty} |w_k| \;=\; \infty \qquad \text{and} \qquad \lim_{k\to\infty} \frac{sq(w_k)}{|w_k|} \;=\; \frac{9}{10}.$$

Surprisingly the maximal number of runs is reached by the words with recurrences of length only 5. This contrasts with the situation of Fibbonaci words, though standard Sturmian words are natural extension of Fibonacci words. If this length drops to 4, the asymtotic behaviour of the maximal number of squares falls down significantly below $\frac{9}{10}|w|$. The structure of Sturmian words rich in squares has been discovered by us experimentally and verified theoretically. The upper bound is much harder, its proof is not a matter of simple calculations. The summation formulas for the number of squares are complicated, no closed formula is known. Some nontrivial reductions were necessary.

1 Introduction

A square in a string is a subword of the form ww, where w is nonempty. The squares are a simplest form of repetitions, despite the simple formulation many combinatorial problems related to squares are not well understood. The subject of computing maximal number of squares and repetitions in words is one of the fundamental topics in combinatorics on words [18,22] initiated by A. Thue [28], as well as it is important in other areas: lossless compression, word representation, computational biology, etc.

* The research supported by Ministry of Science and Higher Education of Poland, grant N N206 258035.

** Supported by grant N206 004 32/0806 of the Polish Ministry of Science and Higher Education.

Marcin Piątkowski, Wojciech Rytter : Asymptotic Behaviour of the Maximal Number of Squares in Standard Sturmian Words, pp. 237–248.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

Let $sq(w)$ be the number of distinct squares in the word w and $sq(n)$ be the maximal number of distinct squares in the word of length n. The behaviour of the function $sq(n)$ is not well understood, though the subject of squares was studied by many authors, see [9,10,17]. The best known results related to the value of $sq(n)$ are, see [13,15,16]:

$$n - o(n) \ \leq \ sq(n) \ \leq \ 2n - O(\log n).$$

In this paper we concentrate on the asymptotic behaviour of the maximal number of squares in class of standard Sturmian words $\mathcal{S}$. We show: for all $w \in \mathcal{S} \quad sq(w) \leq \frac{9}{10}\,|w|$ and there is an infinite sequence of strictly growing words $\{w_k\} \in \mathcal{S}$ such that

$$\lim_{k\to\infty} \frac{sq(w_k)}{|w_k|} \ = \ \frac{9}{10}\,.$$

There are known efficient algorithms for the computation of integer powers in words, see [2,6,11,23,24]. The powers in words are related to maximal repetitions, also called *runs*. It is surprising that the known bounds for the number of runs are much tighter than for squares, this is due to the work of many people [3,7,8,14,19,20,25,26,27].

One of interesting questions related to squares is the relation of their number to the number of runs. In case of Fibonacci words the number of squares and runs differ only by 1.

The results of this paper show that the maximal number of squares and the maximal number of runs are possibly not closely related, since in case of well structured words (Sturmian words) the *density ratio* of squares (the asymptotic quotient of the maximal number of squares by the size of the string) is $\frac{9}{10}$ and for runs it is $\frac{8}{10}$.

2 Standard Sturmian words

The *standard Sturmian words* (*standard words*, in short) are generalization of Fibonacci words and have a very simple *grammar-based* representation which has some algorithmic consequences.

Let $\mathcal{S}$ denote the set of all standard Sturmian words. These words are defined over a binary alphabet $\Sigma = \{a, b\}$ and are described by recurrences (or grammar-based representation) corresponding to so called *directive sequences*: integer sequences

$$\gamma = (\gamma_0, \gamma_1, \ldots, \gamma_n),$$

where $\gamma_0 \geq 0$, $\gamma_i > 0$ for $0 < i \leq n$.

The word x_{n+1} corresponding to γ, denoted by $\mathrm{Sw}(\gamma)$, is defined by recurrences:

$$\begin{array}{ll} x_{-1} = b, & x_0 = a, \\ x_1 = x_0^{\gamma_0} x_{-1}, & x_2 = x_1^{\gamma_1} x_0, \\ \cdots & \cdots \\ x_n = x_{n-1}^{\gamma_{n-1}} x_{n-2}, & x_{n+1} = x_n^{\gamma_n} x_{n-1}. \end{array} \tag{1}$$

Fibonacci words are standard Sturmian words given by the directive sequences of the form $\gamma = (1, 1, \ldots, 1)$ (n-th Fibonacci word F_n corresponds to a sequence of

n ones). We consider here standard words starting with the letter a, hence assume $\gamma_0 > 0$. The case $\gamma_0 = 0$ can be considered similarly.

For even $n > 0$ a standard word x_n has the suffix ba, and for odd $n > 0$ it has the suffix ab. The number $N = |x_{n+1}|$ is the (real) size, while $n+1$ can be thought as the compressed size.

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

$$\mathrm{Sw}(1, 2, 1, 3, 1) \;=\; abababababaababababaabababaababaab$$

$$x_{-1} = b, \quad x_0 = a, \quad x_1 = x_0^1 x_{-1} = a\ b, \quad x_2 = x_1^2 x_0 = ab\ ab\ a,$$
$$x_3 = x_2^1 x_1 = ababa\ ab, \quad x_4 = x_3^3 x_2 = ababaab\ ababaab\ ababaab\ ababa,$$
$$x_5 = x_4^1 x_3 = ababaababababaababababaababababa\ ababaab$$

The *grammar-based compression* consists in describing a given word by a context-free grammar G generating this (single) word. The size of the grammar G is the total length of all productions of G. In particular each directive sequence of a standard Sturmian word corresponds to such a compression – the sequence of recurrences corresponding to the directive sequence. In this case the size of the grammar is proportional to the length of the directive sequence.

For some lexicographic properties and structure of repetitions of standard Sturmian words see [5,3,1,4].

3 Summation formulas for the number of squares

The exact formulas for the number of squares in standard Sturmian words were given by Damanik and Lenz in [11]. In this section we reformulate their formulas to have compact version more suitable for the asymptotic analysis. The formulas are rather complicated and such an analysis is nontrivial. It will be done in the section 5.

Denote $q_i = |x_i|$, where x_i are as in equation (1). The following lemma charecterize the possible lengths of periods of squares in Sturmian words.

Lemma 2. ([11])
Let $w = \mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n)$ be a standard Sturmian word. Each primitive period of a square in w has the length kq_i for $1 \le k \le \gamma_i$ or $kq_i + q_{i-1}$ for $1 \le k < \gamma_i$.

The squares in standard Sturmian word w with period of the length kq_i for $1 \le k \le \gamma_i$ or $kq_i + q_{i-1}$ for $1 \le k < \gamma_i$ are said to be of type i.

Example 3.
Consider the word from Example 1:

$$\mathrm{Sw}(1, 2, 1, 3, 1) \;=\; abababababaababababaabababaababaab.$$

We have:
one square of type 0: $a{\cdot}a$,
three squares of type 1 (period 2, 3): $ab{\cdot}ab$, $ba{\cdot}ba$, $aba{\cdot}aba$,
three squares of type 2 (period 5): $ababa{\cdot}ababa$, $babaab{\cdot}babaab$, $abaab{\cdot}abaab$,
and eleven squares of type 3 (with periods 7,14):

$ababaab \cdot ababaab$, $babaaba \cdot babaaba$, $abaabab \cdot abaabab$, $baababa \cdot baababa$,

$aababab \cdot aababab$, $abababa \cdot abababa$, $bababaa \cdot bababaa$,

$ababaabababaab \cdot ababaabababaab$, $babaabababaaba \cdot babaabababaaba$,

$abaababababaabab \cdot abaababababaabab$, $baabababaababa \cdot baabababaababa$.

Let $sq_i(\gamma_0, \gamma_1, \ldots, \gamma_n)$, for $1 \leq i \leq n$, be the number of squares of the type i and let $sq_0(\gamma_0, \gamma_1, \ldots, \gamma_n)$ be the number of squares with period of the form a^+ in the word $\mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n)$.

We slightly abuse the notation and denote $sq(\gamma_0, \gamma_1, ..., \gamma_n) = sq\big(\mathrm{Sw}(\gamma_0, \gamma_1, ..., \gamma_n)\big)$.

Denote $d(0) = \left\lfloor \frac{\gamma_0+1}{2} \right\rfloor$ and for $1 \leq i \leq n$ and $\gamma = (\gamma_0, \gamma_1, \ldots, \gamma_n)$:

$$d_1(i) = \begin{cases} \frac{\gamma_i}{2} q_i + q_{i-1} - 1 & \text{for even } \gamma_i \\ \frac{\gamma_i}{2} q_i + \frac{1}{2}\, q_i & \text{for odd } \gamma_i \end{cases}$$

$$d(i) = d_1(i) + \gamma_i\, q_i - q_i - \gamma_i + 1.$$

Let $\mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n)$ be a standard Sturmian word. Then $sq(\gamma_0, \gamma_1, \ldots, \gamma_n)$ is determined as follows, see [11]:

Summation formulas:

(1) $sq(\gamma_0, \gamma_1, \ldots, \gamma_n) = \sum_{i=0}^{n} sq_i(\gamma_0, \gamma_1, \ldots, \gamma_n)$.

(2) $(0 \leq i \leq n-3)$ or $(i = n-2 \;\&\; \gamma_n \geq 2) \Rightarrow sq_i(\gamma) = d(i)$.

(3) $\gamma_n = 1 \Rightarrow sq_{n-2}(\gamma) = \begin{cases} d(n-2) - q_{n-3} + 1 & \text{for even } \gamma_{n-2} \\ d(n-2) - q_{n-2} + q_{n-3} + 1 & \text{otherwise} \end{cases}$

(4) $\gamma_n = 1 \Rightarrow sq_{n-1}(\gamma) = \begin{cases} d_1(n-1) - q_{n-2} + 1 & \text{for even } \gamma_{n-1} \\ d_1(n-1) - q_{n-1} + q_{n-2} - 1 & \text{otherwise} \end{cases}$

(5) $\gamma_n > 1 \Rightarrow sq_{n-1}(\gamma) = \begin{cases} d(n-1) - q_{n-2} + 1 & \text{for even } \gamma_{n-1} \\ d(n-1) - q_{n-1} + q_{n-2} - 1 & \text{otherwise} \end{cases}$

(6) $sq_n(\gamma) = \begin{cases} d_1(n) - q_n + 2 & \text{for even } \gamma_n \\ d_1(n) - q_n & \text{otherwise} \end{cases}$

4 Sturmian words with many squares

In this section we present and analyse the sequence $\{w_k\}$ of Sturmian words achieving asymptotically maximal ratio:

$$\lim_{k\to\infty} |w_k| = \infty \qquad \text{and} \qquad \lim_{k\to\infty} \frac{squares(w_k)}{|w_k|} = \frac{9}{10}\,.$$

Recall that the squares with periods kq_i for $1 \le k \le \gamma_i$ or $kq_i + q_{i-1}$ are said to be of the type i.

Consider the words

$$w_k \;=\; \mathrm{Sw}(k, k, 2, 1, 1).$$

Example 4.

$$\begin{aligned} w_1 &= \mathrm{Sw}(1,1,2,1,1) &&= (aba)^2ab(aba)^3ab,\\ w_2 &= \mathrm{Sw}(2,2,2,1,1) &&= \Big((aab)^2a\Big)^2 aab\Big((aab)^2a\Big)^3 aab,\\ w_3 &= \mathrm{Sw}(3,3,2,1,1) &&= \Big((aaab)^3a\Big)^2 aaab\Big((aaab)^3a\Big)^3 aaab. \end{aligned}$$

Sw(3, 3, 2, 1, 1) is illustrated in Figure 1.

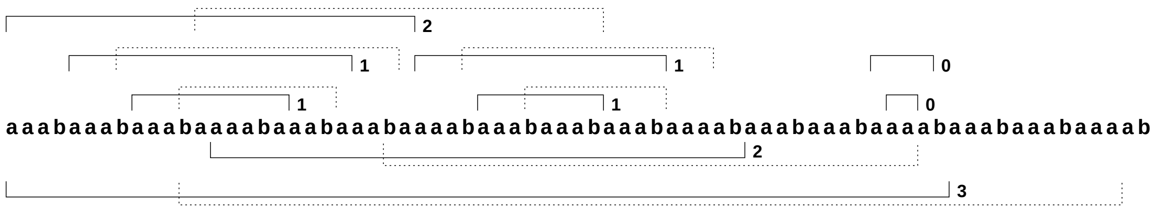


Figure 1. The squares in word Sw(3, 3, 2, 1, 1) with their *shifts* and types.

Theorem 5.
We have $sq(k, k, 2, 1, 1) \longrightarrow \frac{9}{10} \cdot \Big|\mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n)\Big|$ *for* $k \longrightarrow \infty$.

Proof.
Let $\gamma = (k, k, 2, 1, 1)$. We have:

$$\mathrm{Sw}(\gamma) \;=\; \Big((a^kb)^ka\Big)^2 a^kb\Big((a^kb)^ka\Big)^3 a^kb$$

and

$$|\mathrm{Sw}(\gamma)| \;=\; 5k^2 + 7k + 7.$$

We compute separately the number of squares for each type $0 \le i \le 4$.

There are two cases depending on the parity of k and we can assume that $k > 1$.

Case 1: k is odd.
We have (according to our formulas):

$$\begin{aligned}
sq_0(\gamma) &= \frac{1}{2}\Big(k+1\Big),\\
sq_1(\gamma) &= \frac{1}{2}\Big(3k^2+1\Big),\\
sq_2(\gamma) &= 2k^2+2k+1,\\
sq_3(\gamma) &= k^2+k,\\
sq_4(\gamma) &= 0,\\
sq(\gamma) &= \frac{1}{2}\Big(9k^2+7k+4\Big).
\end{aligned}$$

Finally

$$\lim_{k\to\infty}\frac{sq(\gamma)}{|\mathrm{Sw}(\gamma)|} = \lim_{k\to\infty}\frac{9k^2+7k+4}{10k^2+14k+14} = 0.9\,.$$

Case 2: k is even.
We have (according to our formulas):

$$\begin{aligned}
sq_0(\gamma) &= \frac{1}{2}k,\\
sq_1(\gamma) &= \frac{1}{2}\Big(3k^2-k\Big),\\
sq_2(\gamma) &= 2k^2+2k+1,\\
sq_3(\gamma) &= k^2+k,\\
sq_4(\gamma) &= 0,\\
sq(\gamma) &= \frac{1}{2}\Big(9k^2+6k+2\Big).
\end{aligned}$$

Finally

$$\lim_{k\to\infty}\frac{sq(\gamma)}{|\mathrm{Sw}(\gamma)|} = \lim_{k\to\infty}\frac{9k^2+6k+2}{10k^2+14k+14} = 0.9.$$

This concludes the proof.

5 Asymptotic behaviour of the maximal number of squares

The formulas (1-6) from the section 3 give together the value of $sq(\gamma)$, however there is no close simple formula. Therefore tight asymptotic estimations are nontrivial. We start with an estimation for short γ. The proof of the following simple lemma is omitted in this version.

Lemma 6. [Short γ]
$sq(\gamma_0,\gamma_1,\gamma_2) \le \frac{7}{3}|\mathrm{Sw}(\gamma_0,\gamma_1,\gamma_2)|$ and $sq(\gamma_0,\gamma_1,\gamma_2) \le |\mathrm{Sw}(\gamma_0,\gamma_1,\gamma_2)| - 4$.

For a word w of length at least 2 denote by $exch_2(w)$ the word resulting from w by exchanging the last two letters.

Lemma 7. $sq(w) \leq sq\big(exch_2(w)\big) + 4.$

Proof.
It is known, see [13], that there are at most two *last* occurrences of different squares at a single position in a string. If we reverse the word then this corresponds to the end-positions of the *first* occurrences. Hence at the last two positions at most 4 different squares can end which do not appear earlier in the same word with the last two letters removed. This completes the proof.

The next two lemmas allows us to restrict the values of last two elements of the directive sequence in the asymptotic estimation of $sq(\gamma)$.

Lemma 8. [Reduction of γ_n]
Let $\mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n)$ *be a standard Sturmian word. If* $\gamma_n > 1$ *then*

$$sq(\gamma_0, \ldots, \gamma_{n-1}, \gamma_n) \leq sq(\gamma_0, \ldots, \gamma_{n-1}, \gamma_n - 1, 1) + 4.$$

Proof.
The words $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n - 1, 1)$ and $\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)$ differ only on the last two letters, see [18]. Hence

$$\mathrm{Sw}(\gamma_0, \ldots, \gamma_n - 1, 1) \;=\; exch_2\big(\mathrm{Sw}(\gamma_0, \ldots, \gamma_n)\big).$$

Now the thesis follows from the Lemma 7.

Lemma 9. [Reduction of γ_{n-1}]
Let $x \;=\; SW(\gamma_0, \ldots, \gamma_{n-2}, \gamma_{n-1}, 1),\;\; x' \;=\; SW(\gamma_0, \ldots, \gamma_{n-2}, 1, 1),$
$x'' \;=\; SW(\gamma_0, \ldots, \gamma_{n-2}, 2, 1).$
Then

$$\left(sq(x') \;\leq\; \frac{9}{10}\,|x'| \text{ and } sq(x'') < \frac{9}{10}\,|x''|\right) \;\Rightarrow\; sq(x) \;\leq\; \frac{9}{10}\,|x|.$$

Proof.
If γ_{n-1} is odd then let $\varDelta = \gamma_{n-1} - 1$ otherwise let $\varDelta = \gamma_{n-1} - 2$.
Consider what happens when we change γ_{n-2} by the quantity $\varDelta$.
The increase of the number of squares is $\frac{\varDelta}{2}\, q_{n-1}$, while the increase in the length of the word is $\varDelta\, q_{n-1}$. The increase of squares is amortized by half of the increase of the length. Therefore we can subtract $\varDelta$ from γ_{n-1}.

Observation

$$d(i) \;\leq\; \begin{cases} \left(\dfrac{3}{2}\,\gamma_i - 1\right) q_i + q_{i-1} - 1 & \text{for even } \gamma_i \\[2ex] \left(\dfrac{3}{2}\,\gamma_i - \dfrac{1}{2}\right) q_i & \text{for odd } \gamma_i \end{cases}$$

Lemma 10.
For $2 \le r \le n-3$ *we have*

$$\sum_{i=0}^{r} d(i) \; < \; \frac{3}{2}\, q_{r+1} + q_r.$$

Proof.
According to the observation above and implication

$$\gamma_i \ge 2 \quad \Rightarrow \quad q_{i-1} - q_i < -\,\frac{1}{2}\, q_i,$$

we have:

$$d(i) \le \frac{3}{2}\, \gamma_i\, q_i - \frac{1}{2}\, q_i.$$

Observe now that $\gamma_i\, q_i = q_{i+1} - q_{i-1}$. Hence for $r \ge 2$:

$$\sum_{i=1}^{r} \gamma_i\, q_i \quad = \quad q_{r+1} + q_r - q_0 - q_1.$$

Consequently

$$\begin{aligned} \sum_{i=0}^{r} d(i) \; &< \; d(0) + \frac{3}{2} \sum_{i=1}^{r} \gamma_i\, q_i \; - \frac{1}{2}\, q_r \\ &\le \; d(0) + \frac{3}{2}\Big(q_{r+1} + q_r - q_0 - q_1\Big) - \frac{1}{2}\, q_r \\ &\le \; \frac{3}{2}\, q_{r+1} + q_r. \end{aligned}$$

This completes the proof.

Now we are ready to prove the tight bound for the number of squares in standard Sturmian words.

Theorem 11.
Let $\mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n)$ *be a standard Sturmian word. Then*

$$sq(\gamma_0, \gamma_1, \ldots, \gamma_n) \;\; \le \;\; \frac{9}{10} \cdot \Big|\mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n)\Big| + 4.$$

Proof.
First assume that:

$$\boxed{\gamma_n = 1 \quad \text{and} \quad \gamma_{n-1} \in \{1, 2\}}$$

Let us shorten the notation and denote:

$$A = q_{n-2}, \quad B = q_{n-3}, \quad \alpha = \gamma_{n-2}.$$

We have, due to Lemma 10, the following fact (in terms of A and B):

Claim 1.

$$\sum_{i=0}^{n-3} sq_i(\gamma) \;=\; \sum_{i=0}^{n-3} d(i) \;\leq\; \frac{3}{2}\,A + B.$$

This, together with the fact that $sq_n(\gamma_0, \gamma_1, \ldots, \gamma_{n-1}, 1) = 0$, implies:

Claim 2.

$$\boxed{sq(\gamma) \;\leq\; \varPhi(\gamma) \;\stackrel{def}{=}\; \frac{3}{2}\,A + B + sq_{n-1}(\gamma) + sq_{n-2}(\gamma).}$$

Our goal is to prove the inequality

$$\varPhi(\gamma) \leq \frac{9}{10}\,|w|.$$

Using our terminology we can write:

(a) $\big|\mathrm{Sw}(\gamma)\big| \;=\; \begin{cases} 2\,\alpha\,A + A + 2B & \text{for } \;\gamma_{n-1} = 1 \\ 3\,\alpha\,A + A + 3B & \text{for } \;\gamma_{n-1} = 2 \end{cases}$

(b) $sq_{n-2}(\gamma) \;\leq\; \begin{cases} \frac{3}{2}\,\alpha\,A - A & \text{for even } \gamma_{n-2} \\ \frac{3}{2}\,\alpha\,A - \frac{3}{2}A + B + 1 & \text{for odd } \gamma_{n-2} \end{cases}$

(c) $sq_{n-1}(\gamma) \;\leq\; \begin{cases} \alpha\,A + B & \text{for } \;\gamma_{n-1} = 2 \\ A - 1 & \text{for } \;\gamma_{n-1} = 1 \end{cases}$

There are 4 cases depending on $\gamma_{n-1} \in \{1, 2\}$ and the parity of α.

Case 1: ($\gamma_{n-1} = 1$, α is even)

In this case inequality $\varPhi(\gamma) \leq \frac{9}{10}\,|w|$ reduces to:

$$\boxed{\frac{3}{2}\Big(\alpha + 1\Big)\;A + B \;\leq\; \frac{9}{10}\;\Big((2\;\alpha + 1)\;A + 2\;B\Big).}$$

This reduces to:

$$\frac{3}{2}\Big(\alpha + 1\Big) \leq \frac{9}{10}\;\Big(2\;\alpha + 1\Big).$$

The last inequality is reduced to $0.6 \leq 0.3\;\alpha$, which obviously holds for $\alpha \geq 2$.

This completes the proof of this case.

Case 2: ($\gamma_{n-1} = 1$, α is odd)

In this case the inequality $\Phi(\gamma) \leq \frac{9}{10}\,|w|$ reduces to:

$$\boxed{\left(\frac{3}{2}\,\alpha{+}1\right)\,A{+}2\,B \leq \frac{9}{10}\,\Big((2\,\alpha{+}1)\,A{+}2\,B\Big),}$$

which holds for $\alpha \geq 1,\ A > B > 0$.

Case 3: ($\gamma_{n-1} = 2$, α is even)

In this case

$$\Phi(\gamma) \leq \left(\frac{5}{2}\,\alpha + \frac{1}{2}\right)\,A + 2\,B.$$

Consequently the inequality $\Phi(\gamma) \leq \frac{9}{10}\,|w|$ reduces to:

$$\boxed{\left(\frac{5}{2}\,\alpha{+}\frac{1}{2}\right)\,A{+}2\,B\ \leq\ \frac{9}{10}\,\Big(3\,\alpha\,A{+}A{+}3B\Big).}$$

This holds since $\alpha \geq 2,\ A > B > 0$.

Case 4: ($\gamma_{n-1} = 2$, α is odd)

In this case

$$\Phi(\gamma)\ \leq\ \frac{5}{2}\,\alpha\,A + 3\,B + 1.$$

Now the inequality $\Phi(\gamma) \leq \frac{9}{10}\,|w|$ reduces to:

$$\boxed{\frac{5}{2}\,\alpha\,A + 3\,B + 1\ \leq\ \frac{9}{10}\,\Big(3\,\alpha\,A + A + 3B\Big).}$$

This holds for $\alpha \geq 1,\ A > B > 0$.

We proved that

$$sq(\gamma_0, \gamma_1, \ldots, \gamma_{n-2}, 1, 1)\ \leq\ \frac{9}{10}\,|\mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_{n-2}, 1, 1)|$$

and

$$sq(\gamma_0, \gamma_1, \ldots, \gamma_{n-2}, 2, 1)\ \leq\ \frac{9}{10}\,|\mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_{n-2}, 2, 1)|.$$

This implies, that in general case, due to Lemma 8 and Lemma 9, we have

$$sq(\gamma_0, \gamma_1, \ldots, \gamma_n)\ \leq\ \frac{9}{10}\,|\mathrm{Sw}(\gamma_0, \gamma_1, \ldots, \gamma_n)| + 4,$$

which completes the proof of the theorem.

6 Final remarks

The *maximal repetition* (the *run*, in short) in a word w is a nonempty subword $w[i..j] = u^k v$ ($k \geq 2$), where u is of the minimal length and v is proper prefix (possibly empty) of u, that can not be extended (neither $x[i-1..j]$ nor $x[i..j+1]$ is a run with period $|u|$).

Let $\rho(w)$ be the number of runs in the word w. For n-th Fibonacci word F_n we have:

$$\begin{aligned} sq(F_n) &= 2|F_{n-2}| - 2, \\ \rho(F_n) &= 2|F_{n-2}| - 3, \end{aligned}$$

hence $sq(F_n) = \rho(f_n) + 1$, see [12,21].

For standard Sturmian words the situation is different. We have:

$$\frac{\rho(w)}{|w|} \longrightarrow 0.8 \qquad \text{and} \qquad \frac{sq(w)}{|w|} \longrightarrow 0.9,$$

see [3] for more details.

The maximal number of runs is reached for the standard Sturmian words of the form $v_k = \mathrm{Sw}(1, 2, k, k)$. Using the formulas (1-6) from the section 3 we have:

$$sq(v_k) = \begin{cases} \frac{5}{2}k^2 + \frac{5}{2}k + 4 & \text{for even } k \\ \frac{5}{2}k^2 + 5k - \frac{5}{2} & \text{for odd } k \end{cases}$$

and

$$|v_k| = 5k^2 + 2k + 5,$$

consequently

$$\frac{sq(v_k)}{|v_k|} \longrightarrow \frac{1}{2}.$$

We have shown in the section 4 that the maximal number of squares is achieved for the Sturmian words of the form $w_k = \mathrm{Sw}(k, k, 2, 1, 1)$. Now we compute the number of runs for w_k using formulas from [3]. We have:

$$\rho(w_k) = 9k + 7$$

and

$$|w_k| = 5k^2 + 7k + 7,$$

hence

$$\frac{\rho(w_k)}{|w_k|} \longrightarrow 0.$$

The results above show that the maximal number of squares and the maximal number of runs for standard Sturmian words are not closely related. The asymptotical limits are close, but both are reached for different type of words.

References

1. J. Allouche and J. Shallit: *Automatic Sequences: Theory, Applications, Generalizations*, Cambridge University Press, 2003.
2. A. Apostolico and F. P. Preparata: *Optimal off-line detection of repetitions in a string.* Theor. Comput. Sci., 22 1983, pp. 297–315.
3. P. Baturo, M. Piątkowski, and W. Rytter: *The number of runs in Sturmian words*, in CIAA 2008, 2008, pp. 252–261.
4. P. Baturo, M. Piątkowski, and W. Rytter: *Usefulness of directed acyclic subword graphs in problems related to standard Sturmian words*, in PSC 2008, 2008, pp. 193–207.
5. P. Baturo and W. Rytter: *Occurrence and lexicographic properties of standard Sturmian words*, LATA, 2007.
6. M. Crochemore: *An optimal algorithm for computing the repetitions in a word.* Inf. Process. Lett., 12(5) 1981, pp. 244–250.
7. M. Crochemore and L. Ilie: *Maximal repetitions in strings.* J. Comput. Syst. Sci., 74(5) 2008, pp. 796–807.
8. M. Crochemore, L. Ilie, and L. Tinta: *Towards a solution to the "runs" conjecture*, in CPM, P. Ferragina and G. M. Landau, eds., vol. 5029 of Lecture Notes in Computer Science, Springer, 2008, pp. 290–302.
9. M. Crochemore and W. Rytter: *Squares, cubes, and time-space efficient string searching.* Algorithmica, 13(5) 1995, pp. 405–425.
10. M. Crochemore and W. Rytter: *Jewels of Stringology*, World Scientific, 2003.
11. D. Damanik and D. Lenz: *Powers in Sturmian sequences.* Eur. J. Comb., 24(4) 2003, pp. 377–390.
12. A. S. Fraenkel and J. Simpson: *The exact number of squares in Fibonacci words.* Theor. Comput. Sci., 218(1) 1999, pp. 95–106.
13. A. S. Freankel and J. Simpson: *How many squares can a string contain?* J. of Combinatorial Theory Series A, 82 1998, pp. 112–120.
14. M. Giraud: *Not so many runs in strings*, in LATA, C. Martín-Vide, F. Otto, and H. Fernau, eds., vol. 5196 of Lecture Notes in Computer Science, Springer, 2008, pp. 232–239.
15. L. Ilie: *A simple proof that a word of length n has at most $2n$ distinct squares.* J. of Combinatorial Theory Series A, 112 2005, pp. 163–164.
16. L. Ilie: *A note on the number of squares in a word.* Theoretical Computer Science, 380 2007, pp. 373–376.
17. C. S. Iliopoulos, D. Moore, and W. F. Smyth: *A characterization of the squares in a Fibonacci string.* Theor. Comput. Sci., 172(1-2) 1997, pp. 281–291.
18. J. Karhumaki: *Combinatorics on words*, Notes in PDF, `http://www.math.utu.fi/en/home/karhumak/`.
19. R. M. Kolpakov and G. Kucherov: *Finding maximal repetitions in a word in linear time*, in FOCS, 1999, pp. 596–604.
20. R. M. Kolpakov and G. Kucherov: *On maximal repetitions in words*, in FCT, G. Ciobanu and G. Paun, eds., vol. 1684 of Lecture Notes in Computer Science, Springer, 1999, pp. 374–385.
21. R. M. Kolpakov and G. Kucherov: *On maximal repetitions in words*, in FCT '99: Proceedings of the 12th International Symposium on Fundamentals of Computation Theory, London, UK, 1999, Springer-Verlag, pp. 374–385.
22. M. Lothaire: *Applied Combinatorics on Words*, Cambridge University Press, Cambridge, UK, 2005.
23. M. G. Main: *Detecting leftmost maximal periodicities.* Discrete Applied Mathematics, 25(1–2) 1989, pp. 145–153.
24. M. G. Main and R. J. Lorentz: *An $O(n \log n)$ algorithm for finding all repetitions in a string.* J. Algorithms, 5(3) 1984, pp. 422–432.
25. S. J. Puglisi, J. Simpson, and W. F. Smyth: *How many runs can a string contain?* Theor. Comput. Sci., 401(1-3) 2008, pp. 165–171.
26. W. Rytter: *The number of runs in a string: Improved analysis of the linear upper bound*, in STACS, B. Durand and W. Thomas, eds., vol. 3884 of Lecture Notes in Computer Science, Springer, 2006, pp. 184–195.
27. W. Rytter: *The number of runs in a string.* Inf. Comput., 205(9) 2007, pp. 1459–1469.
28. A. Thue: *Über unendliche Zeichenreihen.* Norske Vid. Selsk. Skr. I Math-Nat., 7 1906, pp. 1–22.

Parallel algorithms for degenerate and weighted sequences derived from high throughput sequencing technologies

Costas S. Iliopoulos[1,2], Mirka Miller[1,3,4], and Solon P. Pissis[1]

[1] Dept. of Computer Science, King's College London, London WC2R 2LS, England
csi@dcs.kcl.ac.uk, solon.pissis@kcl.ac.uk
[2] Digital Ecosystems & Business Intelligence Institute, Curtin University, GPO Box U1987 Perth WA 6845, Australia
[3] School of Electrical Engineering and Computer Science, The University of Newcastle Callaghan NSW 2308, Australia
[4] Dept. of Mathematics, University of West Bohemia, Pilsen, Czech Republic
mirka.miller@newcastle.edu.au

Abstract. Novel high throughput sequencing technologies have redefined the way genome sequencing is performed. They are able to produce millions of short sequences in a single experiment and with a much lower cost than previous methods. In this paper, we address the problem of efficiently mapping and classifying millions of degenerate and weighted sequences to a reference genome, based on whether they occur exactly once in the genome or not, and by taking into consideration probability scores. In particular, we design parallel algorithms for *Massive Exact and Approximate Unique Pattern Matching* for degenerate and weighted sequences derived from high throughput sequencing technologies.

Keywords: parallel algorithms, string algorithms, high throughput sequencing technologies

1 Introduction

The computational biology applications that have been developed for decades are strongly related to the technology that generates the data they consider. The algorithms, and the application parameters, are tuned in such a way that they abolished intrinsic limitations of the technology. As an example, the length of the data to be processed, or the quality/error rate that accompanies this data, are crucial elements that are considered for choosing the appropriate data structure for preprocessing, storing, analyzing, and comparing sequences. Moreover, the way solutions are improved reflects both computer science and biotechnology advances.

Among the large number of equipment that produce data, the DNA sequencers play a central role. DNA sequencing is the generic term for all biochemical methods that determine the order of the nucleotide bases in a DNA sequence. It consists of obtaining (generally relatively short) fragments of a DNA sequence (typically less than a thousand bp - base pair). The Sanger sequencing method [17,18] has been the workhorse technology for DNA sequencing for almost 30 years. It has been slowly replaced by technologies that used different colored fluorescent dyes [16,22] and polyacrylamide gels. Later, the gels were replaced by capillaries, increasing the length of individual obtained fragments from 450 to 850 bp. Despite the many technological advances, obtaining the complete sequence of a genome was carried out in very large

Costas S. Iliopoulos, Mirka Miller, Solon P. Pissis: Parallel algorithms for degenerate and weighted sequences derived from high throughput sequencing technologies, pp. 249–262.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

dedicated "sequencing factories", which require hundreds of automatic sequencers using highly automated pipelines.

Along the years, sophisticated algorithms have been developed for assembling whole genomes, from a simple bacterial genome [6] to the human genome [7]. These algorithms were following the progress of the sequencing technologies, and were fully taking into account all the biases introduced by the equipment.

Very recent advances, based either on sequencing-by-synthesis (SBS) or on hybridization and ligation, are producing millions of short reads overnight. Depending on the technology (454 Life Science, Solexa/Illumina or Polony Sequencing, to name a few), the size of the fragments can range from a dozen of base pairs to several hundreds.

These high throughput sequencing technologies have the potential to assemble a bacterial genome during a single experiment and at a moderate cost [8] and are aimed in sequencing DNA genomic sequences. One such technology, Pyrosequencing™, massively parallelises the sequencing via microchip sensors and nanofluids, and it produces reads that are approximately 200 bp long, and may not improve beyond 300 bp in the near future [24]. In contrast, the technology developed by the Solexa/Illumina [2], generates millions of very short mate-pair reads ranging from 25 [26] to 50 [8] bp long, although in the future this number may be increased to 75. The results of these new technologies mark the beginning of a new era of high throughput short read sequencing that moves away form the traditional Sanger methods. The common denominator of these technologies is the fact that they are able to produce a massive amount of relatively short reads. Due to this massive amount of data generated by the above systems, efficient algorithms for mapping short sequences to a reference genome are in great demand.

Popular alignment programs like *BLAST* or *BLAT* are not successful because they focus on the alignment of fewer and longer sequences [11]. Recently, a new thread of applications addressing the short sequences mapping problem has been devised for this particular objective. These applications are based on the pigeonhole principle, and make use of hashing and short key indexing techniques.

ELAND is the mapping algorithm developed as part of the Illumina pipeline. It is optimized to map very short reads of $20 - 32$ bp ignoring additional bases when the reads are longer, whilst allowing at most two mismatches between the read and the genomic sequence [21]. *SOAP* [14] supports multi-threaded parallel computing and allows up to two mismatches, or a gap of 1-3 bp without any other mismatch. *SeqMap* [11] allows up to 5 mixed mismatches and inserted/deleted nucleotides in mapping.

RMAP [21] and *MAQ* [13] are ungapped mapping programs, which take read qualities, base position probabilities, and mate-pair information into account. Their strategies resemble the strategies developed by Ewing *et al.* in [5], where at each position of the reads a quality score is assigned, which encodes the probability that the base at that position is either rightly or wrongly positioned.

The last two applications show the necessity for a measure of accuracy concerning the mapping methods. Accuracy can be quantified in terms of sensitivity and specificity. Possible causes of limitations in the accuracy of these experiments include sequencing errors arising from any part of the high throughput experiment, variation between sampled genome that generated the reads and the reference genome, as well as ambiguities caused by repeats in the reference genome [21].

Therefore, the limitations of the equipment used, or the natural polymorphisms that can be observed between individual samples can give rise to uncertain sequences, where in some positions more than one nucleotide can be present. These sequences, where more than one base are possible in certain positions, are called *degenerate* or *indeterminate* sequences.

Figure 1 presents the sequence logo of a degenerate DNA sequence, which is the consensus DNA sequence derived from different reads at the same location. In this consensus degenerate sequence, one can note that in some positions more than one base occurs, and in fact all bases (A, C, G, T in the case of DNA sequences) may occur.

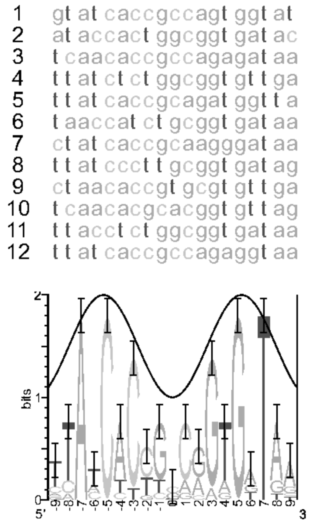


Figure 1. A sequence logo of a biological degenerate sequence. Picture taken from [19].

Degenerate string pattern matching has mainly been handled by bit mapping techniques (Shift-Or method) [1,28]. These techniques have been used to find matches for a degenerate pattern in a string [9], and the *agrep* utility [27] has been virtually one of the few practical algorithms available for degenerate pattern matching.

Very often, each position of a sequence is accompanied by probabilities of each base occuring in the specific position. In the case of the high throughput experiments, these quality scores, which accompany the raw sequence data, describe the confidence of bases in each read [21]. The sequencing quality scores assign a probability to the four possible nucleotides for each sequenced base. Bases with low quality scores are more likely to be sequencing errors. These sequences, where the probability of every symbol's occurrence at every location is given, are called *weighted* sequences.

Weighted sequences are also used to represent relatively short sequences such as binding sites, as well as long sequences such as protein families profiles [3]. Additionally, they have been used to represent complete chromosome sequences that were obtained using the traditional method of whole-genome shotgun strategy.

In this paper, we present parallel algorithms for addressing the problem of efficiently mapping uniquely occuring short reads to a reference genome. In particular, we design parallel algorithms for *Massive Exact and Approximate Unique Pattern Matching* for degenerate and weighted sequences derived from high throughput sequencing technologies. Our approach differs from the above mapping programs in three key points:

- it preprocesses the genomic sequence based on the reads length, by using word-level parallelism, before mapping the reads to it. This provides efficiency to the method.
- it does not index and hash the reads, but instead it converts each read to a unique arithmetic value. This results to a much higher sensitivity in terms of the number of reads perfectly mapped to the reference genome.
- it directly classifies the mapped reads into unique and duplicate matches, i.e. into reads that occur exactly once in the genome and into reads that occur more than once. The uniqueness of a mapped read guarantees an adequate placement on the sequence, and provides anchors that will be used for placing mate-pair reads, and other connected reads as well. It also identifies something that is totally region specific, while most of the genome is repetitive.

The rest of the paper is structured as follows. In Section 2, we present the preliminaries. In Section 3, we define the problem of Massive Exact and Approximate Unique Pattern Matching for degenerate and weighted sequences. In Section 4 and Section 5, we present the parallel algorithms for solving the exact and the approximate case, respectively. Finally, we briefly conclude with some future work in Section 6.

2 Preliminaries

A *string* is a sequence of zero or more symbols from an alphabet Σ. The set of all strings over Σ is denoted by Σ^*. The length of a string x is denoted by $|x|$. The *empty* string, that is the string of length zero, is denoted by ϵ. The i-th symbol of a string x is denoted by $x[i]$.

A string w is a *substring* of x if $x = uwv$, where $u, v \in \Sigma^*$. We denote by $x[i \dots j]$ the substring of x that starts at position i and ends at position j. Conversely, x is called a *superstring* of w. A string w is a *prefix* of x if $x = wy$, for $y \in \Sigma^*$. Similarly, w is a *suffix* of x if $x = yw$, for $y \in \Sigma^*$.

In this work, we are considering the finite alphabet Σ for DNA sequences, where $\Sigma = \{A, C, G, T\}$.

A *degenerate* string is a sequence $t = t[1 \dots n]$, where $t[i] \subseteq \Sigma$ for each i. When a position of the string is degenerate, and it can match more than one element from the alphabet Σ, we say that this position has *non-solid* symbol. If in a position only one element of the alphabet Σ is present, we refer to this symbol as *solid*.

A *weighted* string over alphabet Σ is a sequence $s = s[1 \dots n]$ of sets of couples. In particular, each $s[i]$ is a set $((q_1, \pi_i(q_1)), (q_2, \pi_i(q_2)), \dots, (q_{|\Sigma|}, \pi_i(q_{|\Sigma|}))$, where $\pi_i(q_j)$ is the occurrence probability of character q_j at position i. A symbol q_j occurs at position i of a weighted sequence $s = s[1 \dots n]$ if and only if the probability of occurrence of symbol q_j at position i is greater than zero, i.e. $\pi_i(q_j) > 0$. For every position $1 \leq i \leq n$, $\sum_{j=1}^{|\Sigma|} \pi_i(q_j) = 1$. For example, $\binom{A\ 0.8}{C\ 0.2}$ is a non-solid symbol, implying that base A occurs with probability 80 % and C with probability 20 %.

3 Problems definition

We denote the generated short reads as the set $p_0, p_1, \dots, p_{r-1}$ and we call them *patterns*. Notice that r is a very large integer number ($r > 10^7$). Due to the massive amount of data, specialized solutions are needed to various sequencing-related problems. The length ℓ of each pattern is nowadays typically between 25 and 50 bp long, and we denote that constant range, without loss of generality, as $\ell_{min} \leq \ell \leq \ell_{max}$. We assume that the data is derived from high quality sequencing methods and therefore we will consider patterns with at most $\mu = 3$ non-solid symbols. We are given a genomic solid sequence $t = t[1 \ldots n]$ and a positive threshold $k \geq 0$.

We define the *Massive Exact and Approximate Unique Pattern Matching* problem for degenerate and weighted sequences as follows.

Problem 1.
Find whether the degenerate pattern $p_i = p_i[1 \ldots \ell]$, for all $0 \leq i < r$, of length $\ell_{min} \leq \ell \leq \ell_{max}$, with at most μ non-solid symbols, occurs with at most k-mismatches in $t = t[1 \ldots n]$, exactly once.

Problem 2.
Find whether the weighted pattern $p_i = p_i[1 \ldots \ell]$, for all $0 \leq i < r$, of length $\ell_{min} \leq \ell \leq \ell_{max}$, with at most μ non-solid symbols, occurs with at most k-mismatches in $t = t[1 \ldots n]$, exactly once, with probability at least c, if $\prod_{i=1}^{\ell} \pi_i(q_i) \geq c$.

We mainly focus on the following classes of both problems:

Class 1. p_i occurs in t exactly once
Class 2. p_i occurs with at most 1-mismatch in t, exactly once
Class 3. p_i occurs with at most 2-mismatches in t, exactly once

Class 2 and Class 3 correspond to cases where the pattern either contains a sequencing error (quality score associated with read is indicating it), or a small difference between a mutant and the reference genome, which will have an impact on the proteins that have to be translated, as explored in [23,25].

4 Massive Exact Unique Pattern Matching in Parallel

In this section, we solve the problem of *Class 1*. The focus is to find occurrences of pattern p_i, for all $0 \leq i < r$, in text $t = t[1 \ldots n]$. In particular, we are interested in whether p_i occurs in t exactly once.

The proposed algorithm makes use of the message-passing paradigm, by using p processing elements. The following assumptions for the model of communications in the parallel computer are made. The parallel computer comprises a number of nodes. Each node comprises one or several identical processors interconnected by a switched communication network. The time taken to send a message of size n between any two nodes is independent of the distance between nodes and can be modelled as $t_{comm} = t_s + nt_w$, where t_s is the latency or start-up time of the message, and t_w is the transfer time per data. The links between two nodes are full-duplex and single-ported: a message can be transferred in both directions by the link at the same time, and only one message can be sent and one message can be received at the same time.

In addition, the proposed parallel algorithm makes use of word-level parallelism by compacting strings into single computer words that we call *signatures*. We get the signature $\sigma(x)$ of a string x, by transforming it to its binary equivalent using 2-bits-per-base encoding of the DNA alphabet (see Table 1), and packing its decimal value into a computer word (see Table 2).

The idea of employing signatures is long known to computer scientists, introduced by Dömölki in [4] in 1964 for his Shift-Or algorithm, a string matching algorithm based on only few bitwise logical operations. The most well known application is the *four Russians* algorithm, which packs rows of boolean matrices into computer words speeding up boolean matrix multiplication. A randomised version of *fingerprints* (modulo a prime number) was employed by Karp and Rabin in [12] for solving the pattern matching problem. Their method cannot be used in our pattern matching problem as our signatures are small and, thus, there is no practical speed up by reducing it modulo a prime number.

A	0 0
C	0 1
G	1 0
T	1 1

Table 1. Binary Encoding of DNA alphabet

String x	A G C A T
Binary form	0 0 1 0 0 1 0 0 1 1
Signature $\sigma(x)$	1 4 7

Table 2. Signature of $AGCAT$

Our aim is to preprocess text t and create two sets of lists $\Lambda_{\ell_{min}}, \ldots, \Lambda_{\ell_{max}}$ and $\Lambda'_{\ell_{min}}, \ldots, \Lambda'_{\ell_{max}}$. Each list Λ_ℓ, for all $\ell_{min} \leq \ell \leq \ell_{max}$, holds each duplicate substring of length ℓ of t. Each list Λ'_ℓ, for all $\ell_{min} \leq \ell \leq \ell_{max}$, holds each unique substring of length ℓ of t.

An outline of the parallel algorithm, for all $\ell_{min} \leq \ell \leq \ell_{max}$, is as follows:

Problem Partitioning. We use a *data decomposition* approach to partition the text t with the sliding window mechanism into a set of substrings $z_1, z_2, \ldots, z_{n-\ell+1}$, where $z_i = t[i \ldots i+\ell-1]$, for all $1 \leq i \leq n-\ell+1$.

Step 1. We assume that text t is stored locally on the master processor. We make sure that the load is evenly balanced by distributing $z_1, z_2, \ldots, z_{n-\ell+1}$ among the p available processors. Each processor ρ_q, for all $0 \leq q < p$, is allocated a fair amount a_q of substrings, as shown in Equation 1.

$$a_q = \begin{cases} \lceil \frac{n-\ell+1}{p} \rceil, & \text{if } q < n-\ell+1 \mod p \\ \lfloor \frac{n-\ell+1}{p} \rfloor, & \textit{otherwise} \end{cases} \tag{1}$$

We denote $z_{first_q}, \ldots, z_{last_q}$ as the set of the allocated substrings of length ℓ of processor ρ_q.

Example. Table 3 shows the processors allocation for the case of $t = GGGTCTA$, $\ell = 3$ and $p = 3$.

Step 2. Each processor ρ_q compacts each allocated substring z_i, for all $first_q \leq i \leq last_q$, into a signature $\sigma(z_i)$, packs it in a couple $(i,\ \sigma(z_i))$, where i represents the matching position of z_i in t, and adds the couple to a local list Z_q. Notice that, as

$\rho_\mathbf{q}$	$\mathbf{a_q}$	$\mathbf{first_q}$	$\mathbf{last_q}$	**Allocated substrings**
ρ_0	2	1	2	$z_1 = GGG$, $z_2 = GGT$
ρ_1	2	3	4	$z_3 = GTC$, $z_4 = TCT$
ρ_2	1	5	5	$z_5 = CTA$

Table 3. Processors allocation for $t = GGGTCTA$, $\ell = 3$ and $p = 3$

soon as we compact z_{first_q} into $\sigma(z_{\mathit{first}_q})$, then each $\sigma(z_i)$, for all $\mathit{first}_q + 1 \leq i \leq \mathit{last}_q$, can be retrieved in constant time (using "shift"-type of operation).

Step 3. We sort the local lists Z_q based on the signature's field, in parallel, using *Parallel Sorting by Regular Sampling* (PSRS) [20], a practical parallel deterministic sorting algorithm. Notice that parallel sorting means rearranging the elements of the local lists Z_q, so that each processor ρ_q still has a fair amount in Z_q, but with the smallest signatures stored in sorted order by processor ρ_0, ρ_1 etc.

Step 4. Each processor ρ_q runs sequentially through its sorted list Z_q and checks whether the signatures in $Z_q[x]$ and $Z_q[x+1]$ are equal, for all $0 \leq x < |A_q| - 1$. If they are equal, then ρ_q adds $Z_q[x]$ to a new list L_q. If not, then $Z_q[x]$ is added to a new list L'_q.

Step 5. Each processor ρ_q, for all $1 \leq q < p$, sends the first element in Z_q to the neighbour processor ρ_{q-1}. Then, each processor ρ_q, for all $0 \leq q < p - 1$, compares the signature of the last element in Z_q, to the signature of the element received from processor ρ_{q+1}. If they are equal, then processor ρ_q adds the element to the list L_q, else it is added to the list L'_q.

Step 6. We perform a *gather* operation, in which processor ρ_0 collects a unique message, local list L_q, from each processor ρ_q, for all $1 \leq q < p$, and stores each local list L_q in rank order, resulting in a new combined sorted list Λ_ℓ. We do the same with the local list L'_q, resulting in a new sorted combined list Λ'_ℓ. Processor ρ_0 performs a *one-to-all* broadcast to send both lists Λ_ℓ and Λ'_ℓ to all other processors.

Main Step. Assume that the two sets of lists $\Lambda_{\ell_{min}}, \ldots, \Lambda_{\ell_{max}}$ and $\Lambda'_{\ell_{min}}, \ldots, \Lambda'_{\ell_{max}}$ are created and stored on each processor ρ_q. We extend the set of patterns $p_0, p_1, \ldots, p_{r-1}$ to a new set $p'_0, p'_1, \ldots, p'_{r'-1}$, $r < r'$, as follows.

We make sure that each processor ρ_q is allocated a fair amount of query patterns from the set $p_0, p_1, \ldots, p_{r-1}$, in a similar way as in step 1.

1. **Problem 1.** For each degenerate pattern p_i of length ℓ with λ non-solid symbols, such that $\lambda \leq \mu$, we create $\prod_{j=1}^{\ell} |p_i[j]|$ new patterns, each differing in λ positions.
2. **Problem 2.** For each weighted pattern p_i of length ℓ with λ non-solid symbols, such that $\lambda \leq \mu$, we create $\prod_{j=1}^{\ell} |p_i[j]|$ new patterns, each differing in λ positions. We select each of those patterns, say $s = s[1 \ldots \ell]$, with $s[1] = (q_1, \pi_1(q_1))$, $s[2] = (q_2, \pi_2(q_2)), \ldots, s[\ell] = (q_\ell, \pi_\ell(q_\ell))$, that satisfy $\prod_{j=1}^{\ell} \pi_j(q_j) \geq c$.

Then, each processor can determine, by using a binary search, whether an allocated pattern p'_i of length ℓ occurs in t exactly once, in $\mathcal{O}(\log n)$ time. If $\sigma(p'_i) \in \Lambda'_\ell$, then p'_i is a unique pattern, and the algorithm returns its matching position in t. If $\sigma(p'_i) \in \Lambda_\ell$, then p'_i occurs in t more than once. If $\sigma(p'_i) \notin \Lambda_\ell$ and $\sigma(p'_i) \notin \Lambda'_\ell$, then p'_i does not occur in t.

Notice that in a case when $2\ell > w$, where w is the word size of the machine (e.g. 32 or 64 in practice), our algorithm can easily be adopted by storing the signatures in $\lceil 2\ell/w \rceil$ computer words.

Theorem 1. *Given the text $t = t[1 \ldots n]$, the set of patterns $p_0, p_1, \ldots, p_{r-1}$, the length of each pattern $\ell_{min} \leq \ell \leq \ell_{max}$, the word size of the machine w, and the number of processors p, the parallel algorithm solves the Class 1 of Problem 1 and Problem 2 in $\mathcal{O}(\lceil \ell_{max}/w \rceil (\frac{n}{p} \log \frac{n}{p} + \frac{r}{p} \ell_{max} \log n))$ computation time and $\mathcal{O}(n \log p + r)$ communication time.*

Proof. In step 1, assuming that the text t is kept locally on master processor, the data distribution can be done in $\mathcal{O}(t_s \log p + t_w \frac{n}{p}(p-1))$ communication time. In step 2, each processor creates a fair amount of signatures in $\mathcal{O}(\lceil \ell_{max}/w \rceil \frac{n}{p})$ computation time. In step 3, the *PSRS* algorithm can be executed in $\mathcal{O}(\lceil \ell_{max}/w \rceil \frac{n}{p} \log \frac{n}{p})$ computation time, where $n \geq p^3$, and $\mathcal{O}(n/\sqrt{p})$ communication time [20]. In step 4, the sequential run through the local list Z_q takes $\mathcal{O}(\lceil \ell_{max}/w \rceil \frac{n}{p})$ computation time. Step 5 involves $\mathcal{O}(1)$ point-to-point simple message transfers and comparisons. In step 6, the *gather* operation can be done in $\mathcal{O}(t_s \log p + t_w \frac{n}{p}(p-1))$, and the *one-to-all* broadcast take $\mathcal{O}((t_s + t_w n) \log p)$ communication time.

Assuming that the two sets (of a constant number) of lists are created, the main step runs in $\mathcal{O}(\lceil \ell_{max}/w \rceil \frac{r}{p} \ell_{max} \log n)$ computation time, for the binary search, and $\mathcal{O}(t_s \log p + t_w \frac{r}{p}(p-1))$ communication time, for the patterns distribution. Notice that, since $|\Sigma| = 4$ and $\mu = 3$, the number of the new created patterns is treated as constant.

Hence, asymptotically, the overall time is $\mathcal{O}(\lceil \ell_{max}/w \rceil (\frac{n}{p} \log \frac{n}{p} + \frac{r}{p} \ell_{max} \log n))$ computation time, and $\mathcal{O}(n \log p + r)$ communication time. □

5 Massive Approximate Unique Pattern Matching in Parallel

In this section we solve the problem of *Class 2* and *Class 3*. The focus is to find occurrences of p_i, for all $0 \leq i < r$, in text $t = t[1 \ldots n]$ with at most k-mismatches. In particular, we are interested in whether p_i occurs with at most 1-mismatch in t exactly once for the problem of *Class 2*, or with at most 2-mismatches exactly once for the problem of *Class 3*.

The proposed parallel algorithm makes use of the message-passing paradigm, by using p processing elements, and word-level parallelism, by compacting strings into signatures, and applying a bit-vector algorithm for efficient approximate string matching with mismatches.

Our aim is to preprocess text t and create two sets of lists $\Lambda_{\ell_{min}}, \ldots, \Lambda_{\ell_{max}}$ and $\Lambda'_{\ell_{min}}, \ldots, \Lambda'_{\ell_{max}}$. Each list Λ_ℓ, for all $\ell_{min} \leq \ell \leq \ell_{max}$, holds each duplicate substring of length ℓ of t with at most k-mismatches. Each list Λ'_ℓ, for all $\ell_{min} \leq \ell \leq \ell_{max}$, holds each unique substring of length ℓ of t.

5.1 The bit-vector algorithm for fixed-length approximate string matching with *k*-mismatches

Iliopoulos, Mouchard and Pinzon in [10] presented the Max-Shift algorithm, a bit-vector algorithm that solves the *fixed-length approximate string matching* problem:

given a text t of length n, a pattern ρ of length m and an integer ℓ, compute the optimal alignment of all substrings of ρ of length ℓ and a substring of t. The focus of the MAX-SHIFT algorithm is on computing matrix D', which contains the best scores of the alignments of all substrings of pattern ρ of length ℓ and any contiguous substring of the text t.

The MAX-SHIFT algorithm makes use of word-level parallelism in order to compute matrix D' efficiently, similar to the manner used by Myers in [15]. The algorithm is based on the $\mathcal{O}(1)$ time computation of each $D'[i,j]$ by using bit-vector operations, under the assumption that $\ell \leq w$, where w is the number of bits in a machine word or $\mathcal{O}(\ell/w)$-time for the general case. The algorithm maintains a bit-vector matrix $B[0 \ldots m, 0 \ldots n]$, where the bit integer $B[i,j]$, holds the binary encoding of the path in D' to obtain the optimal alignment at i, j with the differences occurring as leftmost as possible.

Here the key idea is to devise a bit-vector algorithm for the *fixed-length approximate string matching with at most k-mismatches* problem: given a text t of length n, a pattern ρ of length m and an integer ℓ, find all substrings of ρ of length ℓ that match any contiguous substring of t of length ℓ with at most k-mismatches. If we assign ρ=t, we can extract all the duplicate substrings of length ℓ of t with at most k-mismatches. The focus is on computing matrix M, which contains the number of mismatches of all substrings of pattern ρ of length ℓ and any contiguous substring of the text t of length ℓ.

Example. Let the text $t = \rho = GGGTCTA$ and $\ell = 3$. Table 4 shows the matrix M.

		0	1	2	3	4	5	6	7
		ϵ	G	G	G	T	C	T	A
0	ϵ	0	0	0	0	0	0	0	0
1	G	1	0	0	0	1	1	1	1
2	G	2	1	0	0	1	2	2	2
3	G	3	2	1	0	1	2	3	3
4	T	3	3	2	1	0	2	2	3
5	C	3	3	3	2	2	0	3	2
6	T	3	3	3	3	2	3	0	3
7	A	3	3	3	3	3	2	3	0

Table 4. Matrix M for $t = \rho = GGGTCTA$ and $\ell = 3$

We maintain the bit-vector $B[i,j] = b_\ell \cdots b_1$, where $b_\lambda = 1,\ 1 \leq \lambda \leq \ell$, if there is a mismatch of a contiguous substring of the text $t[i-\ell+1 \ldots i]$ and $t[j-\ell+1 \ldots j]$ in the λ^{th} position. Otherwise we set $b_\lambda = 0$.

Given the restraint that the integer ℓ is less than the length of the computer word w, then the bit-vector operations allow to update each entry of the matrix B in constant time (using "shift"-type of operation on the bit-vector). The maintenance of the bit-vector is done via operations defined as follows:

1. $shiftc(x)$: shifts and truncates the leftmost bit of x.
2. $\delta_H(x,y)$: returns the minimum number of replacements required to transform x into y

The Bit-Vector-Mismatches algorithm for computing the bit-vector matrix B and matrix M is outlined in Figure 2.

Bit-Vector-Mismatches
▷Input: t, n, ρ, m, ℓ
▷Output: B, M

```
1   begin
2     ▷ Initialization
3     B[0...m, 0] ← min(i, ℓ) 1's; B[0, 0...n] ← 0
4     M[0...m, 0] ← min(i, ℓ); M[0, 0...n] ← 0
5     ▷ Matrix B and Matrix M computation
6     for i ← 1 until m do
7       for j ← 1 until n do
8           B[i, j] ← shiftc(B[i − 1, j − 1]) OR δ_H(ρ[i], t[j])
9           M[i, j] ← ones(B[i, j])
10  end
```

Figure 2. The Bit-Vector-Mismatches algorithm for computing matrix B and matrix M

Example. Let the text $t = \rho = GGGTCTA$ and $\ell = 3$. Table 5 shows the bit-vector matrix B. Consider the case when $i = 7$ and $j = 5$. Cell $B[7,5] = 101$ denotes that substrings $t[3\dots 5] = CTA$ and $t[5\dots 7] = GTC$ have a mismatch in position 1, a match in position 2, and a mismatch in position 3, resulting in a total of two mismatches, as shown in cell $M[7,5]$.

		0	1	2	3	4	5	6	7
		ϵ	G	G	G	T	C	T	A
0	ϵ	0	0	0	0	0	0	0	0
1	G	1	0	0	0	1	1	1	1
2	G	11	10	00	00	01	11	11	11
3	G	111	110	100	000	001	011	111	111
4	T	111	111	101	001	000	011	110	111
5	C	111	111	111	011	011	000	111	101
6	T	111	111	111	111	110	111	000	111
7	A	111	111	111	111	111	101	111	000

Table 5. The bit-vector matrix B for $t = \rho = GGGTCTA$ and $\ell = 3$

Assume that the bit-vector matrix $B[0\dots m, 0\dots n]$ is given. We can use the function $ones(v)$, which returns the number of 1's (bits set on) in the bit-vector v, to compute matrix M (see Figure 2, line 9).

5.2 The parallel algorithm

The key idea behind parallelising the Bit-Vector-Mismatches algorithm, is that cell $B[i, j]$ can be computed only in terms of $B[i-1, j-1]$ (see Figure 2, line 8).

An outline of the parallel algorithm, for all $\ell_{min} \leq \ell \leq \ell_{max}$, is as follows:

Problem Partitioning. We use a *functional decomposition* approach, in which the initial focus is on the computation that is to be performed rather than on the data manipulated by the computation. We assume that the text t (and the pattern $\rho = t$) is stored locally on each processor. This can be done by using a *one-to-all* broadcast operation in $(t_s + t_w n) \log p$ communication time, which is asymptotically $\mathcal{O}(n \log p)$. We partition the problem of computing matrix B (and M) into a set of diagonal vectors $\Delta_0, \Delta_1, \ldots, \Delta_{n+m}$, as shown in Equation 2.

$$\Delta_\nu[x] = \begin{cases} B[\nu - x,\ x] \ : \ 0 \leq x \leq \nu, & \text{(a)} \\ B[m - x,\ \nu - m + x] \ : \ 0 \leq x < m + 1, & \text{(b)} \\ B[m - x,\ \nu - m + x] \ : \ 0 \leq x < n + m - \nu + 1, & \text{(c)} \end{cases} \quad (2)$$

where,
(a) if $0 \leq \nu < m$
(b) if $m \leq \nu < n$
(c) if $n \leq \nu < n + m + 1$

Step 1. We make sure that the load is evenly balanced among the p available processors in each diagonal Δ_ν. Each processor ρ_q, for all $0 \leq q < p$, is allocated a fair amount $a_q[\nu]$ of cells in each diagonal Δ_ν, as shown in Equation 3.

$$a_q[\nu] = \begin{cases} \lceil \frac{|\Delta_\nu|}{p} \rceil, & \text{if } q < |\Delta_\nu| \mod p \\ \lfloor \frac{|\Delta_\nu|}{p} \rfloor, & \textit{otherwise} \end{cases} \quad (3)$$

We denote $\Delta_\nu[\mathit{first}_q[\nu]], \ldots, \Delta_\nu[\mathit{last}_q[\nu]]$ as the set of the allocated cells of processor ρ_q in diagonal Δ_ν.

Step 2. Each processor ρ_q computes each allocated cell $\Delta_\nu[x]$, for all $\mathit{first}_q[\nu] \leq x \leq \mathit{last}_q[\nu]$, in each diagonal Δ_ν, using the Bit-Vector-Mismatches algorithm.

Step 3. It is possible that in a certain diagonal Δ_ν, $\nu > 0$, a processor will need a cell or a pair of cells, which were not computed on its local memory in diagonal $\Delta_{\nu-1}$. We need a communication pattern in each diagonal Δ_ν, for all $0 \leq \nu < n + m$, which minimises the data exchange between the processors. It is obvious, that in each diagonal, each processor needs only to communicate with its neighbours (boundary cells swaps).

Step 4. On every occasion a processor ρ_q computes a cell $M[i, j] \leq k$, where $i \geq \ell$ and $j \geq \ell$, we notice two possible cases:

1. if $M[i, j] = 0$ and $i = j$, then substring $t[i - \ell + 1 \ldots i]$ occurs in t at least once. We compact substring $t[i - \ell + 1 \ldots i]$ into a signature $\sigma(t[i - \ell + 1 \ldots i])$, pack it in a couple $(i - \ell + 1, \sigma(t[i - \ell + 1 \ldots i]))$, and add the couple to a new list Z_q.

2. if $i \neq j$, then substrings $t[i-\ell+1\dots i]$ and $t[j-\ell+1\dots j]$ are considered to be duplicates with at most k-mismatches. We compact both substrings into the signatures $\sigma(t[i-\ell+1\dots i])$ and $\sigma(t[j-\ell+1\dots j])$, pack them in couples $(i-\ell+1, \sigma(t[i-\ell+1\dots i]))$ and $(j-\ell+1, \sigma(t[j-\ell+1\dots j]))$, and add the couples to the list Z_q.

Step 5. Assume that the diagonal supersteps $\Delta_0, \Delta_1, \dots, \Delta_{n+m}$ are executed. The local lists Z_q are constructed, and so, we follow the steps 3-6 of the parallel algorithm in Section 4.

Main step. Assume that the two sets of lists $\Lambda_{\ell_{min}}, \dots, \Lambda_{\ell_{max}}$ and $\Lambda'_{\ell_{min}}, \dots, \Lambda'_{\ell_{max}}$ are created and stored on each processor ρ_q. We extend the set of patterns $p_0, p_1, \dots, p_{r-1}$ to a new set $p'_0, p'_1, \dots, p'_{r'-1}$, $r < r'$, as in Section 4. Then, each processor can determine by using a binary search, whether an allocated pattern p'_i of length ℓ occurs in t exactly once, in $\mathcal{O}(\log n)$ time. If $\sigma(p'_i) \in \Lambda'$, then p'_i is a unique pattern with at most k-mismatches, and the algorithm returns its matching position in t. If $\sigma(p'_i) \in \Lambda_\ell$, then p'_i occurs in t more than once.

Notice that, in a case where $\sigma(p'_i) \notin \Lambda_\ell$ and $\sigma(p'_i) \notin \Lambda'_\ell$, then p'_i does not occur in t, and we can check whether the k-mismatches occur inside the pattern p'_i as follows.

1. **Class 2 and Class 3.** We construct a new set of patterns x_j, for all $0 \leq j < |\Sigma|.\ell$, differing from p'_i in one position, and we compact each x_j into a signature $\sigma(x_j)$. If $\sigma(x_j) \in \Lambda'_\ell$, then p'_i is a unique pattern with at most 1-mismatch, and the algorithm returns its matching position in t. If $\sigma(x_j) \in \Lambda_\ell$, then we discard pattern p'_i as it has to occur in t exactly once. If $\sigma(x_j) \notin \Lambda_\ell$ and $\sigma(x_j) \notin \Lambda'_\ell$ then p'_i does not occur in t.
2. **Class 3.** We construct a new set of patterns y_j, for all $0 \leq j < |\Sigma|^2.\binom{\ell}{2}$, differing from p'_i in two positions, and we compact each y_j into a signature $\sigma(y_j)$. If $\sigma(y_j) \in \Lambda'_\ell$, then p'_i is a unique pattern with at most 2-mismatches, and the algorithm returns its matching position in t. If $\sigma(y_j) \in \Lambda_\ell$, then we discard pattern p'_i as it has to occur in t exactly once. If $\sigma(y_j) \notin \Lambda_\ell$ and $\sigma(y_j) \notin \Lambda'_\ell$ then p'_i does not occur in t.

In general, for the problem of k-mismatches, for each pattern p'_i of length ℓ that does not occur in t, we construct k new sets of patterns, each containing $|\Sigma|^\lambda.\binom{\ell}{\lambda}$ patterns differing from p'_i in λ positions, for all $1 \leq \lambda \leq k$.

Theorem 2. *Given the text $t = t[1\dots n]$, the set of patterns $p_0, p_1, \dots, p_{r-1}$, the length of each pattern $\ell_{min} \leq \ell \leq \ell_{max}$, the word size of the machine w, and the number of processors p, the parallel algorithm solves the Class 2 and Class 3 of Problem 1 and Problem 2 in $\mathcal{O}(\lceil \ell_{max}/w \rceil (\frac{n^2}{p} + \frac{\ell^3_{max} r}{p} \log p))$ computation time, and $\mathcal{O}(n \log p + r)$ communication time.*

Proof. We partition the problem of computing matrix B (and M) into a set of $n+m+1$ diagonal vectors, thus $\mathcal{O}(n)$ supersteps (since $n = m$). In step 1, the allocation can be done in $\mathcal{O}(1)$ time. In step 2, the cells computation requires $\mathcal{O}(\lceil \ell_{max}/w \rceil \frac{n}{p})$ time. In step 3, the data exchange between the processors involves $\mathcal{O}(1)$ point-to-point simple message transfers. In step 4, the local lists Z_q are constructed in $\mathcal{O}(\lceil \ell_{max}/w \rceil \frac{n}{p})$ time. Assuming that the diagonal supersteps are executed, step 5 can be done in $\mathcal{O}(\lceil \ell_{max}/w \rceil (\frac{n}{p} \log \frac{n}{p}))$ computation time, and $\mathcal{O}(n \log p)$ communication time (see Section 4).

Assuming that the two sets (of a constant number) of lists are created, the main step runs in $\mathcal{O}(\lceil \ell_{max}/w \rceil (\frac{\ell_{max}^3 r}{p} \log n))$ computation time, for the binary search, and $\mathcal{O}(t_s \log p + t_w \frac{r}{p}(p-1))$ communication time, for the patterns distribution.

Hence, asymptotically, the overall time is $\mathcal{O}(\lceil \ell_{max}/w \rceil (\frac{n^2}{p} + \frac{\ell_{max}^3 r}{p} \log p))$ computation time, and $\mathcal{O}(n \log p + r)$ communication time. □

Also, the space complexity can be reduced to $\mathcal{O}(n)$ by noting that each diagonal Δ_ν depends only on diagonal $\Delta_{\nu-1}$.

6 Conclusion

In this paper, we have presented parallel algorithms to tackle the data emerging from the new high throughput sequencing technologies in biology. The new technologies produce a huge number of very short sequences and these sequences need to be classified, tagged and recognised as parts of a reference genome. Our algorithms can manipulate this data for degenerate and weighted sequences for Massive Exact and Approximate Unique Pattern matching. Implementation of the algorithms described in this paper is under way and will be presented in the near future.

References

1. R. Baeza-Yates and G. Gonnet: *A new approach to text searching*. Communications of the ACM, 35 1992, pp. 74–82.
2. S. Bennett: *Solexa ltd*. Pharmacogenomics, 5(4) 2004, pp. 433–438.
3. M. Christodoulakis, C. S. Iliopoulos, L. Mouchard, K. Perdikuri, A. Tsakalidis, and K. Tsichlas: *Computation of repetitions and regularities on biological weighted sequences*. Journal of Computational Biology, 13(6) 2006, pp. 1214–1231.
4. B. Dömölki: *An algorithm for syntactic analysis*. Computational Linguistics, 8 1964, pp. 29–46.
5. B. Ewing, L. Hillier, M. C. Wendl, and P. Green: *Base-Calling of Automated Sequencer Traces UsingPhred.I. Accuracy Assessment*. Genome Research, 8(3) 1998, pp. 175–185.
6. R. D. Fleischmann, M. D. Adams, O. White, R. A. Clayton, E. F. Kirkness, A. R. Kerlavage, C. J. Bult, J. F. Tomb, B. A. Dougherty, and J. M. Merrick: *Whole-genome random sequencing and assembly of* Haemophilus influenzae. Science, 269 1995, pp. 496–512.
7. C. Genomics: *The sequence of the human genome*. Science, 291 2001, pp. 1304–1351.
8. D. Hernandez, P. Francois, L. Farinelli, M. Osteras, and J. Schrenzel: *De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer*. Genome Res, March 2008.
9. J. Holub, W. F. Smyth, and S. Wang: *Fast pattern-matching on indeterminate strings*. J. of Discrete Algorithms, 6(1) 2008, pp. 37–50.
10. C. S. Iliopoulos, L. Mouchard, and Y. Pinzon: *The max-shift algorithm for approximate string matching*, in Proceedings of the 5th Workshop on Algorithm Engineering (WAE'01), Aarhus, Denmark, 2001, G.S. Brodal and D. Frigioni and A.Marchetti-Spaccamela, eds., pp. 13–25.
11. H. Jiang and W. H. Wong: *Seqmap: mapping massive amount of oligonucleotides to the genome*. Bioinformatics, 24(20) 2008, pp. 2395–2396.
12. R. M. Karp and M. O. Rabin: *Efficient randomized pattern-matching algorithms*. IBM J. Res. Dev., 31(2) 1987, pp. 249–260.
13. H. Li, J. Ruan, and R. Durbin: *Mapping short DNA sequencing reads and calling variants using mapping quality scores*. Genome Research, 18(11) 2008, pp. 1851–1858.
14. R. Li, Y. Li, K. Kristiansen, and J. Wang: *SOAP: short oligonucleotide alignment program*. Bioinformatics, 24(5) 2008, pp. 713–714.

15. E. W. Myers: *A fast bit-vector algorithm for approximate string matching based on dynamic progamming*. Journal of the ACM 46, 1999, pp. 395–415.
16. J. M. Prober, G. L. Trainor, R. J. Dam, F. W. Hobbs, C. W. Robertson, R. J. Zagursky, A. J. Cocuzza, M. A. Jensen, and K. Baumeister: *A system for rapid DNA sequencing with fluorescent chain-terminating dideoxynucleotides*. Science, 238 1987, pp. 336–341.
17. F. Sanger and A. R. Coulson: *A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase*. J. Mol. Biol., 94 1975, pp. 441–448.
18. F. Sanger, S. Nicklen, and A. R. Coulson: *DNA sequencing with chain-terminating inhibitors*. Proc. Natl. Acad. Sci. USA, 74 1977, pp. 5463–5467.
19. M. C. Shaner, I. M. Blair, and T. D. Schneider: *Sequence logos: A powerful, yet simple, tool*, in Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences, Volume 1: Architecture and Biotechnology Computing, T. N. Mudge, V. Milutinovic, and L. Hunter, eds., IEEE Computer Society Press, 1993, pp. 813–821.
20. H. Shi and J. Schaeffer: *Parallel sorting by regular sampling*. J. Parallel Distrib. Comput., 14(4) 1992, pp. 361–372.
21. A. Smith, Z. Xuan, and M. Zhang: *Using quality scores and longer reads improves accuracy of solexa read mapping*. BMC Bioinformatics, 9(1) 2008, p. 128.
22. L. M. Smith, J. Z. Sanders, R. J. Kaiser, P. Hughes, C. Dodd, C. R. Connell, C. Heiner, S. B. Kent, and L. E. Hood: *Fluorescence detection in automated DNA sequence analysis*. Nature, 321 1986, pp. 674–679.
23. M. Sultan, M. H. Schulz, H. Richard, A. Magen, A. Klingenhoff, M. Scherf, M. Seifert, T. Borodina, A. Soldatov, D. Parkhomchuk, D. Schmidt, S. O'Keeffe, S. Haas, M. Vingron, H. Lehrach, and M.-L. Yaspo: *A Global View of Gene Activity and Alternative Splicing by Deep Sequencing of the Human Transcriptome*. Science, 321(5891) 2008, pp. 956–960.
24. A. Sundquist, M. Ronaghi, H. Tang, P. Pevzner, and S. Batzoglou: *Whole-genome sequencing and assembly with high-throughput, short-read technologies*. PLoS ONE, 2 2007.
25. Z. Wang, M. Gerstein, and M. Snyder: *Rna-seq: a revolutionary tool for transcriptomics*. Nature reviews. Genetics, November 2008.
26. R. L. Warren, G. G. Sutton, S. J. Jones, and R. A. Holt: *Assembling millions of short DNA sequences using SSAKE*. Bioinformatics, 23(4) February 2007, pp. 500–501.
27. S. Wu and U. Manber: *Agrep- a fast approximate pattern-matching tool*. Proceedings USENIX Winter 1992 Technical Conference, San Francisco, CA, 1992, pp. 153–162.
28. S. Wu and U. Manber: *Fast text searching: allowing errors*. Commun. ACM, 35(10) 1992, pp. 83–91.

Finding all covers of an indeterminate string in $O(n)$ time on average

Md. Faizul Bari, M. Sohel Rahman, and Rifat Shahriyar

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka, Bangladesh
{faizulbari, msrahman, rifat}@cse.buet.ac.bd

Abstract. We study the problem of finding all the covers of an indeterminate string. An indeterminate string is a sequence $T = T[1]T[2]\cdots T[n]$, where $T[i] \subseteq \Sigma$ for each i, and Σ is a given alphabet of fixed size. Here we describe an algorithm for finding all the covers of a string x. The algorithm is applicable for both regular and indeterminate strings. Our algorithm starts with the border array and uses pattern matching technique of the Aho-Corasick Automaton to compute all the covers of x from the border array. On average the algorithm requires $O(n)$ time to find out all the covers, where n is the length of x. Finally, we extend our algorithm to compute the cover array of x in $O(n^2)$ time and $O(n)$ space complexity.

Keywords: indeterminate strings, covers, cover array, Aho-Corasick Automaton, string regularities

1 Introduction

Characterizing and finding regularities in strings are important problems in many areas of science. In molecular biology, repetitive elements in chromosomes determine the likelihood of certain diseases. In probability theory, regularities are important in the analysis of stochastic processes. In computer science, repetitive elements in strings are important in e.g. data compression, computational music analysis, coding, automata and formal language theory. As a result, in the last 20 years, string regularities have drawn a lot of attention from different disciplines of science.

The most common forms of string regularities are periods and repeats and there are several $O(n \log n)$ time algorithms for finding repetitions [6,8], in a string x, where n is the length of x. Apostolico and Breslauer [4] gave an optimal $O(\log \log n)$-time parallel algorithm for finding all the repetitions of a string of length n. The preprocessing of the Knuth-Morris-Pratt algorithm [14] finds all periods of every prefix of x in linear time.

In many cases, it is desirable to relax the meaning of repetition. For instance, if we allow overlapping and concatenations of periods in a string we get the notion of *covers*. After periods and repeats, cover is the most popular form of regularities in strings. The idea of cover generalizes the idea of periods or repeats. A substring c of a string x is called a *cover* of x if and only if x can be constructed by concatenation and superposition of c. Another common string regularity is the *seed* of a string. A seed is an extended cover in the sense that it is a cover of a superstring of x.

Clearly, x is always a cover of itself. If a proper substring pc of x is also a cover of x, then pc is called a *proper cover* of x. For example, the string $x = abcababcabcabcab$ has covers x and $abcab$. Here, $abcab$ is a proper cover. A string that has a proper cover is called *coverable*; otherwise it is *superprimitive*. The notion of covers was introduced

Md. Faizul Bari, M. Sohel Rahman, Rifat Shahriyar: Finding all covers of an indeterminate string in $O(n)$ time on average, pp. 263–271.
Proceedings of PSC 2009, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04403-2

by Apostolico, Farach and Iliopoulos in [5], where a linear-time algorithm to test the superprimitivity of a string was given. Moore and Smyth in [16] gave linear time algorithms for finding all covers of a string.

In this paper, we are interested in regularities of indeterminate strings. An indeterminate string is a sequence $T = T[1]T[2]\cdots T[n]$, where $T[i] \subseteq \Sigma$ for $1 \leq i \leq n$, and Σ is a given fixed alphabet. If $|T[i]| > 1$, then the position i of T is referred to as an *indeterminate position*. The simplest form of indeterminate string is one in which each indeterminate position can only contain a don't care character, denoted by '*'; the don't care character matches any letter in the alphabet Σ. Effectively, $* = \{\sigma_i \in \Sigma \mid 1 \leq i \leq |\Sigma|\}$. The pattern matching problem with don't care characters has been solved by Fischer and Paterson [9] more than 30 years ago. However, although the algorithm in [9] is efficient in theory, it is not very useful in practice. Pattern matching problem for indeterminate string has also been investigated in the literature, albeit with little success. For example, in [13], an algorithm was presented which works well only if the alphabet size is small. Pattern matching for indeterminate strings has mainly been handled by bit mapping techniques (Shift-Or method) [7,18]. These techniques have been used to find matches for an indeterminate pattern p in a string x [11] and the `agrep` utility [17] has been virtually one of the few practical algorithms available for indeterminate pattern-matching.

In [11] the authors extended the notion of indeterminate string matching by distinguishing two distinct forms of indeterminate match, namely, *quantum* and *deterministic*. Roughly speaking, a quantum match allows an indeterminate letter to match two or more distinct letters during a single matching process; a determinate match restricts each indeterminate letter to a single match [11].

Very recently, the issue of regularities in indeterminate string has received some attention. For example, in [2], the authors investigated the regularities of *conservative* indeterminate strings. In a conservative indeterminate string the number indeterminate positions is bounded by a constant. The authors in [2] presented efficient algorithms for finding the smallest *conservative cover* (number of indeterminate position in the cover is bounded by a given constant), λ-conservative covers (conservative covers having length λ) and λ-conservative seeds. On the other hand, Antoniou et al. presented an $O(n \log n)$ algorithm to find the smallest cover of an indeterminate string in [3] and showed that their algorithm can be easily extended to compute all the covers of x. The later algorithm runs in $O(n^2 \log n)$ time.

In this paper, we devise an algorithm for computing all the covers of an indeterminate string x of length n in $O(n^2)$ time in the worst case. We also show that our algorithm works in $O(n)$ time on the average. We also extend our algorithm to compute the cover array of x in $O(n^2)$ time and $O(n)$ space complexity in the worst case. Notably, our algorithm, unlike the algorithm of [2], does not enforce the restriction that the cover or the input string x must be conservative.

The rest of this paper is organized as follows. Section 2 gives account of definitions and notations. Section 3 presents our algorithm to find out all the covers of x. In Section 4, we extend our algorithm to compute the cover array. Finally, Section 5 gives the conclusions.

2 Preliminaries

A string is a sequence of zero or more symbols from an alphabet Σ. The set of all strings over Σ is denoted by Σ^*. The *length* of a string x is denoted by $|x|$. The *empty string*, the string of length zero, is denoted by ϵ. The i-th symbol of a string x is denoted by $x[i]$. A string $w \in \Sigma$, is a *substring* of x if $x = uwv$, where $u, v \in \Sigma^*$. We denote by $x[i \dots j]$ the substring of x that starts at position i and ends at position j. Conversely, x is called a *superstring* of w. A string $w \in \Sigma$ is a *prefix* (*suffix*) of x if $x = wy$ ($x = yw$), for $y \in \Sigma^*$. A string w is a *subsequence* of x (or x a *supersequence* of w) if w is obtained by deleting zero or more symbols at any positions from x. For example, *ace* is a subsequence of *abcabbcde*. For a given set S of strings, a string w is called a common subsequence of S if s is a subsequence of every string in S.

The string xy is the *concatenation* of the strings x and y. The concatenation of k copies of x is denoted by x^k. For two strings $x = x[1 \dots n]$ and $y = y[1 \dots m]$ such that $x[n-i+1 \dots n] = y[1 \dots i]$ for some $i \geq 1$ (i.e., x has a suffix equal to a prefix of y), the string $x[1 \dots n]y[i+1 \dots m]$ is said to be a *superposition* of x and y. We also say that x *overlaps* with y. A substring y of x is called a *repetition* in x, if $x = uy^kv$, where u, y, v are substrings of x and $k \geq 2$, $|y| \neq 0$. For example, if $x = aababab$, then a (appearing in positions 1 and 2) and ab (appearing in positions 2, 4 and 6) are repetitions in x; in particular $a^2 = aa$ is called a *square* and $(ab)^3 = ababab$ is called a *cube*. A non-empty substring w is called a *period* of a string x, if x can be written as $x = w^kw'$ where $k \geq 1$ and w' is a prefix of w. The shortest period of x is called *the period* of x. For example, if $x = abcabcab$, then abc, $abcabc$ and the string x itself are periods of x, while abc is the period of x.

A substring w of x is called a *cover* of x, if x can be constructed by concatenating or overlapping copies of w. We also say that w covers x. For example, if $x = ababaaba$, then aba and x are covers of x. If x has a cover $w \neq x$, x is said to be *quasiperiodic*; otherwise, x is *superprimitive*. The *cover array* γ, is a data structure used to store the length of the longest proper cover of every prefix of x; that is for all $i \in \{1, \dots, n\}$, $\gamma[i]$ = length of the longest proper cover of $x[1 \dots i]$ or 0. In fact, since every cover of any cover of x is also a cover of x, it turns out that, the cover array γ describes all the covers of every prefix of x. A substring w of x is called a *seed* of x, if w covers a superstring of x[1]. For example, aba and $ababa$ are some seeds of $x = ababaab$. A border u of x is a prefix of x that is also a suffix of x; thus $u = x[1 \dots b] = x[n-b+1 \dots n]$ for some $b \in \{0, \dots, n-1\}$. The border array of x is an array β such that for all $i \in \{1, \dots, n\}$, $\beta[i]$ = length of the longest border of $x[1 \dots i]$. Since every border of any border of x is also a border of x, β encodes all the borders of every prefix of x. Depending on the matching of letters, borders of indeterminate strings can be of two types, namely, the *quantum border* and the *deterministic border*. Roughly speaking, a quantum match allows an indeterminate letter to match two or more distinct letters during a single matching process, whereas, a determinate match restricts each indeterminate letter to a single match. The notion of these two type of borders was introduced in [10].

An indeterminate string is a sequence $T = T[1]T[2] \cdots T[n]$, where $T[i] \subseteq \Sigma$ for each i, and Σ is a given alphabet of fixed size. When a position of the string is indeterminate, and it can match more than one element from the alphabet Σ, we say that this position has non-solid symbol. If in a position we have only one element of the alphabet Σ present, then we refer to this symbol as solid. In an indeterminate string a non-solid position can contain up to $|\Sigma|$ symbols. So, to check whether the two positions match in the traditional way, we would need $|\Sigma|^2$ time in the worst

[1] Note that, x is a superstring of itself. Therefore, every cover is also a seed but the reverse is not necessarily true.

case. To perform this task efficiently we use the following idea originally borrowed from [15] and later used in [3,2]. We use a bit vector of length $|\Sigma|$ as follows:

$$\forall T \subseteq \Sigma, \qquad \nu[T] = [\nu_{t_1}, \nu_{t_2}, \ldots, \nu_{t_{|\Sigma|}}],$$
$$\text{where } \forall t_i \in \Sigma \qquad \nu_{t_i} = \begin{cases} 1, \text{ if } t_i \in T \\ 0, \text{ otherwise} \end{cases} \tag{1}$$

During the string matching process instead of comparing $T[i]$ with $T[j]$, we test bit vectors $\nu[T[i]]$ and $\nu[T[j]]$ using bit operation *AND*. Thus we can compare any two symbols in constant time since the alphabet size is fixed.

In our algorithm we heavily use the Aho-Corasick Automaton invented by Aho and Corasick in [1]. The Aho-Corasick Automaton for a given finite set P of patterns is a Deterministic Finite Automaton G accepting the sets of all words containing a word of P as a suffix. More formally, $G = (Q, \Sigma, g, f, q_0, F)$, where function Q is the set of states, Σ is the alphabet, g is the forward transition, f is the failure link i.e. $f(q_i) = q_j$, if and only if S_j is the longest suffix of S_i that is also a prefix of any pattern, q_0 is the initial state and F is the set of final (terminal) states [1]. The construction of the AC automaton can be done in $O(d)$-time and space complexity, where d is the size of the dictionary, i.e. the sum of the lengths of the patterns which the AC automata will match.

3 Our Algorithm

We start with a formal definition of the problem we handle in this paper.

Problem 1. Computing All Covers of an Indeterminate String over a fixed alphabet.
Input: We are given an indeterminate string x, of length n on a fixed alphabet Σ.
Output: We need to compute all the covers of x.

Our algorithm depends on the following facts:

Fact 1. *Every cover of string x is also a border of x.*

Fact 2. *If u and c are covers of x and $|u| < |c|$ then u must be a cover of c.*

The running time analysis of our algorithm depends on Lemma 3 which is a extension of the analysis provided in [12] by Iliopoulos et al. where they have showed that number of borders of a regular string with a don't care is bounded by 3.5 on average. Here we have extended it for indeterminate strings and proved that, the expected number of borders of an indeterminate string is also bounded by a constant.

Lemma 3. *The expected number of borders of an indeterminate string is bounded by a constant.*

Proof (Proof of Lemma 3). We suppose that the alphabet Σ consists of ordinary letters 1, 2, ..., α, $\alpha \geq 2$. First we consider the probability of two symbols of a string being equal. Equality occurs in the following cases:

Symbol	Match To	Number of cases
$\sigma \in \{1, 2, \ldots, \alpha\}$	$\sigma \in \{1, 2, \ldots, \alpha\}$	α
$\sigma \in S,\, S \subseteq \Sigma$	$\sigma \in S,\, S \subseteq \Sigma,\, \lvert S\rvert > 1$	$\sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}$

Thus the total number of equality cases is $\alpha + \sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}$ and the number of overall cases is $2^{2\alpha}$. Therefore the probability of two symbols of a string being equal is

$$\frac{\alpha + \sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}}{2^{2\alpha}}$$

Now let's consider the probability of string x having a border of length k. If we let $P[\mathit{expression}]$ denotes the probability that the *expression* hold, then

$$\begin{aligned} P\Big[x[1\ldots k] = x[n-k+1\ldots n]\Big] &= P\Big[x[1] = x[n-k+1]\Big] \times \cdots \times P\Big[x[k] = x[n]\Big] \\ &= \left(\frac{\alpha + \sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}}{2^{2\alpha}}\right)^k \end{aligned}$$

From this it follows that the expected number of borders is

$$\sum_{k=1}^{n-1} \left(\frac{\alpha + \sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}}{2^{2\alpha}}\right)^k$$

This summation assumes its maximum value when α is equal to 12 and the summation is bounded above by

$$\begin{aligned} \sum_{k=1}^{n-1} \left(\frac{\alpha + \sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}}{2^{2\alpha}}\right)^k &\leq \left(\frac{16221211}{16777216}\right) + \left(\frac{16221211}{16777216}\right)^2 + \cdots \\ &\quad + \left(\frac{16221211}{16777216}\right)^{n-1} \\ &= \frac{1 - \left(\frac{16221211}{16777216}\right)^n}{1 - \left(\frac{16221211}{16777216}\right)} - 1 \\ &= \frac{1}{\frac{556005}{16777216}} - 1 \\ &= 29.1746 \end{aligned}$$

So, the expected number of borders of an indeterminate string is bounded by 29.1746. $\square$

Now by using Fact 1, we can compute all the covers of x from its border array. This can be done simply by checking each border and finding out whether it covers x or not. Our algorithm is based on this approach. Broadly speaking, our algorithm consists of two steps. In the first step, the deterministic border array of x is computed.

For this we have used the algorithm introduced by Holub and Smyth in [10], that can compute the deterministic border array of an indeterminate string x in expected $O(n)$ time and space complexity, where n is the length of x. In the second step, we check each border whether it is a cover of x or not. Utilizing Fact 2 and the pattern matching technique of Aho-Corasick Automaton, this step can be performed in $O(n)$ time and space complexity on average. In what follows, we explain the steps of the algorithm in more details.

3.1 First Step: Computing the Border Array

In the first step, we utilize the algorithm provided by Holub and Smyth [10] for computing the deterministic border of an indeterminate string. The output of the algorithm is a two dimensional list β. Each entry β_i of β contains a list of pair (b, ν_a), where b is the length of the border and ν_a represents the required assignment and the list is kept sorted in decreasing order of border lengths of $x[1 \dots i]$. So the first entry of β_i represents the largest border of $x[1 \dots i]$. The algorithm is given below just for completeness.

Algorithm 1 Computing deterministic border array of string x

$\beta_i[j] \leftarrow \phi, \quad \forall i, j \in \{1..n\}$
$\nu_i \leftarrow \nu[x[i]], \quad \forall i \in \{1..n\}$ {set bit vector for each $x[i]$}
for $i \leftarrow 1$ to $n-1$ **do**
  for all $b, \beta_i[b] \neq \phi$ **do**
    if $2b - i + 1 < 0$ **then**
      $p \leftarrow \nu_{b+1}\ AND\ \nu_{i+1}$
    else
      $p \leftarrow \beta_i[b+1]\ AND\ \nu_{i+1}$
    end if
    if $p \neq \mathbf{0}^{|\Sigma|}$ **then**
      $\beta_{i+1}[b+1] \leftarrow p$
    end if
  end for
  $p \leftarrow \nu_1\ AND\ \nu_{i+1}$
  if $p \neq \mathbf{0}^{|\Sigma|}$ **then**
    $\beta_{i+1}[1] \leftarrow p$
  end if
end for

If we assume that the maximum number of borders of any prefix of x is m, then the worst case running time of the algorithm is $O(nm)$. But from Lemma 3 we know that the expected number of borders of an indeterminate string is bounded by a constant. As a result the expected running time of the above algorithm is $O(n)$.

3.2 Second Step: Checking the Border for a Cover

In the second step, we find out the covers of string x. Here we need only the last entry of the border array, β_n, where $n = |x|$. If $\beta_n = \{b_1, b_2, b_3\}$ then we can say that x has three borders, namely $x[1 \dots b_1]$, $x[1 \dots b_2]$ and $x[1 \dots b_3]$ of length b_1, b_2 and b_3 respectively and $b_1 > b_2 > b_3$. If the number of borders of x is m then number of entry in β_n is m. We iterate over the entries of β_n and check each border in turn to find out whether it covers x or not. To identify a border as a cover of x we use the pattern matching technique of an Aho-Corasick automaton. Simple speaking, we build an Aho-Corasick automaton with the dictionary containing the border of x and parse x through the automaton to find out whether x can be covered by it or not. Suppose in iteration i, we have the length of the ith border of β_n equal to b. In this iteration, we build an Aho-Corasick automaton for the following dictionary:

$$D = \{x[1]x[2]\cdots x[b]\}, \qquad \text{where } \forall j \in \text{ 1 to b }, \; x[j] \in \Sigma \tag{2}$$

Then we parse the input string x through the automaton to find out the positions where the pattern $c = x[1 \ldots b]$ occurs in x. We store the starting index of the occurrences of c in x in a position array P of size $n = |x|$. We initialize P with all entries set to zero. If c occurs at index i of x then we set $P[i] = 1$. Now if the distance between any two consecutive 1's is greater than the length of the border b then the border fails to cover x, otherwise c is identified as a cover of x. We store the length of the covers in an array AC. At the end of the process AC contains the length of all the covers of x. The definition of AC can be given as follows:

$$AC = \{c_1, c_2, \ldots, c_k\}, \qquad \text{where } \forall i \in 1 \text{ to k}, \; c_i \text{ is a cover of } x \tag{3}$$

Algorithm 2 formally presents the steps of a function $isCover()$, which is the heart of Step 2 described above.

Algorithm 2 Function isCover(x, c)

1: Construct the Aho-Corasick automaton for c
2: parse x and compute the positions where c occurs in x and put the positions in the array Pos
3: **for** $i = 2$ to $|Pos|$ **do**
4:     **if** $Pos[i] - Pos[i-1] > |c|$ **then**
5:         Return FALSE
6:     **end if**
7: **end for**
8: Return TRUE

The time and space complexity of Algorithm 2 can be deduced as follows. Clearly, Steps 3 and 2 run in $O(n)$. Now, the complexity of Step 1 is linear in the size of the dictionary on which the automaton is build. Here the length of the string in the dictionary can be $n-1$ in the worst case. So, the time and space complexity of this algorithm is $O(n)$.

A further improvement in running time is achieved as follows. According to Fact 2, if u and c are covers of x and $|u| < |c|$ then u must be a cover of c. Now if $\beta_n = \{b_1, b_2, \ldots, b_m\}$ then from the definition of border array $b_1 > b_2 > \cdots > b_n$. Now if in any iteration we find a b_i that is a cover of x then from Fact 2, we can say that for all $j \in \{i+1, \ldots, m\}$, if b_j is a cover of x if and only if b_j is a cover of b_i. So instead of parsing x we can parse b_i for the subsequent automatons and as $|b_i| \leq |x|$ this policy improves the overall running time of the algorithm. Algorithm 3 formally present the overall process of Step 2 described above.

Algorithm 3 Computing all covers of x

$k \leftarrow n$
$AC \leftarrow \phi$ $\{AC$ is a list used to store the covers of $x\}$
for all $b \in \beta_n$ **do**
  if $isCover(x[1..k], x[1..b]) = true$ **then**
    $m \leftarrow b$
    $AC.Add(k)$
  end if
end for

The running time of Algorithm 3 is $O(nm)$, where m is number of borders of x or alternatively number of entries in β_n. Again, from Lemma 3 we can say that the number of borders of an indeterminate string is bounded by a constant on average. Hence, the expected running time of Algorithm 3 is $O(n)$.

It follows from above that our algorithm for finding all the covers of an indeterminate string of length n runs in $O(n)$ time on the average. The worst case complexity of our algorithm is $O(nm)$, i.e., $O(n^2)$, which is also an improvement since the current best known algorithm [3] for finding all covers requires $O(n^2 \log n)$ in the worst case.

4 Computing Cover Array

The cover array γ, is to store the length of the longest proper cover of every prefix of x; that is for all $i \in \{1, \ldots, n\}$, $\gamma[i]$ = length of the longest proper cover of $x[1 \ldots i]$ or 0. Our algorithm can readily be extended to compute the cover array of x. Algorithm 3 can be used here after some modification to compute the cover array of x. Here we only need the length of the largest border of each prefix of x. This information is stored in the first entry of each β_i of the border array. If the border array is implemented using any traditional two dimensional list even then the first entry of each list can be accessed in constant time. Let us assume that $\beta_i[1]$ denotes the first entry of the list β_i that is $\beta_i[1]$ is the length of the largest border of $x[1 \ldots i]$.

Algorithm 4 Computing cover array γ of x

$\gamma[i] \leftarrow 0 \quad \forall\, i \in \{1 \ldots n\}$
for $i \leftarrow 1$ to n **do**
  if $isCover(x, x[1 \ldots \beta_{i[1]}]) = true$ **then**
    $\gamma[i] \leftarrow \beta_{i[1]}$
  end if
end for

The $isCover(x, c)$ function (Algorithm 2) is used to identify the covers of x. The algorithm iterates over the first entries of the border array β and checks the borders one by one. If the border $x[1 \ldots \beta_i[1]]$ is identified as a cover then $\gamma[i]$ is set equal to $\beta_i[1]$. otherwise its set to zero. The steps are formally presented in Algorithm 4.

As the worst case running time of the $isCover(x, c)$ function is $O(n)$ and the algorithm iterates over the n lists of the border array β, the running time of Algorithm 4 is $O(n^2)$. The space requirement to store the cover array γ is clearly linear in the length of x, so the space complexity is $O(n)$.

5 Conclusions

In this paper we have presented an average case $O(n)$ time and space complex algorithm for computing all the covers of a given indeterminate string x of length n. We have also presented an algorithm for computing the cover array γ of an indeterminate string. This algorithm requires $O(n^2)$ time and $O(n)$ space in the worst case. Both of these algorithms are improvement over existing algorithms.

References

1. A. V. Aho and M. J. Corasick: *Efficient string matching: an aid to bibliographic search*. Communications of the ACM, 18(6) 1975, pp. 333–340.
2. P. Antoniou, M. Crochemore, C. S. Iliopoulos, I. Jayasekera, and G. M. Landau: *Conservative string covering of indeterminate strings*. Proceedings of the Prague Stringology Conference 2008, 2008, pp. 108–115.
3. P. Antoniou, C. S. Iliopoulos, I. Jayasekera, and W. Rytter: *Computing repetitive structures in indeterminate strings*. Proceedings of the 3rd IAPR International Conference on Pattern Recognition in Bioinformatics (PRIB 2008), 2008.
4. A. Apostolico and D. Breslauer: *An optimal $O(\log\log n)$-time parallel algorithm for detecting all squares in a string*. SIAM Journal of Computing, 25(6) 1996, pp. 1318–1331.
5. A. Apostolico, M. Farach, and C. S. Iliopoulos: *Optimal superprimitivity testing for strings*. Information Processing Letters, 39(11) 1991, pp. 17–20.
6. A. Apostolico and F. P. Preparata: *Optimal off-line detection of repetitions in a string*. Theory of Computer Science, 22 1983, pp. 297–315.
7. R. Baeza-Yates and G. H. Gonnet: *A new approach to text searching*. Communications of the ACM, 35(10) 1992, pp. 74–82.
8. M. Crochemore: *An optimal algorithm for computing the repetitions in a word*. Information Processing Letters, 12(5) 1981, pp. 244–250.
9. M. J. Fischer and M. S. Paterson: *String-matching and other products*. Technical report, Cambridge, MA, USA, 1974.
10. J. Holub and W. F. Smyth: *Algorithms on indeterminate strings*. Miller, M., Park, K. (eds.): Proceedings of the 14th Australasian Workshop on Combinatorial Algorithms AWOCA'03, 2003, pp. 36–45.
11. J. Holub, W. F. Smyth, and S. Wang: *Fast pattern-matching on indeterminate strings*. Journal of Discrete Algorithms, 6(1) 2008, pp. 37–50.
12. C. S. Iliopoulos, M. Mohamed, L. Mouchard, K. G. Perdikuri, W. F. Smyth, and A. K. Tsakalidis: *String regularities with don't cares*. Nordic Journal of Computing, 10(1) 2003, pp. 40–51.
13. C. S. Iliopoulos, L. Mouchard, and M. S. Rahman: *A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching*. Mathematics in Computer Science, 1(4) 2008, pp. 557–569.
14. 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.
15. I. Lee, A. Apostolico, C. S. Iliopoulos, and K. Park: *Finding approximate occurrences of a pattern that contains gaps*, in Proceedings 14-th Australasian Workshop on Combinatorial Algorithms, SNU Press, 2003, pp. 89–100.
16. D. Moore and W. F. Smyth: *An optimal algorithm to compute all the covers of a string*. Information Processing Letters, 50(5) 1994, pp. 239–246.
17. S. Wu and U. Manber: *Agrep – a fast approximate pattern-matching tool*. In Proceedings USENIX Winter 1992 Technical Conference, San Francisco, CA, 1992, pp. 153–162.
18. S. Wu and U. Manber: *Fast text searching: allowing errors*. Communications of the ACM, 35(10) 1992, pp. 83–91.

Author Index