Proceedings of the Prague Stringology Conference '04

Edited by Milan Šimánek and Jan Holub

August 2004

Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13
121 35 Prague 2
Czech Republic

Program Committee

Gabriela Andrejková, Jun-ichi Aoe, Maxime Crochemore, Jan Holub,
Costas S. Iliopoulos, Thierry Lecroq, Bořivoj Melichar (chair), Yoan J. Pinzon, Marie-France Sagot, Bruce W. Watson

Organizing Committee

Miroslav Balík, Jan Holub, Bořivoj Melichar, Milan Šimánek

URL

`http://cs.felk.cvut.cz/psc`

Proceedings of the Prague Stringology Conference '04
Published by Vydavatelství ČVUT, Zikova 4, 16635 Praha 6, Czech Republic
Edited by Milan Šimánek and Jan Holub
Contact: Prague Stringology Club
Katedra počítačů, ČVUT–FEL
Karlovo nám. 13, Praha 2, Czech Republic.
E-mail: psc@cs.felk.cvut.cz Phone: +420-2-2435-7470
Printed by Ediční středisko ČVUT, Zikova 4, Praha 6

ISBN 80-01-03050-4

Contents

Preface

The Prague Stringology Conference 2004 (PSC'04) was held at the Department of Computer Science and Engineering of the Czech Technical University in Prague, Czech Republic, on August 30–September 1, 2004. 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 programme committee and thirteen were selected for presentation at the conference, based on originality and quality. This volume contains not only these selected papers but also invited talk devoted to 2-dimensional pattern matching.

In the years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conferences in 2001–2003 preceded this conference. The proceedings of these workshops and the conferences had been published by Czech Technical University and are available on WWW pages of the Prague Stringology Club (PSC). Selected contributions were published in special issues of the journal Kybernetika, the Nordic Journal of Computing and the Journal of Automata, Languages and Combinatorics.

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 PSC is to study algorithms on strings and sequences with emphasis on finite automata theory. The first event organized by PSC was the workshop PSCW'96 featuring only a handful 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, 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'04 as well as the reviewers. Special thanks goes to all the members of the programme committee, without whose efforts it would not have been possible to put together such a stimulating program of PSC'04. 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 2004
Jan Holub

Theoretical Issues of Searching Aerial Photographs: A Bird's Eye View*

Amihood Amir

Bar-Ilan University and Georgia Tech
Department of Computer Science
52900 Ramat-Gan
ISRAEL

e-mail: `amir@cs.biu.ac.il`

Abstract. We review some pattern matching algorithms and techniques motivated by the discrete theory of image processing.

The problem inspiring this research is that of searching an aerial photograph for all appearances of some object.

The issues we discuss are digitization, local errors, rotation and scaling.

We review deterministic serial techniques that are used for multidimensional pattern matching and discuss their strengths and weaknesses.

Keywords: multidimensional pattern matching, local errors, rotations, scaling, digitization, discrete image processing.

1 Motivation

String Matching is one of the most widely studied problems in computer science [35]. Part of its appeal is in its direct applicability to "real world" problems. Some variation of the Boyer-Moore [21] algorithm is directly implemented in the search command of practically all text editors. The longest common subsequence dynamic programming algorithm [22] is implemented in the *UNIX* "`diff`" command. The largest overlap heuristic for finding the shortest common superstring [46] is used in DNA sequencing. In this respect string matching is somewhat of an anomaly in theoretical computer science. Theoretical algorithms can not often be used as "off-the-shelf" solutions for practical problems.

We consider one of the important roles of theoretical computer science, that of providing an algorithmic theory for various application domains. Usually that process starts with abstracting the practical problem into several "pure form" combinatorial problems. It is hoped that understanding the solution of these problems will aid in an efficient solution of the original application. In this paper we review some of the algorithms and techniques that were motivated by image processing. Covering all aspects of the problem is clearly a mammoth undertaking. We concentrate on serial

*Partially supported by NSF grant CCR-01-04494 and ISF grant 82/01.

deterministic algorithms. The reader should be aware that there is also a wide body of work on probabilistic, randomized, and parallel approaches to the problem.

The main practical motivation for this survey is the problem of **searching aerial photographs**. The (ambitious) practical goal of this application is to input an aerial photograph and a template of some object (a pattern). The output is all locations on the aerial photograph where the template object appears. In reality we need to grapple with several problems:

1. Local errors. These may arise from atmospheric distortions, transmission noise, level of detail, the digitization process, or occlusion of pattern parts by other objects.

2. Scaling. The size of the object in the input aerial photo may differ from that in the template.

3. Rotation. The object may be facing different directions in the aerial photograph and in the template.

There are other issues of interest in two dimensional matching. Among them are:

Compressed Matching: Digitized images are known to be extremely space consuming. However, regularities in the images can be exploited to reduce the necessary storage area. Thus we find that many systems store images in a compressed form (e.g. jpeg). If searching for appearances of a pattern in an image can be done *without decompressing* then compression becomes a **time saving** tool in addition to its being a space saving device. We will not delve into compression issues in this paper, although many of the techniques mentioned here have been used in compressed matching algorithms.

Dictionary Matching: The aerial photograph model we described is by no means the only possible vision paradigm. While it may be important to find an object in a given image, biological vision is somewhat an "inverse" of that process. Rather than searching for a small template in a large image, we have in our minds a tremendous database of objects we have seen (or imagined). When presented with an image we recognize it with amazing speed. Thus it is interesting to come up with efficient algorithms for quickly recognizing objects from a preprocessed dictionary, in a given image. As in the case of compressed matching, we will not have the opportunity to say much of this exciting area of research, although here too many algorithms use techniques that will be addressed herein.

For ease of perusal we enclose a table of contents for this paper:

2 Exact Matching

The most restrictive possible abstraction of the aerial photograph problem is that of seeking an *exact matching* of the pattern in the image, where both pattern and image are *rectangles*. Throughout this paper we define our problems in terms of *squares* rather than *rectangles*. In almost all cases the reason is simply for convenience of notation, and the results directly generalize. We explicitly mention those results that only apply to squares.

The *Exact Two Dimensional Matching Problem* is defined as follows: Let Σ be an alphabet. *INPUT: Text* array $T[n \times n]$ and *pattern* array $P[m \times m]$. *OUTPUT:* All locations $[i, j]$ in T where there is an *occurrence* of the pattern, i.e. $T[i+k, j+l] = P[k+1, l+1] \;\; 0 \leq k, l \leq n-1$.

2.1 Linear Reductions

A natural way of solving any generalized problem is by reducing it to a special case whose solution is known. It is not surprising that the early solutions to the two dimensional exact matching problem use exact string matching algorithms in one way or another.

The Knuth-Morris-Pratt [39] algorithm basically follows the idea of matching the text and pattern character by character until a mismatch occurs. Then the pattern is slid forward for the greatest overlap with the old pattern position, and the comparison resumes from there. This idea can be generalized in the following way:

Starting from the leftmost column and moving to the right, proceed down the columns and compare a *pattern row* with a *length-m text subrow* starting at the scanned text location. Proceed in this fashion until a mismatch occurs. Upon a mismatch, slide the pattern *down* for the greatest overlap with the old pattern position and resume comparisons from there.

The idea is obvious, but its straightforward implementation would take time $O(n^2m)$, since this is a basic KMP on the text, but every comparison takes time $O(m)$. It is an improvement over the naive $O(n^2m^2)$ algorithm, but not good enough. What is needed is a method for *quick* comparison of a pattern row and a length-m text subrow. Two solutions are possible.

2.1.1 Automata Methods

Bird [20] and, independently, Baker [18] proposed the following solution. Preprocess the text and identify all occurrences of all pattern rows. Represent each different row by a new symbol and place this symbol at the text location where the row occurs. The problem is now exactly that of string matching, where we are seeking all occurrences of the string composed of the new symbols in the order that their respective rows appear in the pattern. The string matching part can be done in time $O(n^2)$. The only question is how to efficiently identify the occurrences of all pattern rows.

This was done by using the Aho and Corasick [2] dictionary matching algorithm. The *Dictionary Matching Problem* is the following: Preprocess a *dictionary* of patterns $\{P_1 = p_{11} \cdots p_{1m_1}, P_2 = p_{21} \cdots p_{2m_2}, ..., P_k = p_{k1} \cdots p_{km_k}\}$. Subsequently, for every *INPUT:* Text $T = t_1 \cdots t_n$. *OUTPUT:* All locations in the text where there is a match with any dictionary pattern.

Aho and Corasick preprocess the dictionary in time $O(\sum_{i=1}^{k} m_i \log|\Sigma|)$ and subsequently search an input text in time $O(n \log|\Sigma| + occ)$, where occ = number of patterns that occur in the text. If all patterns are of the same length then only a single pattern can end at any text location. The time then becomes $O(n \log|\Sigma|)$.

Returning to two dimensional matching. We may view each distinct pattern row as a separate pattern in a dictionary. The result is a dictionary matching problem where all dictionary patterns have the same length. Thus the Bird-Baker solution is the following:

1. Find all occurrences of all pattern rows in the text. Mark the end of each distinct row's occurrence with a new special symbol.
2. Scan the text down the columns, from left to right. Run the Knuth-Morris-Pratt (KMP) algorithm searching for the *string* composed of the new symbols representing the distinct pattern rows.

Time: 1. Using Aho and Corasick, $O(n^2 \log|\Sigma|)$. 2. $O(n^2)$.

Total Time: $O(n^2 \log|\Sigma|)$.

2.1.2 Suffix Tree Methods

Recall that our aim is to use the KMP algorithm for solving the exact two dimensional matching problem. What we seek is a method for constant time comparison of pattern rows with length-m text subrows (and with each other). Bird and Baker solved this problem by performing the comparisons in advance. In this section we will see a method where this comparison can be done while scanning the text.

Definition: Let $S = s_1 \cdots s_n$ be a string. A *suffix tree* of S is a trie of all suffixes of S (i.e. $\{s_n,\ s_{n-1}s_n,\ s_{n-2}s_{n-1}s_n,\ ...,\ s_2s_3 \cdots s_n,\ s_1s_2 \cdots s_n\}$) where every path of single outdegree node is compressed to a single node.

Many different methods for constructing suffix trees and suffix arrays have been developed [49, 44, 26, 47]. The importance of suffix trees for our purpose is that they has the following properties: 1) The leaves represent exactly the suffixes of S, and 2) The *lowest common ancestor* (LCA) of any two nodes is the *longest common prefix* of the strings they represent.

We can thus make the following observation [15]. Let S be a string composed of the concatenation of all text rows followed by all pattern rows. The following queries are equivalent:

1. Pattern row P_{i_0} equals text subrow $T_{i_1 j}$, where $j_0 \leq j \leq j_0 + k - 1$.
2. The length of the longest common prefix of the suffixes of S starting at P_{i_0} and $T_{i_1 j_1}$ is at least m.
3. The length of the LCA in S's suffix tree of the nodes representing the suffixes that start at P_{i_0} and $T_{i_1 j_1}$ is at least m.

The suffix tree for the concatenation of the text and pattern rows can be constructed in time $O(n^2 \log\log|\Sigma|)$. All we need now is a method for finding the lowest common ancestor of two nodes in a tree in constant time.

It was pointed out by Landau and Vishkin [41], that the Harel and Tarjan [37] algorithm does precisely that. Harel and Tarjan showed that given any n-node tree, one can preprocess the tree in time $O(n)$ in a manner that allows subsequent LCA queries in time $O(1)$.

We now have all the components for a different exact two dimensional matching algorithm [15].

1. Construct the text and pattern suffix tree and preprocess for LCA.
2. Scan the text down the columns, from left to right. Run the KMP algorithm modified in a way that symbol comparison is replaced by checking if the LCA length is at least m.

Time: 1. $O(n^2 \log\log|\Sigma|)$ 2. $O(n^2)$.

Total Time: $O(n^2 \log\log|\Sigma|)$.

It should be stressed that more modern and direct methods for solving this problem exist, using suffix arrays [38]. Also, other algorithms for computing the LCA in a tree exist [45, 19, 24].

2.2 Convolutions

Convolutions were officially introduced to the field of pattern matching Fischer and Paterson [27]. Denote by $A \otimes B$ the *convolution* of arrays A and B. A convolution uses two initial functions, A and B, to produce a third function $A \otimes B$. We formally define a discrete convolution.

Definition: Let A be a real-valued function whose domain is $\{0, ..., n\}$ and B a real-valued function whose domain is $\{0, ..., m\}$. We may view A and B as arrays of numbers, whose lengths are $n+1$ and $m+1$, respectively. The *discrete convolution of A and B* is the polynomial multiplication

$$A \otimes B[j] = \sum_{i=1}^{m} A[j+i]B[i], \quad j = 0, ..., n-m.$$

In the general case, the convolution can be computed by using the Fast Fourier Transform (FFT) [25]. This can be done in time $O(n \log m)$, in a computational model with word size $O(\log m)$. Fischer and Paterson used convolutions for solving exact string matching with "don't cares" (a special character that matches all symbols) in time $O(\log |\Sigma| n \log m)$.

We can use the string matching with don't care problem to solve the two dimensional matching problem as follows (actually the same idea can be used for d-dimensional matching [13]).

Without loss of generality, we may assume that the text is of size $2m \times 2m$. This can be achieved by dividing the text into four overlapping grids of $2m \times 2m$ matrices. In the following discussion we assume,then, that $n = 2m$.

Let $T_{n\times n}$ be the text matrix, $P_{m\times m}$ be the pattern matrix. Let $L_{[1:n^2]}$ be the linear representation of T in row major order, and let $M_{[1:(m-1)n+m]}$ be the vector:

$$M_i = \begin{cases} P_{1,i} & \text{for } i = 1 \text{ to } m \\ \phi & \text{for } i = m+1 \text{ to } n \\ P_{2,i-n} & \text{for } i = n+1 \text{ to } m+n \\ \phi & \text{for } i = n+m+1 \text{ to } 2n \\ . & \\ . & \\ . & \\ P_{m,i-(m-1)n} & \text{for } i = (m-1)n+1 \text{ to } (m-1)n+m \end{cases}$$

Matching M in L (and taking care of boundary conditions) is equivalent to matching P in T. What is actually being done is padding the pattern with wildcards up to the size of the text dimension. The boundary condition can then be handled on line.

Time: The reduction is to string matching with don't cares with text of size n^2 and pattern of size $O(mn)$. This is solved by the convolutions method in time $O(|\log \Sigma| n^2 \log m)$, but since $n = 2m$ the time is $O(|\log \Sigma| m^2 \log m)$. For a general n the time is $O(|\log \Sigma| n^2 \log m)$.

2.3 Periodicity Analysis

All the previously discussed two dimensional matching algorithms are reductions of the problem into one dimension. These reductions all cost at least an additional $\log |\Sigma|$ factor. A uniquely two dimensional approach to pattern matching was introduced [4]. This technique analyzes the two dimensional structure of the pattern and makes use of it in the text scanning step. We will see that this allows an alphabet independent text scanning. This technique also proved useful in compressed matching [7], and in developing optimal parallel algorithms for two dimensional matching [8, 23].

The two-dimensional periodicity idea: A periodic pattern may contain locations, other than the origin, where the pattern can be superimposed on itself without mismatch. Suppose our pattern is *non periodic*, i.e. there are no such locations, other than the origin. We could then narrow down the number of *potential candidates* for a pattern appearance in the text in a fashion that insures that all such candidates are "sufficiently far" from each other. Verification of a candidate could then be done in the naive character-by-character comparison, but the time would still be linear because the candidates do not overlap.

In the next sections we will look more closely into two dimensional periodicity and its application to exact matching.

2.3.1 Two Dimensional Periodicity

In a periodic string, a smallest period can be found whose concatenation generates the entire string. In two dimensions, if an array were to extend infinitely so as to cover the plane, the one-dimensional notion of a period could be generalized to a unit cell of a lattice. But, a rectangular array is not infinite and may cut a unit cell in many different ways at its edges.

Instead of defining two-dimensional periodicity on the basis of some subunit of the array, Amir and Benson [4] use the idea of *self-overlap*. This idea applies also to strings. A string w is periodic if the longest prefix p of w that is also a suffix of w is at least half the length of w. For example, if $w = abcabcabcab$, then $p = abcabcab$ and since it is over half as long as w, w is periodic. This definition implies that w may overlap itself starting in the fourth position.

The preceding idea easily generalizes to two-dimensions as illustrated by the following preliminary definitions. Let A be an array. A *prefix* of A is a rectangular subarray that contains one corner element of A. A *suffix* is a rectangular subarray that contains the diagonally opposite corner. A is *periodic* if the largest prefix that is also a suffix has dimensions greater than half those of A. Again, this implies that A may overlap itself if the prefix of one copy of A is aligned with the suffix of a second copy of A.

Notice that the choice of the corner in which to put the prefix is arbitrary. Because of the symmetry, the prefix may be assigned to either the upper left or lower left corners of A. This clearly gives us two directions in which A can be periodic. Following [4] we classify the type of periodicity of A based on whether it has periodicity in either or both of these directions. To simplify the discussion, we describe square arrays. The results can be extended to all rectangular arrays (see [4]).

We begin with some formal definitions of two-dimensional periodicity and related concepts. Let $A[0 \ldots m-1, 0 \ldots m-1]$ be an $m \times m$ square array. Each element of A contains a symbol from an alphabet Σ. We can divide the array into four *quadrants*, labeled in a counterclockwise direction from upper left, quadrants *I*,*II*,*III*, and *IV*. Given two copies of A, one directly on top of the other. The two copies are said to be *in register* because some (in this case all) of the elements overlap, and overlapping elements contain the same symbol. If we can slide the upper copy over the lower copy to a point where the copies are again in register, then at least one of the corner elements of the upper array will overlap an element of the lower array. The element in the lower copy that is under this corner is the *source*. We say that the array is *quadrant I symmetric* if an overlapping corner is element $A(0,0)$. The element in the lower copy is a *quadrant I source*. Quadrants II, III and IV symmetry and sources are similarly defined.

Let the array be quadrant I symmetric and let the upper and lower copies be in register when element $A(0,0)$ overlaps element $A(r,c)$, the source. Then there exists a *quadrant I symmetry vector* $\vec{v}_I = r\vec{y} + c\vec{x}$ where $\vec{x}$ is the horizontal unit vector in the direction of increasing column index and $\vec{y}$ is the vertical unit vector in the direction of increasing row index. If the array is quadrant II symmetric, then the upper and lower copies are in register when $A(m-1,0)$ overlaps $A(r,c)$. The *quadrant II symmetry vector* is $v_{II} = (r-(m-1))\vec{y} + c\vec{x}$. Note that the coefficient on $\vec{y}$ is negative for quadrant II. The quadrants III and IV symmetry vectors are defined similarly.

The *length* of a symmetry vector is the maximum of the absolute values of its coefficients. If the length of a symmetry vector is $< \frac{m}{2}$, then the vector is *periodic*.

For the classification scheme, we need to pick the shortest symmetry vector for each of quadrants I and II. But, there may be several shortest vectors in a given quadrant. Also, the same orthogonal vector may be shortest in both quadrants. Let B_I be the set of shortest non-vertical vectors in quadrant I and let B_{II} be the set of shortest non-horizontal vectors in quadrant II. The *basis vectors for array* A are vector $\vec{v}_1 \in B_I$ (if any) with smallest r value and the vector $\vec{v}_2 \in B_{II}$ (if any) with smallest c value. In other words, $\vec{v}_1$ is the closest to horizontal in B_I and $\vec{v}_2$ is the closest to vertical in B_{II}.

The four categories of two-dimensional periodicity are:

- **Non-periodic**—A has no periodic vectors.

- **Lattice periodic**—Both quadrants I and II of A have a periodic basis vector. All quadrant I sources which occur in quadrant I fall on the nodes of a lattice which is defined by these vectors. The same is true for quadrant II sources in quadrant II. Specifically, let $\vec{v}_1$ and $\vec{v}_2$ be the periodic basis vectors in quadrants I and II respectively. Then, for all integers i, j such that $(0,0) + i\vec{v}_1 + j\vec{v}_2$ is an element of quadrant I, that element is a quadrant I source and no other elements in quadrant I are quadrant I sources. Similarly, for all $\hat{\imath}, \hat{\jmath}$ such that $(m-1,0) + \hat{\imath}\vec{v}_1 + \hat{\jmath}\vec{v}_2$ is an element of quadrant II, that element is a quadrant II source and no others.

- **Line periodic**—One quadrant of A has a periodic vector and one does not. The sources in the quadrant with the periodic vector all fall on one line. Specifically, if quadrant I is the quadrant with the periodic basis vector $\vec{v}_1$, then for all i such that $(0,0) + i\vec{v}_1$ is an element in quadrant I, that element is a quadrant I source and no others.

- **Radiant periodic**—This category is identical to the line periodic category, except that in the quadrant with the periodic vector, the sources fall on several lines which all radiate from the quadrant's corner. We do not describe the exact location of the sources because these depend on the specific array, except we note that none is a linear combination of both basis vectors for the array.

It should be noted that later applications required some finer grained characterizations of periodicity [36, 14].

2.3.2 The Dueling Method

Dueling was first used by Vishkin for efficient parallel string matching algorithms [48]. The idea is to provide, in constant time, a method that eliminates one of two competing candidates for pattern occurrence. This elimination is based on identifying locations where the two candidates expect conflicting symbols. Vishkin used string periodicity properties to guarantee that such locations exist for every two overlapping candidates.

The dueling idea was extended to two dimensions by amir, Benson and Farach [6, 5]. It turned out that even where there is no periodicity, a judicious use of dueling can provide a simple and alphabet-independent $O(n^2)$ algorithm for two dimensional exact matching.

Text processing is accomplished in two stages: Candidate Consistency and Candidate verification. A *candidate* is a location in the text where the pattern may occur. We denote a candidate starting at text location $T[r, c]$ by (r, c). We say that two candidates (r, c) and (x, y) are *consistent* if they expect the same text characters in their region of overlap (two candidates with no overlap are trivially consistent).

Initially, we have no information about the text and therefore all text locations are candidates. However, not all text locations are consistent. During the candidate consistency phase, we eliminate candidates until all remaining candidates are pairwise consistent. During the candidate verification phase, we check the candidates against the text to see which candidates represent actual occurrences of patterns. We exploit the consistency of the surviving candidates to rule out large sets of candidates with single text comparisons (since all consistent candidates expect the same text character).

Candidate Consistency: This is done with two sweeps of the text. The first eliminates inconsistent candidates within each column, and the second eliminates all inconsistent candidates in the text. The result of these two sweeps are potential sources, none of which can conflict with any other. This means that if we verify that one of these potential sources is indeed the source of a pattern occurrence then all potential sources *within* the verified area are *guaranteed* to overlap the verified area. Thus, verification need not ever backtrack. The details of the $O(n^2)$ candidate consistency algorithms can be found in [6].

Candidate Verification: As mentioned above, we are guaranteed that all consistent candidate sources overlap consistently with the pattern. We only need to verify that they are indeed pattern sources. This can be done in linear time by the time-tested sport cheer - *the wave*.

The idea of the wave is for each element to jump up and wave a pair $\langle i, j\rangle$ whose meaning is that this element has to be tested against $P[i, j]$. There may be several such options for some locations, but any will do because the candidate sources are now all compatible. The waved pair, in addition to knowledge of candidate sources, causes the element immediately below the waving location to wave its own pair. When all column waves are done we do row waves and every text element now needs a single comparison. For further details on the wave, see [6, 11].

3 Approximate Matching of Rectangular Arrays

One possible string matching generalization that has been researched is *approximate string matching* - finding all occurrences of a pattern in a text where *differences* are allowed.

Three types of *differences* were distinguished [43]:

1. A pattern character corresponds to a different character in the text (*mismatch*).
2. A text character is deleted (*deletion*).
3. A pattern character is deleted (*insertion*).

Two problems were considered in the one dimensional case: The *string matching with k mismatches problem (the k mismatches problem)* - find all occurrences of the pattern in the text with at most k type-1 differences. The *string matching with k differences problem (the k differences problem)* - find all occurrences of the pattern in the text with at most k differences of type 1. 2. or 3.

Abrahamson [1] used a divide-and-conquer approach in conjunction with the convolutions method to solve the string mismatches problem in time $O(n\sqrt{m}\log m)$. His algorithm writes, for every text location, the number of mismatches that will occur it the pattern is compared with the text starting at that location.

Landau and Vishkin [41] gave a $O(nk)$ algorithm for the k-differences problem.

In this section we consider approximate pattern matching where both text and pattern are rectangles.

INPUT: Text array $T[n \times n]$ and $P[m \times m]$ where all elements of P and T are in alphabet Σ, and integer k. *OUTPUT:* All occurrences of the pattern in the text with at most k differences.

The definition of insertion and deletion in multidimensions need clarification. The effect of insertion and deletion may be different depending on the implementation. We illustrate with a two dimensional example. If a matrix is transmitted serially a deleted character means an appropriate shift of the entire array. However, it may be the case that the array is transmitted column by column with an EOD indication between them. In that case, a deletion or insertion affects only the column it appears in. Following Krithivasan and Sitalakshmi [40] and Amir and Landau [13] we assume the latter situation. It is clear that the case where a deletion or insertion affects only the row it appears in can be handled in a similar manner.

Krithivasan and Sitalakshmi [40] solved this problem in time $O(n^2mk)$. This was improved by Amir and Landau to $O(n^2k^2)$ ($O(n^2k)$ if only mismatch errors are allowed).

The idea was using dynamic programming to handle the insertion and deletion problems, and suffix trees to identify runs of matching pattern and text substrings.

4 Approximate Matching of Non Rectangular Patterns

The approximate two dimensional problem we saw in section 3 was defined with both the pattern and text being rectangular matrices. In reality, it is usually necessary to match *non-rectangular* shapes. The techniques presented in section 3 seem inadequate in dealing with nonrectangular arrays.

4.1 Mismatches

In section 2.2 it was shown that multi-dimensional matching can be reduced to string matching by appropriate padding with don't care characters. Such a padding allows solving the exact two-dimensional matching problem, or the k-mismatches problem for *any shape* in time $O(|\Sigma| n^2 \log m)$. We simply pad the matrix appropriately so only the given shape is matched.

The $|\Sigma|$ factor in the complexity results from the fact that we need to do $|\Sigma|$ convolutions. In each one we count the number of pattern mismatches for a different alphabet symbol ([13]).

This method is clearly efficient for bounded small alphabets. For unbounded alphabets we may use the Abrahamson-Kosaraju divide-and-conquer technique to achieve time $O(n^2\sqrt{m}\log m)$.

4.2 Mismatches, Insertions and Deletions

Pattern matching provides many examples of powerful techniques that solves various different problems. However, when some criteria are combined, there is no ready solution. For example, convolutions solve the "don't care" problem, and the mismatches problem, but can not be used when insertions and deletions are introduced. Automata methods or suffix trees work mainly for exact matching. But if presented with the problem of matching with differences **and** don't cares then there is no known efficient method that can solve it.

Amir and Farach [12] made the first advance in the direction of efficiently solving the k-difference matching problem for non-rectangular patterns. A novel method was used, that combines the power of convolutions, dynamic programming and subword trees. It proved effective in solving the two-dimensional k-difference matching problem for *half-rectangular* patterns in time $O(kn^2\sqrt{m \log m}\sqrt{k \log k} + k^2n^2)$, where n^2 is the area of the text and m is the height of the pattern.

Definition: A *left half-rectangular* pattern is a list of variable-length rows, $P_1, ..., P_m$. The pattern is represented by stacking each row P_i above row P_{i+1} with $P_{i,1}$ directly above $P_{i+1,1}$.

Intuitively, the leftmost border of the pattern is a vertical line, and every horizontal cut of the figure is a single segment. One may similarly define a right, top or bottom half-rectangle depending on whether the right, top or bottom border is a straight edge.

This algorithm is efficient for any pattern that can be split into a "small" number of half-rectangular shapes. An example is any convex shape in an orientation where the longest diameter is vertical. We are searching for all locations where a half-rectangular pattern matches the text allowing no more than k mismatches, insertions (in rows) and deletions (in rows) errors.

To achieve this result some new tools were needed. Efficient solutions to two problems were provided. These problems are the *smaller matching problem* and the *k-aligned ones with location problem*.

The smaller matching problem is: *INPUT:* Text string $T = T_0, .., .T_{n-1}$ and pattern string $P = P_0, ..., P_{m-1}$ where $T_i, P_i \in N$. *OUTPUT:* All locations i in T where $T_{i+k-1} \geq P_k \; k = 1, ..., m$. In words, every matched element of the pattern is not greater than the corresponding text element.

The smaller matching problem with a forest partial order is defined similarly with the exception that the order relation is that induced by a given forest. Both these problems can be solved in time $O(n\sqrt{m}\log m)$.

The motivation for the k-aligned ones with locations problem stems from the use of convolutions in pattern matching. The power behind all known convolution-based string matching algorithms is multiplication of polynomials with binary coefficients $(0, 1)$. Polynomial multiplication can be done efficiently by using Fast Fourier Transform [3]. The result of such a polynomial multiplication is the *number* of 1's in the pattern that are aligned with 1's in the text at each position. However, all information about the *location* of these aligned 1's is lost. These locations were found in time $O(k^3 n \log m \log k)$ in [12].

Specifically, the k-aligned ones with location problem is: *INPUT:* Text string $T = T_0, ..., T_{n-1}$ and pattern string $P_0, ..., P_{m-1}$ where $T_i, P_i \in \{0, 1\}$. *OUTPUT:* All locations i in T where

$$\sum_{l=0}^{m} T_{l+i} P_l \leq k$$

and for each such i, all indices $i_1, ..., i_k$ where $P_{i_j} = T_{l+i_j} = 1$.

Recently, using superimposed codes, the k-aligned ones with locations problem has been solved $O(kn \log m \log k)$ [17].

5 Scaled Matching

All the problems we have seen so far were useful mainly in solving matching with "local errors" prolems. We mentioned that in reality we may be interested in matching patterns whose occurrence in the text is of different scale than provided by the pattern. For example, reading a newspaper one encounters letters of the alphabet in various sizes.

A "clean" version of the problem may be defined as follows [15]:

The string $aa...a$ where the symbol a is repeated k times (to be denoted a^k), is referred to as *a scaled to k*. Similarly, consider a string $A = a_1 \cdots a_l$. *A scaled to k* (A^k) is the string $a_1^k, ..., a_l^k$.

Let $P[m \times m]$ be a two-dimensional matrix over a finite alphabet Σ. Then P *scaled to* k (P^k) is the $km \times km$ matrix where every symbol $P[i,j]$ of P is replaced by a $k \times k$ matrix whose elements all equal the symbol in $P[i,j]$. More precisely,

$$P^k[i,j] = P[\lceil \frac{i}{k} \rceil, \lceil \frac{j}{k} \rceil].$$

The problem of *two-dimensional pattern matching with scaling* is defined as follows: *INPUT:* Pattern matrix $P[i,j] \quad i = 1, ...m; j = 1, ..., m$ and Text matrix $T[i,j] \quad i = 1, ..., n; j = 1, ..., n$ where $n > m$. *OUTPUT:* all locations in T where an occurrence of P scaled to k starts, for any $k = 1, ..., \lfloor \frac{n}{m} \rfloor$.

The basic algorithmic design strategy of Amir-Landau-Vishkin [15] can be viewed as realizing the following approach: For each scale k, try to select only a fraction of $\frac{1}{k}$ among the n columns and seek k-occurrences only in these columns. Since each selected column intersects n rows, this leads to consideration of $O(\frac{n^2}{k})$ elements. Summing over all scales, we get $O(n^2)$ multiplied by the harmonic sum $\sum_{i=1}^{\frac{n}{m}} \frac{1}{i}$, whose limit is $\log \frac{n}{m}$ making the total number of elements scanned $O(n^2 \log \frac{n}{m})$.

A final intuitive step was to select also a $\frac{1}{k}$ fraction of the rows. Since $\sum_{i=1}^{\frac{n}{m}} \frac{1}{i^2}$ is bounded by a constant, the number of elements decreases now to

$$O(n^2 \sum_{i=1}^{\frac{n}{m}} \frac{1}{i^2}) = O(n^2).$$

A simpler, alphabet-independent algorithm, that can be generalized to dictionary scaled matching was presented in [11].

A key technique in all discrete scaling algorithms is the *Range Minimum Problem.* Defined as follows:

Definition: Let $L = [l_1, ..., l_n]$ be an array of n numbers. A *Range Minimum* query is of the form:

> Given a range of indices $[i, ..., j]$, where $1 \le i \le j \le n$, return an index $k \quad i \le k \le j$ such that $l_k = \min\{l_i, ..., l_j\}$.

In [34] it was shown that a list of length n can be preprocessed in time $O(n)$ such that subsequent range minimum queries can be answered in constant time.

In scaled matching we are naturally interested in locations where there are many consecutive rows (columns) with the same elements. The range-minimum queries allow finding these areas in constant time. This can be achieved by preprocessing for every text location the longest common prefix of it, and the subrow immediately above it.

6 The Geometric Model

Everything we have seen so far suffers greatly from the encounter with "real-life problems". There is some justification for dealing with discrete scales in a combinatorial sense, since it is not clear what is a fraction of a pixel. However, in reality an object may appear in non-discrete scales. It is necessary to, both, define the combinatorial meaning of such scaling, and present efficient algorithms for the problem's solution. The rotation problem, presents similar challenges. What is the discrete meaning of a rotated pattern? The answer to both above problems involves a *Geometric Model* for two-dimensional matching.

Until now, we considered the text and pattern to be matrices of alphabet symbols. The new idea, first proposed by Landau and Vishkin [42] is to consider the text and pattern as large rectangles composed of *unit squares*. These unit squares are "colored" by a picture of reality. For the sake of scaling and rotation, we consider the color of the center of a unit square as the color of the square. We will define the meaning of this geometric model for scaling and rotation in more detail in sections 6.1 and 6.2. However, for historical reasons we mention that Landau and Vishkin's motivation for defining the geometric model was the digitization process. For all intents the granularity of the world is so fine as to be considered continuous. Nevertheless, when a photo is taken, the image is projected onto a pixel map with much coarser granularity. Landau and Vishkin viewed the process as sampling unit squares and assigning an image pixel the color of its sampled center. This idea is used for the geometric definition of rotation and scaling to sizes that are not natural numbers.

6.1 Scaling

Amir, Butman, Lewenstein and Porat [10] present a definition for *scaled pattern matching* with arbitrary real scales. The definition is pleasing in a "real-world" sense. Below see "lenna" scaled to non-discrete scales by this definition and the results look natural (see Figure 1). This definition was inspired by the idea of digitizing analog signals by sampling, however, it does not assume an underlying continuous function thus stays on the combinatorial pattern matching field. This seems to be the natural way to define combinatorially the meaning of scaling in the signal processing sense.

Figure 1: An original image, scaled by 1.3 and scaled by 2, using the combinatorial definition of scaling.

This definition, that had been sought by researchers in pattern matching since at least 1990, captures *scaling* as it occurs in images, yet has the necessary combinatorial features that allows developing deterministic algorithms and analysing their worst-case complexity. Indeed Amir, Butman, Lewenstein and Porat [10] present a two dimensional efficient algorithm for real scaled pattern matching.

The definition of two-dimensional scaled matching is an extension of the one dimensional definition.

Definition Let T be a two-dimensional $n \times n$ array over some finite alphabet Σ.

1. The *unit pixels array* for T (T^{1X}) consists of n^2 unit squares, called *pixels* in the real plane R^2. The corners of the pixel $T[i,j]$ are $(i-1, j-1), (i, j-1), (i-1, j)$, and (i, j). Hence the pixels of T form a regular $n \times n$ array covering the area between $(0,0), (n,0), (0,n)$, and (n,n). Point $(0,0)$ is the *origin* of the unit pixel array. The *center* of each pixel is the geometric center point of its square location. Each pixel $T[i,j]$ is identified with the value from Σ that the original array T had in that position. We say that the pixel has a *color* from Σ. See figure 2 for an example of the grid and pixel centers of a 7×7 array.

2. Let $r \in \Re$, $r > 1$. The *r-ary pixels array* for T (T^{rX}) consists of n^2 *r-squares*, each of dimension $r \times r$ whose *origin* is $(0,0)$ and covering the area between $(0,0), (nr,0), (0,nr)$, and (nr,nr). The corners of the pixel $T[i,j]$ are $((i-1)r, (j-1)r), (ir, (j-1)r), ((i-1)r, jr)$, and (ir, jr). The *center* of each pixel is the geometric center point of its square location.

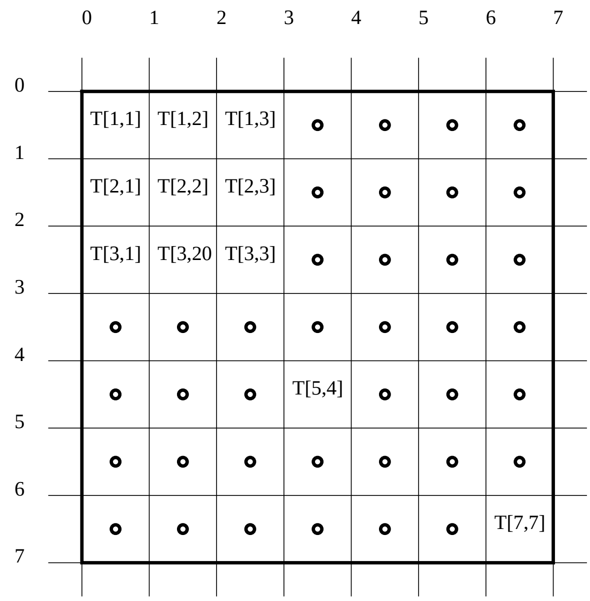


Figure 2: The grid and pixel centers of a unit pixel array for a 7×7 array.

Notation: Let $r \in \Re$. $\|r\|$ denotes the *rounding* of r, i.e.

$$\|r\| = \begin{cases} \lfloor r \rfloor & \text{if the fraction part of } r \text{ is less than .5;} \\ \lceil r \rceil & \text{otherwise.} \end{cases}$$

In this definition we round 0.5 up. There may be cases where we need to round 0.5 down. For this we denote:

$$||\lfloor r \rfloor|| = \begin{cases} \lfloor r \rfloor & \text{if the fraction part of } r \text{ is not more than .5;} \\ \lceil r \rceil & \text{otherwise.} \end{cases}$$

Definition Let T be an $n \times n$ text array and P be an $m \times m$ pattern array over alphabet Σ. Let $r \in \Re$, $1 \leq r \leq \frac{n}{m}$.

We say that there is an *occurrence of* P *scaled to* r at text location $[i, j]$ if the following condition hold:

Let T^{1X} be the unit pixels array of T and P^{rX} be the r-ary pixel arrays of P. Translate P^{rX} onto T^{1X} in a manner that the origin of P^{rX} coincides with location $(i-1, j-1)$ of T^{1X}. Every center of a pixel in T^{1X} which is within the area covered by $(i-1, j-1), (i-1, j-1+mr), (i-1+mr, j-1)$ and $(i-1+mr, j-1+mr)$ has the same color as the r-square of P^{rX} in which it falls.

The colors of the centers of the pixels in T^{1X} which are within the area covered by $(i-1, j-1), (i-1, j-1+mr), (i-1+mr, j-1)$ and $(i-1+mr, j-1+mr)$ define a $\|mr\| \times \|mr\|$ array over Σ. This array is denoted by P^r and called P *scaled to* r.

It is possible to find all scaled occurrences of an $m \times m$ pattern in an $n \times n$ text in time $O(n^2m^2)$. Such an algorithm, while not trivial, is nonetheless achievable with known techniques. In [10] an $O(nm^3 + n^2m \log m)$ algorithm was presented. Suitable trade-offs lead to an algorithm whose running time is $O(n^{1.5}m^2\sqrt{\log m})$.

The efficiency of the algorithm results from the properties of scaling. The scaling definition needs to accommodate a conflict between two notions, the continuous (represented by the real-number scale), and the discrete (represented by the array representation of the images). Understanding, and properly using, the shift from the continuous to the discrete and back are key to the efficiency of the algorithms.

6.2 Rotation

The pattern matching with rotation problem is that of finding all occurrences of a two dimensional pattern in a text, in all possible rotations. An efficient solution to the problem proved elusive even though many researchers were thinking about it for over a decade. Part of the problem was lack of a rigorous definition to capture the concept of rotation in a discrete pattern.

The major breakthrough came when Fredriksson and Ukkonen [31] resorted to a geometric interpretation of text and pattern and provided the following definition.

Let P be a two-dimensional $m \times m$ array and T be a two-dimensional $n \times n$ array over some finite alphabet Σ. As in the previous section, the array of *unit pixels* for T consists of n^2 unit squares, called *pixels* in the real plane R^2. The corners of the pixel for $T[i, j]$ are $(i-1, j-1)$, $(i, j-1)$, $(i-1, j)$, and (i, j). Hence the pixels for T form a regular $n \times n$ array covering the area between $(0, 0)$, $(n, 0)$, $(0, n)$, and (n, n). The *center* of each pixel is the geometric center point of the pixel. Each pixel $T[i, j]$ is identified with the value from Σ that the original text had in that position. We say that the pixel has a *color* from Σ.

The array of pixels for pattern P is defined similarly. A different treatment is necessary for patterns with odd sizes and for patterns with even sizes. For simplicity's sake we assume throughout the rest of this paper that the pattern is of size $m \times m$ and m is even. The *rotation pivot* of the pattern is its exact center, the point $(\frac{m}{2}, \frac{m}{2}) \in R^2$. See Figure 3 for an example of the rotation pivot of a 4×4 pattern P.

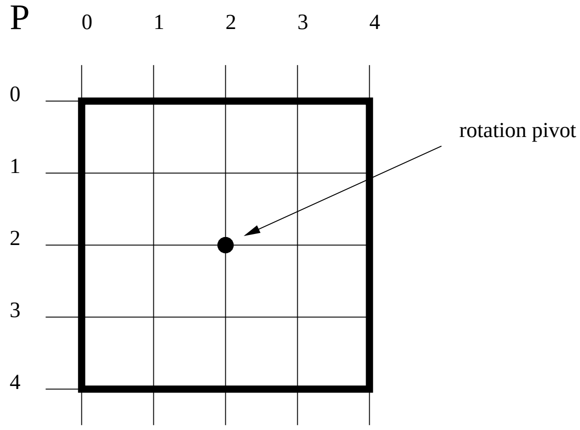


Figure 3: The rotation pivot of a 4×4 pattern P.

Consider now a rigid motion (translation and rotation) that moves P on top of T. Consider the special case where the translation moves the grid of P precisely on top of the grid of T, such that the grid lines coincide.

Assume that the rotation pivot of P is at location (i, j) on the text grid, and that the pattern lies *under* the text. The pattern is now rotated, centered at (i, j), creating an angle α between the x-axes of T and P. P is said to be at *location* $((i, j), \alpha)$ *under* T. Pattern P is said to have an *occurrence* at location $((i, j), \alpha)$ if the *center* of each pixel in T has the same color as the pixel of P under it, if there is such a pixel. When the center of a text pixel is exactly over a vertical (horizontal) border between text pixels, the color of the pattern pixel left (below) to the border is chosen. Consider some occurrence of P at location $((i_0, j_0), \alpha)$. This occurrence defines a non-rectangular substring of T that consists of all the pixel of T whose centers are inside pixels of P. We call this string P *rotated by* α, and denote it by P^α. Note that there is an occurrence of P at location $((i, j), \alpha)$ if and only if P^α occurs at (i, j).

Fredriksson, Navarro and Ukkonen [29] give two possible definitions for rotation. One is as described above and the second is, in some way, the opposite. P is placed *over* the text T. More precisely, assume that the rotation pivot of P is on top of location (i, j) on the text grid. The pattern is now rotated, centered at (i, j), creating an angle α between the x-axes of T and P. P is said to be at *location* $((i, j), \alpha)$ *over* T. Pattern P is said to have an *occurrence* at location $((i, j), \alpha)$ if the center of each pixel in P has the same color as the pixel of T under it.

While the two definitions of rotation, "over" and "under", seem to be quite similar, they are not identical. For example, in the "pattern over text" model there exist angles for which two pattern pixel centers may find themselves in the same text pixel. Alternately, there are angles where there are "holes" in the rotated pattern, namely there is a text pixel that does not have in it a center of a pattern pixel, but all text pixels around it have centers of pattern pixels. See Figure 4 for an example.

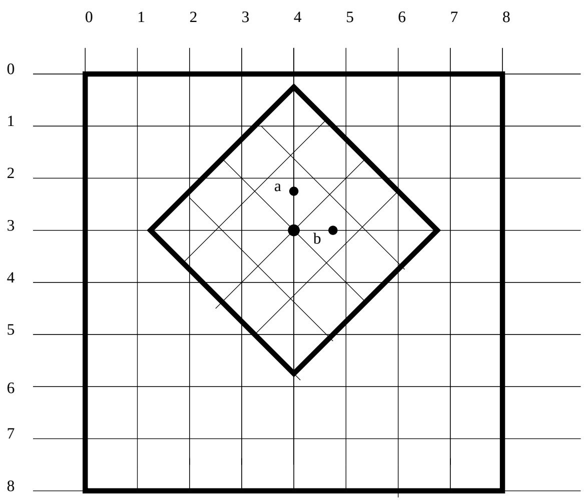


Figure 4: An example of a "hole" in the pattern. Text pixel $T[3,5]$ has no pattern pixel over it, but the pixels $T[2,5]$ and $T[3,6]$ have pattern pixel centers.

The challenges of "discretizing" a continuous image are not simple. In the Image Processing field, stochastic and probabilistic tools need to be used because the images are "smoothed" to compensate for the fact that the image is presented in a far coarser granularity than in reality. The aim of the the pattern matching community has been to fully discretize the process, thus our different definitions. However, this puts us in a situation where some "gut" decisions need to be made regarding the model that best represents "reality". It is our feeling that in this context the "pattern under text" model is more intuitive since it does not allow anomalies such as having two pattern centers over the same text pixel (a contradiction) nor does it create "holes" in the rotated pattern For examples of the rotated patterns in the two models see Figure 5.

Most of the algorithms for rotated matching are filtering algorithms that behave well on average but that have a bad worst case complexity (e.g. [32, 29, 33]). In three papers ([30, 9, 28]), there is a $O(n^2m^3)$ worst case algorithm for rotated matching. All worst-case algorithms basically work by enumerating all possible rotated patterns and solving a two dimensional dictionary matching problem on the text. In [9] it was proven that there are $\Theta(m^3)$ such rotated patterns. The high complexity results from the fact that the dictionary patterns have "don't care" symbols in them and thus, essentially, every pattern needs to be sought separately.

In [16], Amir, Kapah and Tsur present the first rotated matching algorithms whose time is better than $O(n^2m^3)$. The scanning time of their algorithms is $O(n^2m^2)$. These results are achieved by identifying monotonicity properties on the rotated patterns. These properties allow using *transitivity*-based dictionary matching algorithms, cutting the worst-case time by an m factor.

7 Conclusions and Open Problems

We have scanned some of the problems and techniques in two dimensional matching. Clearly, we are still far from our motivation of actually finding a given template in an

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

(a)

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

(b)

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

(c)

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

(d)

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

(e)

	1	2	
9	6	7	4
13	10	11	8
	15	16	

(f)

Figure 5: An example of some possible 2-dimensional arrays that represent one pattern. Fig (a) — the original pattern. Figures (b)–(d) are computed in the "pattern over the text" model. Fig (b) — a representation of the pattern rotated by 19^0. Fig (c) — Pattern rotated by 21^0. Fig (d) — Pattern rotated by 26^0. Figures (e)–(f) are computed in the "pattern under the text" model. Fig (e) — Pattern rotated by 17^0. Fig (f) — Pattern rotated by 26^0.

image. What we have are various techniques for solving different subproblems, but we need one method to solve them all. We need a scaled-rotated-approximate-dictionary matching of nonrectangular patterns and that seems a great challenge indeed.

There are many technical open problems that were left in the wake of the results described in this survey, such as a real-time suffix tree construction algorithm, approximate indexing and dictionary matching algorihtms, and even more efficient algorithms for rotations. Other problems are new methods for general convolutions, multidimensional extensions that are dimension-independent, dictionary matching with "don't cares". But the grand inspiration continues to be integration of these solutions to a general matching algorithm.

We may never reach that goal, but the way sure is exciting...

References

[1] K. Abrahamson. Generalized string matching. *SIAM J. Comp.*, 16(6):1039–1051, 1987.

[2] A.V. Aho and M.J. Corasick. Efficient string matching. *Comm. ACM*, 18(6):333–340, 1975.

[3] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[4] A. Amir and G. Benson. Two-dimensional periodicity and its application. *Proc. of 3rd Symposium on Discrete Algorithms, Orlando, FL*, pages 440–452, Jan 1992.

[5] A. Amir, G. Benson, and M. Farach. The truth, the whole truth, and nothing but the truth: Alphabet independent two dimensional witness table construction. Technical Report GIT-CC-92/52, Georgia Institute of Technology, August 1992.

[6] A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two dimensional pattern matching. *SIAM J. Comp.*, 23(2):313–323, 1994.

[7] A. Amir, G. Benson, and M. Farach. Optimal two-dimensional compressed matching. *Proc. ICALP 94*, pages 215–226, 1994.

[8] A. Amir, G. Benson, and M. Farach. Optimal parallel two dimensional text searching on a crew pram. *Information and Computation*, 144(1):1–17, July 1998.

[9] A. Amir, A. Butman, M. Crocehmore, G.M. Landau, and M. Schaps. Two-dimensional pattern matching with rotations. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, number 2676 in LNCS, pages 17–31. Springer, 2003.

[10] A. Amir, A. Butman, M. Lewenstein, and E. Porat. Real two dimensional scaled matching. In *Proc. 8th Workshop on Algorithms and Data Structures (WADS)*, pages 353–364, 2003.

[11] A. Amir and G. Calinescu. Alphabet independent and dictionary scaled matching. *J. of Algorithms*, 36:34–62, 2000.

[12] A. Amir and M. Farach. Efficient 2-dimensional approximate matching of non-rectangular figures. *Proc. of 2nd Symposium on Discrete Algorithms, San Francisco, CA*, pages 212–223, Jan 1991.

[13] A. Amir and G. Landau. Fast parallel and serial multidimensional approximate array matching. *Theoretical Computer Science*, 81:97–115, 1991.

[14] A. Amir, G. Landau, and D. Sokol. Inplace run-length 2d compressed search. *Theoretical Computer Science*, 290(3):1361–1383, 2003.

[15] A. Amir, G.M. Landau, and U. Vishkin. Efficient pattern matching with scaling. *Proceedings of First Symposium on Discrete Algorithms, San Fransisco, CA*, pages 344–357, 1990.

[16] A. Amir, D. Tsur, and O. Kapah. Faster two dimensional pattern matching with rotations. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2004. to appear.

[17] Y. Aumann, M. Lewenstein, N. Lewenstein, and D. Tsur. Pealing codes with applications to finding witnesses. submitted for publication, 2004.

[18] T.J. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comp.*, 7:533–541, 1978.

[19] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly parallelizable problems. *Proc. 21st ACM Symposium on Theory of Computation*, pages 309–319, 1989.

[20] R.S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, 1977.

[21] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

[22] V. Chvatal, D.A. Klarner, and D.E. Knuth. Selected combinatorial research problems. Technical Report STAN-CS-72-292, Stanford University, 1972.

[23] R. Cole, M. Crochemore, Z. Galil, L. Gąsieniec, R. Harihan, S. Muthukrishnan, K. Park, and W. Rytter. Optimally fast parallel algorithms for preprocessing and pattern matching in one and two dimensions. *Proc. 34th IEEE FOCS*, pages 248–258, 1993.

[24] R. Cole and R. Hariharan. Dynamic lca queries in trees. In *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 235–244, 1999.

[25] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1992.

[26] M. Farach. Optimal suffix tree construction with large alphabets. *Proc. 38th IEEE Symposium on Foundations of Computer Science*, pages 137–143, 1997.

[27] M.J. Fischer and M.S. Paterson. String matching and other products. *Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings*, 7:113–125, 1974.

[28] K. Fredriksson, V. Mäkinen, and G. Navarro. Rotation and lighting invariant template matching. In *Proceedings of the 6th Latin American Symposium on Theoretical Informatics (LATIN'04)*, LNCS, 2004. To appear. Available at `http://www.dcc.uchile.cl/ gnavarro/ps/latin04.ps.gzx`.

[29] K. Fredriksson, G. Navarro, and E. Ukkonen. An index for two dimensional string matching allowing rotations. In *Prof. IFIP International Conference on Theoretical Computer Science (IFIP TCS)*, volume 1872 of *LNCS*, pages 59–75. Springer, 2000.

[30] K. Fredriksson, G. Navarro, and E. Ukkonen. Optimal exact and fast approximate two dimensional pattern matching allowing rotations. In *Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 2373 of *LNCS*, pages 235–248. Springer, 2002.

[31] K. Fredriksson and E. Ukkonen. A rotation invariant filter for two-dimensional string matching. In *Proc. 9th Annual Symposium on Combinatorial Pattern Matching (CPM 98)*, pages 118–125. Springer, LNCS 1448, 1998.

[32] K. Fredriksson and E. Ukkonen. A rotation invariant filter for two-dimensional string matching. In *Proc. 9th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 1448 of *LNCS*, pages 118–125. Springer, 1998.

[33] K. Fredriksson and E. Ukkonen. Combinatorial methods for approximate pattern matching under rotations and translations in 3d arrays. In *Proc. 7th Symposium on String Processing and Information Retrieval (SPIRE'2000)*, pages 96–104. IEEE CS Press, 2000.

[34] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. *Proc. 16th ACM Symposium on Theory of Computing*, 67:135–143, 1984.

[35] Z. Galil. Open problems in stringology. In Z. Galil A. Apostolico, editor, *Combinatorial Algorithms on Words*, volume 12, pages 1–8. NATO ASI Series F, 1985.

[36] Z. Galil and K. Park. Alphabet-independent two-dimensional witness computation. *SIAM J. Comp.*, 25(5):907–935, October 1996.

[37] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestor. *Computer and System Science*, 13:338–355, 1984.

[38] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 03)*, pages 943–955, 2003. LNCS 2719.

[39] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comp.*, 6:323–350, 1977.

[40] K. Krithivansan and R. Sitalakshmi. Efficient two dimensional pattern matching in the presence of errors. *Information Sciences*, 13:169–184, 1987.

[41] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

[42] G. M. Landau and U. Vishkin. Pattern matching in a digitized image. *Algorithmica*, 12(3/4):375–408, 1994.

[43] V. I. Levenshtein. Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966.

[44] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23:262–272, 1976.

[45] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comp.*, 17:1253–1262, 1988.

[46] E. Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5:313–323, 1990.

[47] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[48] U. Vishkin. Optimal parallel pattern matching in strings. *Proc.* 12*th ICALP*, pages 91–113, 1985.

[49] P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

Algorithms for the Constrained Longest Common Subsequence Problems

Abdullah N. Arslan[1] and Ömer Eğecioğlu[2*]

[1] Department of Computer Science
University of Vermont
Burlington, VT 05405, USA
e-mail: aarslan@cs.uvm.edu

[2] Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106, USA
e-mail: omer@cs.ucsb.edu

Abstract. Given strings S_1, S_2, and P, the constrained longest common subsequence problem for S_1 and S_2 with respect to P is to find a longest common subsequence *lcs* of S_1 and S_2 such that P is a subsequence of this *lcs*. We present an algorithm which improves the time complexity of the problem from the previously known $O(rn^2m^2)$ to $O(rnm)$ where r, n, and m are the lengths of P, S_1, and S_2, respectively. As a generalization of this, we extend the definition of the problem so that the *lcs* sought contains a subsequence whose edit distance from P is less than a given parameter d. For the latter problem, we propose an algorithm whose time complexity is $O(drnm)$.

Keywords: Longest common subsequence, constrained subsequence, edit distance, dynamic programming.

1 Introduction

A subsequence of a string S is obtained by deleting zero or more symbols of S. The *longest common subsequence* (*lcs*) problem for two strings is to find a common subsequence in both strings having maximum possible length. The *lcs* problem has many applications, and it has been studied extensively, see for example [1, 4, 2, 3, 5, 7]. The problem has a simple dynamic programming formulation. To compute an *lcs* between two strings of lengths n, and m, we use the *edit graph*. The edit graph is a directed acyclic graph having $(n+1)(m+1)$ lattice points (i, j) for $0 \leq i \leq n$, and $0 \leq j \leq m$ as vertices. Vertex $(0, 0)$ appears at the top-left corner, and the vertex (n, m) is at the bottom-right corner of this rectangular grid. To vertex (i, j) there are incoming arcs from its neighbors at $(i-1, j), (i, j-1)$, and $(i-1, j-1)$ which represent, respectively, insert, delete, and either substitute or match operations. The *lcs* calculation counts the number of matches on the paths from vertex $(0, 0)$ to (n, m), and the problem aims to maximize this number. The time complexity lower bound

*Work done in part while on sabbatical at Sabanci University, Istanbul, Turkey during 2003-2004.

for the problem is $\Omega(n^2)$ for $n \geq m$ if the elementary operations are "equal/unequal", and the alphabet size is unrestricted [1]. If the alphabet is fixed the best known time complexity is $O(n^2/\log n)$ when $n = m$ [5]. A survey of practical *lcs* algorithms can be found in [2].

Given strings S_1, S_2, and P, the constrained longest common subsequence problem [6] for S_1 and S_2 with respect to P is to find a longest common subsequence *lcs* of S_1 and S_2 such that P is a subsequence of this *lcs*. For example, for $S_1 = \texttt{bbaba}$, and $S_2 = \texttt{abbaa}$, $\texttt{bbaa}$ is an (unrestricted) *lcs* for S_1 and S_2, and $\texttt{aba}$ is an *lcs* for S_1 and S_2 with respect to $P = \texttt{ab}$, as shown in Figure 1.

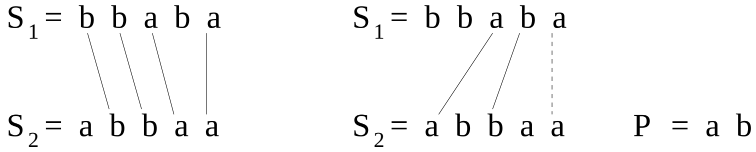


Figure 1: For $S_1 = \texttt{bbaba}$, and $S_2 = \texttt{abbaa}$, the length of an *lcs* is 4 (left). When constrained to contain $P = \texttt{ab}$ as a subsequence, the length of an *lcs* drops to 3 (right).

The problem is motivated by practical applications: For example in the computation of the homology of two biological sequences it is important to take into account a common specific or putative structure [6].

Let n, m, r denote the lengths of the strings S_1, S_2, and P, respectively. Tsai [6] gave a dynamic programming formulation for the constrained longest common subsequence problem and a resulting algorithm whose time complexity is $O(rn^2m^2)$. In this paper we present a different dynamic programming formulation with which we improve the time complexity of the problem down to $O(rnm)$. We achieve improved results by changing the order of the dimensions in the formulation. We also extend the definition of the problem so that the *lcs* sought is forced to contain a subsequence whose *edit distance* from P is less than a given positive integer parameter d. For this latter problem we propose an algorithm whose time complexity is $O(drnm)$. Taking $d = 1$ specializes to the original constrained *lcs* problem as this choice of d forces the subsequence to contain P itself. We describe these results in section 2.

2 Algorithms

Let $|S_1| = n$, $|S_2| = m$ with $n \geq m$, and $|P| = r$. Let $S[i]$ denote the ith symbol of string S. Let $S[i..j] = S[i]S[i+1]\cdots S[j]$ be the substring of consecutive letters in S from position i to position j inclusive for $i \leq j$, and the empty string otherwise.

Denote by $L_{i,j,k}$ the length of an *lcs* for $S_1[1..i]$ and $S_2[1..j]$ with respect to $P[1..k]$. This simply means that the common subsequence is constrained to contain P as a subsequence in turn. We calculate the values $L_{i,j,k}$ by a dynamic programming formulation. Then $L_{n,m,r}$ is the length of an *lcs* of S_1 and S_2 containing P as a subsequence.

Theorem 1 *For all i, j, k, $1 \leq i \leq n$, $1 \leq j \leq m$, $0 \leq k \leq r$, $L_{i,j,k}$ satisfies*

$$L_{i,j,k} = \max\{L'_{i,j,k},\ L_{i,j-1,k},\ L_{i-1,j,k}\} \tag{1}$$

where

$$L'_{i,j,k} = \max\{L''_{i,j,k},\ L'''_{i,j,k}\} \tag{2}$$

and

$$L''_{i,j,k} = \begin{cases} 1 + L_{i-1,j-1,k-1} & \textit{if } (k = 1 \textit{ or } (k > 1 \textit{ and } L_{i-1,j-1,k-1} > 0)) \\ & \quad \textit{and } S_1[i] = S_2[j] = P[k] \\ 0 & \textit{otherwise} \end{cases}$$

$$L'''_{i,j,k} = \begin{cases} 1 + L_{i-1,j-1,k} & \textit{if } (k = 0 \textit{ or } L_{i-1,j-1,k} > 0) \textit{ and } S_1[i] = S_2[j] \\ 0 & \textit{otherwise} \end{cases}$$

with boundary conditions $L_{i,0,k} = 0$*,* $L_{0,j,k} = 0$*, for all* i, j, k*,* $0 \le i \le n$*,* $0 \le j \le m$*,* $0 \le k \le r$*.*

Proof We prove the correctness of our formulation by induction on k for all i, j.

We will consider all possible ways of obtaining an *lcs* with respect to $P[1..k]$ at any node i, j. Essentially there are three cases to consider:

1. An *lcs* ending at the node $(i, j-1)$ is extended with the horizontal arc $((i, j-1), (i, j))$ ending at node (i, j),

2. An *lcs* ending at $(i-1, j)$ is extended with the vertical arc $((i-1, j), (i, j))$ ending at node (i, j),

3. An *lcs* ending at node $(i-1, j-1)$ is extended with the diagonal arc $((i-1, j-1), (i, j))$ ending at node (i, j). In this case we distinguish between subcases depending on whether the diagonal arc is a matching for the given strings along with the pattern, or is a matching for the given strings only at the current indices.

The possible *lcs* extensions referred to in items 1 and 2 above are accounted for by $L_{i,j-1,k}$ and $L_{i-1,j,k}$ respectively in the statement of the theorem. The quantities $L''_{i,j,k}$ and $L'''_{i,j,k}$ in the statement of the theorem keep track of the two further possibilities described in item 3.

In the base case: when $k = 0$ (i.e. when P is the empty string) $L''_{i,j,k}$ is identically 0. Therefore $L'_{i,j,k} = L'''_{i,j,k}$ in (2). Since $k = 0$, the conjunction in the definition of $L'''_{i,j,k}$ is always satisfied. We see that putting $L_{i,j} = L_{i,j,0}$, (1) becomes

$$L_{i,j} = \max\{L'_{i,j},\ L_{i,j-1},\ L_{i-1,j}\}$$

where

$$L'_{i,j} = \begin{cases} 1 + L_{i-1,j-1} & \text{if } S_1[i] = S_2[j] \\ 0 & \textit{otherwise} \end{cases}$$

which is the classical dynamic programming formulation for the ordinary *lcs* between S_1 and S_2 [7].

Assume that for $k-1$ $(k \ge 1)$, $L_{i,j,k-1}$ computed by (1) is the length of an *lcs* for $S_1[1..i]$ and $S_2[1..j]$ with respect to $P[1..k-1]$ for all i, j and consider the calculation of $L_{i,j,k}$ when $k > 1$.

We define a *path* at node (i, j) as a simple path in the edit graph which includes at least one matching arc, starts at node $(0, 0)$, and ends at node (i, j). A path with respect to $P[1..k]$ includes matching diagonal arcs ending at a sequence of $k \geq 1$ distinct nodes $(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)$ such that for all ℓ, $1 \leq \ell \leq k$, $S_1[a_\ell] = S_2[b_\ell] = P[\ell]$. We define *#match* on a path as the number of matches between the symbols of S_1, and S_2, not necessarily involving symbols in P. An *lcs path* with respect to $P[1..k]$ ending at node (i, j) is a path with respect to $P[1..k]$ ending at node (i, j) with maximum *#match*. Thus $L_{i,j,k}$ is *#match* on an *lcs* path at node (i, j) with respect to $P[1..k]$. Evidently *#match* = *#match*(i, j, k) is a function of the indices i, j, k. We will omit these parameters when they are clear from the context.

We can extend any *lcs* path with respect to $P[1..k]$ ending at node $(i, j-1)$ with the horizontal arc $((i, j-1), (i, j))$ to obtain a path with respect to $P[1..k]$ ending at node (i, j). Such an extension does not change *#match* on the path, and $L_{i,j,k} \geq L_{i,j-1,k}$.

Similarly we can extend any *lcs* path with respect to $P[1..k]$ ending at node $(i-1, j)$ with the vertical arc $((i-1, j), (i, j))$ to obtain a path with respect to $P[1..k]$ ending at node (i, j). This extension does not change *#match* on the path either, and $L_{i,j,k} \geq L_{i-1,j,k}$. Therefore, $L_{i,j,k} \geq \max\{L_{i,j-1,k},\ L_{i-1,j,k}\}$.

By using a matching arc $((i-1, j-1), (i, j))$, we can obtain paths with respect to $P[1..k]$ at node (i, j) by extending *lcs* paths with either respect to $P[1..k-1]$, or with respect to $P[1..k]$ ending at node $(i-1, j-1)$. These two possibilities are accounted for by $L''_{i,j,k}$ and $L'''_{i,j,k}$ in the dynamic programming formulation, respectively.

First consider *lcs* paths with respect to $P[1..k-1]$ ending at node $(i-1, j-1)$. We will show that $L''_{i,j,k}$ stores the maximum *#match* on paths obtained at node (i, j) by extending these paths.

If $S_1[i] = S_2[j] = P[k]$ then: If $k = 1$ then this is the first time the letter $P[1]$ appears as a matching arc on a path ending at node (i, j) since we are considering *lcs* paths with respect to $P[1..k-1]$ ending at node $(i-1, j-1)$ and $S_1[i] = S_2[j] = P[1]$. Therefore, the *lcs* length relative to $P[1]$ at (i, j) is $L''_{i,j,1} = 1 + L_{i-1,j-1,0}$, which is one more than the length of an ordinary *lcs* between $S_1[1..i-1]$ and $S_2[1..j-1]$. If $k > 1$ and if there is an *lcs* path with respect to $P[1..k-1]$ ending at node $(i-1, j-1)$ (i.e. if $L_{i-1,j-1,k-1} > 0$) then we can extend this path with a new match, and *#match* in the resulting path ending at node (i, j) becomes $L''_{i,j,k} = 1 + L_{i-1,j-1,k-1}$.

Next we consider *lcs* paths with respect to $P[1..k]$ ending at node $(i-1, j-1)$. We will show that $L'''_{i,j,k}$ stores the maximum *#match* on paths obtained at node (i, j) by extending these paths.

If $S_1[i] = S_2[j]$ then: Since the $k = 0$ case is considered earlier in the base case of the induction, we only consider the case when $k > 1$. If there is an *lcs* path with respect to $P[1..k]$ ending at node $(i-1, j-1)$ (i.e. if $L_{i-1,j-1,k} > 0$) then we can extend this path by adding a new match (which does not involve P), and *#match* in the resulting path relative to $P[1..k]$ ending at node (i, j) becomes $L'''_{i,j,k} = 1 + L_{i-1,j-1,k}$.

After setting $L'_{i,j,k} = \max\{L''_{i,j,k},\ L'''_{i,j,k}\}$, the quantity $L'_{i,j,k}$ is equal to the maximum *#match* on paths with respect to $P[1..k]$ ending at node (i, j) ending with the arc $((i-1, j-1), (i, j))$. If there is no such path then $L'_{i,j,k} = 0$. Therefore $L_{i,j,k} \geq \max\{L'_{i,j,k},\ L_{i,j-1,k},\ L_{i-1,j,k}\}$.

From all possible *lcs* paths ending at neighboring nodes of (i, j) we can find their extensions ending at node (i, j), and we can obtain an *lcs* path ending at node (i, j) with respect to $P[1..k]$ for all k. We calculate, and store in $L_{i,j,k}$ such *lcs* lengths. Now consider the structure of an *lcs* path with respect to $P[1..k]$ ending at node (i, j). As

	b	b	a	b	a
a	0	0	1	1	1
b	1	1	1	2	2
b	1	2	2	2	2
a	1	2	3	3	3
a	1	2	3	3	4

$k = 0$

	b	b	a	b	a
a	0	0	1	1	1
b	0	0	1	2	2
b	0	0	1	2	2
a	0	0	3	3	3
a	0	0	3	3	4

$k = 1$

	b	b	a	b	a
a	0	0	0	0	0
b	0	0	0	2	2
b	0	0	0	2	2
a	0	0	0	2	3
a	0	0	0	2	3

$k = 2$

Figure 2: For $S_1 = \texttt{abbaa}, S_2 = \texttt{bbaba}$, and $P = \texttt{ab}$, the tables of values $L_{i,j,k}$ = the length of an *lcs* for $S_1[1..i]$ and $S_2[1..j]$ with respect to $P[1..k]$.

typical in dynamic programming formulations, we consider the possible cases of the last arc on such a path to obtain $L_{i,j,k} \leq \max\{L'_{i,j,k},\ L_{i,j-1,k},\ L_{i-1,j,k}\}$ which proves the theorem. •

Example: Figure 2 shows the contents of the dynamic programming tables for $S_1 = \texttt{bbaba}$, and $S_2 = \texttt{abbaa}$, and $P = \texttt{ab}$ for $k = 0, 1, 2$. For $k = 0$, the calculated values are simply the ordinary dynamic programming *lcs* table for S_1 and S_2.

All $L_{i,j,k}$ can be computed in $O(rnm)$ time, using $O(rm)$ space using the formulation in Theorem 1 by noting that we only need rows $i-1$, and i during the calculations at row i. If actual *lcs* is desired then we can carry the *lcs* information for each k along with the calculations. This requires $O(rnm)$ space. By keeping track, on *lcs* for each k, of only the match points (i', j') of $P[u]$ for all u, $1 \leq u \leq r$, the space complexity can be reduced to $O(r^2m)$. In this case, the *lcs* for $k = r$ needs to be recovered using ordinary *lcs* computations to connect the consecutive match points.
Remark: Space complexity can further be improved by applying a technique used in unconstrained *lcs* computation [3]. We can compute, instead of the entire *lcs* for each k, middle vertex $(n/2, j)$ (assume for simplicity that n is even) at which an *lcs* with respect to $P[1..k]$ passes. This can be done in $O(rm)$ space, and we can compute for all k the *lcs* length $L_{n/2,j,k}$ from vertex $(0, 0)$ to vertex $(n/2, j)$, and *lcs* length from $(n/2, j)$ to (n, m). The latter is done in the reverse edit graph by calculating *lcs* from (n, m) to $(n/2, j)$, hence we denote it by $L^{reverse}_{n/2,j,l}$ for $0 \leq \ell \leq k$. Then for every k,

$$\max_{j, 0 \leq \ell \leq k} L_{n/2,j,l} + L^{reverse}_{n/2,j,k-l}$$

is the *lcs* length for k, and it identifies a middle vertex. After the middle vertex $(n/2, j)$ on *lcs* for every k is found, the problem of finding the *lcs* from $(0, 0)$ to (n, m) can be solved in two parts: find the *lcs* from $(0, 0)$ to $(n/2, j)$, and find the *lcs* from $(n/2, j)$ to (n, m) for all k. These two subproblems can be solved recursively by finding the middle points. This way *lcs* can be obtained using $O(rm)$ space. The time complexity remains $O(rnm)$ because n is halved each time, and the area (in terms of number of vertices) covered in the edit graph is $O(nm)$, and at each vertex the total time spent is $O(r)$.

Next we propose a generalization of the constrained longest common subsequence problem. Given strings S_1, S_2, and P, and a positive integer d the *edit distance* *constrained longest common subsequence* problem for S_1 and S_2 with respect to string P, and distance d is to find a longest common subsequence *lcs* of S_1 and S_2 such that this *lcs* has a subsequence whose edit distance from P is smaller than d. Edit distance between two strings is the minimum number of edit operations required to transform one string to the other. The edit operations are insert, delete, and substitute.

Let $L_{i,j,k,t}$ be the length of an *lcs* for $S_1[1..i]$ and $S_2[1..j]$ such that the common subsequence contains a subsequence whose edit distance from $P[1..k]$ is exactly t.

Example: Suppose $S_1 = \texttt{bbaba}$, $S_2 = \texttt{abbaa}$ and $P = \texttt{ab}$. We have calculated before that the length of an *lcs* for S_1 and S_2 relative to P is 3. Thus $L_{5,5,2,0} = 3$. On the other hand the *lcs* `bbaa` of S_1 and S_2 contains the subsequence `a`, which is edit distance 1 away from P. Therefore $L_{5,5,2,1} = 4$.

We calculate all $L_{i,j,k,t}$ by a dynamic programming formulation. Optimal value of the edit distance constrained *lcs* problem is $\max\limits_{0\le t<d} L_{n,m,r,t}$.

Theorem 2 *For all i,j,k,t, $1 \le i \le n$, $1 \le j \le m$, $0 \le k \le r$, $0 \le t < d$, $L_{i,j,k,t}$ satisfies*

$$L_{i,j,k,t} = \max\{L'_{i,j,k,t},\ L_{i,j-1,k,t},\ L_{i-1,j,k,t}\} \tag{3}$$

where

$$L'_{i,j,k,t} = \max\{L''_{i,j,k,t},\ L'''_{i,j,k,t},\ L''''_{i,j,k,t}\} \tag{4}$$

where

$$L''_{i,j,k,t} = \begin{cases} 1 + L_{i-1,j-1,k-1,t} & \text{if } ((k=1 \text{ and } t=0) \text{ or} \\ & (k>1 \text{ and } L_{i-1,j-1,k-1,t} > 0)) \\ & \quad \text{and } S_1[i] = S_2[j] = P[k] \\ & \\ 0 & \text{otherwise} \end{cases}$$

$$L'''_{i,j,k,t} = \begin{cases} 1 + L_{i-1,j-1,0,0} & \text{if } (k=0 \text{ and } t=1) \text{ and } S_1[i] = S_2[j] \\ 1 + L_{i-1,j-1,k,t} & \text{else if } (k=0 \text{ or } L_{i-1,j-1,k,t} > 0) \\ & \quad \text{and } S_1[i] = S_2[j] \\ 0 & \text{otherwise} \end{cases}$$

$$L''''_{i,j,k,t} = \max\{D_{i,j,k,t},\ X_{i,j,k,t},\ I_{i,j,k,t}\} \tag{5}$$

where

$$\begin{aligned} D_{i,j,k,t} &= \begin{cases} L_{i,j,k-1,t-1} & \text{if } t \ge 1 \\ 0 & \text{otherwise} \end{cases} \\ X_{i,j,k,t} &= \begin{cases} L_{i,j,k-1,t-1} & \text{if } t \ge 1 \text{ and } S_1[i] = S_2[j] \\ 0 & \text{otherwise} \end{cases} \\ I_{i,j,k,t} &= \begin{cases} L_{i,j,k,t-1} & \text{if } t \ge 1 \text{ and } S_1[i] = S_2[j] \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

with boundary conditions $L_{i,0,k,0} = 0$, $L_{0,j,k,0} = 0$, for all i,j,k, $0 \le i \le n$, $0 \le j \le m$, $0 \le k \le r$.

Proof We claim that $L_{i,j,k,t}$ is the optimum length for any lcs for $S_1[1..i]$ and $S_2[1..j]$ such that the lcs contains a subsequence whose edit distance is t from $P[1..k]$. We prove the correctness of our formulation by induction on t for all i, j, k.

In the base case: when $t = 0$ the formulation becomes the same formulation as in Theorem 1, since now the lcs is required to contain P itself as a subsequence. Therefore, the correctness of this case follows from Theorem 1.

Assume that for $t - 1$ ($t \geq 1$), for all i, j, k, $L_{i,j,k,t-1}$ is the optimum length for any lcs for $S_1[1..i]$ and $S_2[1..j]$ such that the lcs contains a subsequence whose edit distance is t from $P[1..k]$. Consider the calculation of $L_{i,j,k,t}$ for all i, j, k when $t > 1$.

Our formulation uses the following observation: Let cs be a subsequence of an lcs of S_1 and S_2. The minimum edit distance between cs and P can be calculated using insert, delete, and substitute operations in P, and using no operations in cs. To see this consider the edit operations between the symbols in cs, and in P. If an edit distance calculation deletes a symbol s in cs, we can instead insert the symbol s in P; if a minimum edit distance calculation inserts a symbol s in cs, we can instead delete the symbol s in P; and if a minimum edit distance calculation substitutes a symbol s' for s in cs, we can instead substitute a symbol s for s' in P to obtain the same edit distance.

We define an *edit path* at node (i, j) at distance t from $P[1..k]$ as a simple path from node $(0, 0)$ to node (i, j), which includes a sequence of $l \geq 1$ distinct nodes $(a_1, b_1), (a_2, b_2), \cdots, (a_l, b_l)$ such that the edit distance between the string $S_1[a_1]S_2[a_2] \ldots S_1[a_l]$ $(= S_2[b_1]\ S_2[b_2] \ldots S_2[b_l])$, and $P[1..k]$ is exactly t. We define $\#match$ on a given edit path to node (i, j) as the number of matching diagonal arcs on the path between the symbols in $S_1[1..i]$, and the symbols in $S_2[1..j]$, not necessarily involving matches in P. An optimal edit path at node (i, j) at distance t from $P[1..k]$ is an edit path at node (i, j) at distance t from $P[1..k]$ with maximum $\#match$. Thus $L_{i,j,k,t}$ is $\#match$ on an optimal edit path at node (i, j) at distance t from $P[1..k]$. In this case, $\#match = \#match(i, j, k, t)$ is a function of the indices i, j, k, t, but we omit these parameters when they are clear from the context.

We can extend any optimal edit path at node $(i, j - 1)$ at distance t from $P[1..k]$ with the horizontal arc $((i, j-1), (i, j))$ to obtain an edit path at node (i, j) at distance t from $P[1..k]$. Such an extension does not change $\#match$ on the resulting edit path, and $L_{i,j,k,t} \geq L_{i,j-1,k,t}$.

Similarly we can extend any optimal edit path at node $(i-1, j)$ at distance t from $P[1..k]$ with the vertical arc $((i - 1, j), (i, j))$ to obtain an edit path at node (i, j) at distance t from $P[1..k]$. This extension does not change $\#match$ on the resulting edit path, and $L_{i,j,k,t} \geq L_{i-1,j,k,t}$. Therefore, $L_{i,j,k,t} \geq \max\{L_{i,j-1,k,t},\ L_{i-1,j,k,t}\}$.

By using a matching arc $((i - 1, j - 1), (i, j))$, we can obtain edit paths at node (i, j) at distance t from $P[1..k]$ by extending optimal edit paths at node $(i - 1, j - 1)$ at distance $t - 1$, or t from $P[1..k - 1]$, or $P[1..k]$.

First consider optimal edit paths at node $(i-1, j-1)$ at distance t from $P[1..k-1]$. We will show that $L''_{i,j,k,t}$ stores the maximum $\#match$ obtained at node (i, j) by extending these edit paths.

If $S_1[i] = S_2[j] = P[k]$ then: We do not need to consider the case when $k = 1$ and $t = 0$ since $t = 0$ case is considered in the base case of the induction. If $k > 1$ and if there is an optimal edit path at node (i, j) at distance t from $P[1..k]$ (i.e. if

$L_{i-1,j-1,k-1,t} > 0$) then we can extend this edit path with a new match, and $\#match$ on the resulting edit path at node (i, j) at distance t from $P[1..k]$ becomes $L''_{i,j,k,t} = 1 + L_{i-1,j-1,k-1,t}$.

Next we consider optimal edit paths at node $(i-1, j-1)$ at distance t from $P[1..k]$. We will show that $L'''_{i,j,k,t}$ stores the maximum $\#match$ obtained at node (i, j) by extending these edit paths.

If $S_1[i] = S_2[j]$ then: If $k = 0$ and $t = 1$ then: We can extend an *lcs* path ending at node $(i-1, j-1)$ with respect to $P[1..k]$ with a match. In this case, $\#match$ in the resulting edit path is one more than $L_{i-1,j-1,0,0}$. Therefore, $L'''_{i,j,0,1} = 1 + L_{i-1,j-1,0,0}$. Otherwise if $k = 0$ then we can extend an optimal edit path at node $(i-1, j-1)$ at distance t from $P[1..k]$ with a match, and $\#match$ on the resulting edit path is $L'''_{i,j,k,t} = 1 + L_{i-1,j-1,k,t}$.

Any edit path at node (i, j) at distance $t-1$ from $P[1..k-1]$, or $P[1..k]$ can be modified by applying an edit operation in P. We can modify an edit path at node (i, j) at distance $t-1$ from $P[1..k-1]$ by deleting $P[k]$. Then on the resulting edit path $\#match$ remains the same, and the distance increases by 1. Therefore, we set $D_{i,j,k,t} = L_{i,j,k-1,t-1}$, and take it into account in $L''''_{i,j,k,t}$. We can modify an edit path at node (i, j) at distance $t-1$ from $P[1..k-1]$ by substituting $S_1[i] = S_2[j]$ for $P[k]$. Then on the resulting edit path $\#match$ remains the same, and the distance increases by 1. Therefore, we set $X_{i,j,k,t} = L_{i,j,k-1,t-1}$ if $S_1[i] = S_2[j]$, and take it into account in $L''''_{i,j,k,t}$. We can also modify an edit path at node (i, j) at distance $t-1$ from $P[1..k]$ by inserting $S_1[i] = S_2[j]$ in P after position k. Then on the resulting edit path $\#match$ remains the same, and the distance increases by 1. Therefore, we set $I_{i,j,k,t} = L_{i,j,k,t-1}$ if $S_1[i] = S_2[j]$, and take it into account in $L''''_{i,j,k,t}$. Combining all these $L''''_{i,j,k,t} = \max\{D_{i,j,k,t},\ X_{i,j,k,t},\ I_{i,j,k,t}\}$.

After setting $L'_{i,j,k,t} = \max\{L''_{i,j,k,t},\ L'''_{i,j,k,t},\ L''''_{i,j,k,t}\}$, $L'_{i,j,k,t}$ stores the maximum $\#match$ on edit paths at node (i, j) at distance t from $P[1..k]$ whose last arc is $((i-1, j-1), (i, j))$. If there is no such edit path then $L'_{i,j,k,t} = 0$.

From all possible optimal edit paths at neighboring nodes of (i, j) we can obtain their extensions ending at node (i, j), and we can find an optimal edit path at node (i, j) at distance t from $P[1..k]$ for all k, t. We calculate, and store in $L_{i,j,k,t}$ maximum $\#match$ in such optimal edit paths. Considering the possible cases of the last arc on an optimal edit path at node (i, j) at distance t from $P[1..k]$ we also have $L_{i,j,k,t} \leq \max\{L'_{i,j,k,t},\ L_{i,j-1,k,t},\ L_{i-1,j,k,t}\}$. This concludes the proof of the theorem. •

All $L_{n,m,r,t}$ for $t = 0, 1, \cdots, d-1$ can be computed in $O(drnm)$ time, and using $O(drm)$ space using the formulation in Theorem 2 by noting that we only need rows $i-1$, and i during the calculations at row i. If an actual optimal edit path is desired then we can carry the edit path information for every k and t along with the calculations. This requires $O(drnm)$ space since edit paths can be of length $O(n)$.

If we store match points (where the symbols of S_1, S_2, and P match) on these edit paths then we can reduce the required space to $O(dr^2m)$. In this case, the optimal edit path of the problem needs to be recovered using ordinary *lcs* computations to connect the consecutive match points.

Remark: Space complexity can further be improved by using the technique we used in our first algorithm. We can compute, instead of the entire edit path for each k, and t, a middle vertex $(n/2, j)$ (assume for simplicity that n is even) at which an edit path at distance t from $P[1..k]$ passes. This can be done in $O(drm)$ space, and we can compute for all k, and t, $\#match$ $L_{n/2,j,l,u}$ on optimal edit path from vertex $(0,0)$ to vertex $(n/2,j)$, and $\#match$ on optimal edit path from $(n/2,j)$ to (n,m) where $0 \le \ell \le k$, and $0 \le u \le t$. The latter, denoted by $L^{reverse}_{n/2,j,k-l,t-u}$, can be calculated in the reverse edit graph. Then for all k,t,

$$\max_{j,0\le\ell\le k,0\le u\le t} L_{n/2,j,l,u} + L^{reverse}_{n/2,j,k-l,t-u}$$

is the optimum $\#match$ for k,t, and it identifies a middle vertex. After the middle vertex $(n/2,j)$ on optimal edit path for every k,t is found, the problem of finding an optimal edit path from $(0,0)$ to (n,m) can be solved in two parts: find an optimal edit path from $(0,0)$ to $(n/2,j)$, and find and optimal edit path from $(n/2,j)$ to (n,m) for all k,t. These two subproblems can be solved recursively. As a results an optimal edit path can be obtained using $O(drm)$ space. The time complexity remains $O(rnm)$ because n is halved each time, and the area (in terms of number of vertices) covered in the edit graph is $O(nm)$, and at each vertex the total time spent is $O(dr)$.

3 Conclusion

We have improved the time complexity of the constrained *lcs* problem from $O(rn^2m^2)$ to $O(rnm)$ where n, and m are the lengths of the given strings, and r is the pattern length. This improvement is achieved by a dynamic programming formulation which is different from what was proposed in [6]. In our formulation, the dimensions are ordered differently. We also extended the problem definition to use edit distances, and presented an $O(drnm)$ time algorithm for the resulting edit distance constrained *lcs* problem.

References

[1] A.V. Aho, D.S. Hirschberg, and J.D. Ullman. Bounds on the complexity of the longest common subsequence problem. *J. ACM*, 23(1):1–12, 1976.

[2] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. *SPIRE, A Coruna, Spain*, pp. 39–48, 2000.

[3] D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of ACM*, 18:341–343, 1975.

[4] D.S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24:664–675, 1977.

[5] W.J. Masek and M.S. Paterson. A faster algorithm computing string edit distances. *J. Comput. System Sci.*, 20:18–31, 1980.

[6] Yin-Te Tsai. The constrained common sequence problem. *Information Processing Letters*, 88:173–176, 2003.

[7] R.A. Wagner and M.J. Fisher. The string-to-string correction problem. *J. ACM*, 21:168–173, 1974.

Efficient Algorithms for the δ-Approximate String Matching Problem in Musical Sequences

Domenico Cantone, Salvatore Cristofaro and Simone Faro

Dipartimento di Matematica e Informatica, Università di Catania, Italy

e-mail: {cantone, cristofaro, faro}@dmi.unict.it

Abstract. The δ-approximate string matching problem, recently introduced in connection with applications to music retrieval, is a generalization of the exact string matching problem for alphabets of integer numbers. In the δ-approximate variant, (exact) matching between any pair of symbols/integers a and b is replaced by the notion of δ-matching $=_\delta$, where $a =_\delta b$ if and only if $|a - b| \leq \delta$ for a given value of the approximation bound δ.
After surveying the state-of-the-art, we describe some new effective algorithms for the δ-matching problem, obtained by adapting existing string matching algorithms. The algorithms discussed in the paper are then compared with respect to a large set of experimental tests. From these, in particular it turns out that two of our newly proposed algorithms often achieve the best performances, especially in the case of large alphabets and short patterns, which typically occurs in practical situations in music retrieval.

Keywords: String algorithms, approximate string matching, musical information retrieval.

1 Introduction

Given a text T and a pattern P over some alphabet Σ, the *string matching problem* consists in finding *all* occurrences of the pattern P in the text T. It is a very extensively studied problem in computer science, mainly due to its direct applications to such diverse areas as text, image and signal processing, speech analysis and recognition, information retrieval, computational biology and chemistry, etc.

In the last few years also the approximate pattern matching problem has received much attention and algorithms which find all approximate repetitions of a given pattern in a sequence have been proposed, based on notions of approximation particularly useful in specific fields such as molecular biology [KMGL88, MJ93], musical applications [CIR98], or image processing [KPR00].

In this paper we focus on a variant of the approximate string matching problem which naturally arises in music information retrieval, namely the *δ-approximate string matching problem.*

Musical sequences can be schematically viewed as sequences of integer numbers, representing either the notes in the chromatic or diatonic notation (absolute pitch encoding), or the intervals, in number of semitones, between consecutive notes (pitch

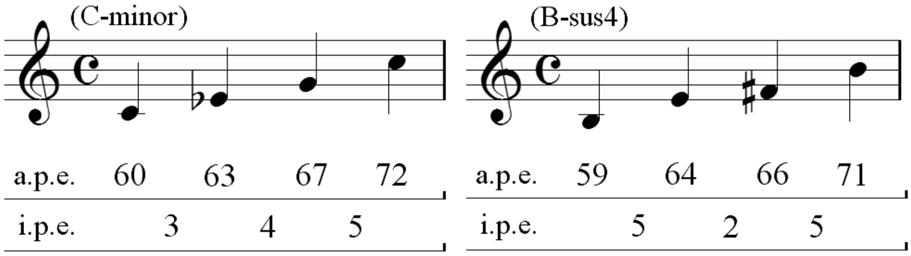


Figure 1: Representation of the C-minor and B-sus4 chords in the absolute pitch encoding (a.p.e.) and in the pitch interval encoding (i.p.e.)

interval encoding); see the examples in Figure 1. The second representation is generally of greater interest for applications in tonal music, since absolute pitch encoding disregards tonal qualities of pitches. Note durations and note accents can also be encoded in a numeric form, giving rise to more meaningful alphabets whose symbols can be really regarded as sets of parameters. This is the reason why alphabets used for music representation are generally quite large.

δ-approximate string matching algorithms are very effective to search for all similar but not necessarily identical occurrences of given (short) melodies in musical scores. We recall that in the δ-approximate matching problem two numeric strings of the same length match if corresponding integers differ by at most a fixed bound δ. For instance, the chords C-minor and B-sus4 match if a tolerance of $\delta = 1$ is allowed in the absolute pitch encoding (where C-minor$= (60, 63, 67, 72)$ and B-sus4$= (59, 64, 66, 71)$), whereas if we use the pitch interval encoding, a tolerance of $\delta = 2$ is needed to get a match (in this case we have C-minor$= (3, 4, 5)$ and B-sus4$= (5, 2, 5)$); see Figure 1. Notice that when $\delta = 0$, the δ-approximate string matching problem reduces to the exact string matching problem.

A stronger restriction can be introduced to δ-approximate matching by imposing a limit γ to the sum of the absolute differences between corresponding integers. This further restriction is generally referred to as (δ, γ)-approximate matching. However, in this paper we consider only the general case in which $\gamma = +\infty$.

A significant amount of research has been devoted to adapt solutions for exact string matching to δ-approximate matching (see for instance [CCI$^+$99, CILP01, CIL$^+$02]). In this respect, Boyer-Moore-type algorithms are of particular interest, since they are very fast. We recall that they are based on variations of the well-known ideas introduced in the Boyer-Moore algorithm [BM77], namely right-to-left scanning, bad-character and good-suffix heuristics. For instance, the Fast-Search and the Forward-Fast-Search algorithms [CF03a, CF03b] require that the bad-character heuristics is used only if the mismatching character is the last character of the pattern, otherwise the good-suffix heuristics is to be used.

The main results of this paper are adaptations of the Fast-Search and Forward-Fast-Search algorithms to δ-approximate matching. In addition, we propose adaptations of the Quick-Search and the Berry-Ravindran algorithms [Sun90, BR99], which are among the most efficient algorithms for exact string matching.

The paper is organized as follows. In Section 2 we introduce the basic notions and give a formal definition of the δ-approximate matching problem. In Section 3 we survey some of the most efficient algorithms for computing δ-approximate repetitions in musical sequences. Then in Sections 4 and 5 we present new efficient variants, by suitably adapting known exact string matching algorithms. Experimental data obtained by running under various conditions the most efficient reviewed algorithms are presented and compared in Section 6. Finally, we draw our conclusions in Section 7.

2 Basic definitions and properties

Before entering into details, we need a bit of notations and terminology. A string P is represented as a finite array $P[0\,..\,m-1]$, with $m \geq 0$. In such a case we say that P has length m and write $\mathrm{length}(P) = m$. In particular, for $m = 0$ we obtain the empty string, also denoted by ε. By $P[i]$ we denote the $(i+1)$-st character of P, for $0 \leq i < \mathrm{length}(P)$. 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 $0 \leq i \leq j < \mathrm{length}(P)$. Moreover, for any $i, j \in \mathbb{Z}$, we put

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

For any two strings P and Q, we write $Q \sqsupset P$ to indicate that Q is a suffix of P, i.e., $Q = P[i\,..\,\mathrm{length}(P)-1]$, for some $0 \leq i \leq \mathrm{length}(P)$. Similarly, we write $Q \sqsubset P$ to indicate that Q is a prefix of P, i.e., $Q = P[0\,..\,i-1]$, for some $0 \leq i \leq \mathrm{length}(P)$. In addition, we write $P.Q$ to denote the concatenation of P and Q and P^k to denote the concatenation of k copies of P. A prefix Q of P is a *period* of P if P is a prefix of Q^k, for a sufficiently large k. The shortest period of P is called *the period* of P.

Let Σ be an alphabet of integer numbers and let $\delta \geq 0$ be an integer. Two symbols a and b of Σ are said to be *δ-approximate* (or that a and b *δ-match*), in which case we write $a =_\delta b$, if $|a - b| \leq \delta$. Two strings P and Q over the alphabet Σ are said to be δ-approximate (or that P and Q δ-match), in which case we write $P \stackrel{\delta}{=} Q$, if

$$\mathrm{length}(P) = \mathrm{length}(Q), \quad \text{ and } P[i] =_\delta Q[i], \text{ for } i = 0, ..., \mathrm{length}(P) - 1\,.$$

Moreover, we write $Q \stackrel{\delta}{\sqsupset} P$, and say that Q is a *δ-suffix* of P, if $Q \stackrel{\delta}{=} P[i\,..\,\mathrm{length}(P) - 1]$, for some $0 \leq i \leq \mathrm{length}(P)$. The following elementary property can be verified immediately.

Property 2.1 *Let P, Q, and R be strings over an alphabet Σ of integer numbers and let $\delta \geq 0$ be a given integer number. If $P \stackrel{\delta}{=} Q$ and $R \stackrel{\delta}{\sqsupset} Q$, then $R \stackrel{2\delta}{\sqsupset} P$.*

Let T be a text of length n and let P be a pattern of length m (over a given alphabet of integers). When the symbol $P[0]$ is aligned with the symbol $T[s]$ of the text, so that the symbol $P[i]$ is aligned with the symbol $T[s+i]$, for $i = 0, \ldots, m-1$, we say that the pattern P has *shift* s in T. In this case, the substring $T[s\,..\,s+m-1]$ is called the *current window* of the text. If $T[s\,..\,s+m-1] \stackrel{\delta}{=} P$, we say that the shift s is *δ-valid*. Then the problem of δ-approximate pattern matching consists in finding all δ-valid shifts of text T for a given pattern P.

3 Fast δ-Approximate Pattern Matching

The problem of δ-approximate matching in musical sequences has been formally defined in [CCI+99], where an algorithm based on the bitwise technique, the Shift-And algorithm, has been presented. The Shift-And algorithm uses a constant time state computation, for each character in the text, so that the overall time complexity of the searching phase is $\mathcal{O}(n)$.

Two years later, a number of efficient algorithms for the exact string matching problem have been adapted in [CILP01] to the δ-approximate matching problem, obtaining three algorithms based on occurrence heuristics (δ-Tuned-Boyer-Moore, δ-Skip-Search, and δ-Maximal-Shift) which are faster than the Shift-And algorithm.

Still later, other adaptations of exact string matching algorithms to the δ-approximate matching problem have been proposed in [CIL+02]. The resulting algorithms, based on the substring heuristics, are δ-BM1, δ-BM2, and δ-BM3.

Finally, a bit-parallel algorithm, δ-BNDM, which outperforms previous algorithms, has been recently presented in [CIPN03].

Next, we briefly review the most efficient algorithms for δ-approximate matching mentioned above.

3.1 δ-Boyer-Moore Algorithms

The Boyer-Moore algorithm [BM77] for exact pattern matching is the progenitor of several algorithmic variants which aim at computing efficiently shift increments which are close to optimal. Specifically, the Boyer-Moore algorithm checks whether a shift s of a text T is valid by scanning the pattern P from right to left. At the end of the matching phase, the Boyer-Moore algorithm computes the shift increment as the maximum value suggested by the *good-suffix rule* and the *bad-character rule*.

The Boyer-Moore bad-character rule can be easily adapted to δ-approximate matching. Suppose that the first δ-mismatch occurs at position i of the pattern, i.e. $P[i+1,...,m-1] \stackrel{\delta}{=} T[s+i+1,...,s+m-1]$ and $T[s+i] \neq_\delta P[i]$. Then the δ-bad-character rule suggests the shift increment $\delta\text{-}bc_P(T[s+i]) + i - m + 1$, where

$$\delta\text{-}bc_P(c) =_{\text{Def}} \min(\{0 \leq k < m \mid P[m-1-k] =_\delta c\} \cup \{m\}) \ ,$$

for $c \in \Sigma$.

The δ-Tuned-Boyer-Moore [CILP01] is an adaptation of Tuned-Boyer-Moore [HS91], which in turn is a very efficient simplification of the original Boyer-Moore algorithm although it runs in time $\mathcal{O}(nm)$ in the worst case. Specifically, each of its iterations can be divided into two phases: *last character localization* and δ*-matching phase*. The first phase searches for a δ-match of the character $P[m-1]$, by applying until needed rounds of three consecutive shifts based on the δ-bad-character rule, where the check $\delta\text{-}bc_P(T[s+m-1]) = 0$ is performed only at the end of each round. The subsequent matching phase then tries to δ-match the rest of the pattern $P[0\,..\,m-2]$ with the corresponding characters of the text, proceeding from right to left. At the end of the matching phase, the δ-Tuned-Boyer-Moore algorithm computes the shift advancement in such a way that character $T[s+m-1]$ is aligned with the rightmost position i on $P[0\,..\,m-2]$ such that $P[m-1] =_{2\delta} P[i]$, if present. If such position is not present, the shift is incremented by m.

The δ-Maximal-Shift algorithm, presented in [CILP01], is a modification of the Maximal-Shift algorithm [Sun90]. Rather than scanning the pattern from right to left, the δ-Maximal-Shift algorithm scans the pattern according to an ordering which guarantees larger shifts advancements. This is better formalized by means of a permutation $\pi : \{0, 1, ..., m\} \rightarrow \{0, 1, ..., m\}$ and a function δ-$max : \{0, 1, ..., m\} \rightarrow \{0, 1, ..., m\}$ such that, for $0 \leq i < m$,

$$\delta\text{-}max(\pi(i)) = \min\{l \,|\, P[\pi(j)-l] =_{2\delta} P[\pi(j)] \text{ and } P[\pi(i)-l] \neq_{\delta} P[\pi(i)], \text{ for } 1 \leq j < i\}$$

and, for $0 \leq i < m - 1$,

$$\delta\text{-}max(\pi(i)) \leq \delta\text{-}max(\pi(i+1)) \,,$$

where P is a given pattern of length m. Furthermore, one sets $\pi(m) = m$ and δ-$max(m)$ equal to the length of the period of P.

During the matching phase, characters are scanned in the pattern following the ordering $\pi(0), \pi(1), \ldots, \pi(m-1)$. Moreover, if the first δ-mismatch occurs while comparing characters $P[\pi(i)]$ and $T[s + \pi(i)]$, then the current shift s is incremented by $\max\{\delta\text{-}max(\pi(i)), \delta\text{-}bc(T[s+m])\}$. It turns out that the δ-Maximal-Shift algorithm has $\mathcal{O}(nm)$ time complexity.

3.2 δ-Reverse-Factor and δ-Alpha-Skip-Search Algorithms

Unlike the Boyer-Moore algorithm, the Reverse-Factor algorithm [CCG+94] and the Alpha-Skip-Search algorithm [CLP98] compute shifts which match prefixes of the pattern, rather than suffixes. These algorithms have a quadratic worst-case time complexity, but are very fast in practice (cf. [Lec00]). Moreover, it has been shown that on the average the Reverse-Factor algorithm inspects $\mathcal{O}(n \log(m)/m)$ text characters, reaching the best bound shown by Yao in 1979 (cf. [Yao79]).

Adaptations of Reverse-Factor and Alpha-Skip-Search algorithms are presented in [CIL+02] under the names δ-BM2 and δ-BM1, respectively.

The δ-BM1 algorithm, or δ-Alpha-Skip-Search algorithm, preliminary computes a δ-suffix trie $\mathcal{T}_x$ of all the factors of length $\ell = \lfloor \log_{|\Sigma|} m \rfloor$ occurring in the pattern P, where Σ is the alphabet. The leaves of the δ-suffix trie $\mathcal{T}_x$ represent all strings y such that $y \stackrel{\delta}{=} x$, for some factor x of P of length ℓ. For each leaf of $\mathcal{T}_x$, a bucket is maintained which stores all positions at which the factor associated to the leaf occurs in P. The searching phase of the δ-Alpha-Skip-Search algorithm consists then in looking for each shift position s into the bucket of the factor $T[s..s + \ell - 1]$, if any, and in checking naively the corresponding windows of the text. A shift advancement of size $m - \ell + 1$ then takes place.

The δ-BM2 algorithm, or δ-Reverse-Factor algorithm, computes the smallest δ-suffix automaton of the reverse of the pattern P by simply minimizing the δ-suffix trie $\mathcal{T}_x$. In this way one obtains a deterministic finite automaton whose accepted language is the set of strings y such that $y \stackrel{\delta}{=} x$, for some factor x of P of length $\lfloor \log_{|\Sigma|} m \rfloor$. Then, much the same strategy of the Reverse-Factor algorithm can be applied to the case of δ-approximate matching.

3.3 δ-Hashing Algorithms

To describe the δ-BM3 algorithm, we need some further notation. Let P be a pattern over an alphabet Σ of integer numbers, let $k < \text{length}(P)$ be a fixed integer, and let $\delta \geq 0$ be a given approximation bound. We denote by $sub(P, k)$ the set of all substrings of P of length k and we define the following intervals:

$$\begin{aligned} \mathcal{I} &= [k \cdot \min \Sigma \,..\, k \cdot \max \Sigma] \\ \mathcal{I}_x &= [\max\{\text{hash}(x) - k\delta, k \cdot \min \Sigma\} \,..\, \min\{\text{hash}(x) + k\delta, k \cdot \max \Sigma\}]\,, \end{aligned}$$

for $x \in sub(P, k)$ and where hash(x) denotes the sum of the symbols of x. It can be easily shown that $\mathcal{I}_x \subseteq \mathcal{I}$, for each $x \in sub(P, k)$.

The δ-BM3 algorithm [CIL+02] begins by constructing the following hash table $\mathcal{H}$, indexed by the interval $\mathcal{I}$. For each $i \in \mathcal{I}$, the i-th bucket of the table $\mathcal{H}$ collects the positions of all subwords $x \in sub(P, k)$ such that $i \in \mathcal{I}_x$. Then, given a text T, for each shift position s the searching phase of the δ-BM3 algorithm consists in examining the subword $y = T[s+m-k \,..\, s+m-1]$. For each element j in the bucket at position $hash(y)$, the algorithm checks naively whether P occurs at position $s + m - k - j$ in T. It turns out that the choice of k influences the running-time of the algorithm. Generally, a value of $k = 2$ constitutes a good choice.

The δ-Skip-Search algorithm [CILP01] is an adaptation of the Skip-Search algorithm [CLP98] to δ-approximate matching. However, it can also be seen as a variant of the δ-BM3 algorithm, with $k = 1$.

Both δ-Skip-Search and δ-BM3 algorithms are fast in practice although their worst-case time complexity is $\mathcal{O}(nm)$.

3.4 The δ-BNDM Algorithm

The Backward Nondeterministic DAWG Matching algorithm (BNDM for short) for exact string matching has been presented in [NR98] as a combination of the bit-parallel algorithm Shift-Or [BYG89] and the BDM algorithm [CCG+94] based on suffix automata. The aim of the BNDM algorithm is to combine the property of skipping characters (as in the BDM algorithm) with that of simulating nondeterministic automata (as the Shift-Or algorithm). It turns out that the BNDM algorithm obtains better results in terms of running time than the Shift-Or and the BDM algorithms. Its complexity is $\mathcal{O}(nm)$ in the worst case.

An adaptation of the BNDM algorithm to δ-approximate matching, the δ-BNDM algorithm, has been presented in [CIPN03]. The δ-BNDM algorithms is very simple and efficient, especially in the case of long patterns, and is considered a stronger choice for the δ-approximate matching problem.

4 δ-Fast-Search algorithms

The Fast-Search [CF03a] and the Forward-Fast-Search [CF03b] algorithms are very recent members of the large family of Boyer-Moore type string matching algorithms. Their searching strategy is based on an efficient mixing of the bad-character and good-suffix rules, as given in the original Boyer-Moore algorithm. Recent experimental results [CF03b] conducted over an extensive family of string matching algorithms show that the Fast-Search algorithms obtain, in most cases, the best results in terms of running times and number of text character inspections.

After reviewing the main features of the Fast-Search algorithms, we shall show how they can be adapted to δ-approximate matching problem.

4.1 Fast-Search and Forward-Fast-Search algorithms

The Fast-Search algorithm is a very simple, yet efficient, variant of the Boyer-Moore algorithm. Again, let P be a pattern of length m and let T be a text of length n over a finite alphabet Σ. The Fast-Search algorithm computes its shift increments by applying the bad-character rule if and only if a mismatch occurs during the first character comparison, namely, while comparing characters $P[m-1]$ and $T[s+m-1]$, where s is the current shift. Otherwise it uses the good-suffix rule.

Specifically, if the first mismatch occurs at position $i < m-1$ of the pattern P, the good-suffix rule suggests to align the substring $T[s+i+1\,..\,s+m-1] = P[i+1\,..\,m-1]$ with its rightmost occurrence in P preceded by a character different from $P[i]$. If such an occurrence does not exist, the good-suffix rule suggests a shift increment which allows to match the longest suffix of $T[s+i+1\,..\,s+m-1]$ with a prefix of P.

More formally, if the first mismatch occurs at position i of the pattern P, the good-suffix rule states that the shift can be safely incremented by $gs_P(i+1)$ positions, where

$$\begin{aligned} gs_P(j) =_{\text{Def}} \min\{0 < k \leq m \quad | \quad & P[j-k\,..\,m-k-1] \sqsupset P \\ & \text{and } (k \leq j-1 \rightarrow P[j-1] \neq P[j-1-k])\}\ , \end{aligned}$$

for $j = 0, 1, \ldots, m$. (The situation in which an occurrence of the pattern P is found can be regarded as a mismatch at position -1.)

A more effective implementation of the Fast-Search algorithm is obtained along the same lines of the Tuned-Boyer-Moore algorithm: the bad-character rule can be iterated until the last character $P[m-1]$ of the pattern is matched correctly against the text. At this point it is known that $T[s+m-1] = P[m-1]$, so that the subsequent matching phase can start with the $(m-2)$-*nd* character of the pattern. At the end of the matching phase the algorithm uses the good-suffix rule for shifting. Moreover the Fast-Search algorithm benefits from the introduction of an external sentinel, which allows to compute correctly the last shifts with no extra checks.

The Forward-Fast-Search algorithm maintains the same structure of the Fast-Search algorithm, but it is based upon a modified version of the good-suffix rule, called *forward good-suffix* rule, which uses a look-ahead character to determine larger shift advancements. Thus, if the first mismatch occurs at position $i < m-1$ of the pattern P, the forward good-suffix rule suggests to align the substring $T[s+i+1\,..\,s+m]$ with its rightmost occurrence in P preceded by a character different from $P[i]$. If such an occurrence does not exist, the forward good-suffix rule proposes a shift increment which allows to match the longest suffix of $T[s+i+1\,..\,s+m]$ with a prefix of P. This corresponds to advance the shift s by $\overrightarrow{gs}_P(i+1, T[s+m])$ positions, where

$$\begin{aligned} \overrightarrow{gs}_P(j,c) =_{\text{Def}} \min(\{0 < k \leq m \quad | \quad & P[j-k\,..\,m-k-1] \sqsupset P \\ & \text{and } (k \leq j-1 \rightarrow P[j-1] \neq P[j-1-k]) \\ & \text{and } P[m-k] = c\} \ \cup \ \{m+1\})\ , \end{aligned}$$

for $j = 0, 1, \ldots, m$ and $c \in \Sigma$.

The good-suffix rule and the forward good-suffix rule require tables of size m and $m \cdot |\Sigma|$, respectively. These can be constructed in time $\mathcal{O}(m)$ and $\mathcal{O}(m \cdot \max(m, |\Sigma|))$, respectively.

4.2 Adaptations to δ-approximate matching

In this section we show how to adapt the Fast-Search and Forward-Fast-Search algorithms to δ-approximate matching.

A modification of the bad-character rule to δ-approximate matching has been already presented in Section 3.1. Now we show how the good-suffix rule and the forward good-suffix rule can also be adapted to match δ-approximate repetitions of suffixes of the pattern.

Let us suppose that while comparing the pattern P with the window $T[s \mathinner{..} s + m - 1]$, proceeding from right to left, the first δ-mismatch occurs at position i, i.e. $P[i] \neq_\delta T[s + i]$ and $P[i + 1 \mathinner{..} m - 1] \stackrel{\delta}{=} T[s + i + 1 \mathinner{..} s + m - 1]$ (if we have a δ-match, then $i = 0$ and the condition $P[i] \neq_\delta T[s + i]$ should not be considered). Let $0 < k \leq m$ be such that $s + k + m - 1 \leq n$ and $P[i + 1 - k \mathinner{..} m - 1 - k] \stackrel{2\delta}{\not\sqsupset} P$, where $\stackrel{\delta}{\sqsupset}$ is the δ-suffix relation defined in Section 2. Then the shift $s + k$ is not δ-valid. Indeed, if $s + k$ were δ-valid, we would have $P \stackrel{\delta}{=} T[s + k \mathinner{..} s + k + m - 1]$, so that $P[i+1-k \mathinner{..} m-1-k] \stackrel{\delta}{\sqsupset} T[s+i+1 \mathinner{..} s+m-1]$. Therefore, by Property 2.1, we would get $P[i+1-k \mathinner{..} m-1-k] \stackrel{2\delta}{\sqsupset} P[i+1 \mathinner{..} m-1]$, which yields $P[i+1-k \mathinner{..} m-1-k] \stackrel{2\delta}{\sqsupset} P$, a contradiction. It is also easy to verify that if an integer k satisfies both $0 < k \leq i$ and $P[i] = P[i - k]$, then again the shift $s + k$ is not δ-valid. The above considerations allow us to state the following *δ-good-suffix rule*: if the first δ-mismatch occurs at position i of the pattern P, then the shift can be safely incremented by $\delta\text{-}gs_P(i + 1)$ positions, where

$$\begin{aligned}\delta\text{-}gs_P(j) =_{\text{Def}} \min\{0 < k \leq m \quad | \quad & P[j - k \mathinner{..} m - k - 1] \stackrel{2\delta}{\sqsupset} P[j \ldots m - 1] \\ & \text{and } (k \leq j - 1 \rightarrow P[j - 1] \neq P[j - 1 - k])\} \ ,\end{aligned}$$

for $j = 0, 1, \ldots, m$.

Much in the same way, one can verify the correctness of the following *δ-forward good-suffix rule*: if the first δ-mismatch occurs at position i of the pattern P, then the shift can be safely incremented by $\delta\text{-}\overrightarrow{gs}_P(i + 1, T[s + m])$ positions, where

$$\begin{aligned}\delta\text{-}\overrightarrow{gs}_P(j, c) =_{\text{Def}} \min(\{0 < k \leq m \quad | \quad & P[j - k \mathinner{..} m - k - 1] \stackrel{2\delta}{\sqsupset} P[j \ldots m - 1] \\ & \text{and } (k \leq j - 1 \rightarrow P[j - 1] \neq P[j - 1 - k]) \\ & \text{and } P[m - k] =_\delta c\} \ \cup \ \{m + 1\}) \ ,\end{aligned}$$

for $j = 0, 1, \ldots, m$ and $c \in \Sigma$.

The δ-good-suffix rule and the δ-forward good-suffix rule require tables of size m and $(m \cdot |\Sigma|)$, respectively. These can be easily constructed in time $\mathcal{O}(m)$ and $\mathcal{O}(\delta \cdot m \cdot \max(m, |\Sigma|))$, respectively.

The δ-Fast-Search and δ-Forward-Fast-Search algorithms can be implemented much along the same lines of the δ-Tuned-Boyer-Moore algorithm.

δ-Fast-Search (P, T)
- $n = \text{length}(T)$
- $m = \text{length}(P)$
- $T' = T.P[m-1]^{m+1}$
- $\delta\text{-}bc =$ precompute-δ-bad-character(P)
- $\delta\text{-}gs =$ precompute-δ-good-suffix(P)
- $s = 0$
- **while** $\delta\text{-}bc[T'[s+m-1]] > 0$ **do**
  - $s = s + \delta\text{-}bc[T'[s+m-1]]$
- **while** $s \leq n - m$ **do**
  - $j = m - 2$
  - **while** $j \geq 0$ **and** $P[j] =_\delta T'[s+j]$
    - **do** $j = j - 1$
  - **if** $j < 0$ **then**
    - print(s)
  - $s = s + \delta\text{-}gs[j+1]$
  - **while** $\delta\text{-}bc[T'[s+m-1]] > 0$ **do**
    - $s = s + \delta\text{-}bc[T'[s+m-1]]$

δ-Forward-Fast-Search (P, T)
- $n = \text{length}(T)$
- $m = \text{length}(P)$
- $T' = T.P[m-1]^{m+1}$
- $\delta\text{-}bc =$ precompute-δ-bad-character(P)
- $\delta\text{-}\overrightarrow{gs} =$ precompute-δ-forward-good-suffix(P)
- $s = 0$
- **while** $\delta\text{-}bc[T'[s+m-1]] > 0$ **do**
  - $s = s + \delta\text{-}bc[T'[s+m-1]]$
- **while** $s \leq n - m$ **do**
  - $j = m - 2$
  - **while** $j \geq 0$ **and** $P[j] =_\delta T'[s+j]$
    - **do** $j = j - 1$
  - **if** $j < 0$ **then**
    - print(s)
  - $s = s + \delta\text{-}\overrightarrow{gs}[j+1, T[s+m]]$
  - **while** $\delta\text{-}bc[T'[s+m-1]] > 0$ **do**
    - $s = s + \delta\text{-}bc[T'[s+m-1]]$

Figure 2: δ-Fast-Search and δ-Forward-Fast-Search algorithms

Each iteration of both algorithms can be divided into two phases. In the first phase, called *character localization* phase, the δ-bad-character rule is iterated until the last character $P[m-1]$ of the pattern is δ-matched correctly against the text. More precisely, starting from a shift position s, if we denote by j_i the total shift advancement after the i-th iteration of the δ-bad-character rule, then we have the following recurrence:

$$j_i = j_{i-1} + \delta\text{-}bc_P(T[s + j_{i-1} + m - 1]) \ .$$

Therefore, the δ-bad-character rule is applied k times in a row, where $k = \min\{i \mid T[s + j_i + m - 1] =_\delta P[m-1]\}$, with an overall shift advancement of j_k.

At this point we have that $T[s + j_k + m - 1] =_\delta P[m-1]$, so that the subsequent *δ-matching* phase can test for a δ-occurrence of the pattern by comparing only the remaining $(m-1)$ characters of the pattern. At the end of the δ-matching phase, the δ-good-suffix or the δ-forward-good-suffix rule is applied for computing the next shift.

In order to compute correctly the last shift with no extra checks, it is convenient to add $m+1$ copies of the character $P[m-1]$ at the end of the text T, obtaining the new text $T' = T.P[m-1]^{m+1}$. Plainly, all the δ-valid shifts of P in T are exactly the δ-valid shifts s of P in T' such that $s \leq n - m$, where, as usual, n and m denote respectively the lengths of T and P. The codes of the δ-Fast-Search and δ-Forward-Fast-Search algorithms are presented in Figure 2.

5 Other interesting efficient variants

In this section we present adaptations to δ-approximate matching of two efficient exact string matching algorithms based on the bad-character rule, i.e. the Quick-Search algorithm and the Berry-Ravindran algorithm.

The Quick-Search algorithm, presented in [Sun90], uses a simple modification of the original heuristics of the Boyer-Moore algorithm. Specifically, it is based on the following observation: when a mismatch character is encountered, the pattern is always shifted to the right by at least one character, but never by more than m characters. Thus, the character $T[s+m]$ is always involved in testing for the next alignment. So, one can apply the bad-character rule to $T[s+m]$, rather than to the mismatching character, obtaining larger shift advancements. Moreover the good-suffix rule of the original Boyer-Moore algorithm is not used at all.

Extending this idea to δ-approximate matching we obtain the δ-Quick-Search algorithm which, after each δ-matching phase, advances the shift by $\delta\text{-}qbc_P(T[s+m])$ positions, where

$$\delta\text{-}qbc_P(c) =_{\mathrm{Def}} \min(\{0 < k \leq m \mid P[m-k] =_\delta c\} \cup \{m+1\}) \ .$$

The function $\delta\text{-}qbc_P$ can be precomputed in $\mathcal{O}(m \cdot \delta + |\Sigma|)$-time and $\mathcal{O}(|\Sigma|)$-space complexity.

The δ-Berry-Ravindran algorithm is a modification of the Berry-Ravindran algorithm [BR99]. It extends the δ-Quick-Search algorithm in that its bad-character rule uses the two characters $T[s+m]$ and $T[s+m+1]$ rather than just the last character $T[s+m]$ of the window. Thus, at the end of each matching phase with shift s, the δ-Berry-Ravindran algorithm advances the pattern in such a way that the substring of the text $T[s+m\,..\,s+m+1]$ is aligned with the rightmost δ-occurrence in P of a substring c_1c_2 such that $T[s+m \ldots s+m+1] \stackrel{\delta}{=} c_1c_2$.

The precomputation of the table used by this version of δ-bad-character rule requires $\mathcal{O}(m \cdot \delta^2 + |\Sigma|^2)$-time and $\mathcal{O}(|\Sigma|^2)$-space complexity.

Experimental results confirm the good practical behavior of the Quick-Search and Berry-Ravindran algorithms even in the case of their δ-variants (see next section).

6 Experimental Results

In this section we report experimental data related to the most efficient δ-approximate string matching algorithms described above, namely δ-Tuned-Boyer-Moore (δ-TBM), δ-Quick-Search (δ-QS), δ-Berry-Ravindran (δ-BR), δ-BNDM (δ-BNDM), δ-Fast-Search (δ-FS), and δ-Forward-Fast-Search (δ-FFS).

We have chosen to compare them in terms of their running time. All algorithms have been implemented in the **C** programming language and were used to search for the same patterns in large fixed text sequences on a PC with a Pentium IV processor at 2.6GHz. In particular, they have been tested on three Randσ problems, for $\sigma = 30, 60, 120$, and on a real music sequence with patterns of length $m = 2, 4, 6, 8, 10, 15, 20, 25$, and 30.

Each Randσ problem consists in searching a set of 300 random patterns of a given length in a 20Mb random text sequence over a common alphabet of size σ.

The tests on the real music text buffer have been performed on a 9.3Mb file obtained by combining a set of classical pieces, in MIDI format, by J.S. Bach. The resulting text buffer has been translated in the pitch interval encoding with an alphabet of 55 symbols. For each pattern length m, we have randomly selected 200 substrings of length m in the file which subsequently have been searched for in the same file.

For both Randσ problems and real music problems, the value of the bound δ has been set to 1, 2, and 4.

In the following tables running times have been expressed in milliseconds and, for each length of the pattern, the best results have been bold-faced.

From the experimental results, it turns out that the δ-BNDM algorithm is a very good choice for the δ-approximate matching problem, especially when the pattern is long or the size of the alphabet is small.

However, the δ-Fast-Search algorithms compares well with the δ-BNDM algorithm and outperforms it in the case of quite short patterns and large alphabets, which occurs most frequently in real musical information retrieval problems.

Observe also that the δ-Tuned-Boyer-Moore algorithm and the δ-Quick-Search algorithm obtain good results in most cases and, additionally, the δ-Quick-Search algorithm reaches competitive results in the case of long patterns and large alphabets.

$\sigma = 30, \delta = 1$	2	4	6	8	10	15	20	25	30
δ-QS	93.08	68.88	59.19	54.48	51.71	49.12	48.23	48.16	47.85
δ-TBM	75.83	55.44	49.84	47.82	47.42	46.63	46.02	45.57	46.02
δ-BR	140.67	102.82	84.48	73.38	65.99	55.33	51.11	48.68	47.44
δ-FFS	74.67	54.84	49.45	**47.42**	**46.57**	**45.85**	**45.39**	45.73	45.10
δ-FS	**74.35**	**54.11**	**49.21**	47.63	46.92	46.00	45.56	45.54	45.64
δ-BNDM	125.89	90.41	73.61	62.92	55.80	48.03	46.10	**45.29**	**44.46**

Rand30 problem with $\delta = 1$

$\sigma = 30, \delta = 2$	2	4	6	8	10	15	20	25	30
δ-QS	118.91	87.03	73.50	67.25	63.04	58.90	57.20	56.82	56.26
δ-TBM	94.59	69.52	59.58	56.80	54.40	53.07	52.49	52.40	52.00
δ-BR	171.74	125.04	102.18	87.26	78.73	65.21	57.87	53.57	51.79
δ-FFS	**93.44**	**67.40**	**58.26**	**54.41**	**52.21**	**49.68**	48.36	48.10	47.92
δ-FS	93.80	68.19	58.54	55.76	52.95	51.40	51.29	50.05	49.91
δ-BNDM	158.89	107.54	80.57	66.42	58.79	50.21	**47.57**	**46.78**	**45.26**

Rand30 problem with $\delta = 2$

$\sigma = 30, \delta = 4$	2	4	6	8	10	15	20	25	30
δ-QS	166.00	132.47	112.36	105.59	101.12	98.74	97.91	98.79	97.91
δ-TBM	**124.15**	103.95	92.90	91.22	88.08	88.79	87.16	87.17	86.58
δ-BR	229.80	178.12	145.58	126.55	114.72	96.98	88.09	83.17	79.27
δ-FFS	127.36	**101.16**	**85.88**	**80.17**	75.31	70.70	67.14	65.06	63.38
δ-FS	131.92	105.31	92.44	88.90	84.92	84.04	80.76	80.22	78.55
δ-BNDM	215.78	134.42	98.63	82.78	**72.84**	**58.07**	**51.43**	**48.19**	**47.18**

Rand30 problem with $\delta = 4$

$\sigma = 60, \delta = 1$	2	4	6	8	10	15	20	25	30
δ-QS	74.92	57.49	51.07	48.74	47.46	46.02	45.45	45.35	45.27
δ-TBM	62.48	49.22	46.21	45.96	46.53	45.54	44.85	44.95	44.49
δ-BR	122.15	89.97	74.42	65.24	58.70	50.39	48.54	46.46	46.25
δ-FFS	62.63	49.08	46.71	45.39	**45.16**	45.14	**44.14**	44.71	44.17
δ-FS	**62.01**	**48.71**	**45.60**	**45.17**	45.40	**44.99**	44.49	**44.35**	**44.10**
δ-BNDM	102.68	72.16	62.16	56.48	52.32	47.60	45.79	44.60	44.22

Rand60 problem with $\delta = 1$

$\sigma = 60, \delta = 2$	2	4	6	8	10	15	20	25	30
δ-QS	86.46	64.67	55.84	52.05	50.04	47.76	47.03	46.48	46.11
δ-TBM	71.12	52.55	48.45	46.71	46.81	46.10	45.41	45.50	45.56
δ-BR	134.57	98.74	81.02	70.64	63.66	53.73	50.15	48.07	46.85
δ-FFS	**70.06**	52.58	48.42	**46.31**	**45.85**	45.60	45.40	**44.77**	45.03
δ-FS	70.21	**51.79**	**47.32**	46.36	46.02	**45.17**	**44.97**	44.91	**44.43**
δ-BNDM	117.93	84.61	70.87	62.07	55.01	47.77	45.73	44.98	44.77

Rand60 problem with $\delta = 2$

$\sigma = 60, \delta = 4$	2	4	6	8	10	15	20	25	30
δ-QS	112.38	82.29	70.06	62.81	59.55	55.80	54.66	53.67	53.60
δ-TBM	89.94	65.92	56.49	53.92	52.14	50.57	49.99	50.34	49.69
δ-BR	164.51	120.44	98.35	84.06	75.90	62.74	56.23	52.55	50.81
δ-FFS	88.41	**63.94**	**54.83**	**52.07**	**50.56**	**48.34**	47.63	47.05	46.61
δ-FS	**88.33**	64.39	56.29	52.89	51.29	49.69	48.79	48.80	48.47
δ-BNDM	150.64	103.41	78.90	65.46	56.91	49.35	**46.95**	**46.40**	**45.50**

Rand60 problem with $\delta = 4$

$\sigma = 120, \delta = 1$	2	4	6	8	10	15	20	25	30
δ-QS	67.32	52.41	48.78	46.86	46.42	45.28	44.31	44.61	43.92
δ-TBM	57.44	47.79	46.17	45.60	45.85	45.21	45.72	44.51	44.28
δ-BR	114.24	85.12	70.86	62.88	56.61	49.86	46.96	47.11	46.31
δ-FFS	57.62	**47.74**	**45.39**	45.16	45.67	**44.56**	45.06	44.32	**43.89**
δ-FS	**57.15**	47.86	45.96	**44.82**	**45.06**	44.78	**43.20**	**44.13**	43.95
δ-BNDM	91.98	62.92	53.25	49.97	48.40	45.89	46.07	44.85	44.68

Rand120 problem with $\delta = 1$

$\sigma = 120, \delta = 2$	2	4	6	8	10	15	20	25	30
δ-QS	72.31	55.66	50.43	48.24	47.56	46.54	45.39	43.60	44.89
δ-TBM	60.73	48.71	46.57	45.60	45.82	44.80	45.39	46.17	44.93
δ-BR	119.57	89.09	73.42	64.91	58.82	50.80	47.82	46.08	45.98
δ-FFS	60.76	48.53	46.17	45.57	45.65	45.46	**44.60**	45.71	44.71
δ-FS	**60.30**	**48.25**	**45.89**	**45.06**	**44.91**	**44.69**	44.86	**42.88**	**43.89**
δ-BNDM	99.36	69.31	58.94	54.62	51.12	47.22	45.74	46.49	44.49

Rand120 problem with $\delta = 2$

$\sigma = 120, \delta = 4$	2	4	6	8	10	15	20	25	30
δ-QS	82.92	62.78	54.59	50.91	49.22	47.32	46.66	46.28	46.06
δ-TBM	69.13	51.62	48.01	47.07	46.36	45.63	45.76	45.30	45.28
δ-BR	132.63	97.17	80.17	69.71	62.69	53.69	49.81	48.12	47.18
δ-FFS	67.86	51.45	47.40	46.29	46.53	45.64	45.14	44.88	44.67
δ-FS	**67.61**	**50.91**	**47.32**	**45.92**	**45.60**	**45.43**	**44.99**	44.95	44.78
δ-BNDM	114.37	82.24	69.07	60.88	54.61	47.62	45.96	**44.79**	**44.42**

Rand120 problem with $\delta = 4$

$\sigma = 55, \delta = 1$	2	4	6	8	10	15	20	25	30
δ-QS	11.62	8.30	7.54	7.17	6.66	5.68	5.95	5.59	5.36
δ-TBM	9.40	7.33	6.47	5.82	5.62	5.51	5.29	5.51	5.83
δ-BR	16.16	12.56	10.10	8.79	7.64	6.97	6.11	5.87	5.44
δ-FFS	**8.76**	**6.58**	6.44	6.22	5.61	5.29	5.26	5.19	5.18
δ-FS	9.16	6.72	**5.79**	**5.51**	**5.59**	**5.06**	**5.19**	**5.11**	5.26
δ-BNDM	15.17	10.33	8.08	6.97	6.54	6.08	5.47	**5.11**	**5.01**

Results on the Real Music problem with $\delta = 1$

$\sigma = 55, \delta = 2$	2	4	6	8	10	15	20	25	30
δ-QS	14.53	11.41	9.79	9.51	8.63	8.03	7.61	7.68	7.20
δ-TBM	11.55	9.39	8.42	8.39	7.86	7.17	7.28	6.78	6.93
δ-BR	20.36	16.21	13.20	12.26	10.68	9.00	8.39	7.42	7.14
δ-FFS	11.35	**8.78**	**7.21**	**6.64**	6.73	**5.34**	5.43	6.10	5.89
δ-FS	**11.23**	8.97	7.78	7.74	7.78	6.88	6.60	6.38	6.34
δ-BNDM	18.36	12.85	9.72	8.06	**6.71**	6.19	**5.09**	**5.24**	**5.42**

Results on the Real Music problem with $\delta = 2$

$\sigma = 55, \delta = 4$	2	4	6	8	10	15	20	25	30
δ-QS	17.64	16.70	15.79	14.99	14.63	13.69	12.90	13.39	12.83
δ-TBM	15.18	15.19	14.30	13.80	13.02	12.54	11.76	11.92	11.54
δ-BR	26.12	23.17	21.36	19.73	18.33	15.60	14.12	14.22	13.14
δ-FFS	**14.02**	**13.28**	**12.23**	**11.33**	10.32	9.84	9.57	9.18	9.26
δ-FS	14.52	14.20	13.37	12.43	12.18	11.46	10.42	10.39	9.83
δ-BNDM	21.92	17.03	14.60	11.86	**9.80**	**7.79**	**6.75**	**5.85**	**5.60**

Results on the Real Music problem with $\delta = 4$

7 Conclusion

As reported in [CIL$^+$02], typical problems arising in musical analysis and musical information retrieval generally use representations of musical scores requiring large alphabets. In such problems the length of the pattern is generally short (10-20 notes): thus the need of approximate searching algorithms that perform well for small patterns and large alphabets.

In this paper we have focused our attention on δ-approximate string matching algorithms, which are very effective in searching for all similar but not necessarily identical occurrences of given melodies in musical scores. In particular we have presented two new efficient algorithms, δ-Fast-Search and δ-Forward-Fast-Search, which outperform known algorithms in the case of small patterns and large alphabets.

References

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[BR99] T. Berry and S. Ravindran. A fast string matching algorithm and experimental results. In J. Holub and M. Šimánek, editors, *Proceedings of the Prague Stringology Club Workshop '99*, pages 16–28, Czech Technical University, Prague, Czech Republic, 1999. Collaborative Report DC–99–05.

[BYG89] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. In N. J. Belkin and C. J. van Rijsbergen, editors, *Proceedings of the 12th International Conference on Research and Development in Information Retrieval*, pages 168–175, Cambridge, MA, 1989. ACM Press.

[CCG+94] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.

[CCI+99] E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon. Algorithms for computing approximate repetitions in musical sequences. In R. Raman and J. Simpson, editors, *Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms*, pages 129–144, Pert, WA, Australia, 1999.

[CF03a] D. Cantone and S. Faro. Fast-Search: A new efficient variant of the Boyer-Moore string matching algorithm. In K. Jansen, M. Margraf, M. Mastrolli, and J.D.P. Rolim, editors, *Proceedings of the 9th Workshop on Experimental Algorithms (WEA 2003)*, volume 2647 of *Lecture Notes in Computer Science*, pages 47–58. Springer-Verlag, Berlin, 2003.

[CF03b] D. Cantone and S. Faro. Forward-Fast-Search: Another fast variant of the Boyer-Moore string matching algorithm. In M. Šimánek, editor, *Proceedings of the Prague Stringology Conference '03*, pages 10–24, Czech Technical University, Prague, Czech Republic, 2003.

[CIL+02] M. Crochemore, C. S. Iliopoulos, T. Lecroq, W. Plandowski, and W. Rytter. Three heuristics for δ-matching: δ-BM algorithms. In A. Apostolico and M. Takeda, editors, *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, number 2373 in Lecture Notes in Computer Science, pages 178–189, Fukuoka, Japan, 2002. Springer-Verlag, Berlin.

[CILP01] M. Crochemore, C. S. Iliopoulos, T. Lecroq, and Y. J. Pinzon. Approximate string matching in musical sequences. In M. Balík and M. Šimánek, editors, *Proceedings of the Prague Stringology Conference '01*, pages 26–36, Prague, Czech Republic, 2001. Annual Report DC–2001–06.

[CIPN03] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and G. Navarro. A bit-parallel suffix automaton approach for (δ, γ)-matching in music retrieval. In Edleno S. De Moura and A. L. Oliveira, editors, *Proc. of the 10th International Symposium on String Processing and Information Retrieval (SPIRE'03)*, number 2857 in lncs, pages 211–223. Svb, 2003.

[CIR98] T. Crawford, C. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, 11:71–100, 1998.

[CLP98] C. Charras, T. Lecroq, and J.D. Pehoushek. A very fast string matching algorithm for small alphabets and long patterns. In M. Farach-Colton, editor, *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, number 1448 in Lecture Notes in Computer Science, pages 55–64. Springer-Verlag, Berlin, 1998.

[HS91] A. Hume and D. M. Sunday. Fast string searching. *Softw. Pract. Exp.*, 21(11):1221–1248, 1991.

[KMGL88] S. Karlin, M. Morris, G. Ghandour, and M. Y. Leung. Efficient algorithms for molecular sequence analysis. *Proceedings of the National Academy of Science*, 85:841–845, 1988.

[KPR00] J. Karhumäki, W. Plandowski, and W. Rytter. Pattern-matching problems for two-dimensional images described by finite automata. *Nordic J. Comput.*, 7(1):1–13, 2000.

[Lec00] T. Lecroq. New experimental results on exact string-matching. Rapport LIFAR 2000.03, Université de Rouen, France, 2000.

[MJ93] A. Milosavljevic and J. Jurka. Discovering simple DNA sequences by the algorithmic significance method. *Comp. Appl. BioSci.*, 9(4):407–411, 1993.

[NR98] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. Technical Report TR/DC–98–1, Department of Computer Science, University of Chile, 1998.

[Sun90] D. M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, 1990.

[Yao79] A. C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.

A Simple Lossless Compression Heuristic for Grey Scale Images

L. Cinque[1], S. De Agostino[1], F. Liberati[1] and B. Westgeest[2]

[1] Computer Science Department
University "La Sapienza"
Via Salaria 113, 00198 Rome, Italy
e-mail: deagostino@di.uniroma1.it

[2] Computer Science Department
Armstrong Atlantic State University
Savannah, Georgia 31419, USA

Abstract. In this paper, we show a simple lossless compression heuristic for gray scale images. The main advantage of this approach is that it provides a highly parallelizable compressor and decompressor. In fact, it can be applied independently to each block of 8x8 pixels, achieving 80 percent of the compression obtained with LOCO-I (JPEG-LS), the current lossless standard in low-complexity applications. The compressed form of each block employs a header and a fixed length code, and the sequential implementations of the encoder and decoder are 50 to 60 percent faster than LOCO-I.

Keywords: grey scale image, lossless compression, differential coding, parallelization

1 Introduction

Lossless image compression is often realized by extending string compression methods to two-dimensional data. Standard lossless image compression methods extend model driven text compression [1], consisting of two distinct and independent phases: *modeling* [13] and *coding* [12]. In the coding phase, arithmetic encoders enable the best model driven compressors both for bi-level images (JBIG [7]) and for grey scale and color images (CALIC [18]), but they are often ruled out because too complex. The compression gap between simpler techniques and state of the art compressors can be significant. Storer [15] and Storer and Helfgott [16] extended dictionary text compression [14] to bi-level images to avoid arithmetic encoders, achieving 70 percent of the compression of JBIG1 on the CCITT bi-level image test set. Such method is suitable for high speed applications by means of a simple hashing scheme. A polylogarithmic time work-optimal parallel implementation of this method was also presented to further speed up the computation on the PRAM EREW [3, 4]. Such implementation requires more sophisticated architectures (pyramids or meshes of trees) [9] than a simple array of processors to be executed on a distributed memory system. The extension of this method to grey scale and color images was left as an open problem, but it seems not feasible since the high cardinality of the alphabet causes an unpractical exponential blow-up of the hash table used in the implementation.

With grey scale and color images, the modeling phase consists of three components: the determination of the context of the next pixel, the prediction of the next pixel and a probabilistic model for the *prediction residual*, which is the value difference between the actual pixel and the predicted one. In the coding phase, the prediction residuals are encoded. A first step toward a good low complexity compression scheme was FELICS [8], which involves Golomb-Rice codes [6, 10] rather than arithmetic ones . With the same complexity level for compression (but with a 10 percent slower decompressor) LOCO-I [17] attains significantly better compression than FELICS, only a few percentage points of CALIC. As explained in section 2, also polylogarithmic time parallel implementations of FELICS and LOCO would require more sophisticated architectures than a simple array of processors.

The use of prediction residuals for grey scale and color image compression relies on the fact that most of the times there are minimal variations of color in the neighborood of one pixel. Therefore, differently than for bi-level images we should be able to implement an extremely local procedure which is able to achieve a satisfying degree of compression by working independently on different very small blocks. In this paper, we show such procedure. We present the heuristic for grey scale images, but it could be applied to color images by working on the different components [2]. The main advantage of this approach is that it provides a highly parallelizable compressor and decompressor. In fact, it can be applied independently to each block of 8x8 pixels, achieving 80 percent of the compression obtained with LOCO-I (JPEG-LS), the current lossless standard in low-complexity applications. The compressed form of each block employs a header and a fixed length code, and the sequential implementations of the encoder and decoder are 50 to 60 percent faster than LOCO-I. Two different techniques might be applied to compress the block. One is the simple idea of reducing the alphabet size by looking at the values occurring in the block. The other one is to encode the difference between the pixel value and the smallest one in the block. Observe that this second technique can be interpreted in terms of the model driven method, where the block is the context, the smallest value is the prediction and the fixed length code encodes the prediction residual. More precisely, since the code is fixed length the method can be seen as a two-dimensional extension of differential coding [5]. Differential coding, often applied to multimedia data compression, transmits the difference between a given signal sample and another sample.

In section 2, we sketch how FELICS and LOCO-I work and discuss their parallel complexity. In section 3, we explain our heuristic. Conclusions are given in section 4.

2 FELICS and LOCO-I

In this section, we sketch how FELICS and LOCO-I work and discuss their parallel complexity. As explained in the introduction, these are context-based method.

In FELICS the context for the current pixel P is determined by the last two pixels N_1 and N_2 read in a raster scan. If P is in the range defined by N_1 and N_2, then it is encoded with an adjusted binary code. Otherwise, the value distance from the closer of values N_1 and N_2 is encoded using a Golomb-Rice code. Golomb-Rice codes use a positive integer parameter m to encode a non-negative integer n. Given the value m, $\lfloor n/m \rfloor$ is encoded in unary and $n \bmod m$ in binary. A typical method of computing the parameter for Golomb-Rice code is to divide the image in 8x8 pixel blocks and select in a set of reasonable values the best one for each block with an exaustive search [11]. This method is not used in [8] because a better exaustive search is proposed for a sequential implementation. That is, to mantain for each context a cumulative total of the code length we would have at the current step for each reasonable parameter and to pick the best one each time.

Observe that the coding process would be highly parallelizable with the parameter selection of [11], since for each pixel of a block the parameter selection is completely independent from the rest of the image. A distributed memory system as simple as an array of processors would be able to perform the algorithm. With the other method instead, the design of a highly parallelized procedure becomes a much more complex issue, involving prefix computation and more sophisticated architectures than a simple array of processors [9]. However, with both methods the decoding process is hardly parallelizable, since to decode the current pixel the knowledge of the two previous pixels is required.

LOCO-I employs a more involved context modeling procedure where the context of pixel P in position (i, j) is determined by the pixels in positions $(i, j-1)$, $(i, j-2)$, $(i-1, j-1)$, $(i-1, j)$ and $(i-1, j+1)$. A count of prediction residuals and an accumulated sum of magnitudes of prediction residuals seen so far is maintained in order to select the parameter for the Golomb-Rice code. Moreover, in order to improve the compression performance a count of context occurences and an accumulated sum of prediction residuals encoded so far for each context also is maintained. These computations are used to reduce the prediction error. Further improvement is obtained by switching the Golomb-Rice code to encode characters from an extended alphabet when long runs of a character from the original alphabet are detected. All these operations require prefix computation in a highly parallelized version of the algorithm.

In conclusion, FELICS and LOCO methods do not provide highly parallelizable encoders and decoders implementable on a simple array of processors. The heuristic we present in the next section works independently on each 8x8 block of pixels. Since no information is shared among the blocks, a simple array of processors suffices to realize a constant time work-optimal parallel implementation.

3 A Simple Heuristic for Grey Scale Images

As previously mentioned, the heuristic applies independently to blocks of 8x8 pixels of the image. We can assume the heuristic reads the image with a raster scan on each block. The heuristic apply at most three different ways of compressing the block and chooses the best one. The first one is the following.

The smallest pixel value is computed on the block. The header consists of three fields of 1 bit, 3 bits and 8 bits repectively. The first bit is set to 1 to indicate that we compress a block of 64 pixels. This is because one of the three methods will partition the block in four sub-blocks of 16 pixels each and compress each of these smaller areas. The 3-bits field stores the minimum number of bits required to encode in binary the distance between the smallest pixel value and every other pixel value

255	255	255	254	254	110	110	110
255	255	255	254	254	110	110	110
255	255	255	254	254	110	110	110
255	255	255	254	254	110	110	110
255	255	254	128	127	128	129	130
255	253	253	128	128	129	130	131
254	253	252	129	129	130	131	132
253	252	251	130	130	130	254	255

Figure 1: An 8x8 pixel block of a grey scale image.

in the block. The 8-bits field stores the smallest pixel value. If the number of bits required to encode the distance, say k, is at most 5, then a code of fixed length k is used to encode the 64 pixels, by giving the difference between the pixel value and the smallest one in the block. To speed up the procedure, if k is less or equal to 2 the other techniques are not tried because we reach a satisfying compression ratio on the block. Otherwise, two more techniques are experimented on the block.

One technique is to detect all the different pixel values in the 8x8 block and create a reduced alphabet. Then, encode each pixel in the block using a fixed length code for this alphabet. The employment of this technique is declared by setting the 1-bit field to 1 and the 3-bits field to 110. Then, an additional three bits field stores the reduced alphabet size d with an adjusted binary code in the range $2 \leq d \leq 9$. The last componenent of the header is the alphabet itself, a concatenation of d bytes. Then, a code of fixed length $\lceil \log d \rceil$ bits is used to encode the 64 pixels.

The other technique compresses the four 4x4 pixel sub-blocks. The 1-bit field is set to 0. Four fields follow the flag bit, one for each 4x4 block. The two previous techniques are applied to the blocks and the best one is chosen. If the first technique is applied to a block, the corresponding field stores values from 0 to 7 rather than from 0 to 5 as for the 8x8 block. If such value is in between 0 and 6, the field stores three bits. Otherwise, the three bits (111) are followed by three more. This is because 111 is used to denote the application of the second technique to the block as well, which is less frequent to happen. In this case, the reduced alphabet size stored in this three additional bits has range from 2 to 7, it is encoded with an adjusted binary code from 000 to 101 and the alphabet follows. 110 denotes the application of the first technique with distances expressed in seven bits and 111 denotes that the block is not compressed. After the four fields, the compressed forms of the blocks follow, which are similar to the ones described for the 8x8 block. When the 8x8 block is not compressed, 111 follows the flag bit set to 1.

Image	OURS	LOCO
1	1.22	1.52
2	1.57	2.00
3	1.75	2.31
4	1.52	1.93
5	1.22	1.55
6	1.39	1.75
7	1.57	2.22
8	1.19	1.51
9	1.60	2.05
10	1.56	2.04
11	1.43	1.83
12	1.63	2.10
13	1.15	1.34
14	1.30	1.63
15	1.67	2.07
16	1.51	1.97
17	1.51	1.96
18	1.30	1.58
19	1.41	1.80
20	2.00	2.55
21	1.45	1.77
22	1.41	1.76
23	1.75	2.29
24	1.39	1.74
Avg	1.48	1.89

Figure 2: Compression ratios (uncompressed / compressed).

We now show how the heuristic works on the example of Figure 1.

Since the difference between 110, the smallest pixel value, and 255 requires a code with fixed length 8 and the number of different values in the 8x8 block is 12, the technique employed to compress the block is to work separately on the 4x4 sub-blocks. Each block will be encoded with a raster scan (row by row). The upper left block has 254 as smallest pixel value and 255 is the only other value. Therefore, after setting the 1-bit field to zero the corresponding field is set to 001. The compressed form after the header is 1110111011101110. The reduced alphabet technique is more expensive since the raw pixel values must be given. On the hand, the upper right block needs the reduced alphabet technique. In fact, one byte is required to express the difference between 110 and 254. Therefore, the corresponding field is set to 111000, which indicates that the reduced alphabet size is 2, and the sequence of two bytes 0110111011111110 follows. The compressed form after the header is 1000100010001000. The lower left block has 8 different values so we do not use the reduced alphabet technique since the alphabet size should be between 2 and 7. The smallest pixel value in the block is 128 and the largest difference is 127 with the pixel value 255. Since a code of fixed length 7 is required, the corresponding field is 111110. The compressed form after the header is (we introduce a space between pixel encodings in the text to make it more readable): 1111111 1111111 1111110 0000000 1111111 1111101 1111101 0000000 1111110 1111101 1111100 0000001 1111101 1111100 1111011 0000010. Observe that

Image	OURS	LOCO
1	33.8	64.7
2	32.9	60.5
3	32.2	59.5
4	33.0	60.6
5	34.1	67.4
6	32.2	60.7
7	34.9	60.1
8	34.7	67.0
9	32.7	58.6
10	35.3	60.0
11	33.2	62.5
12	34.4	60.3
13	34.0	68.2
14	33.9	64.5
15	32.3	60.5
16	33.3	60.5
17	33.3	60.0
18	33.4	64.9
19	33.2	61.9
20	25.5	48.1
21	32.8	61.1
22	33.4	62.8
23	33.2	56.6
24	33.2	63.8
Avg.	33.1	61.4

Figure 3: Compression times (ms.).

the compression of the block would have been the same if we had allowed the reduced alphabet size to grow up to 8. However, experimentally we found more advantageous to exclude this case in favor of the other technique. Our heuristic does not compress the lower right block since it has 8 different values and the difference between pixel values 127 and 255 requires 8 bits. Therefore, the corresponding field is 111111 and the uncompressed block follows.

We experimented this technique on the kodak image test set, which is an extension of the standard jpeg image test set. We reached 70 to 85 percent of LOCO-I compression ratio (Figure 2) (78 percent in average). The executions of our algorithm and LOCO were compared with a Intel Pentium 4, 2.00 GHz processor on a RedHat Linux platform. Our compression heuristic turned out to be about 50 percent faster (Figure 3). When we compared the decompression times, we obtain an even greater speedup (around 60 percent) in comparison with LOCO (Figure 4). This is not surprising since, as we mentioned in the introduction, while the LOCO compressor is more or less as fast as FELICS, the decompressor is 10 percent slower.

Conclusions

In this paper, we showed a simple lossless compression heuristic for grey scale images. The main advantage of this approach is that it provides a highly parallelizable compressor and decompressor since it can be applied independently to each block of 8x8

Image	OURS	LOCO
1	30.8	69.7
2	26.3	65.3
3	24.4	64.2
4	26.0	65.2
5	31.2	69.2
6	33.8	64.9
7	25.7	65.1
8	32.1	69.9
9	26.4	64.2
10	25.4	64.7
11	26.9	66.6
12	26.1	63.8
13	32.6	70.5
14	29.6	67.7
15	23.6	68.4
16	26.1	64.6
17	25.7	66.0
18	28.7	68.8
19	27.0	66.1
20	20.1	52.2
21	26.3	65.3
22	27.0	67.0
23	22.9	62.9
24	27.3	67.7
Avg.	27.2	65.8

Figure 4: Decompression times (ms.).

pixels. The compressed form of each block employs a header and a fixed length code. Two different techniques might be applied to compress the block. One is the simple idea of reducing the alphabet size by looking at the values occurring in the block. The other one is to encode the difference between the pixel value and the smallest one in the block. It was interesting to see that this technique achieves about 80 percent of the compression performance of LOCO-I and the compressor and decompressor are 50 to 60 percent faster. Also, our technique is definitely the easiest to implement and can be applied as well to color images.

References

[1] Bell T.C., Cleary J.G. and Witten I.H [1990]. *Text Compression*, Prentice Hall.

[2] Cinque L., De Agostino S. and Liberati F. [2004]. "A Simple Lossless Compression Heuristic for RGB Images", *IEEE Data Compression Conference*, 533.

[3] De Agostino S. [2002]. "A Work-Optimal Parallel Implementation of Lossless Image Compression by String Matching", *Proceedings Prague Stringology Club Conference*, 1-8.

[4] Cinque L., De Agostino S. and Liberati F. [2003]. "A Work-Optimal Parallel Implementation of Lossless Image Compression by String Matching", *Nordic Journal of Computing*, **10**, 13-20.

[5] Gibson J. D. [1980]. "Adaptive prediction in speech differential encoding system", *Proceedings of the IEEE*, **68**, 488-525.

[6] Golomb S. W. [1966]. "Run-Length Encodings", *IEEE Transactions on Information Theory* **12**, 399-401.

[7] Howard P. G., Kossentini F., Martinis B., Forchammer S., Rucklidge W. J. and Ono F. [1998]. "The Emerging JBIG2 Standard", *IEEE Transactions on Circuits and Systems for Video Technology*, **8**, 838-848.

[8] Howard P. G. and Vitter J. S. [1993]. "Fast and Efficient Lossles Image Compression", *IEEE Data Compression Conference*, 351-360.

[9] Leighton F. T. [1992]. *Introduction to Parallel Algorithms and Architectures*, Morgan-Kaufmann.

[10] Rice R. F. [1979]. "Some Practical Universal Noiseless Coding Technique - part I", *Technical Report JPL-79-22*, Jet Propulsion Laboratory, Pasadena, California, USA.

[11] Rice R. F. [1991]. "Some Practical Universal Noiseless Coding Technique - part III", *Technical Report JPL-91-3*, Jet Propulsion Laboratory, Pasadena, California, USA.

[12] Rissanen J. [1976]. "Generalized Kraft Inequality and Arithmetic Coding", *IBM Journal on Research and Development* **20**, 198-203.

[13] Rissanen J. and Langdon G. G. [1981]. "Universal Modeling and Coding", *IEEE Transactions on Information Theory* **27**, 12-23.

[14] Storer J.A. [1988]. *Data Compression: Methods and Theory* (Computer Science Press).

[15] Storer J. A.[1996] "Lossless Image Compression using Generalized LZ1-Type Methods", *IEEE Data Compression Conference*, 290-299.

[16] Storer J. A. and Helfgott H. [1997] "Lossless Image Compression by Block Matching", *The Computer Journal* **40**, 137-145.

[17] Wimberger M. J., Seroussi G and Sapiro G. [1996] "LOCO-I: A Low Complexity, Context Based, Lossless Image Compression Algorithm", *IEEE Data Compression Conference*, 140-149.

[18] Wu X. and Memon N. D. [1997] "Context-Based, Adaptive, Lossless Image Coding", *IEEE Transactions on Communications*, **45**, 437-444.

BDD-Based Analysis of Gapped q-Gram Filters*

Marc Fontaine[1], Stefan Burkhardt[2] and Juha Kärkkäinen[2]

[1] Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany
e-mail: stburk@mpi-sb.mpg.de
e-mail: fontaine@studcs.uni-sb.de

[2] Department of Computer Science
P.O.Box 68 (Gustaf Hällströmin katu 2 B)
FI-00014 University of Helsinki, Finland
e-mail: Juha.Karkkainen@cs.helsinki.fi

Abstract. Recently, there has been a surge of interest in gapped q-gram filters for approximate string matching. Important design parameters for filters are for example the value of q, the filter-threshold and in particular the shape (aka seed) of the filter. A good choice of parameters can improve the performance of a q-gram filter by orders of magnitude and optimising these parameters is a nontrivial combinatorial problem. We describe a new method for analysing gapped q-gram filters. This method is simple and generic. It applies to a variety of filters, overcomes many restrictions that are present in existing algorithms and can easily be extended to new filter variants. To implement our approach, we use an extended version of BDDs (Binary Decision Diagrams), a data structure that efficiently represents sets of bit-strings. In a second step, we define a new class of multi-shape filters and analyse these filters with the BDD-based approach. Experiments show that multi-shape filters can outperform the best single-shape filters, which are currently in use, in many aspects. The BDD-based algorithm is crucial for the design and analysis of these new and better multi-shape filters. Our results apply to the k-mismatches problem, i.e. approximate string matching with Hamming distance.

1 Introduction

String matching involves searching a given string or textual database T for occurrences of substrings that match a search pattern P. The approximate string matching problem allows the search pattern and the matches to have some difference or distance according to a given distance function.

Many applications depend on efficient solutions of this problem, especially in the field of bio-informatics, where databases may consist of sequences of 10^9 nucleotides of DNA or of long sequences of amino acids.

*This work was conducted in part at the MPI for computer science, Saarbrücken with support from the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and at the University of Helsinki supported by the Academy of Finland grant 201560.

Filter algorithms are a common approach for approximate string matching. They speedup string matching by quickly generating a set of potential matches and discarding the rest of the database. The true matches can then be found in a second step, the verification phase, by inspecting all potential matches. Designing a good filter usually means optimising the tradeoff between the complexity of the filtration phase and the efficiency of the filter.

Many efficient filters work with precomputed indexes, in particular, indexes that are based on *gapped q-grams* or *shapes*. For example the three 3-grams of string `ACAGCT` for shape `##-#` are `AC-G`,`CA-C` and `AG-T`. A matching pair of q-grams between a pattern and a substring of T is called a *hit*. The q-gram index stores the positions of all q-grams of the database and allows to find hits efficiently. If the number of hits between the search pattern and a substring of the database exceeds a certain *threshold* t, that substring is called a *potential match*.

As first shown in [6, 7], the performance of the filter depends crucially on the shape. Good shapes are found by analysing large sets of shapes, because no method for directly generating good shapes has been found yet. Even analysing a single shape is non-trivial and a lot of effort has gone into developing methods for this purpose. Recently gapped shapes have been the focus of quite a bit of attention [16, 9, 10, 12].

In [6, 7], Burkhardt and Kärkkäinen compute the *optimal threshold*. It is the highest threshold that still allows the filter to return all true matches, i.e., substrings that are within a fixed Hamming distance from the pattern. They also compute a measure called the minimum coverage, which provides a rough estimate of how many false matches get through the filter. The true positive rates and the false positive rates were determined experimentally for selected shapes.

If the threshold or Hamming distance is increased, the filter also discards some true matches, which are then called *false negatives*. In [5] an abstract measure for the false negative probability, the so-called *recognition rate* was defined and analysed experimentally. The exact computation of both false positive and false negative rates was done by Ma, Tromp and Li [15]. However, their algorithms are restricted to filters that count only non-overlapping hits. This is a significant restriction as the lower correlation of overlapping q-grams is a big advantage of gapped q-grams over ungapped ones [7].

Brejová, Brown and Vinař [1, 2] develop new variants of the algorithm of Ma, Tromp and Li. In [1], a true match is defined not directly by a Hamming distance but as a probability distribution represented by a hidden Markov model. In [2], they further generalise the approach to approximate hits and multiple shapes. However, the restriction to non-overlapping hits remains in all of their work. Very recently, we became aware of several papers using multiple gapped shapes for approximate string matching in various different approaches [13, 17, 18, 14].

We present a new and flexible method for computing various properties of q-gram filters. This algorithm is based on a simple natural abstraction of the problem and applies to a general class of filters. At the same time, it overcomes many restrictions present in previous algorithms in particular the non-overlapping hit restriction.

Our method consists of two steps. The first step is an algorithm based on sets of bit-strings. These sets can be of exponential size and we have to use a compact and efficient representation of the sets to actually implement our algorithm. A data structure called BDDs [3, 4] (Binary decision diagrams or Binary decomposition diagrams) implements such a representation of sets. Only the use of a data structure like BDDs makes it feasible to run our algorithm.

In the second step the BDDs generated in the first step can be used to efficiently compute interesting properties of the sets they represent. Most properties can be computed in linear time of the size of the BDDs. This clear split of the problem into two steps distinguishes our method from previous algorithms, which were mostly based on dynamic programming.

In the second part of this work, we apply our method to design new and better filters. The basic idea in this part is to filter with a set of shapes simultaneously. Multi-shape filters have been used before [8]. Our new idea is to use a carefully selected set of shapes together with a specifically computed *filtration criterion*. This filtration criterion replaces the optimal threshold of a single shape filter.

The BDD-based algorithm allows us to compute the best filtration criterion for a set of shapes and at the same time determine the important quality measures of the resulting multi-shape filter. To investigate the potential of multi-shape filters based on a specific filtration criterion, we analyse large sets of randomly generated filters. These experiments show that good multi-shape filters are very rare, but the experiments also yield filters that are superior to single-shape filters in several important aspects.

2 Representing Match-Mismatch-Patterns with BDDs

Let A and B be two strings of length l. We call the bit-string $p(A,B) \in \{0,1\}^l$ the match-mismatch-pattern of the two strings. A "1" in $p(A,B)$ denotes a matching position and "0" a mismatch. The Hamming distance of A and B is then the number of zeros in $p(A,B)$. We represent the number of zeros and ones in a bit-string p by $|p|_0$ and $|p|_1$. The *k-differences* version of approximate string matching allows a pattern string and a match to have a Hamming distance of at most k.

For most filters, the match-mismatch-pattern of an alignment between the pattern string and the database at some position x contains enough information to decide whether x is returned as a potential match or not. Therefore it is, in principle, sufficient to enumerate all possible match-mismatch-patterns to analyse the performance of filters for the *k-differences* problem.

A drawback of this brute-force-approach is, that there are 2^l possible match-mismatch-patterns for strings of length l and realistic filters usually work with pattern length $l \geq 50$.

To overcome this complexity-problem we use a data structure called BDDs. BDDs allow a compact and efficient representation of sets of equal-length bit-strings. They can be seen as an abstract data structure that supports the following operations.

- Creation of a new BDD for the base-cases $\emptyset$ and $\{\epsilon\}$.
- Composition of two BDDs $S = comp(S_0, S_1)$
- Decomposition of a BDD into two BDDs according to the first position of the bit-strings in the BDD.

- Computing $\cup$ and $\cap$ of two BDDs and the complement $\neg$ of a BDD

The composition $comp(S_0, S_1)$ represents the set:

$$comp(S_0, S_1) = \{s \mid (s = 0a \wedge a \in S_0) \vee (s = 1b \wedge b \in S_1)\}$$

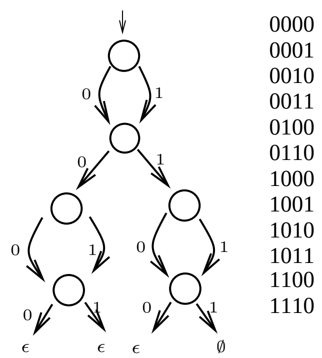
For two sets A and B that are given as decompositions $A = comp(A_0, A_1)$ and $B = comp(B_0, B_1)$, $A \cup B$ and $A \cap B$ can be computed recursively as:

$$A \cup B = comp(A_0 \cup B_0, A_1 \cup B_1)$$

and:

$$A \cap B = comp(A_0 \cap B_0, A_1 \cap B_1)$$

BDDs are implemented with DAGs (Directed Acyclic Graphs) and they are similar to finite automata without loops. In a BDD always the minimal, smallest possible DAG is used to represent a set of bit-strings and equal sets are represented by one canonical node of the DAG. A collection of BDDs can share the structure of a single DAG and BDD-implementations usually make use of hash-tables to maintain the canonical-representation-property. Hash tables are also used to avoid re-computations during the computation of $\cup$ and $\cap$.



A simple BDD and the set it represents

BDDs have many different applications where they often make it possible to handle exponential size sets within non-exponential complexity. An introduction to BDDs can be found in [3, 4], where BDDs are used to represent boolean functions over a finite set of variables. The actual performance of BDDs in an application depends on the structure of the sets they represent. For sets of size 2^l the space complexity of BDDs can range from $O(l)$ to $\Theta(2^l)$.

It is important to note, that we use BBDs only to analyse q-gram filters. This is a one-time computation and the complexity of the BDDs does not interfere with the complexity of the filters under consideration. The theoretical complexity of BDDs is therefore secondary in our application. Our experience is, that BDDs work well to reduce the complexity of filter analysis. They make is possible to analyse all interesting filters within reasonable time and space limits.

In [11] Fontaine described a extension of standard BDDs, which uses $\{0,1\}^*$ as an additional base case for the decomposition (so called $*$-BDDs). This extension allows more compact representation than standard BDDs and it was used for all our experiments. For code listings and runtime measurements of a prototype $*$-BDD implementation see [11]. The prototype implementation only consists of about 10kb of C++ code and computing the filter properties for a typical shape only takes a few seconds.

3 q-Gram Similarity-Based Filters

We use strings from $\{\texttt{\#}, \texttt{-}\}^*$ to denote different shapes. `#` stands for a position that must match, whereas `-` is a "don't care" or wild-card position. $span(s) = |s|$ is the span of a shape s.

Let A and B be two strings of length l and s a shape. A position $0 \leq i \leq l - span(s)$ is a hit of shape s if $\forall 0 \leq n < span(s) : s(n) = \texttt{\#} \Rightarrow A(i+n) = B(i+n)$.

The number of different hits of a shape s for two strings A and B is called the q-gram similarity $qgs_s(A, B)$ of the two strings. For strings of length l the q-gram-similarity can be at most $l - span(s) + 1$.

A=	`ACTGTACTGCCGTACT`
B=	`ACTGTAATGCAGTACT`
$p(A, B)$=	1111110111011111
shape s=	`###---##--##`
$qgs_s(A, B)$=	2

```
ACTGTACTGCCGTACT
###---?#--?#
 ###---##--##  <- hit
  ###---##--## <- hit
   ###---#?--##
    ##?---?#--##
ACTGTAATGCAGTACT
```

Match-mismatch-pattern and q-gram-similarity

A q-gram filter computes the set of potential matches with the help of a threshold t. A potential match is the position of a substring in the database with a q-gram similarity of at least t with the pattern string. Increasing the threshold of a filter reduces the number of potential matches at the cost of a decreased filter sensitivity, i.e. the filter is more likely to overlook true matches.

The match-mismatch-pattern of two strings contains sufficient information to compute their q-gram-similarity. Therefore a filter can be analysed by looking at all possible match-mismatch-patterns. We can partition the set of all possible match-mismatch-patterns according to the q-gram-similarity they represent for a given shape.

For any fixed shape s and any $h, l \in \mathbb{N}_0$ we define:

$$P_l^h = \{p \in \{0,1\}^l \mid s \text{ produces exactly } h \text{ hits in } p\}$$

It follows that the set PM of match-mismatch-pattern that represent a *potential match* is:

$$PM = \bigcup_{h \geq t} P_l^h$$

A set P_l^h can easily be computed based on the sets P_{l-1}^{h-1} and P_{l-1}^h. A match-mismatch-pattern $p \in \{0,1\}^l$ is in P_l^h if:
either: its suffix of length $l-1$ is in P_{l-1}^{h-1} and it has an additional hit of shape s at position 0
or: its suffix of length $l-1$ is in P_{l-1}^h and it does not have an additional hit at position 0.

This algorithm can be formulated as a simple equation for sets

$$P_l^h = (expand(P_{l-1}^{h-1}) \cap S_l(s)) \cup (expand(P_{l-1}^h) \cap \bar{S}_l(s))$$

with the following three definitions:

$$S_l(s) = \{p \in \{0,1\}^l \mid s \text{ has a hit in } p \text{ at position } 0\}$$

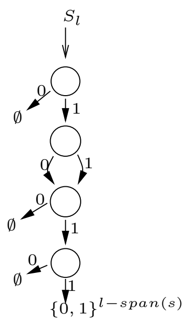
$$\bar{S}_l(s) = \{p \in \{0,1\}^l \mid s \text{ does not have a hit in } p \text{ at position } 0\}$$

$$\mathit{expand}(M) = \{x \mid x = 0m \vee x = 1m, m \in M\}$$

BDDs directly support $\cup$ and $\cap$, and $\mathit{expand}(M)$ can be implemented as $\mathit{expand}(M) = \mathit{comp}(M, M)$. BDDs also support the creation of $S_l(s)$ and $\bar{S}_l(s)$ for any shape s. $S_l(s)$ can be computed recursively as:

$$S_l(s) = \begin{cases} \{0,1\}^l & \text{if } s = \epsilon \\ \emptyset & \text{if } l < \mathit{span}(s) \\ \mathit{comp}(\emptyset, S_{l-1}(r)) & \text{if } s = \texttt{\#}r \\ \mathit{comp}(S_{l-1}(r), S_{l-1}(r)) & \text{if } s = \texttt{-}r \end{cases}$$

Shape=`#-##`
$P_4^1 = S_4 = \{1011, 1111\}$
$\mathit{expand}(P_4^1) = \{01011, 01111, 11011, 11111\}$
$P_5^1 = \{01011, 01111, 11011, 10110, 11110, 10111\}$
$P_5^2 = \{11111\}$
$P_5^0 = \{00000, 00001, 00010, \ldots\}$



P_l^h and $S_l(s)$ for shape `#-##`

As an alternative to our definition of the q-gram-similarity $qgs(A, B)$ it is possible to require individual hits to be non-overlapping [15, 1]. For such filters the set PM can be computed with an algorithm similar to the one described above. (Compute the sets $P_l^{(h,i)}$, where i is the offset of the first hit.)

4 Filter Analysis with BDDs

The algorithm described in the previous section allows us to generate BDD-representations for the sets P_l^h. These BDD-representations can be used to compute many interesting properties of the sets and thereby the underlying filters. Note that the computation of the various properties is independent of what filter the sets P_l^h represent and how they were computed. This is in contrast to previous approaches using dynamic programming where the filter definition is deeply involved in the property computation.

4.1 Specificity

The specificity of a filter describes its ability to reduce a large database to a small set of potential matches. For a given random model, the filter specificity is equivalent to the probability that a random substring of length l is a potential match of a random search pattern.

Every match-mismatch-pattern p describes one possible event that can occur while aligning a database and a search pattern and we can use several probability models to assign probabilities to these events. We can then simply extend these probabilities from one match-mismatch-pattern to sets of match-mismatch-patterns by summing up the probabilities of the elements of the sets.

For example, to analyse a filter for a DNA database, we might assume that the database and pattern string are independent random strings with an even distribution of the letters $\{A, C, G, T\}$. It follows that every single character has a $\frac{1}{4}$ chance of being a match and the probability of any match-mismatch-pattern p is:

$$prob(p) = (\frac{1}{4})^{|p|_1}(\frac{3}{4})^{|p|_0}$$

With this we can compute the probability of a potential match, i.e the specificity of the filters as:

$$specificity = \sum_{p \in PM} prob(p)$$

Given the binary decomposition $comp(P_0, P_1)$ of a set P the probability $Prob(P)$ of the set is:

$$Prob(P) = (\frac{3}{4}) * Prob(P_0) + (\frac{1}{4}) * Prob(P_1)$$

The base-cases for the binary decomposition are also the base-cases for this recursion:

$$Prob(\emptyset) = 0 \quad Prob(\epsilon) = 1$$

This shows that, if BDDs are used to represent the sets, $Prob(P) = \sum_{p \in P} prob(p)$ can be computed in linear time of the size of the BDDs.

It can be seen that a similar approach allows to compute the probabilities of sets for many different probability models efficiently. In particular it is also possible to use hidden Markov models (HMMs) as probability model. HMMs have been used in [1] to model real DNA sequences of different species.

4.2 Recognition Rate

For approximate string matching with Hamming distance we can define the recognition rate $r(j)$ of a filter as the expected fraction of potential matches among substrings of the database with *exactly* Hamming distance j. The match-mismatch-patterns of length l and Hamming distance j can easily be computed with the single-character shape `#` as $P_l^{l-j}(\texttt{#})$. It follows that a filter with potential matches PM has the recognition rate:

$$r(j) = \frac{Prob(PM \cap P_l^{l-j}(\texttt{#}))}{Prob(P_l^{l-j}(\texttt{#}))}$$

Recognition rates have been defined and determined experimentally in [5].

4.3 Threshold

The set of potential matches of a filter with shape s, and with it the recognition rates of the filter, heavily depends on the threshold t.

A filter is *lossless* for a threshold t and Hamming distance k if $\forall j \leq k : r(j) = 1$, otherwise it is lossy. If one is interested in a fixed maximal Hamming distance k and lossless filtering, then there exists an optimal threshold t_{best}. A dynamic programming algorithm for computing t_{best} is described in [7].

BDD-based threshold computation is also possible. For each set P_l^h we compute:

$$m(P) = \min_{p \in P} |p|_0$$

We use the notation $|p|_0$ for the number of occurrences of "0" in string p. $m(P_l^h)$ is the minimum number of mismatching positions of any match-mismatch-pattern $p \in P_l^h$. This minimum can be found in linear time in the size of the BDD. Any set P_l^h with $m(P_l^h) \leq k$ contains at least one match-mismatch-pattern with Hamming distance at most k. The optimal threshold t_{best} for a lossless filter is the smallest h such that $m(P_l^h) \leq k$.

shape $s =$ `#-#---#-#-#------#`
$span(s) = 18$
pattern length $l = 50$
number of hits $h \in \{0, \ldots, 33\}$

h	0	1	2	3	4	5	6	7	8	9	...	31	32	33
$m(P_l^h)$	8	7	7	6	6	6	5	5	5	4	...	1	1	0
$k = 7$		$t_{best} = 1$												
$k = 6$				$t_{best} = 3$										
$k = 5$							$t_{best} = 6$							
$k = 4$										$t_{best} = 9$				

Computing the threshold t_{best}

5 Multi-shape Filters

Shapes can be better than contiguous q-grams because they introduce irregularity in the way the mismatching positions affect the q-grams. For good shapes, only a few worst case configurations of the mismatching characters affect many q-grams. A reasonable approach to further improve the performance of filters is therefore to use two or more somehow *orthogonal* shapes in parallel. The idea is, that those configurations of mismatches, that are particularly bad for one shape, are better covered by a second shape and vice versa.

Designing a good multi-shape filter is a nontrivial combinatorial problem, just like finding good individual shapes. One could assume that the best individual shapes also form the best multi-shape filter, however our experiments suggest that this is often not the case.

Multi-shape filters are the most important application for our BDD-based approach. The extension of our algorithm to multi-shape filters is straight-forward and it leads to a new concept: the generic *filtration criterion* C. The generic filtration criterion C replaces the threshold t of a single-shape filter. It enables a multi-shape filter to make full use of the relations between the single shapes.

A filter with n shapes $s_1 \dots s_n$ can use the q-gram similarities $h_1 = qgs_{s_1}(M,P) \dots$ $h_n = qgs_{s_n}(M,P)$ to decide whether M is a potential match or not. (P is the pattern string and M is any substring of the database.) We call a set $C \subset \mathbb{N}^n$ a *filtration criterion* for the shapes $s_1 \dots s_n$ and define:

$$M \text{ is a potential match } \Leftrightarrow (h_1, \dots, h_n) \in C$$

This generic filtration criterion C can model many different strategies for multi-shape filters. For example it can model filters that require at least one hit of one shape, filters the require one hit of each shape, filters that sum up the hits of the shapes, or filters that use each shape with its individual threshold t_{best}.

In Section 3 we used the notation $P_l^h(s)$ for the set of all match-mismatch-patterns with exactly h hits of a single fixed shape s. To analyse multi-shape filters we extend this notation to sets of shapes $\{s_1, \dots, s_n\}$. We define $P_l^{(h_1,\dots,h_n)}(s_1, \dots, s_n)$ as the set of all match match-mismatch-patterns with exactly h_i hits of shape s_i $(1 \leq i \leq n)$. The sets $P_l^{(h_1,\dots,h_n)}(s_1, \dots, s_n)$ can be computed as:

$$P_l^{(h_1,\dots,h_n)}(s_1, \dots, s_n) = \bigcap_{1 \leq i \leq n} P_l^{h_i}(s_i)$$

With this, the set of match-mismatch-patterns, that represent a potential match according to a filtration criterion C is:

$$PM = \bigcup_{(h_1,\dots,h_n) \in C} P_l^{(h_1,\dots,h_n)}(s_1, \dots, s_n)$$

Together with the set PM, all statistical performance measures (recognition rate, specificity), which we computed for single-shape filters in Section 3, are now also available for our model of multi-shape filters.

The definition of $P_l^{(h_1,\dots,h_n)}(s_1, \dots, s_n)$ also makes it possible to compute a *optimal filtration criterion* C_{best} for a lossless filter with some fixed Hamming distance k. It is:

$$C_{best} = \{(h_1 \dots h_n) \mid m(P_l^{(h_1,\dots,h_n)}(s_1, \dots, s_n)) \leq k\}$$

C_{best} replaces the threshold t_{best} of single shape filters. To reduce the high complexity involved in the computation of C_{best} Fontaine [11] describes a straight forward approximation.

6 Designing Better Filters

The design of a filter is always a compromise between three objectives:

- high sensitivity
- fast filtration phase
- high specificity of the filter, i.e. a fast verification phase

There are several trade-offs between these objectives. For example, a higher sensitivity is usually at the cost of a lower specificity and a faster filtration often yields lower sensitivities and specificities [5, 15, 8].

Using a well chosen shape for the q-grams and the appropriate threshold can greatly improve overall filter performance compared to filtering with ungapped q-grams [6, 7]. In this section we will show that multi-shape filters with a carefully selected set of shapes and a specifically computed filtration criterion can further boost filter performance for all three objectives compared to single-shape filters.

A good estimate for the runtime of a q-gram filter is the number of hits in the database that have to be processed. It is roughly proportional to $|\Sigma|^{-q}$. (This assumes a database with a random distribution of letters from Σ and it is also a good estimate for example for DNA sequences [7].) High values of q are desirable because they make the filtration fast however they also mean lower sensitivities.

In this section we only consider q-gram filters that work *lossless* for a fixed Hamming distance k and we use k to compare the sensitivities of such filters (a higher value of k means a higher sensitivity). To compare the specificities of different filters, we always use the shapes with the optimal threshold t_{best} (the optimal filtration criterion C_{best} for multi-shape filters) that still guarantees lossless filtering for the fixed k. For all experiments in this section, we use a pattern length $l = 50$ and assume a DNA-like database with $|\Sigma| = 4$.

There is a trade-off between k and the highest value of q that can be used for lossless filtering. For example for pattern length $l = 50$ and $k = 5$ the highest possible q for a lossless single-shape filter is $q = 10$, for $k = 6$ it is $q = 9$. Similar constraints between q and k also exist for multi-shape filters. However we found that they can have higher values of *both q and k* than is possible for single shapes. Therefore multi-shape filters make it possible to increase q, which makes them faster, or increase k, i.e the sensitivity. In some cases it is even possible to increase q and k at the same time. This is not at the cost of a lower specificity, but instead it is even possible to increase the specificity also.

Pairs of shapes: $q = 10$**,** $k = 6$

Consider for example the following three lossless two-shape filters for $k = 6$:

Three good two-shapes filters			
$k = 6$, $l = 50$,$\|\Sigma\| = 4$			
	s_1	s_2	$specificity$
a)	`##-##---##-####`	`###-#-###----#--##`	$8.091782 * 10^{-8}$
b)	`#-##-###-####`	`####----###--##-#`	$9.306443 * 10^{-8}$
c)	`#-##-##--#####`	`####--#--#---##--##`	$7.763605 * 10^{-8}$

	$\neg C_{best}$
a) and c)	$\{(0,0),(0,1),(1,0),(1,1),(2,0)\}$ not $(0,2)$!
b)	$\{(0,0),(0,1),(0,2),(1,0),(1,1),(2,0)\}$

The best single shape filter for this problem is `######-#-##` with $q = 9$, $t_{best} = 2$ and a specificity of $3.838350 * 10^{-6}$. Compared to this single shape filter, each of these three multi-shape filters with two ($q = 10$)-shapes improves the specificity by a factor of 50. The runtime of a multi-shape filter is approximately the sum of the run-times computed for its individual shapes. This means that the filters with two ($q = 10$)-shapes for $|\Sigma| = 4$ are also about two times as fast as a $q = 9$ single-shape filter.

The three multi-shape filters of this example were found by scanning 5000 pairs of random ($q = 10$)-shapes. In this sample set, good pairs were extremely rare. 3439 of the pairs, i.e. more than two-thirds, did not yield a lossless filter for $k = 6$ at all. It is interesting that none of the six shapes that comprise the three best pairs, we found, work particularly well as a single-shape filter. This suggests that combining good single-shape filters is not necessarily the best method to construct a good multi-shape filter. Also note, that 5000 pairs of shapes is a relatively small random set. It is very likely that much better two-shape filters can be found with more extensive experiments.

4-tuples of shapes: $q = 10$, $k = 8$

In a second experiment we fixed q to 10 and tried to increase k. We generated $1,500,000$ filters with 4-tuples of random ($q = 10$)-shapes and analysed each of these 4-tuples with our algorithm. In this sample set, the good 4-tuple filters were again rare. Nevertheless, we found 15 lossless filters for $k = 8$ with specificities of about 10^{-3}. For comparison, the highest possible k for a lossless single-shape filter with $q = 9$ is $k = 6$. (A single-shape filter with $q = 9$, is about as fast as our 4-tuple filters.) The filtration criterion of the 15 4-tuples filters for $k = 8$ is that they require at least one hit of any of the four shapes.

4-tuples of shapes: $q = 10$, $k = 7$

Alternatively, each of the 15 good 4-tuple filters, we found, can also be used for $k = 7$ with a stricter filtration criterion. Although the computation of the exact filtration criterion C_{best} for this problem has a high complexity, it is easy to compute an suitable approximation C_{approx} [11]. The complement $\neg C_{approx}$ of one such approximation consists of 40 elements. This filtration criterion $\neg C_{approx}$ guarantees lossless filtering for $k = 7$ and a specificity of $5.2288 * 10^{-8}$.

The best set of four shapes out of 1,500,000 random	
$l = 50$, $k = 7$, $specificity = 5.228823e - 08$	
s_1=##-#-###---#--#----##	s_2=##-#--#--#-#-#--###
s_3=###-#-##-#-###	s_4=##-###-----##--#-#--#

$\neg C$ for the best 4-tuple filter and $k = 7$
$\{(0,0,0,0),(0,0,0,1),(0,0,0,2),(0,0,0,3),(0,0,0,4),(0,0,0,5),(0,0,1,0),$ $(0,0,1,1),(0,0,1,2),(0,0,2,0),(0,0,2,1),(0,0,3,0),(0,0,3,1),(0,0,4,0),$ $(0,0,4,1),(0,0,5,0),(0,1,0,0),(0,1,0,1),(0,1,0,2),(0,1,1,0),(0,1,1,1),$ $(0,1,2,0),(0,1,3,0),(0,1,4,0),(0,2,0,0),(0,2,0,1),(0,2,1,0),(0,3,0,0),$ $(1,0,0,0),(1,0,0,1),(1,0,0,2),(1,0,0,4),(1,0,1,0),(1,0,2,0),(1,0,4,0),$ $(1,1,0,0),(1,1,0,1),(1,2,0,0),(2,0,0,0),(5,0,0,0)\}$

The experiments show that multi-shape filters can have significantly better specificities and work for higher values of k than single-shape filters. At the same time they can also speed up the filtration. It remains an open question if there is an algorithm to construct good sets of shapes for multi-shape filters.

7 Conclusion

We described a new method for the analysis of gapped q-gram filters. This method uses bit-strings, we call them match-mismatch-patterns, to describe possible alignments between the database and the search patterns. Sets of match-mismatch-patterns provide a simple abstraction of filter algorithms for the k-differences problem.

The first step of our approach is to generate sets of match-mismatch-patterns, in particular the set of match-mismatch-patterns representing the potential matches. To implement this step efficiently, we use BDDs as a data structure to represent sets of bit-strings. In the second step, we can then use these BDD representations to compute many interesting properties of filters like the recognition rate and specificity for various probability models.

Our approach is simple and general and applies to a variety of filter algorithms. For example, it can model single-shape filters with any threshold and generic multi-shape filters. Previous algorithms for filter-related problems were often based on dynamic programming. Compared to dynamic programming, our approach is more general and more natural and allows many interesting extensions.

The most important application of our approach is the analysis of multi-shape filters, which work with a set of shapes in parallel. For any set of shapes, our approach can compute an optimal filtration criterion C_{best}, which guarantees lossless filtering for the k-differences problem and also the sensitivities and specificities of the resulting multi-shape filter.

We found, that good multi-shape filters with a carefully selected set of shapes and a specifically computed filtration criterion C_{best} are much better than single-shape filters. They allow higher specificities and sensitivities than single shape filters and higher values of k are possible (for lossless filtering). Multi-shape filters can also be faster than single shape filters, because they still work with higher values of q.

The BDD-based approach makes it possible to find good multi-shape filters by scanning a large number of randomly generated candidates. However, only a small fraction of these candidates show the desired properties. Since full enumeration as for single-shape filters [6] is not possible for multi-shape filters, a constructive algorithm to generate good sets of shapes remains an interesting open problem.

References

[1] B. Brejová, D. G. Brown, and T. Vinař. Optimal spaced seeds for hidden Markow models, with applications to homologous coding regions. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *LNCS*, pages 42–54. Springer, 2003.

[2] B. Brejová, D. G. Brown, and T. Vinař. Vector seeds: an extension to spaced seeds allows substantial improvements in sensitivity and specificity. In *Proc. 3rd International Workshop on Algorithms and Bioinformatics*, volume 2812 of *Lecture Notes in Bioinformatics*, pages 39–54. Springer, 2003.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986(8).

[4] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24:293–318, 1992(3).

[5] S. Burkhardt. *Filter Algorithms for Approximate String Matching*. PhD thesis, Department of Computer Science, Saarland University, 2002. http://www.mpi-sb.mpg.de/~stburk/thesis.ps.

[6] S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching*, volume 2089 of *LNCS*, pages 73–85. Springer, 2001.

[7] S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. *Fundamenta Informaticae*, 56(1–2):51–70, 2003.

[8] A. Califano and I. Rigoutsos. FLASH: A fast look-up algorithm for string homology. In *Proc. 1st International Conference on Intelligent Systems for Molecular Biology*, pages 56–64. AAAI Press, 1993.

[9] K. P. Choi, F. Zeng, and L. Zhang. Good spaced seeds for homology search. *Bioinformatics*, 20(7):1054–1059, 2004.

[10] K. P. Choi and L. Zhang. Sensitivity analysis and efficient method for identifying optimal spaced seeds. *Journal of Computer and System Sciences*, 68:22–40, 2004.

[11] M. Fontaine. Computing the filtration efficiency of shape-index-filters for approximate string matching. Master's thesis, Dept. of Computer Science, Saarland University, Nov 2003. http://www.mpi-sb.mpg.de/~fontaine/thesis.ps.

[12] U. Keich, M. Li, B. Ma, and J. Tromp. On spaced seeds for similarity search. *Discrete Applied Mathematics*, 138(3):253–263, 2004.

[13] G. Kucherov, L. Noé, and M. Roytberg. Multi-seed lossless filtration. To appear in CPM 2004.

[14] M. Li, B. Ma, D. Kisman, and J. Tromp. PatternHunter II: Highly Sensitive and Fast Homology Search. *Journal of Bioinformatics and Computational Biology*, 2004. To appear. Early version in GIW 2003.

[15] B. Ma, J. Tromp, and M. Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18:440–445, 2002.

[16] L. Noè and G. Kucherov. YASS: Similarity search in DNA sequences. Technical report, INRIA Tech report 4852, 2003.

[17] Y. Sun and J. Buhler. Designing multiple simultaneous seeds for DNA similarity search. In *Proceedings of the eighth annual international conference on Computational molecular biology*, pages 76–84, 2004.

[18] J. Xu, D. Brown, M. Li, and B. Ma. Optimizing multiple spaced seeds for homology search. To appear in CPM 2004.

Sorting suffixes of two-pattern strings

Frantisek Franek and W. F. Smyth*

Algorithms Research Group
Department of Computing & Software
McMaster University
Hamilton, Ontario
Canada L8S 4L7

e-mail: {franek, smyth}@mcmaster.ca

Abstract. Recently, several authors presented linear recursive algorithms for sorting suffixes of a string. All these algorithms employ a similar three-step approach, based on an initial division of the suffixes of $\boldsymbol{x}$ into two sets: in step 1 sort the first set using recursive reduction of the problem, in step 2 determine the order of the suffixes in the second set based on the order of the suffixes in the first set, and in step 3 merge the two sets together. To optimize such an algorithm either for space or time, it may not be sufficient to optimize one of the three steps, since in doing so, one might increase the resources required for the others to an unacceptable extent.

Franek, Lu, and Smyth introduced two-pattern strings as a generalization of Sturmian strings. Like Sturmian strings, two-pattern strings are generated by iterated morphisms, but they exhibit a much richer structure.

In this paper we show that the suffixes of two-pattern strings can be sorted in linear time using a variant of the three step approach outlined above. It turns out that, given the order of the suffixes in a two-pattern string, one can almost directly list in linear time all the suffixes of its expansion under a two-pattern morphism.

1 Introduction

Ever since Manber and Myers in [MM93] introduced suffix arrays as data structures comparable to suffix trees for most pattern matching tasks in strings, yet requiring significantly less memory, the search was on for a linear time algorithm for their construction. Such an algorithm for suffix tree construction had been known since 1997 [F97]. In 2003 to our knowledge three different groups of researchers independently proposed linear recursive algorithms to sort string suffixes: [KA03, KSPP03, KS03]. Though different, all three algorithms employ three steps, based on a separation of the suffixes into two sets. In step 1 the first set is ordered using recursive reduction of the problem, in step 2 the suffixes of the second set are sorted based on the order of the suffixes in the first set, and in step 3 both ordered sets are merged together. The fact that all three algorithms follow this basic approach, yet use a completely different

*also Department of Computing, Curtin University, Perth WA 6845, Australia.

separation into sets, a different way of ordering the second set based on the first set, and a different merge technique, points to some common fundamental aspect of these algorithms. To optimize such an algorithm either for space or time, it may not be sufficient to optimize one of the three steps, since in doing so, one might increase the resources required for the others to an unacceptable extent.

Two-pattern strings were introduced in [FLS03] as a generalization of Sturmian strings. Like Sturmian strings, two-pattern strings are generated by iterated morphisms, but they exhibit a much richer structure. It was shown in [FLS04] that the iterated construction of these strings could be used to compute all the repetitions and near-repetitions in time linear in string length.

This paper was motivated by our investigation of the three different linear suffix sorting algorithms discussed above and our desire to fully understand the underlying phenomena. Thus, we investigated whether the recursive nature of two-pattern strings could be used in sorting of the suffixes in the approach of the three algorithms mentioned. As it turned out, the "natural" recursive reduction of two-pattern strings can be used for step 1, and then steps 2 and 3 can be simplified into a single step: from having the suffixes of the reduced string ordered, one can almost directly list the suffixes of the two-pattern string in the right order.

For the sake of completeness, let us recall the definition of a two-pattern string (see [FLS03]), including all supporting definitions. Throughout this paper, a **binary string** means a string over the alphabet $\{a, b\}$.

Definition 1.1 *A binary string* $\boldsymbol{q}$ *is said to be* $\boldsymbol{p}$**-regular** *if and only if* $\boldsymbol{q} = \boldsymbol{upvu}$ *for some choice of (possibly empty) substrings* $\boldsymbol{u}$ *and* $\boldsymbol{v}$.

Definition 1.2 *An ordered pair* $(\boldsymbol{p}, \boldsymbol{q})$ *of nonempty binary strings is said to be* **suitable** *if and only if*

- $\boldsymbol{p}$ *is* **primitive** *(that is,* $\boldsymbol{p}$ *has no nonempty border);*
- $\boldsymbol{p}$ *is not a suffix of* $\boldsymbol{q}$*;*
- $\boldsymbol{q}$ *is neither a prefix nor a suffix of* $\boldsymbol{p}$*;*
- $\boldsymbol{q}$ *is not* $\boldsymbol{p}$*-regular.*

Note: *Since a two-pattern string is a concatenation of blocks* $\boldsymbol{p}^i\boldsymbol{q}$ *and* $\boldsymbol{p}^j\boldsymbol{q}$*, the above two definitions make sure that* $\boldsymbol{p}$ *and* $\boldsymbol{q}$ *are dissimilar enough to be recognized efficiently.*

Definition 1.3 $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]_\lambda$ *is an* **expansion of scope** $\boldsymbol{\lambda}$*, if* $(\boldsymbol{p}, \boldsymbol{q})$ *is suitable,* $|\boldsymbol{p}| \leq \lambda$*,* $|\boldsymbol{q}| \leq \lambda$*,* $1 \leq i, j$*,* $i \neq j$ *are integers, and* λ *is an integer* ≥ 1.

Note: *An expansion* $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]$ *is applied to a binary string* $\boldsymbol{x}$ *in the following fashion: each occurrence of a in* $\boldsymbol{x}$ *is replaced by* $\boldsymbol{p}^i\boldsymbol{q}$ *and each occurrence of b by* $\boldsymbol{p}^j\boldsymbol{q}$*. The resulting string is denoted as* $\sigma(\boldsymbol{x})$*. We define* $\sigma(\varepsilon) = \varepsilon$*. The composition of two expansions* $\boldsymbol{\sigma_1}$ *and* $\boldsymbol{\sigma_2}$*,* $(\boldsymbol{\sigma_1} \circ \boldsymbol{\sigma_2})(x)$ *is defined by* $(\boldsymbol{\sigma_1} \circ \boldsymbol{\sigma_2})(x) = \boldsymbol{\sigma_1}(\boldsymbol{\sigma_2}(x))$*. The role of the scope* λ *is to limit the size of* $\boldsymbol{p}$ *and* $\boldsymbol{q}$ *that can be used in the following definition.*

Definition 1.4 *A binary string* $\boldsymbol{x}$ *is a* **two-pattern string of scope** $\boldsymbol{\lambda}$ *if there exists a sequence* $\{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ *of expansions of scope* λ *so that* $\boldsymbol{x} = \sigma_1 \circ \cdots \circ \sigma_m(a)$.

It was mentioned at the end of [FLS03] that if the definition of $\boldsymbol{p}$-regularity were made more restrictive, a larger class of complete two-pattern strings could be obtained. The more restrictive definition, sufficient to give two-pattern strings all their desired properties, contained a few typographical errors as it was given in [FLS03], and so we provide a corrected definition here:

Definition 1.5 *A binary string* $\boldsymbol{q}$ *is said to be* $\boldsymbol{p}$**-regular** *(*$\boldsymbol{p}$ *a binary string) if and only if there exist (possibly empty) strings* $\boldsymbol{u}, \boldsymbol{v}$ *together with nonnegative integers* $n_1, n_2, \ldots, n_k$, $k \geq 1$, $r \geq 0$, *such that*

- *the integers* n_i *assume at most two distinct values — that is,*

$$\left|\{n_i : \ i \in 1..k\}\right| \leq 2;$$

- $\boldsymbol{q} = (\boldsymbol{u}\boldsymbol{p}^r\boldsymbol{v}\boldsymbol{p}^{n_1})(\boldsymbol{u}\boldsymbol{p}^r\boldsymbol{v}\boldsymbol{p}^{n_2})\cdots(\boldsymbol{u}\boldsymbol{p}^r\boldsymbol{v}\boldsymbol{p}^{n_k})\boldsymbol{u}$ *for some* $\boldsymbol{u}$, $\boldsymbol{v}$, $r \geq 0$, *where* $\boldsymbol{v} = \varepsilon$ *if* $r = 0$.

Note: *the definition 1.5 can be used to replace the definition 1.1. In fact, all the proofs accompanying this paper are compliant with the more restrictive definition 1.5.*

Certain finite fragments of the well-known infinite Fibonacci string and the equally well-known infinite Sturmian strings are in fact complete two-pattern strings of scope $\lambda = 1$ (see [FLS03]).

Here are a few simple examples of two-pattern strings:

1. a, now apply $\boldsymbol{\sigma_2} = [ab, ba, 2, 3]$ to it, we get
2. $\boldsymbol{\sigma_2}(a) = ababba$, now apply $\boldsymbol{\sigma_1} = [abb, aa, 1, 4]$ to it, we get
3. $\boldsymbol{\sigma_1}(\boldsymbol{\sigma_2}(ababba)) = abbaa(abb)^4aaabbaa(abb)^4aa(abb)^4aaabbaa$.

Strings 1, 2, and 3 are all two-pattern strings of scope 3 (string 2 is in fact of scope 2, and string 1 is in fact of scope 1).

It was shown in [FLS03] that complete two-pattern strings can be recognized in linear time: the recognition algorithm outputs an essentially unique sequence of expansions to construct the string from a. So in the following we can assume that not only do we have a complete two-pattern string, but also the sequence of expansions that iteratively generates the string.

In the next section we describe the principles underlying the algorithm for sorting suffixes of a two-pattern string. In Section 3 we provide an overview of the algorithm itself, while Section 5 we list some of the main lemmas on which the algorithm is based. We conclude with Section 6.

2 The Principles Underlying the Algorithm

For the sake of clarity and brevity, we introduce several symbols: we use the symbol $\boldsymbol{u} < \boldsymbol{v}$ for strings $\boldsymbol{u}$, $\boldsymbol{v}$ to express that $\boldsymbol{u}$ is lexicographically smaller than $\boldsymbol{v}$. We use the symbol $\prec$ in $\boldsymbol{u} \prec \boldsymbol{v}$ (or $\succ$ in $\boldsymbol{u} \succ \boldsymbol{v}$) to express the fact that $\boldsymbol{u} < \boldsymbol{v}$ yet $\boldsymbol{u}$ is not a prefix of $\boldsymbol{v}$ (or $\boldsymbol{v} < \boldsymbol{u}$ yet $\boldsymbol{v}$ is not a prefix of $\boldsymbol{u}$). Note that $\boldsymbol{u} < \boldsymbol{v}$ iff ($\boldsymbol{u} \prec \boldsymbol{v}$ or $\boldsymbol{u}$ is a prefix of $\boldsymbol{v}$). We use the symbol $\boldsymbol{u} \succ\!\!\prec \boldsymbol{v}$ to indicate that either $\boldsymbol{u} \prec \boldsymbol{v}$ or $\boldsymbol{u} \succ \boldsymbol{v}$.

For a binary string $\boldsymbol{u}$, we will use $\overline{\boldsymbol{u}}$ to denote its ones-complement; that is, the string formed by interchanging a's and b's in $\boldsymbol{u}$.

In accordance with [FLS03], if $\boldsymbol{x}$, $\boldsymbol{y}$ are complete two-pattern strings, σ an expansion, and $\boldsymbol{y} = \sigma(\boldsymbol{x})$, then the occurrences of copies of $\boldsymbol{p}$ and copies of $\boldsymbol{q}$ in the concatenation of blocks $\boldsymbol{p}^i\boldsymbol{q}$ and $\boldsymbol{p}^j\boldsymbol{q}$ as defined by $\sigma(\boldsymbol{x})$ are called **restrained** copies. Any other occurrence of $\boldsymbol{p}$ or $\boldsymbol{q}$ is referred to as **free**. A consecutive sequence of restrained copies of $\boldsymbol{p}$'s and/or $\boldsymbol{q}$'s will also be referred to as a **restrained configuration** or a **restrained substring** of $\boldsymbol{y}$.

Throughout the following discussion we assume that the scope λ is fixed and that $\boldsymbol{y} = \sigma(\boldsymbol{x})$, where $\boldsymbol{x}$ is a complete two-pattern string of scope λ and $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]_\lambda$ an expansion of scope λ. Moreover we assume that all suffixes of $\boldsymbol{x}$ are lexicographically sorted: $\boldsymbol{\rho}_1 < \cdots < \boldsymbol{\rho}_{|\boldsymbol{x}|}$. We then describe how to order the suffixes of $\boldsymbol{y}$. We may assume further that $\boldsymbol{q} < \boldsymbol{p}$. If it were not the case, according to Lemma 5.2 (see section 5), $\overline{\boldsymbol{q}} < \overline{\boldsymbol{p}}$, we sort all of the suffixes of $\overline{\boldsymbol{y}} = \overline{\sigma}(\boldsymbol{x})$, where $\overline{\sigma} = [\overline{\boldsymbol{p}}, \overline{\boldsymbol{q}}, i, j]_\lambda$, and reversing the order, we get all suffixes of $\boldsymbol{y}$ ordered properly.

Since we are assuming $\boldsymbol{q} < \boldsymbol{p}$, according to Lemma 5.1 (see section 5), for any suffixes $\boldsymbol{\rho}_1$, $\boldsymbol{\rho}_2$ of $\boldsymbol{x}$, if $\boldsymbol{\rho}_1 < \boldsymbol{\rho}_2$, then $\sigma(\boldsymbol{\rho}_1) < \sigma(\boldsymbol{\rho}_2)$. In simple terms, the assumption $\boldsymbol{q} < \boldsymbol{p}$ makes all expansions to preserve the order of suffixes.

We put all the suffixes of $\boldsymbol{y}$ into disjoint buckets of five types $\boldsymbol{A}$–$\boldsymbol{E}$. Their definitions follow (*note that the expansion* $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]_\lambda$ *is fixed*):

- For every nontrivial suffix $\boldsymbol{\delta}$ of $\boldsymbol{p}$ and for every integer k, $0 < k < i$, $\boldsymbol{A}_{\boldsymbol{\delta},k} = \{\boldsymbol{\delta}\boldsymbol{p}^k\boldsymbol{q}\sigma(\boldsymbol{\rho}) :\ \boldsymbol{\rho}$ is a proper suffix of $\boldsymbol{x}$ *or* $\boldsymbol{\rho} = \varepsilon\}$;
- for every nontrivial suffix $\boldsymbol{\delta}$ of $\boldsymbol{p}$ that is also a suffix of $\boldsymbol{q}$, $\boldsymbol{A}_{\boldsymbol{\delta},i} = \{\boldsymbol{\delta}\boldsymbol{p}^i\boldsymbol{q}\sigma(\boldsymbol{\rho}) :\ \boldsymbol{\rho}$ is a proper suffix of $\boldsymbol{x}$ *or* $\boldsymbol{\rho} = \varepsilon\}$;
- for every nontrivial suffix $\boldsymbol{\delta}$ of $\boldsymbol{p}$ that is not a suffix of $\boldsymbol{q}$, $\boldsymbol{A}_{\boldsymbol{\delta},i} = \{\boldsymbol{\delta}\boldsymbol{p}^i\boldsymbol{q}\sigma(\boldsymbol{\rho}) :\ b\boldsymbol{\rho}$ is a proper suffix of $\boldsymbol{x}, \boldsymbol{\rho}$ can be empty$\}$;
- for every nontrivial suffix $\boldsymbol{\delta}$ of $\boldsymbol{p}$ and for every integer k, $i < k < j$, $\boldsymbol{A}_{\boldsymbol{\delta},k} = \{\boldsymbol{\delta}\boldsymbol{p}^k\boldsymbol{q}\sigma(\boldsymbol{\rho}) :\ b\boldsymbol{\rho}$ is a proper suffix of $\boldsymbol{x}, \boldsymbol{\rho}$ can be empty$\}$.
- for every nontrivial suffix $\boldsymbol{\delta}$ of $\boldsymbol{p}$, $\boldsymbol{B}_{\boldsymbol{\delta}} = \{\boldsymbol{\delta}\boldsymbol{q}\sigma(\boldsymbol{\rho}) : \boldsymbol{\rho}$ is a proper nontrivial suffix of $\boldsymbol{x}\}$;
- for every nontrivial suffix $\boldsymbol{\delta}$ of $\boldsymbol{q}$ that is not a suffix of $\boldsymbol{p}$, $\boldsymbol{C}_{\boldsymbol{\delta}} = \{\boldsymbol{\delta}\boldsymbol{p}^i\boldsymbol{q}\sigma(\boldsymbol{\rho}) :\ a\boldsymbol{\rho}$ is a proper suffix of $\boldsymbol{x}, \boldsymbol{\rho}$ can be empty$\}$;
- for every nontrivial suffix $\boldsymbol{\delta}$ of $\boldsymbol{q}$, $\boldsymbol{D}_{\boldsymbol{\delta}} = \{\boldsymbol{\delta}\boldsymbol{p}^j\boldsymbol{q}\sigma(\boldsymbol{\rho}) :\ b\boldsymbol{\rho}$ is a proper suffix of $\boldsymbol{x}, \boldsymbol{\rho}$ can be empty$\}$;
- $\boldsymbol{E} = \{\boldsymbol{\delta}\boldsymbol{q} :\ \boldsymbol{\delta}$ is a nontrivial suffix of $\boldsymbol{p}\} \cup \{\boldsymbol{\delta} :\ \boldsymbol{\delta}$ is a nontrivial suffix of $\boldsymbol{q}\}$.

(where the term *proper suffix* refers to a suffix that is not equal to the whole string and the term *trivial suffix* refers to the empty suffix).

It is straightforward to check that any suffix of $\boldsymbol{y}$ belongs to one of the buckets $\boldsymbol{A}$–$\boldsymbol{E}$ (for proof see the supplement, see below). We are going to order the suffixes in buckets $\boldsymbol{A}$–$\boldsymbol{D}$ based on the ordering of the suffixes for $\boldsymbol{x}$ (Step 1), then merge in the suffixes from $\boldsymbol{E}$ (Steps 2 & 3); since $|\boldsymbol{E}| \leq 2\lambda$, this will not destroy the linearity of the algorithm. Note that the order within each bucket is determined by the order of suffixes of $\boldsymbol{x}$:

in the bucket $\boldsymbol{A}_{\boldsymbol{\delta},k}$: $\boldsymbol{\delta p}^k\boldsymbol{q}\sigma(\boldsymbol{\rho}_1) < \boldsymbol{\delta p}^k\boldsymbol{q}\sigma(\boldsymbol{\rho}_2)$ if $\boldsymbol{\rho}_1 < \boldsymbol{\rho}_2$;
in the bucket $\boldsymbol{B}_{\boldsymbol{\delta}}$: $\boldsymbol{\delta q}\sigma(\boldsymbol{\rho}_1) < \boldsymbol{\delta q}\sigma(\boldsymbol{\rho}_2)$ if $\boldsymbol{\rho}_1 < \boldsymbol{\rho}_2$;
in the bucket $\boldsymbol{C}_{\boldsymbol{\delta}}$: $\boldsymbol{\delta p}^i\boldsymbol{q}\sigma(\boldsymbol{\rho}_1) < \boldsymbol{\delta p}^i\boldsymbol{q}\sigma(\boldsymbol{\rho}_2)$ if $\boldsymbol{\rho}_1 < \boldsymbol{\rho}_2$;
and in the bucket $\boldsymbol{D}_{\boldsymbol{\delta}}$: $\boldsymbol{\delta p}^j\boldsymbol{q}\sigma(\boldsymbol{\rho}_1) < \boldsymbol{\delta p}^j\boldsymbol{q}\sigma(\boldsymbol{\rho}_2)$ if $\boldsymbol{\rho}_1 < \boldsymbol{\rho}_2$.

Thus, it is straightforward to list the suffixes in each bucket in the correct order, given the order of the suffixes of $\boldsymbol{x}$.

We make use of the following notation: if X, Y are sets of suffixes of $\boldsymbol{y}$, we write $X \ll Y$ iff $(\forall x \in X)(\forall y \in Y)(x < y)$. The major observation our algorithm is based on is that the buckets are linearly ordered by $\ll$; that is, pairwise orderings can be made between bucket pairs of types

$$\boldsymbol{AA}, \boldsymbol{AB}, \boldsymbol{AC}, \boldsymbol{AD}, \boldsymbol{BB}, \boldsymbol{BC}, \boldsymbol{BD}, \boldsymbol{CC}, \boldsymbol{CD}, \boldsymbol{DD}, \tag{1}$$

based on five mutually exclusive (and exhaustive) conditions on any pair $\boldsymbol{\delta_1}$, $\boldsymbol{\delta_2}$ of suffixes of $\boldsymbol{p}$ and/or $\boldsymbol{q}$:

(C1) $\boldsymbol{\delta}_1 \prec \boldsymbol{\delta}_2$;

(C2) $\boldsymbol{\delta}_1 \succ \boldsymbol{\delta}_2$;

(C3) $\boldsymbol{\delta}_1$ is a proper prefix of $\boldsymbol{\delta}_2$;

(C4) $\boldsymbol{\delta}_2$ is a proper prefix of $\boldsymbol{\delta}_1$;

(C5) $\boldsymbol{\delta}_1 = \boldsymbol{\delta}_2 = \boldsymbol{\delta}$.

Observe that, given $\boldsymbol{\delta_1}$ and $\boldsymbol{\delta_2}$, to determine which of these conditions holds requires at most λ letter comparisons (since $|\boldsymbol{\delta_1}| \leq \lambda$, $|\boldsymbol{\delta_2}| \leq \lambda$).

Thus, for example, two $\boldsymbol{A}$ buckets can be compared as follows:

(C1) $\boldsymbol{A}_{\boldsymbol{\delta}_1,k_1} \ll \boldsymbol{A}_{\boldsymbol{\delta}_2,k_2}$.

(C2) $\boldsymbol{A}_{\boldsymbol{\delta}_2,k_2} \ll \boldsymbol{A}_{\boldsymbol{\delta}_1,k_1}$.

(C3) Let $\boldsymbol{\delta}_2 = \boldsymbol{\delta}_1\boldsymbol{\delta}_1'$ for some nonempty $\boldsymbol{\delta}_1'$:

(a) if $\boldsymbol{\delta}_1' \prec \boldsymbol{p}$, then $\boldsymbol{A}_{\boldsymbol{\delta}_2,k_2} \ll \boldsymbol{A}_{\boldsymbol{\delta}_1,k_1}$;
(b) otherwise, $\boldsymbol{A}_{\boldsymbol{\delta}_1,k_1} \ll \boldsymbol{A}_{\boldsymbol{\delta}_2,k_2}$.

(C4) Let $\boldsymbol{\delta}_1 = \boldsymbol{\delta}_2\boldsymbol{\delta}_2'$ for some nonempty $\boldsymbol{\delta}_2'$:

(a) If $\boldsymbol{\delta}_2' \prec \boldsymbol{p}$, then $\boldsymbol{A}_{\boldsymbol{\delta}_1,k_1} \ll \boldsymbol{A}_{\boldsymbol{\delta}_2,k_2}$;
(b) otherwise, $\boldsymbol{A}_{\boldsymbol{\delta}_2,k_2} \ll \boldsymbol{A}_{\boldsymbol{\delta}_1,k_1}$.

(C5) (a) If $k_1 < k_2$, then $\boldsymbol{A}_{\boldsymbol{\delta},k_1} \ll \boldsymbol{A}_{\boldsymbol{\delta},k_2}$;

(b) if $k_1 = k_2$, then $\boldsymbol{A}_{\boldsymbol{\delta},k_1} = \boldsymbol{A}_{\boldsymbol{\delta},k_2}$;

(c) if $k_1 > k_2$, then $\boldsymbol{A}_{\boldsymbol{\delta},k_2} \ll \boldsymbol{A}_{\boldsymbol{\delta},k_1}$.

It is not very hard to prove that this ordering is correct. The demonstration for cases (C1), (C2) and (C5) is straightforward. For (C3), observe that we are comparing $\boldsymbol{\delta}_1\boldsymbol{p}^{k_1}\boldsymbol{q}\cdots$ with $\boldsymbol{\delta}_2\boldsymbol{p}^{k_2}\boldsymbol{q}\cdots$, hence $\boldsymbol{p}^{k_1}\boldsymbol{q}\cdots$ with $\boldsymbol{\delta}_1'\boldsymbol{p}^{k_2}\boldsymbol{q}\cdots$. Since $\boldsymbol{\delta}_1'$ is a suffix of $\boldsymbol{\delta}_2$, it is also a suffix of $\boldsymbol{p}$ and so cannot be a prefix of $\boldsymbol{p}$. It follows that either $\boldsymbol{\delta}_1' \prec \boldsymbol{p}$ or $\boldsymbol{\delta}_1' \succ \boldsymbol{p}$, and the result follows. The proof for (C4) is exactly analogous.

Furthermore the $\boldsymbol{AA}$ ordering is efficient, since the cases (a) and (b) in (C3) and (C4) can be processed in at most λ constant-time steps in addition to the λ steps that may be required to identify which condition holds: thus a total of at most 2λ steps altogether.

The results for the other pairs listed in (1) are similar: the details vary slightly from one case to another. The main result is that any of the pairs can be processed in at most 3λ steps, a constant. To avoid distracting the reader with unnecessary and uninteresting detail, we do not include the other cases here. For those details, please access the web supplement of this paper at

`http://www.cas.mcmaster.ca/~franek/web-publications.html`

This supplement will be soon available as a technical report of Department of Computing and Software, McMaster University, Hamilton, Ontario, L8S 4K1 Canada.

3 The High-Level Logic of the Algorithm

We describe only the recursive step (Step 1) that takes us from $\boldsymbol{x}$ and its sorted suffixes to the corresponding sorted suffixes of $\boldsymbol{y} = \sigma(\boldsymbol{x})$, where $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]_\lambda$. Recall that we assume $\boldsymbol{q} < \boldsymbol{p}$.

1. Create names $(A, \boldsymbol{\delta})$ for every suffix $\boldsymbol{\delta}$ of $\boldsymbol{p}$. (*This requires at most λ steps. Each name will be eventually replaced by a sequence of buckets, see below.*)

2. Sort the names according to the order described in the previous section for mutual comparison of the four $\boldsymbol{A}$ buckets (of course, according to (C1)-(C4) only). (*This requires at most $2\lambda^3$ steps as we are sorting λ names and each comparison requires $\leq 2\lambda$ steps.*)

3. Replace every name $(A, \boldsymbol{\delta})$ by a sequence of names $(A, \boldsymbol{\delta}, k)$, $1 \leq k < j$. Let us call the resulting sequence BUCKETS. (*Now we have the names of $\boldsymbol{A}$ buckets in the proper order. This requires at most $|\boldsymbol{y}|$ steps as the size of* BUCKETS *is $\leq |\boldsymbol{y}|$. Each name $(A, \boldsymbol{\delta}, k)$ will eventually be replaced by a corresponding bucket $\boldsymbol{A}_{\delta,k}$, see below.*)

4. Create names $(B, \boldsymbol{\delta})$ for every suffix $\boldsymbol{\delta}$ of $\boldsymbol{p}$. (*This requires at most λ steps. Each name $(B, \boldsymbol{\delta})$ will eventually be replaced by a corresponding bucket $\boldsymbol{B}_{\boldsymbol{\delta}}$, see below.*)

5. Merge into BUCKETS all names $(B, \boldsymbol{\delta})$ according to comparisons as described in comparing $\boldsymbol{A}$ buckets to $\boldsymbol{B}$ buckets. (*This requires at most* $|\text{BUCKETS}|3\lambda^2$ *steps, as we are merging in* λ *names and each comparison requires* $\leq 3\lambda$ *steps, hence at most* $|\boldsymbol{y}|3\lambda^2$ *steps.*)

6. Create names $(C, \boldsymbol{\delta})$ for every suffix $\boldsymbol{\delta}$ of $\boldsymbol{q}$ that is not a suffix of $\boldsymbol{p}$. (*This requires at most* λ^2 *steps. Each name* $(C, \boldsymbol{\delta})$ *will eventually be replaced by a bucket* $\boldsymbol{C}_{\boldsymbol{\delta}}$*, see below.*)

7. Merge into BUCKETS all names $(C, \boldsymbol{\delta})$ according to comparisons as described in comparing $\boldsymbol{A}$ buckets to $\boldsymbol{C}$ buckets and $\boldsymbol{B}$ buckets to $\boldsymbol{C}$ buckets. (*This requires at most* $|\text{BUCKETS}|3\lambda^2$ *steps, hence at most* $|\boldsymbol{y}|3\lambda^2$ *steps.*)

8. Create names $(D, \boldsymbol{\delta})$ for every suffix $\boldsymbol{\delta}$ of $\boldsymbol{q}$. (*This requires at most* λ *steps. Each name* $(D, \boldsymbol{\delta})$ *will eventually be replaced by a bucket* $\boldsymbol{D}_{\boldsymbol{\delta}}$*, see below.*)

9. Merge into BUCKETS all names $(D, \boldsymbol{\delta})$ according to comparisons as described in comparing $\boldsymbol{A}$ buckets to $\boldsymbol{D}$ buckets, $\boldsymbol{B}$ buckets to $\boldsymbol{D}$ buckets, $\boldsymbol{C}$ buckets to $\boldsymbol{D}$ buckets. (*Now we have all required bucket names, except* $\boldsymbol{E}$*, in proper order. This requires at most* $|\text{BUCKETS}|3\lambda^2$ *steps, hence at most* $|\boldsymbol{y}|3\lambda^2$ *steps.*)

10. Traverse BUCKETS and replace each name by a sequence of suffixes according to the sequence of suffixes of $\boldsymbol{x}$. Let us call this sequence SUFFIXES. (*We turned the names into proper buckets and merged them all together in a single list. Now we have all suffixes from buckets* $\boldsymbol{A}$*–*$\boldsymbol{D}$ *in proper order. This requires at most* $|\boldsymbol{y}|$ *steps as the size of* SUFFIXES *is* $\leq |\boldsymbol{y}|$.)

11. Merge into SUFFIXES the suffixes from the bucket $\boldsymbol{E}$. (*This requires at most* $|\text{SUFFIXES}|4\lambda^2$ *steps, as we are merging in* 2λ *suffixes, each of length* $\leq 2\lambda$*, hence at most* $|\boldsymbol{y}|4\lambda^2$ *steps.*)

SUFFIXES is now a sorted list of all suffixes of $\boldsymbol{y}$ and it took less than $\alpha|\boldsymbol{y}|$ steps, where we set $\alpha = 2\lambda^3 + 14\lambda^2 + 3\lambda + 2$. Since every reduction of a complete two-pattern string at least halves its length, altogether the algorithm with all iterative steps included took less than $\boldsymbol{\alpha}n + \boldsymbol{\alpha}\frac{n}{2} + \boldsymbol{\alpha}\frac{n}{4} + \cdots < 2\boldsymbol{\alpha}n$ steps, where n is the size of the input string.

4 An example

Let $\boldsymbol{x} = ababa$, and let $\sigma = [ba, ab, 1, 2]$. (Thus $\boldsymbol{q} = ab < \boldsymbol{p} = ba$.) Hence $\boldsymbol{y} = \sigma(\boldsymbol{x}) = baabbabaabbaabbabaabbaab$.
All nontrivial proper suffixes of $\boldsymbol{x}$ are a, aba, b, and $baba$. All nontrivial suffixes of $\boldsymbol{p}$ are ba and a, and all nontrivial suffixes of $\boldsymbol{q}$ are ab and b. Let see the buckets:
$\boldsymbol{A}_{ba,1} = \{babaab\sigma(a), babaab\sigma(aba)\} =$
$\{babaabbaab, babaabbaabbabaabbaab\} = \{\boldsymbol{y}[15..24], \boldsymbol{y}[5..24]\}$.
$\boldsymbol{A}_{a,1} = \{abaab\sigma(a), abaab\sigma(aba)\} = \{abaabbaab, abaabbaabbabaabbaab\} =$
$\{\boldsymbol{y}[16..24], \boldsymbol{y}[6..24]\}$.
$\boldsymbol{B}_{ba} = \{baab\sigma(a), baab\sigma(aba), baab\sigma(ba), baab\sigma(baba)\} =$
$\{baabbaab, baabbaabbabaabbaab, baabbabaabbaab,$

$baabbabaabbaabbabaabbaab\} = \{\boldsymbol{y}[17..24], \boldsymbol{y}[7..24], \boldsymbol{y}[11..24], \boldsymbol{y}[1..24]\}$
$\boldsymbol{B}_a = \{aab\sigma(a), aab\sigma(aba), aab\sigma(ba), aab\sigma(baba)\} =$
$\{aabbaab, aabbaabbabaabbaab, aabbabaabbaab,$
$aabbabaabbaabbabaabbaab\} = \{\boldsymbol{y}[18..24], \boldsymbol{y}[8..24], \boldsymbol{y}[12..24], \boldsymbol{y}[2..24]\}$
$\boldsymbol{C}_{ab} = \{abbaab\sigma(\varepsilon), abbaab\sigma(a)\} = \{abbaab, abbaabbabaabbaab\} =$
$\{\boldsymbol{y}[19..24], \boldsymbol{y}[9..24]\}$
$\boldsymbol{C}_b = \{bbaab\sigma(\varepsilon), bbaab\sigma(ba)\} = \{bbaab, bbaabbabaabbaab\} =$
$\{\boldsymbol{y}[20..24], \boldsymbol{y}[10..24]\}$
$\boldsymbol{D}_{ab} = \{abbabaab\sigma(a), abbabaab\sigma(aba)\} =$
$\{abbabaabbaab, abbabaabbaabbabaabbaab\} = \{\boldsymbol{y}[13..24], \boldsymbol{y}[3..24]\}$
$\boldsymbol{D}_b = \{bbabaab\sigma(a), bbabaab\sigma(aba)\} =$
$\{bbabaabbaab, bbabaabbaabbabaabbaab\} = \{\boldsymbol{y}[14..24], \boldsymbol{y}[4..24]\}$
$\boldsymbol{E} = \{baab, aab, ab, b\} = \{\boldsymbol{y}[21..24], \boldsymbol{y}[22..24], \boldsymbol{y}[23..24], \boldsymbol{y}[24..24]\}$

First note that we really got all nontrivial suffixes of $\boldsymbol{y}$. Also note that the suffixes in the buckets are in proper order. Let us see the mutual relationship of buckets: $\boldsymbol{A}_{ba,1} \gg \boldsymbol{A}_{a,1}$ (by (C1)), $\boldsymbol{A}_{ba,1} \gg \boldsymbol{B}_{ba}$ (by (C5)), $\boldsymbol{A}_{ba,1} \gg \boldsymbol{B}_a$ (by (C2)), $\boldsymbol{A}_{ba,1} \gg \boldsymbol{C}_{ab}$ (by (C2)), $\boldsymbol{A}_{ba,1} \ll \boldsymbol{C}_b$ (by (C4a)), $\boldsymbol{A}_{ba,1} \gg \boldsymbol{D}_{ab}$ (by (C2)), $\boldsymbol{A}_{ba,1} \ll \boldsymbol{D}_b$ (by (C4a)), $\boldsymbol{A}_{a,1} \ll \boldsymbol{B}_{ba}$ (by (C1)), $\boldsymbol{A}_{a,1} \gg \boldsymbol{B}_a$ (by (C5)), $\boldsymbol{A}_{a,1} \ll \boldsymbol{C}_{ab}$ (by (C3b)), $\boldsymbol{A}_{a,1} \ll \boldsymbol{C}_b$ (by (C1)), $\boldsymbol{A}_{a,1} \ll \boldsymbol{D}_{ab}$ (by (C3b)), $\boldsymbol{A}_{a,1} \ll \boldsymbol{D}_b$ (by (C1)), $\boldsymbol{B}_{ba} \gg \boldsymbol{B}_a$ (by (C2)), $\boldsymbol{B}_{ba} \gg \boldsymbol{C}_{ab}$ (by (C2)), $\boldsymbol{B}_{ba} \ll \boldsymbol{C}_b$ (by (C4b)), $\boldsymbol{B}_{ba} \gg \boldsymbol{D}_{ab}$ (by (C2)), $\boldsymbol{B}_{ba} \ll \boldsymbol{D}_b$ (by (C4a)), $\boldsymbol{B}_a \ll \boldsymbol{C}_{ab}$ (by (C3b)), $\boldsymbol{B}_a \ll \boldsymbol{C}_b$ (by (C1)), $\boldsymbol{B}_a \ll \boldsymbol{D}_{ab}$ (by (C3b)), $\boldsymbol{B}_a \ll \boldsymbol{D}_b$ (by (C1)), $\boldsymbol{C}_{ab} \ll \boldsymbol{C}_b$ (by (C1)), $\boldsymbol{C}_{ab} \ll \boldsymbol{D}_{ab}$ (by (C5)), $\boldsymbol{C}_{ab} \ll \boldsymbol{D}_b$ (by (C1)), $\boldsymbol{C}_b \gg \boldsymbol{D}_{ab}$ (by (C2)), $\boldsymbol{C}_b \ll \boldsymbol{D}_b$ (by (C5)), $\boldsymbol{D}_{ab} \ll \boldsymbol{D}_b$ (by (C1)).

Now follow the 11 steps.

1. create names $(A, ba), (A, a)$
2. sort them: $(A, a), (A, ba)$ (according to (C1))
3. "refine" the names to BUCKETS=$(A, a, 1), (A, ba, 1)$
4. create names to $(B, ba), (B, a)$
5. merge them into BUCKETS=$(B, a), (A, a, 1), (B, ba), (A, ba, 1)$
6. create names to $(C, ab), (C, b)$
7. merge them into BUCKETS= $(B, a), (A, a, 1), (C, ab), (B, ba),$
   $(A, ba, 1), (C, b)$
8. create names to $(D, ba), (D, a)$
9. merge them into BUCKETS=$(B, a), (A, a, 1), (C, ab), (D, ab), (B, ba),$
   $(A, ba, 1), (C, b), (D, b)$
10. replace the names by buckets: SUFFIXES=
    $(\boldsymbol{y}[18..24], \boldsymbol{y}[8..24], \boldsymbol{y}[12..24], \boldsymbol{y}[2..24])$, $(\boldsymbol{y}[16..24], \boldsymbol{y}[6..24])$,
    $(\boldsymbol{y}[19..24], \boldsymbol{y}[9..24])$, $(\boldsymbol{y}[13..24], \boldsymbol{y}[3..24])$,

$(\boldsymbol{y}[17..24], \boldsymbol{y}[7..24], \boldsymbol{y}[11..24], \boldsymbol{y}[1..24])$, $(\boldsymbol{y}[15..24], \boldsymbol{y}[5..24])$, $(\boldsymbol{y}[20..24], \boldsymbol{y}[10..24])$, $(\boldsymbol{y}[14..24], \boldsymbol{y}[4..24])$

11. merge in $\boldsymbol{E}$ bucket: SUFFIXES= $\boldsymbol{y}[22..24]$, $\boldsymbol{y}[18..24]$, $\boldsymbol{y}[8..24]$, $\boldsymbol{y}[12..24]$, $\boldsymbol{y}[2..24]$, $\boldsymbol{y}[23..24]$, $\boldsymbol{y}[16..24]$, $\boldsymbol{y}[6..24]$, $\boldsymbol{y}[19..24]$, $\boldsymbol{y}[9..24]$, $\boldsymbol{y}[13..24]$, $\boldsymbol{y}[3..24]$, $\boldsymbol{y}[24..24]$, $\boldsymbol{y}[21..24]$, $\boldsymbol{y}[17..24]$, $\boldsymbol{y}[7..24]$, $\boldsymbol{y}[11..24]$, $\boldsymbol{y}[1..24]$, $\boldsymbol{y}[15..24]$, $\boldsymbol{y}[5..24]$, $\boldsymbol{y}[20..24]$, $\boldsymbol{y}[10..24]$, $\boldsymbol{y}[14..24]$, $\boldsymbol{y}[4..24]$

5 The Supporting Lemmas

For the proofs, see the supplement as mention before.

The first lemma establishes that the ordering of suffixes is invariant under an expansion with $\boldsymbol{q} < \boldsymbol{p}$.

Lemma 5.1 *Let $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]_\lambda$ be an expansion and $\boldsymbol{q} < \boldsymbol{p}$. Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two-pattern strings of scope λ and let $\boldsymbol{y} = \sigma(\boldsymbol{x})$. Let $\boldsymbol{\rho}_1$, $\boldsymbol{\rho}_2$ be suffixes of $\boldsymbol{x}$ so that $\boldsymbol{\rho}_1 < \boldsymbol{\rho}_2$.*
Then $\sigma(\boldsymbol{\rho}_1) < \sigma(\boldsymbol{\rho}_2)$.

The next lemma tells us that interchanging a and b in a binary string reverses the order of the suffixes.

Lemma 5.2 *Let $\boldsymbol{\rho}_1 < \cdots < \boldsymbol{\rho}_n$ be the sequence of all suffixes of a binary string $\boldsymbol{u}$ in an ascending lexicographic order. Then $\overline{\boldsymbol{\rho}_1} > \cdots > \overline{\boldsymbol{\rho}_n}$ is the sequence of all suffixes of $\overline{\boldsymbol{u}}$ in a descending lexicographic order.*

The next three lemmas are technical lemmas required for some of the proofs (see website referenced above) that the pairs (1) can be processed correctly in $O(3\lambda)$ time. Essentially these lemmas tell us that the ordering of restrained suffixes of $\boldsymbol{y}$ can be accomplished in at most 2λ constant-time algorithmic steps.

Lemma 5.3 *Let $\boldsymbol{x}$, $\boldsymbol{y}$ be two-pattern strings of scope λ, $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]_\lambda$ an expansion, and $\boldsymbol{y} = \sigma(\boldsymbol{x})$. Let $\boldsymbol{u}$ be a non-empty binary string and let $\boldsymbol{uqp}$ be a suffix of a restrained configuration $\boldsymbol{pqp}$ of $\boldsymbol{y}$ and let $\boldsymbol{qp}$ be a restrained configuration of $\boldsymbol{y}$. Then $\boldsymbol{uqp} \succ\!\!\prec \boldsymbol{qp}$ and whether $\boldsymbol{uqp} \prec \boldsymbol{qp}$ or $\boldsymbol{uqp} \succ \boldsymbol{qp}$ can be determined in $\leq 2\lambda$ steps.*

Lemma 5.4 *Let $\boldsymbol{x}$, $\boldsymbol{y}$ be two-pattern strings of scope λ, $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]_\lambda$ an expansion, and $\boldsymbol{y} = \sigma(\boldsymbol{x})$. Let $\boldsymbol{u}$ be a non-empty binary string and let $\boldsymbol{up}$ be a suffix of a restrained configuration $\boldsymbol{qp}$ of $\boldsymbol{y}$. Let $1 \leq k$, and let $\boldsymbol{p}^k\boldsymbol{q}$ be a restrained configuration of $\boldsymbol{y}$. Then $\boldsymbol{up} \succ\!\!\prec \boldsymbol{p}^k\boldsymbol{q}$ and whether $\boldsymbol{up} \prec \boldsymbol{p}^k\boldsymbol{q}$ or $\boldsymbol{up} \succ \boldsymbol{p}^k\boldsymbol{q}$ can be determined in $\leq 2\lambda$ steps.*

Lemma 5.5 *Let $\boldsymbol{x}$, $\boldsymbol{y}$ be two-pattern strings of scope λ, $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]_\lambda$ an expansion, and $\boldsymbol{y} = \sigma(\boldsymbol{x})$. Let $\boldsymbol{u}$ be a non-empty binary string and let $\boldsymbol{up}^k\boldsymbol{q}$, $1 \leq k$, be a suffix of a restrained configuration $\boldsymbol{p}^{k+1}\boldsymbol{q}$ or $\boldsymbol{qp}^k\boldsymbol{q}$ of $\boldsymbol{y}$. Let $\boldsymbol{qp}$ be a restrained configuration of $\boldsymbol{y}$. Then $\boldsymbol{up}^k\boldsymbol{q} \succ\!\!\prec \boldsymbol{qp}$ and whether $\boldsymbol{up}^k\boldsymbol{q} \prec \boldsymbol{qp}$ or $\boldsymbol{up}^k\boldsymbol{q} \succ \boldsymbol{qp}$ can be determined in $\leq 2\lambda$ steps.*

6 Conclusion

Even though it is known that suffixes for all strings can be sorted in linear time using recursive algorithms, our research verified that for the class of complete two-pattern strings the sorting can be done iteratively, also in linear time. The analysis shows that the approach presented here is rather straightforward, thus providing additional evidence of how two-pattern strings are well-suited for computational processing, the main goal of this effort.

References

[F97] M. Farach, **Optimal suffix tree construction with large alphabets**, in *Proc. 38th Annual Symposium on Foundations of Computer Science*, IEEE (1997) pp. 137–143.

[FLS03] F. Franek, W. Lu, and W. F. Smyth, **Two-pattern strings I — a recognition algorithm**, *J. Discrete Algorithms 1–5/6* (2003) pp. 445–460.

[FLS04] F. Franek, W. Lu, and W. F. Smyth, **Two-pattern strings II — computing all repetitions and near-repetitions**, submitted to *J. Discrete Algorithms*.

[KA03] P. Ko and S. Aluru, **Space efficient linear time construction of suffix arrays**, *Proceedings of the 14th Annual Symposium CPM*, LNCS 2676, Springer (2003) pp. 200–210.

[KSPP03] D. K. Kim, J. S. Sim, H. Park, and K. Park, **Linear-time construction of suffix arrays**, *Proceedings of the 14th Annual Symposium CPM*, LNCS 2676, Springer (2003) pp. 186–199.

[KS03] J. Kärkkäinen and P. Sanders, **Simple linear work suffix array construction**, *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, LNCS 2719, Springer (2003) pp. 943–955.

[MM93] U. Manber and G. Myers, **Suffix arrays: a new method for on-line string searches**, *SIAM Journal on Computing 22–5* (1993) pp. 935–948.

Acknowledgements

The first author would like to acknowledge the support and hospitality of the School of Computing, Curtin University, Perth, Australia during the research for this paper. The research of both authors was supported in part by their respective research grants from the Natural Sciences and Engineering Research Council of Canada.

A Note on Bit-Parallel Alignment Computation

Heikki Hyyrö

PRESTO, Japan Science and Technology Agency

e-mail: `Heikki.Hyyro@cs.uta.fi`

Abstract. The edit distance between strings A and B is defined as the minimum number of edit operations needed in converting A into B or vice versa. Typically the allowed edit operations are one or more of the following: an insertion, a deletion or a substitution of a character, or a transposition between two adjacent characters. Simple edit distance allows the first two operation types, Levenshtein edit distance the first three, and Damerau distance all four. There exist very efficient $O(\lceil m/w \rceil n)$ bit-parallel algorithms for computing each of these three distances, where m is the length of A, n is the length of B, and w is the computed word size. In this paper we discuss augmenting the bit-parallel algorithms to recover an optimal alignment between A and B. Such an alignment depicts how to transform A into B by using $ed(A, B)$ operations, where $ed(A, B)$ is the used edit distance (one of the three mentioned above). Previously Iliopoulos and Pinzon have given such an algorithm for the longest common subsequence, which in effect corresponds to the simple edit distance. We propose a simpler method, which is faster and also more general in that our method can be used with any of the above three distances.

Keywords: Longest common subsequence, Levenshtein edit distance, Damerau edit distance, bit-parallelism, edit script, alignment

1 Introduction

Edit distance is a classic measure of similarity between two strings. It is generally defined as the minimum number of edit operations that are needed in order to transform one of the strings into the other. There are different types of distances depending on what kind of operations are allowed. Two common and widely studied distances are simple edit distance and Levenshtein edit distance [Lev66]. The simple edit distance permits a single edit operation to insert or delete a character. In addition to these two, Levenshtein distance allows also the operation of substituting a character with another. Damerau distance [Dam64], which is mainly used in spelling correction, extends the Levenshtein distance by allowing a fourth operation of transposing two adjacent characters. The simple edit distance is often used indirectly in its dual form of computing the length of the longest common subsequence between the two strings. Fig. 1 shows an example of these edit distances.

Throughout this paper we will use the following notation. A_i is the ith character of a string A, and $A_{i..j}$ is the substring of A that begins from its ith character and

a)			b)			c)		
	D:	go**l**d → god		D:	go**l**d → god		T:	g**ol**d → g**lo**d
	I:	god → g**l**od		I:	god → g**l**od		S:	glo**d** → glo**w**
	D:	glo**d** → glo		S:	glo**d** → glo**w**			
	I:	glo → glo**w**						

Figure 1: An example of editing the string A = "gold" into the string B = "glow". Figure a) uses only insertions (I) and deletions (D), as permitted by the simple edit distance. Figure b) corresponds to Levenshtein edit distance and uses also a substitution (S). Figure c) corresponds to Damerau distance that permits also the operation of transposing two adjacent characters (T).

ends at its jth character. If $i > j$, we define $A_{i..j}$ to denote the empty string ϵ. String C is a subsequence of A if A can be transformed into C by deleting zero or more characters from A.

The two compared strings will be denoted by A and B. We will denote the length of A by m and the length of B by n. The edit distance between A and B is denoted by $ed(A, B)$. We distinguish between different types of edit distance by using a subscript: We refer to the simple edit distance, Levenshtein distance and Damerau distance between A and B as $ed_S(A, B)$, $ed_L(A, B)$ and $ed_D(A, B)$, respectively. The length of the longest common subsequence between A and B is LLCS(A, B).

The classic and very flexible solution for computing various edit distances is based on dynamic programming. The three distances we discuss can be computed in $O(mn)$ time by filling an $(m+1) \times (n+1)$ dynamic programming matrix. Depending on the particular distance, several enhancements over the basic scheme have been proposed. We refer the reader to for example [Nav01, Gus97, BHR00] for an overview on the various algorithms for the different distances. For our purposes it is sufficient to mention that the $O(\lceil m/w \rceil n)$ bit-parallel algorithms [AD86, Mye99, CIPR01, Hyy03, Hyy04], where w is the computer word size, are typically very practical choices at least when the alphabet size is moderate (e.g. ASCII character set). These algorithms encode the differences between adjacent cells in the dynamic programming matrix into computer words of length w by using a constant number of bits per cell, and are then able to compute all values within a single word in parallel.

In this paper we consider the case of editing A into B. There are one or more minimal *edit scripts* that correspond to the value $ed(A, B)$. A minimal edit script describes a set of $ed(A, B)$ operations which transform A into B. There are applications, such as file comparison, where this information is essential. An edit script can be recovered from the dynamic programming matrix once it has been filled.

One common way to describe an edit script is to show the corresponding alignment for A and B. In this paper we discuss a simple scheme to efficiently recover an optimal alignment after the difference-encoded counterpart of the dynamic programming matrix has been computed by a bit-parallel algorithm. Previously Iliopoulos and Pinzon [IP02] have proposed this type of a method for recovering a longest common subsequence for A and B. There is a close relationship between $LLCS(A, B)$ and $ed_S(A, B)$: $ed_S(A, B) = m + n - 2 \times LLCS(A, B)$. The longest common subsequence gives effectively the same information as an optimal alignment for $ed_S(A, B)$. But the method of Iliopoulos and Pinzon is unnecessarily complicated and specifically designed for LLCS(A, B) (or $ed_S(A, B)$). Our scheme is simpler, can be used with any of the three discussed edit distances, and we also verify experimentally that it is considerably faster than the method of Iliopoulos and Pinzon.

2 Dynamic programming

The dynamic programming methods fill an $(m+1) \times (n+1)$ dynamic programming matrix D, in which each cell $D[i, j]$ will eventually hold the value $ed(A_{1..i}, B_{1..j})$. As a specific example we will review the basic dynamic programming solution for Levenshtein edit distance. The other two distances are computed in a very similar manner.

The first step is to fill trivially known boundary values. Since all three distances permit insertions and deletions, they share the same boundary values $D[0, j] = ed(A_{1..0}, B_{1..j}) = ed(\epsilon, B_{1..j}) = j$ and $D[i, 0] = ed(A_{1..i}, B_{1..0}) = ed(A_{1..i}, \epsilon) = i$. The remaining cells of D are then computed by using an appropriate recurrence. The complete recurrence for Levenshtein distance is as follows.

$D[i, 0] = i$, for $i \in 0 \ldots m$.
$D[0, j] = j$, for $j \in 0 \ldots n$.
When $1 \leq i \leq m$ and $1 \leq j \leq n$,

$$D[i, j] = \begin{cases} D[i-1, j-1], \text{ if } A_i = B_j. \\ 1 + \min(D[i-1, j], D[i, j-1], D[i-1, j-1]), \text{ otherwise.} \end{cases}$$

Here the three options in the minimum clause correspond to deleting A_i, inserting B_j after A_i, or substituting A_i with B_j, respectively.

A common way of computing the cells is to proceed in a columnwise manner. First the cells $D[1, 1], D[2, 1], \ldots D[m, 1]$, then the cells $D[1, 2], D[2, 2], \ldots D[m, 2]$, and so on until column n. Finally the desired edit distance is $ed(A, B) = D[m, n]$.

When matrix D has been filled, a sequence of $ed(A, B)$ edit operations that transforms A into B can be recovered by backtracking from the cell $D[m, n]$ towards the cell $D[0, 0]$. At each step we move from $D[i, j]$ into $D[i-1, j]$, $D[i-1, j-1]$ or $D[i, j-1]$, the only restriction being that the consecutively visited cell values have to correspond to a minimal choice made in the recurrence. The corresponding edit operations can then be recorded along the way until the cell $D[0, 0]$ is reached.

Computing LLCS(A, B) can be done in similar manner. Let L be the corresponding $(m+1) \times (n+1)$ dynamic programming matrix. The condition $L[i, j] =$ LLCS$(A_{1..i}, B_{1..j})$ will hold after L has been filled according to the following recurrence.

$L[i, 0] = 0$, for $i \in 0 \ldots m$.
$L[0, j] = 0$, for $j \in 0 \ldots n$.
When $1 \leq i \leq m$ and $1 \leq j \leq n$,

$$L[i, j] = \begin{cases} L[i-1, j-1] + 1, \text{ if } A_i = B_j. \\ 1 + \max(L[i-1, j], L[i, j-1]), \text{ otherwise.} \end{cases}$$

Instead of explicitly enumerating the operations of an edit script, similar information can be given in the form of an alignment between A and B. An alignment shows the strings A and B on two rows in such manner, that each others counterpart characters in A and B are placed into the same horizontal position (the same column). In case of inserting or deleting, one of the counterparts is an empty space. In case of a substitution, the counterpart is the substituted character. In case of a transposition between two adjacent characters, the two pairs are shown above each other. And obviously, characters that are matched in the edit sequence are each others counterparts. Fig. 2 shows an example.

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	**0**	1	2	3	4	5	6
u	2	1	**0**	1	2	3	4	5
r	3	2	1	**0**	1	2	3	4
v	4	3	2	1	**1**	2	3	4
e	5	4	3	2	2	**1**	**2**	3
y	6	5	4	3	3	2	2	**2**

```
s   u   r   v   e       y
            |       |
s   u   r   g   e   r   y
```

Figure 2: On the left: The dynamic programming matrix D for computing Levenshtein edit distance between the strings A = "survey" and B = "surgery". The cells that are traversed during a backtrack from $D[6,7]$ into $D[0,0]$ are shown in bold. It goes as follows: $D[6,7] \rightarrow D[5,6]$: match $A_6 = B_7$. $D[5,6] \rightarrow D[5,5]$: insert B_6. $D[5,5] \rightarrow D[4,4]$: match $A_5 = B_5$. $D[4,4] \rightarrow D[3,3]$: substitute A_4 with B_4. $D[3,3] \rightarrow D[2,2] \rightarrow D[1,1] \rightarrow D[0,0]$: match $A_{1..3} = B_{1..3}$. On the right: an optimal alignment that corresponds to the shown edit script trace in D. The inserted 'r' has a space as its counterpart.

3 Bit-parallel algorithms

In general, bit-parallel algorithms are based on exploiting the fact that computers process information in chunks of w bits, where w is the computer word size. If one can encode several data items into a single length-w bit-vector, then it may be possible to manipulate several items in parallel during a single computer operation. Of course the feasibility of this scheme depends highly on the type of the information and the operations one wishes to perform on them. The three types of edit distance that we discuss have turned out to be very suitable for bit-parallel computation. The bit-parallel algorithms for them reach the highest possible level of parallelism, manipulating w items at once.

In this paper we use the following notation in describing bit-operations: '&' denotes bitwise "AND", '|' denotes bitwise "OR", '^' denotes bitwise "XOR", '~' denotes bit complementation, and '<<' and '>>' denote shifting the bit-vector left and right, respectively, using zero filling in both directions. The ith bit of the bit vector V is referred to as $V[i]$ and bit-positions are assumed to grow from right to left. In addition we use a superscript to denote bit-repetition. As an example let $V = 1001110$ be a bit vector. Then $V[1] = V[5] = V[6] = 0$, $V[2] = V[3] = V[4] = V[7] = 1$, and we could also write $V = 10^21^30$.

The bit-parallel algorithms we build upon rely on the adjacency properties of D or L. It is known that two adjacent cells in a column or a row differ by at most 1. That is, the conditions $D[i-1,j]-1 \leq D[i,j] \leq D[i-1,j]+1$ and $D[i,j-1]-1 \leq D[i,j] \leq D[i,j-1]+1$ hold. In L the condition is stricter: the values never decrease along a column or a row, and so $L[i-1,j] \leq L[i,j] \leq L[i-1,j]+1$ and $L[i,j-1] \leq L[i,j] \leq L[i,j-1]+1$. These rules allow us to encode the values in each column of D by the following length-m bit-vectors:

The vertical positive delta vector VP_j:
$VP_j[i] = 1$ if and only if $D[i,j] - D[i-1,j] = 1$.

The vertical negative delta vector VN_j:
$VN_j[i] = 1$ if and only if $D[i,j] - D[i-1,j] = -1$.

Now $D[i,j] = D[0,j] + \sum_{k=1}^{i}(VP_j[k] - VN_j[k])$ if we interpret a set bit as $+1$ and an unset bit as 0.

In case of the simple edit distance $ed_S(A,B)$, only the vector VP_j is needed if one computes LLCS(A,B) instead and defines VP_j to encode differences in L instead of D.

In general we need $O(\lceil m/w \rceil)$ bit-vectors of length w in order to represent a length-m bit-vector. When each length-w segment of the bit-vectors can be computed in constant time, the overall running time for computing the vertical delta vectors for $j = 1 \ldots n$ is $O(\lceil m/w \rceil n)$. Among the discussed edit distances, the first $O(\lceil m/w \rceil n)$ bit-parallel algorithm was given by Allison and Dix [AD86] for computing LLCS(A,B). Later Myers [Mye99] presented an $O(\lceil m/w \rceil n)$ algorithm for approximate string matching under Levenshtein edit distance. That algorithm can be easily modified for computing edit distance [HN02], and a way to modify it for Damerau distance was presented in [Hyy03]. Crochemore et al. [CIPR01] and Hyyrö [Hyy04] have given alternative $O(\lceil m/w \rceil n)$ algorithms for computing LLCS(A,B).

We will not go into details of the bit-parallel algorithms themselves. For this paper the relevant thing is that we may assume that all vectors VP_j and VN_j for D (or VP_j for L) may be computed in $O(\lceil m/w \rceil n)$ time. We concentrate on the post-processing step of recovering an alignment once these vectors are known for $j = 1 \ldots n$.

4 Tracing a script

In case of the whole matrix D, recovering an alignment is simple since the backtracking procedure can directly check the values in the neighboring cells. But this is slightly more complicated if we assume only the existence of the vertical delta vectors VP_j and VN_j. The method of Iliopoulos and Pinzon [IP02] resorted to computing also horizontal differences to overcome this difficulty, although the algorithm in itself did not directly correspond to a backtracking procedure. We now note some rules that enable backtracking in D when only the vertical deltas are known.

Let us begin by considering the longest common subsequence computation. Assume that the backtracking procedure in matrix L is in the cell $L[i,j]$. From the recurrence of L we know that we can move vertically to the cell $L[i-1,j]$ if and only if $L[i,j] = L[i-1,j]$ (or $VP_j[i] = 0$). This poses no problems. Therefore let the first phase of the backtracking involve going vertically towards row 0 as long as possible, that is, as long as the corresponding bits in the vector VP_j are not set. Once we cannot move vertically, we are either at row 0 or the condition $L[i-1,j] = L[i,j]-1$ holds. In the first case we are done, as the remaining steps must go directly along row 0 to $L[0,0]$. In the latter case we know that $L[i-1,j-1] \leq L[i-1,j] = L[i,j]-1$. Consider now having the equality $L[i-1,j-1] = L[i,j-1]$ at row i in column $j-1$. Since the adjacency property states that $L[i,j-1] \geq L[i,j]-1$, we then have $L[i-1,j-1] = L[i,j-1] = L[i,j]-1$, and the only possible source for the value $L[i,j]$ is a match $A_i = B_j$. On the other hand, if $L[i-1,j-1] = L[i,j-1]-1$, then either $L[i,j-1] = L[i-1,j]+1 = L[i,j]$ or $L[i,j-1] = L[i-1,j] = L[i,j]-1$. The latter case would also have the value $L[i-1,j-1] = L[i,j]-2$, which contradicts with the recurrence for L as there is no possible source for the value $L[i,j]$. Thus the former case $L[i,j-1] = L[i-1,j]+1 = L[i,j]$ holds and we can move horizontally to the cell $L[i,j-1]$.

Now we have the following rule for cell $L[i,j]$:

$VP_j[i] = 0$: Move to the cell $L[i-1,j]$, and record that the counterpart of A_i is a space.

$VP_j[i] = 1$: Move to column $j-1$ and check the value $VP_{j-1}[i]$. If it is 1, then go to the cell $L[i,j-1]$ and record that the counterpart of B_j is a space. Otherwise go to the cell $L[i-1,j-1]$ and record that the counterpart of A_i is B_j (and they match).

Let us now consider Levenshtein or Damerau distances in similar manner. If the backtracking is in cell $D[i,j]$, we can go to the cell $D[i-1,j]$ if and only if $VP_j[i] = 1$. If $VP_j[i] = 0$, let us consider when the only choice for the backtracking is to move into $D[i,j-1]$. That happens only if we have $D[i,j-1] = D[i,j]-1$ and $D[i,j] = D[i-1,j-1]$. But in this case we must have $D[i,j-1] = D[i-1,j-1]-1$, a condition we can check from $VN_{j-1}[i]$. This gives the following backtracking rule for cell $D[i,j]$ for Levenshtein and Damerau distances.

$VP_j[i] = 1$: Move to the cell $D[i-1,j]$, and record that the counterpart of A_i is a space.

$VP_j[i] = 0$: Move to column $j-1$ and check the value $VN_{j-1}[i]$. If it is 1, then go to the cell $D[i,j-1]$ and record that the counterpart of B_j is a space. Otherwise go to the cell $D[i-1,j-1]$ and record that the counterpart of A_i is B_j (it may be a match, a substitution, or a part of a transposed character-pair).

These rules for the two distances are inherently similar and enable composing a single procedure for backtracking that works with all three distances. One just needs to feed the checked vectors as parameters, possibly in negated form. Fig. 3 shows the pseudocode for this kind of a general scheme. The shown pseudocode operates on bit-vectors of length m.

Our basic backtracking procedure takes $O(m+n)$ time. If implemented exactly as originally described in [IP02], the method of Iliopoulos and Pinzon takes $O(\lceil m/w \rceil n)$ time in the post-processing stage. But a simple modification of concentrating only on the currently processed length-w part of the matrix column enables us to implement it in $O(\lceil m/w \rceil + n)$ time. Also our backtracking method can be modified in a corresponding way to have the running time $O(\lceil m/w \rceil + n)$.

```
RecoverAlignment(delta1, delta2)
    i ← m, j ← n
    While i > 0 and j > 0 Do
        If the bit delta1_j[i] is set Then
            Output the pair (A_i, ‘ ’)
            i ← i − 1
        Else
            If the bit delta2_{j−1}[i] is set Then
                Output the pair (‘ ’, B_j)
            Else
                Output the pair (A_i, B_j)
                i ← i − 1
            j ← j − 1
    While i > 0 Do
        Output the pair (A_i, ‘ ’)
        i ← i − 1
    While j > 0 Do
        Output the pair (‘ ’, B_j)
        j ← j − 1
```

Figure 3: The general scheme for recovering an alignment from the vertical delta vectors. In the case of matrix L for LLCS(A, B), the corresponding alignment is recovered by executing **RecoverAlignment(**$\sim VP$**,** VP**)**. In the case of matrix D, one should execute **RecoverAlignment(**VP**,** VN**)**.

5 Test results

We implemented the backtracking procedure and tested it in the case of L matrix of longest common subsequence computation. This choice was made so that we could compare its performance against the method of Iliopoulos and Pinzon. Both tested methods were implemented by us. Instead of the original $O(\lceil m/w \rceil n)$ post-processing phase, we used a more efficient $O(\lceil m/w \rceil + n)$ scheme in the method of Iliopoulos and Pinzon. Our method used the basic $O(m+n)$ backtracking scheme. Despite the fact that backtracking is a rather low-cost procedure in comparison to the cost of first computing the vertical delta vectors, we chose to measure overall execution time that includes both computing the vectors and backtracking in them. The tested strings were randomly generated, and we used alphabet sizes 4 and 25. The computer was a 1.3 Ghz Intel Pentium M with 256 MB RAM and Windows XP operating system, and the code was compiled with MS Visual C++ 6.0 with full optimization options. The number of repetitions varied depending on the case in order to get feasible timings. The results are shown in Fig. 4. The numbers show the percentage of the run time of the method of Iliopoulos and Pinzon (IP) when compared to our scheme. Even though the backtracking should have a very low cost in comparison to the computation of the vectors, using our method instead has a noticeable impact even for a relatively high m, n.

$n = m$	30	50	100	300	500	1000	3000	5000
IP($\sigma = 4$)	195	155	145	114	111	106	103	103
IP($\sigma = 25$)	211	160	156	117	114	106	102	101

Figure 4: The results for the method of Iliopoulos and Pinzon as a percentage of the run time of our method. We tested with alphabet sizes $\sigma = 4$ and $\sigma = 25$.

6 Conclusion

Bit-parallel algorithms are in many cases the most efficient choice in practice for computing the simple, Levenshtein or Damerau distance, or for computing the length of the longest common subsequence. In this paper we proposed and evaluated a simple and uniform way to recover an optimal alignment for the compared strings after a bit-parallel algorithm has computed all vertical delta vectors of the corresponding dynamic programming matrix. We found that our method is more efficient than the previous method proposed by Iliopoulos and Pinzon [IP02]. Our method has also the benefit that the same scheme works with all three distances we discussed. The discussed methods for retrieving an alignment need $O(\lceil m/w \rceil n)$ space for storing the vertical delta vectors for $j = 1 \ldots n$. Thus if A and/or B are long, the space requirements may become too large. In such cases one should use for example the divide-and-conquer scheme proposed by Hirschberg [Hir78] that requires only linear space. In [CIP01] Crochemore, Iliopoulos and Pinzon discussed combining that scheme with bit-parallel LLCS(A, B) computation.

References

[AD86] L. Allison and T. L. Dix. A bit-string longest common subsequence algorithm. *Information Processing Letters*, 23:305–310, 1986.

[BHR00] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proc. 7th International Symposium on String Processing and Information Retrieval (SPIRE'00)*, pages 39–48, 2000.

[CIP01] M. Crochemore, C. S. Iliopoulos, and Y. J. Pinzon. Speeding-up Hirschberg and Hunt-Szymanski LCS algorithms. In *Proc. 8th International Symposium on String Processing and Information Retrieval (SPIRE'01)*, pages 59–67. IEEE CS Press, 2001.

[CIPR01] M. Crochemore, C. S. Iliopoulos, Y. J. Pinzon, and J. F. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, 80:279–285, 2001.

[Dam64] F. Damerau. A technique for computer detection and correction of spelling errors. *Comm. of the ACM*, 7(3):171–176, 1964.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

[Hir78] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Information Processing Letters*, 7(1):40–41, 1978.

[HN02] H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Proc. 13th Combinatorial Pattern Matching (CPM 2002)*, LNCS 2373, pages 203–224, 2002.

[Hyy03] H. Hyyrö. Bit-parallel approximate string matching algorithms with transposition. In *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE'03)*, LNCS 2857, pages 66–79, 2003.

[Hyy04] H. Hyyrö. Bit-parallel LCS-length computation revisited. In *Proc. 15th Australasian Workshop on Combinatorial Algorithms (AWOCA 2004)*, 2004.

[IP02] C. S. Iliopoulos and Y. J. Pinzon. Recovering an lcs in $O(n^2/w)$ time and space. *Columbian Journal of Computation*, 3(1):41–51, 2002.

[Lev66] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[Mye99] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.

[Nav01] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

A First Approach to Finding Common Motifs With Gaps

Costas S. Iliopoulos[1*], James McHugh[2], Pierre Peterlongo[3], Nadia Pisanti[4†], Wojciech Rytter[5] and Marie-France Sagot[6,1‡]

[1] Dept. Computer Science, King's College London, London WC2R 2LS, England, and School of Computing, Curtin University of Technology, GPO Box 1987 U, WA.
e-mail: `csi@dcs.kcl.ac.uk` `http://www.dcs.kcl.ac.uk/staff/csi`

[2] New Jersey Institute of Technology College of Computing Sciences 323 M.L.King Blvd. University Heights Newark, NJ 07102-1982, USA
e-mail: `mchugh@oak.njit.edu` `http://www.cs.njit.edu/~mchugh`

[3] Institut Gaspard-Monge, Universite de Marne-la-Valle, Cite Descartes, Champs sur Marne, 77454 Marne-la-Valle CEDEX 2, France
e-mail: `pierre.peterlongo@univ-mlv.fr`

[4] Department of Computer Science, University of Pisa, Italy
e-mail: `pisanti@di.unipi.it` `http://www.di.unipi.it/~pisanti`

[5] New Jersey Institute of Technology College of Computing Sciences 323 M.L.King Blvd. University Heights Newark, NJ 07102-1982, USA
e-mail: `rytter@oak.njit.edu` `http://www.cs.njit.edu/~rytter`

[6] Inria Rhône-Alpes, UMR 5558 Biométrie et Biologie Évolutive Université Claude Bernard Lyon 1. 43, Bd du 11 novembre 1918 69622 Villeurbanne cedex France
e-mail: `Marie-France.Sagot@inria.fr`
`http://www.inrialpes.fr/helix/people/sagot`

Abstract. We present three linear algorithms for as many formulations of the problem of finding motifs with gaps. The three versions of the problem are distinct in that they assume different constraints on the size of the gaps. The outline of the algorithm is always the same, although this is adapted each time to the specific problem, while maintaining a linear time complexity with respect to the input size. The approach we suggest is based on a re-writing of the text that uses a new alphabet made of labels representing words of the original input text. The computational complexity of the algorithm allows to use it also to find long motifs. The algorithm is in fact general enough that it could be applied to several variants of the problem other those suggested in this paper.

*The work was partially supported by a NATO grant PST.CLG.977017, a Marie Curie Fellowship, a Royal Society and a Wellcome Foundation grant.

†Partially supported by Programme Bioinformatique inter EPST and the French Ministry of Research through the ACI NIM.

‡Partially supported by Programme Bioinformatique inter EPST and the French Ministry of Research through the ACI NIM, and by Royal Society, Nato and Wellcome Foundation Grants.

1 Introduction

The inference of common motifs is an interesting problem in a variety of text algorithm applications [1, 2, 4]. Given several input strings, the goal is to find words that are shared by all (or by some) of them. In many applications, such as for instance biology, the requirement should be a certain degree of *similarity*, rather than identity, between a motif and its occurrences in the different input strings. This is what makes the problem computationally difficult and algorithmically challenging. Such similarity can be expressed in various ways. Among the most classical, in one case similarity is modelled by means of a limited edit (or Hamming) distance between a motif and its occurrences while in another don't care symbols are allowed in a motif. Our approach falls under this latter category. We are therefore interested in motifs with don't cares. A don't care is a special symbol that matches any letter of the alphabet. We further focus our attention on statements of the problem where don't cares appear concentrated in distinct sets of contiguous positions, that is, we are interested in finding common motifs with *gaps*. In computational biology, this particular problem may have applications to the inference of co-regulated genes, that is, to the detection of common binding sites whose structure often consists of motifs separated by gaps of approximatively known size. The hypothesis is that genes whose promoter regions share a binding site are co-regulated. Common motifs located in the regions upstream of genes are good candidates to be such common binding sites that could then be experimentally tested. Figure 1 shows an example of gapped motif.

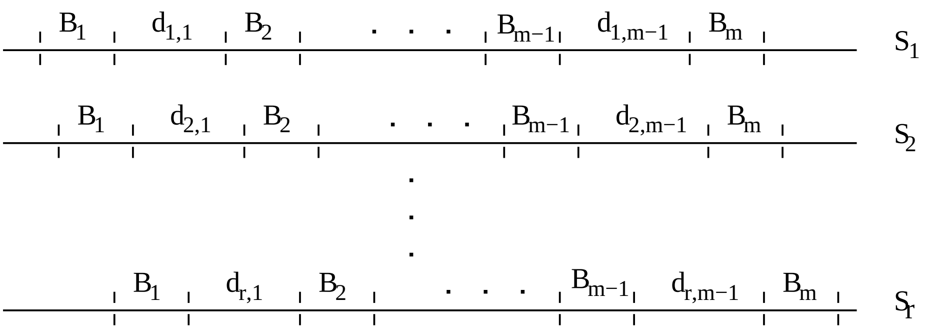


Figure 1: An example of a motif with gaps that occurs in every string, where by $d_{i,j}$ we mean a gap of size $d_{i,j}$.

Section 2 formally introduces the general problem and describes the data structure used in the algorithms presented in this paper. Section 3 defines the simplest formulation of the problem, that is, when all gaps have the same fixed size, and shows a linear solution for this problem. This will serve as a starting point for the solution to two variants of the problems where the sizes of the gaps are more flexible. Section 4 presents the first variant where each gap inside a motif has variable size but for each occurrence the total sum of such sizes is upper bounded by a fixed value. A solution for this problem is then provided that does not increase the complexity of the first algorithm. Finally, Section 5 addresses, and solves by keeping the same time complexity, another variant of the problem where each gap is of variable size but each such size is upper bounded by a fixed value.

2 Preliminaries

A *string* is a sequence of zero or more symbols from an alphabet Σ. A string s of length n is represented by $s_1 s_2 \cdots s_n$, where $s_i \in \Sigma$ for $1 \leq i \leq n$. A string w is a *substring*, or *word* of s if $s = uwv$ for $u, v \in \Sigma^*$; in this case we also say that the string w occurs at position $|u| + 1$ of the string s. We denote by $s[i..j]$ with $1 \leq i \leq j \leq n$ the word $s_i s_{i+1} \cdots s_j$ of s. A string w is a *prefix* of s if $s = wu$ for $u \in \Sigma^*$. Similarly, w is a *suffix* of s if $s = uw$ for $u \in \Sigma^*$. The don't care symbol that matches any other symbol of Σ is denoted by $*$.

The general problem addressed in this paper can be formalized in the following way.

We are given a set of r input strings $\{S_1, S_2, \ldots, S_r\}$ and we want to find gapped motifs $B = B_1 *^{d_1} B_2 *^{d_2} \cdots *^{d_{m-1}} B_m$ occurring in each of them. An example in shown in Fig. 1.

We consider three variants of the problem that basically differ in the way the size of the gaps are constrained. Assuming that the number m of motifs as well as the gap sizes (bounded by input parameters) are constants, we present linear algorithms for all three variants.

The proposed algorithms make use of a *generalised* and *truncated* suffix tree. A generalised suffix tree [5] is a suffix tree for a set of more than one input string where the information of which string each indexed suffix belongs to is stored in the nodes of the tree. A truncated suffix tree [6] is a suffix tree that stores only suffixes up to a certain length l. In practice, it is a suffix tree pruned at level l.

3 Finding motifs with fixed gaps

We start by considering the most constrained version of the problem, that is when the size of the gaps is always the same. More flexible formulations of the problem will be addressed in the next sections.

Formally, given a set of strings $\{S_1, S_2, \ldots, S_r\}$ and integers k, m, d, the problem consists in finding words $B_1, B_2, \ldots, B_m$ such that:

1. $|B_1| = |B_2| = \ldots = |B_m| = k$;
2. $B_1 *^d B_2 *^d \cdots *^d B_m$ occurs in S_i for $i = 1..r$.

Observe that technically we can allow d to be zero. This possibility becomes more interesting when dealing with gaps of variable size. The strings $B_1 *^d B_2 *^d \cdots *^d B_m$ are said to form *a gapped motif*; in the literature the sequence of words $B_1, B_2, \ldots, B_m$ is also called *a chain* and each word B_i a *block* of a gapped motif. The size of the blocks B_j's is also assumed to be fixed and the same for all blocks.

We now give an algorithm that solves the problem stated above.
Step 1 Build the generalised suffix tree, truncated at depth k, for the input strings $S_1, S_2, \ldots, S_r$. This data structure indexes all words of length k (which we call the *"k-words"*) of the set of input strings. We label each "k-word" by the number of the corresponding leaf in the suffix tree (the number of distinct such k-words is at most linear in the input size). The k-words are thus labelled in lexicographic order.

STEP 2 Build a new set of strings $\{\tilde{S}_1, \ldots \tilde{S}_r\}$ as follows. The new string $\tilde{S}_i$ is obtained from S_i by replacing each symbol with the label of the k-word that starts there. In other words, $\tilde{S}_i[j]$ becomes the label assigned to the k-word $S_i[j..j+k-1]$.
STEP 3 Search for all possible chains of m blocks by doing $m-1$ skips of length $d+k$ in $\tilde{S}_i$ (the motifs have now length $|B_1 *^d B_2 *^d \ldots *^d B_m| = m(d+k)-d$). The *chains* sought are the following:

$$C_i[j] = \tilde{S}_i[j]\tilde{S}_i[j+d+k]\tilde{S}_i[j+2(d+k)] \ldots \tilde{S}_i[j+(m-1)(d+k)],$$
$$\forall j \in \{1..n-m(d+k)+d\}$$

STEP 4 Let $\hat{S}_i$ be the set of chains $C_i[j]$, $\forall j \in \{1..n-m(d+k)+d\}$. Obtain the sets $\hat{S}_i$ for all i.
STEP 5 Build the generalised suffix tree for the set of sets of chains $\{\hat{S}_1, \ldots \hat{S}_r\}$. This tree has depth m. The leaves are labelled by the set of strings S_i each chain in $\{\hat{S}_1, \ldots \hat{S}_r\}$ is derived from.
STEP 6 The motifs sought correspond to the leaves of this second generalised suffix tree with labels covering all strings.

Observe that in general the $\hat{S}_i$'s are multisets and not sets (some of the chains may appear repeated). The redundancy may be detected at Step 4 (by not including in a set $\hat{S}_i$ a chain that is already there), or left to be detected at Step 5.

	1	2	3	4	5	6	7	8	9	10	11	12
$S_1 =$	A	C	A	A	A	A	C	A	C	A	A	A
$S_2 =$	A	C	A	C	C	A	A	C	C	A	C	A
$S_3 =$	C	A	C	A	A	A	C	C	A	C	C	A

Figure 2: An example of input

As an example, consider the set of strings of Fig. 2 and the input parameters $k = 2, d = 1$ and $m = 3$. We want to find gapped motifs of the form $b_{1,1}b_{1,2}*b_{2,1}b_{2,2}*b_{3,1}b_{3,2}$ that are common to all three strings. We first construct the truncated suffix tree as shown in Fig. 3.

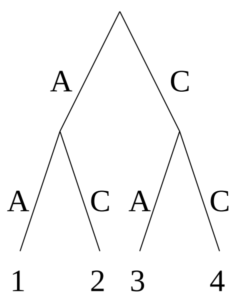


Figure 3: Truncated suffix tree for the strings of Fig. 2 and with $k = 2$

We then compute $\tilde{S}_1, \tilde{S}_2, \tilde{S}_3$ by re-writing the S_i's using the labels of the leaves. From $\tilde{S}_1, \tilde{S}_2, \tilde{S}_3$, we obtain the set of chains $\hat{S}_1, \hat{S}_2, \hat{S}_3$. The result is shown in Fig. 4

Finally, we construct the generalised suffix tree at depth $m = 3$ of the three sets of chains $\hat{S}_1, \hat{S}_2, \hat{S}_3$. The result is shown in Fig. 5. The leaf circled spells a chain that

	1	2	3	4	5	6	7	8	9	10	11	12
$\tilde{S}_1$ =	2	3	1	1	1	2	3	2	3	1	1	−
$\tilde{S}_2$ =	2	3	2	4	3	1	2	4	3	2	3	−
$\tilde{S}_3$ =	3	2	3	1	1	2	4	3	2	4	3	−

$\hat{S}_1$ = 2 1 3, 3 1 2, 1 2 3, 1 3 1, 1 2 1

$\hat{S}_2$ = 2 4 2, 3 3 4, 2 1 3, 4 2 2, 3 4 3

$\hat{S}_3$ = 3 1 4, 2 1 3, 3 2 2, 1 4 4, 1 3 3

Figure 4: Strings of Fig. 2 after steps 3 ($\tilde{S}_i$) and 4 ($\hat{S}_i$) of the algorithm

belongs to each one of the sets $\hat{S}_i$. Hence, it corresponds to a gapped motif with the required structure that occurs in every input string. This is the (only) output of our example, corresponding to the gapped motif $AC*AA*CA$ occurring in S_1 at position 1, in S_2 at position 3, and in S_3 at position 2 as shown in Fig. 6.

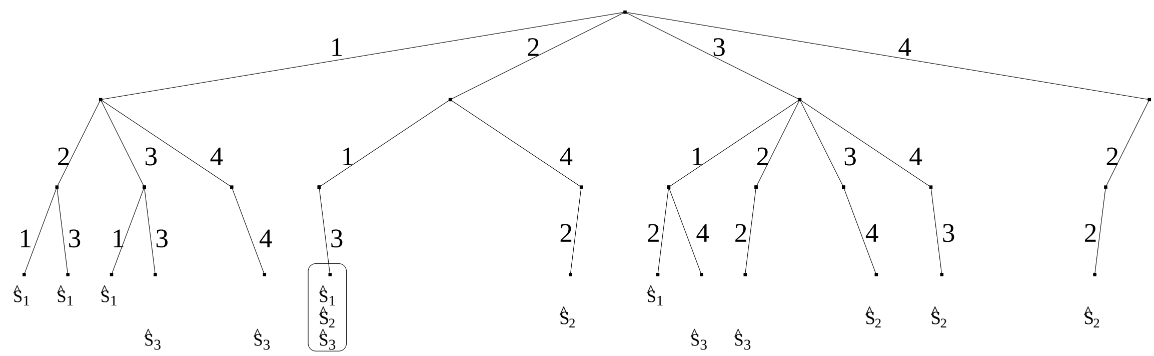


Figure 5: The generalised truncated suffix tree for $\hat{S}_1, \hat{S}_2, \hat{S}_3$. The circled leaf presents a common motif occurring in the three strings

	1	2	3	4	5	6	7	8	9	10	11	12
S_1 =	A	C	~~A~~	A	A	~~A~~	C	A	C	A	A	A
S_2 =	A	C	A	C	~~C~~	A	A	~~C~~	C	A	C	A
S_3 =	C	A	C	~~A~~	A	A	~~C~~	C	A	C	C	A

Figure 6: A motif with fixed gaps with $k = 2, m = 3$ and $D = 1$

The correctness of the algorithm is straightforward. The string $C_i[j]$ given as output in Step 6 is an encoding of a gapped motif that satisfies the input parameters. Furthermore, the construction of the generalised suffix tree of Step 5 provides the evidence that $C_i[j]$ is common to all the strings of the input set $\{S_1, S_2, \ldots, S_r\}$.

Let us now discuss the complexity of the algorithm. The construction of the generalised suffix tree at Step 1 requires $O(nr \log |\Sigma|)$ time, that is $O(nr)$ assuming $|\Sigma|$ is a constant and $n = |S_i|$ for all S_i's. The transformation of the strings S into $\tilde{S}$ requires linear $O(nr)$ time. In Step 3, we construct $n - m(d + k) + d$ chains of length $m(d+k)-d$, thus Step 3 is bounded by $O(rn)$ assuming that m, k and d are constants. Since at Step 5 the alphabet has size $O(rn)$, we resort to the suffix tree construction of [3] whose time complexity is independent of the alphabet's size. Therefore, the construction of the generalised suffix tree of Step 5 requires $O(rn)$ time, and Step 6 is also linear. Thus the overall complexity of the algorithm is $O(rn)$.

4 Finding motifs with bounded sum of variable gaps

We now consider the problem of finding motifs with gaps of variable sizes, but whose sum is upper bounded by a user defined parameter. Again, the gapped motif must occur at least once in each one of the input strings. Given a set of strings $\{S_1, S_2, \ldots, S_r\}$, and given integers k, m, d, D, the problem is then to find strings $B_1, B_2, \ldots, B_m$ such that:

1. $|B_1| = |B_2| = \ldots = |B_m| = k$;
2. $B_1 *^{d_{i,1}} B_2 *^{d_{i,2}} \ldots *^{d_{i,m-1}} B_m$ occurs in S_i for $i = 1..r$;
3. $\sum_{j=1}^{m} d_{i,j} \leq D$ for $i = 1..r$;
4. $0 \leq d_{i,j}$.

$\hat{S}_1 =$	2 1 3, 3 1 2, 1 2 3, 1 3 1, 1 2 1,	$(d_{1,1}=1, d_{1,2}=1)$
	2 1 2, 3 2 3, 1 3 1, 1 2 1,	$(d_{1,1}=2, d_{1,2}=1)$
	2 1 2, 3 1 3, 1 2 1, 1 3 1, ...	$(d_{1,1}=1, d_{1,2}=2)$
$\hat{S}_2 =$	2 4 2, 3 3 4, 2 1 3, 4 2 2, 3 4 3,	$(d_{2,1}=1, d_{2,2}=1)$
	2 3 4, 3 1 3, 2 2 2, 4 4 3,	$(d_{2,1}=2, d_{2,2}=1)$
	2 4 4, 3 3 3, 2 1 2, 4 2 3, ...	$(d_{2,1}=1, d_{2,2}=2)$
$\hat{S}_3 =$	3 1 4, 2 1 3, 3 2 2, 1 4 4, 1 3 3,	$(d_{3,1}=1, d_{3,2}=1)$
	3 1 3, 2 2 2, 3 4 4, 1 3 3,	$(d_{3,1}=2, d_{3,2}=1)$
	3 1 3, 2 1 2, 3 2 4, 1 4 3, ...	$(d_{3,1}=1, d_{3,2}=2)$

Figure 7: The strings $\{\hat{S}_1, \ldots \hat{S}_r\}$ after step 4. The end of each line shows the gap lengths applied to obtain the values (omitted if equal to zero).

What follows is a solution to the problem just stated.

Step 1 As was done in the first step of the previous algorithm, build the truncated generalised suffix tree for the input set of strings, and label each "k-word" with the number of the corresponding leaf in the suffix tree.

Step 2 Build the strings $\{\tilde{S}_1, \ldots, \tilde{S}_r\}$.

Step 3 In order to compute the different chains, we now use a window of width $D + km$. Let $W_{i,j}$ be the window appearing in string i at position j, and let $C_i[j]$ be the set of all non-overlapping words of length m that occur in $W_{i,j}$:

$$C_i[j] = \Big\{ \tilde{S}_i[p_1], \tilde{S}_i[p_2], \ldots, \tilde{S}_i[p_m] \;\; | \;\; p_1 = j, \;\; \forall \;\; l > 1, k \leq p_l - p_{l-1} < k + D \text{ and } p_m - p_1 \leq D + (m-1)k \text{ and } p_m < n - k \Big\}$$

Without the condition $(p_m - p_1 \leq D + (m-1)k)$, $C_i[j]$ would contain D^{m-1} elements. Thus there are at most D^{m-1} elements in $C_{i,j}$.

Step 4 Obtain the sets of chains $\hat{S}_i$.

Step 5 Construct the generalised suffix tree for the set of sets of chains $\{\hat{S}_1, \ldots, \hat{S}_r\}$.

Step 6 The motifs sought correspond as before to the leaves of this second generalised suffix tree with labels covering all strings.

Consider the same input strings as in the previous example (see Fig. 2), the same values for k and m, and $D = 3$. The $\tilde{S}_i$ strings are thus the same as before, while the $\hat{S}_i$ strings for motifs with bounded sum of variable gaps are shown in Fig. 7.

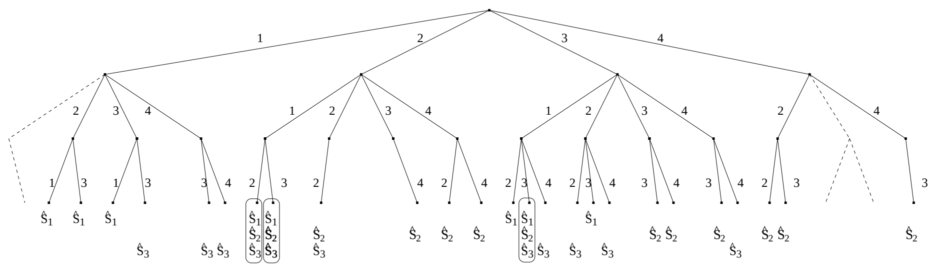


Figure 8: Part of the generalised truncated suffix tree for $\hat{S}_1, \hat{S}_2, \hat{S}_3$. Each circled leaf represents a common motif occurring in the three strings. Dashed paths indicate simply that all the tree is not drawn.

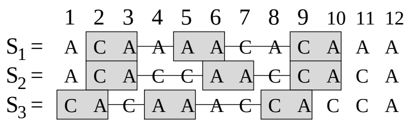


Figure 9: A gapped motif found for the bounded sum of variable gaps problem with $k = 2, m = 3$ and $D = 3$

Part of the generalised suffix tree for the set of chains $\hat{S}_1, \hat{S}_2, \hat{S}_3$, is shown in Fig. 8.

In this tree, three leaves present motifs common to all the strings. For instance, the one given by the path 313 corresponds to the gapped motif $CA *^{d_{i,1}} AA *^{d_{i,2}} CA$ with $0 \leq d_{i,j}$ and $\sum_{j=1}^{m} d_{i,j} \leq D$, $\forall i = 1..r$ which occurs in all the strings as shown in Figure 9.

The correctness of the algorithm is straightforward. The output string $C_i[j]$ in Step 6 is an encoding of a motif with bounded sum of gaps according to the input parameters that is common to all the strings $\{S_1, S_2, \ldots, S_r\}$.

If one assumes that m and D are constants, the construction of the generalised suffix tree at Step 1 requires $O(nr \log |\Sigma|)$ time, where $n = |S_i|$. The transformation of the strings S into the set of chains $\tilde{S}$ requires linear $O(nr)$ time.

For each position i in a string, the number of motifs that may have an occurrence starting at i is at most D^{m-1}. Thus, if one considers all positions in the strings, there are at most $r \cdot n \cdot D^{m-1}$ motifs. Hence, the overall cost of Step 3 is bounded by $O(rn)$ assuming that m and D are constants.

Step 5 requires $O(rn)$ time, and Step 6 is linear. The overall complexity of the algorithm is therefore again $O(rn)$, and hence linear in the input size.

5 Finding motifs with variable gaps of bounded size

In this section, we address a third variant of the problem. We have again that the size of the gaps is variable, but this time we set an upper bound on the size of each individual gap.

Given a set of strings $\{S_1, S_2, \ldots, S_r\}$ and given integers k, m, D, the problem consists now in finding strings $B_1, B_2, \ldots, B_m$ such that:

1. $|B_1| = |B_2| = \ldots = |B_m| = k$
2. $B_1 *^{d_{i,1}} B_2 *^{d_{i,2}} \ldots *^{d_{i,m-1}} B_m$ occurs in S_i for $i = 1..r$;
3. $1 \leq d_{i,j} \leq D$ for $i = 1..r, j = 1..m$.

Notice that putting an upper bound on the size of each gap implies also putting a bound on the size of their sum, as is the case of the previous variant of the problem.

$\hat{S}_1 =$ ...		$\hat{S}_2 =$ ...		$\hat{S}_3 =$ ...	
2 2 3, 3 3 1, 1 2 1,	($d_{1,1}$=3, $d_{1,2}$=1)	2 1 3, 3 2 2, 2 4 3,	($d_{2,1}$=3, $d_{2,2}$=1)	3 2 2, 2 4 4, 3 3 3,	($d_{3,1}$=3, $d_{3,2}$=1)
2 1 3, 3 2 1, 1 3 1,	($d_{1,1}$=2, $d_{1,2}$=2)	2 3 3, 3 1 2, 2 2 3,	($d_{2,1}$=2, $d_{2,2}$=2)	3 1 2, 2 2 4, 3 4 3,	($d_{3,1}$=2, $d_{3,2}$=2)
2 1 1, 3 2 1, ...	($d_{1,1}$=2, $d_{1,2}$=3)	2 3 2, 3 1 3, ...	($d_{2,1}$=2, $d_{2,2}$=3)	3 1 4, 2 2 3, ...	($d_{3,1}$=2, $d_{3,2}$=3)

Figure 10: Part of the strings $\hat{S}_i$ after step 4. The end of each line shows the gap lengths applied to obtain the values.

What follows is an algorithm that solves this last variant of the problem.

STEP 1 Compute the truncated generalised suffix tree for the k-words of the set of input strings, and as before label each "k-word" with the number of the corresponding leaf in the suffix tree.
STEP 2 Construct the strings $\{\tilde{S}_1, \ldots, \tilde{S}_r\}$ as in the previous algorithms.
STEP 3 Let $C_i[j]$ be the set of all the possible chains $\tilde{S}_i[p_1], \tilde{S}_i[p_2], \ldots, \tilde{S}_i[p_m]$ in the string $\tilde{S}_i$ such that $p_1 = j$ and $\forall\ \ j > 1, k \geq p_j - p_{j-1} < k + D$.

Formally,

$$C_i[j] = \Big\{ \tilde{S}_i[p_1], \tilde{S}_i[p_2], \ldots, \tilde{S}_i[p_m] \ \mid\ p_1 = j \text{ and } \forall\ \ l > 1, k \leq p_l - p_{l-1} < k + D \text{ and } p_m < n - k \Big\}.$$

Notice that $C_i[j]$ contains at most D^{m-1} elements.
STEP 4 Obtain the set of chains $\hat{S}_i$.
STEP 5 As for the first algorithm, build the generalised suffix tree for $\{\hat{S}_1, \ldots \hat{S}_r\}$.
STEP 6 The motifs sought correspond as before to the leaves of this second generalised suffix tree with labels covering all strings.

For example, with the strings shown in Fig. 2, the same values for k and m (i.e., $k = 2, m = 3$), and $D = 3$, part of the $\hat{S}_i$ strings obtained for this last variant of the problem is shown in Fig. 10.

Part of the generalised suffix tree for the set of chains $\hat{S}_1, \hat{S}_2, \hat{S}_3$, is shown in Fig. 11.

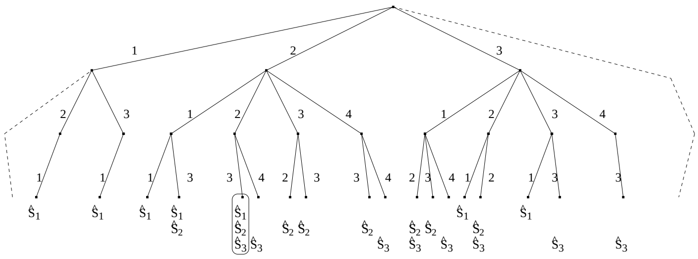


Figure 11: A part of the generalised truncated suffix tree for $\hat{S}_1, \hat{S}_2, \hat{S}_3$. The circled leaf presents a common motif occurring in the three strings. The dashed paths indicate simply that all the tree is not drawn.

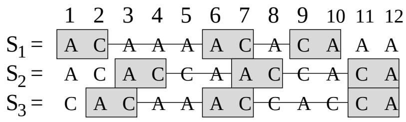


Figure 12: A gapped motif found for the bounded variable gaps problem with $k = 2, m = 3$ and $D = 3$

In this tree, one of the leaves corresponds to a motif common to all the strings. It denotes the path 223 corresponding to the gapped motif $AC *^{d_{i,1}} AC *^{d_{i,2}} CA$ with $0 \leq d_{i,j} \leq D \ \ \forall i = 1..r$ which is present in all the strings as shown in Figure 12.

The string $C_i[j]$ output in Step 6 is an encoding of a structured motif with each gap bounded by a fixed value that is common to all the input strings $\{S_1, S_2, \ldots, S_r\}$.

The construction of the generalised suffix tree at Step 1 requires $O(nr \log |\Sigma|)$ time, where $n = |S_i|$. The transformation of the strings S into $\tilde{S}$ requires linear $O(nr)$ time. For each position i in each string, the number of motifs that may have an occurrence at i is at most D^{m-1}. Hence, there are overall $r * n * D^{m-1}$ motifs. Step 3 takes $O(rn)$ assuming that m and D are constants. Again, the construction of the generalised suffix tree of Step 5, as well as Step 6, can be done in linear time. The overall complexity of the algorithm for the third variant of the problem is therefore $O(rn)$.

6 Conclusion

We have presented algorithms for three different variants of the problem of finding motifs with gaps. The approach we suggest allows to find motifs with gaps maintaining a linear time complexity with respect to the input size. The technique we applied is general enough that it can be used for several other variants of the problem besides those addressed in this paper.

Another interesting further direction to explore would be to allow errors (*i.e.*, substitutions and insertions deletions) inside the blocks.

References

[1] M. Crochemore, W. Rytter, *Text algorithms*, Oxford Press, 1994.

[2] T. Crawford, C.S. Iliopoulos, R. Raman, String matching techniques for musical similarity and melodic recognition, *Computing in Musicology*, Vol. 11, pp. 73–100, 1998.

[3] M. Farach, Optimal suffix tree construction with large alphabets, in *Foundations of Computer Science (FOCS '97)*, 137–143, 1997.

[4] C. Charras, T. Lecroq, *Handbook of Exact String Matching Algorithms*, King's College London publications 2004.

[5] P. Bieganski, J. Riedl, J. V. Carlis, Generalized Suffix Trees for Biological Sequence Data: Applications and Implementation, in *Proc. Hawaii International Conference on System Sciences*, 1994.

[6] J. Allali, M.-F. Sagot, The at-most k-deep factor tree, Internal Report IGM 2004-03 (Institut Gaspard Monge), 2004.

A Fully Compressed Pattern Matching Algorithm for Simple Collage Systems

Shunsuke Inenaga[1], Ayumi Shinohara[2,3] and Masayuki Takeda[2,3]

[1] Department of Computer Science, P.O. Box 26 (Teollisuuskatu 23) FIN-00014
University of Helsinki, Finland
e-mail: `inenaga@cs.helsinki.fi`

[2] Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan

[3] SORST, Japan Science and Technology Agency (JST)
e-mail: {`ayumi, takeda`}`@i.kyushu-u.ac.jp`

Abstract. We study the *fully compressed pattern matching problem* (*FCPM problem*): Given $\mathcal{T}$ and $\mathcal{P}$ which are descriptions of text T and pattern P respectively, find the occurrences of P in T *without decompressing* $\mathcal{T}$ *or* $\mathcal{P}$. This problem is rather challenging since patterns are also given in a compressed form. In this paper we present an FCPM algorithm for *simple collage systems*. Collage systems are a general framework that can represent various kinds of dictionary-based compressions, and simple collage systems are a subclass that includes LZW and LZ78 compressions. Collage systems are of the form $\langle \mathcal{D}, \mathcal{S} \rangle$, where $\mathcal{D}$ is a dictionary and $\mathcal{S}$ is a sequence of variables from $\mathcal{D}$. Our FCPM algorithm performs in $O(\|\mathcal{D}\|^2 + mn \log |\mathcal{S}|)$ time, where $n = |\mathcal{T}| = \|\mathcal{D}\| + |\mathcal{S}|$ and $m = |\mathcal{P}|$. This is faster than the previous best result of $O(m^2n^2)$ time.

Keywords: string processing, text compression, fully compressed pattern matching, collage systems, algorithm

1 Introduction

The *pattern matching problem*, which is the most fundamental problem in Stringology, is the following: Given text T and pattern P, find the occurrences of P in T. The *compressed pattern matching problem* (*CPM problem*) [1] is a more challenging version of the above problem, where text T is given in a compressed form $\mathcal{T}$, and the aim is to find the pattern occurrences *without decompressing* $\mathcal{T}$. This problem has been intensively studied for a variety of text compression schemes, e.g. [2, 4, 3, 17].

Classically, effectiveness of compression schemes was measured by only compression ratio and (de)compression speeds. As regards recent increasing demands for fast CPM, CPM speed has become another measurement. Shibata et al. [21] proposed a CPM algorithm for byte-pair encoding (BPE) [5] which is even faster than pattern matching in uncompressed texts. Though BPE is less effective in compression speed and ratio, BPE has gathered much attention due to its potential for fast CPM. Another good example is Manber's text compression designed to achieve fast CPM [15].

An ultimate extension of the CPM problem is the *fully compressed pattern matching problem* (*FCPM problem*) [9] where *both* text T and pattern P are given in a compressed form. We formalize this problem as follows: Given $\mathcal{T}$ and $\mathcal{P}$ that are descriptions of text T and pattern P respectively, find the occurrences of P in T *without decompressing* $\mathcal{T}$ *or* $\mathcal{P}$. Miyazaki et al. [18] presented an algorithm to solve the FCPM problem for *straight line programs*, in $O(m^2n^2)$ time using $O(mn)$ space, where $m = |\mathcal{P}|$ and $n = |\mathcal{T}|$. We refer to [20] for more details of the FCPM problem.

Collage systems [10] are a general framework that enables us to capture the essence of CPM for various dictionary-based compressions. Dictionary-based compression generates a dictionary of repeating segments of a given string and in this way a compressed representation of the string is obtained. A collage system is a pair $\langle \mathcal{D}, \mathcal{S} \rangle$ where $\mathcal{D}$ is a dictionary and $\mathcal{S}$ is a sequence of variables from $\mathcal{D}$. Collage systems cover dictionary-based compressions such as LZ family [24, 22, 25, 23] and run-length encoding, as well as grammar-based compressions such as BPE [5], RE-PAIR [14], SEQUITUR [19], grammar transform [11, 13, 12], and straight line programs [9].

In this paper, we treat *simple collage systems* which are a subclass of collage systems. Simple collage systems include LZ78 [25] and LZW [23] compressions. Although simple collage systems in general give weaker compression, CPM on simple collage systems can be accelerated and thus they are still quite attractive [16].

We reveal another yet potential benefit of simple collage systems by proposing an efficient FCPM algorithm. The proposed algorithm runs in $O(\|\mathcal{D}\|^2 + mn \log |\mathcal{S}|)$ time using $O(\|\mathcal{D}\|^2 + mn)$ space. Although our algorithm requires more space than the algorithm of [18], ours is faster than that. It should also be mentioned that Gąsieniec and Rytter [7] addressed an FCPM algorithm running in $O((m+n)\log(m+n))$ time for LZW compression, but actually their algorithm *explicitly decompresses* part of $\mathcal{T}$ or $\mathcal{P}$ when the decompressed size does not exceed n. Hence their algorithm does not suit the FCPM problem setting where pattern matching *without* decompressing is required. On the other hand, the algorithm proposed in this paper permits us to solve the FCPM problem without any explicit decompression.

2 Preliminary

Let $\mathcal{N}$ be the set of natural numbers, and $\mathcal{N}^+$ be the set of positive integers. 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 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 $1 \leq i \leq j \leq |T|$. A *period* of a string T is an integer p ($1 \leq p \leq |T|$) such that $T[i] = T[i+p]$ for any $i = 1, 2, \ldots, |T| - p$.

Collage systems [10] are a general framework that enables us to capture the structure of different types of dictionary-based compressions. A collage system is a pair $\langle \mathcal{D}, \mathcal{S} \rangle$ such that $\mathcal{D}$ is a sequence of assignments

$$X_1 = expr_1, X_2 = expr_2, \ldots, X_h = expr_h,$$

where X_k are variables and $expr_k$ are expressions of any of the form

$$\begin{array}{cll}
a & \text{where } a \in (\Sigma \cup \varepsilon), & (\textit{primitive assignment}) \\
X_i X_j & \text{where } i, j < k, & (\textit{concatenation}) \\
{}^{[j]}X_i & \text{where } i < k \text{ and } j \in \mathcal{N}^+, & (\textit{prefix truncation}) \\
X_i^{[j]} & \text{where } i < k \text{ and } j \in \mathcal{N}^+, & (\textit{suffix truncation}) \\
(X_i)^j & \text{where } i < k \text{ and } j \in \mathcal{N}^+, & (\textit{repetition})
\end{array}$$

and $\mathcal{S}$ is a sequence of variables $X_{i_1}, X_{i_2}, \ldots, X_{i_s}$ obtained from $\mathcal{D}$. The size of $\mathcal{D}$ is h and is denoted by $\|\mathcal{D}\|$, and the size of $\mathcal{S}$ is s and is denoted by $|\mathcal{S}|$. The total size of the collage system $\langle \mathcal{D}, \mathcal{S} \rangle$ is $n = \|\mathcal{D}\| + |\mathcal{S}| = h + s$.

LZW [23] and LZ78 [25] compressions can be represented by the following collage systems:

LZW. $\mathcal{S} = X_{i_1}, X_{i_2}, \ldots, X_{i_s}$ and $\mathcal{D}$ is the following:

$$\begin{array}{l}
X_1 = a_1;\ X_2 = a_2;\ \ldots;\ X_q = a_q; \\
X_{q+1} = X_{i_1} X_{\sigma(i_2)};\ X_{q+2} = X_{i_2} X_{\sigma(i_3)};\ \ldots;\ X_{q+s-1} = X_{i_{s-1}} X_{\sigma(i_s)},
\end{array}$$

where the alphabet is $\Sigma = \{a_1, a_2, \ldots, a_q\}$, $1 \le i_1 \le q$, and $\sigma(j)$ denotes the integer k $(1 \le k \le q)$ such that a_k is the first symbol of X_j.

LZ78. $\mathcal{S} = X_1, X_2, \ldots, X_s$ and $\mathcal{D}$ is the following:

$$X_0 = \varepsilon;\ X_1 = X_{i_1} b_1;\ X_2 = X_{i_2} b_2;\ \ldots;\ X_s = X_{i_s} b_s;$$

where b_j is a symbol in Σ.

We remark that LZW is a simplification of LZ78.

Definition 1 *A collage system is said to be* regular *if it contains primitive assignments and concatenations only. A regular collage system is said to be* simple *if, for any variable $X = X_\ell X_r$, $|X_\ell| = 1$ or $|X_r| = 1$.*

Simple collage systems were first introduced by Matsumoto et al. [16]. LZW and LZ78 compressions are a simple collage system.

In this paper, we study the *fully compressed pattern matching problem for simple collage systems*: Given two simple collage systems that are the descriptions of text T and pattern P, find all occurrences of P in T. Namely, we compute the following set:

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

We emphasize that our goal is to solve this problem *without decompressing either of the two simple collage systems.* Our result is the following:

Theorem 1 *Given two simple collage systems $\langle \mathcal{D}, \mathcal{S} \rangle$ and $\langle \mathcal{D}', \mathcal{S}' \rangle$ that are the description of T and P respectively, $Occ(T, P)$ can be computed in $O(\|\mathcal{D}\|^2 + mn \log |\mathcal{S}|)$ time using $O(\|\mathcal{D}\|^2 + mn)$ space, where $n = \|\mathcal{D}\| + |\mathcal{S}|$ and $m = \|\mathcal{D}'\| + |\mathcal{S}'|$.*

3 Overview of algorithm

3.1 Translation to straight line programs

Consider a regular collage system $\langle \mathcal{D}, \mathcal{S} \rangle$. Note that $\mathcal{S} = X_{i_1}, X_{i_2}, \ldots, X_{i_s}$ can be translated in linear time to a sequence of assignments of size s. For instance, $\mathcal{S} = X_1, X_2, X_3, X_4$ can be rewritten to $X_5 = X_1X_2$; $X_6 = X_5X_3$; $X_7 = X_6X_4$, and $S = X_7$. Therefore, a regular collage system, which represents string $T \in \Sigma^*$, can be seen as a context free grammar of the Chomsky normal form that generates only T. This means that regular collage systems correspond to *straight line programs* (*SLPs*) introduced in [9]. In the sequel, for string $T \in \Sigma^*$, let $\mathcal{T}$ denote the SLP representing T. The size of $\mathcal{T}$ is denoted by $\|\mathcal{T}\|$, and $\|\mathcal{T}\| = \|\mathcal{D}\| + |\mathcal{S}| = h + s = n$.

Now we introduce *simple* straight line programs (*SSLP*) that correspond to simple collage systems.

Definition 2 *An SSLP $\mathcal{T}$ is a sequence of assignments such that*

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

where X_i are variables and $expr_i$ are expressions of any of the form

$$\begin{array}{cll} a & \textit{where } a \in \Sigma & \text{(primitive)}, \\ X_\ell X' & \textit{where } \ell < i \textit{ and } X' = a & \text{(right simple)}, \\ X' X_r & \textit{where } r < i \textit{ and } X' = a & \text{(left simple)}, \\ X_\ell X_r & \textit{where } \ell, r < i & \text{(complex)}, \end{array}$$

and $\mathcal{T} = X_n$. Moreover, each type of variable satisfies the following properties:

- *For any right simple variable $X_i = X_\ell X'$, X_ℓ is either simple or primitive.*

- *For any left simple variable $X_i = X' X_r$, X_r is either simple or primitive.*

- *For any complex variable $X_i = X_\ell X_r$, X_r is either simple or primitive.*

An example of an SSLP $\mathcal{T}$ for string $T = \texttt{abaababab b}$ is as follows:

$$X_1 = \mathtt{a}, X_2 = \mathtt{b}, X_3 = X_1X_2, X_4 = X_1X_3, X_5 = X_3X_1, X_6 = X_2X_2,$$
$$X_7 = X_3X_4, X_8 = X_7X_5, X_9 = X_8X_6,$$

and $\mathcal{T} = X_9$. See also Figure 1 that illustrates the derivation tree of $\mathcal{T}$. X_1 and X_2 are primitive variables, X_3, X_4, X_5 and X_6 are simple variables, and X_7, X_8 and X_9 are complex variables.

For any simple collage system $\langle \mathcal{D}, \mathcal{S} \rangle$, let $\mathcal{T}$ be its corresponding SSLP. Let $\|\mathcal{D}\| = h$ and $|\mathcal{S}| = s$. Then the total number of primitive and simple variables in $\mathcal{T}$ is h, and the number of complex variables in $\mathcal{T}$ is s.

In the sequel, we consider computing $Occ(T, P)$ for given SSLPs $\mathcal{T}$ and $\mathcal{P}$. We use X and X_i for variables of $\mathcal{T}$, and Y and Y_j for variables of $\mathcal{P}$. When not confusing, X_i (Y_j, respectively) also denotes the string derived from X_i (Y_j, respectively). Let $\|\mathcal{T}\| = n$ and $\|\mathcal{P}\| = m$.

Proposition 1 *For any simple variable X, $|X| = \|X\|$, where $\|X\|$ denotes the number of variables in X.*

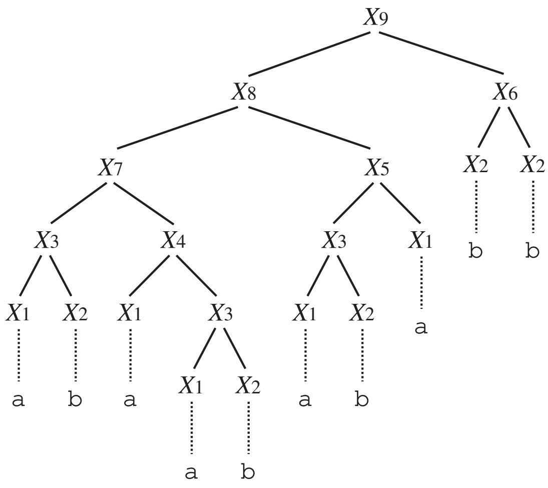


Figure 1: Derivation tree of SSLP for string `abaabababb`.

3.2 Basic idea of algorithm

In this section, we show a basis of our algorithm that outputs a compact representation of $Occ(T, P)$ for given SSLPs $\mathcal{T}, \mathcal{P}$.

For strings $X, Y \in \Sigma^*$ and integer $k \in \mathcal{N}$, we define the set of all occurrences of Y that cover or touch the position k in X by

$$Occ^{\uparrow}(X, Y, k) = \{i \in Occ(X, Y) \mid k - |Y| \leq i \leq k\}.$$

In the following, $[i, j]$ denotes the set $\{i, i+1, \ldots, j\}$ of consecutive integers. For a set U of integers and an integer k, we denote $U \oplus k = \{i + k \mid i \in U\}$ and $U \ominus k = \{i - k \mid i \in U\}$.

Observation 1 ([8]) *For any strings $X, Y \in \Sigma^*$ and integer $k \in \mathcal{N}$,*

$$Occ^{\uparrow}(X, Y, k) = Occ(X, Y) \cap [k - |Y|, k].$$

Lemma 1 ([8]) *For any strings $X, Y \in \Sigma^*$ and integer $k \in \mathcal{N}$, $Occ^{\uparrow}(X, Y, k)$ forms a single arithmetic progression.*

For positive integers $p, d \in \mathcal{N}^+$ and non-negative integer $t \in \mathcal{N}$, we define $\langle p, d, t \rangle = \{p + (i-1)d \mid i \in [1, t]\}$. Note that t denotes the cardinality of the set $\langle p, d, t \rangle$. By Lemma 1, $Occ^{\uparrow}(X, Y, k)$ can be represented as the triple $\langle p, d, t \rangle$ with the minimum element p, the common difference d, and the length t of the progression. By *'computing $Occ^{\uparrow}(X, Y, k)$'*, we mean to calculate the triple $\langle p, d, t \rangle$ such that $\langle p, d, t \rangle = Occ^{\uparrow}(X, Y, k)$.

Observation 2 *Assume each of sets A_1 and A_2 of integers forms a single arithmetic progression, and is represented by a triple $\langle p, d, t \rangle$. Then, the union $A_1 \cup A_2$ can be computed in constant time.*

Lemma 2 ([8]) *For strings $X, Y \in \Sigma^*$ and integer $k \in \mathcal{N}$, let $\langle p, d, t \rangle = Occ^{\uparrow}(X, Y, k)$. If $t \geq 1$, then d is the shortest period of $X[p : q + |Y| - 1]$ where $q = p + (t-1)d$.*

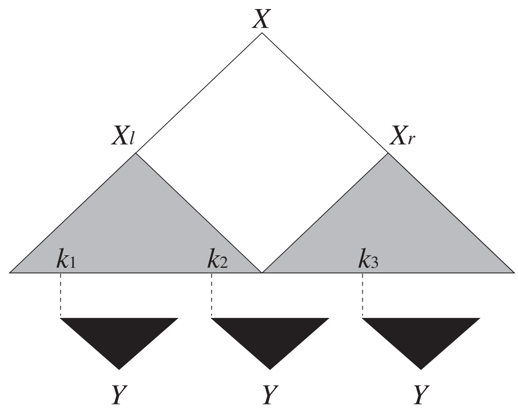


Figure 2: $k_1, k_2, k_3 \in Occ(X, Y)$, where $k_1 \in Occ(X_\ell, Y)$, $k_2 \in Occ^{\triangle}(X, Y)$ and $k_3 \in Occ(X_r, Y)$.

Lemma 3 ([8]) *For any strings $X, Y_1, Y_2 \in \Sigma^*$ and integers $k_1, k_2 \in \mathcal{N}$, the intersection $Occ^{\uparrow}(X, Y_1, k_1) \cap (Occ^{\uparrow}(X, Y_2, k_2) \ominus |Y_1|)$ can be computed in $O(1)$ time, provided that $Occ^{\uparrow}(X, Y_1, k_1)$ and $Occ^{\uparrow}(X, Y_2, k_2)$ are already computed.*

For variables $X = X_\ell X_r$ and Y, we denote $Occ^{\triangle}(X, Y) = Occ^{\uparrow}(X, Y, |X_\ell| + 1)$. The following observation is explained in Figure 2.

Observation 3 ([18]) *For any variables $X = X_\ell X_r$ and Y,*

$$Occ(X, Y) = Occ(X_\ell, Y) \cup Occ^{\triangle}(X, Y) \cup (Occ(X_r, Y) \oplus |X_\ell|).$$

Observation 3 implies that $Occ(X_n, Y)$ can be represented by a combination of

$$\{Occ^{\triangle}(X_i, Y)\}_{i=1}^{n} = Occ^{\triangle}(X_1, Y), Occ^{\triangle}(X_2, Y), \ldots, Occ^{\triangle}(X_n, Y).$$

Thus, the desired output $Occ(T, P) = Occ(X_n, Y_m)$ can be expressed as a combination of $\{Occ^{\triangle}(X_i, Y_m)\}_{i=1}^{n}$ that requires $O(n)$ space. Hereby, computing $Occ(T, P)$ is reduced to computing $Occ^{\triangle}(X_i, Y_m)$ for every $i = 1, 2, \ldots, n$. In computing each $Occ^{\triangle}(X_i, Y_j)$ recursively, the same set $Occ^{\triangle}(X_{i'}, Y_{j'})$ might repeatedly be referred to, for $i' < i$ and $j' < j$. Therefore we take the dynamic programming strategy. We use an $m \times n$ table *App* where each entry *App*$[i, j]$ at row i and column j stores the triple for $Occ^{\triangle}(X_i, Y_i)$. We compute each *App*$[i, j]$ in a bottom-up manner, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. In the following sections, we will show that the whole table *App* can be computed in $O(h^2 + mn \log s)$ time using $O(h^2 + mn)$ space, where h is the number of simple variables in $\mathcal{T}$ and s is the number of complex variables in $\mathcal{T}$. This leads to the result of Theorem 1.

4 Details of algorithm

In this section, we show how to compute each $Occ^{\triangle}(X_i, Y_j)$ efficiently. Our result is as follows:

Lemma 4 *For any variables X_i of $\mathcal{T}$ and Y_j of $\mathcal{P}$, $Occ^{\triangle}(X_i, Y_j)$ can be computed in $O(\log s)$ time, with extra $O(h^2 + mn)$ work time and space.*

The key to prove this lemma is, given integer k, to pre-compute $Occ^{\uparrow}(X_{i'}, Y_{j'}, k)$ for any $1 \le i' < i$ and $1 \le j' < j$. In case that X is simple, we have the following lemma:

Lemma 5 *Let X be any simple variable of $\mathcal{T}$ and Y be any variable of $\mathcal{P}$. Given integer $k \in \mathcal{N}$, $Occ^{\uparrow}(X, Y, k)$ can be computed in $O(1)$ time, with extra $O(h^2 + mh)$ work time and space.*

Proof. Let $b = k - |Y|$ and $e = k + |Y| - 1$. Let X_b denote any descendant of X for which the beginning position of X_b in X is b. Similarly, let X_e denote any descendant of X for which the ending position of X_e in X is e. That is, $X[b : b + |X_b| - 1] = X_b$ and $X[e - |X_e| + 1 : e] = X_e$.

(1) when $|X_b| \ge |X_e|$. In this case we have

$$\begin{aligned} Occ^{\uparrow}(X, Y, k) &= Occ^{\uparrow}(X, Y, b + |Y|) \\ &= Occ^{\uparrow}(X_b, Y, |Y| + 1) \oplus (b - 1). \end{aligned}$$

(2) when $|X_b| < |X_e|$. In this case we have

$$\begin{aligned} Occ^{\uparrow}(X, Y, k) &= Occ^{\uparrow}(X, Y, e - |Y| + 1) \\ &= Occ^{\uparrow}(X_e, Y, |Y| + |X_e| + 1) \oplus (e - |X_e|). \end{aligned}$$

Let us now consider how to compute $Occ^{\uparrow}(X_b, Y, |Y|+1)$ in case (1). $Occ^{\uparrow}(X_e, Y, |Y| + |X_e| + 1)$ in case (2) can be computed similarly. Let $X_b = X_\ell X_r$. Depending on the type of X_b, we have the two following cases:

(i) when X_b is right simple (see Figure 3, left).
Let $\langle p, d, t \rangle = Occ^{\uparrow}(X_\ell, Y, |Y| + 1)$.

$$\begin{aligned} & Occ^{\uparrow}(X_b, Y, |Y| + 1) \\ &= \begin{cases} \langle p, d', t+1 \rangle & \text{if } |X_b| - |Y| + 1 \in Occ^{\triangle}(X_b, Y) \text{ and } |X_b| \le 2|Y|, \\ \langle p, d, t \rangle & \text{otherwise,} \end{cases} \\ & \text{where } d' = \begin{cases} 0 & \text{if } t = 0, \\ p - 1 & \text{if } t = 1, \\ d & \text{if } t > 1. \end{cases} \end{aligned}$$

(ii) when X_b is left simple (see Figure 3, right).
Let $\langle p, d, t \rangle = Occ^{\uparrow}(X_r, Y, |Y| + 1)$.

- when $t = 0$.

$$Occ^{\uparrow}(X_b, Y, |Y| + 1) = \begin{cases} \langle 1, 0, 1 \rangle & \text{if } 1 \in Occ^{\triangle}(X_b, Y), \\ \emptyset & \text{otherwise.} \end{cases}$$

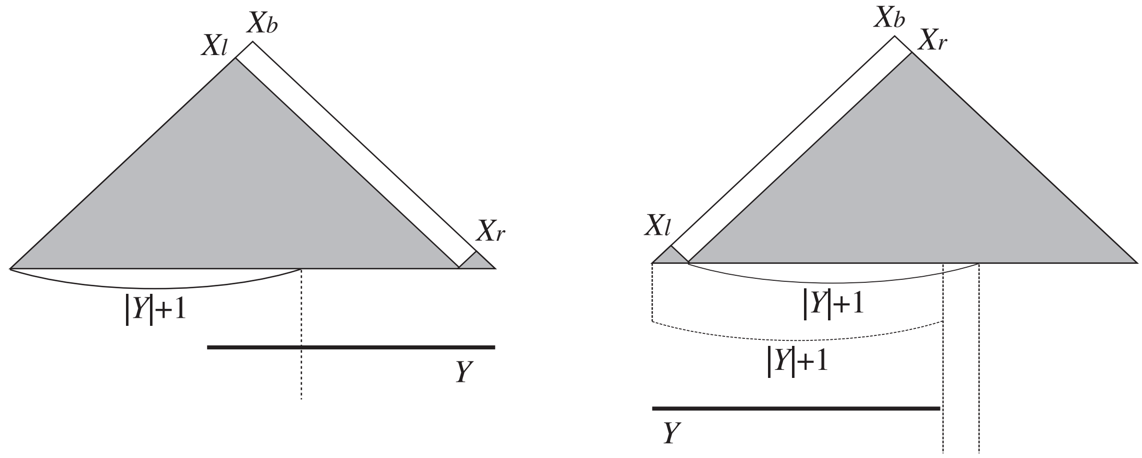


Figure 3: Two cases for $Occ^{\uparrow}(X_b, Y, |Y|+1)$, where $X_b = X_\ell X_r$. To the left is the case where $|X_r| = 1$, and to the right is the case where $|X_\ell| = 1$.

- when $t \geq 1$. Let $q = p + (t-1)d$.

$$Occ^{\uparrow}(X_b, Y, |Y|+1)$$
$$= \begin{cases} \langle p+1, d, t\rangle & \text{if } q < |Y|+1 \text{ and } 1 \notin Occ^{\triangle}(X_b, Y), \\ \langle 1, d', t+1\rangle & \text{if } q < |Y|+1 \text{ and } 1 \in Occ^{\triangle}(X_b, Y), \\ \langle p+1, d, t-1\rangle & \text{if } q = |Y|+1 \text{ and } 1 \notin Occ^{\triangle}(X_b, Y), \\ \langle 1, d', t\rangle & \text{if } q = |Y|+1 \text{ and } 1 \in Occ^{\triangle}(X_b, Y), \end{cases}$$

$$\text{where } d' = \begin{cases} p & \text{if } t = 1, \\ d & \text{if } t > 1. \end{cases}$$

Checking whether $|Y|+1 \in Occ^{\triangle}(X_b, Y)$ and whether $1 \in Occ^{\triangle}(X_b, Y)$ can be done in $O(1)$ time since $Occ^{\triangle}(X_b, Y)$ forms a single arithmetic progression by Lemma 1. We here take the dynamic programming strategy. We use an $h \times m$ matrix R where each entry $R[i, j]$ at row i and column j stores the triple representing $Occ^{\uparrow}(X_i, Y_j, |Y_j|+1)$. We compute each $R[i, j]$ in a bottom-up manner, for $i = 1, \ldots, h$ and $j = 1, \ldots, m$. Each $R[i, j]$ can be computed in $O(1)$ time as shown above. Also, each $R[i, j]$ requires $O(1)$ space by Lemma 1. Hence we can construct the whole table R in $O(mh)$ time and space.

Now we show that it is possible to find X_b in constant time after an $O(h^2)$ time preprocessing. We use an $h \times h$ matrix Beg in which each row i corresponds to simple variable X_i, and each column j corresponds to each position j in X_i. Each entry of the matrix stores the following information:

$$Beg[i:j] \quad = \quad \begin{cases} X_j & \text{if } X_i[j : j + |X_j| - 1] = X_j \text{ for some } X_j, \\ nil & \text{otherwise.} \end{cases}$$

For our purpose, X_j can be any simple variable satisfying the condition. To be specific, however, we use the smallest possible variable as X_j. By Proposition 1, for any simple variable X_i we have $|X_i| = \|X_i\|$. Thus finding the smallest variable corresponding to each position in X_i is feasible in $O(\|X_i\|)$ time in total. Therefore, matrix Beg can

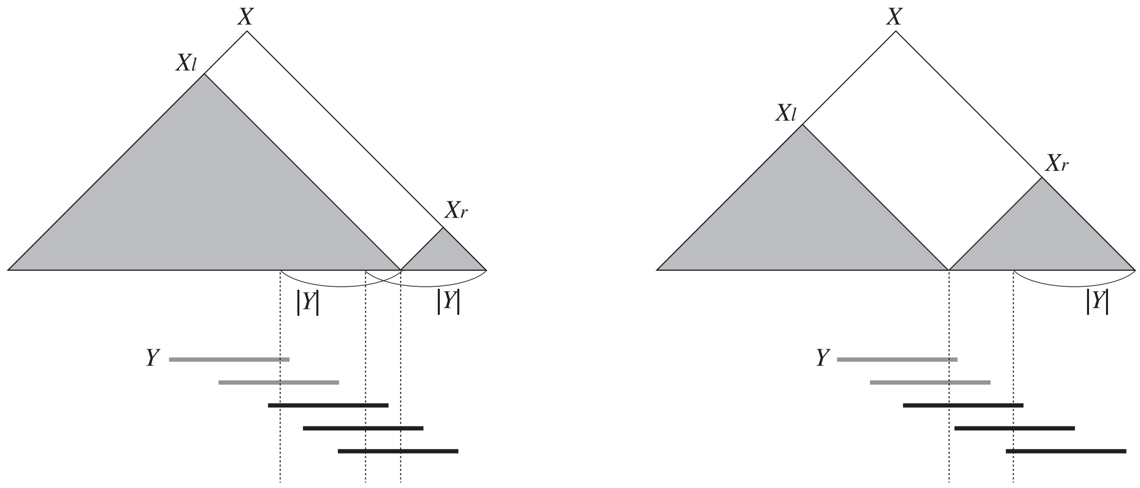


Figure 4: Two cases for $Occ^{\uparrow}(X, Y, |X| - |Y| + 1)$. To the left is the case where $|X_r| \leq |Y|$, and to the right is the case where $|X_r| > |Y|$.

be computed in $O(h^2)$ time and space. Once having these *Beg* computed, for any position b with respect to X, we can retrieve X_b in constant time.

In total, the extra time and space requirement is $O(h^2 + mh)$. This completes the proof. □

As a counterpart to Lemma 5 with respect to simple variables, in case that X is complex we have the following lemma:

Lemma 6 *Let X be any complex variable of $\mathcal{T}$ and Y be any variable of $\mathcal{P}$. Given integer $k \in \mathcal{N}$, $Occ^{\uparrow}(X, Y, k)$ can be computed in $O(\log s)$ time with extra $O(ms)$ work time and space.*

To prove Lemma 6 above, we need to establish Lemma 7 and Lemma 8 below.

Lemma 7 *Let $X = X_\ell X_r$ be any complex variable of $\mathcal{T}$ and let Y be any variable of $\mathcal{P}$. Assume $Occ^{\uparrow}(X_\ell, Y, |X_\ell| - |Y| + 1)$ and $Occ^{\triangle}(X, Y)$ are already computed. Then $Occ^{\uparrow}(X, Y, |X| - |Y| + 1)$ can be computed in $O(1)$ time, with extra $O(ms)$ work space.*

Proof. Let $A = Occ^{\uparrow}(X, Y, |X| - |Y| + 1)$. Depending on the length of X_r with respect to the length of Y, we have the following cases:

(1) when $|X_r| \leq |Y|$ (Figure 4, left).
In this case, it stands that:

$$A = (Occ^{\uparrow}(X_\ell, Y, |X_\ell| - |Y| + 1) \cap [|X| - 2|Y| + 1, |X| - |Y| + 1]) \cup Occ^{\triangle}(X, Y).$$

(2) when $|X_r| > |Y|$ (Figure 4, right).
In this case, it stands that:

$$A = (Occ^{\triangle}(X, Y) \cap [|X| - 2|Y| + 1, |X_\ell| + 1]) \cup Occ^{\uparrow}(X_r, Y, |X_r| - |Y| + 1).$$

Due to Lemma 5, $Occ^{\uparrow}(X_r, Y, |X_r| - |Y| + 1)$ of case (2) can be computed in constant time since X_r is simple. By Lemma 1 and Observation 2 the union operations can be done in $O(1)$ time.

What remains is how to compute $Occ^{\uparrow}(X_\ell, Y, |X_\ell| - |Y| + 1)$ in case (1). We construct an $s \times m$ matrix where each entry at row i and column j stores the triple representing $Occ^{\uparrow}(X_i, Y_j, |X_i| - |Y_j| + 1)$ where X_i is a complex variable. Using this matrix, $Occ^{\uparrow}(X_\ell, Y, |X_\ell| - |Y| + 1)$ of case (1) can be referred to in constant time. Each entry takes $O(1)$ space by Lemma 1, and thus the whole matrix requires $O(ms)$ space. This matrix can be constructed in $O(ms)$ time. $\square$

For any complex variable $X = X_\ell X_r$, let $range(X)$ denote the range $[r_1, r_2]$ such that $T[r_1, r_2] = X_r$. It is clear that for each complex variable its range is uniquely determined, since each complex variable appears in $\mathcal{T}$ exactly once.

Lemma 8 *Given integer $k \in \mathcal{N}$, we can retrieve in $O(\log s)$ time the complex variable X such that $range(X) = [r_1, r_2]$ and $r_1 \leq k \leq r_2$, after a preprocessing taking $O(s)$ time and space.*

Proof. We construct a balanced binary search tree where each node consists of a pair of a complex variable and its range. The sequence of complex variables $X_{i_1}, X_{i_2}, \ldots, X_{i_s}$ corresponds to $range(X_{i_1}), range(X_{i_2}), \ldots, range(X_{i_s}) = [1, |X_{i_1}|], [|X_{i_1}| + 1, |X_{i_2}|], \ldots [|X_{i_{s-1}}| + 1, |X_{i_s}|]$. This means that the ranges are already sorted in decreasing order. Therefore, we can construct a balanced binary search tree in $O(s)$ time and space.

Given integer k, at each node of the balanced tree corresponding to some variable X, we examine whether $k \in range(X) = [r_1, r_2]$. If $r_1 \leq k \leq r_2$, X is the desired variable. If $k < r_1$, we take the left edge of the node. If $k > r_2$, we take the right edge of the node. This way we can retrieve the desired complex variable in $O(\log s)$ time. $\square$

We are now ready to prove Lemma 6 as follows.

Proof. Let $A = Occ^{\uparrow}(X, Y, k)$. Let X_{ℓ_1} be the complex variable such that $k \in range(X_{\ell_1})$, and let $X_{\ell_1} = X_{\ell(\ell_1)} X_{r(\ell_1)}$. Let X_{ℓ_2} be the complex variable satisfying $k - |Y| \in range(X_{\ell_2})$, and let $X_{\ell_2} = X_{\ell(\ell_2)} X_{r(\ell_2)}$. There are the three following cases:

(1) when $k - |Y| \geq |X_{\ell(\ell_1)}| + 1$ and $k + |Y| - 1 \leq |X_{\ell_1}|$ (Figure 5, left).
In this case, we have $A = Occ^{\uparrow}(X_{r(\ell_1)}, Y, k) \oplus |X_{\ell(\ell_1)}|$.

(2) when $k - |Y| < |X_{\ell(\ell_1)}| + 1$ and $k + |Y| - 1 \leq |X_{\ell_1}|$ (Figure 5, right).
In this case, we have

$$A = (Occ^{\triangle}(X_{\ell_1}, Y) \cap [k - |Y|, X_{\ell(\ell_1)} + 1]) \cup (Occ^{\uparrow}(X_{r(\ell_1)}, Y, k) \oplus |X_{\ell(\ell_1)}|).$$

(3) when $k + |Y| - 1 > |X_{\ell_1}|$ (Figure 6).
In this case, we have

$$\begin{aligned} A \;=\; & (Occ^{\uparrow}(X_{\ell(\ell_2)}, Y, |X_{\ell(\ell_2)}| - |Y| + 1) \cap [k - |Y|, |X_{\ell(\ell_2)}| - |Y| + 1]) \\ & \cup (Occ^{\triangle}(X_{\ell_2}, Y) \cap [|X_{\ell(\ell_2)}| - |Y| + 1, k]). \end{aligned}$$

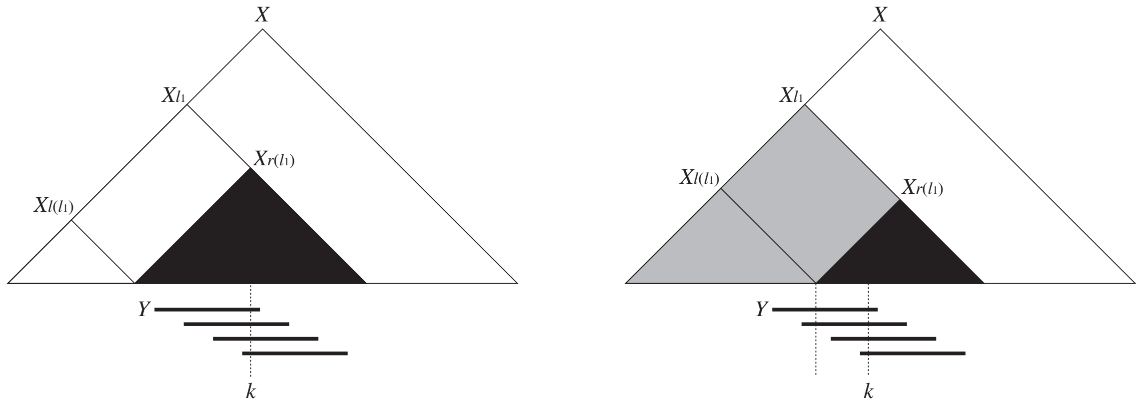


Figure 5: In the left case, all the occurrences are covered by $Occ^{\uparrow}(X_{r(\ell_1)}, Y, k) \oplus |X_{\ell(\ell_1)}|$. In the right case, the first and second occurrences are covered by $Occ^{\triangle}(X_{\ell_1}, Y)$ and the third and fourth occurrences by $Occ^{\uparrow}(X_{r(\ell_1)}, Y, k) \oplus |X_{\ell(\ell_1)}|$.

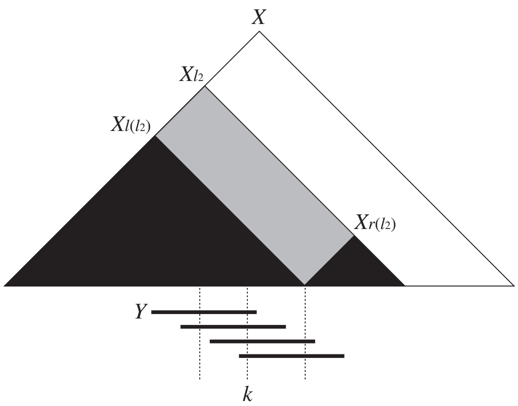


Figure 6: In this case, the first and second occurrences are covered by $Occ^{\uparrow}(X_{\ell(\ell_2)}, Y, |X_{\ell(\ell_2)}| - |Y| + 1)$ and the third and fourth occurrences are covered by $Occ^{\triangle}(X_{\ell_2}, Y)$.

Due to Lemma 8, X_{ℓ_1} and X_{ℓ_2} can be found in $O(\log s)$ time. Since $X_{r(\ell_1)}$ is simple, $Occ^{\uparrow}(X_{r(\ell_1)}, Y, k)$ of cases (1) and (2) can be computed in $O(1)$ time by Lemma 5. According to Lemma 7, $Occ^{\uparrow}(X_{\ell(\ell_2)}, Y, |X_{\ell(\ell_2)}| - |Y| + 1)$ of case (3) can be computed in $O(1)$ time. By Observation 2, the union operations can be done in $O(1)$ time. Thus, in any case $A = Occ^{\uparrow}(X, Y, k)$ can be computed in $O(\log s)$ time. By Lemma 7 and Lemma 8, the extra work time and space are $O(ms)$. This completes the proof. □

Now we have got Lemma 5 and Lemma 6 proved. Using these lemmas, we can prove Lemma 4 as follows:

Proof. Let $X_i = X_\ell X_r$ and $Y_j = Y_\ell Y_r$. Then, as seen in Figure 7, we have

$$\begin{aligned} Occ^{\triangle}(X_i, Y_j) &= (Occ^{\triangle}(X_i, Y_\ell) \cap (Occ(X_r, Y_r) \oplus |X_\ell| \ominus |Y_\ell|)) \\ &\quad \cup (Occ(X_\ell, Y_\ell) \cap (Occ^{\triangle}(X_i, Y_r) \ominus |Y_\ell|)). \end{aligned}$$

Let $A = Occ^{\triangle}(X_i, Y_\ell) \cap (Occ(X_r, Y_r) \oplus |X_\ell| \ominus |Y_\ell|)$ and $B = Occ(X_\ell, Y_\ell) \cap$

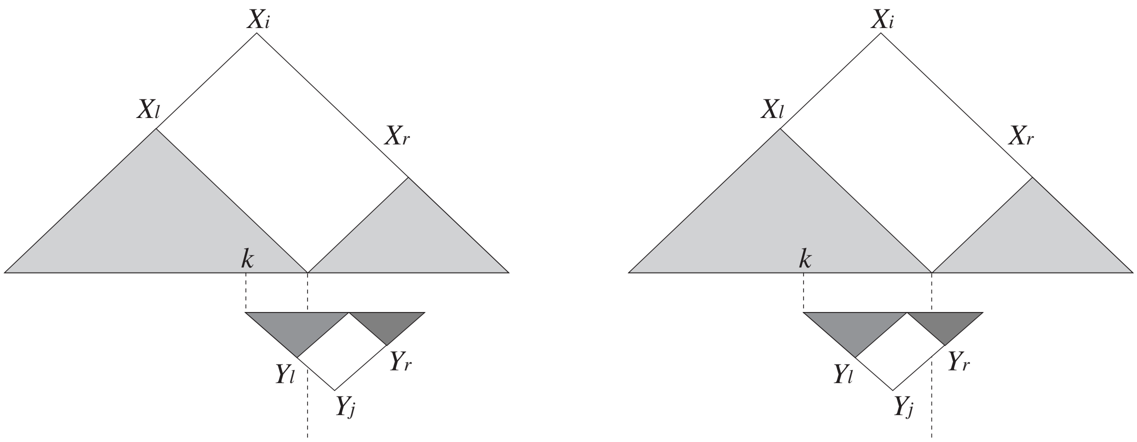


Figure 7: $k \in Occ^{\triangle}(X,Y)$ if and only if either $k \in Occ^{\triangle}(X,Y_\ell)$ and $k+|Y_\ell| \in Occ(X,Y_r)$ (left case), or $k \in Occ(X,Y_\ell)$ and $k+|Y_\ell| \in Occ^{\triangle}(X,Y_r)$ (right case).

$(Occ^{\triangle}(X_i,Y_r) \ominus |Y_\ell|)$. Since $Occ^{\triangle}(X_i,Y_j)$ forms a single arithmetic progression by Lemma 1, the union operation of $A \cup B$ can be done in constant time. Therefore, the key is how to compute A and B efficiently.

Now we show how to compute set A. Let $z = |X_\ell| - |Y_\ell|$. Let $\langle p_1, d_1, t_1 \rangle = Occ^{\triangle}(X_i, Y_\ell)$ and $q_1 = p_1 + (t_1 - 1)d_1$. Depending on the value of t_1, we have the following cases:

(1) when $t_1 = 0$.
In this case we have $A = \emptyset$.

(2) when $t_1 = 1$.
In this case, $Occ^{\triangle}(X_i,Y_\ell) = \{p_1\}$. It stands that

$$\begin{aligned}
A &= \{p_1\} \cap (Occ(X_r,Y_r) \oplus z) \\
&= (\{p_1 - z\} \cap Occ(X_r,Y_r)) \oplus z) \\
&= (\{p_1 - z\} \cap [p_1 - z - |Y_r|, p_1 - z] \cap Occ(X_r,Y_r)) \oplus z) \\
&= (\{p_1 - z\} \cap Occ^{\uparrow}(X_r,Y_r,p_1 - z)) \oplus z) \quad \text{(By Observation 1)} \\
&= \begin{cases} \{p_1\} & \text{if } p_1 - z \in Occ^{\uparrow}(X_r,Y_r,p_1 - z), \\ \emptyset & \text{otherwise.} \end{cases}
\end{aligned}$$

Since X_r is simple, $Occ^{\uparrow}(X_r,Y_r,p_1 - z)$ can be computed in constant time by Lemma 5. Checking whether $p_1 - z \in Occ^{\uparrow}(X_r,Y_r,p_1 - z)$ or not can be done in constant time since $Occ^{\uparrow}(X_r,Y_r,p_1 - z)$ forms a single arithmetic progression by Lemma 1.

(3) when $t_1 > 1$.

There are two sub-cases depending on the length of Y_r with respect to $q_1 - p_1 = (t_1 - 1)d_1 \geq d_1$, as follows.

- when $|Y_r| \geq q_1 - p_1$ (see the left of Figure 8). By this assumption, we have

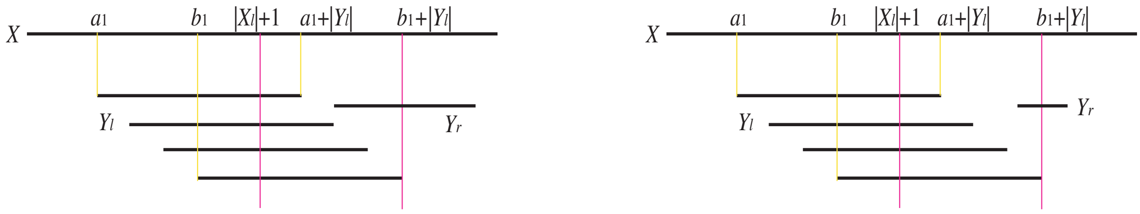


Figure 8: Long case (left) and short case (right).

$q_1 - |Y_r| \leq p_1$, which implies $[p_1, q_1] \subseteq [q_1 - |Y_r|, q_1]$. Thus

$$\begin{aligned} A &= \langle p_1, d_1, t_1 \rangle \cap (Occ(X_r, Y_r) \oplus z) \\ &= (\langle p_1, d_1, t_1 \rangle \cap [p_1, q_1]) \cap (Occ(X_r, Y_r) \oplus z) \\ &= (\langle p_1, d_1, t_1 \rangle \cap [q_1 - |Y_r|, q_1]) \cap (Occ(X_r, Y_r) \oplus z) \\ &= \langle p_1, d_1, t_1 \rangle \cap ([q_1 - |Y_r|, q_1] \cap (Occ(X_r, Y_r) \oplus z)) \\ &= \langle p_1, d_1, t_1 \rangle \cap (([q_1 - |Y_r| - z, q_1 - z] \cap Occ(X_r, Y_r)) \oplus z) \\ &= \langle p_1, d_1, t_1 \rangle \cap (Occ^{\uparrow}(X_r, Y_r, q_1 - z) \oplus z), \end{aligned}$$

where the last equality is due to Observation 1. Since X_r is simple, due to Lemma 5, $Occ^{\uparrow}(X_r, Y_r, q_1 - z)$ can be computed in $O(1)$ time. By Lemma 3, $\langle p_1, d_1, t_1 \rangle \cap (Occ^{\uparrow}(X_r, Y_r, q_1 - z) \ominus |Y_\ell|)$ can be computed in constant time.

- when $|Y_r| < q_1 - p_1$ (see the right of Figure 8). The basic idea is the same as the previous case, but computing $Occ^{\uparrow}(X_r, Y_r, q_1 - z)$ is not enough, since $|Y_r|$ is 'too short'. However, we can fill up the gap as follows.

$$\begin{aligned} A &= \langle p_1, d_1, t_1 \rangle \cap (Occ(X_r, Y_r) \oplus z) \\ &= (\langle p_1, d_1, t_1 \rangle \cap [p_1, q_1]) \cap (Occ(X_r, Y_r) \oplus z) \\ &= (\langle p_1, d_1, t_1 \rangle \cap ([p_1, q_1 - |Y_r| - 1] \cup [q_1 - |Y_r|, q_1])) \cap (Occ(X_r, Y_r) \oplus z) \\ &= \langle p_1, d_1, t_1 \rangle \cap (S \cup Occ^{\uparrow}(X_r, Y_r, q_1 - z)) \oplus z), \\ &\qquad \text{where } S = [p_1 - z, q_1 - z - |Y_r| - 1] \cap Occ(X_r, Y_r). \end{aligned}$$

By Lemma 2, d_1 is the shortest period of $X_i[p_1 : q_1 + |Y_\ell| - 1]$. For this string, we have

$$\begin{aligned} & X_i[p_1 : q_1 + |Y_\ell| - 1] \\ =\ & X_\ell[p_1 : |X_\ell|] X_r[1 : q_1 + |Y_\ell| - 1 - |X_\ell|] \\ =\ & X_\ell[p_1 : |X_\ell|] X_r[1 : q_1 - z - 1] \\ =\ & X_\ell[p_1 : |X_\ell|] X_r[1 : p_1 - z - 1] X_r[p_1 - z : q_1 - z - 1] \\ =\ & X_i[p_1 : p_1 + |Y_\ell| - 1] X_r[p_1 - z : q_1 - z - 1]. \end{aligned}$$

Therefore, $X_r[p_1 - z : q_1 - z - 1] = u^{t_1}$ where u is the suffix of Y_ℓ of length d_1. Thus,

$$S = \begin{cases} \langle p_1 - z, d_1, t' \rangle & \text{if } p_1 - z \in Occ(X_r, Y_r), \\ \emptyset & \text{otherwise}, \end{cases}$$

where t' is the maximum integer satisfying $p_1 - z + (t'-1)d_1 \leq q_1 - z - |Y_r| - 1$. According to Observation 2, the union operation of $S \cup Occ^{\uparrow}(Xr, Y_r, q_1 - z)$ can be done in constant time in both cases. By Observation 1, checking whether $p_1 - z \in Occ(X_r, Y_r)$ or not can be reduced to checking if $p_1 - z \in Occ^{\uparrow}(X_r, Y_r, p_1 - z)$. Since X_r is simple, it can be done in $O(1)$ time by Lemma 1 and Lemma 5. Finally, the intersection operation can be done in constant time by Lemma 3.

Therefore, in any case we can compute A in constant time.

Now we consider computing $B = Occ(X_\ell, Y_\ell) \cap (Occ^{\triangle}(X_i, Y_r) \ominus |Y_\ell|)$. Let $\langle p_2, d_2, t_2 \rangle = Occ^{\triangle}(X_i, Y_r)$. We now have to consider how to compute $Occ^{\uparrow}(X_\ell, Y_\ell, p_2 - |Y_\ell|)$ efficiently. When X_ℓ is simple, we can use the same strategy as computing A. In case where X_ℓ is complex, $Occ^{\uparrow}(X_\ell, Y_\ell, p_2 - |Y_\ell|)$ can be computed in $O(\log s)$ time by Lemma 6.

Due to Lemma 5 and Lemma 6, the total extra work time and space are $O(h^2 + mh) + O(ms) = O(h^2 + m(h + s)) = O(h^2 + mn)$. This completes the proof. $\square$

We have proven that each $Occ^{\triangle}(X, Y)$ can be computed in $O(\log s)$ time with extra $O(h^2 + mn)$ work time and space. Thus, the whole time complexity is $O(h^2 + mn) + O(mn \log s) = O(h^2 + mn \log s)$, and the whole space complexity is $O(h^2 + mn)$. This leads to the result of Theorem 1.

5 Conclusions

Miyazaki et al. [18] presented an algorithm to solve the FCPM problem for straight line programs in $O(m^2 n^2)$ time and with $O(mn)$ space. Since simple collage systems can be translated to straight line programs, their algorithm gives us an $O(m^2 n^2)$ time solution to the FCPM problem for simple collage systems. In this paper we developed an FCPM algorithm for simple collage systems which runs in $O(\|\mathcal{D}\|^2 + mn \log |\mathcal{S}|)$ time using $O(\|\mathcal{D}\|^2 + mn)$ space. Since $n = \|\mathcal{D}\| + |\mathcal{S}|$, the proposed algorithm is faster than that of [18] which runs in $O(m^2 n^2)$ time.

An interesting extension of this research is to consider the FCPM problem for *composition systems* [24]. Composition systems can be seen as collage systems without repetitions. Since it is known that LZ77 compression can be translated into a composition system of size $O(n \log n)$, an efficient FCPM algorithm for composition systems would lead to a better solution for the FCPM problem with LZ77 compression. We remark that the only known FCPM algorithm for LZ77 compression takes $O((n + m)^5)$ time [6], which is still very far from desired optimal time complexity.

References

[1] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. DCC'92*, page 279. IEEE Computer Society, 1992.

[2] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *J. Computer and System Sciences*, 52(6):299–307, 1996.

[3] T. Eilam-Tzoreff and U. Vishkin. Matching patterns in strings subject to multilinear transformations. *Theoretical Computer Science*, 60:231–254, 1988.

[4] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.

[5] P. Gage. A new algorithm for data compression. *The C Users Journal*, 12(2), 1994.

[6] L. Gąsieniec, M. Karpinski, W. Plandowski, and W. Rytter. Efficient algorithms for Lempel-Ziv encoding (extended abstract). In *Proc. SWAT'96*, volume 1097 of *LNCS*, pages 392–403. Springer-Verlag, 1996.

[7] L. Gąsieniec and W. Rytter. Almost optimal fully LZW-compressed pattern matching. In *Proc. DCC'99*, pages 316–325. IEEE Computer Society, 1999.

[8] S. Inenaga, A. Shinohara, and M. Takeda. An efficient pattern matching algorithm for OBDD text compression. Technical Report DOI-TR-CS-222, Department of Informatics, Kyushu University, 2003.

[9] M. Karpinski, W. Rytter, and A. Shinohara. An efficient pattern-matching algorithm for strings with short descriptions. *Nord. J. Comput.*, 4(2):172–186, 1997.

[10] T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Collage system: a unifying framework for compressed pattern matching. *Theoretical Computer Science*, 298:253–272, 2003.

[11] J. Kieffer and E. Yang. Grammar-based codes: a new class of universal lossless source codes. *IEEE Trans. Inform. Theory*, 46(3):737–754, 2000.

[12] J. Kieffer and E. Yang. Grammar-based codes for universal lossless data compression. *Communications in Information and Systems*, 2(2):29–52, 2002.

[13] J. Kieffer, E. Yang, G. Nelson, and P. Cosman. Universal lossless compression via multilevel pattern matching. *IEEE Trans. Inform. Theory*, 46(4):1227–1245, 2000.

[14] J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proc. DCC'99*, pages 296–305. IEEE Computer Society, 1999.

[15] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Trans. Inf. Syst.*, 15(2):124–136, 1997.

[16] T. Matsumoto, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Bit-parallel approach to approximate string matching in compressed texts. In *Proc. SPIRE'00*, pages 221–228, 2000.

[17] S. Mitarai, M. Hirao, T. Matsumoto, A. Shinohara, M. Takeda, and S. Arikawa. Compressed pattern matching for SEQUITUR. In *Proc. DCC'01*, pages 469–480. IEEE Computer Society, 2001.

[18] M. Miyazaki, A. Shinohara, and M. Takeda. An improved pattern matching algorithm for strings in terms of straight line programs. *J. Discrete Algorithms*, 1(1):187–204, 2000.

[19] C. Nevill-Manning and I. Witten. Identifying hierarchical structure in sequences: a linear-time algorithm. *J. Artificial Intelligence Research*, 7:67–82, 1997.

[20] W. Rytter. Algorithms on compressed strings and arrays. In *Proc. SOFSEM'99*, volume 1725 of *LNCS*, pages 48–65. Springer-Verlag, 1999.

[21] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Speeding up pattern matching by text compression. In *Proc. CIAC'00*, volume 1767 of *LNCS*, pages 306–315. Springer-Verlag, 2000.

[22] J. A. Storer and T. G. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982.

[23] T. Welch. A technique for high performance data compression. *IEEE Comput. Magazine*, 17(6):8–19, 1984.

[24] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory*, 23:337–343, 1977.

[25] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. Inform. Theory*, 24:530–536, 1978.

Semi-Lossless Text Compression

Yair Kaufman and Shmuel T. Klein

Department of Computer Science
Bar Ilan University, Ramat-Gan 52900, Israel
Tel: (972–3) 531 8865 Fax: (972–3) 736 0498

e-mail: {kaufmay,tomi}@cs.biu.ac.il

Abstract. A new notion, that of semi-lossless text compression, is introduced, and its applicability in various settings is investigated. First results suggest that it might be hard to exploit the additional redundancy of English texts, but the new methods could be useful in applications where the correct spelling is not important, such as in short emails, and the new notion raises some interesting research problems in several different areas of Computer Science.

Keywords: text compression, lossy, lossless compression

1 Introduction

One widespread partition when coming to classify data compression methods is into lossless and lossy methods. *Lossless* methods include usually those applied on text files or other data for which no loss of information can be tolerated, *lossy* techniques are generally applied to image files as well as to video and audio data, for which the overall knowledge a user might extract does not seem significantly reduced even if a part of the data is omitted.

Even though most lossy compression methods include some lossless techniques as one of their components, the research methods and goals of the corresponding communities are in fact quite different. While researchers in text compression are primarily concerned with good compression performance (in terms of speed and of space, both of the file to be compressed and of the RAM required by the method at hand), a major topic in image compression is finding a good tradeoff between the size of the compressed file and the ability of a human observer to find the differences between the original picture and its partially reconstructed copy. Many articles about image compression include side by side two pictures looking almost identical, the one labeled “original” and the other labeled “compressed”. Obviously, the latter is rather the decompressed, reconstructed, image, the real compressed one consisting of a close to random sequence of zeros and ones, which would not yield any visual information when displayed as a raster file.

The basic idea behind lossy compression is thus the fact that even if not all of the available data is presented, the human brain can often make up for the missing parts and guess, at least partially, whatever has been omitted, so that overall one has the feeling that nothing has been lost. We try, in this paper, to transfer this paradigm also into the framework of text compression, to which usually only lossless techniques have been applied.

A hint to the fact that strict losslessness might be relaxed can be found by anybody who tries to read a newspaper, and mostly succeeds in understanding all the required information in spite of occasional typing errors and other mistakes. It turns out that we are able to understand English text even if there are many more errors, as suggested by the following paragraph, which circulated recently on the Internet[1]

> Aoccdrnig to a rscheearch at Cmabrigde Uinervtisy, it deosn't mttaer in waht oredr the ltteers in a wrod are in; the olny iprmoetnt tihng is taht frist and lsat ltteer be at the rghit pclae. The rset can be a toatl mses and you can sitll raed it wouthit porbelm. Tihs is bcuseae the huamn mnid deos not raed ervey lteter by istlef, but the wrod as a wlohe.

If indeed it is true that under certain constraints the exact letter order can be altered without impairing our understanding of the information contained in English text, it follows that the order of the characters induced by English grammar and syntax may contain more redundancy than one thought so far, and eliminating this redundancy might yield improved compression. Being a hybrid of the two classes of compression methods mentioned above, we call the type of compression suggested by these ideas *semi-lossless*: the original text will not be fully reconstructed, just as a decompressed JPEG image is not identical to the original, and thereby the method will be lossy; on the other hand, again similarly to the decompressed image for which our eyes and brain fill in the omitted parts, here it is the knowledge of English that will enable the extraction of the full information of the original text, so that at least from the information point of view, if not from the physically stored file, the method can be considered as lossless.

A priori, the expexted gain from playing with the order of the characters within a word is not very large, as the average word in English is rather short (about 5 characters). The applicability of semi-lossless text compression might thus be restricted, most users preferring to get a clean text, even at the price of marginally lower compression. The new methods could therefore be useful in applications where the correct spelling is not important, such as in short emails or SMS notes sent between cellular phones, which already use some widely known shortcuts (that R acceptd by any 1). Moreover, the new notion raises many interesting research problems, some of which mentioned in the sequel, which may find applications in several different areas of Computer Science.

In the next section we suggest some approaches to semi-lossless text compression and discuss their usefulness as general data compressors. Section 3 then brings some preliminary experimental results, and we conclude in Section 4 with some possible extensions of this work.

[1] See, e.g., http://csunx4.bsc.edu/bmyers/language.htm, but there are dozens of pointers to this or similar phrases

2 Semi-lossless text compression techniques

Lossy text compression has already been suggested by Witten at al. in [9], which includes several quite amusing examples. We shall, however, concentrate on methods in which there is, at least a priori, no loss of information, and take the rule cited in the quotation in the introduction as a premise, namely, that if the first and last letters of the printed words are left in place, the remaining letters within each word can appear in any order.

This "law" is clearly not universal and relies on the assumption that the reader has a good knowledge of English. We are not concerned with checking the validity of this assumption, nor with suggesting alternative rules. This would rather fall into a domain investigated by psychologists, and the interested reader is referred to the vast literature dealing with several aspects of this subject, see, e.g., [2, 6] and the pointers appearing in their references. For our discussion it does not even really matter whether the given rule is true, or whether it should be reinforced (leaving the first, last and one or more additional letters in place) or could be relaxed (fixing only the first letter, or even none, allowing any permutation of any word). All we assume is that some rule exists according to which not all the characters of a text have to be restored to their original position for the text to be understandable. Such a rule will obviously depend on and vary according to language, potential readers and genre and type of the given text.

Taking therefore the quoted law (first and last letters fixed, the rest in any other position) as working assumption, it suggests the following generic compression and decompression algorithm:

Compression:

1. Process the words sequentially and if the current word is not special (number, proper name, etc.), do
2. keep first and last letters in place, but rearrange the others into "special" order;
3. apply some encoder on the rearranged text.

Decompression:

1. Decode the compressed words sequentially, and if the current word is not special, do
2. keep first and last letters in place; choose a random permutation of the other letters and send them to output.

Since the order of the characters (except the first and last of each word) is not restricted, it might be useful to choose a *special* order referred to in Step 2 of the compression, that will subsequently improve the encoding mentioned in Step 3. The reason for Step 2 of the decompression process is avoiding a constant bias introduced by the suggested partial order. It might be that seeing always the same permutations according to the special order chosen may interfere with our ability to recover the original word. Introducing the randomization restores for the reader the feeling of arbitrariness which, possibly, is necessary for correct decoding.

In the following sub-section, we explore some of the possibilities for choosing such a special order.

2.1 Choosing a special order of the characters

One possibility that comes to mind is arranging these letters in alphabetic order. The reason such a strategy is expected to improve compression is similar to the argument showing why the Burrows-Wheeler Transform (BWT) [1] actually works so well.

The BWT works on a string of length n and applies all the n cyclic rotations on it, yielding an $n \times n$ matrix which is then lexicographically sorted by rows. The first column of the sorted matrix is thus sorted, but BWT stores the *last* column of the matrix, which together with a pointer to the index of the original string in the matrix lets the file to be recovered. The last column is usually not sorted, but it corresponds to sorted contexts, and is therefore often very close to be sorted, which is why it is more compressible than the original string. The compression scheme based on BWT uses a move-to-front strategy to exploit this nearly sorted nature of the string to be compressed. Returning to our problem, if the characters in each word can be arranged alphabetically, this may similarly yield improved compression using move-to-front and/or run-length coding if the strings are long enough.

Another possibility would be to arrange the characters by frequency. The distribution of characters in English text is well-known, (see, e.g., [3]), and sorting the letters following the order `E`, `T`, `A`, `O`, `N`, `I`, `S`, etc., increases the probability of short displacements in move-to-front schemes. However, frequency of occurrence alone does not take the tight connections between certain characters into consideration.

A more precise rule would therefore be trying to group the characters based on the probabilities of a given letter to appear after another one. A strict approach gets quickly into loops, for example, `E` is most likely followed by `R`, which in turn has `E` as its most probable successor. A simple greedy algorithm would thus be:

1. Start with an arbitrary character, x;
2. While not all characters are processed
   - Choose, among remaining characters, the successor s of x with highest probability;
   - $x \longleftarrow s$

Following the probabilities in [3], one possible sequence this may yield is:

`A N D E R O U T H I S P L Y M B J - X C K - F - G - Q - V - W - Z.`

While the beginning of this sequence seems reasonable, there are some evident shortcomings: `P` as successor of `S` is only the 9th choice, because the eight preceding ones (in order: `T`, `E`, `I`, `S`, `O`, `A`, `U` and `H`) all appeared earlier. Towards the end, all the remaining potential successors have probability practically zero, indicated by the dashes, so the choice is arbitrary. Note also that choosing the successor with highest probability might push the second best choice too far away. The second most frequent successor of `A` is `T`, which appears only in seventh position after `A`.

These speculations lead to the following formulation of the problem: we seek an ordering of the letters maximizing the overall probability of the letter successions. More formally, let $\Sigma = \{x_1, \ldots, x_n\}$ be the alphabet, and let $P[x, y]$ denote the probability of character y appearing as successor of x; we look for a permutation

$\sigma : \Sigma \longrightarrow \Sigma$ of the n characters, such that

$$\prod_{i=1}^{n-1} P[\sigma(i), \sigma(i+1)] \tag{1}$$

is maximized. The following transformation shows that this is in fact an instance of the Minimum Traveling Salesperson Problem. Consider a full graph $G = (V, V \times V)$, with $V = \Sigma$, and define the weight $w(x, y)$ of an edge (x, y) as

$$w(x, y) = -\log P[x, y].$$

Finding a permutation maximizing (1) is then equivalent to finding a Hamiltonian path of minimum weight in G. Unfortunately, this is an NP-complete problem, and since in our case, there is no reason to assume that the triangle inequality holds for the weights, it might even be hard to find a good approximation.

We now turn to the more technical details of choosing a specific compression scheme.

2.2 Choosing the compression technique

A simple statistical encoder, such as Huffman or arithmetic coding, applied independently to the individual characters will, of course, not yield any additional compression at all. The set of encoded characters remains the same, only their order is altered. To be able to take advantage of the partial reordering, a method is needed that takes previous characters into account.

A simple example would be run-length encoding, which is not likely to be useful. Run-length coding is widely used for images or fax-transmission, but in natural language text there are hardly any repeated strings of length longer than 2 (in German, there are some rare examples of runs of length 3, such as in `Schifffracht`). In our case, where the internal characters appear in sorted order, the lengths of runs are still limited by the number of times a given letter appears within a word. But the average word length, in English, is only about 5, so that no significant runs may be expected (German provides here again an extreme case: there is a street in Vienna named `Abrahamasantaclaragasse`, which would give a run of 9 `a`'s).

We may expect better performance when using Huffman or arithmetic coding in connection with a Markov model of order $k \geq 1$, meaning that each character is encoded as a function of the k characters preceding it. Though even natural text is well compressed by such a model as it captures many of its characteristic features (`q` followed by `u`, high probability for `e` following `th`, etc.), having identical characters grouped together may even cause better compression. However, the additional space requirements of higher order Markov models may be prohibitive.

Adaptive methods like Lempel-Ziv variants seem at first sight not applicable. In an adaptive encoding, the current item to be encoded relies on previously seen text, and if the item is not reliably restored, a subsequent pointer to it may give wrong results. Consider, for example, the string

$\cdots$ `a b c x y z c b a d e f w t w c e d f a v` $\cdots$

to be encoded by LZSS [4], and suppose that the whole string consists of internal characters (not the first or last in a word). The string `cba` can then be replaced by a pointer to the preceding `abc`, and `edfa` could point to `adef`, so that the modified LZSS encoding would be

$$\cdots \ \texttt{a b c x y z} \ (6,3) \ \texttt{d e f w t w c} \ (8,4) \ \texttt{v} \ \cdots .$$

But while the first pointer (6,3) would be decoded to `abc`, as expected, the second pointer (8,4) would now refer to the substring `cdef`, which is *not* a permutation of the original `edfa`. Note that the problem here is caused by the overlap between a substring, `cba`, that is replaced by an (*offset*, *length*) pointer, and a substring, `adef`, which is the target of such a pointer. In the absence of such overlaps, the encoding scheme works correctly.

One strategy to avoid the problem would thus be to forbid such overlaps, but this would affect compression efficiency. Another possibility is to adapt LZSS to work in this case, by keeping a copy of the currently decoded text, and search in it, rather than in the original text processed so far, for earlier occurrences of the current string to be encoded or its permutations. Returning to the example above, after having encoded `cba`, the processed string would look as

$$\cdots \ \texttt{a b c x y z a b c} \ \Big| \ \texttt{d e f w t w c e d f a v} \cdots$$

where the vertical bar indicates the current position, and to its left appears the reconstructed, rather than the original, text. While the bar now moves further to the right, the string `edfa` cannot be encoded as before. However, in this example, even a better substitution is possible, replacing `wcedf` by a pointer to `cdefw`, so that the encoded string finally looks as

$$\cdots \ \texttt{a b c x y z} \ (6,3) \ \texttt{d e f w t} \ (6,5) \ \texttt{a v} \ \cdots .$$

In fact, a correct algorithm based on LZSS is even more involved. Fast implementations of LZSS, like LZRW1 [8] or Microsoft's DoubleSpace [7] find the recurring strings by locating, using hashing, a previous occurrence of the character pair following the current position, and then extending the strings as far as possible by checking if the subsequent characters coincide. In our case, such a greedy approach may fail, e.g., for the string

$$\cdots \ \texttt{x y z t} \quad \texttt{a b c d e f g} \cdots \texttt{x z y t} \quad \texttt{a b e d c h k} \cdots .$$

The second occurrence of `ab` would point to the first one, but trying to extend the strings would fail in the first two attempts, `abe` and `abed` not matching `abc` and `abcd`, respectively, and only the third attempt would succeed, with `abedc` matching `abcde` modulo the reordering. Moreover, word boundaries have to be taken into account because of the constraint that first and last letters have to remain in place. The processing must therefore be by a combination of trying to extend partial matches by entire words and, once this fails, trying to match prefixes of the last word dealt with, proceeding backwards from the longest to the shorter ones. In the above example the word `xzyt` is first matched to `xyzt`, trying then to match `abedchk` to `abcdefg` fails, so we try to backtrack. `abedch` does not match `abcdef`, but `abedc` does match `abcde`, which gives the string `xzyt abedc` as required match.

Similar problems to those of LZSS would arise in LZ78 variants like LZW [5]. Instead of pointing to earlier strings in the already processed text, the compressed file consists of a series of pointers to an external dictionary, which is built on the fly. Here again, relaxing the rules and letting a pointer refer not necessarily to the string to be replaced, but possibly to any of its permutations, may yield some savings: the overall number of strings is reduced, implying that more good strings can be stored, or that the necessary pointers can be shorter. But as above, decoding may be erroneous, because the strings stored by LZW are overlapping, specifically, the last character of the nth stored string is also the first of the $n+1$st.

The problem may be more severe in this case, because eliminating one of the strings stored in an LZW dictionary will affect all the subsequent entries and therefore change all subsequent pointers, whereas for LZSS, all the changes are locally restricted.

2.3 Combining character ordering and compression technique

A different approach than trying to adapt Lempel-Ziv type methods would be to restrict ourselves to dealing with bigrams, trigrams, or generally, any k-grams with $k > 1$. Each word is considered on its own, and decomposed into a sequence of such consecutive k-grams, leaving, as before, the first and last letters in place. Special care is needed to deal with the last k-gram in this sequence within a word, which might require a smaller k. Then each k-gram is mapped to a representative, in a predetermined order (alphabetic, `ETAONI`, `ANDERO`, etc.). Finally, the items obtained by this decomposition are Huffman coded. Since the number of different k-grams is reduced from $|\Sigma|^k$ to $\binom{|\Sigma|}{k}$, a savings of about 50% for $k = 2$, and more for higher k, the average Huffman codeword lengths are expected to be lower. Moreover, the overhead of storing the different k-grams is also reduced.

An alternative would be to process the k-grams sequentially, without taking word boundaries into account. Each k-gram would again be mapped to a reordered one, but flag-bits would be used to indicate if there has been a reordering and which one. For bigrams, a single bit suffices to indicate whether to switch a pair, and the bit is needed only for those pairs following or preceding a space.

Block sorting using the BWT could also be adapted to our case. As mentioned earlier, the last column, which is the one stored by the algorithm, is almost sorted. Suppose we have a sequence of the form `A`, `A`, `A`, `B`, `A` $\cdots$ in this column. If we can change the order of the characters, we might want to remove the `B` from within the sequence of `A`s. Work on a general algorithm based on this idea is ongoing.

3 Experimental results

The first text chosen as testbed for the above semi-lossless algorithms consists of about 3MB of the AP newswire files from the TREC collection. In addition, the methods were applied to Mark Twain's *Tom Sawyer* taken from the Gutenberg Project. To avoid a bias introduced by punctuation and other signs, all non-alphabetic characters, except the space, have been removed, and all the others have been changed to upper case, giving an alphabet of size 27.

Table 1 summarizes some of the results. The first column gives the size of the raw files, the second after having applied simple Huffman coding on the individual letters. All compression figures are given in bits per character (bpc). The next columns deal with bigrams and trigrams, first in a standard fragmentation of the text into bi- or trigrams, then using the reordering for those k-grams that can be changed. For the bigrams the variant with the flag-bit has been applied, for the trigrams, triples including the first or last letter of a word have not been reordered. The figures include the overhead of storing the bi- or trigrams.

	size	Huffman	bigrams		trigrams	
			standard	ordered	standard	ordered
AP	2.57 Mb	4.148	3.791	3.707	3.529	3.437
Tom Sawyer	361 Kb	4.111	3.707	3.687	3.549	3.485

As can be seen, there is a slight improvement, though not a significant one. In fact, even with better parsing strategies than the simple one we used, one should not expect large savings for English text: the average word length being less than 5, and the two corner letters being fixed, the reordering will affect on the average less than 3 letters. However, with schemes going beyond word boundaries, like LZSS, or for other languages and other reordering rules, better results might be expected.

Note that there are far better compression schemes: applying Huffman coding on the basis of words, rather than characters, yields, for AP, 2.136 bpc, and if the internal letters of the words are reordered, 2.135 bpc, saving less than 0.05 percent. But such a scheme requires a large overhead for the storage of the Huffman tree, and can only be justified if the set of different words is stored anyway, e.g., as the dictionary in an Information Retrieval system.

4 Conclusions and future work

The main contribution of this paper is thus not the presentation of some novel compression technique, but rather the introduction of the notion of semi-lossless text compression and the ensuing research problems it raises in compression, pattern matching, computational linguistics and possibly other related areas. We have briefly explored how some of the known compression methods could be adapted to take advantage of the relaxed constraints and work currently on implementing some of the advanced methods.

Much work is still to be done. Here is a partial list of topics one might want to deal with:

- One could try to devise new methods that do not rely on adapting existing ones, but may possibly be totally different and specially adapted to our case.

- Different languages may suggest other rules. In *French*, grammatical suffixes are more abundant and often one or more of the last letters of a word are not even pronounced. Perhaps the rule of keeping specifically the last letter in place is then not adequate? *German* has the ability of concatenating several words into a single one; should the rule then be extended to fix also letters at sub-word boundaries, and how could these boundaries be detected? The average length of a word in *Finnish* is much longer than in English and double letters are more frequent.

- One could adapt ideas from other languages to English. For instance, *Hebrew* is generally written without vowels. This gives a large number of possible interpretations for each word, most of which are grammatically incorrect, but on the average, every word has four possible correct readings. Nevertheless, a native speaker has generally no trouble to pick the right choice quickly enough to read fluently, partly because certain consonants may act as vowels. It would not be reasonable to strip all the vowels from English texts (thgh ths wld gv gd cmprssn!), but perhaps one can devise rules to get rid of most of them, as we do anyway in speed-writing or when sending short electronic notes by computer or on cellular phones.

- Semi-lossless compression is not necessarily restricted to keeping a permutation of the original characters. When typing on cellular or regular phones, each key is assigned to several characters and the requested one is reached by repeatedly pressing the same key. It may be that the sets assigned to each key can be chosen in such a way that pressing only once, and thereby sending a representative of a small set, can still result in a text that is understandable. The size of Σ would be reduced, so one may save space, but also the time necessary to type a message will be greatly shortened.

Another short note many web-user got lately in their mail claimed that English spelling will shortly be simplified[2]. While this was meant as a joke, the idea is another nice example of how semi-lossless techniques could be implemented. The text suggested a five year plan during which many old spelling rules would be gradually abolished or modified, until

> after zis fifz yer, ve vil hav a reli sensibl riten styl. zer vil be no mor trubls or difikultis and evrivun vil find it ezi tu understand ech ozer.

While most of us will easily decipher the quote, note that its length (148 characters) is 14% shorter than its correctly spelled equivalent (172 characters), and the same 14% gain is also obtained if each of the messages is Huffman encoded (600 instead of 694 bits).

References

[1] Burrows M., Wheeler D.J., A block-sorting lossless data compression algorithm, Technical Report SRC 124, Digital Systems Research Center, Palo Alto, CA (1994).

[2] Friedmann N., Gvion A., Letter position dyslexia, *Cognitive Neuropsychology* **18**(8) (2001) 673–696.

[2] http://www.bluegum.com/Humour/Assorted/easier-english.htm

[3] Konheim A.G., *Cryptography, A Primer,* John Wiley & Sons, New York (1981).

[4] Storer J.A., Szymanski, T.G., Data compression via textual substitution, *J. ACM* **29** (1982) 928–951.

[5] Welch T.A., A technique for high performance data compression, *IEEE Computer,* **17** (1984) 8–19.

[6] Whitney C., How the brain encodes the order of letters in a printed word: The SERIOL model and selective literature review, *Psychonomic Bulletin & Review* **8**(2) (2001) 221–245.

[7] Whiting D.L., George G.A., Ivey G.E., Data Compression Apparatus and Method, U.S. Patent 5,126,739 (1992).

[8] Williams R.N., An extremely fast Ziv-Lempel data compression algorithm, *Proc. Data Compression Conference DCC–91*, Snowbird, Utah (1991) 362–371.

[9] Witten I.H., Bell T.C., Moffat A., Nevill-Manning C.G., Smith T.C., Thimbleby H., Semantic and generative models for lossy text compression, *The Computer Journal* **37**(2) (1994) 83–87.

Conditional Inequalities and the Shortest Common Superstring Problem

Uli Laube and Maik Weinard

Institut für Informatik
Johann Wolfgang Goethe-Universität Frankfurt am Main
Robert-Mayer-Straße 11-15
60054 Frankfurt am Main, Germany

e-mail: {laube,weinard}@thi.cs.uni-frankfurt.de

Abstract. We investigate the shortest common superstring problem (SCSSP). As SCSSP is APX-complete it cannot be approximated within an arbitrarily small performance ratio. One heuristic that is widely used is the notorious greedy heuristic. It is known, that the performance ratio of this heuristic is at least 2 and not worse than 4. It is conjectured that the greedy heuristic's performance ratio is in fact 2 (the greedy conjecture). Even the best algorithms introduced for SCSSP can only guarantee an upper bound of 2.5.

In [11] an even stronger version of the greedy conjecture is proven for a restricted class of orders in which strings are merged. We extend these results by broadening the class for which this stronger version can be established. We also show that the Triple inequality, introduced in [11] and crucial for their results, is inherently insufficient to carry the proof for the greedy conjecture in the general case. Finally we describe how linear programming can be used to support research along this line.

Keywords: Shortest Superstring, Greedy Heuristic, Performance Ratio

1 Introduction

Given a set of n strings $S = \{s_1, \dots, s_n\}$, the shortest common superstring problem (SCSSP) is to find a string s such that each s_i is a substring of s, and such that s is as short as possible. We may assume without loss of generality, that no string s_i is a substring of another string s_j for $i \neq j$.

Apart from being an interesting problem in itself, SCSSP models parts of the reconstruction process in DNA-sequencing [4], since DNA-fragments can be described as strings. Data compression is another area where SCSSP is used [8].

SCSSP is proven to be NP-complete by Maier and Storer [6]. It is known from the work of Blum et al. [2] that SCSSP is APX-complete. According to Arora et al. [1] such problems do not have a polynomial-time approximation scheme, unless P=NP.

The crucial part of the problem is to find the best order in which the strings s_i should appear in the superstring. Once an order is fixed the superstring can be easily constructed by greedily pulling a string as far as possible over its predecessor in the given order, hence exploiting the greatest possible *overlap*.

Definition 1 *Consider two strings a and b. The* overlap *of a and b is the longest proper suffix of a that is also a proper prefix of b. Its length is denoted by $|a, b|$.*

For a given permutation π the superstring described by π (denoted s_π) has the length

$$|s_\pi| = \sum_{i=1}^{n} |s_i| - \sum_{i=1}^{n-1} |s_{\pi(i)}, s_{\pi(i+1)}|.$$

The *greedy heuristic* can be used to approximate the superstring s by repeatedly merging strings with maximal overlap until only one string is left:

1. Input: A set S of n strings.
2. while $|S| > 1$
   (a) Choose $a, b \in S$ with maximal overlap $|a, b|$ and $a \neq b$.
   (b) Let c be the partial superstring that is created by concatenating a and the suffix of b, that does not belong to (a, b).
   (c) Let $S := \big(S \setminus \{a, b\}\big) \cup \{c\}$.
3. Output: The one string left in S.

As we obtain a partial superstring in every step, the output will be a superstring for the strings in S.

It has been conjectured by Turner [10] that this greedy heuristic has an approximation factor of 2 (the *greedy conjecture*). A simple example given by Turner [10] establishes that the approximation factor of the greedy heuristic is no better than 2. Blum et al. [2] prove that the greedy heuristic achieves an approximation factor of 4 and these are still the best known upper and lower bounds for the performance of the greedy heuristic.

A 3-approximation algorithm derived from the greedy heuristic is presented by Blum et al. in the same paper. Since then a series of results has been published, improving the approximation factor to $2\frac{1}{2}$ [9]. This was achieved by developing more and more sophisticated algorithms.

However it is known, that the greedy heuristic has approximation factor 2, if one is interested in maximizing the total amount of overlap exploited (as opposed to minimizing the length of the resulting superstring). Moreover a straightforward variation of the greedy heuristic is able to find optimal cycle covers. Hence we seeked insight as to why the greedy superstring conjecture appears so hard to verify. Therefore we searched systematically for interesting instances of the problem by introducing *greedy orders*. In section 2 we will describe our approach and introduce the *mice game*, which resembles the task of proving the greedy superstring conjecture in a tricky way and allows to exploit *conditional linear inequalities* like the prominent Monge inequality.

First results of this approach are shown in Weinard and Schnitger [11]: It is possible to verify an even stronger version of the greedy conjecture for a restricted class of orders in which strings are merged. A second result shows that the established conditional inequalities are insufficient to prove the greedy conjecture even in the classical form. A new conditional inequality – called the Triple inequality – is introduced, that carries the proof for the stronger version. We will briefly describe these results in section 2.

In this paper we extend the results from [11]: We increase the number of greedy orders for which the stronger conjecture holds from $\Theta(2^n)$ to $\Theta(4^n)$. We further show that the Triple inequality [11] is inherently too weak to prove the greedy conjecture for the general case. These results will be presented in sections 3 and 4.

To support a systematical search for interesting instances of the problem we developed the software tool SINDBAD that turned out to be extremely helpful in achieving the results of [11] and the results of this paper. We will describe the major features of SINDBAD in section 5.

2 From SCSSP to the Mice Game

The difficult part of SCSSP is to find an optimal order in which to arrange the n strings. We can assume without loss of generality, that the greedy heuristic merges the strings into the order $s_1, s_2, \ldots, s_n$. The greedy heuristic achieves this by repeatedly merging two partial superstrings. We call the order in which the partial superstrings are merged the *greedy order*. We define the greedy order as a sequence of pairs (s_i, s_{i+1}) that indicate, in which order the ends of the partial superstrings are merged by the greedy heuristic.

The length of a superstring is the sum of the lengths of the strings in $S = \{s_1, s_2, \ldots, s_n\}$ minus the overlaps of the consecutive strings

$$|s_\pi| = \sum_{i=1}^{n} |s_i| - \sum_{i=1}^{n-1} \left|s_{\pi(i)}, s_{\pi(i+1)}\right| \tag{1}$$

given a permutation π, which defines the superstring. A *cycle cover* $\mathcal{C}_\sigma$ of the strings in S is a set of disjoint cycles $C_1, C_2, \ldots, C_r$. The length of a cycle cover is

$$|\mathcal{C}_\sigma| = \sum_{i=1}^{n} |s_i| - \sum_{i=1}^{n} \left|s_i, s_{\sigma(i)}\right| \tag{2}$$

for a bijection σ, which indicates the successor of string s_i in the cycle cover.

The classical greedy conjecture is

$$|s| \leq 2 \cdot |s^*|, \tag{3}$$

where $|s|$ is the length of the superstring defined by the greedy heuristic and $|s^*|$ is the length of the optimal superstring. In [11] a stronger version is proposed:

$$|s| \leq |s^*| + |\mathcal{C}^*|, \tag{4}$$

where $|\mathcal{C}^*|$ is the length of an optimal cycle cover. This conjecture is stronger because the length of the optimal cycle cover is not longer than $|s^*|$. For technical reasons – following [11] – our goal is to prove a variation of the stronger version:

$$\forall \pi \forall \sigma : |s^\circ| \leq |s^\circ_\pi| + |\mathcal{C}_\sigma| \tag{5}$$

where $|s^\circ|$ is the length of the superstring s when it is closed as a cycle. Therefore $|s^\circ| = |s| - |s_n, s_1|$ where $|s|$ is the length of the superstring that is determined by the greedy order. We compare $|s^\circ|$ with the length of an alternative closed superstring s°_π plus the length of an alternative cycle cover $\mathcal{C}_\sigma$. Thus $|s^\circ_\pi| = |s_\pi| - \big|s_{\pi(n)}, s_{\pi(1)}\big|$.

In [11] it is shown that equation (5) implies equation (4). Replacing the terms in equation (5) with the expressions above and rearranging sums yields:

$$\forall\,\pi\forall\,\sigma : \sum_{i=1}^{n-1}\big|s_{\pi(i)}, s_{\pi(i+1)}\big| + \big|s_{\pi(n)}, s_{\pi(1)}\big| + \sum_{i=1}^{n}\big|s_i, s_{\sigma(i)}\big| \leq \sum_{i=1}^{n}|s_i| + \sum_{i=1}^{n-1}\big|s_i, s_{i+1}\big| + |s_n, s_1| \quad (6)$$

Hence we have to bound the $2n$ terms on the left-hand side by the $2n$ terms on the right-hand side. To achieve this we need to exploit properties of strings. Two trivial inequalities that can be used follow from the definition of the overlap. Let s_i, s_j be two strings with $i \neq j$ then $|s_i, s_j| < |s_i|$ and $|s_i, s_j| < |s_j|$ hold.

Definition 2 *Let s and u be strings and $|u| = p$. The string s is p-periodic if and only if s is a prefix of u^k for some k. If s is p_s-periodic and p_s is minimal, then p_s is a* minimum period *of s.*

A consequence of Definition 2 is the equality $|s_i, s_i| = |s_i| - |p_{s_i}|$ and hence $|s_i, s_i| < |s_i|$ follows. These simple inequalities are of course insufficient to prove equation (6).

Our approach is to use *conditional inequalities*, i.e. linear inequalities that hold whenever a condition, that is also a set of linear inequalities, holds. An example of a conditional inequality is the one observed by Gaspard Monge [7] in 1781.

Lemma 1 *Let a, b, c, d be strings. Given that $|a, d| \leq |a, b|$ and $|c, b| \leq |a, b|$ the inequality $|a, d| + |c, b| \leq |a, b| + |c, d|$ holds. Moreover the variant $|a, d| + |c, a| \leq |a| + |c, d|$ holds without prerequisites.*

In [11] another conditional linear inequality, the Triple inequality, is introduced.

Lemma 2 *Let a, b, c, d, x be strings. Given that* $\max\big\{|a, x|, |x, b|\big\} \geq |a, b|, |x, d|, |c, x|$ *then $|a, b| + |x, d| + |c, x| \leq |a, x| + |x, b| + |c, d| + |p_x|$ holds.*

In order to apply these conditional inequalities we need to fix the greedy orders because the greedy order provides a partial order on the overlaps. This order will establish the premises of some conditional inequalities thereby allowing us to exploit the conditioned inequality. As a consequence we have to prove (6) for all $g \in \mathcal{G}$, where $\mathcal{G}$ is the set of all greedy orders. Note that for a given n, there are $(n-1)!$ greedy orders. The following example illustrates these concepts.

Example 1 Let us use conditional inequalities and prove our inequality (6) for $n = 5$, a fixed greedy order $\big((s_1, s_2), (s_4, s_5), (s_3, s_4), (s_2, s_3)\big)$, an alternative superstring $s_5, s_4, s_1,\ s_3, s_2$ and $\big\{(s_1, s_5, s_3, s_2, s_1), (s_4, s_4)\big\}$ as a cycle cover. Hence $\pi = (5, 4, 1, 3, 2)$ and $\sigma(1) = 5, \sigma(2) = 1, \sigma(3) = 2, \sigma(4) = 4, \sigma(5) = 3$.

The partial order on the right of Fig.1 is the one induced by the given greedy order. At first the heuristic picks the overlap (s_1, s_2) and hence this overlap is larger than every other possible overlap. When, in the second step, greedy chooses to use (s_4, s_5) some of the other overlaps are no longer an option due to the choice in the

$$
\begin{aligned}
|s_3, s_2| &\leq |s_2| \\
|s_1, s_5| + |s_3, s_2| &\leq |s_1, s_2| + |s_3, s_5| \\
|s_1, s_3| + |s_2, s_1| &\leq |s_1| + |s_2, s_3| \\
|s_2, s_5| + |s_5, s_4| &\leq |s_2, s_4| + |s_5| \\
|s_2, s_4| &\leq |s_3, s_4| \\
|s_3, s_5| + |s_4, s_1| &\leq |s_3, s_1| + |s_4, s_5| \\
|s_3, s_1| + |s_5, s_3| &\leq |s_3| + |s_5, s_1| \\
|s_4, s_4| &\leq |s_4|
\end{aligned}
$$

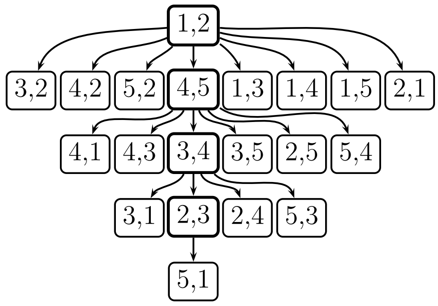


Figure 1: Proof and partial order

first round. Hence we can only exploit that (s_4, s_5) is larger than the overlaps still usable at the time. On the left we give a set of inequalities that sum up to equation (6) for this special case. The second and the sixth inequality are instances of the Monge inequality and their applicability can be verified with the partial order. (For example inequality 6 requires $|s_4, s_5| \geq |s_3, s_5|$ and $|s_4, s_5| \geq |s_4, s_1|$.)

This setup leads to the question: Which (conditional) inequalities should be used and which terms, that do not appear on either side of (6), should be introduced? For instance the introduction of the valid inequality $|s_1, s_5| + |s_4, s_1| \leq |s_1| + |s_4, s_5|$ in our example above would've made it impossible to complete the proof. We are implicitly asked to pick the *appropriate* linear inequalities. This task can be visualized as a game, that was first introduced in [11] and that is called the *mice game*.

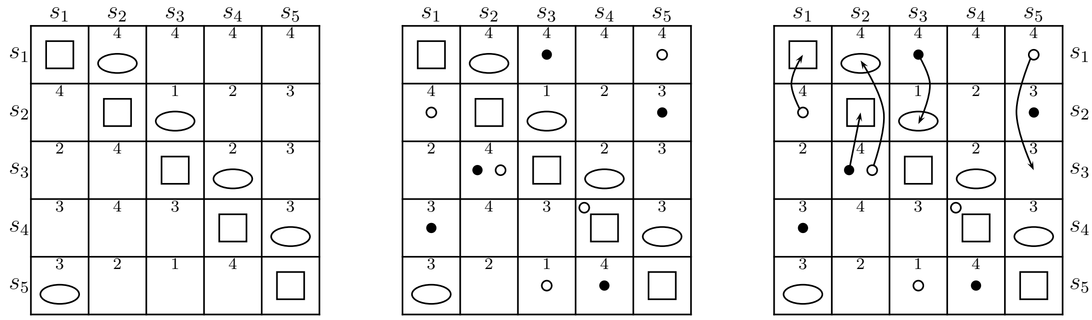
The mice game is played on a $n \times n$-board (see Figure 2(a)). The cells represent the length of the n^2 possible overlaps of the n strings. The cells on the diagonal have two interpretations: They represent the length of the self-overlap $|s_i, s_i|$ and the length of string $|s_i|$ itself.

We assign a *rank* to the cells of the board, based on the partial order of the overlaps as given by the greedy order. We assign a rank of $n - r$ to the cell whose pair was chosen as r^{th} pair and to all the cells whose pair was thereby eliminated as a possible choice for the greedy heuristic. Hence we know that an offdiagonal cell (s_i, s_{i+1}) represents a value at least as big as the value of every cell with equal or lower rank. (Note that the ranks correspond to the levels in Figure 1.)

Example 1 *continued.* Let us revisit our previous example. Figure 2(a) shows the board. The cells on the diagonal are divided into an inner and an outer part. The inner part represents the length of the string $|s_i|$ itself (the diagonal holes) and the outer part the length of the self-overlap of the string $|s_i, s_i|$. The cells with an ellipse will be refered to as the greedy holes.

The permutation π and the bijection σ determine the cells that contain the *mice* at the begining of the game. The cells are $\left|s_{\pi(i)}, s_{\pi(i+1)}\right|$, $\left|s_{\pi(n)}, s_{\pi(1)}\right|$ and $\left|s_i, s_{\sigma(i)}\right|$, the start-configuration of the game (Figure 2(b)). The mice shown as circles are placed according to the bijection that defines the cycle cover $\big\{(s_1, s_5, s_3, s_2, s_1), (s_4, s_4)\big\}$. The mice shown in black are placed according to the permutation that defines the alternative superstring s_5, s_4, s_1, s_3, s_2. For the course of the game the mice are indistinguishable, they are shown differently here to illustrate their origin.

A move is described by a set of start-cells and a set of end-cells. The move is justified by an inequality that guarantees that the sum of the lengths represented



(a) Board with ranks. (b) A start-configuration. (c) First three moves.

Figure 2: The board of the mice game

by the start-cells is not greater than the sum of the lengths represented by the end-cells. The moves inherit the name of the inequality that justifies them. Figure 2(c) shows the moves that correspond to the first three inequalities from the first part of Example 1. Firstly a diagonal insertion, secondly a greedy monge and thirdly a diagonal monge. (If we move a mouse from the outer to the inner part of a diagonal, using $|s_i, s_i| < |s_i|$ we call this *discarding a period.*) Applying the moves of all the inequalities listed in the example, brings all the mice into their holes, which is the objective of the game. A hole can only accomodate one mouse. Sucessfully moving the mice into their holes will be refered to as *winning the game*. Not used in this example is the Triple inequality, its intertrepation as a move in the game is shown below. The premises of the inequality are indicated by a sequence of arrows.

Figure 3: The horizontal and vertical Triple.

Winning the mice game hence corresponds to finding a derivation of the right-hand side (the end-configuration of the game) of equation (6), starting from the left-hand side (the start-configuration of the game). That is a proof of equation (6) for permutation π and bijection σ.

Note that we only use the properties of the strings described by the conditional linear inequalities. We do not have to care about the structure of the strings themselves, only their lengths and the lengths of their overlaps are sufficient in this case.

If we do this for every permutation π and bijection σ, we have proven the conjecture for a single greedy order (a single board). As we want to prove equation (6) for arbitrary n, we cannot handle all π and σ individually. We have to search for a winning strategy for the mice game, that is a set of rules according to which the mice should be moved. The *Rank Descending Algorithm* [11] is a winning strategy for the mice game for the restricted case of *linear greedy orders*. A linear greedy order is an order in which the greedy heuristic starts with an arbitrary overlap (s_i, s_{i+1}) and in later steps, when $s_j, \ldots, s_k$ is already created, either picks (s_{j-1}, s_j) or (s_k, s_{k+1}).

3 Extension of the Rank Descending Algorithm

We now introduce our extension of the simple Rank-Descending-Algorithm (RDA), the Duplex Rank-Descending-Algorithm (DPX-RDA), that will prove the greedy conjecture for $\Theta(4^n)$ greedy orders, thus squaring the number of orders covered in comparison to the simple RDA, that could only cover $\Theta(2^n)$ orders [11].

Definition 3 *A greedy order corresponds to one distinct distribution of ranks on the offdiagonal of the mice board.*

1. *A greedy order is linear, if there exists i such that $rank(1,2) < rank(2,3) < \ldots < rank(i,i+1) > rank(i+1,i+2) > \ldots > rank(n-1,n)$.*

2. *A greedy order is bilinear, if there exist i,k and w such that $rank(1,2) < rank(2,3) < \ldots < rank(i,i+1) > rank(i+1,i+2) > \ldots > rank(w,w+1) = 1 < rank(w+1,w+2) < \ldots < rank(k,k+1) > rank(k+1,k+2) > \ldots > rank(n-1,n)$*

The concept of subboards is essential for understanding both the RDA and the DPX-RDA. We extend the definition in [11] by introducing definitions 4.1b and 4.1c.

Definition 4 *1a. The subboard $Board_{i,j}$ (with $i \leq j$) is the set of all cells in the intersection of rows and columns $\{i \ldots j\}$.*

1b. The subboard $Board_{i,j}$ (with $i > j$) is the set of all cells in the intersection of rows and columns $\{1 \ldots j\} \cup \{i \ldots n\}$.

1c. For $i \leq j$ we say that $Board_{i,j}$ and $Board_{j+1,i-1}$ are complementary boards.

2. Let $B = Board_{i,j}$. The horizontal [vertical] frame of B is the set of all cells that belong to a row [column] of $Board_{i,j}$, but not to one of its columns [rows]. The frame of B is the union of its horizontal and vertical frame.

3. Let $B = Board_{i,j}$. We define $G_1(B)$ to be the greedy cell in position $(j, j+1)$, if existing, and $G_2(B)$ to be the greedy cell in position $(i-1,i)$, if existing. We further define the neighbouring diagonal cells, $N(G_1(B)) := (j+1, j+1)$ and $N(G_2(B)) := (i-1, i-1)$, if existing.

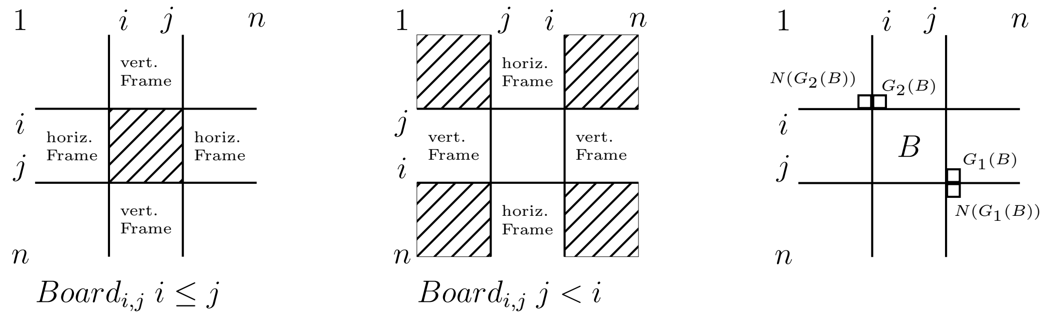


Figure 4: Complementary Boards and Frames

The Duplex-Rank Descending Algorithm

(1) The **input** consists of a superstring s°, a cycle cover $\mathcal{C}$ and a bilinear greedy sequence.

(2) **Preprocessing**

- (2a) Place a mouse on position (u, v), if string s_v immediatly follows s_u in s° or in $\mathcal{C}$. If s_v is the immediate successor of s_u in both s° and $\mathcal{C}$, then (u, v) receives two mice. If a mouse is placed on a diagonal (u, u) it is placed in the outer part.
- (2b) Let i, k and w be chosen according to the definition of bilinearity in Definition 3. Set $B_1 = Board_{i,i}$ and $B_2 = Board_{k,k}$
- (2c) If (i, i) contains a mouse, then discard its period. Otherwise execute a diagonal monge in (i, i). If (k, k) contains a mouse, then discard its period. Otherwise execute a diagonal monge in (k, k).

(3) **Main Loop**

<table>
<tr><td colspan="5">while $B_1 \neq Board_{1,w}$ or $B_2 \neq Board_{w+1,n}$</td></tr>
<tr><td colspan="5">Let G be the highest ranked greedy cell among $G_1(B_1), G_2(B_1), G_1(B_2), G_2(B_2)$</td></tr>
<tr><td colspan="5">Does G contain a mouse?</td></tr>
<tr><td colspan="2">yes</td><td colspan="3">no</td></tr>
<tr><td colspan="2">Does $N(G)$ hold a mouse?</td><td colspan="3">Does $N(G)$ contain a mouse?</td></tr>
<tr><td>yes</td><td>no</td><td>no</td><td colspan="2">yes</td></tr>
<tr><td rowspan="4">**Discard period** in $N(G)$</td><td rowspan="3"></td><td rowspan="3">**Greedy Monge** in G</td><td colspan="2">Is the Triple in $G, N(G)$ legal?</td></tr>
<tr><td>no</td><td>yes</td></tr>
<tr><td>**Greedy Monge** in G</td><td rowspan="2">**Triple**</td></tr>
<tr><td colspan="2">**Diagonal Monge** in $N(G)$</td><td>**Discard period** in $N(G)$</td></tr>
<tr><td colspan="5">Extend the B_i, that G is incident to, to include G</td></tr>
</table>

By the preprocessing step (2) two subboards B_1, B_2 are provided that fulfill the following 5 invariants. In [11] the corresponding invariants for a single subboard were used.

- I1 Every row and every column of the board contains 2 mice.
- I2 Every hole in B_1 and B_2 is filled.
- I3 No diagonal outside of B_1 and B_2 holds a mouse in the inner part.
- I4 No diagonal contains more than one mouse.
- I5.1 For all subboards B' with $B_1 \subseteq B' \neq Board_{1,n}$ the frame of B' is not empty.
- I5.2 For all subboards B' with $B_2 \subseteq B' \neq Board_{1,n}$ the frame of B' is not empty.

It is easy to verify that these invariants hold after the preprocessing step. (I5 holds since s° is a single cycle.) We call a move legal if it does not violate any of the invariants. The main loop of the DPX-RDA grows the two subboards by systematically filling the greedy cell of highest rank that is not yet in B_1 or B_2. The body of the loop is essentially the same as in the simple RDA. Only the stop condition needs adjustment and the set of greedy cells to pick from is different. Note, that w is picked according to Definition 3.

Invariant I1 will be preserved as the moves used, leave the number of mice per row and column unchanged. Furthermore B_i only grows if new holes are filled. Hence I2 is established once the existence of the required moves is shown. If a mouse steps on a diagonal cell that is not about to be included into a B_i it only enters the outer part.

The proof of the existence of the moves required by the DPX-RDA, as well as maintaining invariants I4 and I5 is rather involved even for the simple RDA [11]. Luckily most of the observations establishing the existence of the moves and invariants I4 and I5 follow from [11]. In [11] the invariants are shown for one growing subboard B. These arguments remain valid and we only need to provide additional arguments to make sure that the two boards do not interfere with each other.

Hence the remaining arguments are organized as follows: For the existence of the moves we need Lemma 3. Once we have established that the moves exist, I4 holds, since it holds for both B_i individually. As to I5 we inherit from [11], that no move enlarging B_i will clear the frame of a subboard B' that includes B_i. Lemma 4 will guarantee that a move enlarging one of the B_i will not clear the frame of a board B' that includes the other B_i. Finally in Lemma 5 we argue that the preserved invariants together with the stop condition of the main loop yield a won game.

We call a mouse *free*, if it is not in a hole and not on a diagonal.

Lemma 3 *The rank of the greedy cell G, that is about to be filled at a given time, is high enough to dominate every free mouse on the board.*

Proof: When DPX-RDA tries to fill G, all the greedy cells of higher rank and their neighbouring diagonals are already filled by I2. By I1, there are only two mice in every row and column, no free mouse can be in the row or the column of a greedy cell already taken care of. In fact only the cell located in the bottom left corner of the B_i, that is about to be extendend, has a rank higher than G and could hold a mouse without violating I1. But in this case I5 would be violated, since the frame of B_i itself would be free of mice. ◇

Lemma 4 *If a move in G that extends B_i does not violate I5.i it will not violate I5.k either (with $i, k \in \{1, 2\}, i \neq k$).*

Proof: Assume the opposite, namely that the frame of a board B' with $B_k \subseteq B'$ gets cleared while no subboard $B'' \supseteq B_i$, whose frame gets cleared, exists. Observe that the frames of B' and $\overline{B'}$ are identical. But since B_i and B_k do not intersect, $B_i \subseteq \overline{B'}$ holds and we have a contradiction with $B'' = \overline{B'}$. ◇

Lemma 5 *At the end of the main loop of the DPX-RDA the game is won.*

Proof: All the invariants are preserved and we have $B_1 = Board_{1,w}$ and $B_2 = Board_{w+1,n}$. As all the holes in B_1 and B_2 are filled, the positions of the two mice in rows 1 to $w-1$ and $w+1$ to $n-1$ as well as columns 2 to w and $w+2$ to n are fixed. One of the mice in rows w and n as well as the columns 1 and $w+1$ is also accounted for. Only two possible arrangements for the last two mice do not violate I1: They are either on $(w, 1)$ and $(n, w+1)$ or on the winning position $(w, w+1)$ and $(n, 1)$. The first arrangement contradicts I5 for B_1 and B_2. ◇

It should be noted, that only Lemma 5 can not be extended to *tri-linear* or more complex greedy orders.

4 Limitations of the Triple Inequality

The Monge and the Triple inequalities (plus the trivial ones that correspond to insertions) are used in [11] to prove the strong version of the greedy conjecture (4) for linear greedy orders. In section 3 these inequalities are used to prove the strong version (5) for bilinear greedy orders. In [11] it is shown, that the Triple inequality is crucial, i.e. it is impossible to prove even the weaker classical greedy conjecture with just the Monge inequality.

We will now show that it is not possible to prove the classical greedy conjecture for arbitrary greedy orders with just the elementary inequalities, the Monge inequality *and* the Triple inequality. We do this by providing a 10×10 board that fulfills all these inequalities and still violates $|s| \leq 2 \cdot |s_\pi|$ for a given π.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}
s_1	**i+5**	$\boxed{5}$ **i+4**	$\boxed{7}$ i+3	$\boxed{8}$ i+1	$\boxed{5}$ 0	$\boxed{5}$ 0	$\boxed{5}$ 0	$\boxed{9}$ i+1	$\boxed{5}$ i+2	$\boxed{6}$ i+1
s_2	$\boxed{7}$ i+1	**3i+10**	$\boxed{7}$ **3i+9**	$\boxed{8}$ 3i+3	$\boxed{7}$ i	$\boxed{7}$ i	$\boxed{7}$ 2i+2	$\boxed{9}$ 3i+3	$\boxed{7}$ 3i+5	$\boxed{7}$ 3i+4
s_3	$\boxed{8}$ 0	$\boxed{8}$ 0	**4i+10**	$\boxed{8}$ **4i+4**	$\boxed{8}$ i	$\boxed{8}$ i	$\boxed{8}$ 3i	$\boxed{9}$ 4i+2	$\boxed{8}$ i	$\boxed{8}$ i
s_4	$\boxed{5}$ 0	$\boxed{7}$ 0	$\boxed{8}$ 0	**4i+5**	$\boxed{3}$ **i**	$\boxed{3}$ i	$\boxed{3}$ 0	$\boxed{9}$ i	$\boxed{4}$ i	$\boxed{6}$ i
s_5	$\boxed{3}$ 0	$\boxed{5}$ 0	$\boxed{7}$ 0	$\boxed{8}$ 0	**i+1**	$\boxed{1}$ **0**	$\boxed{2}$ 0	$\boxed{9}$ 0	$\boxed{4}$ 0	$\boxed{6}$ 0
s_6	$\boxed{2}$ 0	$\boxed{5}$ 0	$\boxed{7}$ 0	$\boxed{8}$ i	$\boxed{3}$ i	**i+1**	$\boxed{2}$ **0**	$\boxed{9}$ i	$\boxed{4}$ i	$\boxed{6}$ i
s_7	$\boxed{9}$ 0	$\boxed{9}$ 0	$\boxed{9}$ 0	$\boxed{9}$ 4i+2	$\boxed{9}$ i	$\boxed{9}$ i	**4i+6**	$\boxed{9}$ **4i+4**	$\boxed{9}$ i	$\boxed{9}$ i
s_8	$\boxed{4}$ 0	$\boxed{5}$ 0	$\boxed{7}$ i+2	$\boxed{8}$ 4i	$\boxed{4}$ i	$\boxed{4}$ i	$\boxed{9}$ 4i+2	**5i+6**	$\boxed{4}$ **i+1**	$\boxed{6}$ m
s_9	$\boxed{6}$ i+1	$\boxed{6}$ i+1	$\boxed{7}$ 3i+8	$\boxed{8}$ 3i+3	$\boxed{6}$ i	$\boxed{6}$ i	$\boxed{6}$ 2i+2	$\boxed{9}$ 3i+3	**4i+10**	$\boxed{6}$ **3i+6**
s_{10}	$\boxed{1}$ 0	$\boxed{5}$ i+2	$\boxed{7}$ 2i+4	$\boxed{8}$ 2i+2	$\boxed{3}$ 0	$\boxed{2}$ 0	$\boxed{4}$ i+1	$\boxed{9}$ 2i+2	$\boxed{6}$ 3i+6	**3i+8**
s.o.	0	i+1	0	i	0	i	3i	4i	3i+4	2i+2

This board describes the overlaps and lengths of 10 strings. A greedy order is given by the ranks in the little boxes. The diagonal shows the lengths of the strings and the extra row in the bottom indicates the overlaps of the strings with themselves. It can be checked exhaustively, that this board fulfills all the inequalities mentioned. The length of the strings on the diagonal sum up to $30i + 62$. The greedy locations on the offdiagonal accomodate a total value of $17i + 28$. We use $\pi = (1, 2, 10, 9, 3, 8, 7, 4, 6, 5)$ for the superstring s_π. The locations of the overlaps exploited by this superstring sum up to $24i + 28$. Subtracting the greedy overlaps from the lengths of the strings gives the length of the superstring of the greedy heuristic as $13i + 34$. The length of the alternative superstring s_π is $6i + 34$. As we may choose i arbitrarily, the ratio approaches $2\frac{1}{6}$.

This shows, that a proof of the greedy conjecture needs to exploit string properties beyond the Triple and the Monge inequality.

5 SINDBAD

SINDBAD is the name of a software tool that we developed to support the research on SCSSP. The introduction of greedy orders gives the greedy conjecture a shape with four quantifiers

$$\forall n \; \forall g \in \mathcal{G}_n \; \forall \pi \; \forall \sigma \; : \; |s^\circ| < |s^\circ_\pi| + |C_\sigma|.$$

For most instances (that is values of n, g, π and σ) the proof can be done with the established conditional inequalities. In order to proceed one must find instances, that cannot be covered with the means available. The use of calculation power is the logical consequence.

One important feature is SINDBAD's *enumeration mode*. It can automatically determine all valid moves on the board for a given greedy order and check whether the game on such a board can be won against a given alternative superstring π and a given cycle cover σ. To do this a linear program (LP), depending on the current configuration of the game is generated and evaluated. SINDBAD found the first instance of a game with linear greedy order for which the classical greedy conjecture can not be proven just with insertions and monges [11]. It was nessecary to check boards up to a size of 9×9 to encounter such an instance. The matrix from section 4 was also found using SINDBAD. (For more on the linear program and on how to define an *intermediate performance ratio* for arbitrary game configurations see subsection 5.1.)

SINDBAD also provides a *manual mode* that allows to play the mice game on arbitrary boards against arbitrary π and σ via a graphical interface. Playing the mice game manually allows interesting insights: In a game that can be won, there will still be legal moves that destroy the property, that the game can be won. Being able to locate these bad moves is extremely helpful when working on strategies. The invariants both of the RDA and of the DPX-RDA embody experience, as to what moves destroy the winning property of a game. In a game bound to be lost, one will probably still find valid moves. A *deadend* (that is a configuration with no legal move left) yields an interesting question about strings: The linear program assigns lengths to the strings and overlaps that agree with all the inequalities implemented. If a performance ratio above 2 is still possible, the question is, whether it is possible to construct a set of strings that behave like the solution of the linear program indicates. If so, a counter example would be found. If on the other hand one can pinpoint the reason why the construction of such strings is impossible, one has found a new property of strings that is guaranteed to solve yet unsolved instances of the problem. The Triple inequality was found in exactly that way. It seems fair to say that it would have been very hard to recognize the usefulness of the Triple inequality for our purposes without SINDBAD.

We also made it possible to implement game strategies into SINDBAD. Hence we could quickly find instances for which a certain strategy fails. If such a game can be won, but the strategy fails, one has a chance to improve the strategy. If it cannot be won according to the LP, one gets a hint on the limits of the moves established so far.

Further usefull features include the possibility to work with assumptions. For a given greedy order the partial order *activates* a set of conditional linear inequalities. By adding assumptions like $|a, b| \geq |c, d|$ the set of legal moves grows. If a game can be won under this assumption and also under the assumption $|a,b| \leq |c,d|$, the instance is solved as well. SINDBAD supports the systematical search for these assumptions. SINDBAD can also assign different capacities to the holes and work with different numbers of mice. This makes it possible to work on weaker bounds for tough instances and on stronger bounds for easy instances.

5.1 SINDBAD's Linear Program

Writing a linear program to check the classical greedy conjecture on a given board and given π is simple: Let $x_{i,j}$ represent the lengths of the overlaps and l_i the lengths of the strings themselves. Let Q be the set of all inequalities that hold on the given board.

$$\begin{aligned} max &: \sum_{i=1}^{n} l_i - \sum_{i=1}^{n-1} x_{i,i+1} \\ \text{subject to} &: Q \cup \left\{ \sum_{i=1}^{n} l_i - \sum_{i=1}^{n-1} x_{\pi(i),\pi(i+1)} \leq 1 \right\} \\ \forall i,j &: x_{i,j}, l_i \geq 0 \end{aligned}$$

The linear program assigns values to all lengths and overlaps that comply with the constraints in Q and that produce a superstring s_π of length at most one. The objective function is the length of the greedy superstring. Hence the value of the optimal solution will be the best performance ratio that can be derived with the given set of inequalities Q.

From a linear programming perspective it is worthwhile noticing, that the dual problem [3, 5] of this implementation is a description of the mice game: There is a nonnegative variable y_q associated with every inequality $q \in Q$ (hence y_q is associated with a move in the mice game). A further nonnegative variable t associated with the inequality $\sum_{i=1}^{n} x_i - \sum_{i=1}^{n-1} x_{\pi(i),\pi(i+1)} \leq 1$ appears. As we don't have any inequalities in Q that include constants (i.e. all of them compare a linear combination of lengths and overlaps to 0), the objective function of the dual problem just depends on the variable t, that is to be minimized.

For every variable of the primal program (i.e. strings and overlaps, that is the cells of the miceboard) a constraint arises. For a cell c define its balance as follows:

$$bal(c) := \sum_{\substack{q \in Q \\ q \text{ moves a mouse into } c}} y_q - \sum_{\substack{q \in Q \\ q \text{ moves a mouse out of } c}} y_q. \tag{7}$$

If we interpret the variables y_q as the number of times move q is executed in a game, $bal(c)$ describes the balanced total of mice moving into and out of c during a game. The constraints of our dual problem are:

$$\begin{aligned} bal(c) - t &\leq -1 \text{ iff } c \text{ is a diagonal} \\ bal(c) &\leq 1 \text{ iff } c \text{ is a greedy cell} \\ bal(c) + t &\leq 0 \text{ iff } c \text{ is an initial mice position } (s_{\pi(i)}, s_{\pi(i+1)}) \\ bal(c) &\leq 0 \text{ otherwise.} \end{aligned}$$

(If a cell c should be a greedy cell *and* an initial mice position, its constraint is $bal(c) + t \leq 1$.) Remember, that the optimal solutions of primal and dual problem have the same value. Hence, we can verify that the performance ratio is upper bounded by 2, by giving a legal solution of the above inequalities with $t = 2$. That is, we need to provide a set of legal moves, that move 2 mice out of every initial position and respects the capacity constraints (1 for diagonal and offdiagonal, 0 for every other cell) — the mice game.

This dual representation also shows how the mice game can be adapted in order to prove bounds other than 2. If one is interested in proving a factor of 3 for instance, the game needs to be won with 3 mice on every initial position, a capacity of 2 for the diagonal holes and a capacity of 1 for the greedy holes. If we are interested in non integer performance ratios we can still use the game by scaling the number of mice and the capacities. To prove an upper bound of 2.5, we would play with 5 mice on the initial positions, and capacities 3 resp. 2 for diagonals and greedy cells.

The adaptions necessary to work with cyclicly closed superstrings are obvious. The cell $(n, 1)$ is treated as a greedy cell and $(\pi(n), \pi(1))$ is included as an initial position.

Crucial for our research was the ability to evaluate arbitrary game configurations, that is configurations that might arise during a game and that do not have a straight forward interpretation in terms of superstrings or cycle covers. Observe that the above representation indicates a total of $n \cdot t$ mice, if we work with cyclic closure. In the course of the game these mice (initially placed in groups of t on the initial positions) will spread over more cells and different numbers of mice will be on different cells of the board. In our dual representation the generalisation necessary is rather natural.

$$\begin{aligned} bal(c) - t &\leq -n \cdot t \cdot m(c) - 1 \text{ iff } c \text{ is a diagonal} \\ bal(c) &\leq -n \cdot t \cdot m(c) + 1 \text{ iff } c \text{ is a greedy cell} \\ bal(c) &\leq -n \cdot t \cdot m(c) \text{ otherwise,} \end{aligned}$$

where $m(c)$ indicates the percentage of mice on cell c. (Hence $\sum_c m(c) = 1$ and $n \cdot t \cdot m(c)$ is the number of mice on cell c at a given time.) Observe that we still describe the winning property of the game. To check how these adaptions are resembled in the primal version we need to put the above back into the shape of a linear program.

$$\begin{aligned} min &: t \\ bal(c) + (n \cdot m(c) - 1) \cdot t &\leq -1 \text{ iff } c \text{ is a diagonal} \\ bal(c) + (n \cdot m(c)) \cdot t &\leq 1 \text{ iff } c \text{ is a greedy cell} \\ bal(c) + (n \cdot m(c)) &\leq 0 \text{ otherwise,} \end{aligned}$$

Retransfering we find that we still have the variables l_i and $x_{i,j}$, we still have the constraints from set Q and the objective function is still to maximize the length of the greedy superstring $\sum_{i=1}^{n} l_i - \sum_{i=1}^{n-1} x_{i,i+1} - x_{n,1}$.

The adaption to arbitrary game configurations is only resembled in the one inequality, that initially bounded the superstring s_π. It is replaced by

$$\sum_{i=1}^{n} l_i - n \cdot \left(\sum_{c=(i,j)} m(c) \cdot x_{i,j} + \sum_{c \text{ is diagonal } i} m(c) \cdot l_i \right) \leq 1. \tag{8}$$

Observe that regular starting positions are embedded: In start configurations there are no mice on diagonals and every starting position holds 2 mice, hence a fraction of $\frac{1}{n}$ of all mice on the board.

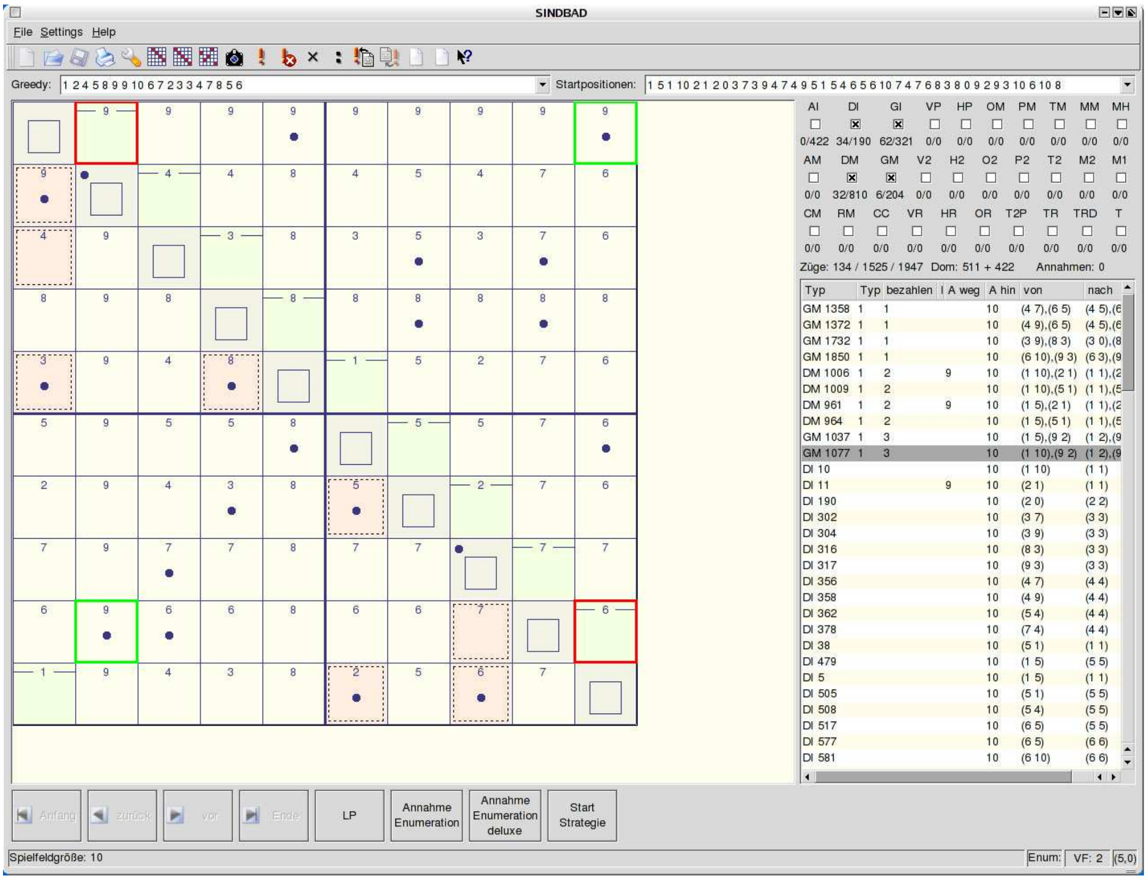


Figure 5: This is how SINDBAD looks like. SINDBAD is a KDE-Application written in C++. We experimented with an interior-point and a simplex based solver. The interior-point solver is a FORTRAN version of Csaba Mészáros' BPMPD solver. (`http://www.sztaki.hu/~meszaros/bpmpd`). The simplex based solver is `SoPlex` created by Roland Wunderling.

6 Conclusion

We have extended the class of greedy orders for which the greedy conjecture can be verified and proved that a proof for the general case is not possible without exploiting string properties beyond those used in [11]. Of course the conjecture for the general case remains *the* open problem. We believe that the stronger version of the greedy conjecture might turn out to be easier to prove. We are not aware of a counterexample for the stronger version for any greedy order. We do believe, that the approach via the dual problem (the mice game) can help focusing further research along this line as we can pinpoint unsolved instances with the help of computers. Hence we are quickly led to *good questions* about strings whose answers will cause progress in the work on the greedy conjecture.

7 Acknowledgements

We would like to thank Roland Wunderling for making SoPlex available [12].

References

[1] S. Arora, C. Lund, R. Motwani, M. Sudan and M. Szegedy: Proof verification and hardness of approximation problems. Journal of the ACM, **45**(3), 501–555, May 1998.

[2] A. Blum, T. Jiang, M. Li, J. Tromp and M. Yannakakis: Linear approximation of shortest superstrings. Journal of the ACM, **41**(4), 630–647, July 1994.

[3] Vašek Chvǎtal: Linear Programming. W. H. Freemann and Company, 1983.

[4] T. Jiang, M. Li: DNA Sequencing and String Learning. Mathematical Systems Theory (now Theory of Computing Systems), **29**(4), 387–405, July/August 1996.

[5] H. Karloff: Linear Programming. Birkhäuser, 1991.

[6] D. Maier and J. A. Storer: A Note on the Complexity of the Superstring Problem. In Proceedings of the 12th Annual Conference on Information Sciences and Systems, 52–56, 1978.

[7] G. Monge: Mémoire sur la théorie des déblais et des remblais. Historie de l'Academie Royale des Sciences, Année MDCCLXXXI, avec les Mémoires de Mathématique et de Physique, pour la même Année, Tirés des Registres de cette Académie, 666–704, 1781.

[8] J. A. Storer: Data Compression: Methods and Theory. Computer Science Press, 1988.

[9] Z. Sweedyk: A $2\frac{1}{2}$-Approximation Algorithm for Shortest Superstring. SIAM Journal on Computing, **29**(3), 954–986, December 1999.

[10] J. Turner: Approximation Algorithms for the Shortest Commom Superstring Problem. Information and Computation, **83**(1), 1–20, October 1989

[11] M. Weinard and G. Schnitger: On the Greedy Superstring Conjecture. In Proceedings of the 23rd Conference on Foundations of Software Technology and Theoretical Computer Science, Mumbai, India, LNCS 2914, 387–398, December 2003. Extended version:
`http://www.thi.informatik.uni-frankfurt.de/~weinard/`
`/indexE.html`

[12] R. Wunderling: Paralleler und Objektorientierter Simplex-Algorithmus. Ph.D. thesis, ZIB technical report TR 96-09, Berlin, 1996.
`http://www.zib.de/PaperWeb/abstracts/TR-96-09`

Combinatorial Characterization of the Language Recognized by Factor and Suffix Oracles

Alban Mancheron and Christophe Moan

L.I.N.A., Université de Nantes, 2, Rue de la Houssinière, B.P. 92208,
44322 Nantes Cedex 3, France
e-mail: {Mancheron, Moan}@lina.univ-nantes.fr

Abstract. Sequence Analysis requires to elaborate data structures which allow both an efficient storage and use. Among these, we can cite Tries [1], Suffix Automata [1, 2], Suffix Trees [1, 3]. Cyril Allauzen, Maxime Crochemore and Mathieu Raffinot introduced [4, 5, 6] a new data structure, linear on the size of the represented word both in time and space, having the smallest number of states, and allowing to accept at least all the substrings of the represented word. They called such a structure a *Factor Oracle*. On the basis of this structure, they developed another one having the same properties excepting the accordance of all the suffix of the represented word. They called it *Suffix Oracle*.
The characterization of the language recognized by the Factor/Suffix Oracle of a word is an open problem for which we provide a solution.

Keywords: Factor Oracle, Suffix Oracle, automata, language, characterization.

1 Introduction

Within text indexation, several structures were developed. The objective of these methods is to represent a text or a word s, *ie.* a succession of symbols taken in an arbitrary alphabet denoted by Σ, in order to "quickly" determine whether this word contains some specific sub-word. In which case, we call this sub-word a *factor* of s.

Cyril Allauzen, Maxime Crochemore and Mathieu Raffinot described a method allowing to build an *acyclic* automaton, accepting at **least** the factors of s, having as few states as possible ($|s|+1$), and being *linear* in the number of transitions ($2\,|s|-1$). They named such an automaton a *Factor Oracle*.

In this automaton, each state is final. Using the same automaton, but only keeping "particular" states as final, one obtains a *Suffix Oracle*.

This structure has several advantages. First of all, the construction algorithm is easy to understand and implement; this is not the case of the most efficient algorithm for building Suffix Tree's. Next, Oracles are homogeneous automata (*ie.* all the transitions going to the same state are labeled with the same symbol). That means that we do not need to label edges. This makes this structure very sparing in memory (much more than Suffix Trees or Tries). Indeed, methods based upon this structure obtain good results. Thus, Lefebvre & *al.* [7, 8, 9] use it for repeated motifs discovery over large genomic data, and obtain results similar to the one obtained using thousands of BLASTn requests, but in a few seconds. They also use the Factor Oracle in text compression [10], and in some cases they have compression ratio comparable to bzip2 (which is one of the most efficient compression algorithm).

Nevertheless, at least two problems linked to these Oracles are still opened: the first one is the characterization of the language recognized by Oracles; the second one is: does there exist an algorithm, linear in time and space, to build an automaton accepting at least the factors/suffixes of a word s being minimal in number of transitions?

The first open problem is really important. Currently, the main difficulty when using Oracles is to distinguish true positives from false positives. That is why we are interested in the first problem. In the following section, we provide several definitions relating to the construction of Oracles. Then we give the characterization of the language recognized by this structure. To conclude, we show some results about the Oracles.

2 Definitions

Subsequently, we use the notations hereafter (some of them are issued from [4, p. 2]): we denote by $Fact(s)$ (resp. $Suff(s)$ and $Pref(s)$) the set of the factors (resp. suffixes and prefixes) of $s \in \Sigma^+$, by $Pref_s(i)$ the prefix of s having length $i \geq 0$. Given $x \in Fact(s)$, we denote by $Nb_s(x)$ the number of occurrences of x in s, and we say that x is *repeated* if $Nb_s(x) \geq 2$.

Definition 2.1 Given a word $s \in \Sigma^+$ and x a factor of s, we define the function Pos as the position of the first occurrence of x in $s = uxv$ $(u, v \in \Sigma^*)$ such that x is not repeated in ux): $Pos_s(x) = |u| + 1$. We also define the function *poccur* such that $poccur_s(x) = |u| + |x| = Pos_s(x) + |x| - 1$ (denoted by $poccur(x, s)$ in [4, p. 2]).

In the following, we define the Oracles, then we give some notations and definitions peculiar to factors, as well as properties about the newly defined objects. Finally, in order to characterize the language recognized by Oracles, we define particular factors and then operations linked to them.

2.1 Oracles

We give below the algorithm of Allauzen & *al.* [4] which describes the Oracle construction (*cf.* algorithm 1). In the same paper, authors give another algorithm which allows to build the same automaton in linear time on the size of s. Nevertheless, because we are only interested in the properties of the Oracle, we do not give it in this paper.

Definition 2.2 [4, pp. 2, 10] Given a word $s \in \Sigma^*$, we define the *Factor Oracle* of s as the automaton obtained by the algorithm 1 (p. 141), where all the states are final. It is denoted by $FO(s)$. We define the *Suffix Oracle* of s as the automaton obtained by the same algorithm, where are final only the states such that there exists a path from the initial state recognizing a suffix of s. It is denoted by $SO(s)$.

Notation 2.1 Given a word $s \in \Sigma^*$, we use the term *Oracle* to indifferently indicating $SO(s)$ or $FO(s)$, and we denote it by $O(s)$.

Algorithm 1: Construction of the Factor Oracle of a word[1]

```
1  Input: Σ % Alphabet (supposed minimal) %
2         s ∈ Σ* % The word to process %
3  Output: Oracle % Factor Oracle of s %
4
5  Begin
6    Create the initial state labeled by e_0
7
8    For i from 1 to |s| Do
9      Create a state labeled by e_i
10     Build a transition from the state e_{i-1} to the state e_i labeled by s[i]
11   End For
12
13   For i from 0 to |s|-1 Do
14     Let u be a word of minimal length recognized in the state e_i
15     For All α ∈ Σ \ {s[i+1]} Do
16       If uα ∈ Fact(s[i-|u|+1..|s|]) Then
17         j ← poccur_{s[i-|u|+1..|s|]}(uα) - |u|
18         Build a transition from the state e_i to e_{i+j} labeled by α
19       End If
20     End For All
21   End For
22 End
```

We have an order relation between states in these Oracles. Indeed, if we have two states e_i and e_j such that $i \leq j$, we can say that $e_i \leq e_j$.

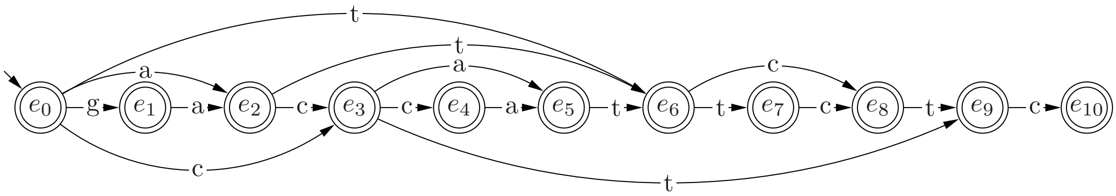


Figure 1: Factor Oracle of the word *gaccattctc*.

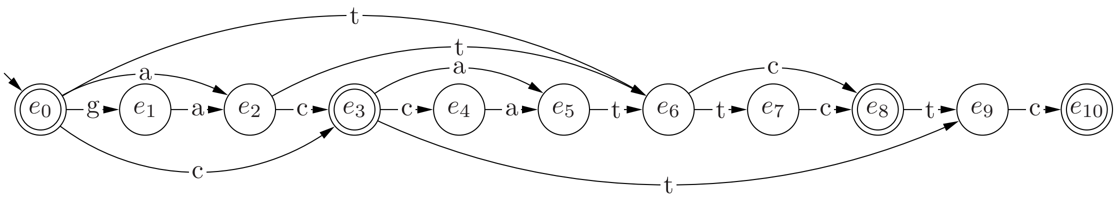


Figure 2: Suffix Oracle of the word *gaccattctc*.

[1]As mentioned in [11], the term $-|u|$ (line 17) is unfortunately missing in the original algorithm.

Definition 2.3 Given a word $s \in \Sigma^*$ and a word x accepted in the state e_i ($0 \leq i \leq |s|$) by the Oracle of s, we define the function *State* as $State(x) = e_i$.

Lemma 2.1 *[4, pp. 2, 3] Given a word $s \in \Sigma^*$ and its Oracle, there is a unique word having minimal length accepted at each state e_i ($0 \leq i \leq |s|$) of $O(s)$. It is denote it by $min(e_i)$.*

Lemma 2.2 *[4, pp. 2, 3] Given a word $s \in \Sigma^*$, its Oracle and an integer i ($0 \leq i \leq |s|$), then $min(e_i) \in Fact(s)$ and $i = poccur_s(min(e_i))$.*

Notation 2.2 Given a word $s \in \Sigma^*$, we denote by $\#_{in}(e_i)$ (resp. $\#_{out}(e_i)$) the number of ingoing (resp. outgoing) transitions in the state e_i ($0 \leq i \leq |s|$) of the Oracle of s.

2.2 Canonical Factors & Contraction Operation

We first introduce some definitions about particular factors from a given word. We use such factors for defining the contraction operation, as well as properties peculiar to this operation. We next define the sets of words we obtain applying this operation. At the end of this section, all that we need to characterize the language of Oracles will be defined.

Definition 2.4 Given a word $s \in \Sigma^*$ and its Oracle, we define the set of *Canonical Factors* of s as following:

$$\mathcal{F}_s = \{min(e_i) \mid 1 \leq i \leq |s| \ \wedge \ (\#_{out}(e_i) > 1 \ \vee \ \#_{in}(e_i) > 1)\}$$

Given a suffix t of s and a Canonical Factor f of s, we say that f is a *conserved* Canonical Factor of s in t if the first occurrence of f in s is contained in t. We denote by $\mathcal{F}_{s,t}$ the set of conserved Canonical Factors of s in t (thus $\mathcal{F}_{s,t} \subseteq \mathcal{F}_s$).

These particular factors enable us to define a set of couple of specific positions in the word s. Those will be used in order to derive new words from s.

Definition 2.5 Given a word $s \in \Sigma^*$ and a Canonical Factor f of s such that:

$$\begin{cases} s & = & ufv & (u, v \in \Sigma^*) \\ fv & = & wfx & (w \in \Sigma^+, x \in \Sigma^*) \\ Pos_s(f) & = & |u| + 1 & \end{cases}$$

then we call the pair $(|u| + 1, |uw| + 1)$ a *contraction* of s by f, and $s' = ufx$ is the result of this contraction.

Notation 2.3 Given a word $s \in \Sigma^*$ and a Canonical Factor $f \in \mathcal{F}_s$, we denote by $\mathcal{C}_s^f$ the set of the contractions of s by f. We denote the set of all the contractions we can operate on s by $\mathcal{C}_s^*$ ($\equiv \bigcup_{f \in \mathcal{F}_s} \mathcal{C}_s^f$). Let t be a suffix of $s = t't$ ($t' \in \Sigma^*$), we denote by $\mathcal{C}_{s,t}^*$ the subset of $\mathcal{C}_s^*$ such that $\mathcal{C}_{s,t}^* = \{(p', q') \mid (p, q) \in \mathcal{C}_s^* \ \wedge \ p > |t'| \ \wedge \ (p', q') = (p - |t'|, q - |t'|)\}$.

Since contractions will be used to produce new words, we only need to consider a subset of the set of contractions.

Definition 2.6 A set $\mathcal{C}$ of contractions is *coherent* if and only if it does not contain two contractions (i_1, j_1), (i_2, j_2) such that: $i_1 < i_2 < j_1 < j_2$. Furthermore, we say that $\mathcal{C}$ is *minimal* if and only if it does not contain two contractions (i_1, j_1) and (i_2, j_2) such that $i_1 \leq i_2 < j_2 \leq j_1$ or such that $i_1 < j_1 = i_2 < j_2$.

Now we can define the operation that, given a word, allows us to build some new specific words.

Definition 2.7 Given a word $s \in \Sigma^*$ and a coherent and minimal set of contractions $\mathcal{C} = \{(p_1, q_1), \ldots, (p_k, q_k)\}$ (associated to the set of canonical factors $\{f_1, \ldots, f_k\}$), then we define the function $Word$ as following:

$$\begin{aligned} Word(s, \mathcal{C}) &= s[1..p_1 - 1]\, s[q_1..p_2 - 1] \ldots s[q_{k-1}..p_k - 1]\, s[q_k..|s|] \\ &= s[1..p_1 - 1]\, f_1\, s[q_1 + |f_1|..p_2 - 1] \ldots\, f_k\, s[q_k + |f_k|..|s|] \end{aligned}$$

We call this sequence *the result of the contractions from $\mathcal{C}$ applied to s.*

From now, we only consider coherent and minimal sets of contractions (since we are interested in the results of contractions, it is easy to see why other sets don't need to be considered anymore). Let us notice that whatever the order of contraction, the obtained word remains the same.

Definition 2.8 We define $\mathcal{E}(s) = \bigcup_{\mathcal{C} \subseteq \mathcal{C}_s^*} Word(s, \mathcal{C})$, and we call this set the *closure* of s.

To illustrate the various definitions given above, we take the example *gaccattctc* (*cf.* figures 1 and 2). Then the set of Canonical Factors is $\mathcal{F}_{gaccattctc} = \{a, c, ca, t, tc, ct\}$, and $\mathcal{C}^*_{gaccattctc} = \{(2,5), (3,4), (3,8), (3,10), (6,7), (6,9), (7,9)\}$. Let $\mathcal{C} = \{(2,5), (7,9)\}$ ($\mathcal{C} \subseteq \mathcal{C}^*_{gaccattctc}$). Hence $Word(gaccattctc, \mathcal{C}) = g\cancel{acc}\mathbf{at}\cancel{tc}\mathbf{tc} = gattc$. The closure of *gaccattctc* is:

$$\mathcal{E}(gaccattctc) = \left\{ \begin{array}{l} gac, gacatc, gacatctc, gacattc, gacattctc, gaccatc, gaccatctc, \\ gaccattc, gaccattctc, gactc, gatc, gatctc, \mathbf{gattc}, gattctc \end{array} \right\}$$

3 Characterization of the language recognized by Oracles

Given a word $s \in \Sigma^*$, we saw how to build the corresponding Factor (resp. Suffix) Oracle. This Oracle allows to recognize at least all the factors (resp. suffixes) of s. Nevertheless, it accepts a certain number of additional words too. For example the word *atc* is accepted by the Factor (resp. Suffix) Oracle of *gaccattctc* (*cf.* figures 1 and 2), whereas it is either a factor nor a suffix of *gaccattctc*. We defined above the set $\mathcal{E}(s)$. In this part, we show that the Suffix Oracle exactly recognizes all the suffixes of the words from $\mathcal{E}(s)$. Then, we use this result to show that the Factor Oracle recognizes exactly all the factors of the words from $\mathcal{E}(s)$.

We first recall some useful lemmas of [4].

Lemma 3.1 *[4, p. 3] Given a word $s \in \Sigma^*$ and an integer i $(0 \leq i \leq |s|)$, then $min(e_i)$ is suffix of all word recognized in the state e_i of the Oracle of s.*

Lemma 3.2 *[4, p. 4] Given a word $s \in \Sigma^*$ and a factor w of s, then w is recognized in the state e_i $(1 \leq i \leq poccur_s(w))$ of the Oracle of s.*

Lemma 3.3 *[4, p. 4] Given a word $s \in \Sigma^*$ and an integer i $(0 \leq i \leq |s|)$, then every path ending by $min(e_i)$ in the Oracle of s leads to a state e_j such that $j \geq i$.*

Lemma 3.4 *[4, p. 5] Given a word $s \in \Sigma^*$ and $w \in \Sigma^*$ a word accepted by the Oracle of s in state e_i, then every suffix of w is also recognized by the Oracle in state e_j such that $j \leq i$.*

The proof of this last Lemma is given in [4] only for the Factor Oracle. We need to extend this result for the Suffix Oracle.

Proof (Lemma 3.4)
If we denote by x a suffix of w, the original Lemma gives us that $State(x) \leq State(w)$. We need to prove that if $State(w)$ is final, then $State(x)$ is final. In order to do this, we have to consider two cases:

Case 1: $|x| \geq |min(e_i)|$
That means that $min(e_i) \in Suff(x)$, thus according to Lemma 3.3, we can conclude that $State(x) \geq State(min(e_i))$, and since $State(min(e_i)) = e_i = State(w)$, then $State(x) = State(w)$.

Case 2: $|x| < |min(e_i)|$
The state e_i being final means that there exists a suffix t of s such that $State(t) = e_i$. According to Lemma 3.1, we deduce that $min(e_i) \in Suff(t) \subseteq Suff(s)$. Since x and $min(e_i)$ are suffixes of w, then $|x| < |min(e_i)| \Rightarrow x \in Suff(min(e_i))$. So x is also suffix of s and, by Definition of the Suffix Oracle, $State(x)$ is final. □

Before tackle demonstrations, we present two lemmas dealing with properties linked to Canonical Factors.

Lemma 3.5 *Given a word $s \in \Sigma^*$, a Canonical Factor $f \in \mathcal{F}_s$ such that $s = ufv$ $(u, v \in \Sigma^*)$ and f is not repeated in uf, and $\mathcal{C} \in \mathcal{C}^*$ a set of contractions. If there exists $w \in \Sigma^*$ such that $Word(uf, \mathcal{C}) = wf$ then wf and f are recognized in the same state in the Oracle of s.*

Proof (Lemma 3.5)
We denote by $\mathcal{C}_i \subseteq \mathcal{C}_s^*$ a set of contractions having cardinality i. In the same way, we denote by $w_i f$ the word obtained applying contractions $\mathcal{C}_i$ to uf (warning: $w_i f = Word(uf, \mathcal{C}_i) \not\Rightarrow w_i = Word(u, \mathcal{C}_i)$). Let us show by induction on the size of $\mathcal{C}_i$ that $State(Word(uf, \mathcal{C}_i)) = State(f)$ $(\forall\, \mathcal{C}_i \in \mathcal{C}_s^*)$.
Let $e_x = State(f)$ ($f = min(e_x)$ by Definition of f) and $e_{x'_i} = State(Word(uf, \mathcal{C}_i))$. If we consider $\mathcal{C}_0$, then $Word(uf, \mathcal{C}_0) = uf$. According to Lemma 3.3, $x'_0 \geq x$. Furthermore, according to Lemma 3.2 applied to uf, we have $x'_0 \leq poccur_s(uf)$. However by Definition of f, $poccur_s(f) = |uf| = poccur_s(uf)$. This implies $x'_0 \leq x$, and finally $x'_0 = x$.

Let us show now that if this lemma is true for a set of contractions $\mathcal{C}_i \subset \mathcal{C}_s^*$, then it is true for a set $\mathcal{C}_{i+1} = \mathcal{C}_i \cup \{(p,q)\}$. We assume without loss of generality that (p,q) is the last contraction (by ascending order over the positions) in $\mathcal{C}_{i+1}$. Let b the Canonical Factor used by this contraction. We can write $uf = s[1..p-1]\, s[p..q-1]\, s[q..|uf|]$. Since we choose (p,q) being the last contraction, all the contractions in $\mathcal{C}_i$ are applicable to $s[1..p-1]$. So there exists $a, c \in \Sigma^*$ such that $w_i f = a\, s[p..|uf|] = abc$, and $d \in \Sigma^*$ such that $w_{i+1} f = a\, s[q..|uf|] = abd$. We also could write $ab = Word(s[1..p-1]\, b, \mathcal{C}_i)$ (the opposite would mean that the contraction (p,q) can't be operate from b), and according to the induction hypothesis, we have $State(ab) = State(b)$. From this, we deduce that $State(abc) = State(bc)$ and $State(abd) = State(bd)$. Since $bd(= s[q..|uf|])$ is a suffix of $bc(= s[p..|uf|])$, according to the Lemma 3.4:

$$\begin{array}{rlcl} & State(bd) & \leq & State(bc) \\ \Leftrightarrow & State(abd) & \leq & State(abc) \\ \Leftrightarrow & State(w_{i+1}f) & \leq & State(w_i f) \\ \Leftrightarrow & State(w_{i+1}f) & \leq & State(f) \end{array}$$

But, according to Lemma 3.3, we have $State(w_{i+1}f) \geq State(f)$, consequently we obtain $State(w_{i+1}f) = State(f)$. So, this lemma is true for all $\mathcal{C}_i \subseteq \mathcal{C}_s^*$. □

Lemma 3.6 *Let s be word in Σ^*, $O(s)$ be its Oracle, and e_i be a state of $O(s)$ such that $u = min(e_i)$ and $u \in \mathcal{F}_s$. Let p be a transition issued from e_i labeled by α to a state e_{i+j} $(j > 1)$. Then there exists at the position $(i + j - |u|)$ of s an occurrence of $u\alpha$. Moreover, we have the contraction $(i - |u| + 1, i + j - |u|)$ of s by u.*

Proof (Lemma 3.6)
By construction (*cf.* algorithm 1), the transition p from e_i to e_{i+j} is added because there exists a position j in $s[i - |u| + 1..|s|]$ such that: $j = poccur_{s[i-|u|+1..|s|]}(u\alpha) - |u|$. We also have $u\alpha \in Fact(s)$ since $u\alpha \in Fact(s[i - |u| + 1|s|])$. Cleophas & *al.* [11] have proved that since $u = min(e_i)$ and $u\alpha \in Fact(s)$, then $i - |u| + poccur_{s[i-|u|+1..|s|]}(u\alpha) = poccur_s(u\alpha)$. Hence, we have $i + j = poccur_s(u\alpha)$, and finally $s[i + j - |u|, i + j] = u\alpha$. □

Algorithm 2: Obtaining the contractions generating w starting from t in the Oracle of s

Initialization: $S^0 = t$, $S^0_w = w$, $\mathcal{C}_0 = \emptyset$, $sdec = |s| - |t|$ % t is a suffix of s %

Input: $S^i \in \Sigma^*$ % A suffix of s that can still be ''contracted'' %
$S^i_w \in \Sigma^*$ % The word to process %
$\mathcal{C}_i$ % Set of contractions %
Output: a set of contractions

Begin
$p_i \leftarrow$ longest common prefix between S^i and S^i_w (Property 3.1, item 1)
$e_{r_i} \leftarrow State(p_i)$ (Property 3.1, item 2)
$f_i \leftarrow min(e_{r_i})$
If $(|p_i| < |S^i_w|)$ **Then**
$e_{r'_i} \leftarrow$ Transition$(e_{r_i}, S^i_w[|p_i| + 1])$ (Property 3.1, item 4)
$\mathcal{C}_{i+1} \leftarrow \mathcal{C}_i \cup \{c_i\}$, $c_i = (r_i - |f_i| + 1 - sdec, r'_i - |f_i| - sdec)$ (Property 3.2, item 2)

15 $\quad\quad S_w^{i+1} \leftarrow S_w^i[|p_i| - |f_i| + 1..|S_w^i|]$ (Property 3.1, item 3)
16 $\quad\quad S^{i+1} \leftarrow t[r'_i - |f_i| - sdec..|t|]$ (Property 3.1, item 3)
17 $\quad\quad$ **Return** <u>Contractor</u>$(S^{i+1}, S_w^{i+1}, \mathcal{C}_{i+1})$
18 $\quad$ **Else**
19 $\quad\quad$ **If** $(|S^i| > |S_w^i|)$ **Then**
20 $\quad\quad\quad \mathcal{C}_{i+1} \leftarrow \mathcal{C}_i \cup \{c_i\}$, $c_i = (r_i - |f_i| + 1 - sdec, |s| - |f_i| + 1 - sdec)$ (Property 3.3)
21 $\quad\quad$ **Else**
22 $\quad\quad\quad \mathcal{C}_{i+1} \leftarrow \mathcal{C}_i$ (Property 3.3)
23 $\quad\quad$ **End If**
24 $\quad\quad$ **Return** $\mathcal{C}_{i+1}$
25 $\quad$ **End If**
26 **End**

Our goal in this part is to give a characterization of the language accepted by the Oracle of a word s. To do that, we use the algorithm *Contractor* (*cf.* algorithm 2). Given a word $s \in \Sigma^*$ and its Suffix Oracle $SO(s)$, *Contractor* needs a word w accepted by $SO(s)$ and a suffix t of s chosen such that $\begin{cases} w[1] & = t[1] \\ |t| & \text{maximal} \end{cases}$. The result of *Contractor* is a set $\mathcal{C}$ of contractions such that $w = Word(t, \mathcal{C})$. After a first brief presentation of *Contractor*, we will introduce the notations of the algorithm.

We saw (in the Definition) that $Word(t, C)$, for a set of contractions C, is a concatenation of substrings of t. We can see these sub-words as prefixes of suffixes of t. A jump from one substring to the next one is a contraction. The question is now how to find the correct suffixes and their prefixes. The answer is *Contractor*. This is a recursive algorithm that finds all the contractions used to contract t in w, by searching the suffixes of t which we talk about. The main idea of *Contractor* is to read the words t and w from left to right, and when the one-to-one characters differ, to use a contraction in t to reach a further position in order to allows the reading of the same characters than w.

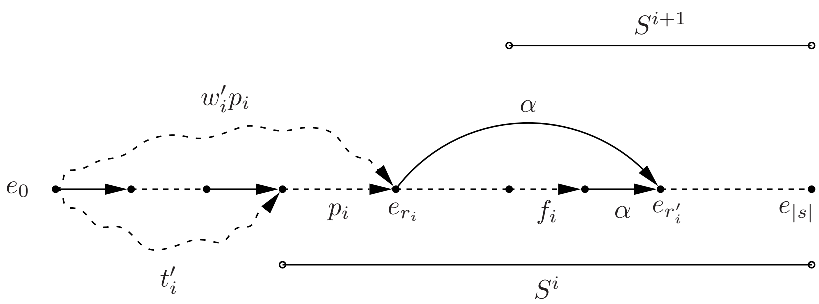


Figure 3: Illustration of a step in the algorithm *Contractor* ($\alpha = S_w^i[|p_i| + 1]$).

The inputs are words S^i and S_w^i ($i \geq 0$), and $\mathcal{C}_i$ a set of contractions. Initially, we have $S^0 = t$, $S_w^0 = w$ and $\mathcal{C}_0 = \emptyset$. We denote by p_i (line 9) the longest prefix of S^i and S_w^i. So, we can write:

$$\begin{cases} S^i & = & p_i S'^i \\ S_w^i & = & p_i S_w'^i \end{cases} \tag{3.1}$$

Let $e_{r_i} = State(p_i)$ (line 10) and $f_i = min(e_{r_i})$ (line 11). Due to Lemma 3.1, we have:

$$p_i = p'_i f_i \ (p'_i \in \Sigma^*) \tag{3.2}$$

About the other variables, $e_{r'_i}$ (line 13) is the state reached by the transition from e_{r_i} and labeled by $\alpha = S_w^i[|p_i|+1] = S_w'^i[1]$, $\mathcal{C}_{i+1}$ is a set of contractions (which has cardinality $i+1$). We need to use the variable $sdec = |s| - |t|$ to translate the indexes of each contraction. Indeed, the positions for a contraction are computed using the indexes of the states (each state e_i is linked to the i^{th} character of s, not to the character $(i - |s| + |t|)$ of t). Thus, a contraction would be correct for s, but not for t Hence, we proceed as for the Definition of $\mathcal{C}^*_{s,t}$, *ie.* we remove $|s| - |t|$.

The figures 3 and 4 illustrates *Contractor*, and are useful to understand the properties below. The following Property 3.1 claims some interesting characteristics of the variables used by *Contractor*.

Property 3.1
For all $i \geq 0$, the following assertions are true:

1. $f_i\alpha \in Pref(p_{i+1})$.
2. $S^i = t[r_i - |p_i| + 1 - sdec..|t|]$.
3. S^{i+1} and S_w^{i+1} are respectively suffixes of S^i and S_w^i; S^i and S_w^i $(i \geq 0)$ are respectively suffixes of t and w.
4. The transition from e_{r_i} to $e_{r'_i}$ and labeled by α always exists.

Proof (Property 3.1)

1. Since $f_i = min(e_{r_i})$, and according to Lemma 3.6, we can write $s[r'_i - |f_i|..r'_i] = t[r'_i - |f_i| - sdec..r'_i - sdec] = f_i\alpha$. So S^{i+1} begins with $f_i\alpha$, and S_w^{i+1} too (line 15).
2. For $i = 0$ (initialization case), $S^0 = t$ and t is the longest suffix of s beginning by $w[1]$. Then we can easily see that if $e_q = State(S^0[1])(q > 0)$, then $t[q - sdec..|t|] = S^0$ and $State(p_0) = q + |p_0| - 1 = e_{r_i}$. Thus $S^0 = s[r_0 - |p_0| + 1 - sdec..|s|]$.
   Now, let us see the recursive case. We have $S^{i+1} = t[r'_i - |f_i| - sdec..|t|]$ (*Contractor*, line 16). Since S_w^{i+1} begins by $f_i\alpha$ (*cf.* item 1), $r_{i+1} = r'_i + |p_{i+1}| - |f_i| - 1$. Finally
   $$S^{i+1} = t[r'_i - |f_i| - sdec..|t|] = t[r_{i+1} - |p_{i+1}| + 1 - sdec..|t|].$$
3. This is obvious for S_w^i because S_w^{i+1} is suffix of S_w^i by construction (line 15) and $S_w^0 = w$. Concerning S^i, we have $S^0 = t$ thus the property is true for $i = 0$. Let us suppose that S^i is suffix of t, and show it for $i+1$. We prove now that S^{i+1} is suffix of S^i. From the preceding point (item 2), we have $S^i = t[r_i - |p_i| + 1 - sdec..|t|]$. In *Contractor*, we have $S^{i+1} = t[r'_i - |f_i| - sdec..|t|]$ (line 16). According to equality 3.2, $r_i - |p_i| = r_i - |p'_i| - |f_i|$. Because $|p'_i| \geq 0$, we obtain $r_i - |f_i| \geq r_i - |p_i|$. Furthermore $r'_i > r_i$. Finally $r'_i - |f_i| > r_i - |p_i|$ and S^{i+1} is a suffix of S^i.
4. According to item 3 in this Property, S_w^i is suffix of w. Then S_w^i is recognized by $O(s)$ (Lemma 3.4). According to equality 3.1 with $S_w'^i[1] = \alpha$, the transition must exists. That implies that $\#_{out}(e_{r_i}) \geq 2$, and then, by Definition of the Canonical Factors, we deduce that $f_i = min(e_{r_i}) \in \mathcal{F}_s$.

□

From equality 3.1 and the above Property 3.1 (item 4), we can write:

$$\left\{\begin{array}{lllll} t & = & t'_i S^i & & (t'_i \in \Sigma^*) \\ w & = & w'_i S^i_w & = \; w'_i p'_i S^{i+1}_w & (w'_i \in \Sigma^*) \end{array}\right. \tag{3.3}$$

Before giving more explanations about *Contractor*, we need to prove the items of the following property.

Property 3.2
For all $i \geq 0$:

1. $State(w'_i p_i) = State(t'_i p_i) = State(p_i) = e_{r_i}$.
2. c_i is a contraction of $t'_i S^i$ (resp. $w'_i S^i$) by f_i. The result of this contraction is $t'_i p'_i S^{i+1}$ (resp. $w'_i p'_i S^{i+1} = w'_{i+1} S^{i+1}$).

Proof (Property 3.2)

1. This is obvious for $i = 0$ because $t'_i = w'_i = \epsilon$. Let us suppose the property is true for i, and prove this is true for $i+1$. From Property 3.1 (item 2), we deduce that the word read in $O(s)$ starting from $e_{r_i - |p_i|}$ to $e_{|s|}$ by using only "main" transitions (*ie.* transitions of type $e_j \to e_{j+1}$) is S^i. According to Property 3.1 (item 3) we deduce:
$$S^i = uS^{i+1} \; (u \in \Sigma^*) \tag{3.4}$$
So, there exists the state e_q ($q > r_i - |p_i|$) such that the word read from e_q to $e_{|s|}$ using only "main" transitions is S^{i+1}. In particular, $q = r'_i - |f_i| - 1$. We have $t'_{i+1} = t'_i u$ (*cf.* equality 3.3 and 3.4) and $State(t'_i u) = e_q$. Then, since $f_i = min(e_{r_i})$ and since there exists a transition from e_{r_i} to $e_{r'_i}$ labeled by α (*cf.* Property 3.1, item 4), we have $State(t'_{i+1} f_i \alpha) = State(t'_i u f_i \alpha) = State(f_i \alpha) = e_{r'_i}$. Furthermore $p_{i+1} = f_i \alpha v$ ($v \in \Sigma^*$). So we can deduce that $State(t'_{i+1} f_i \alpha v) = State(t'_{i+1} p_{i+1}) = State(p_{i+1})$.
2. From the equalities 3.1, 3.2 and 3.3, we deduce that:
$$t = t'_i S^i = t'_i p'_i f_i S'^i \tag{3.5}$$
Since $S^{i+1} \in Suff(S^i)$, we have $S^i = uS^{i+1}$ ($u \in \Sigma^+$). Hence, we deduce from equality 3.5 that $t'_i p'_i f_i S'^i = t'_i u S^{i+1}$. According to the Property 3.1 (item 1), we have $t'_i p'_i f_i S'^i = t'_i u f_i \alpha u'$ ($u' \in \Sigma^*$). Because we have $State(t'_i p'_i f_i) = State(f_i)$ and $|u| > |p'_i|$ ($S'^i[1] \neq \alpha$), we can contract $t'_i S^i$ by f_i; the result is:
$$t'_i p'_i f_i \alpha u' = t'_i p'_i S^{i+1} \tag{3.6}$$
Since $State(w'_i p_i) = State(t'_i p_i)$, we can deduce that $w'_i S^i = w'_i p_i S'^i$ is contracted by f_i in $w'_i p'_i S^{i+1}$. According to equality 3.3, we deduce that $w'_{i+1} = w'_i p'_i$. Then $w'_i p'_i S^{i+1} = w'_{i+1} S^{i+1}$.

□

The Property 3.2 shows us that c_i (a contraction of $t'_i S^i$ by f_i) is a contraction for t and, more interesting, for $w'_i S^i$. Before concluding about these contractions, we need to examine the termination of *Contractor* and its final case. For all $i \geq 0$, we have either $|S^i_w| > |S^{i+1}_w|$, nor $|S^i_w| = |S^{i+1}_w|$ and $|p_{i+1}| > |p_i|$ (if $f_i = p_i$). Since $p_i > 0$, we deduce that we finally obtain $p_j = S^j_w$ ($j > i$). The following property concerns the final cases of *Contractor*.

Property 3.3
Let $j \geq 0$ be such that $p_j = S^j_w$. If $|S^j_w| \neq |S^j|$, then t needs a last contraction. Else $\mathcal{C}_j$ is the final set.

Proof (Property 3.3)
The word obtained up to now with the contraction of $\mathcal{C}_j$ is $w'_j p_j S'^j$ (*cf.* Property 3.2, item 2). If $S^j_w = S^j$, then $S'^j = \epsilon$ and $\mathcal{C}_j$ is complete (line 20). According to Property 3.2 (item 1), we have $State(w'_j p_j) = e_{r_j}$. Thus $min(e_{r_j}) \in Suff(w)$ (Lemma 3.1) and $min(e_{r_j}) \in Suff(t)$ (by Definition of the final state in a Suffix Oracle). Then a last contraction completes the set of contractions (line 22). □

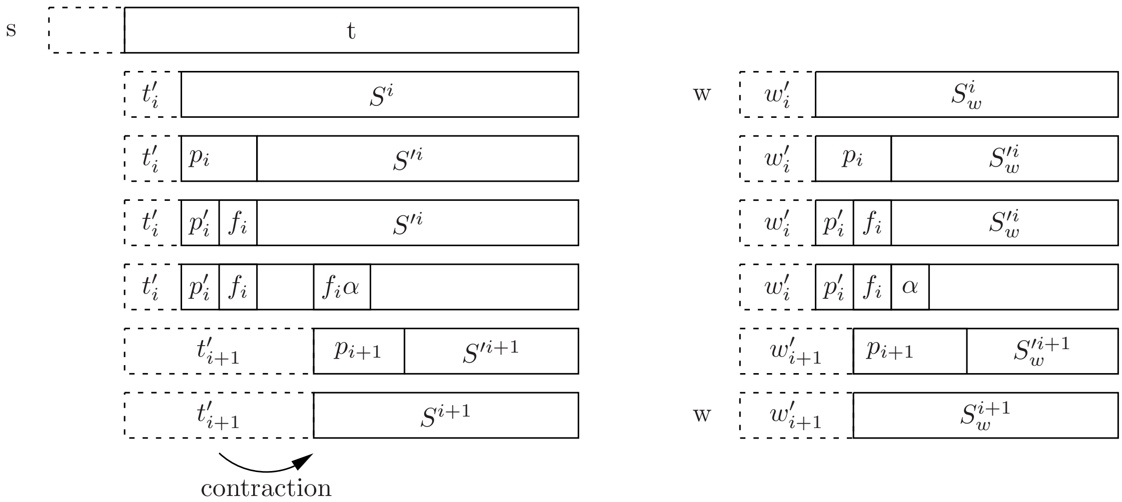


Figure 4: Visualization of *Contractor* on S^i and S^i_w.

Now, let us see how a step of *Contractor* works. We consider the i^{th} call of *Contractor*, whose inputs are $S^i = p_i S'^i$, $S^i_w = p_i S'^i_w$ and $\mathcal{C}_i$. The contractions already used to contract the beginning of t (*ie.* t'_i) into the beginning of w (*ie.* w'_i) are in $\mathcal{C}_i$. At this point we consider the longest common prefix (denoted by p_i) of S^i and S^i_w (p_i is both a factor of t and w, Property 3.1). The algorithm has two cases. If $|p_i| = |S^i_w|$, we are in a final case we described above. Else, the prefix p_i is not S^i_w, and then we need at least one other contraction until $|p_i| = |S^i_w|$. Thus we search for another suffix S^{i+1} of t with which we can continue to contract. From Property 3.2, we have the contraction is the right one, and we continue with the suffix S^{i+1}. When we reach the end of the process (*ie.* the end of w), we return the last up-to-date set $\mathcal{C}_{i+1}$ and $w = Word(t, \mathcal{C}_{i+1})$.

We can notice that:

1. $\mathcal{C}$ is not always minimal. The algorithm could be modified but would become more difficult to understand. However, the minimality is not an objective here.

2. $\mathcal{C}$ is coherent. Let (a, b) and (c, d) be two contractions added successively to $\mathcal{C}$. We have $a < b$ and $c < d$ because $r'_i > r_i$ and $|s| > r_i$ (*cf.* lines 14 and 20). Next, either $e_{r_{i+1}} = State(p_{i+1}) = e_{r'_i}$ and so $b = c$, or $e_{r_{i+1}} > e_{r'_i}$ (because we can have $p_{i+1} = f_i \alpha v$ ($\alpha = S^i_w[|p_i| + 1]$) where $v \neq \epsilon$, and thus $b < c$.

Lemma 3.7 *Given a word $s \in \Sigma^*$, its Suffix Oracle, a word $w \in \Sigma^*$ accepted by $SO(s)$, and t being the longest suffix of s such that $w[1] = t[1]$, then* Contractor$(t, w, \emptyset)$ *returns a set $\mathcal{C}$ such that $w = Word(t, C)$.*

Proof (Lemma 3.7)
Let $j \geq 0$ such that $S^{j+1}_w = p_{j+1}$. Then, according to Property 3.2, we deduce that $\mathcal{C}_{j+1}$ is a coherent set of contractions of t. Then, we have:

$$Word(t, \mathcal{C}_{j+1}) = w'_{j+1} S^{j+1} = w'_{j+1} S^{j+1}_w u = w'_{j+1} p_{j+1} u \quad (u \in \Sigma^*)$$

because p_{j+1} is prefix of S^{j+1}. If $u = \epsilon$, we have $Word(t, \mathcal{C}_{j+1}) = w$ (equality 3.3). Else (*cf.* Property 3.3) a ultimate contraction c_{j+1} contracts $w'_{j+1} S^{j+1}_w u$ by f_{j+1} in $w'_{j+1} S^{j+1}_w = w = Word(t, \mathcal{C}_{j+1} \cup \{c_{j+1}\})$.
Finally *Contractor* provide a set $\mathcal{C}$ such that $w = Word(t, \mathcal{C})$. □

The following two theorems are the main purpose of this paper.

Theorem 3.1 *Exactly all the suffixes of the words from $\mathcal{E}(s)$ are recognized by the Suffix Oracle of s .*

Proof (Theorem 3.1)
'$\Rightarrow$': *Each suffix of a word from $\mathcal{E}(s)$ is recognized by the Suffix Oracle of s.*
According to Lemma 3.4, if w is accepted by $SO(s)$, then each suffix of w is also accepted by $SO(s)$, so we only need to prove that each word from $\mathcal{E}(s)$ is accepted by $SO(s)$.
Let $\mathcal{C} \in \mathcal{C}^*_s$ be a set of contractions applicable to s. Let us build $w = Word(s, \mathcal{C})$, and show that w is accepted by $SO(s)$. Let $\mathcal{C}_i$ be the set of the first i contractions of $\mathcal{C}$ (chosen without loss of generality by ascending order over the positions), (x_j, y_j) be the j^{th} contraction, and $f_j \in \mathcal{F}_s$ the Canonical Factor used by (x_j, y_j) $(1 \leq j \leq i)$. We note $w_j = Word(s, \mathcal{C}_j)$. The property (P) to check is that w_i is accepted by $SO(s)$. Because $w_0 = s$, the property (P) is true for $i = 0$. Let us suppose that it is true for i, and show that (P) is true for $i + 1$. We have:

$$\left\{ \begin{array}{lcl} w_i & = & s[1..x_1 - 1]\, s[y_1..x_2 - 1] \ldots s[y_i..|s|] \\ s[y_i..y_i + |f_i| - 1] & = & f_i \end{array} \right.$$

By Definition of the Canonical Factors, f_{i+1} does not occur in s before the position x_{i+1} $(x_{i+1} > y_i)$. Thus we can write, in particular, w_i and w_{i+1} as:

$$\left\{ \begin{array}{lcl} w_i & = & v' f_{i+1} u \\ w_{i+1} & = & v' f_{i+1} u' \end{array} \right. \text{with} \left\{ \begin{array}{lcll} v' & = & s[1..x_1 - 1] s[y_1..x_2 - 1] \ldots s[y_i..x_{i+1} - 1] & \\ f_{i+1} u & = & u'' f_{i+1} u' & (u'' \in \Sigma^+) \end{array} \right.$$

Because the contraction concerns (x_{i+1}, y_{i+1}), then we also have $|s|-|f_{i+1}u|+1 = x_{i+1}$ and $|s|-|f_{i+1}u'|+1 = y_{i+1}$. This is true because the contractions are in ascending order, so the word s is not yet modified after the positions x_{i+1} and y_{i+1} (hence $f_{i+1}u$ and $f_{i+1}u'$ are suffixes of s). Let q be the state of $SO(s)$ such that $q = State(f_{i+1})$. According to the Lemma 3.5:

$$State(v' f_{i+1}) = q \tag{3.7}$$

Furthermore, $f_{i+1}u'$ is a suffix of s, so it is necessary recognized by $SO(s)$. This requires to pass through the state q when the word $f_{i+1}u'$ is read in $SO(s)$. Thus, starting from q, we can read u', and reach a final state. So, according to equality 3.7, $w_{i+1} = v' f_{i+1} u'$ is accepted by $SO(s)$. To conclude, we just showed that w_i is recognized by $SO(s)$, for all $i \leq |\mathcal{C}|$. Finally, Lemma 3.4 allows to conclude that each suffix of a word of $\mathcal{E}(s)$ is recognized by $SO(s)$.

'$\Leftarrow$': *Each word recognized by the Suffix Oracle of s is suffix of a word from* $\mathcal{E}(s)$.
Let w be a word accepted by the Suffix Oracle of s, and t be the longest suffix of s ($s = s't$) such that $w[1] = t[1]$. Then there exists a set $\mathcal{C}$ of contractions such that $w = Word(t, \mathcal{C})$ (Lemma 3.7). Since $\mathcal{C} \subseteq \mathcal{C}^*_{s,t}$, there exists a set $\mathcal{C}' \subseteq \mathcal{C}^*_s$, obtained by translating the indexes of $\mathcal{C}$ with *sdec*, such that $s'w = Word(s't, \mathcal{C}')$. Because $s'w \in \mathcal{E}(s)$, we can conclude that each word accepted by $SO(s)$ is a suffix of a word from $\mathcal{E}(s)$. □

On the basis of this previous result, we can give a similar theorem, which is available for the Factor Oracle instead of being available for the Suffix Oracle.

Theorem 3.2 *Exactly all the factors of the words from $\mathcal{E}(s)$ are recognized by the Factor Oracle of s .*

Proof (Theorem 3.2)
'$\Rightarrow$': *Each factor m of a word from $\mathcal{E}(s)$ is recognized by the Factor Oracle of s.*
Let $SO(s)$ be the Suffix Oracle of s, and $u \in \mathcal{E}(s)$ be such that m is prefix of a suffix of u, denoted by mv ($v \in \Sigma^*$). Then mv is accepted by $SO(s)$ (*cf.* Theorem 3.1), thus there exists a single path $(e_0 \to e_{x_1} \to \ldots \to e_{x_{|m|+|v|}})$ in $SO(s)$ that recognizes mv. Therefore, there exists a path $(e_0 \to e_{x_1} \to \ldots \to e_{x_{|m|}})$ (with $e_{x_{|m|}}$ final) that recognizes m.

'$\Leftarrow$': *Each word m recognized by the Factor Oracle of s is factor of a word from* $\mathcal{E}(s)$.
Let $SO(s)$ be the Suffix Oracle of s. If m is recognized by $SO(s)$ then m is a suffix of a word of $\mathcal{E}(s)$ (*cf.* Theorem 3.1). Let us suppose that m is not recognized by $SO(s)$, then m is recognized by $FO(s)$ in the state $e_{x_{|m|}}$ (not final in $SO(s)$). By construction, $e_{x_{|m|}} \in \{e_k | 0 \leq k \leq |s|\}$, the set of the states of $FO(s)$, with $(e_0 \to e_1 \to \ldots \to e_{|s|})$ the path that accepts the word s itself (with $e_{|s|}$, among others, final in $SO(s)$). Thus, there exits a path from $e_{x_{|m|}}$ to $e_{|s|}$ in $SO(s)$. So, m is prefix of a word recognized by $SO(s)$. That implies that m is prefix of a suffix of some $u \in \mathcal{E}(s)$. Therefore, m is a factor of a word of $\mathcal{E}(s)$. □

4 Properties upon Oracles & Future Works

In the conclusion of their article, Cleophas & *al.* [11] show that the Oracle is not minimal in number of transitions among the set of homogeneous automata.

Furthermore, if we consider the set of homogeneous automata recognizing at least all the factors (resp. suffixes) of s, having the same number of states and at most the same number of transitions than the Factor (resp. Suffix) Oracle, we show that the Oracle is not minimal on the number of accepted words. We can see that the Oracle of *axttyabcdeatzattwu* (*cf.* figure 5) has 35 transitions. The Factor Oracle accepts 247 words and the Suffix Oracle accepts 39 words, though there exists another homogeneous automaton (*cf.* figure 6) recognizing at least all the factors (resp. suffixes) of *axttyabcdeatzattwu*, and having only 34 transitions. The "Factor" version of this automaton recognizes only 236 words and its "Suffix" version accepts only 30 words. This example shows that the Oracle is not minimal in number of accepted words among the set of homogeneous automata having the same number of states and *less* transitions.

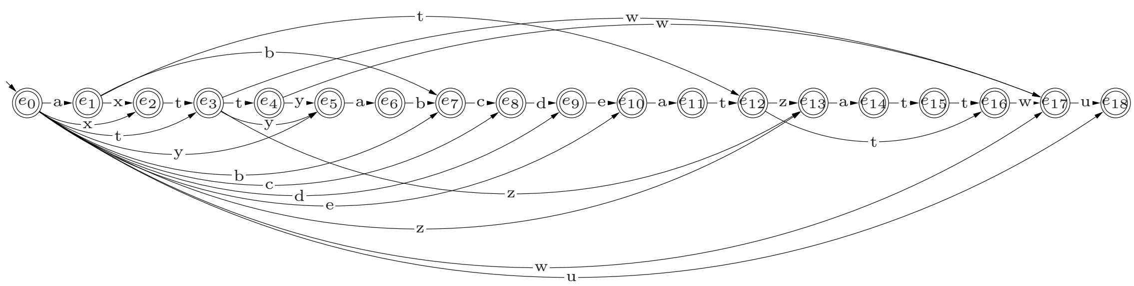


Figure 5: Factor Oracle of the word *axttyabcdeatzattwu*.

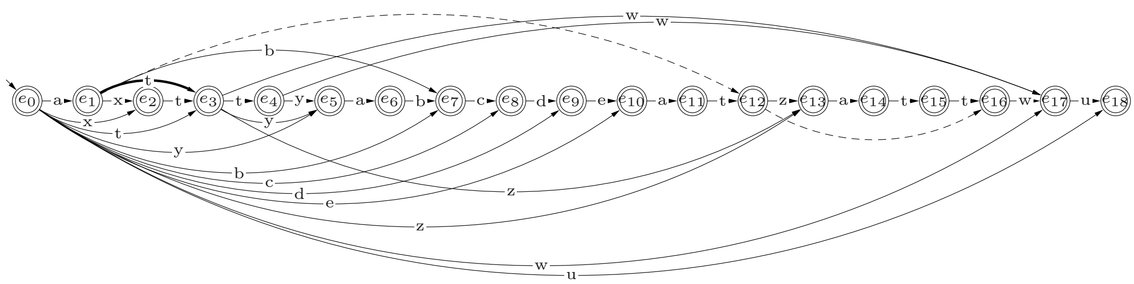


Figure 6: This automaton (considering only the continuous lines) accepts at least all the factors of the word *axttyabcdeatzattwu*. The bold transition is the only one which is not present in the Factor Oracle of this word (*cf.* figure 5) though the two dotted ones are present in the Factor Oracle, but not in this automaton.

We observe that, in some cases, the number of words accepted by Oracles does not allow to give confidence to this structure when it is used for detect factors or suffixes of a word. Because, even if the number of false positive can sometimes be null (*eg.* $aaaaaa\ldots$), it can also be exponential. Indeed, we can build a word s such that each subset of $\mathcal{C}_s^*$ is coherent and minimal. For example: $s = aabbccddee\ldots$. The set $\mathcal{C}_s^*$ of contractions which are available on such a word is $\{(1,2),(3,4),(5,6),\ldots,(|s|-1,|s|)\}$. If we consider any (non-empty) subset $\mathcal{C} \subseteq (\mathcal{C}_s^* \setminus \{(1,2)\})$ of contractions, it is easy to notice that $Word(s,\mathcal{C}) \notin Fact(s)$. Besides, all the words obtained from such subsets are pairwise different.

The number of these subsets is:

$$\sum_{i=1}^{|\mathcal{C}_s^*|-1} \binom{|\mathcal{C}_s^*|-1}{i} = \sum_{i=1}^{\frac{|s|}{2}-1} \binom{\frac{|s|}{2}-1}{i} = 2^{\frac{|s|}{2}-1} - 1$$

So the number of words that will be accepted by the Oracles but are not factor/suffix of s is $\mathcal{O}(2^{|s|})$.

In order to better benefit from this structure, it has to be improved, or to be slightly modified. However, it could be useful for future works to improve the knowledges about the Oracle structure. Effectively, it could be interesting to have either an empirical nor a statistical estimation of the accuracy (time and quality of the results) of the Oracle when substituted to Tries or Suffix Trees in algorithms.

References

[1] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: computer science and computational biology.* Cambridge University Press, 1997.

[2] Anselm Blumer, Janet Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel Seiferas. The smallest automaton recognizing the subwords of a text. *Theorical Computer Science*, 40(1):31–55, 1985.

[3] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[4] Cyril Allauzen, Maxime Crochemore, and Mathieu Raffinot. Oracle des facteurs, Oracle des Suffixes. Technical Report 99-08, Institut Gaspard-Monge, Université de Marne-la-Vallée, 1999.

[5] Cyril Allauzen, Maxime Crochemore, and Mathieu Raffinot. Factor Oracle: A New Structure for Pattern Matching. In *Conference on Current Trends in Theory and Practice of Informatics*, pages 295–310, 1999.

[6] Cyril Allauzen, Maxime Crochemore, and Mathieu Raffinot. Efficient Experimental String Matching by Weak Factor Recognition. In *Proceedings of the 12th conference on Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 51–72. Springer-Verlag, 2001.

[7] Arnaud Lefebvre and Thierry Lecroq. Computing repeated factors with a factor oracle. In L. Brankovic and J. Ryan, editors, *Proceedings of the 11th Australasian Workshop On Combinatorial Algorithms*, pages 145–158, Hunter Valley, Australia, 2000.

[8] Arnaud Lefebvre and Thierry Lecroq. A heuristic for computing repeats with a factor oracle: application to biological sequences. *International Journal of Computer Mathematics*, 79(12):1303–1315, 2002.

[9] Arnaud Lefebvre, Thierry Lecroq, Hélène Dauchel, and Joël Alexandre. FORRepeats: detects repeats on entire chromosomes and between genomes. *Bioinformatics*, 19(3):319–326, 2003.

[10] Arnaud Lefebvre and Thierry Lecroq. Compror: on-line lossless data compression with a factor oracle. *Information Processing Letters*, 83(1):1–6, 2002.

[11] Loek Cleophas, Gerard Zwaan, and Bruce Watson. Constructing Factor Oracles. In *Proceedings of the 3^rd^ Prague Stringology Conference*, 2003.

A Framework for the Dynamic Implementation of Finite Automata for Performance Enhancement

Ernest Ketcha Ngassam[1], Bruce W. Watson[2], and Derrick G. Kourie[2]

[1] School of Computing, University of South Africa,
Pretoria 0003, South Africa

[2] Department of Computer Science, University of Pretoria,
Pretoria 0002, South Africa

e-mail: ngassek@unisa.ac.za, {bwatson, dkourie}@cs.up.ac.za

Abstract. The aim of this work is to provide a model for the dynamic implementation of finite automata for enhanced performance. Investigations have shown that hardcoded finite automata outperforms the traditional table-driven implementation up to some threshold. Moreover, the kind of string being recognized plays a major role in the overall processing speed of the string recognizer. Various experiments are depicted to show when the advantages of using hardcoding as basis for implementing finite automata (instead of using the classical table-driven approach) become manifest. The model, a dynamic algorithm that combines both hardcoding and table-driven is introduced.

Keywords: Finite Automata, Hardcoding, Performance

1 Introduction

To the best of our knowledge, hardcoding of finite automata (FAs) in the context of right linear languages was first suggested by Knuth et al in [Kmp77]. An intensive investigation of the behavior of hardcoded FAs in comparison with the traditional table-driven approach was suggested by Ketcha in [Ket03]. A conclusion of Ketcha's work was that hardcoding outperforms table-driven up to some threshold. This threshold is clearly dependent on such factors as processor configuration, alphabet size and number of states of the FA under consideration. In fact, for the hardware configuration used in Ketcha's experiments, it has been found that a threshold at about 360 states is relatively robust for alphabet sizes in the range of 40 to 80. For smaller alphabet sizes it is less than 1000 states. In principle, therefore, experimentation can be used to derive a set of rules to determine an estimate of the breakeven point between these two implementation strategies.

In this paper, we propose the notion of dynamic implementation of FAs to enhance performance. Table-driven (TD) implementation of large FAs often implies some latency due to insufficient memory. The problem is then to design and implement an algorithm that takes into consideration the size of the automaton upon which the recognizer is based and also the kind of string being tested for acceptance. The end result of such an algorithm is to provide a flexible and powerful tool that takes advantage of the strengths of both table-driven and hardcoded techniques for the construction of optimal recognizers. This paper introduces the idea of dynamic Implementation of FAs for Performance (DIFAP). Unlike the traditional statically based approach that always yields to some latencies, DIFAP aims to provide a highly flexible framework enabling implementers to gain considerable time. Of course, factors such as hardware and operating system capabilities may constitute a bottleneck for efficient implementation of FAs. However, in this preliminary work, we only consider the kind of string as well as the automaton size in the design of DIFAP.

Using the gnu C++ compiler and Netwide Assembler (NASM) to encode table-driven and hardcoded algorithms respectively, we perform various experiments in order to capture the advantage of using either approach under specific considerations. All the experiments were conducted under the Linux operating system on an Intel Pentium 4 with 512 MB of RAM.

The structure of the remaining part of this paper is as follows: in Section 2, we report on the performance of randomly generated FAs over randomly generated strings. This illustrates that the threshold of efficiency of hardcoding over table-driven implementation is relatively independent of the alphabet size. Section 3 investigates some string patterns where hardcoding implementation always outperforms the table-driven algorithm. In Section 4, we introduce the design of DIFAP as the basic algorithm for efficient implementation of FAs under any circumstances. Section 5 provides a conclusion of the work and points to future directions to be taken in further investigating and implementing DIFAP.

2 Experiments based on random strings

Experiments have been conducted, based on the random generation of 100 different automata of size varying between 10 and 1000 states. For each such group of automata generated, the alphabet size under consideration was respectively, 10, 15, 20, 25, 30, 35, 40, 45, and 50 symbols. For each automaton of size n that was generated, a random accepting string of size $n-1$ was also generated. (We only relied on accepting strings instead of rejecting strings since the latter are most likely to require less time to be processed by the recognizer.) This randomly generated string was processed 50 times. The minimum, maximum and average time in clock cycles (ccs) to process the string over the 50 processing cycles were collected. In order to prevent side effects due to operating system and CPU overheads, the data relating to minimum processing times was considered to be the most accurate metric upon which further experimentation should be based. This minimum time metric was then divided by its number of symbols processed in order to estimate the time required to accept a single symbol. Such an approach was first suggested in [Kwk03a]. The collected data was then plotted to visualize the effect of the recognizer on randomly generated strings.

Figure 1 depicts the graphs for 10 and 50 symbols alphabet related to the table-driven experiments. The figure clearly shows that as the alphabet size grows, the processing time per symbol grows as well.

The hardcoded experimental results are depicted in figure 2. The generated hardcode (HC) relied on the approach suggested in [Kwk03a] using a jump table to hardcode each entry of the transition matrix. The hardcode is structured in blocks of

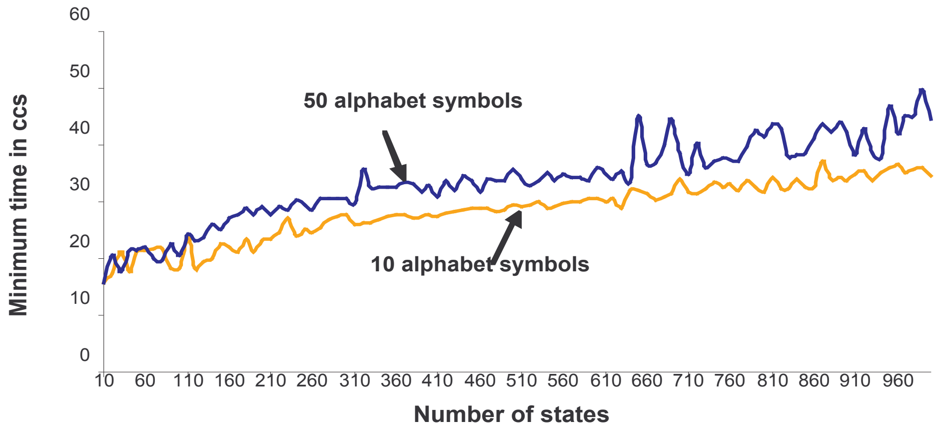


Figure 1: Performance based on table-driven experiments using random strings

assembly code, each block relating to a given state. These blocks are arranged contiguously in memory and in previous hardcode experiments, the order of the blocks corresponded to the order of rows in the transition table as shown in Source code 1. In those experiments, the strings tested were such that the flow of control through the assembly code caused jumps between contiguous blocks. Here, the same strings were used. However, to simulate the effect of a "random" string of a given length, the blocks were arranged in memory in a random order, so that jumps now take place to random places in memory[1]. The plotted graphs clearly show that for an automaton based on a 10 symbols alphabet of size 1000 states, the average time required to accept a symbol is about 70 ccs. This time is somewhat worse for an automaton of the same size based on 50 symbols alphabet (approximately 350 ccs). A plausible explanation for this is that the time is related to cache misses, due to the growth in the code size as the alphabet size grows, since the number of instructions required to encode the jump table becomes considerably larger. However, in the graphs, we can observe that in the region of 10 to 360 states, the average processing speed is always less than 50 ccs, irrespective of the number of alphabet symbols under consideration. It appears that in this range, the size of the code can still fit into the hardware's cache memory, and therefore results in efficient processing.

Source-code 1 *Extract hardcoded implementation of a recognizer using NASM*

```
asm_main:
...
mov edx, string
STATE_0:
   ;test if more symbol to process
```

[1] The notion of a random or average string is somewhat ill-defined. Nevertheless, the time taken to recognize such strings characterizes the hardcode behaviour for input that lies somewhere between best and worst case.

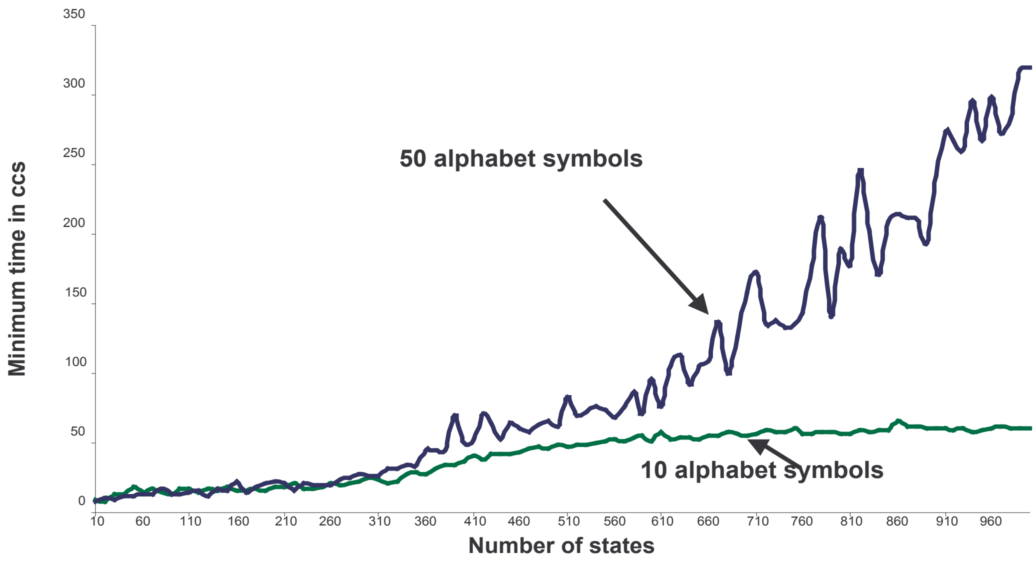


Figure 2: Performance based on hardcoded experiments using random strings

```
    jne   ACCEPT     ; no more symbol to process
    ;get the current symbol from the string
    mov      ebx, edx
    inc edx  ; points to the next symbol for further processing
    ; get the state where the symbol transits to
    shl     ebx, 2
    mov     esi, edx
    add     esi, ebx
    mov     ebx, [esi]
    shl     ebx,2
    mov     esi,ST_0    ; Label to access appropriate transition entry
    add     esi,ebx     ; related to the symbol being tested
    jmp     [esi]       ; jump to the appropriate entry of the transition table
    ST_0_TR0:           ; transition for the first symbol of the alphabet
     jmp STATE_1
    ST_0_TR1:           ; transition for the second symbol of the alphabet
     jmp STATE_1
    ST_0_TR_2:
     jmp reject         ; no transition (rejecting symbol)
    ...
STATE_1:
    ...
REJECT:
      ;do action reject
ACCEPT:
      ;do action accept
```

In order to more concisely observe the threshold of efficiency of one approach over another, we subtracted each table-driven data item from the corresponding data item of its hardcoded counterpart. This allowed us to capture the extent to which hardcode outperforms table-driven, or otherwise. Figure 3 shows the resulting plotted graphs. It clearly shows that for each automaton of a given alphabet size, the threshold of efficiency of the hardcoded version over the table-driven version is in the region of about 360 states. Such a result suggests that for any automaton based on any number of alphabet symbols in the range tested, the hardcode implementation is, on average, faster than the table-driven implementation if the number of states that make up the

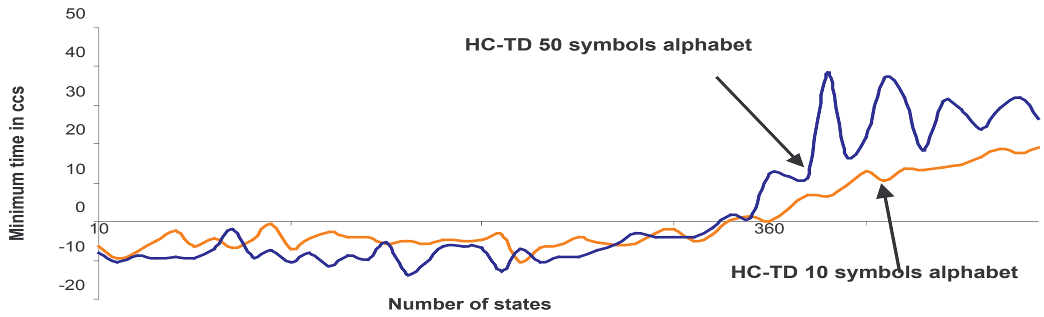


Figure 3: Performance comparison between hardcode and table-driven

automaton is less than 360. This is apparently due to the fact that, in this range of states, the hardcode is (mostly) in cache at run-time whereas table-driven processing is subject to random memory accesses. Above about 360 states, hardcode instructions from the main memory have to be accessed more frequently, and consequently there is a degradation in the average per state processing speed of the hardcode recognizer. The rate at which cache misses occur appears to be quite heavily dependent on the alphabet size. There is also a gradual degradation in the average per state processing speed of the table-driven recognizer. The behaviour in the table-driven is slightly more stable and seemingly less critically dependent on alphabet size. The table-driven code fits comfortably into cache, and so the issue of cache misses at the code level does not arise. The degradation in performance is therefore primarily due to the effect of data cache misses, the rate of which increases as the table size grows.

These observations allow us to project the worst-case scenarios for the two implementation strategies. These will occur when each and every state transition results in a cache miss. In the case of the table-driven recognizer, this will be a data cache miss, and in the case of the hardcode recognizer – a code cache miss. It is not obvious how experiments could be constructed to simulate this precise behaviour. However, it is not necessary to do so, for the above experiments already indicate the relative merits of the two approaches when a high rate of cache misses occur: the table-driven recognizer is then faster.

An interesting consequence of such an observation is the fact that, using the present threshold of efficiency, we can explore within that region various string patterns that may be subject to high hardcoded efficiency without having to restrict ourselves to the alphabet size and the size of the automaton being processed. The next section discusses such strings.

3 Experiments based on string patterns

In this section, we present two type of strings that may be subject to some efficient processing when hardcoded implementation is chosen as the basis for implementing

FAs. In [Kwk03a], we showed that for a single state of some randomly generated automaton, hardcoding using jump table outperformed the traditional table-driven implementation. In the subsection below, we extend the experiment to an accepting string that simply remains on a single state during the entire recognition process. The second case considers the conditions under which hardcode will continue to outperform the table driven version, even if the strings become relatively long. This is inferred from the results obtained from the next subsection and from the results relating to the threshold region (up to which hardcoded outperforms table-driven processing).

3.1 Strings that keep the FA in a single state

Let consider the automaton modelled in Figure 4 having 5 states with two accepting states 3 and 4. The strings *abab* and *cdef* of size 4 are both part of the language modelled by the automaton. In theory, the total time required to accept or reject each string should be roughly the same. However, we notice that for the string *abab*, once the device reads the first symbol *a*, it jumps to the final state 3 and remains there until the entire string is processed. On the other hand, the recognizer will transverse several different states in order to accept the string *cdef*. This observation indicates that in practice, the time required to accept strings of same size with different patterns may differ considerably from one another. The experiment randomly generated various

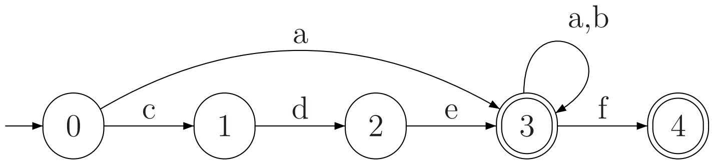


Figure 4: A state diagram that accepts the strings *abab* and *cdef*

automata of sizes between 10 and 1000 states using respectively 10, 15, 20, 25, 30, 35, 40, 45, and 50 symbols alphabet. Figure 5 depicts the difference in time between the table-driven and hardcoded implementation. It clearly shows that hardcode is consistently about 8 ccs faster than the table-driven implementation.

These results illustrate the fact that for strings following the pattern such as the one for *abab* above, the processor has sufficient space in its cache to hold the code relating to a single state. Since it always visits the same state over and over, no cache misses occur (neither for the hardcoded, nor the table-driven version) and there is a consequent high processing speed. It should be specifically noted that there are also no *data cache* misses in the the table-driven implementation. The scenario therefore represents the very best case behaviour for both the table driven and the hardcode implementations. It demonstrates that hardcode outperforms table-driven in this best case context.

The next experiment confirms that when long term behaviour tends towards best case behaviour, then the benefits of the hardcoded approach become increasingly apparent. This case is depicted in the next subsection.

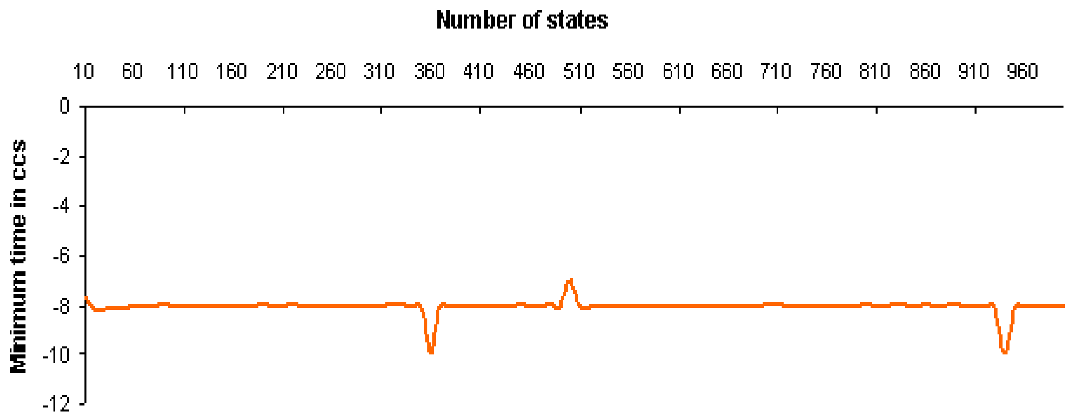


Figure 5: Comparative performance based on recurrent access on a single state

3.2 Strings that drive the FA randomly through a limited number of states

Consider the automaton modelled in figure 6. State t depicts some "sink state" in which the FA will remain after some other arbitrary set of states within the automaton have been visited. The state t is also assumed to be an accepting state. Based on the previous experiment, we would expect that the longer the FA remained in state t, the more the hardcoded implementation would enjoy an advantage over the table-driven version.

To verify this observation, and as a sort of sanity check on our results up to this point, hardcoded and table-driven implementations were set up to test strings of length $n-1$ in FAs with n states. The experiment was designed so that the behaviour in processing the first 300 states was random – in the same sense as previously described. However, thereafter the FA remains in the same state – i.e. the best case scenario prevails.

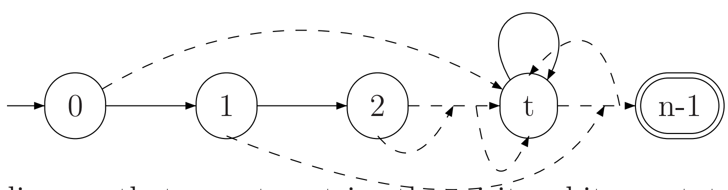


Figure 6: A state diagram that accepts a string that visits arbitrary states and remains on state t for some time

Figure 7 depicts the graphs obtained from the experiment. Unsurprisingly, it shows that hardcoding generally outperforms the table-driven implementation. However, there is also a suggestion in the data that in the longer term, the asymptotic improvement tends towards the 8ccs improvement observed in figure 5.

Of course, many more experiments similar to those described above could be run. An overall and general observation in regard to all these experiments is that they enable us to identify various ways in which the hardcoded implementation of

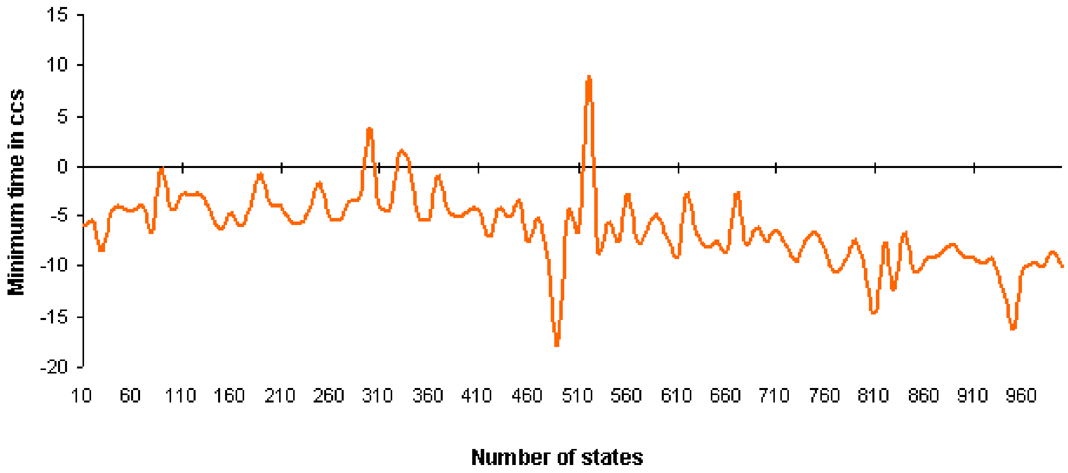


Figure 7: Comparison based on limited access on states

FAs may outperform the traditional table-driven implementation. The next section considers how such information could be used to capitalize on the advantages offered by both approaches.

4 A Preliminary Dynamic Algorithm

The experiments depicted in the previous sections clearly show that the efficiency of a string recognizer is highly dependent on the nature of the string being recognized. This suggests that if likely patterns of strings to be input are known in advance – at least in some probabilistic sense – then it may be possible to put in place a time optimizing mechanism to carry out the string recognition. Consequently, the idea of dynamically adapting the implementation strategy of the FA according to the expected input (or partially inspected) string may be considered. We use the acronym DIFAP to refer to this notion, designating Dynamic Implementation of FAs for Performance enhancement. Figure 8 depicts the overall design of a DIFAP system. When it is first invoked, the implementer provides the specification of the automaton to be used, regardless the type of string to be recognized. DIFAP then analyzes the specification and choose the appropriate way of implementing the automaton depending of the size of the derived automaton. In terms of the currently available data, if the size is less than 360 states, this means that hardcoding is likely to be the optimal approach in representing the automaton irrespective of the kind of string to be tested. On the other hand, if the size of the automaton is above 360 states, as suggested in Section 2, a hardcode implementation might be indicated if long term behaviour is likely to tend towards best case behaviour, a table-driven implementation will be the appropriate choice in the absence of such information. However, in the latter case, DIFAP relies on the kind of string received as input to adapt itself progressively to an implementation approach that is optimal in some sense (e.g. optimal in relation to the history of strings processed to date), resulting in improved average processing speed.

The figure indicates that for bigger automata size, a knowledge table (KT) is first checked. At this stage, we do not prescribe what information should be kept in the KT. We merely observe that the current input string could, in principle, undergo some preliminary scan to identify whether its overall structure conforms to some set of general patterns that favour hardcode over table-driven. If that is not the case, the table-driven version of the automaton specification is generated and is used by the recognizer to check whether the string is part of the language described by the FA or not. Otherwise, the hardcoded version of the FA's specification is generated and used for recognition.

Not indicated in the figure is the possibility of post-processing: after a string has been tested, the string and the test outcome could be used to update information in the KT. As a very simple example, we might decide to concretely implement the KT as a table of the FA's states, in which a count is kept of the number of times a state has been visited. This information could be used to rearrange the order of rows (which represent states) in the transition matrix used by the table-driven approach, in the hope of minimizing data cache misses when this implementation strategy is used. Alternatively, the same information could be used to dictate the blocks of hardcode that should preferentially be loaded into cache, in circumstances in which hardcoding is indicated. However, the foregoing should not be construed as the only way in which the KT can be implemented. We conjecture that there are many creative possibilities within this broad model that merit deeper investigation in the future.

One of the advantage of using such a dynamic algorithm is that the structure of the automaton does not always remains in the system after processing. Each automaton is always regenerated into its executable when the system is invoked. The only structure that permanently remains in the system is the algebraic specification of the automaton. This results therefore in some degree of minimization of memory load for automata of considerable size. However, there is no need to always regenerate automata of size less than 360 states since they will always be implemented in hardcode. That is the reason why in the figure no deletion of the generated hardcode is indicated when the "size less than 360" path is followed.

In an implementation of DIFAP, attention should be given to the following parts of the algorithm to minimize latencies:

- *Time to generate the recognizer:* Unless directly implemented by hand, any FA-related problem always requires a formal specification of the grammar that describes the automaton before its corresponding automaton is encoded. This is a general problem, and one specific to DIFAP. The DIFAP implementation could therefore use generator techniques similar to those used in efficient code generator tools such as YACC[2] which as been proven to be amongst the best tool available to create directly executable parsers. Unlike parsers, DIFAP's code generator will generate directly executable string recognizers.

- *Time taken to check the knowledge table:* One should take care to ensure that the matter of checking the KT does not degenerate into a time-inefficient exercise that negates any benefit from using the optimal string recognition strategy. Efficient algorithms should be devised that take minimal time to access the table and to chose the appropriate path to follow. This part of DIFAP may

[2]Yet Another Compilers Compiler

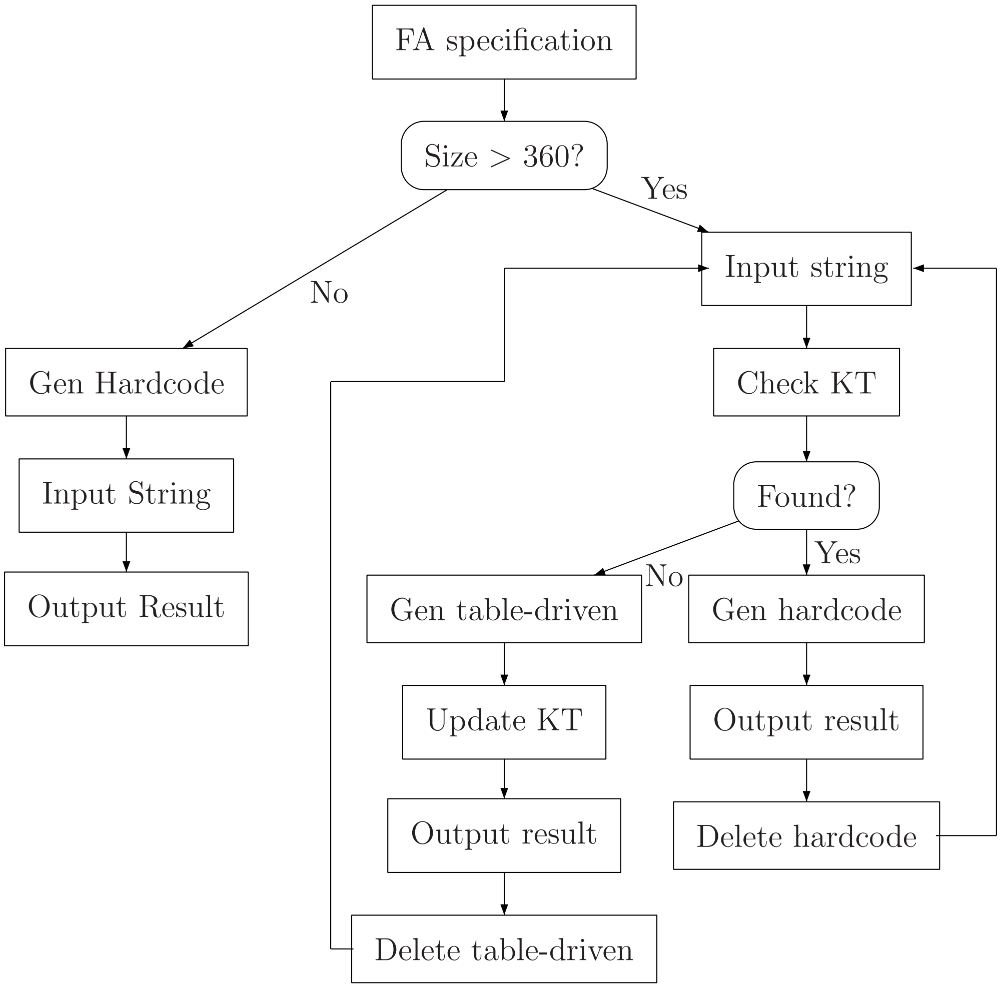


Figure 8: A Preliminary design of DIFAP

constitute a bottleneck. Intensive investigations will be made to provide an efficient approach to access the table and retrieve appropriate information.

- *Time required to update the knowledge table:* Although the precise nature and scope of the KT have not been identified here, it is envisaged that it will, itself, be a dynamic structure, changing over time in relation to the history of strings analyzed to date. However, there does not appear to be any reason for adapting the KT prior to processing the input string. Its update is something that can happen at a post-processing stage, and does not appear to be time-critical.

4.1 Applications of DIFAP

Finite automata are used to model several computational problems. Many of those problems are currently solved using the traditional table-driven approach. However, it might be of great advantage to use the DIFAP model in order to overcome some speed latencies. The following is a high level survey of various kinds of problem that could benefit from the DIFAP approach:

- *Compiler construction:* Lexical analysis is the part of the compiler that deals with FAs. A lexical analyzer generator such as LEX usually generates table-driven code and uses it to scan the current identifier being converted into token. However, the automaton generated is often of relatively small size. This means that, instead of always using table-driven implementation, the compiler implementor could opt for the hardcoding implementation that might yield faster processing.

- *String Pattern Matching:* Even though a lot of work as been done to solve problems of exact and approximate string keyword pattern matching using efficient algorithms (e.g. [Wat95]) their practical implementation is sometimes very inefficient according to [Nr02]. In some circumstances, it might be beneficial to solve the problem of online string matching using hardcoding. Since many problems deal with a large amount of text, constructing a single automaton based on the text might be highly inefficient. Nonetheless, the text can be decomposed into small block on which we can derive its corresponding hardcode in order to achieve better processing.

- *Cellular automata* Implementing cellular automata to solve computational genetic problems requires, in general, small blocks of automata. Therefore, hardcoding might be of value in improving the processing speed of some of these problem where speed is a major factor.

- *Network Intrusion Detection:* In general, the size of the automaton generated depends on the variety of traffic to which the network is subject. Since Network Intrusion Detection algorithms can also construct FAs on the fly, DIFAP appears to be a suitable approach to adopt in such problems: the generated automaton will then be the one deemed to be most efficient in the given context.

- *Other FA application:* There are many other computational problems that rely on FA processing, where DIFAP could be of potential value. Aspects of natural language processing come time mind. However, it is not our intention to exhaustively enumerate all potential domains of application.

The foregoing indicates that there are indeed a significant number of problem domains that make the further elaboration of the DIFAP model a worthwhile endeavour. It is known that in many of these domains, efficient solutions have already been established. However, since DIFAP is a dynamic framework, we aim at improving the existing results using the technique. A investigation into each of the domain is therefore of concern but is out of the scope of this introductory work.

5 Conclusion and Future Work

In this paper, we have shown that the processing speed of a string recognizer not only depends on the length of the string being recognized but also on the kind of the string under consideration. Alternative ways of implementing finite automata without relying on the traditional table-driven approach has been revisited after some intensive work done by Ketcha et al in [Kwk03a, Kwk03b, Ket03]. Ketcha's work showed that hardcoding of FAs might be a better ways of implementing FAs up to some threshold – approximately 360 states, in the context of the hardware and alphabet size used in those experiments. The various investigations performed has suggested the notion of Dynamic Implementation of Finite Automata for Performance. The DIFAP concept, provides automata implementers with an algorithm that is flexible enough to take advantage of both hardware and software considerations of the environment in which the string recognizer is being processed. The actual design of DIFAP relies on the constraints related to the automaton's structure as well as the kinds of string being processed. In the near future an implementation of DIFAP is envisaged, and various experiments will be conducted in order to characterize its efficiency. Other DIFAP issues not fully elaborated above will be to dynamically decide on matters such as the use of optimal data structures for particular kinds of FAs (e.g. linked list would appear to be a better way of representative very sparse transition functions, rather than a conventional two-dimensional matrix). Other DIFAP considerations include dynamic decisions about when to use stretched or jammed automata [Nwk03], and whether hybrid implementations that combine aspects of hardcode into a table driven implementation (or vice-versa) would be beneficial. However, these are all currently regarded as matters for future research. The final goal of DIFAP will be to design domain specific algorithms so that each problem can be solved on a highly efficient way.

References

[Kim02] Paul Kimmel. The Visual Basic .Net Developper's Book. Addison-Wesley, 2003.

[Ket03] E. Ketcha Ngassam. Hardcoding Finite Automata. MSC Dissertation. University of Pretoria, 2003.

[Kmp77] D. E Knuth and J.H Morris, Jr and V. R. Pratt. Fast Pattern Matching in Strings. SIAM J. Comput. Volume 6, 323-350, 1977.

[Kwk03a] E. Ketcha Ngassam, Bruce. W. Watson, and Derrick. G. Kourie, Preliminary Experiments in Hardcoding Finite Automata, Poster paper, CIAA, Santa Barbara, 299-300, September 2003.

[Kwk03b] E. Ketcha Ngassam, Bruce. W. Watson, and Derrick. G. Kourie, Hardcoding Finite State Automata Processing, SAICSIT, Johannesburg, 111-121, September 2003.

[Nwk03] Noud De Beijer, Bruce W. Watson and Derrick G. Kourie, Stretching and Jamming of Automata, SAICIST, Johannesburg, 198-207, September 2003.

[Nr02] Gonzalo Navarro, and Mathieu Raffinot. Flexible Pattern Matching In String: Practical on-line search for texts and biological sequences. Cambridge University Press 2002.

[Wat95] Bruce W. Watson. Taxonomies and Toolkits of Regular Languages Algorithms. PhD Thesis. Technical University of Eindhoven, 1995.

Arithmetic Coding in Parallel

Jan Šupol and Bořivoj Melichar

Department of Computer Science & Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13, 121 35 Prague 2

e-mail: {supolj,melichar}@fel.cvut.cz

Abstract. We present a cost optimal parallel algorithm for the computation of arithmetic coding. We solve the problem in $\mathcal{O}(log\ n)$ time using $n/log\ n$ processors on EREW PRAM. This leads to $\mathcal{O}(n)$ total cost.

Keywords: arithmetic coding, NC algorithm, EREW PRAM, PPS, parallel text compression.

1 Introduction

There is still a need for data coding. The growing demand for network communication and for storage of data signals from space are not the only examples of coding needs. Many algorithms have been developed for text compression.

One of these is arithmetic coding [Mo98, Wi87], which is more efficient than the widely known Huffman algorithm [Hu52]. The latter rarely produces the best variable-size code, the arithmetic coding overcomes this problem. Arithmetic coding can be generated in $\mathcal{O}(n)$ time sequentially, and we present a well scalable NC parallel algorithm that generates the code in $\mathcal{O}(log\ n)$ time on EREW PRAM with $n/log\ n$ processors. This leads to $\mathcal{O}(n)$ total cost and a cost optimal algorithm.

Despite the large number of papers on the parallel Huffman algorithm (the last known [Lb99] is work optimal) there are only a few papers on parallel arithmetic coding. Most of these are based on a quasi-arithmetic coding [Ho92]. We know only two exceptions. The first [Yo98] is based on an N-processor hypercube and is not cost optimal. The second [Ji94] is mainly focused on the hardware implementation. Authors expected the processing speed of their tree-based parallel structure eight times as high as the speed of a sequential coder. This is still $\mathcal{O}(n)$ parallel time.

This paper is organized as follows. Section 2 provides a description of the sequential arithmetic coding algorithm. Section 3 presents some basic definitions. Section 4 describes the parallel prefix computation needed by our algorithm. Section 5 presents our parallel arithmetic coding algorithm. Section 6 describes the time complexity of our algorithm. Section 7 contains our conclusion. Note that this paper does not mention the decoding process.

2 Sequential Arithmetic Coding

First we review the sequential algorithm. Let $A = [a_0, a_1, \ldots, a_{m-1}]$ be the source alphabet containing m symbols and an associated set of frequencies $F = [f_0, f_1, \ldots, f_{m-1}]$ shows the occurrences of each symbol. Next we compute the array of probabilities $R = [r_0, r_1, \ldots, r_{m-1}]$ such that $r_i = f_i/T$ where $T = \sum_{i=0}^{m-1} f_i$, the array of high ranges $H = [h_0, h_1, \ldots, h_{m-1}]$ such that $h_i = \sum_{x=0}^{i} r_x$, the array of low ranges $L = [l_0, l_1, \ldots, l_{m-1}]$ such that $l_0 = 0$ and $l_i = h_{i-1}, i > 0$. Table 1 shows an example.

A	F	R	L	H
S	5	5/10=0.5	0.5	1.0
W	1	1/10=0.1	0.4	0.5
I	2	2/10=0.2	0.2	0.4
M	1	1/10=0.1	0.1	0.2
␣	1	1/10=0.1	0.0	0.1

Table 1: Frequencies, probabilities and ranges of five symbols.

The string of symbols $S = [s^0, s^1, \ldots, s^{n-1}]$ is encoded as follows. The first character s^0 can be encoded by a number within an interval $[l_y, h_y)$ associated to a character $y = s^0, y \in A$. This notation $[a, b)$ means the range of real numbers from a to b, not including b. Let us define these two bounds as *LowRange* and *HighRange*.

As more symbols are input and processed, *LowRange* and *HighRange* are updated according to

$$\mathit{LowRange}^j = \mathit{LowRange}^{j-1} + (\mathit{HighRange}^{j-1} - \mathit{LowRange}^{j-1}) \times l_x,$$

$$\mathit{HighRange}^j = \mathit{LowRange}^{j-1} + (\mathit{HighRange}^{j-1} - \mathit{LowRange}^{j-1}) \times h_x,$$

where h_x and l_x are low and high ranges of new character $x \in A$, $\mathit{LowRange}^{-1} = 0$, $\mathit{HighRange}^{-1} = 1$. Table 2 indicates an example for the word "SWISS".

A	L&H	The calculation of low and high ranges
S	L	$0.0 + (1.0 - 0.0)\times 0.5 = 0.5$
	H	$0.0 + (1.0 - 0.0)\times 1.0 = 1.0$
W	L	$0.5 + (1.0 - 0.5)\times 0.4 = 0.70$
	H	$0.5 + (1.0 - 0.5)\times 0.5 = 0.75$
I	L	$0.7 + (0.75 - 0.7)\times 0.2 = 0.71$
	H	$0.7 + (0.75 - 0.7)\times 0.4 = 0.72$
S	L	$0.71 + (0.72 - 0.71)\times 0.5 = 0.715$
	H	$0.71 + (0.72 - 0.71)\times 1.0 = 0.720$
S	L	$0.715 + (0.72 - 0.715)\times 0.5 = 0.7175$
	H	$0.715 + (0.72 - 0.715)\times 1.0 = 0.7200$

Table 2: The process of arithmetic encoding.

3 Definitions

Our parallel algorithm is designed to run on the Parallel Random Access Machine (PRAM), which is a very simple synchronous model of the SIMD computer([Le92, Qu94, Tv94]). PRAM includes many submodels of parallel machines that differ from each other by conditions of access to the shared memory. Our algorithm works on the Exclusive Read Exclusive Write (EREW) PRAM model, which means that no two processors can access the same cell of the shared memory.

We define sequential time $SU(n)$ as the worst time of the best known sequential algorithm where n is the size of the input data. Parallel time $T(n,p)$ is the time elapsed from the beginning of a p-processor parallel algorithm solving a problem instance of size n until the last (slowest) processor finishes the execution.

Consider a synchronous p-processor algorithm A with $\tau = T(n,p)$ parallel steps. Let p_i be the number of processors active (working) at step $i \in \{1, 2, \ldots, \tau\}$ of A. Then the synchronous parallel work of A is

$$W(n,p) = T_1 + T_2 + \cdots + T_\tau.$$

Parallel cost (also called processor-time product) is defined as

$$C(n,p) = p \times T(n,p).$$

It is obvious that

$$SU(n) \leq W(n,p) \leq C(n,p).$$

If $SU(n) = W(n,p)$ then the algorithm is work optimal. If $SU(n) = C(n,p)$ then the algorithm is cost optimal.

The efficiency of the parallel algorithm is defined as

$$E(n,p) = \frac{SU(n)}{C(n,p)}.$$

Let E_0 be the constant such that $0 < E_0 < 1$. Then isoefficiency function $\psi_1(p)$ is the asymptotically minimum function such that

$$\forall n_p = \Omega(\psi_1(p)) : E(n_p, p) \geq E_0.$$

Hence, $\psi_1(p)$ gives asymptotically the lower bound on the instance size of a problem that can be solved by p processors with efficiency at least E_0.

Scalability is the ability to adapt itself to a changing number of processors or or to changing size of the input data. Good scalability means that if we want to use new processors we have to increase the size of our problem only a little. Fast growth of function ψ_1 provides poor scalability.

We say that class NC (Nick's class) is a set of algorithms that can be computed with at most polylogarithmic time and with at most a polynomial number of processors. These algorithms provide a high level of parallelization.

4 Parallel Prefix Computation

As far as our parallel algorithm is based on the parallel prefix algorithm we show how it works. The problem is defined as follows [La80]. Let $S = [s^0, s^1, \ldots, s^{n-1}]$ be the array of numbers. The prefix problem is to compute all the prefixes of the products

$$s^0 \otimes s^1 \otimes \cdots \otimes s^{n-1},$$

where $\otimes$ is an associative operation.

Fig. 1 shows the algorithm that assumes n processors $p^0, p^1, \ldots, p^{n-1}$ and array $M = [m^0, m^1, \ldots, m^{n-1}]$ of numbers stored in the shared memory. Every processor p^i also has a register y^i. From now on we will use EREW PRAM with similar conditions.

```
for i := 0, 1, ..., n − 1 do_in_parallel
   y^i := M[i];
for j := 0, 1, ..., ⌈log n⌉ − 1 do_sequentially
begin
   for i := 2^j, 2^j + 1, ..., n − 1 do_in_parallel
      y^i := y^i ⊗ M[i − 2^j];
   for i := 2^j, 2^j + 1, ..., n − 1 do_in_parallel
      M[i] = y^i;
end
```

Figure 1: Parallel prefix algorithm.

Fig. 2 indicates a parallel prefix algorithm computing an array of 7 numbers with the associative operation of sum. This is then called the parallel prefix sum.

Here we show the parallel time $T(n,p)$ of the parallel prefix computation on EREW PRAM. First we suppose that $p < n$. Each processor simulates n/p processors. This sequentially sums n/p numbers. This takes at most $4n/p$ steps (read first number, read second number, sum and write the result). After that the processors run the parallel prefix algorithm in time $\mathcal{O}(log\ p)$. So the parallel time, cost, efficiency and function ψ_1 take

$$\begin{aligned} T(n,p) &= \mathcal{O}(n/p + log\ p), \\ C(n,p) &= \mathcal{O}(n + p\ log\ p), \\ E(n,p) &= \mathcal{O}(\frac{n}{n + p\ log\ p}), \\ \psi_1(p) &= \mathcal{O}(p\ log\ p). \end{aligned}$$

We can say that the parallel prefix algorithm is a well scalable NC algorithm due to the definitions in Section 3. If $p = n$ then

$$T(n,n) = \mathcal{O}(n/n + log\ n) = \mathcal{O}(log\ n),$$

$$C(n,n) = \mathcal{O}(n + n\ log\ n) = \mathcal{O}(n\ log\ n).$$

However, when $p = n/log\ n$ then

$$T(n, n/log\ n) = \mathcal{O}(n\ log\ n/n + log\ n - log\ log\ n) = \mathcal{O}(log\ n),$$

$$C(n, n/log\ n) = \mathcal{O}(n + n/log\ n(log\ n - log\ log\ n)) = \mathcal{O}(n).$$

Hence, we have obtained a parallel cost optimal algorithm.

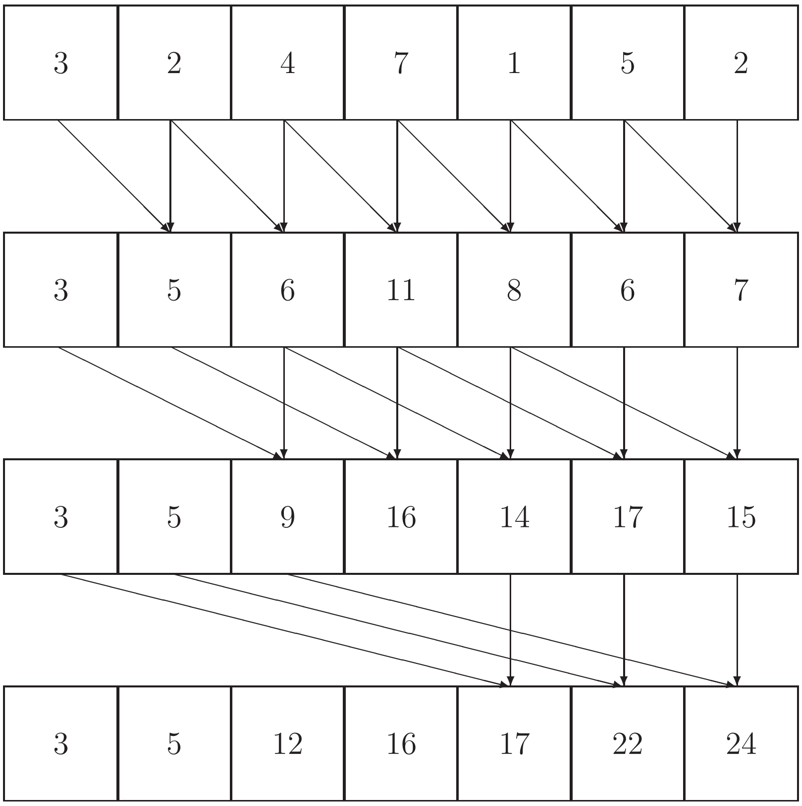


Figure 2: Parallel prefix sum example.

5 Parallel Arithmetic Coding

Recall that we use the array $A = [a_0, a_1, \ldots, a_{m-1}]$ of the source alphabet containing m symbols, the associated set of frequencies $F = [f_0, f_1, \ldots, f_{m-1}]$, the associated set of probabilities $R = [r_0, r_1, \ldots, r_{m-1}]$ so that $r_i = f_i/T$ where $T = \sum_{i=0}^{m-1} f_i$, the array of low ranges $L = [l_0, l_1, \ldots, l_{m-1}]$, the array of high ranges $H = [h_0, h_1, \ldots, h_{m-1}]$ so that $l_0 = 0$, $l_i = h_{i-1}, i > 0$ and $h_i = l_i + r_i$.

Our idea of parallelism is that we have a string $S = [s^0, s^1, \ldots, s^{n-1}]$ of n characters to encode. Each processor p^j is associated with a character s^j and computes variables *LowRange* and *HighRange* for that character.

5.1 Preliminaries

We suppose that we have an array $Range = [range^0, range^1, \ldots, range^{n-1}]$ for our algorithm. Each $range^j$ is initialized with probability r_y such that $a_y = s^j$ where j is the index of the j-th character in the input string and $s^j \in A$. We also suppose that we have an array $Low = [low^0, low^1, \ldots, low^{n-1}]$. Each low^j is initialized with value l_y such that $a_y = s^j$. We need at least one variable *high* initialized with value h_y such that $a_y = s^{n-1}$.

5.2 Changes in Sequential Algorithm

Let us return to sequential arithmetic coding and try to change the algorithm a bit so that it can be parallelized. Recall the bounds computation

$$LowRange^j = LowRange^{j-1} + (HighRange^{j-1} - LowRange^{j-1}) \times l_x,$$

$$HighRange^j = LowRange^{j-1} + (HighRange^{j-1} - LowRange^{j-1}) \times h_x,$$

where h_x and l_x are low and high ranges of new character $x \in A$, $LowRange^{-1} = 0$, $HighRange^{-1} = 1$ and mark the cumulative lower and higher bounds

$$LR^j = (HighRange^{j-1} - LowRange^{j-1}) \times l_x,$$

$$HR^j = (HighRange^{j-1} - LowRange^{j-1}) \times h_x.$$

So the values *LowRange* and *HighRange* are updated as

$$LowRange^j = LowRange^{j-1} + LR^j,$$

$$HighRange^j = LowRange^{j-1} + HR^j$$

and we now focus only on the variables *LR* and *HR* now.

$$\begin{aligned} LR^j &= (HighRange^{j-1} - LowRange^{j-1}) \times l_x = \\ &= (LowRange^{j-2} + HR^{j-1} - LowRange^{j-2} - LR^{j-1}) \times l_x = \\ &= (HR^{j-1} - LR^{j-1}) \times l_x, \end{aligned}$$

$$\begin{aligned} HR^j &= (HighRange^{j-1} - LowRange^{j-1}) \times h_x = \\ &= (LowRange^{j-2} + HR^{j-1} - LowRange^{j-2} - LR^{j-1}) \times h_x = \\ &= (HR^{j-1} - LR^{j-1}) \times h_x. \end{aligned}$$

Moreover, $LowRange^j$ can be computed as

$$\begin{aligned} LowRange^j &= LR^j + LowRange^{j-1} = LR^j + LR^{j-1} + LowRange^{j-2} = \\ &= \cdots = LR^j + LR^{j-1} + \cdots + LR^0 + LowRange^{-1} = \\ &= \sum_{x=0}^{j} LR^x + LowRange^{-1} = \sum_{x=0}^{j} LR^x \end{aligned}$$

because $LowRange^{-1} = 0$.

The change in our algorithm is that we first compute the cumulative lower and higher bounds and next we simply compute the sum of these cumulative bounds so that we obtain the final bounds *LowRange* and *HighRange*.

Let us see how the variables *LR* and *HR* can be computed for the word “SWISS”. We declare that LR^0 is the *LR* variable for the first character s^0 = “S'' and l_x, h_x, r_x are lower range, higher range and probability of character $x \in A$. LR^{-1} and HR^{-1} are initial cumulative bounds for a number that represents the encoded text S. For arithmetic coding this number is defined by default as an interval [0,1). That is why $LR^{-1} = LowRange^{-1} = 0$ and $HR^{-1} = HighRange^{-1} = 1$.

$LR^{-1} = 0$
$HR^{-1} = 1$
$LR^0 = (HR^{-1} - LR^{-1}) \times l_s = 1.0 \times 0.5 = 0.5$
$HR^0 = (HR^{-1} - LR^{-1}) \times h_s = 1.0 \times 1.0 = 1.0$
$LR^1 = (HR^0 - LR^0) \times l_w = (h_s - l_s) \times l_w = r_s \times l_w = 0.5 \times 0.4 = 0.2$
$HR^1 = (HR^0 - LR^0) \times h_w = (h_s - l_s) \times h_w = r_s \times h_w = 0.5 \times 0.5 = 0.25$
$LR^2 = (HR^1 - LR^1) \times l_i = (r_s \times h_w - r_s \times l_w) \times l_i = r_s \times r_w \times l_i = 0.5 \times 0.1 \times 0.2 = 0.01$
$HR^2 = (HR^1 - LR^1) \times h_i = (r_s \times h_w - r_s \times l_w) \times h_i = r_s \times r_w \times h_i = 0.5 \times 0.1 \times 0.4 = 0.02$
$LR^3 = (HR^2 - LR^2) \times l_s = (r_s \times r_w \times h_i - r_s \times r_w \times l_i) \times l_s = r_s \times r_w \times r_i \times l_s = 0.005$
$HR^3 = (HR^2 - LR^2) \times h_s = (r_s \times r_w \times h_i - r_s \times r_w \times l_i) \times h_s = r_s \times r_w \times r_i \times h_s = 0.01$
...

So it is obvious that the lower bound of the j-th character LR^j and the higher bound of the j-th character HR^j can be computed as

$$LR^j = \left(\prod_{x=0}^{j-1} r^x\right) \times l^j, j > 0,$$

$$HR^j = \left(\prod_{x=0}^{j-1} r^x\right) \times h^j, j > 0.$$

5.3 Parallel Prefix Production

for $i := 0, 1, \ldots, n-1$ **do_in_parallel**
  $y^i := Range[i]$;
for $j := 0, 1, \ldots, \lceil log\ n \rceil - 1$ **do_sequentially**
begin
  for $i := 2^j, 2^j + 1, \ldots, n-1$ **do_in_parallel**
    $y^i := y^i \times Range[i - 2^j]$;
  for $i := 2^j, 2^j + 1, \ldots, n-1$ **do_in_parallel**
    $Range[i] = y^i$;
end

Figure 3: Parallel prefix production algorithm.

These new LR and HR variables are exactly what we need, because $\prod_{x=0}^{j-1} r^x$ can be computed in parallel as we immediately show. Computation of $\prod_{x=0}^{j} r^x = \prod_{x=0}^{j} range^x$ can be done by the parallel prefix production algorithm explained in Section 4, as shown in Fig. 3. Table 3 indicates the parallel prefix algorithm in our example for the word "SWISS".

S	W	I	S	S
0.5	0.1	0.2	0.5	0.5
0.5	0.05	0.02	0.1	0.25
0.5	0.05	0.01	0.005	0.005
0.5	0.05	0.01	0.005	0.0025

Table 3: Parallel prefix production example for the word "SWISS".

5.4 Cumulative Bounds Computation

If we have computed $\prod_{x=0}^{j-1} r^x$ we can obtain the variables LR^j and HR^j simply as the product of $\prod_{x=0}^{j-1} r^x \times l^j$ and $\prod_{x=0}^{j-1} r^x \times h^j$. Parallel algorithm computing the variables LR and the variable HR^{n-1} is shown in Fig. 4. The variables HR are not exactly needed, except for the last one HR^{n-1}. If these variables are required, they can be computed in a similar way. The value HR^{n-1}, which is the cumulative high range, is computed after the parallel prefix production computation as

$$HR^{n-1} = (\prod_{x=0}^{n-2} r^x) \times h^{n-1}.$$

Table 4 shows this computation in our example for the word "SWISS". Note that the results correspond to the cumulative bounds in our sequential example.

do_sequentially
begin
  $y^{n-1} := High;$
  $y^{n-1} := y^{n-1} \times Range[n-2];$
  $High := y^{n-1};$
  $y^0 := 1;$
end
for $i := 1, 2, \ldots, n-1$ **do_in_parallel**
  $y^i := Range[i-1];$
for $i := 0, 1, \ldots, n-1$ **do_in_parallel**
begin
  $y^i := y^i \times Low[i];$
  $Low[i] := y^i;$
end

Figure 4: Parallel computation of the variables LR and HR^{n-1}.

Now we have computed the cumulative high and low ranges. The array *Low* contains the LR values and the field *High* contains the value HR^{n-1}. Next we have to compute the sum of these cumulative ranges LR so that we shall obtain the required bounds *HighRange* and *LowRange* for arithmetic compression of string S.

L/H	S	W	I	S	S
LR	0.5	0.2	0.01	0.005	0.0025
HR	1	0.25	0.02	0.01	0.005

Table 4: Low and high ranges.

5.5 Computation of Low and High Ranges

In Section 5.4 we computed the cumulative bounds *LR* and *HR*. Here we show how to obtain the bounds earlier declared as *LowRange* and *HighRange* for the compressed text. These values can be computed as shown in Section 5.2 as

$$LowRange^j = (\sum_{x=0}^{j-1} LR^x) + LR^j,$$

$$HighRange^j = (\sum_{x=0}^{j-1} LR^x) + HR^j.$$

To compute the sum we can use the parallel prefix algorithm once more, exactly the parallel prefix sum shown in the former text. Finally, after computing the sum, the variable $HighRange^{n-1}$ is obtained as

$$HighRange^{n-1} = LowRange^{n-2} + HR^{n-1}.$$

This algorithm is shown in Fig. 5. The array *Low* contains the values *LowRange* and the field *High* contains the value $HighRange^{n-1}$. Our example for the word "SWISS" is shown in Table 5.

```
for i := 0, 1, ..., n − 1 do_in_parallel
  y^i := Low[i];
for j := 0, 1, ..., ⌈log n⌉ − 1 do_sequentially
begin
  for i := 2^j, 2^j + 1, ..., n − 1 do_in_parallel
    y^i := y^i + Low[i − 2^j];
  for i := 2^j, 2^j + 1, ..., n − 1 do_in_parallel
    Low[i] = y^i;
end
do_sequentially
begin
  y^(n−1) := High;
  y^(n−1) := y^(n−1) + Low[n − 2];
  High := y^(n−1);
end
```

Figure 5: *LowRange* and $HighRange^{n-1}$ computation algorithm.

S	W	I	S	S
0.5	0.2	0.01	0.005	0.0025
0.5	0.7	0.21	0.015	0.0075
0.5	0.7	0.71	0.715	0.2175
0.5	0.7	0.71	0.715	0.7175

Table 5: Parallel prefix sum example.

6 Time and Cost Complexities

Our algorithm does not say how to set the arrays *Range*, *Low* and the variable *High* in a preliminary phase. However, having set the arrays A, R, L and H, this can be done in time $\mathcal{O}(1)$ on CREW PRAM with a good hash function that returns an index in the array A of an input character from the input string S.

Our EREW PRAM algorithm consists of three phases. In the first phase, the parallel prefix production is computed. As shown in Section 4, this can be done in time $\mathcal{O}(n/p + log\ p)$ where p is the number of used processors and n is the size of the input. In the second phase, shown in Fig. 4, we have computed the cumulative bounds LR and HR in time $\mathcal{O}(n/p)$. The third phase, the parallel prefix sum shown in Fig. 5, also takes $\mathcal{O}(n/p + log\ p)$ time. The computation of $HighRange^{n-1}$ takes only $\mathcal{O}(1)$ time in any phase. So the time and cost of our algorithm are

$$T(n,p) = \mathcal{O}(n/p + log\ p),$$

$$C(n,p) = \mathcal{O}(n + p\ \ log\ p).$$

If $p = n/log\ n$ then the total time is $\mathcal{O}(log\ n)$ and the cost is $\mathcal{O}(n)$.

Because our algorithm consists mainly of parallel prefix computation, it inherits its best properties. Our algorithm is therefore a well scalable *NC* algorithm, and it can be implemented as the cost optimal algorithm.

7 Conclusions

We have presented a parallel *NC* algorithm for computation of arithmetic coding. We have solved the problem in $\mathcal{O}(log\, n)$ time using $n/log\, n$ processors on EREW PRAM. Our algorithm leads to $\mathcal{O}(n)$ total cost and is cost optimal.

The preliminary phase is a weakness of our algorithm. However, if we were able to construct a good adaptive parallel arithmetic coding based on our algorithm, it could solve this problem.

Another question is how to make a good parallel arithmetic decoding algorithm.

References

[Ho92] Howard, Paul G., Jeffrey Scott Witter (1992): Parallel Losseless Image Compression Using Huffman and Arithmetic Coding. Proceedings of the IEEE Data Compression Conference, 299-308.

[Hu52] Huffman, David (1952): A method for the construction of minimum-redundancy codes. Proceedings of the Inst. Radio Engineers, **40**: 1098-1101.

[Ji94] Jiang J., S. Jones (1994): Parallel design of arithmetic coding. IEE Proceedings-Computers and Digital Techniques, **141**(6):327-333, November.

[La80] Ladner, Richard and Michael J. Fisher (1980): Parallel Prefix Computation. Journal of the ACM, **27**(4):831-838, October.

[Lb99] Laber, Eduardo Sany, Ruy Luiz Milidi and Artur Alves Pessoa (1999): A Work Efficient Parallel Algorithm for Constructing Huffman Codes. Proceedings of the IEEE Data Compression Conference DCC'99.

[Le92] Lewis, T.G. and H. El-Rewini (1992): Introduction to Parallel Computing. Prentice Hall.

[Qu94] Quinn, M.J.(1994): Parallel Computing Theory and Practise. McGraw-Hill.

[Mo98] Moffat, Alistar, Redford Neal, and Ian H.Witten (1998): Arithmetic Coding Revisited. ACM Transactions on Information Systems, **16**(3):256-294, July.

[Tv94] Casavant, T.L., P. Tvrdík and F. Plášil, editors (1994): Parallel Computers: Architectures, Languages, and Algorithms. IEEE CS Press.

[Wi87] Witten, Ian H., Redford Neal and John G. Cleary (1987): Arithmetic coding for Data Compression. Communications of the ACM **30**(6):520-540.

[Yo98] Youssef A. (1998): Parallel Algorithms for Entropy Coding Techniques. Proceedings of European Parallel and Distributed Systems. ACTA Press.