Research Report

Proceedings of the Prague Stringology Conference '03

Edited by Milan Šimánek

September 2003

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), Bruce W. Watson,
Geraint Wiggins

Organizing Committee

Miroslav Balík, Jan Holub, Bořivoj Melichar, Milan Šimánek

URL

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

Proceedings of the Prague Stringology Conference '03
Published by Vydavatelství ČVUT, Zikova 4, 16635 Praha 6, Czech Republic
Edited by Milan Šimánek
Contact: Prague Stringology Club
Katedra počítaču, 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 80-01-02823-2

Table of Contents

Preface

The Prague Stringology Conference 2003 (PSC'03) was held at the Department of Computer Science and Engineering of the Czech Technical University in Prague, Czech Republic, on September 22–24, 2003. 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 eleven were selected for presentation at the conference, based on originality and quality. This volume contains these selected papers.

In the years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conferences in 2001 and 2002 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 a special issue of the journal Kybernetika and those selected from PSC'02 were published in a special issue of the Nordic Journal of Computing.

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'03 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'03. Last, but not least, my thanks go to the members of the organizing committee for ensuring such a smooth running of the conference.

In Hamilton, Ontario, Canada
on August 2003
Jan Holub

The Transformation Distance Problem Revisited

Behshad Behzadi and Jean-Marc Steyaert

LIX, École Polytechnique
Palaiseau cedex 91128, France

e-mail: {behzadi,steyaert}@lix.polytechnique.fr

Abstract. Evolution acts in several ways on biological sequences: either by mutating an element, or by inserting, deleting or copying a segment of the sequence. Varré et al. [VDR98] defined a transformation distance for the sequences, in which the evolutionary operations are copy, reverse copy and insertion of a segment. They also proposed an algorithm to calculate the transformation distance. This algorithm is $O(n^4)$ in time and $O(n^4)$ in space, where n is the size of the sequences. In this paper, we propose an improved algorithm which costs $O(n^2)$ in time and $O(n^2)$ in space. Furthermore, we extend the operation set by adding point deletions. We present an algorithm which is $O(n^3)$ in time and $O(n^2)$ in space for this extended case.

Keywords: dynamic programming, pattern matching

1 Introduction

Building models and tools to quantify evolution is an important domain of biology. Evolutionary trees or diagrams are based on statistical methods which exploit comparison methods between genomic sequences. Many comparison models have been proposed according to the type of physico-chemical phenomena that underly the evolutionary process [Do81]. Different evolutionary operation sets are studied. Mutation, deletion and insertion were the first operations dealt with [SaKr83]. Duplication and contraction were then added to the operation set [BeRi02, BeSt03]. All these operations were acting on single letters, representing bases, aminoacids or more complex sequences: they are called point transformations. Segment operations are also very important to study. In a number of papers [VDR97, VDR98, VDR99], Varré et al. have studied an evolutionary distance based on the amount of segment moves that Nature needed (or is supposed to have needed) to transfer a sequence from one species to the equivalent sequence in another one. Their model is concerned with segments copy with or without reversal and on segment insertion: it is thus a very simple and robust model which can easily be explained from biological mechanisms. They developed this study on DNA sequences, but the basic concepts and algorithms apply as well to proteins or satellites.

The algorithm they propose to compute the minimal transformation sequence is based on an encoding into a graph formalism, from which one can get the solution by computing shortest paths. This gives an $O(n^4)$ answer both in space and time[1].

[1] Even $O(n^6)$ in the last french version [Va00].

In fact it is possible to give a direct solution based on dynamic programming which costs only $O(n^2)$ in time and space. This solution is obviously more efficient for long sequences and makes the problem tractable even for very long sequences.

In the second section we describe the model and the problem description.

In the third section our algorithm for calculating the transformation distance is presented. Firstly, in the preprocessing part we show how to find efficiently the existence of all the substrings of one string in another one. Then the core of the algorithm is presented, which is basically a dynamic programming algorithm.

In section 4, we introduce the point deletions in our model and we give an algorithm to solve the transformation distance problem in presence of point deletions: this algorithm runs in time $O(n^3)$ and space $O(n^2)$.

Finally, section 5 is dedicated to conclusions and remarks.

2 Model and Problem Description

The symbols are elements from an alphabet Σ. The set of all finite-length strings formed using symbols from alphabet Σ is denoted by Σ^*. In this paper, we use the letters x, y, z,... for the symbols in Σ and S, T, P, R, ... for strings over Σ^*. The empty string is denoted by ϵ. The length of a string S is denoted by $|S|$. The *concatenation* of a string P and R, denoted PR, has length $|P| + |R|$ and consists of the symbols from P followed by the symbols from R.

We will denote by $S[i]$ the symbol in position i of the string S (the first symbol of a string S is $S[1]$). The substring of S starting at position i and ending at position j is denoted by $S[i..j] = S[i]S[i+1]\ldots S[j]$. The *reverse* of a string S is denoted by S^{-1}. Thus, if n is the length of S, $S^{-1}[i..j] = S[(n-j+1)..(n-i+1)]^{-1}$ and $S[i..j]^{-1} = S^{-1}[(n-j+1)..(n-i+1)]$. We say that a string P is a *prefix* of a string S, denoted $P \sqsubseteq S$, if $S = PR$ for some string $R \in \Sigma^*$. Similarly, we say that a string P is a *suffix* of a string S, denoted by $P \sqsupseteq S$, if $S = RP$ for some $R \in \Sigma^*$. For brevity of notation, we denote the k-symbol prefix $P[1..k]$ of a string pattern $P[1..m]$ by P_k. Thus, $P_0 = \epsilon$ and $P_m = P = P[1..m]$. We recall the definition of a *subsequence*: Given a string $S[1..n]$, another string $R[1..k]$ is a *subsequence* of S, denoted by $R \prec S$, if there exists a strictly increasing sequence $< i_1, i_2, \ldots, i_k >$ of indices of S such that for all $j = 1, 2, \ldots, k$, we have $S[i_j] = R[j]$. For example, if $S = xxyzyyzx$, $R = zzxx$ and $P = xxzz$, then P is a subsequence of S, while R is not a subsequence of S. When a string S is a subsequence of a string T, T is called a *supersequence* of S, denoted by $T \succ S$. In the last example, S is a supersequence of P.

Varré et al. [VDR98, VDR99] propose a new measure which evaluates segment-based dissimilarity between two strings: the source string S and the target string T. This measure is related to the process of constructing the target string T with *segment operations*[2]. The construction starts with the empty string ϵ and proceeds from left to right by adding segments (concatenation), one segment per operation. The left-to-right generation is not a restriction but a fact that can be formally proved. A list of operations is called a *script*. Three types of segment operations are considered: the *copy* adds segments that are contained in the source string S, the *reverse copy* adds

[2] In this paper we use segment as an equivalent word for substring.

the segments that are contained in S in reverse order, and the *insertion* adds segments that are not necessarily contained in S. The measure depends on a parameter that is the *Minimum Factor Length (MFL)*; it is the minimum length of the segments that can be copied or reverse copied. Depending on the number of common segments between S and T, there exist several scripts for constructing the target T. Among these scripts, some are more likely; in order to identify them, we introduce a cost function for each operation. $InsertCost(T[i..j])$ is the cost of insertion of substring $T[i..j]$. $CopyCost(T[i..j])$ is the cost of copying the segment $T[i..j]$ from S if it is contained in S. Finally $RevCopyCost(T[i..j])$ is the cost of copying substring $T[i..j]$ from S if the reverse of this substring is contained in the source S. The cost of a script is the sum of the costs of its operations. The *minimal scripts* are all scripts of minimum cost and the *transformation distance*[3] *(TD)* is the cost of a minimal script. The problem which we solve in this paper is the computation of the transformation distance. It is clear that it is also possible to get a minimal script.

3 Algorithm

In this section we describe the algorithm to determine the transformation distance between two strings. The algorithm consists of two parts. The first part is a preprocessing part in which we determine for each substring of target string T, whether it exists in the source string S or not. In the second part, which is the core algorithm, we determine the transformation distance with help of the information that we obtained in the preprocessing part. This core algorithm is a dynamic programming algorithm.

3.1 Preprocessing

Deciding whether a given substring exists in S or not, and finding its position in the case of presence, needs to apply a *string matching* algorithm. For this aim, we design an algorithm based on KMP (Knutt-Moris-Pratt) string matching algorithm with some changes. Let $FP[i, j]$ be the the first position of occurrence of the substring $T[i..j]$ in S if such an occurrence exists and ∞ otherwise. Similarly $FPR[i, j]$ is the first position of an occurrence of $T^{-1}[i..j]$ in S. We need to recall the definition of *prefix function* π (adapted from the original KMP one), which is needed for precomputation. Given a pattern $P[1..m]$, the prefix function for pattern P is the function $\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m-1\}$ such that $\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$. That is, π_q is the length of the longest prefix of P that is a proper suffix of P_q. We have the following lemma for the prefix functions.

Lemma 1 The prefix function of P_k is a restriction of prefix function of P to the set $\{1, 2, \ldots, k\}$.

Proof: The proof is immediate by the definition of the prefix function because $\pi[i]$ for a given i can be obtained only from $P_{i-1} = P[1..(i-1)]$ and $P[i]$.

Although simple, this lemma is a corner-stone of the algorithm. It shows that, one can search for the presence of the prefixes of a pattern string in the source string, in the

[3] Although this measure is not a mathematical distance but we will use the term transformation distance which was introduced by Varré et al. [VDR98, VDR99].

```
Algorithm 1 Prefix-Matcher(A, S, P, index)   %% index = |T| + 1− length of the
n ← length[S]                                %%   suffix P being searched in S
m ← length[P]                                %% A[n×n] : A[i, i+q] ≠ ∞ iff the prefix
q ← 0                                        %%   of P of length q+1 occurs in S
for i ← 1 to n
do  while q > 0 and P[q+1] ≠ S[i]
        do q ← π[q]
    if P[q+1] = S[i] then
        q ← q + 1
    if A[index, index + q] = ∞ then
        A[index, index + q] = i − q
    if q = m then
        Exit                                 %% the suffix P has been discovered
```

Figure 1: Prefix-Matcher

```
Algorithm 2 PreProcessing(S, T)
FillArray(FP, ∞)
FillArray(FPR, ∞)
n ← length[T]
for k ← 1 to n
do  P ← T[k..n]
    Prefix-Matcher(FP, S, P, k)              %% direct pattern
    PR ← T^{-1}[k..n]
    Prefix-Matcher(FPR, S, PR, k)            %% reverse pattern
```

Figure 2: PreProcessing

same time of searching for the complete pattern, without increasing the complexity of the search. The algorithm is given in pseudocode in figure 1 as the procedure **Prefix-Matcher**. The complexity of the Prefix-Matcher algorithm is $O(n)$ in time. For the proof of the complexity and correctness of this algorithm, see chapter 34.4 of [CLR90]. Prefix-Matcher finds the position of the first occurrence of all prefixes of a pattern string P in string S. In the **PreProcessing** algorithm (figure 2), we call the Prefix-Matcher with patterns $T[1..n], T[2..n], ..., T[n]$. Thus, we have the position of the first occurrences of all of the substrings of T in S. Similarly, the first position of all substrings of T^{-1} are found in S. The total complexity the preprocessing part is $O(n^2)$ in time and $O(n^2)$ in space.

3.2 Core Algorithm

As the scripts construct the target string T from left to right by adding segments, dynamic programming is an ideal tool for computing the transformation distance. The core part of the algorithm determines the transformation distance between S and T by a dynamic programming algorithm. Let $C[k]$ be the minimum production cost of $T[1..k]$ using the segments of S. The algorithm is given in figure 3. We make use of generic functions *CopyCost*, *RevCopyCost* and *InsertCost* as defined at the end of section 2. These functions are defined using the PreProcessing algorithm: arrays

Algorithm 3 TransformationDistance(S, T)
1. **PreProcessing**(S, T)
2. $C[0] \leftarrow 0$
3. **for** k $\leftarrow$ 1 **to** $|T|$
4. $$C[k] \leftarrow \min_{0<i\leq k} \begin{cases} C[i-1] + CopyCost(T[i..k]) & \textbf{if } FP[i,k] < \infty \\ C[i-1] + RevCopyCost(T[i..k]) & \textbf{if } FPR[n-k+1, n-i+1] < \infty \\ C[i-1] + InsertCost(T[i..k]) & \\ \infty & \end{cases}$$
5. **return** $C[n]$

Figure 3: Transformation Distance: a dynamic programming solution

FP and *FPR*. In order to fix ideas, one can consider that these costs are proportional to the length of the searched segment (and ∞ if this segment does not occur in S). In fact any sub-additive function would be convenient.

Proposition 1 The recurrence relations of Algorithm 3, correctly determine the transformation distance of S and T.

Proof: We prove by induction on k that after the algorithm execution, $C[k]$ contains the minimum production cost of target $T[1..k]$ with the source string S. $C[0]$ is initialized to 0, because the cost of production of ϵ from S is zero.

Now, we suppose that $C[i]$ is calculated correctly for all $i < k$ for some positive value of k. Let us consider the calculation of $C[k]$. The last operation in a minimal script which generates $T[1..k]$, creates a suffix of $T[1..k]$. Let this suffix be $T[i..k]$. As the script is minimal, the script without its last operation is a minimal script for $T[1..(i-1)]$. The minimum cost of the script for $T[1..(i-1)]$ is $C[i-1]$ by induction hypothesis. If $T[i..k]$ exists in S and the last operation of the minimal script is a copy operation, the minimal cost of the script is $C[i-1] + CopyCost(T[i..k])$. Similarly, if the reverse of $T[i..k]$ exists in S and the last operation in the minimal script of $T[1..k]$ is a reverse copy operation, the minimal cost of the script is $C[i-1] + RevCopyCost(T[i..k])$. Finally, if the last operation in the minimal script of $T[1..k]$ is an insertion, the minimal cost of the script is $C[i-1] + InsertCost(T[i..k])$ (see figure 4). Thus, $C[n]$ is the minimum cost of production of $T = T[1..n]$ and the algorithm determines correctly the transformation distance of S and T.

Note that when the length of the substring $T[i..k]$ is smaller than MFL, *CopyCost*$(T[i..k])$ and *RevCopyCost*$(T[i..k])$ are equal to ∞.

The complexity of Algorithm 3 is $O(n^2)$ in time and $O(n)$ in space. So the total complexity of our algorithm (preprocessing + core algorithm) is $O(n^2)$ in time and $O(n^2)$ in space.

4 An Additional Operation: Point Deletion

In this section, we extend the set of evolutionary operations by adding the *point deletion* operation. During a point deletion (or simply deletion) operation, a symbol of the string which is under evolution is eliminated. This is an important operation from

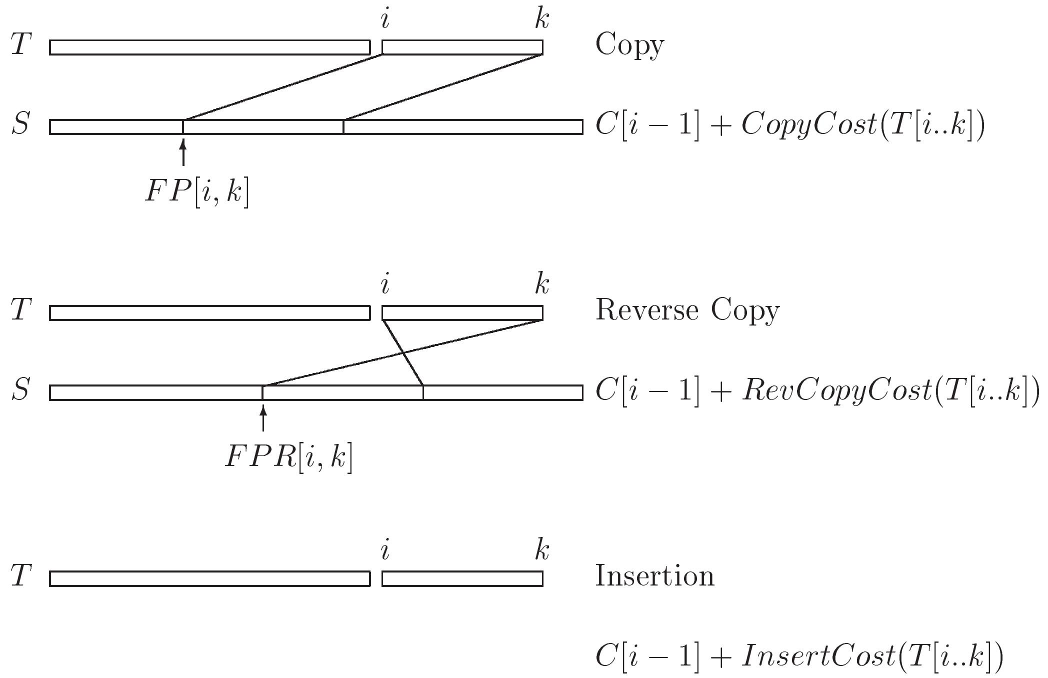


Figure 4: The three different possibilities for generation of a suffix of $T[1..k]$

the biological point of view; in the real evolution of biological sequences, in several cases after or during the copy operations some bases (symbols) are eliminated. We denote the cost of deletion of a symbol by *DelCost*. For simplicity, we suppose that the cost of deletion of every unique symbol is the same. Since we have only point deletions, deleting a segment of k symbols amounts to delete the k symbols one by one, which will cost $k \times DelCost$. As before, our objective is to find the minimum cost for a script generating a target string T, with the help of segments of a source string S. As the costs are independent of time, we consider that the deletions are applied only in the latest added segment (rightmost one), at any moment during the evolution. It should be clear that in an optimal transformation, deletions are not applied into an inserted substring (a substring which is the result of an insertion operation). Depending on the assigned costs, deletions can be used after the copy or reverse copy operations. We consider a copy operation together with all deletions which are applied to that copied segment as a unit operation. So we have a new operation called *NewCopy* which is a copy operation followed by zero or more deletions on the copied segment. In figure 5 a schema of a NewCopy operation is illustrated. Similarly, *NewRevCopy* is a reverse copy operation followed by zero or more deletions. Solving the extended transformation distance with the point deletions, amounts to solve the transformation distance with the following three operations: Insertion, NewCopy and NewRevCopy. A substring $T[i..j]$ of the target string can be produced by a unique NewCopy operation if and only if $T[i..j]$ is a subsequence string of source S. Conversely, $T[i..j]$ can be produced by a unique NewRevCopy operation if and only if $T[i..j]^{-1}$ is a subsequence string of the source S. In a preprocessing part, the algorithm determines the minimum generation cost by a NewCopy or NewRevCopy operation, for any substring of the target string T. Very similar to the last section

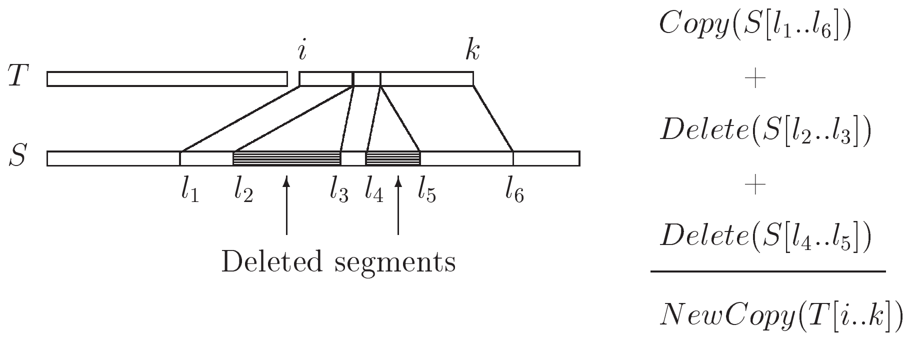


Figure 5: The illustration of NewCopy operation: A copy operation + zero or more deletions

algorithm, a dynamic programming algorithm calculates the extended transformation distance in the new core algorithm.

4.1 New Preprocessing

In the preprocessing part, we compute the costs of these new operations for any substring of the target: $NewCopyCost[i, j]$ is the minimum cost of generating the $T[i..j]$ by a NewCopy operation. Similarly, $NewRevCopyCost[i, j]$ is the minimum cost of generating $T[i..j]$ by a NewRevCopy operation. Computing the $NewCopyCost[i, j]$ amounts to find the shortest substring (with minimum length) of the source string which contains $T[i..j]$ as a subsequence string. By this way, the number of deletions which are needed for this NewCopy operation is minimized. For$NewRevCopyCost[i, j]$, we need to find the shortest substring in S^{-1} which contains $T[i..j]$ as a subsequence.

In the **NewPreProcessing** algorithm listed in figure 6, the cost tables *NewCopyCost* and *LastOcc* are initially filled with ∞ (lines 1-2). The algorithm scans the source from left to right to find the shortest supersequence for each segment of the target. The algorithm uses an auxiliary table *LastOcc* for this aim.

After the k-th letter of S is processed (loop of line 3), the following is true: $LastOcc[i, j]$ is the largest $l \leq k$ such that $S[l..k]$ is a supersequence of $T[i..j]$. The loop on T (line 4) is processed with decreasing indices for memory optimization. Whenever the letter $S[k]$ occurs in j-th position in T (line 5), then there is an opportunity of obtaining a better supersequence for some of $T[i..j]$'s, $i \leq j$. $LastOcc[i, j]$ takes the value $LastOcc[i, j-1]$ (computed for $k-1$) since $S[LastOcc[i, j-1]..k]$ is now the rightmost supersequence for $T[i..j]$ (line 9). Its cost is compared to the cost of the best previous one; if better, the new cost is stored in *NewCopyCost* (lines 11-13). One should observe that rightmost sequences are updated only when a new common letter is scanned. This is necessary and sufficient as stated in the following lemma:

Lemma 2 If $S[l..k]$ is the best supersequence for $T[i..j]$ over $S[1..N]$, then it is the rightmost supersequence for $T[i..j]$ on $S[1..k]$.

Proof: $S[l..k]$ is the best sequence for $T[i..j]$ over $S[1..k]$ then it is better than all $S[l'..k]$ for $l' < l$ and no $S[l''..k]$ can be a supersequence for $l'' < l$.

Algorithm 4 NewPreProcessing(S, T)

FillArray$(NewCopyCost, \infty)$
FillArray$(LastOcc, \infty)$ %% LastOcc is a sub-diagonal array: $LastOcc[i, j] = \infty$ for $i > j$
for $k \leftarrow 1$ **to** $|S|$ %% Source scanned left to right
  for each $j \leftarrow |T|$ **downto** 1 %% find matches in T for S[k]
%% for a fixed k: $LastOcc[i, j]$ =largest l such that $S[l..k] \succ T[i..j]$
    if $S[k] = T[j]$ **then**
      $LastOcc[j, j] \leftarrow k$
      $NewCopyCost[j, j] \leftarrow CopyCost(T[j])$ %% deletions are not needed
      for $i \leftarrow 1$ **to** $j - 1$ %% for all suffixes of T[1..j]
        $LastOcc[i, j] \leftarrow LastOcc[i, j-1]$ %% $S[LastOcc[i, j-1]..k-1]] \succ T[i..j-1]$
        $NumDel \leftarrow k - LastOcc[i, j] - i - j$ %% difference in lengths
        $ThisCost \leftarrow DelCost \times NumDel + CopyCost(S[LastOcc[i, j]..k])$
        if $ThisCost < NewCopyCost[i, j]$ **then**
            $NewCopyCost[i, j] \leftarrow ThisCost$

Figure 6: NewPreProcessing (simplified: reverse copies have been omitted)

Algorithm 5 NewTransformationDistance(S, T)

NewPreProcessing(S,T)
$C[0] \leftarrow 0$
for k $\leftarrow$ 1 **to** n

$$C[k] \leftarrow \min_{0<i\le k} \begin{cases} C[i-1] + NewCopyCost[i,k] & \text{if } FP[i,k] < \infty \\ C[i-1] + NewRevCopyCost[i,k] & \text{if } FPR[i,k] < \infty \\ C[i-1] + InsertCost(T[i..k]) & \\ \infty & \end{cases}$$

return $C[n]$

Figure 7: New Transformation Distance: dynamic programming

4.2 New Core Algorithm

In the core algorithm, the minimum generation costs of the prefixes of the target string T are determined from left to right. This is realized by a dynamic programming algorithm: Let $C[k]$ be the minimum production cost of $T[1..k]$ using the segments of S. The algorithm is given in figure 7. The proof of the following proposition is very similar to the proof of proposition 1:

Proposition 2 The recurrence relations of Algorithm 5, correctly determine the extended transformation distance of S and T.

The complexity of the preprocessing part, is $O(n^3)$ in time and $O(n^2)$ in space. The complexity of the core algorithm is $O(n^2)$ both in time and space. Therefore, the whole complexity of the new algorithm for the calculation of extended transformation distance is $O(n^3)$ in time and $O(n^2)$ in space.

Remarks and Conclusion

In this paper, we presented a new improved algorithm for calculation of the transformation distance problem. We also gave an algorithm for the transformation distance problem in presence of the deletion operations. In this version, costs have been given a special additive form for clarity. In fact a number of variations are possible within our framework: the main property needed on costs seems to be their subadditivity.

In this paper, we state that Algorithm 3 complexity is $O(n^3)$; this stands for the worst case complexity; in fact only a small proportion of pairs $(S[k], T[j])$ imply running the inner loop. Under certain additional statistical hypotheses the average complexity could be less than $O(n^3)$.

References

[BeSt03] Behzadi B. and Steyaert J.-M.: An Improved Algorithm for Generalized Comparison of Minisatellites. CPM 2003.

[BeRi02] Bérard, S., Rivals, E.: Comparison of Minisatellites. Proceedings of the 6th Annual International Conference on Research in Computational Molecular Biology. ACM Press, 2002.

[CLR90] Cormen, T.H., Leiserson, C.E., Rivest R.L.: Introduction to Algorithms. MIT Press, 1990.

[Do81] Doolittle, R.F.: Similar amino acid sequences: chance or common ancestry?, Science,214,149-159, 1981.

[SaKr83] Sankoff, D. and Kruskal, J.B: Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison. Addison-Wesley, 1983.

[Va00] Varré, J.S.: Concepts et algorithmes pour la comparaison de séquences génétiques : une approche informationnelle. PhD thesis, 2000.

[VDR99] Varré, J.S., Delahaye, J.P., Rivals, E.: Transformation Distances: a family of dissimilarity measures based on movements of segments. Bioinformatics, vol. 15, no. 3, pp 194-202, 1999.

[VDR98] Varré, J.S., Delahaye, J.P., Rivals, E.: The Transformation Distance : A Dissimilarity Measure Based On Movements Of Segments,German Conference on Bioinformatics, Koel - Germany, 1998.

[VDR97] Varré, J.S., Delahaye, J.P., Rivals, E.: The Transformation Distance. Genome Informatics Workshop, Tokyo, Japan, 1997.

Forward-Fast-Search: Another Fast Variant of the Boyer-Moore String Matching Algorithm

Domenico Cantone and Simone Faro

Dipartimento di Matematica e Informatica, Università di Catania, Italy

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

Abstract. We present a variation of the Fast-Search string matching algorithm, a recent member of the large family of Boyer-Moore-*like* algorithms, and we compare it with some of the most effective string matching algorithms, such as Horspool, Quick Search, Tuned Boyer-Moore, Reverse Factor, Berry-Ravindran, and Fast-Search itself. All algorithms are compared in terms of run-time efficiency, number of text character inspections, and number of character comparisons.

It turns out that our new proposed variant, though not linear, achieves very good results especially in the case of very short patterns or small alphabets.

Keywords: string matching, experimental algorithms, text processing.

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.

Several string matching algorithms have been proposed over the years. The Boyer-Moore algorithm [BM77] deserves a special mention, since it has been particularly successful and has inspired much work. It is based upon three simple ideas: right-to-left scanning, bad character heuristics, and good suffix heuristics. We will review it at length in Section 2.1.

Many subsequent algorithms have been based on variations on how to apply the two mentioned heuristics. For instance, the Fast-Search algorithm, recently introduced by the authors [CF03], requires 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.

In this paper, we present a variation of the Fast-Search algorithm in which the good suffix heuristics uses also a look-ahead character to determine larger advancements. We also propose a practical algorithm to precompute the table encoding such an extended good suffix rule.

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 $\text{length}(P) = m$. 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 $0 \leq i < \text{length}(P)$. 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 $0 \leq i \leq j < \text{length}(P)$. Moreover, for any $i, j \in \mathbb{Z}$, we put

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

For any two strings P and P', we write $P' \sqsupseteq P$ to indicate that P' is a suffix of P, i.e., $P' = P[i\,..\,\text{length}(P)-1]$, for some $0 \leq i < \text{length}(P)$. Similarly, we write $P' \sqsubseteq P$ to indicate that P' is a prefix of P, i.e., $P' = P[0\,..\,i-1]$, for some $0 \leq i \leq \text{length}(P)$. In addition, we write $P.P'$ to denote the concatenation of P and P'.

Let T be a text of length n and let P be a pattern of length m. When the character $P[0]$ is aligned with the character $T[s]$ of the text, so that the character $P[i]$ is aligned with the character $T[s+i]$, 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 $T[s\,..\,s+m-1] = P$, we say that the shift s is *valid*. Thus the string matching problem can be rephrased as the problem of finding *all* valid shifts of a pattern P relative to a text T.

Most string matching algorithms have the following general structure. First, during a *preprocessing phase*, they calculate useful mappings, in the form of tables, which later are accessed to determine nontrivial shift advancements. Next, starting with shift $s = 0$, they look for all valid shifts, by executing a *matching phase*, which determines whether the shift s is valid and computes a *positive* shift increment Δs. Such increment Δs is used to produce the new shift $s + \Delta s$ to be fed to the subsequent matching phase. Observe that for the correctness of the algorithm it is plainly necessary that each shift increment Δs computed is *safe*, namely the interval $\{s+1, \ldots, s+\Delta s-1\}$ contains no valid shifts.

For instance, in the case of the naive string matching algorithm, there is no preprocessing phase and the matching phase always returns a unitary shift increment, i.e., all possible shifts are actually processed.

The paper is organized as follows. In Section 2 we survey some of the most effective string matching algorithms. Next, in Section 3, we introduce a new variant of the Fast-Search algorithm. Experimental data obtained by running under various conditions all the algorithms reviewed are presented and compared in Section 4. Finally, we draw our conclusions in Section 5.

2 Some Very Fast String Matching Algorithms

In this section we briefly review the Boyer-Moore algorithm and some of its most efficient variants that have been proposed over the years. In particular, we present the Horspool [Hor80], Tuned Boyer-Moore [HS91], Quick-Search [Sun90], Berry-Ravindran [BR99], and the Fast-Search [CF03] algorithms.

We also review the Reverse Factor algorithm [CCG+94], which is based on the smallest suffix automaton of the reverse pattern.

2.1 The Boyer-Moore Algorithm

The Boyer-Moore algorithm [BM77] is the progenitor of several algorithmic variants which aim at computing close to optimal shift increments very efficiently. Specifically, the Boyer-Moore algorithm checks whether s is a valid shift by scanning the pattern P from right to left and, at the end of the matching phase, computes the shift increment as the maximum value suggested by the *good suffix rule* and the *bad character rule* below, using the functions gs_P and bc_P respectively, provided that both of them are applicable.

If the first mismatch occurs at position i of the pattern P, the good suffix rule suggests to align the substring $T[s+i+1 \ldots s+m-1] = P[i+1 \ldots m-1]$ 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 $T[s+i+1 \ldots s+m-1]$ 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

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

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.)

The bad character rule states that if $c = T[s+i] \neq P[i]$ is the first mismatching character, while scanning P and T from right to left with shift s, then P can be safely shifted in such a way that its rightmost occurrence of c, if present, is aligned with position $(s+i)$ in T. In the case in which c does not occur in P, then P can be safely shifted just past position $(s+i)$ in T. More formally, the shift increment suggested by the bad character rule is given by the expression $(i - bc_P(T[s+i]))$, where

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

for $c \in \Sigma$, and where we recall that Σ is the alphabet of the pattern P and text T. Notice that there are situations in which the shift increment given by the bad character rule can be negative.

It turns out that the functions gs_P and bc_P can be computed during the pre-processing phase in time $\mathcal{O}(m)$ and $\mathcal{O}(m + |\Sigma|)$, respectively, and that the overall worst-case running time of the Boyer-Moore algorithm, as described above, is linear (cf. [GO80]).

2.2 The Horspool Algorithm

Horspool suggested a simplification of the original Boyer-Moore algorithm, defining a new variant which, though quadratic, performed better in practical cases (cf. [Hor80]). He just dropped the good suffix rule and proposed to compute the shift advancement in such a way that the rightmost character $T[s+m-1]$ is aligned with its rightmost occurrence on $P[0 \mathinner{..} m-2]$, if present; otherwise the pattern is advanced just past the window. This corresponds to advance the shift by $hbc_P(T[s+m-1])$ positions, where

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

The resulting algorithm performs well in practice and can be immediately translated into programming code (see Baeza-Yates and Régnier [BYR92] for a simple implementation in the **C** programming language).

2.3 The Tuned Boyer-Moore Algorithm

The Tuned Boyer-Moore algorithm [HS91] can be seen as an efficient implementation of the Horspool algorithm. Again, let P be a pattern of length m. Each iteration of the Tuned Boyer-Moore algorithm can be divided into two phases: *last character localization* and *matching phase*. The first phase searches for a match of $P[m-1]$, by applying rounds of three blind shifts (based on the classical bad character rule) until needed. The matching phase tries then to 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 shift advancement is computed according to the Horspool bad character rule. Moreover, to begin with, the algorithm adds m copies of $P[m-1]$ at the end of the text, as a sentinel, to compute the last shifts correctly.

The fact that the blind shifts require no comparison is at the heart of the very good practical behavior of the Tuned Boyer-Moore, despite its quadratic worst-case time complexity (cf. [Lec00]).

2.4 The Quick-Search Algorithm

The Quick-Search algorithm, presented in [Sun90], uses a modification of the original heuristics of the Boyer-Moore algorithm, much along the same lines of the Horspool 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. This corresponds to advance the shift by $qbc_P(T[s+m])$ positions, where

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

Experimental tests have shown that that the Quick-Search algorithm is very fast especially for short patterns (cf. [Lec00]).

2.5 The Berry-Ravindran Algorithm

The Berry-Ravindran algorithm [BR99] extends the 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, where m is the size of the pattern P. Thus, at the end of each matching phase with shift s, the Berry-Ravindran algorithm advances the pattern so that the substring of the text $T[s+m \,..\, s+m+1]$ is aligned with its rightmost occurrence in P.

The precomputation of the table used by the bad character rule requires $\mathcal{O}(|\Sigma|^2)$-space and $\mathcal{O}(m + |\Sigma|^2)$-time complexity, where Σ is the alphabet of the text and pattern. Experimental results [BR99] show that the Berry-Ravindran algorithm is fast in practice and performs a low number of text/pattern character comparisons.

2.6 The Fast-Search 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 main observation upon which the Fast-Search algorithm [CF03] is based is the following: the Horspool bad character rule leads to larger shift increments than the good suffix rule if and only if a mismatch occurs immediately, while comparing the pattern P with the window $T[s \,..\, s+m-1]$, namely when $P[m-1] \neq T[s+m-1]$, where $0 \leq s \leq m-n$ is a shift.

In agreement with the above observation, the Fast-Search algorithm computes its shift increments by applying the Horspool bad character rule only if a mismatch occurs during the first character comparison. Otherwise it uses the good suffix rule.

Notice that $hbc_P(a) = bc_P(a)$, whenever $a \neq P[m-1]$, so that to compute the shift advancement one can use the traditional bad character rule, bc_P, rather then the Horspool bad character rule, hbc_P.

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.

As in the case of the Tuned Boyer-Moore algorithm, the Fast-Search algorithm benefits from the introduction of an external sentinel, which allows to compute correctly the last shifts with no extra checks.

Experimental results [CF03] show that the Fast-Search algorithm obtains the best run-time performances in most cases and, sporadically, it is second only to the Tuned Boyer-Moore algorithm. Concerning the number of text character inspections, it turns out that the Fast-Search algorithm is quite close to the Reverse Factor algorithm, which generally shows the best behavior. We notice, though, that in the case of very short patterns the Fast-Search algorithm reaches the lowest number of text character accesses.

2.7 The Reverse Factor Algorithm

Unlike the variants of the Boyer-Moore algorithm summarized above, the Reverse Factor algorithm computes shifts which match prefixes of the pattern, rather than suffixes. This is made possible by the smallest suffix automaton of the reverse of the pattern P, which is a deterministic finite automaton $S(P)$ whose accepted language is the set of suffixes of P (for a complete description see [CCG+94]).

The Reverse Factor algorithm has a quadratic worst-case time complexity, but it is very fast in practice (cf. [Lec00]). Moreover, it has been shown that on the average it inspects $\mathcal{O}(n \log(m)/m)$ text characters, reaching the best bound shown by Yao in [Yao79].

3 The Forward-Fast-Search Algorithm

In this section we present a new efficient variant of the Boyer-Moore algorithm obtained by modifying the Fast-Search algorithm presented in Section 2.6.

The new algorithmic variant, that we call Forward-Fast-Search, mantains the same structure of the Fast-Search algorithm, but 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.

The forward good suffix requires a precomputed table of size $(m \cdot |\Sigma|)$, where m is the length of the pattern and Σ is the alphabet of the text and pattern.

Concerning the running time, the forward good suffix rule can be precomputed by $|\Sigma|$ iterations of the standard linear precomputation of the Boyer-Moore good suffix rule, yielding a $\mathcal{O}(m \cdot |\Sigma|)$ time complexity. Nevertheless, we propose an alternative, more direct approach which behaves very well in practice, though it requires $\mathcal{O}(m \cdot \max(m, |\Sigma|))$ time in the worst case.

3.1 Strengthening the Good Suffix Rule

3.1.1 The Backward Good Suffix Rule

A first natural way to strengthen the good suffix rule, which yields the *backward good suffix* rule, can be obtained by merging it with the bad character rule as follows. As usual, let us assume that we are comparing a pattern P of length m with the window $T[s \mathinner{..} s+m-1]$ at shift s of a given text T, scanning it from right to left. If the first mismatch occurs at position i of the pattern P, i.e. $P[i+1 \mathinner{..} m-1] = T[s+i+1 \mathinner{..} s+m-1]$ and $P[i] \neq T[s+i]$, then the backward good suffix rule proposes to align the substring $T[s+i+1 \mathinner{..} s+m-1]$ with its rightmost occurrence in P preceded by the *backward* character $T[s+i]$. If such an occurrence does not exist, the backward good suffix rule proposes a shift increment which allows to match the longest suffix of $T[s+i+1 \mathinner{..} s+m-1]$ with a prefix of P. More formally, this corresponds to increment the shift s by $\overleftarrow{gs}_P(i+1, T[s+i])$, where

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

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

3.1.2 The Forward Good Suffix Rule

As observed by Sunday [Sun90], after a matching phase with shift s, the *forward* character $T[s+m]$ is always involved in the subsequent matching phase. Thus, another possible variant of the good suffix rule, which we call *forward good suffix* rule, consists in matching the forward character $T[s+m]$, rather than the mismatched character $T[s+i]$. More precisely, if as above the first mismatch occurs at position i of the pattern P, the forward good suffix rule suggests to align the substring $T[s+i+1 \mathinner{..} s+m]$ 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 $T[s+i+1 \mathinner{..} s+m]$ with a prefix of P. This corresponds to advance the shift s by $\overrightarrow{gs}_P(i+1, T[s+m])$ positions, where

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

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

3.1.3 Comparing the Good Suffix Rule with its Variants

We computed the average shift advancement suggested by the good suffix rule and its backward and forward variants on four Randσ problems, for $\sigma = 2, 4, 8, 20$, with pattern lengths $2, 4, 6, 8, 10, 20, 40, 80$, and 160, where a Randσ problem consists in searching, for each assigned value of the pattern length, a set of 200 random patterns over an alphabet Σ of size σ in a 20Mb random text over the same alphabet Σ.

Experimental results, presented in the tables below, show that the forward and backward good suffix rules propose on the average much larger shift advancements than the standard good suffix rule (up to 400% better). In addition, the forward good suffix rule shows always a slightly better behavior than the backward one, which becomes more sensible in the case of very small alphabets. This is partly due to the fact that the forward character is always used by the forward good suffix rule to compute shift advancements, whereas there are cases in which the backward good suffix rule does not exploit the backward character.

$\sigma = 2$	2	4	6	8	10	20	40	80	160
gs	1.540	2.762	3.869	4.765	5.468	8.464	12.254	16.137	21.807
$\overleftarrow{gs}$	1.540	2.762	3.869	4.765	5.468	8.464	12.254	16.137	21.807
$\overrightarrow{gs}$	**2.269**	**3.642**	**5.026**	**6.310**	**7.394**	**12.21**	**18.200**	**25.586**	**34.798**

$\sigma = 4$	2	4	6	8	10	20	40	80	160
gs	1.750	3.062	4.334	5.196	6.079	8.697	12.382	16.857	22.645
$\overleftarrow{gs}$	1.750	3.540	5.170	6.691	8.097	13.62	21.604	30.540	42.891
$\overrightarrow{gs}$	**2.687**	**4.407**	**6.114**	**7.696**	**9.245**	**15.55**	**25.149**	**36.584**	**51.398**

$\sigma = 8$	2	4	6	8	10	20	40	80	160
gs	1.880	3.453	4.833	5.399	6.656	10.05	13.613	19.510	25.807
$\overleftarrow{gs}$	1.880	3.857	5.692	7.441	9.294	17.63	31.570	51.010	75.734
$\overrightarrow{gs}$	**2.860**	**4.775**	**6.671**	**8.399**	**10.24**	**18.72**	**33.225**	**54.825**	**81.334**

$\sigma = 20$	2	4	6	8	10	20	40	80	160
gs	1.930	3.714	5.238	6.684	8.512	12.81	19.078	25.169	33.975
$\overleftarrow{gs}$	1.930	3.956	5.892	7.919	9.867	19.47	38.167	72.950	136.45
$\overrightarrow{gs}$	**2.946**	**4.929**	**6.896**	**8.868**	**10.85**	**20.44**	**39.206**	**74.084**	**138.22**

Average advancements for some Randσ problems

3.1.4 Implementing the Forward Good Suffix Rule

Given a pattern P of length m over an alphabet Σ, we have plainly

$$\overrightarrow{gs}_P(j, c) = gs_{P.c}(j) \ ,$$

for $j = 0, 1, \ldots, m$ and $c \in \Sigma$, where $P.c$ is the string obtained by concatenating the character c at the end of P. Thus, a natural way to compute the forward good suffix function $\overrightarrow{gs}_P$ consists in computing the standard good suffix functions $gs_{P.c}$, for all $c \in \Sigma$, by means of the $\mathcal{O}(m)$ tricky algorithm firstly given in [KMP77] and then corrected in [Rit80].

Such a procedure is asymptotically optimal, as it has $\mathcal{O}(m \cdot |\Sigma|)$ space and time complexity.

In Figure 1 we propose an alternative procedure to compute the forward good suffix function which, despite its $\mathcal{O}(m \cdot \max(m, |\Sigma|))$ worst-case time complexity, turns out to be very efficient in practice, even for large values of m.

precompute-forward-good-suffix(P)
Initialization:
1. $m = \text{length}(P)$
2. **for** $i = 0$ **to** m **do**
3. **for** $c \in \Sigma$ **do**
4. $\overrightarrow{gs}[i, c] = m + 1$
5. **for** $i = 0$ to $m - 1$ **do**
6. $next[i] = i - 1$

Computation:
7. **for** $slen = 0$ **to** $m - 1$ **do**
8. $last = m - 1$
9. $i = next[last]$
10. **while** $i \geq 0$ **do**
11. **if** $\overrightarrow{gs}[m - slen, P[i + 1]] > m - 1 - i$ **then**
12. **if** $(i - slen < 0$ **or**
13 $(i - slen \geq 0$ **and** $P[i - slen] \neq P[m - 1 - slen]))$ **then**
14. $\overrightarrow{gs}[m - slen, P[i + 1]] = m - 1 - i$
15. **if** $(i - slen \geq 0$ **and** $P[i - slen] = P[last - slen])$ **or**
16. $(i - slen < 0)$ **then**
17. $next[last] = i$
18. $last = i$
19. $i = next[i]$
20. **if** $\overrightarrow{gs}[m - slen, P[0]] > m$ **then**
21. $\overrightarrow{gs}[m - slen, P[0]] = m$
22. $next[last] = -1$
23. **return** $\overrightarrow{gs}$

Figure 1: The function for computing forward good suffixes

After an initialization phase which takes $\mathcal{O}(m \cdot |\Sigma|)$ space and time complexity, the precompute-forward-good-suffix procedure carries out m iterations of its main **for**-loop, starting at line 7. During the k-th iteration, for $k = 1, 2, \ldots, m$, it computes the sequence $\mathcal{S}_k(P)$ of all occurrences in P of the suffix $P[m - k \,..\, m - 1]$ of length k, implicitly represented by means of the array *next*:

$$\begin{aligned}\mathcal{S}_k(P) \;=\; \langle\; & P[next[m-1] - k + 1 \,..\, next[m-1]] \;, \\ & P[next^{(2)}[m-1] - k + 1 \,..\, next^{(2)}[m-1]], \\ & \ldots\ldots \\ & P[next^{(r_k)}[m-1] - k + 1 \,..\, next^{(r_k)}[m-1]] \;\rangle \;\;,\end{aligned} \tag{1}$$

where r_k is such that $next^{(r_k+1)}[m-1] = -1$. For that purpose, lines 15-18 implement the recurrence

$$\mathcal{S}_k(P) = \langle P[j - k + 1 \,..\, j] \mid P[j - k + 2 \,..\, j] \in \mathcal{S}_{k-1}(P) \text{ and } P[j - k + 1] = P[m - k] \rangle \,,$$

where $\mathcal{S}_0(P)$ is also formally given by (1), thanks to the way the array *next* is initialized in lines 5-6. Moreover, during the k-th iteration of the **for**-loop, for each

$P[j-k+1\,..\,j] \in \mathcal{S}_k(P)$, the procedure updates, if necessary, the value $\overrightarrow{gs}(m-k-1, P[j+1])$ by setting it to $(m-1-j)$ (lines 11-14).

Plainly, the procedure in Figure 1 requires $\mathcal{O}(m \cdot |\Sigma|)$ space. To compute its time complexity, it is enough to observe that the k-th execution of the **while**-loop in lines 10-19, for $k = 1, 2, \ldots, m$, takes $\mathcal{O}(|\mathcal{S}_{k-1}(P)|)$ time, giving a total of $\mathcal{O}(\sum_{j=0}^{m-1} |\mathcal{S}_j(P)|) = \mathcal{O}(m^2)$ time in the worst case. This leads to an overall $\mathcal{O}(m \cdot \max(m, |\Sigma|))$ worst-case time complexity, taking into account also the initizialization phase.

Experimental results show that the sum $\sum_{j=0}^{m-1} |\mathcal{S}_j(P)|$ has on the average an almost linear behavior. For instance, the following tables report the average of the sum $\sum_{j=0}^{m-1} |\mathcal{S}_j(P)|$ computed for $100,000$ random patterns of size m over an alphabet of size σ, for $\sigma = 2, 4, 8, 20$ and $m = 2, 4, 6, 8, 10, 20, 40, 80, 160$. The tests relative to a natural language buffer NL have been computed by randomly selecting $100,000$ substrings for each given pattern length over the 3.13Mb file obtained by discarding the nonalphabetic characters from the WinEdt spelling dictionary.

m	2	4	6	8	10	20	40	80	160
m^2 (worst case)	4	16	36	64	100	400	1600	6400	25600
Average for $\sigma = 2$	2.50	7.38	13.07	19.01	25.02	55.09	114.89	234.98	474.57
Average for $\sigma = 4$	2.24	5.46	8.76	12.10	15.45	32.09	65.34	132.06	264.98
Average for $\sigma = 8$	2.12	4.67	7.23	9.81	12.40	25.24	50.93	102.45	204.98
Average for $\sigma = 20$	2.04	4.25	6.46	8.68	10.89	21.96	44.00	88.21	176.63
Average on NL	2.04	4.23	6.47	8.84	11.99	28.57	57.97	111.61	208.00

For the same set of random tests, we also computed the *total* time taken to construct the forward good suffix function $\overrightarrow{gs}$, using the two implementations described earlier, namely the one which has a $\mathcal{O}(m \cdot |\Sigma|)$ worst-case time and space complexity and the procedure precompute-forward-good-suffix. Such implementations are denoted respectively "$\overrightarrow{gs}$ (I)" and "$\overrightarrow{gs}$ (II)" in the tables below, where experimental results are expressed in hundredths of seconds.

$\sigma = 2$	2	4	6	8	10	20	40	80	160
$\overrightarrow{gs}$ (I)	58.1	60.1	63.1	66.1	68.1	81.1	103.2	149.2	**239.3**
$\overrightarrow{gs}$ (II)	**3.0**	**6.0**	**11.0**	**15.1**	**18.0**	**37.0**	**74.1**	**145.3**	288.4

$\sigma = 4$	2	4	6	8	10	20	40	80	160
$\overrightarrow{gs}$ (I)	113.2	117.1	121.2	124.2	128.2	142.2	174.2	235.4	357.5
$\overrightarrow{gs}$ (II)	**3.0**	**6.0**	**10.0**	**13.0**	**16.0**	**33.1**	**64.1**	**126.2**	**250.3**

$\sigma = 8$	2	4	6	8	10	20	40	80	160
$\overrightarrow{gs}$ (I)	225.3	230.4	237.3	240.4	243.3	268.4	313.4	401.6	577.9
$\overrightarrow{gs}$ (II)	**4.0**	**7.0**	**11.0**	**14.0**	**19.0**	**36.1**	**72.1**	**141.2**	**289.4**

$\sigma = 20$	2	4	6	8	10	20	40	80	160
$\overrightarrow{gs}$ (I)	558.8	573.9	580.8	589.8	598.9	642.9	733.1	905.3	1250.8
$\overrightarrow{gs}$ (II)	**5.0**	**11.0**	**16.0**	**20.1**	**26.0**	**50.1**	**98.1**	**195.3**	**394.6**

NL	2	4	6	8	10	20	40	80	160
$\overrightarrow{gs}$ (I)	553.8	565.8	573.8	583.8	592.8	636.9	725.0	895.3	1238.8
$\overrightarrow{gs}$ (II)	**5.0**	**10.0**	**16.0**	**19.0**	**23.1**	**48.1**	**95.1**	**189.3**	**379.5**

```
Forward-Fast-Search(P, T)
    n = length(T)
    m = length(P)
    T' = T.P[m − 1]^(m+1)
    bc = precompute-bad-character(P)
    gs→ = precompute-forward-good-suffix(P)
    s = 0
    while bc[T'[s + m − 1]] > 0 do
        s = s + bc[T'[s + m − 1]]
    while s ≤ n − m do
        j = m − 2
        while j ≥ 0 and P[j] = T'[s + j] do
            j = j − 1
        if j < 0 then
            print(s)
        s = s + gs→[j + 1, T[s + m]]
        while bc[T'[s + m − 1]] > 0 do
            s = s + bc[T'[s + m − 1]]
```

Figure 2: The Forward-Fast-Search algorithm

The analysis of the above experimental results show that for alphabets of size at least 4 the procedure precompute-forward-good-suffix is on the average always faster than the implementation of the forward good suffix function described at the beginning the present section.

3.2 Building up the Forward-Fast-Search Algorithm

The implementation of the Forward-Fast-Search algorithm can be obtained along the same lines of the Fast-Search and the Tuned Boyer-Moore algorithms.

In the first phase, called *character localization* phase, the algorithm iterates the bad character rule until the last character $P[m-1]$ of the pattern is 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 bad character rule, then we have the following recurrence:

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

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

At this point we have that $T[s + j_k + m - 1] = P[m-1]$, so that the subsequent *matching* phase can test for an occurrence of the pattern by comparing only the remaining $(m-1)$ characters of the pattern. At the end of the *matching* phase the algorithm applies the forward good suffix rule instead of the traditional good suffix rule.

As in the case of the Fast-Search and Tuned Boyer-Moore algorithms, the Forward-Fast-Search algorithm benefits from the introduction of an external sentinel: since the forward good suffix rule looks at the character $T[s+m]$ just after the current window, $m+1$ copies of the character $P[m-1]$ are added at the end of the text T, obtaining a new text $T' = T.P[m-1]^{m+1}$. This allows to compute correctly the last shifts with no extra checks. Plainly, all the valid shifts of P in T are the 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 code of the Forward-Fast-Search algorithm is presented in Figure 2.

4 Experimental Results

We present next experimental data which allow to compare the following string matching algorithms under various conditions: Horspool (HOR), Quick-Search (QS), Barry-Ravidran (BR), Tuned Boyer-Moore (TBM), Reverse Factor (RF), Fast-Search (FS), and Forward-Fast-Search (FFS).

We have chosen to compare the algorithms in terms of running time, number of text character inspections, and number of character comparisons.

All algorithms have been implemented in the **C** programming language and were used to search for the same strings in large fixed text buffers on a PC with AMD Athlon processor of 1.19GHz. In particular, the algorithms have been tested on four Randσ problems, for $\sigma = 2, 4, 8, 20$, and on a natural language text buffer NL with patterns of length $m = 2, 4, 6, 8, 10, 20, 40, 80$, and 160.

We recall that each Randσ problem consists in searching a set of 200 random patterns of a given length in a 20Mb random text over a common alphabet of size σ.

The tests on the natural language text buffer NL have been performed on a 3.13Mb file obtained by discarding the nonalphabetic characters from the WinEdt spelling dictionary. For each pattern length m, we have selected 200 random substrings of length m in the file which subsequently have been searched for in the same file.

4.1 Running Times

Experimental results show that the Forward-Fast-Search algorithm obtains the best run-time performance in most cases and, sporadically, it is second only to the Fast-Search algorithm, in the case of natural language texts and long patterns, and to the Berry-Ravidran algorithm, in the case of large alphabets and patterns.

In the following tables, running times are expressed in hundredths of seconds.

$\sigma = 2$	2	4	6	8	10	20	40	80	160
HOR	42.01	44.18	42.86	42.02	46.57	40.24	39.51	38.83	39.95
QS	34.33	41.12	38.35	39.30	42.80	37.42	36.77	36.42	36.54
BR	44.84	49.36	44.42	43.48	47.69	40.66	40.70	40.74	40.54
TBM	33.96	36.54	36.88	36.65	40.53	35.98	36.05	35.54	36.30
RF	249.2	200.0	145.9	114.2	107.3	57.95	36.84	27.95	22.36
FS	41.79	35.36	28.72	25.32	26.15	20.40	18.40	17.99	17.31
FFS	**31.08**	**28.87**	**25.28**	**22.37**	**23.15**	**18.05**	**16.78**	**16.62**	**15.82**

Running times for a Rand2 problem

$\sigma = 4$	2	4	6	8	10	20	40	80	160
HOR	34.66	25.57	22.05	20.76	20.27	19.68	20.05	19.54	20.20
QS	26.49	22.10	19.87	19.35	18.98	18.58	19.05	18.73	19.04
BR	32.20	25.68	22.08	20.31	19.24	17.29	16.66	16.36	16.51
TBM	25.53	20.68	19.15	18.85	18.76	18.50	18.81	18.38	18.78
RF	156.1	98.60	74.84	62.28	53.79	34.73	24.26	20.34	16.67
FS	28.60	20.58	18.91	18.26	17.86	17.22	16.53	16.18	15.82
FFS	**24.87**	**20.06**	**18.35**	**17.65**	**17.22**	**16.23**	**15.61**	**15.33**	**14.40**

Running times for a Rand4 problem

$\sigma = 8$	2	4	6	8	10	20	40	80	160
HOR	27.71	20.19	18.40	17.43	16.84	15.70	15.56	15.62	15.71
QS	20.91	18.27	17.17	16.59	16.25	15.36	15.22	15.23	15.35
BR	25.19	20.55	18.77	17.74	17.02	15.33	14.55	14.55	13.96
TBM	21.09	17.78	16.78	16.77	16.22	15.14	15.11	15.05	15.18
RF	114.8	70.75	54.97	46.27	40.62	27.26	20.58	18.17	15.01
FS	20.66	17.75	16.75	16.41	16.01	15.02	14.89	14.80	14.81
FFS	**20.20**	**17.58**	**16.60**	**16.17**	**15.82**	**14.87**	**14.54**	**14.52**	**13.92**

Running times for a Rand8 problem

$\sigma = 20$	2	4	6	8	10	20	40	80	160
HOR	23.45	18.17	16.58	16.21	15.89	15.21	14.90	14.84	14.98
QS	18.67	16.84	15.78	15.69	15.49	14.98	14.74	14.73	14.79
BR	21.83	18.88	17.32	16.89	16.47	15.47	14.90	**14.42**	**12.60**
TBM	**18.76**	16.78	15.64	**15.44**	15.39	14.85	14.82	14.65	14.65
RF	92.44	54.83	41.67	35.57	31.61	23.12	19.25	17.69	14.72
FS	19.11	16.59	15.57	15.49	**15.24**	**14.81**	14.66	14.65	14.58
FFS	**18.76**	**16.51**	**15.51**	**15.44**	**15.24**	14.83	**14.64**	14.65	14.35

Running times for a Rand20 problem

NL	2	4	6	8	10	20	40	80	160
HOR	3.40	2.65	2.45	2.36	2.36	2.22	2.15	2.11	1.98
QS	2.73	2.42	2.35	**2.24**	2.20	**2.14**	2.09	**2.09**	2.01
BR	3.28	2.87	2.66	2.59	2.47	2.33	2.25	2.21	1.95
TBM	2.77	2.39	2.27	2.25	**2.18**	2.19	2.09	2.12	1.93
RF	13.94	8.33	6.48	5.46	4.87	3.35	2.79	2.68	4.67
FS	2.79	2.45	**2.22**	**2.24**	2.19	**2.14**	**2.06**	**2.09**	**1.91**
FFS	**2.70**	**2.35**	2.26	2.26	**2.18**	2.15	2.13	2.11	2.24

Running times for a natural language problem

4.2 Average Number of Text Character Inspections

For each test, the average number of character inspections has been obtained by taking the total number of times a text character is accessed, either to perform a comparison with a pattern character, or to perform a shift, or to compute a transition in an automaton, and dividing it by the length of the text buffer.

It turns out that the Forward-Fast-Search algorithm is always very close the best results which are generally obtained by the Fast-Search algorithm, for short patterns, and by Reverse-Factor algorithm, for long patterns. We notice, however, that the Forward-Fast-Search algorithm obtains in most cases the second best result and is better than Reverse-Factor, for short patterns, and Fast-Search, for long patterns.

$\sigma = 2$	2	4	6	8	10	20	40	80	160
HOR	**1.00**	1.15	1.26	1.26	1.28	1.24	1.27	1.23	1.27
QS	1.54	1.67	1.63	1.67	1.64	1.61	1.65	1.61	1.60
BR	1.28	1.25	1.20	1.20	1.19	1.19	1.19	1.18	1.16
TBM	1.23	1.35	1.46	1.46	1.47	1.43	1.46	1.42	1.46
RF	1.43	1.06	**.799**	**.615**	**.519**	**.294**	**.169**	**.096**	**.054**
FS	**1.00**	**.929**	.806	.698	.632	.460	.348	.270	.213
FFS	1.15	.993	.833	.703	.621	.410	.289	.210	.161

$\sigma = 4$	2	4	6	8	10	20	40	80	160
HOR	**.714**	.510	.435	.404	.392	.373	.389	.365	.392
QS	1.03	.817	.700	.675	.645	.610	.650	.622	.633
BR	.949	.713	.569	.488	.429	.307	.264	.244	.251
TBM	.841	.591	.504	.468	.454	.432	.450	.422	.446
RF	.886	.528	**.387**	**.316**	**.264**	**.154**	**.089**	**.051**	**.028**
FS	**.714**	**.489**	.398	.356	.330	.273	.239	.200	.177
FFS	.768	.526	.418	.367	.330	.241	.182	.136	.105

$\sigma = 8$	2	4	6	8	10	20	40	80	160
HOR	**.600**	.350	.263	.222	.198	.158	.153	.149	.152
QS	.842	.575	.456	.393	.358	.291	.282	.278	.277
BR	.844	.582	.443	.360	.305	.179	.109	.072	.057
TBM	.663	.386	.291	.245	.218	.174	.168	.164	.167
RF	.674	.381	.278	.225	**.191**	**.112**	**.063**	**.036**	**.020**
FS	**.600**	**.348**	**.260**	**.217**	.193	.150	.137	.126	.117
FFS	.627	.368	.274	.227	.201	.146	.117	.093	.075

$\sigma = 20$	2	4	6	8	10	20	40	80	160
HOR	**.538**	.285	.199	.157	.132	.083	.061	.054	.053
QS	.734	.463	.346	.282	.242	.157	.118	.104	.104
BR	.787	.528	.397	.318	.266	.146	.078	.042	.023
TBM	.563	.297	.208	.164	.137	.086	.063	.056	.056
RF	.565	.302	.214	.170	.143	.084	**.049**	**.027**	**.014**
FS	**.538**	**.284**	**.198**	**.156**	**.131**	**.082**	.060	.053	.052
FFS	.550	.293	.205	.161	.135	**.082**	.060	.049	.043

NL	2	4	6	8	10	20	40	80	160
HOR	**.550**	.300	**.211**	**.171**	.144	.091	.059	.042	.032
QS	.759	.489	.375	.309	.261	.175	.125	.086	.066
BR	.795	.538	.411	.335	.278	.155	.085	.050	.028
TBM	.584	.318	.226	.182	.153	.096	.062	.044	.034
RF	.588	.321	.231	.185	.153	**.084**	**.045**	**.024**	**.013**
FS	**.550**	**.299**	**.211**	**.171**	**.143**	.087	.055	.038	.027
FFS	.565	.312	.220	.180	.152	.088	.054	.036	.026

Average number of text character inspections for some Randσ problems and for a natural language problem

4.3 Average Number of Comparisons

For each test, the average number of character comparisons has been obtained by taking the total number of times a text character is compared with a character in the pattern and dividing it by the total number of characters in the text buffer.

It turns out that the Forward-Fast-Search algorithm achieves the best results in most cases. Sporadically our algorithm is second only to the Berry-Ravindran algorithm which obtains very good results for short patterns and small alphabets. Moreover we observe that Tuned Boyer-Moore, Fast-Search and Forward-Fast-Search algorithms perform a very low number of characters comparisons in the case of large alphabets.

$\sigma = 2$	2	4	6	8	10	20	40	80	160
HOR	1.000	1.159	1.260	1.269	1.281	1.244	1.272	1.235	1.270
QS	.9588	1.109	1.088	1.119	1.095	1.073	1.104	1.079	1.080
BR	**.2631**	**.3766**	.3916	.3989	.3962	.3973	.3969	.3940	.3893
TBM	.3333	.6044	.6995	.7154	.7249	.7082	.7215	.7024	.7205
FS	.3333	.4767	.4466	.3925	.3573	.2609	.1967	.1530	.1248
FFS	.3076	.4224	**.3875**	**.3324**	**.2962**	**.1964**	**.1377**	**.1003**	**.0766**

$\sigma = 4$	2	4	6	8	10	20	40	80	160
HOR	.7143	.5100	.4356	.4041	.3922	.3732	.3890	.3652	.3928
QS	.6053	.4864	.4109	.3908	.3716	.3491	.3719	.3556	.3742
BR	.2747	.2353	.1898	.1628	.1432	.1025	.0883	.0813	.0837
TBM	.1429	.1445	.1264	.1175	.1140	.1085	.1131	.1062	.1141
FS	.1429	.1373	.1141	.1024	.0949	.0784	.0690	.0577	.0526
FFS	**.1323**	**.1272**	**.1041**	**.0913**	**.0822**	**.0601**	**.0454**	**.0341**	**.0263**

$\sigma = 8$	2	4	6	8	10	20	40	80	160
HOR	.6000	.3501	.2639	.2222	.1985	.1586	.1531	.1490	.1522
QS	.4631	.3189	.2505	.2139	.1943	.1559	.1504	.1487	.1524
BR	.2711	.1940	.1479	.1202	.1018	.0598	.0364	.0243	.0190
TBM	.0667	.0482	.0365	.0307	.0274	.0219	.0212	.0206	.0210
FS	.0667	.0477	.0359	.0300	.0267	.0207	.0190	.0175	.0167
FFS	**.0634**	**.0459**	**.0345**	**.0287**	**.0252**	**.0184**	**.0148**	**.0117**	**.0095**

$\sigma = 20$	2	4	6	8	10	20	40	80	160
HOR	.5385	.2844	.1991	.1569	.1316	.0828	.0608	.0541	.0537
QS	.3837	.2427	.1805	.1476	.1263	.0817	.0607	.0538	.0534
BR	.2608	.1760	.1323	.1061	.0887	.0490	.0263	.0141	.0079
TBM	.0256	.0149	.0104	.0082	.0069	.0043	.0032	.0028	.0028
FS	.0256	.0149	.0104	.0082	.0069	.0043	.0032	.0028	.0027
FFS	**.0251**	**.0147**	**.0103**	**.0081**	**.0068**	**.0042**	**.0030**	**.0025**	**.0022**

NL	2	4	6	8	10	20	40	80	160
HOR	.5501	.3000	.2117	.1716	.1445	.0913	.0595	.0420	.0329
QS	.4031	.2605	.2002	.1646	.1393	.0914	.0654	.0455	.0364
BR	.2599	.1794	.1371	.1118	.0927	.0519	.0286	.0168	.0094
TBM	.0345	.0245	.0171	.0142	.0123	.0089	.0061	.0046	.0042
FS	.0345	.0245	.0171	**.0141**	**.0121**	.0066	.0043	.0030	.0025
FFS	**.0333**	**.0244**	**.0168**	.0153	.0140	**.0058**	**.0032**	**.0020**	**.0014**

Average number of comparisons for some Randσ problems and for a natural language problem

5 Conclusion

We presented a new efficient variant of the Boyer-Moore string matching algorithm, named Forward-Fast-Search. As its progenitor Fast-Search, the Forward-Fast-Search algorithm applies repeatedly the bad character rule until the last character of the pattern is matched correctly and then it begins to match the pattern against the text from right to left. At the end of each matching phase, it computes the shift advancement as a function of the matched suffix of the pattern and the first character of the text past the current window (forward good suffix rule).

It turns out that, despite the $\mathcal{O}(m \cdot |\Sigma|)$-space and $\mathcal{O}(m \cdot \max(m, |\Sigma|))$-time complexity required in the worst case to precompute the forward good suffix function, the

Forward-Fast-Search algorithm is very fast in practice and compares well with other fast variants of the Boyer-Moore algorithm.

We plan to evaluate theoretically the average time complexity of the Forward-Fast-Search algorithm, and to adapt it to scanning strategies depending on the character frequencies.

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. *Proc. of the Prague Stringology Club Workshop '99* Czech Technical University, Prague, Czech Republic, Collaborative Report DC–99–05, pp. 16–28, 1999.

[BYR92] R. A. Baeza-Yates and M. Régnier. Average running time of the Boyer-Moore-Horspool algorithm. *Theor. Comput. Sci.*, 92(1):19–31, 1992.

[CF03] D. Cantone and S. Faro. Fast-Search: a new variant of the Boyer-Moore string matching algorithm. In K. Jansen et al. (Eds.), *Proc. of WEA 2003*, LNCS 2647, pp. 47–58, 2003.

[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.

[GO80] L. J. Guibas and A. M. Odiyzko. A new proof of the linearity of the Boyer-Moore string searching algorithm. *SIAM J. Comput.*, 9(4):672–682, 1980.

[Hor80] R. N. Horspool. Practical fast searching in strings. *Softw. Pract. Exp.*, 10(6):501–506, 1980.

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

[KMP77] D. E. Knuth, J. H. Morris, and V. B. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:323-350, 1977.

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

[Rit80] W. Rytter. A correct preprocessing algorithm for Boyer-Moore string searching. *SIAM J. Comput.*, 9:509-512, 1980.

[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.

Approximate Seeds of Strings

Manolis Christodoulakis[1] and Costas S. Iliopoulos[1] and
Kunsoo Park[2*] and Jeong Seop Sim[3]

[1] Department of Computer Science,
King's College London
e-mail: {manolis, csi}@dcs.kcl.ac.uk

[2] School of Computer Science and Engineering,
Seoul National University
e-mail: kpark@theory.snu.ac.kr

[3] Electronics and Telecommunications Research Institute
Daejeon 305-350, Korea
e-mail: simjs@etri.re.kr

Abstract. In this paper we study approximate seeds of strings, that is, substrings of a given string x that cover (by concatenations or overlaps) a superstring of x, under a variety of *distance* rules (the Hamming distance, the edit distance, and the weighted edit distance). We solve the *smallest distance approximate seed* problem and the *restricted smallest approximate seed* problem in polynomial time and we prove that the general *smallest approximate seed* problem is NP-complete.

Keywords: regularities, seeds, approximate seeds, Hamming distance, edit distance, weighted edit distance, penalty matrix.

1 Introduction

Finding *regularities* in strings is useful in a wide area of applications which involve string manipulations. Molecular biology, data compression and computer-assisted music analysis are classic examples. By regularities we mean repeated strings of an approximate nature. Examples of regularities include repetitions, periods, covers and seeds. Regularities in strings have been studied widely the last 20 years.

There are several $O(n \log n)$-time algorithms [11, 6, 27] for finding *repetitions*, that is, equal adjacent substrings, in a string x, where n is the length of x. Apostolico and Breslauer [2] gave an optimal $O(\log \log n)$-time parallel algorithm (i.e., total work is $O(n \log n)$) for finding all the repetitions.

The preprocessing of the Knuth-Morris-Pratt algorithm [22] finds all periods of x in linear time— in fact, all periods of every prefix of x. Apostolico, Breslauer and Galil [3] derived an optimal $O(\log \log n)$-time parallel algorithm for finding all periods.

*Work supported by IMT 2000 Project AB02, MOST grant M1-0309-06-0003, and Royal Society grant.

The fact that in practise it was often desirable to relax the meaning of "repetition", has led more recently to the study of a collection of related patterns—"covers" and "seeds". Covers are similar to periods, but now overlaps, as well as concatenations, are allowed. The notion of covers was introduced by Apostolico, Farach and Iliopoulos in [5], where a linear-time algorithm to test superprimitivity, was given (see also [8, 9, 18]). Moore and Smyth [29] and recently Li and Smyth [25] gave linear time-time algorithms for finding all covers of a string x. In parallel computation, Iliopoulos and Park [19] obtained an optimal $O(\log\log n)$ time algorithm for finding all covers of x. Apostolico and Ehrenfeucht [4] and Iliopoulos and Mouchard [17] considered the problem of finding maximal quasiperiodic substrings of x. A two-dimensional variant of the covering problem was studied in [12, 15], and a minimum covering by substrings of a given length in [20].

An extension of the notion of covers, is that of *seeds*; that is, covers of a superstring of x. The notion of seeds was introduced by Iliopoulos, Moore and Park [16] and an $O(n\log n)$-time algorithm was given for computing all seeds of x. A parallel algorithm for finding all seeds was presented by Berkman, Iliopoulos and Park [7], that requires $O(\log n)$ time and $O(n\log n)$ work.

In applications such as molecular biology and computer-assisted music analysis, finding exact repetitions is not always sufficient. A more appropriate notion is that of *approximate* repetitions ([10, 13]); that is, finding strings that are "similar" to a given pattern, by allowing errors. In this paper, we consider three different kinds of "similarity" (approximation): the *Hamming distance*, the *edit disctance* [1, 35] and a generalization of the edit distance, the *weighted edit distance*, where different costs are assigned to each substitution, insertion and deletion for each pair of symbols.

Approximate repetitions have been studied by Landau and Schmidt [24], who derived an $O(kn\log k\log n)$-time algorithm for finding approximate squares whose edit distance is at most k in a text of length n. Schmidt also gave an $O(n^2\log n)$ algorithm for finding approximate tandem or nontandem repeats in [31] which uses an arbitrary score for similarity of repeated strings. More recently, Sim, Iliopoulos, Park and Smyth provided polynomial time algorithms for finding approximate periods [33] and, Sim, Park, Kim and Lee solved the approximate covers problem in [34].

In this paper, we introduce the notion of approximate seeds, an approximate version of seeds. We solve the *smallest distance approximate seed* problem and the *restricted smallest approximate seed* problem and we prove that the more general *smallest approximate seed* problem is NP-complete.

The paper is organized as follows. In section 2, we present some basic definitions. In section 3, we describe the notion of approximate seeds and we define the three problems studied in this paper. In section 4, we present the algorithms that solve the first two problems and the proof that the third problem is NP-complete. Section 5 contains our conclusion.

2 Preliminaries

A *string* is a sequence of zero or more symbols from an alphabet Σ. The set of all strings over Σ is denoted by Σ^*. The length of a string x is denoted by $|x|$. The *empty string*, the string of length zero, is denoted by ε. The i-th symbol of a string x is denoted by $x[i]$.

A string w is a *substring* of x if $x = uwv$, where $u, v \in \Sigma^*$. We denote by $x[i..j]$ the substring of x that starts at position i and ends at position j. Conversely, x is called a *superstring* of w. A string w is a *prefix* of x if $x = wy$, for $y \in \Sigma^*$. Similarly, w is a *suffix* of x if $x = yw$, for $w \in \Sigma^*$. We call a string w a *subsequence* (also called a subword [14]) of x (or x is a *supersequence* of w) if w is obtained by deleting zero or more symbols at any positions from x. For example, ace is a subsequence of $aabcdef$. For a given set S of strings, a string w is called a *common supersequence* of S if s is a supersequence of every string in S.

The string xy is a *concatenation* of the strings x and y. The concatenation of k copies of x is denoted by x^k. For two strings $x = x[1..n]$ and $y = y[1..m]$ such that $x[n - i + 1..n] = y[1..i]$ for some $i \geq 1$ (that is, such that x has a suffix equal to a prefix of y), the string $x[1..n]y[i + 1..m]$ is said to be a *superposition* of x and y. Alternatively, we may say that x *overlaps* with y.

A substring y of x is called a *repetition* in x, if $x = uy^kv$, where u, y, v are substrings of x and $k \geq 2$, $|y| \neq 0$. For example, if $x = aababab$, then a (appearing in positions 1 and 2) and ab (appearing in positions 2, 4 and 6) are repetitions in x; in particular $a^2 = aa$ is called a *square* and $(ab)^3 = ababab$ is called a *cube*.

A substring w is called a *period* of a string x, if x can be written as $x = w^kw'$ where $k \geq 1$ and w' is a prefix of w. The shortest period of x is called *the period* of x. For example, if $x = abcabcab$, then abc, $abcabc$ and the string x itself are periods of x, while abc is *the* period of x.

A substring w of x is called a *cover* of x, if x can be constructed by concatenating or overlapping copies of w. We also say that w *covers* x. For example, if $x = ababaaba$, then aba and x are covers of x. If x has a cover $w \neq x$, x is said to be *quasiperiodic*; otherwise, x is *superprimitive*.

A substring w of x is called a *seed* of x, if w covers one superstring of x (this can be any superstring of x, including x itself). For example, aba and $ababa$ are some seeds of $x = ababaab$.

We call the *distance* $\delta(x, y)$ between two strings x and y, the minimum cost to transform one string x to the other string y. There are several well known distance functions, described in the next paragraph. The special symbol Δ is used to represent the absence of a character.

2.1 Distance functions

The *edit distance* between two strings is the minimum number of *edit operations* that transform one string into another. The edit operations are the *insertion* of an extraneous symbol (e.g., $\Delta \to a$), the *deletion* of a symbol (e.g., $a \to \Delta$) and the *substitution* of a symbol by another symbol (e.g., $a \to b$). Note that in the edit distance model we only count the *number* of edit operations, considering the cost of each operation equal to 1.

The *Hamming distance* between two strings is the minimum number of *substitutions* (e.g., $a \to b$) that transform one string to the other. Note that the Hamming distance can be defined only when the two strings have the same length, because it does not allow insertions and deletions.

We also consider a generalized version of the edit distance model, the *weighted edit distance*, where the edit operations no longer have the same costs. It makes use

$$
\begin{array}{cccccc}
a & b & c & a & e & \Delta \\
| & | & & & | & \\
a & b & \Delta & d & e & g
\end{array}
$$

Figure 1: Alignment example

of a *penalty matrix*, a matrix that specifies the cost of each substitution for each pair of symbols, and the insertion and deletion cost for each character. A penalty matrix is a metric when it satisfies the following conditions for all $a, b, c \in \Sigma \cup \{\Delta\}$:

- $\delta(a, b) \geq 0$,
- $\delta(a, b) = \delta(b, a)$,
- $\delta(a, a) = 0$, and
- $\delta(a, c) \leq \delta(a, b) + \delta(b, c)$ (triangle inequality).

The similarity between two strings can be seen by using an *alignment*; that is, any pairing of symbols subject to the restriction that if lines were drawn between paired symbols, as in Figure 1, the lines would not cross. The equality of the lengths can be obtained by inserting or deleting zero or more symbols. In our example, the string "abcae" is transformed to "abdeg" by deleting, substituting and inserting a character at positions 3, 4 and 6, respectively. Note that this is not the only possible alignment between the two strings.

We say that a distance function $\delta(x, y)$ is a *relative distance function* if the lengths of strings x and y are considered in the value of $\delta(x, y)$; otherwise it is an *absolute distance function*. The Hamming distance and the edit distance are examples of absolute distance functions. There are two ways to define a relative distance between x and y:

- First, we can fix one of the two strings and define a relative distance function with respect to the fixed string. The *error ratio with respect to* x is defined to be $d/|x|$, where d is an absolute distance between x and y.
- Second, we can define a relative distance function symmetrically. The *symmetric error ratio* is defined to be d/l, where d is an absolute distance between x and y, and $l = (|x| + |y|)/2$ [32]. Note that we may take $l = |x| + |y|$, in which case everything is the same except that the ratio is multiplied by 2.

If d is the edit distance between x and y, the error ratio with respect to x or the symmetric error ratio is called a *relative edit distance*. The weighted edit distance can also be used as a relative distance function because the penalty matrix can contain arbitrary costs.

3 Problem Definitions

Definition 1 Let x and s be strings over Σ^*, δ be a distance function and t be a number. We call s a t-approximate seed of x if and only if there exist strings $s_1, s_2, \ldots, s_r$ ($s_i \neq \varepsilon$) such that

(i) $\delta(s, s_i) \leq t$, for $1 \leq i \leq r$, and

(ii) there exists a superstring $y = uxv$, $|u| < |s|$ and $|v| < |s|$, of x that can be constructed by overlapping or concatenating copies of the strings $s_1, s_2, \ldots, s_r$.

Each s_i, $1 \leq i \leq r$, will be called a seed block of x.

Note that y can be *any* superstring of x, including x itself (in which case, s is an approximate cover). Note, also, that there can be several versions of approximate seeds according to the definition of distance function δ.

An example of an approximate seed is shown in Figure 2. For strings $x = BABACCB$ and $s = ABAB$, s is an approximate seed of x with error 1 (hamming distance), because there exist the strings $s_1 = ABAB, s_2 = ABAC, s_3 = CBAB$, such that the distance between s and each s_i is no more than 1, and by concatenating or overlapping the strings s_1, s_2, s_3 we construct a superstring of x, $y = ABABACCBAB$.

$A\ B\ A\ B\ A\ C\ C\ B\ A\ B$

$s_1 \quad s_2 \quad s_3$

Figure 2: Approximate Seed example.

We consider the following three problems related to approximate seeds.

Problem 1 SMALLEST DISTANCE APPROXIMATE SEED *Let x be a string of length n, s be a string of length m, and δ be a distance function. Find the minimum number t such that s is a t-approximate seed of x.*

In this problem, the string s is given a priori. Thus, it makes no difference whether δ is an absolute distance function or an error ratio with respect to s. If a threshold $k \leq |s|$ on the edit distance is given as input to Problem 1, the problem asks whether s is a k-approximate seed of x or not (the *k-approximate seed* problem). Note that if the edit distance is used for δ, it is trivially true that s is an $|s|$-approximate seed of x.

Problem 2 RESTRICTED SMALLEST APPROXIMATE SEED *Given a string x of length n, find a substring s of x such that: s is a t-approximate seed of x and there is no substring of x that is a k-approximate seed of x for all $k < t$.*

Since any substring of x can be a candidate for s, the length of s is not (a priori) fixed in this problem. Therefore, we need to use a relative distance function (i.e., an error ratio or a weighted edit distance) rather than an absolute distance function. For example, if the absolute edit distance is used, every substring of x of length 1 is a 1-approximate seed of x. Moreover, we assume that s is of length at most $|x|/2$, because, otherwise the longest proper prefix of x (or any long prefix of x) can easily become an approximate seed of x with a small distance. This assumption will be applied to Problem 3, too.

Problem 3 SMALLEST APPROXIMATE SEED *Given a string x of length n, find a string s such that: s is a t-approximate seed of x and there is no substring of x that is a k-approximate seed of x for all $k < t$.*

Problem 3 is a generalization of Problem 2; s can now be any string, not necessarily a substring of x. Obviously, this problem is harder than the previous one; we will prove that it is NP-complete.

4 Algorithms and NP-Completeness

4.1 Problem 1

Our algorithm for Problem 1 consists of two steps. Let $n = |x|$ and $m = |s|$.

1. *Compute the distance between s and every substring of x.*

   We denote by w_{ij} the distance between s and $x[i..j]$, for $1 < i \leq j < n$. Note that, by definition of approximate seeds, $x[i..n]$ can be matched to any prefix of s, and $x[1..j]$ can be matched to any suffix of s (because s has to cover *any* superstring of x). Thus, we denote w_{in} the minimum value of the distances between all prefixes of s and $x[i..n]$, and w_{1j} the minimum value of the distances between all suffixes of s and $x[1..j]$.

2. *Compute the minimum t such that s is a t-approximate seed of x.*

   We use dynamic programming to compute t as follows. Let t_i be the minimum value such that s is a t_i-approximate seed of $x[1..i]$. Let $t_0 = 0$. For $i = 1$ to n, we compute t_i by the following formula:

   $$t_i = \min_{0 \leq h < i} \{\max\{\min_{h \leq j < i} \{t_j\}, w_{h+1,i}\}\} \tag{1}$$

   The value t_n is the minimum t such that s is a t-approximate seed of x.

To compute the distance between two strings, x and y, in step 1, a dynamic programming table, called the *D table*, of size $(|x| + 1) \times (|y| + 1)$, is used. Each entry $D[i, j]$, $0 \leq i \leq |x|$ and $0 \leq j \leq |y|$, stores the minimum cost of transforming $x[1..i]$ to $y[1..j]$. Initially, $D[0, 0] = 0$, $D[i, 0] = D[i - 1, 0] + \delta(x[i], \Delta)$ and $D[0, j] = D[0, j - 1] + \delta(\Delta, y[j])$. Then we can compute all the entries of the D table in $O(|x||y|)$ time by the following recurrence:

$$D[i, j] = \min \begin{cases} D[i-1, j] & + \quad \delta(x[i], \Delta) \\ D[i, j-1] & + \quad \delta(\Delta, y[j]) \\ D[i-1, j-1] & + \quad \delta(x[i], y[j]) \end{cases}$$

where $\delta(a, b)$ is the cost of substituting character a with character b, $\delta(a, \Delta)$ is the cost of deleting a and $\delta(\Delta, a)$ is the cost of inserting a.

The second step of the algorithm is computed as shown in Figure 3. For every h, we cover $x[h + 1..i]$ with one copy of s, with error $w_{h+1,i}$. What is left to be covered is $x[1..h]$. We obtain this by covering either $x[1..h]$, with error $t[h]$, or $x[1..h + 1]$, with error $t[h + 1]$, ... or $x[1..i - 1]$, with error $t[i - 1]$, (in general $x[1..j]$, with error $t[j]$); we choose the $x[1..j]$ (the shaded box) that gives the smallest error. Note that, this box covers a superstring of $x[1..j]$.

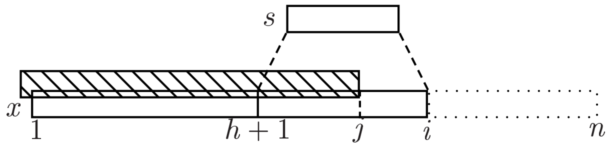


Figure 3: The second step of the algorithm.

Theorem 1 Problem 1 can be solved in $O(mn^2)$ time when a weighted edit distance is used for δ. If the edit or the Hamming distance is used for δ, it can be solved in $O(mn)$ time.

Proof. For an arbitrary penalty matrix, step 1 takes $O(mn^2)$ time, since we make a D table of size $(m+1)\times(n-i+2)$ for each position i of x. The fact that a *superstring* of x, rather than x itself, has to be "covered" does not increase the time complexity, if we use the following procedure: instead of computing a new D-table between each $s[1..k]$ (resp. $s[k..m]$) and $x[i..n]$ (resp. $x[1..j]$), we just make one D-table between s and $x[i..n]$ (resp. s^R $(x[1..j])^R$) and take the minimum value of the last column of this table.

In step 2, we can compute the minimum t in $O(n^2)$ time as follows. The inner *min* loop of formula (1) can be computed in constant time by reusing the *min* values computed in the previous round. The outer *min* loop is repeated i times, for $1 \le i \le n$, i.e., $O(n^2)$ repetitions.

Thus, the total time complexity is $O(mn^2)$.

When the edit distance is used for the measure of similarity, this algorithm for Problem 1 can be improved. In this case, $\delta(a,b)$ is always 1 if $a \neq b$ and $\delta(a,b)=0$ otherwise. Now it is not necessary to compute the edit distances between s and the substrings of x whose lengths are larger than $2m$ because their edit distances with s will exceed m. (It is trivially true that s is an m-approximate seed of x.) Step 1 now takes $O(m^2n)$ time since we make a D table of size $(m+1)\times(2m+1)$ for each position of x. Also, step 2 can be done in $O(mn)$ time since we compare $O(m)$ values at each position of x. Thus, the time complexity is reduced to $O(m^2n)$.

However, we can do better. Step 1 can be solved in $O(mn)$ time by the algorithm due to Landau, Myers and Schmidt [23]. Given two strings x and y and a forward (resp. backward) solution for the comparison between x and y, the algorithm in [23] incrementally computes a solution for x and by (resp. yb) in $O(k)$ time, where b is an additional character and k is a threshold on the edit distance. This can be done due to the relationship between the solution for x and y and the solution for x and by. When $k=m$ (i.e., the threshold is not given) we can compute all the edit distances between s and every substring of x whose length is at most $2m$ in $O(mn)$ time using this algorithm. Recently, Kim and Park [21] gave a simpler $O(mn)$-time algorithm for the same problem. Therefore, we can solve Problem 1, in $O(mn)$ time if the edit distance is used for δ. When the threshold k is given as input for Problem 1, it can be solved in $O(kn)$ time because each step of the above algorithm takes $O(kn)$ time.

If we use the Hamming distance for δ, in step 1 we consider only the substrings of x of length m. (Recall that the Hamming distance is defined only between strings of equal length) Since there are $O(n)$ such substrings, and we need $O(m)$ time to compute the distance between each substring and s, step 1 takes $O(mn)$ time. Also, as in the case of the edit distance, step 2 can be done in $O(mn)$ time (we compare $O(m)$ values at each position of x). Thus, the overall time complexity is $O(mn)$. $\square$

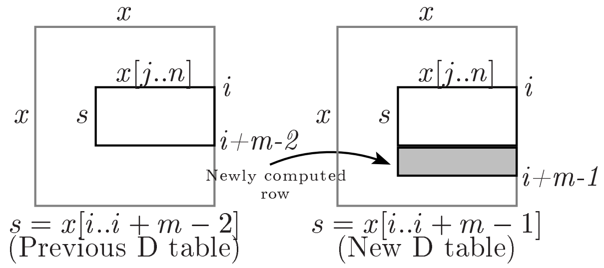


Figure 4: Computing new D tables

4.2 Problem 2

In this problem, we are not given a string s. Any substring of x is now a candidate for approximate seed. Let s be such a candidate string. Recall that, since the length of s is not fixed in this case, we need to use a relative distance function (rather than an absolute distance function); that is, an error ratio, in the case of the Hamming or edit distance, or a weighted edit distance.

When the relative edit distance is used for the measure of similarity, Problem 2 can be solved in $O(n^4)$ time by our algorithm for Problem 1. If we take each substring of x as s and apply the $O(mn)$ algorithm for Problem 1 (that uses the algorithm in [23]), it takes $O(|s|n)$ time for each s. Since there are $O(n^2)$ substrings of x, the overall time is $O(n^4)$.

For weighted edit distances (as well as for relative edit distances), we can solve Problem 2 in $O(n^4)$ time, without using the somewhat complicated algorithm in [23]. Like before, we consider every substring of x as candidate string s, and we solve Problem 1 for x and s. But, we do this, by processing all the substrings of x that start at position i, at the same time, as follows.

Let T be the minimum distance so far. Initially, $T = \infty$. For each i, $1 \le i \le n$, we process the $n - i + 1$ substrings that start at position i as candidate strings. Let m be the length of a chosen substring of x as s. Initially, $m = 1$.

1. Take $x[i..i + m - 1]$ as s and compute w_{hj}, for all $1 \le h \le j \le n$. This computation can be done by making n D tables with s and each of the n suffixes of x. By adding just one row to each of previous D tables (i.e., n D tables when $s = x[i..i + m - 2]$), we can compute these new D tables in $O(n^2)$ time. See Figure 4. (Note that when $m = 1$, we create new D tables.)

2. Compute the minimum distance t such that s is a t-approximate seed of x. This step is similar to the second step of the algorithm for Problem 1. Let t_i be the minimum value such that s is a t_i-approximate seed of $x[1..i]$ and $t_0 = 0$. For $i = 1$ to n, we compute t_i by the following formula:

$$t_i = \min_{0 \le h < i} \{\max\{\min_{h \le j < i} \{t_j\}, w_{h+1,i}\}\}$$

   The value t_n is the minimum t such that s is a t-approximate seed of x. If t_n is smaller than T, we update T with t_n. If $m < n - i + 1$, increase m by 1 and go to step 1.

When all the steps are completed, the final value of T is the minimum distance and the substring s that is a T-approximate seed of x is an answer to Problem 2.

(Note that there can be more than one substring s that are T-approximate seeds of x).

Theorem 2 Problem 2 can be solved in $O(n^4)$ time when a weighted edit distance or a relative edit distance is used for δ. When a relative Hamming distance is used for δ, Problem 2 can be solved in $O(n^3)$ time.

PROOF. For a weighted edit distance, we make n D tables in $O(n^2)$ time in step 1 and compute the minimum distance in $O(n^2)$ time in step 2. For $m = 1$ to $n - i + 1$, we repeat the two steps. Therefore, it takes $O(n^3)$ time for each i and the total time complexity of this algorithm is $O(n^4)$. If a relative edit distance is used, the algorithm can be slightly simplified, as in Problem 1, but it still takes $O(n^4)$ time.

For a relative Hamming distance, it takes $O(n)$ time for each candidate string and since there are $O(n^2)$ candidate strings, the total time complexity is $O(n^3)$. $\square$

4.3 Problem 3

Given a set of strings, the *shortest common supersequence* (SCS) problem is to find a shortest common supersequence of all strings in the set. The SCS problem is NP-complete [26, 30]. We will show that Problem 3 is NP-complete by a reduction from the SCS problem. In this section we will call Problem 3 the *SAS problem* (abbreviation of the smallest approximate seed problem). The decision versions of the SCS and SAS problems are as follows:

Definition 2 (SCS) Given a positive integer m and a finite set S of strings from Σ^* where Σ is a finite alphabet, the SCS problem is to decide if there exists a common supersequence w of S such that $|w| \leq m$.

Definition 3 (SAS) Given a number t, a string x from $(\Sigma')^*$ where Σ' is a finite alphabet, and a penalty matrix, the SAS problem is to decide if there exists a string u such that u is a t-approximate seed of x.

Theorem 3 The SAS problem is NP-complete.

5 Conclusions

In this paper, we solved the *smallest distance approximate seed* problem, in $O(mn)$ time for the Hamming and edit distance and $O(mn^2)$ for the weighted edit distance, and the *restricted smallest approximate seed* problem, in $O(n^4)$ time for the edit and weighted edit distance and $O(n^3)$ for the Hamming distance. We also proved that the *smallest approximate seed* problem is NP-complete.

The significance of our work comes from the fact that we solved the first two problems for approximate seeds, with exactly the same time complexities as those for approximate periods [33] and approximate covers [34], despite the fact that seeds allow overlaps, as well as concatenations, and cover a *superstring* of a string x (rather than covering the string x itself).

References

[1] A. Aho and T. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM J. Computing*, 1:305–312, 1972.

[2] A. Apostolico and D. Breslauer. An optimal $O(\log\log N)$-time parallel algorithm for detecting all squares in a string. *SIAM Journal on Computing*, 25(6):1318–1331, 1996.

[3] A. Apostolico, D. Breslauer, and Z. Galil. Optimal parallel algorithms for periods, palindromes and squares. *Proc. 19th Int. Colloq. Automata Languages and Programming*, 623:296–307, 1992.

[4] A. Apostolico and A. Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*, 119(2):247–265, 1993.

[5] A. Apostolico, M. Farach, and C. S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(1):17–20, 1991.

[6] A. Apostolico and F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theoretical Computer Science*, 22:297–315, 1983.

[7] O. Berkman, C. S. Iliopoulos, and K. Park. The subtree max gap problem with application to parallel string covering. *Information and Computation*, 123(1):127–137, 1995.

[8] D. Breslauer. An on-line string superprimitivity test. *Information Processing Letters*, 44(6):345–347, 1992.

[9] D. Breslauer. Testing string superprimitivity in parallel. *Information Processing Letters*, 49(5):235–241, 1994.

[10] T. Crawford, C. S. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, 11:73–100, 1998.

[11] M. Crochemore. An optimal algorithm for computing repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981.

[12] M. Crochemore, C. S. Iliopoulos, and M. Korda. Two-dimensional prefix string matching and covering on square matrices. *Algorithmica*, 20:353–373, 1998.

[13] M. Crochemore, C. S. Iliopoulos, and H. Yu. Algorithms for computing evolutionary chains in molecular and musical sequences. In *Proc. 9th Australasian Workshop on Combinatorial Algorithms*, pages 172–185, 1998.

[14] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[15] C. S. Iliopoulos and M. Korda. Optimal parallel superprimitivity testing on square arrays. *Parallel Processing Letters*, 6(3):299–308, 1996.

[16] C. S. Iliopoulos, D. Moore, and K. Park. Covering a string. *Algorithmica*, 16:288–297, 1996.

[17] C. S. Iliopoulos and L. Mouchard. An $O(n \log n)$ algorithm for computing all maximal quasiperiodicities in strings. In *Proc. Computing: Australasian Theory Symposium*, pages 262–272. Lecture Notes in Computer Science, 1999.

[18] C. S. Iliopoulos and K. Park. An optimal $O(\log \log n)$-time algorithm for parallel superprimitivity testing. *J. Korea Inform. Sci. Soc.*, 21:1400–1404, 1994.

[19] C. S. Iliopoulos and K. Park. A work-time optimal algorithm for computing all string covers. *Theoretical Computer Science*, 164:299–310, 1996.

[20] C. S. Iliopoulos and W. F. Smyth. On-line algorithms for k-covering. In *Proceedings of the 9th Australasian Workshop On Combinatorial Algorithms*, pages 97–106, Perth, WA, Australia, 1998.

[21] S. Kim and K. Park. A dynamic edit distance table. In *Proc. 11th Symp. Combinatorial Pattern Matching*, volume 1848, pages 60–68. Springer, Berlin, 2000.

[22] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern mathcing in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.

[23] G. M. Landau, E. W. Myers, and J. P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998.

[24] G. M. Landau and J. P. Schmidt. An algorithm for approximate tandem repeats. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching*, number 684, pages 120–133, Padova, Italy, 1993. Springer-Verlag, Berlin.

[25] Y. Li and W. F. Smyth. An optimal on-line algorithm to compute all the covers of a string.

[26] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336, 1978.

[27] M. G. Main and R. J. Lorentz. An algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5:422–532, 1984.

[28] M. Middendorf. More on the complexity of common superstring and supersequence problems. *Theoretical Computer Science*, 125(2):205–228, 1994.

[29] D. Moore and W. F. Smyth. A correction to "An optimal algorithm to compute all the covers of a string". *Information Processing Letters*, 54(2):101–103, 1995.

[30] K. J. Räihä and E. Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science*, 16:187–198, 1981.

[31] J. P. Schmidt. All highest scoring paths in weighted grid graphs and its application to finding all approximate repeats in strings. *SIAM Journal on Computing*, 27(4):972–992, 1998.

[32] P. H. Sellers. Pattern recognition genetic sequences by mismatch density. *Bulletin of Mathematical Biology*, 46(4):501–514, 1984.

[33] J. S. Sim, C. S. Iliopoulos, K. Park, and W. F. Smyth. Approximate periods of strings. *Theoretical Computer Science*, 262:557–568, 2001.

[34] J. S. Sim, K. Park, S. Kim, and J. Lee. Finding approximate covers of strings. *Journal of Korea Information Science Society*, 29(1):16–21, 2002.

[35] R. Wagner and M. Fisher. The string-to-string correction problem. *Journal of the ACM*, 21:168–173, 1974.

Constructing Factor Oracles

Loek Cleophas[1] and Gerard Zwaan[1] and Bruce W. Watson[1,2]

[1] Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

[2] Department of Computer Science, University of Pretoria, Pretoria 0002, South Africa

e-mail: `loek@loekcleophas.com, g.zwaan@tue.nl, bruce@bruce-watson.com`

Abstract. A *factor oracle* is a data structure for weak factor recognition. It is an automaton built on a string p of length m that is acyclic, recognizes at least all factors of p, has $m+1$ states which are all final, and has m to $2m-1$ transitions. In this paper, we give two alternative algorithms for its construction and prove the constructed automata to be equivalent to the automata constructed by the algorithms in [1]. Although these new $\mathcal{O}(m^2)$ algorithms are practically inefficient compared to the $\mathcal{O}(m)$ algorithm given in [1], they give more insight into factor oracles. Our first algorithm constructs a factor oracle based on the suffixes of p in a way that is more intuitive. Some of the crucial properties of factor oracles, which in [1] need several lemmas to be proven, are immediately obvious. Another important property however becomes less obvious. A second algorithm gives a clear insight in the relationship between the trie or dawg recognizing the factors of p and the factor oracle recognizing a superset thereof. We conjecture that an $\mathcal{O}(m)$ version of this trie-based algorithm exists.

Keywords: factor oracle, finite automaton, weak factor recognition, algorithm derivation, pattern matching.

1 Introduction

A *factor oracle* is a data structure for weak factor recognition. It can be described as an automaton built on a string p of length m that (a) is acyclic, (b) recognizes at least all factors of p, (c) has $m+1$ states (which are all final), and (d) has m to $2m-1$ transitions (cf. [1]). Some example factor oracles are given in Figures 1 and 2.

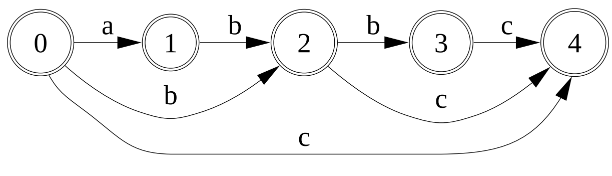


Figure 1: Factor oracle for *abbc* (recognizing $abc \notin \mathbf{fact}(p)$)

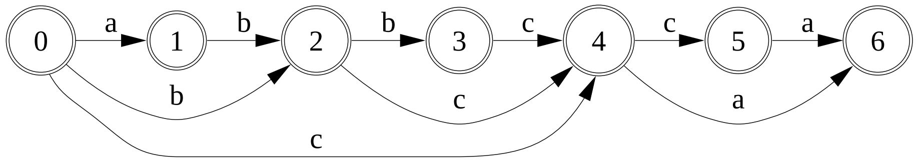


Figure 2: Factor oracle for *abbcca* (recognizing *abc*, *abcc*, *abcca*, *abca*, *abbca*, *bbca*, *bca* $\notin$ **fact**(p))

Factor oracles are introduced in [1] as an alternative to the use of exact factor recognition in many on-line keyword pattern matching algorithms. In such algorithms, a window on a text is read backward while attempting to match a keyword factor. When this fails, the window is shifted using the information on the longest factor matched and the mismatching character.

Instead of an automaton recognizing exactly the set of factors of the keyword, it is possible to use a factor oracle: although it recognizes more strings than just the factors and thus might read backwards longer than necessary, it cannot miss any matches. The advantage of using factor oracles is that they are easier to construct and take less space to represent compared to the automata that were previously used in these factor-based algorithms, such as suffix, factor and subsequence automata. This is the result of the latter automata lacking one or more of the four essential properties of the factor oracle.

The factor oracle is introduced in [1] by means of an $\mathcal{O}(m^2)$ construction algorithm that is used as its definition. Furthermore, an $\mathcal{O}(m)$ sequential construction algorithm is described. It is not obvious by just considering the algorithms that it recognizes at least all factors of p and has m to $2m-1$ transitions (i.e. that (b) and (d) hold). For both algorithms, a number of lemmas are needed to prove this. In this paper, we give two alternative algorithms for the construction of a factor oracle.

Our first algorithm, in Section 2, constructs a factor oracle based on the suffixes of p. This algorithm is $\mathcal{O}(m^2)$ and thus not of practical interest, but it is more intuitive to understand and properties (b) and (d)—two important properties of factor oracles—are immediately obvious from the algorithm. The acyclicity of the factor oracle however—corresponding to property (a)—is not immediately obvious. Our proof of this property (part of Property 6) is rather involved, whereas the property is immediately obvious from the algorithms in [1]. We prove that the alternative construction algorithm and those given in [1] construct equivalent automata in Section 3.

In Section 4 we present our second algorithm, which constructs a factor oracle from the trie recognizing the factors of p. Although this algorithm is $\mathcal{O}(m^2)$ as well, it gives a clear insight in the relationship between the trie and dawg recognizing the factors of p and the factor oracle recognizing a superset thereof. In addition, we conjecture that an $\mathcal{O}(m)$ trie-based algorithm exists.

Finally, Section 5 gives a summary and overview of future work.

1.1 Related Work

An earlier version of this paper appears as [3, Chapter 4]. In that thesis, some properties of the language of a factor oracle are discussed as well. The thesis also discusses pattern matching algorithms—among them those using factor oracles—and the implementation of the factor oracle as part of the SPARE Time pattern matching toolkit, a revised and extended version of SPARE Parts ([9]).

As mentioned before, factor oracles were introduced in [1] as an alternative to the use of exact factor recognition in many on-line keyword pattern matching algorithms. A pattern matching algorithm using the factor oracle is described in that paper as well.

Apart from their use in pattern matching algorithms, factor oracles have been used in a heuristic to compute repeated factors of a string [6] as well as to compress text [7]. An improvement for those uses of factor oracles is introduced in [8] in the form of the *repeat oracle*.

Related to the factor oracle, the *suffix oracle*—in which only those states corresponding to a suffix of p are marked final—is introduced in [1]. In [2] the factor oracle is extended to apply to a set of strings.

1.2 Preliminaries

A *string* $p = p_1...p_m$ of length m is a sequence of characters from an alphabet V. A string u is a *factor* (resp. *prefix*, *suffix*) of a string v if $v = sut$ (resp. $v = ut$, $v = su$), for $s, t \in V^*$. We will use $\mathbf{pref}(p)$, $\mathbf{suff}(p)$ and $\mathbf{fact}(p)$ for the set of prefixes, suffixes and factors of p respectively. A prefix (resp. suffix or factor) is a *proper* prefix (resp. suffix or factor) of a string p if it does not equal p. We write $u \leq_s v$ to denote that u is a suffix of v, and $u <_s v$ to denote that u is a proper suffix of v.

2 Construction Based on Suffixes

Our first alternative algorithm for the construction of a factor oracle constructs a 'skeleton' automaton for p—recognizing $\mathbf{pref}(p)$—and then constructs a path for each of the suffixes of p in order of decreasing length, such that eventually at least $\mathbf{pref}(\mathbf{suff}(p)) = \mathbf{fact}(p)$ is recognized. If such a suffix of p is already recognized, no transition needs to be constructed. If on the other hand the complete suffix is not yet recognized there is a longest prefix of such a suffix that is recognized. A transition on the next, non-recognized symbol is then created, from the state in which this longest prefix of the suffix is recognized, to a state from which there is a path leading to state m that spells out the rest of the suffix.

Build_Oracle_2($p = p_1p_2...p_m$)

for i from 0 to m **do**
  Create a new final state i
end for
for i from 0 to $m-1$ **do**
  Create a new transition from i to $i+1$ by p_{i+1}
end for
for i from 2 to m **do**
  Let the longest path from state 0 that spells a prefix of $p_i...p_m$ end in state j and spell out $p_i...p_k$ $(i-1 \leq k \leq m)$
  if $k \neq m$ **then**

10: Build a new transition from j to $k+1$ by p_{k+1}
11: **end if**
12: **end for**

Note that this algorithm is $\mathcal{O}(m^2)$ (since the operation on line 6 can be implemented using a **while** loop). The factor oracle on p built using this algorithm is referred to as Oracle(p) and the language recognized by it as **factoracle**(p).

The first two properties we give are obvious given our algorithm. They correspond to (b) and (c)-(d) respectively as mentioned in Section 1.

Property 1 **fact**(p) $\subseteq$ **factoracle**(p).
Proof: The algorithm constructs a path for all suffixes of p and all states are final. □

Property 2 For p of length m, Oracle(p) has exactly $m+1$ states and between m and $2m-1$ transitions.
Proof: States can be constructed in steps 1-2 only, and exactly $m+1$ states are constructed there. In step 4 of the algorithm, m transitions are created. In steps 5-8, at most $m-1$ transitions are created. □

Property 3 (Glushkov's property) All transitions reaching a state i of Oracle(p) are labeled by p_i.
Proof: The only steps of the algorithm that create transitions are steps 4 and 8. In both, transitions to a state i are created labeled by p_i. □

Property 4 (Weak determinism) For each state of Oracle(p), no two outgoing transitions of the state are labeled by the same symbol.
Proof: The algorithm never creates an outgoing transition by some symbol if such a transition already exists. □

We now define function *poccur*(u,p) to give the end position of the leftmost occurrence of u in p (equivalent to the same function in [1]):

Definition 1 Function *poccur* $\in V^* \times V^* \to \mathbb{N}$ is defined as

$$\mathit{poccur}(u,p) = \min\{|tu|, p = tuv\} \quad (p,t,u,v \in V^*)$$

□

Note that if $u \notin \textbf{fact}(p)$, $\mathit{poccur}(u,p) = \infty$.

Property 5 For suffixes and prefixes of factors we have:

$$uv \in \textbf{fact}(p) \Rightarrow \mathit{poccur}(v,p) \leq \mathit{poccur}(uv,p) \quad (p,u,v \in V^*)$$
$$uv \in \textbf{fact}(p) \Rightarrow \mathit{poccur}(u,p) \leq \mathit{poccur}(uv,p) - |v| \quad (p,u,v \in V^*)$$

□

We introduce $\mathit{min}(i)$ for the minimum length string recognized in state i—either in a partially constructed or in the complete automaton.

In the following property, we use j_i and k_i to identify the values j and k attain when considering suffix $p_i ... p_m$ of p in steps 5-8 of the algorithm.

Property 6 For the partial automaton constructed according to algorithm **Build_Oracle_2** with all suffixes of p of length greater than $m - i + 1$ already considered in steps 5-8 ($2 \leq i \leq m+1$), we have that

i. it is acyclic

ii. for each h with $1 \leq h < i$, all prefixes of $p_h ... p_m$ are recognized

iii. for each state n and outgoing transition to a state $q \neq n+1$, $q \leq k_{max} + 1$ holds where $k_{max} = \max\{k_h, 1 < h < i \wedge k_h < m\}$

iv. for each state n, $min(n)$ is an element of $\mathbf{fact}(p)$, $min(n)$ is a suffix of each string recognized in n, and $n = poccur(min(n), p)$

v. if $u \in \mathbf{fact}(p)$ is recognized, it is recognized in a state $n \leq poccur(u, p)$

vi. for each state n and each symbol a such that there is a transition from n to a state q by a, $min(n) \cdot a \in \mathbf{fact}(p)$ and $q = poccur(min(n) \cdot a, p)$

vii. for each pair of states n and q, if $min(n) \leq_s min(q)$, then $n \leq q$, and as a result, if $min(n) <_s min(q)$, then $n < q$

viii. if w is recognized in state n, then for any suffix u of w, if u is recognized, it is recognized in state $q \leq n$

Proof: See Appendix A. □

Note that Property 6, i. corresponds to property (a) in Section 1.

3 Equivalence to Original Algorithms

A factor oracle as introduced in [1] is built by the following algorithm:

Build_Oracle($p = p_1 p_2 ... p_m$)

for i from 0 to m **do**
  Create a new final state i
end for
for i from 0 to $m-1$ **do**
  Create a new transition from i to $i+1$ by p_{i+1}
end for
for i from 0 to $m-1$ **do**
  Let u be a minimal length word in state i
  for all $\sigma \in \Sigma, \sigma \neq p_{i+1}$ **do**
    if $u\sigma \in \mathrm{Fact}(p_{i-|u|+1} ... p_m)$ **then**
      Build a new transition from i to[*]
        $i - |u| + poccur(u\sigma, p_{i-|u|+1} ... p_m)$ by σ
    end if
  end for

[*]Note that in [1] the term $-|u|$ is missing in the algorithm, although from the rest of the paper it is clear that it is used in the construction of the automata

14: **end for**

To prove the equivalence of the automata constructed by the two algorithms, we need the following properties.

Property 7 For any state i of both Oracle(p) (i.e. the factor oracle constructed according to algorithm **Build_Oracle_2** and the factor oracle constructed according to algorithm **Build_Oracle**), if $u = min(i)$ then

$$u\sigma \in \mathbf{fact}(p_{i-|u|+1}...p_m) \equiv u\sigma \in \mathbf{fact}(p)$$

Proof: $\Rightarrow$: Trivial. $\Leftarrow$: By Property 6, iv. (for **Build_Oracle_2**) and [1, Lemma 1] (for **Build_Oracle**), $i = poccur(u, p)$. By Property 5, $poccur(u\sigma, p) \geq i$, hence $u\sigma \in \mathbf{fact}(p_{i-|u|+1}...p_m)$. $\square$

Property 8 For any state i of an automaton constructed by either algorithm, if $u = min(i)$ and $u\sigma \in \mathbf{fact}(p)$ then

$$i - |u| + poccur(u\sigma, p_{i-|u|+1}...p_m) = poccur(u\sigma, p)$$

Proof:

$$\begin{array}{ll} & i - |u| + poccur(u\sigma, p_{i-|u|+1}...p_m) \\ = & \quad \{\text{ definition } poccur \ \} \\ & i - |u| + \min\{|tu\sigma|, p_{i-|u|+1}...p_m = tu\sigma v\} \\ = & \quad \{\ u = min(i), \text{ hence recognized in } i = poccur(u, p)\ \} \\ & i - |u| + \min\{|tu\sigma| - (i - |u|), p = tu\sigma v\} \\ = & \quad \{\ u\sigma \in \mathbf{fact}(p), \text{ property of min }\} \\ & i - |u| + \min\{|tu\sigma|, p = tu\sigma v\} - (i - |u|) \\ = & \quad \{\text{ calculus, definition } poccur \ \} \\ & poccur(u\sigma, p) \end{array}$$

$\square$

Property 9 The algorithms **Build_Oracle_2** and **Build_Oracle** construct equivalent automata.

Proof: We prove this by induction on the states. Our induction hypothesis is that for each state j $(0 \leq j < i)$, $min(j)$ is the same in both automata, and the outgoing transitions from state j are equivalent for both automata.

If $i = 0$, $u = min(i) = \varepsilon$ in both automata. Consider a transition created by **Build_Oracle_2**, say to state k by $\sigma \neq p_{i+1}$. Since this transition exists, $u\sigma \in \mathbf{fact}(p)$ and $k = poccur(u\sigma, p)$ (due to Property 6, vi.). Using Properties 7 and 8, such a transition was created by **Build_Oracle** as well. Similarly, consider a transition created by **Build_Oracle**, say to state k by σ. This transition, say on symbol σ, leads to state $k = i - |u| + poccur(u\sigma, p_{i-|u|+1}...p_m)$ and was created since $u\sigma \in \mathbf{fact}(p_{i-|u|+1}...p_m)$ (see the algorithm). Using Properties 7 and 8, such a transition was created by **Build_Oracle_2** as well.

If $i > 0$, using the induction hypothesis and acyclicity of the automata, i has the same incoming transitions and as a result $min(i)$ is the same for both automata. Using the same arguments as in case $i = 0$, the outgoing transitions from state i are equivalent for both automata.

As a result, the two automata are equivalent. □

4 Construction Based on Trie

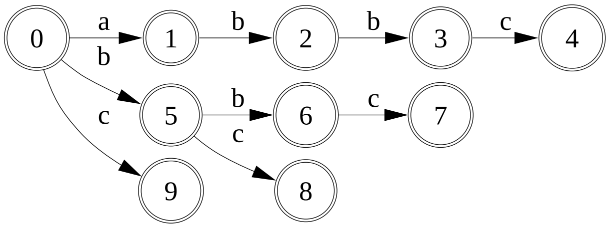


Figure 3: Trie recognizing **fact**$(abbc)$

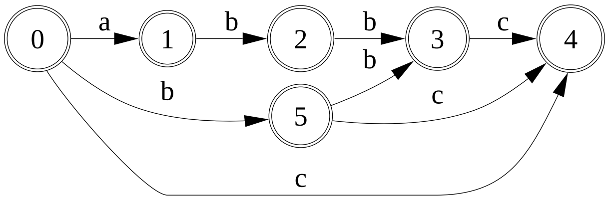


Figure 4: DAWG recognizing **fact**$(abbc)$

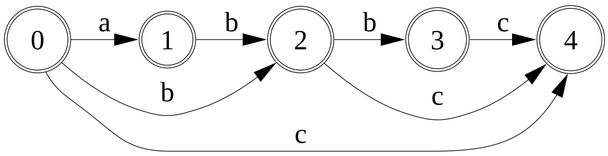


Figure 5: Factor oracle recognizing **fact**$(abbc) \cup \{abc\}$

There is a close relationship between the data structures Trie(**fact**(p)) —the *trie* ([5]) on **fact**(p)—recognizing exactly **fact**(p), DAWG(**fact**(p)) —the *directed acyclic word graph* ([4]) on **fact**(p)—recognizing exactly **fact**(p), and Oracle(p)—the factor oracle on p—which recognizes at least **fact**(p).

It is well known that DAWG(**fact**(p)) can be constructed from Trie(**fact**(p)) by merging states whose right languages are identical (see for example [4]). The factor oracle as defined by Oracle(p) can also be constructed from Trie(**fact**(p)), by merging states whose right languages have identical longest strings (which are suffixes of p). An example of a trie, DAWG and factor oracle for the factors of $abbc$ can be seen in Figures 3-5.

Definition 2 We define Trie(S) as a 5-tuple $<Q, V, \delta, \varepsilon, F>$ where S is a finite set of strings, $Q = \mathbf{pref}(S)$ is the set of states, V is the alphabet, δ is the transition function, defined by

$$\delta(u,a) = \begin{cases} ua & \text{if } ua \in \mathbf{pref}(S) \\ \perp & \text{if } ua \notin \mathbf{pref}(S) \end{cases} \qquad (u \in \mathbf{pref}(S), a \in V),$$

ε is the single start state and $F = S$ is the set of final states. □

Property 10 For $u, v \in \mathbf{fact}(p)$ we have :

$$uv \in \mathbf{fact}(p) \wedge (\forall\, w : uw \in \mathbf{fact}(p) : |w| \leq |v|) \qquad \Rightarrow uv \in \mathbf{suff}(p)$$

$$\begin{aligned} & uv_1 \in \mathbf{fact}(p) \wedge (\forall\, w : uw \in \mathbf{fact}(p) : |w| \leq |v_1|) \\ & \wedge\, uv_2 \in \mathbf{fact}(p) \wedge (\forall\, w : uw \in \mathbf{fact}(p) : |w| \leq |v_2|) \quad \Rightarrow v_1 = v_2 \end{aligned}$$

□

Property 11 For $u \in \mathbf{fact}(p)$ and $C \in \mathbb{N}$,

$$(\forall\, w : uw \in \mathbf{fact}(p) : |w| \leq C) \equiv (\forall\, w : uw \in \mathbf{suff}(p) : |w| \leq C)$$

Proof: $\Rightarrow$: trivial. $\Leftarrow$: Let $ux \in \mathbf{fact}(p)$, then $(\exists y : : uxy \in \mathbf{suff}(p))$, hence $(\exists y : : |xy| \leq C)$, and since $|y| \geq 0$, $|x| \leq C$. □

Using Properties 10 and 11, $max_p(u)$ can be defined as the unique longest string v such that $uv \in \mathbf{suff}(p)$:

Definition 3 Define $max_p(u) = v$ where v is such that

$$uv \in \mathbf{suff}(p) \wedge (\forall\, w : uw \in \mathbf{suff}(p) : |w| \leq |v|)$$

□

We now present our simple trie-based construction algorithm for factor oracles:

Trie_To_Oracle($p = p_1 p_2 ... p_m$)
Construct **Trie**(**fact**(p))
for i from 2 to m **do**
    Merge all states u for which $max_p(u) = p_{i+1}...p_m$ into the single state $p_1...p_i$
end for

The order in which the values of i are considered is not important. In addition, note that it is not necessary to consider the states u for which $max_p(u) = p_2...p_m$ since there is precisely one such state u in Trie(**fact**(p)), $u = p_1$. Due to Property 10, it is sufficient to only consider suffixes of p as longest strings.

Also note that the intermediate automata may be nondeterministic, but the final automaton will be weakly deterministic (as per Property 4).

The above algorithm has complexity $\mathcal{O}(m^2)$ (assuming that $max_p(u)$ was computed during construction of the trie). The construction of a Trie can be done in $\mathcal{O}(m)$ time however, and the merging of the states is similar to minimization of an acyclic automaton, which can also be done in $\mathcal{O}(m)$. We therefore conjecture that an $\mathcal{O}(m)$ trie-based factor oracle construction algorithm exists.

To prove that algorithm **Trie_To_Oracle** constructs Oracle(p), we define a partition on the states of the trie, induced by an equivalence relation on the states.

Definition 4 Relation $\sim_p$ on states of Trie($\mathbf{fact}(p)$) is defined by

$$t \sim_p u \equiv max_p(t) = max_p(u) \quad (t, u \in \mathbf{fact}(p))$$

Note that relation $\sim_p$ is an equivalence relation. □

We now show that the partitioning into sets of states of Trie($\mathbf{fact}(p)$) induced by $\sim_p$, is the same as the partitioning of Trie($\mathbf{fact}(pa)$) induced by $\sim_{pa}$, restricted to the states of Trie($\mathbf{fact}(p)$), i.e.

Property 12

$$t \sim_p u \equiv t \sim_{pa} u \quad (t, u \in \mathbf{fact}(p), a \in V)$$

Proof:

$$\begin{array}{ll}
& t \sim_p u \\
\equiv & \quad \{ \text{ definition } \sim_p \ \} \\
& max_p(t) = max_p(u) \\
\equiv & \quad \{ \ \} \\
& max_p(t)a = max_p(u)a \\
\equiv & \quad \{ \ (\star) \ \} \\
& max_{pa}(t) = max_{pa}(u) \\
\equiv & \quad \{ \text{ definition } \sim_{pa} \ \} \\
& t \sim_{pa} u
\end{array}$$

where we prove ($\star$) by

$$\begin{array}{ll}
& v = max_{pa}(u) \\
\equiv & \quad \{ \text{ definition } max_{pa} \ \} \\
& uv \in \mathbf{suff}(pa) \wedge (\forall\, w : uw \in \mathbf{suff}(pa) : |w| \leq |v|) \\
\equiv & \quad \{ \ u \in \mathbf{fact}(p), \text{ hence } (\exists x : : uxa \in \mathbf{suff}(pa)), \\
& \quad\quad \text{hence } |xa| > 0 \text{ and } |v| > 0; \mathbf{suff}(pa) = \mathbf{suff}(p)a \cup \{\varepsilon\} \ \} \\
& uv \in \mathbf{suff}(p)a \wedge (\forall\, w : uw \in \mathbf{suff}(pa) : |w| \leq |v|) \\
\equiv & \quad \{ \ |v| > 0 \ \} \\
& uv \in \mathbf{suff}(p)a \wedge (\forall\, w : w \neq \varepsilon \wedge uw \in \mathbf{suff}(pa) : |w| \leq |v|) \wedge v = v'a \\
\equiv & \quad \{ \ \mathbf{suff}(pa) = \mathbf{suff}(p)a \cup \{\varepsilon\} \ \} \\
& uv \in \mathbf{suff}(p)a \wedge (\forall\, w : w \neq \varepsilon \wedge uw \in \mathbf{suff}(p)a : |w| \leq |v|) \wedge v = v'a
\end{array}$$

$$\begin{array}{ll}
\equiv & \{\ w = w'a\ \} \\
& uv \in \mathbf{suff}(p)a \wedge (\forall\, w' : uw'a \in \mathbf{suff}(p)a : |w'a| \leq |v'a|) \wedge v = v'a \\
\equiv & \{\ \} \\
& uv \in \mathbf{suff}(p)a \wedge (\forall\, w' : uw' \in \mathbf{suff}(p) : |w'| \leq |v'|) \wedge v = v'a \\
\equiv & \{\ v = v'a\ \} \\
& uv' \in \mathbf{suff}(p) \wedge (\forall\, w' : uw' \in \mathbf{suff}(p) : |w'| \leq |v'|) \wedge v = v'a \\
\equiv & \{\ \text{definition } max_p\ \} \\
& v' = max_p(u) \wedge v = v'a \\
\equiv & \{\ \} \\
& v = max_p(u)a
\end{array}$$

□

Property 13 Algorithm **Trie_To_Oracle** constructs Oracle(p).
Proof: By induction on $|p| = m$. If $m = 0$, $p = \varepsilon$, and Trie($\mathbf{fact}(\varepsilon)$) = Oracle($\varepsilon$). If $m = 1$, $p = a$ ($a \in V$), and Trie($\mathbf{fact}(a)$)=Oracle(a). If $m > 1$, $p = xa$ ($x \in V^*, a \in V$), and we may assume the algorithm to construct part Oracle(x) of Oracle(xa) correctly (using $\mathbf{fact}(ua) = \mathbf{fact}(u) \cup \mathbf{suff}(u)a$, Trie($\mathbf{fact}(xa)$) being an extension of Trie($\mathbf{fact}(x)$), and Oracle(xa) being an extension of Oracle(x) (which is straightforward to see from algorithm **Build_Oracle_2** as well as [1, page 57, after Corollary 4]), and Property 12). Now consider the states of this partially converted automaton in which suffixes of x are recognized. By construction of the trie, there are transitions from these states by a. The factor oracle construction according to algorithm **Oracle_Sequential** in [1] creates Oracle(xa) from Oracle(x)+a (i.e. the factor oracle for x extended with a single new state m reachable from state $m-1$ by symbol $p_m = a$) by creating new transitions to state m from those states in which suffixes of x are recognized and that do not yet have a transition on a. Since **Trie_To_Oracle** merges all states t for which $max_{xa}(t) = a$ into the single state m, Oracle(xa) is constructed correctly from Trie($\mathbf{fact}(xa)$). □

5 Conclusions and Future Work

We have presented two alternative construction algorithms for factor oracles and shown the automata constructed by them to be equivalent to those constructed by the algorithms in [1]. Although both our algorithms are $\mathcal{O}(m^2)$ and thus practically inefficient compared to the $\mathcal{O}(m)$ sequential algorithm given in [1], they give more insight into factor oracles.

Our first algorithm is more intuitive to understand and makes it immediately obvious, without the need for several lemmas, that the factor oracle recognizes at least $\mathbf{fact}(p)$ and has m to $2m-1$ transitions.

Our second algorithm gives a clear insight into the relationship between the trie or dawg recognizing $\mathbf{fact}(p)$ and the factor oracle recognizing a superset thereof. We conjecture that an $\mathcal{O}(m)$ trie-based algorithm for the construction of factor oracles exists.

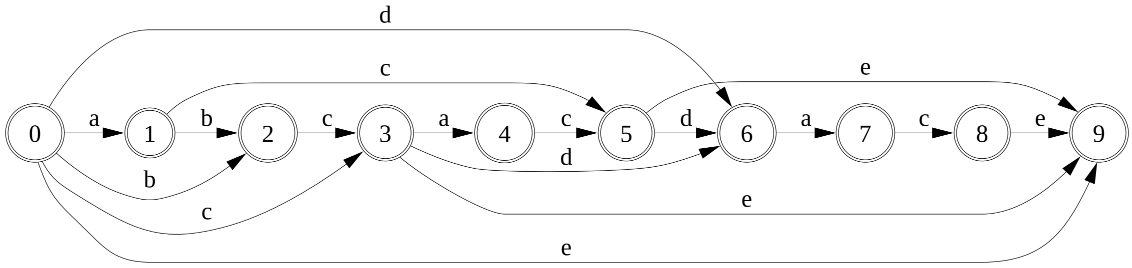


Figure 6: Factor oracle recognizing a superset of $\mathbf{fact}(p)$ (including for example $cace \notin \mathbf{fact}(p)$), for $p = abcacdace$.

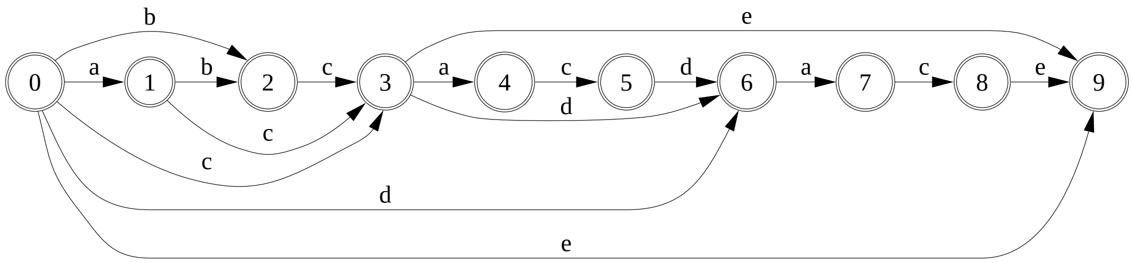


Figure 7: Alternative automaton with $m+1$ states satisfying Glushkov's property yet recognizing a *different* superset of $\mathbf{fact}(p)$ than the factor oracle for p (including for example $acacdace \notin \mathbf{factoracle}(p)$, but not $cace$) and having less transitions, for $p = abcacdace$.

As stated in [1], the factor oracle is not minimal in terms of number of transitions among the automata with $m+1$ states recognizing at least $\mathbf{fact}(p)$. We note that it is not even minimal among the subset of such automata having Glushkov's property (see Figures 6 and 7).

We are working on an automaton-independent definition of the language recognized by the factor oracle. Such a characterization would enable us to calculate how many strings are recognized that are not factors of the original string. This could be useful in determining whether to use a factor oracle-based algorithm in pattern matching or not.

Acknowledgements

We would like to thank Michiel Frishert for reading and commenting on earlier versions of this paper, and the anonymous referees for their helpful comments and suggestions.

References

[1] 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 *LNCS*, pages 51–72, 2001.

[2] Cyril Allauzen and Mathieu Raffinot. Oracle des facteurs d'un ensemble de mots. Technical Report 99-11, Institut Gaspard-Monge, Université de Marne-la-Vallée, June 1999.

[3] Loek G.W.A. Cleophas. Towards SPARE Time: A New Taxonomy and Toolkit of Keyword Pattern Matching Algorithms. MSc thesis, Technische Universiteit Eindhoven, August 2003.

[4] Maxime Crochemore and Wojciech Rytter. *Text Algorithms.* Oxford University Press, 1994.

[5] E. Fredkin. Trie memory. *Communications of the ACM*, 3(10):490–499, 1960.

[6] 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, 2000.

[7] Arnaud Lefebvre and Thierry Lecroq. Compror: on-line losless data compression with a factor oracle. *Inf. Process. Lett.*, 83(1):1–6, 2002.

[8] Arnaud Lefebvre, Thierry Lecroq, and J. Alexandre. Drastic improvements over repeats found with a factor oracle. In E. Billington, D. Donovan, and A. Khodkar, editors, *Proceedings of the 13th Australasian Workshop on Combinatorial Algorithms*, pages 253–265, 2002.

[9] Bruce W. Watson and Loek Cleophas. SPARE Parts: A C++ toolkit for String PAttern REcognition. *Software: Practice and Experience*, 2003. To be published.

A Proof of Property 6

We first consider the automaton constructed in steps 1-4 of the algorithm. It is straightforward to verify that the properties hold for $i = 2$.

Now assume that the properties hold for the automaton with all suffixes of p of length greater than $m - i + 1$ already considered. We prove that they also hold for the automaton after the suffix of length $m - i + 1$, $p_i...p_m$, has been considered.

If $k = m$ in step 6, suffix $p_i...p_m$ is already recognized, no new transition will be created, the automaton does not change and the properties still hold.

If $k < m$, then we need to prove that each of the properties holds for the new automaton.

Ad i: By v., string $p_i...p_k$ is recognized in state $j \leq poccur(p_i...p_k, p)$. Since $p_i...p_k \leq_s p_1...p_k$ and $poccur(p_1...p_k, p) = k$, $poccur(p_i...p_k, p) \leq k$ due to Property 5. Since $j \leq k$, the transition created from j to $k + 1$ is a forward one.

Ad ii: Trivial.

Ad iii: We prove that the property holds for the new automaton by showing that $k = k_i \geq k_{max}$, i.e. k will become the new k_{max}.

If $k_{max} = -\infty$, $k \geq k_{max}$ clearly holds.

If $k_{max} > -\infty$, assume that $k_{max} > k$, then there is an h such that $1 < h < i \wedge k_h < m \wedge k_h = k_{max}$. Factor $p_h...p_k$ is recognized in $g \leq k$ due to ii. and v.

If $g = k$, then $p_h...p_k$ is recognized in k and $p_h...p_m$ is recognized in m; so $k_h = m$ which contradicts $k_h < m$.

If $g < k$, then $p_h...p_k$ is recognized in $g < k$. Since $p_i...p_k$ is recognized in $j = j_i$ and $p_i...p_k \leq_s p_h...p_k$, due to viii., $j \leq g$.

If $j = g$, then $p_h...p_k$ is the longest prefix of $p_h...p_m$ recognized by the old automaton, which contradicts ii.

If $j < g$, then $j < g < k$. We know that $min(g) \leq_s p_h...p_k$ (using iv.), $min(j) \leq_s p_h...p_k$ (using iv. and $p_i...p_k \leq_s p_h...p_k$) and therefore that $min(j) <_s min(g)$ (due to vii.). Let l be the state to which the transition by p_{k+1} from g leads, i.e. l is the state in which $p_h...p_{k+1}$ is recognized. Using vi., we have that $l = poccur(min(g) \cdot p_{k+1}, p)$. Using Property 5 we have that $l \leq poccur(p_h...p_{k+1}, p)$ and the latter is $\leq k+1$ due to the definition of $poccur$ (since $k+1$ marks the end of an occurrence of $p_h...p_{k+1}$). We have $poccur(min(j) \cdot p_{k+1}, p) \leq poccur(min(g) \cdot p_{k+1}, p) = l$ since $min(j) \leq_s min(g)$. We want to prove that $k+1 \leq poccur(min(j) \cdot p_{k+1}, p)$. Assume that $poccur(min(j) \cdot p_{k+1}, p) < k+1$. If the first occurrence of $min(j) \cdot p_{k+1}$ starts before position i of p, then it is a prefix of a suffix of p longer than $p_i...p_m$ and thus by ii. $min(j) \cdot p_{k+1}$ is recognized. Since $min(j)$ is recognized in j, a transition from j by p_{k+1} must exist and we have a contradiction. If the first occurrence of $min(j) \cdot p_{k+1}$ starts at or after position i of p, then there exists a shortest string x such that $x \cdot min(j) \cdot p_{k+1} \in \mathbf{pref}(p_i...p_k)$ and $x \cdot min(j) \cdot p_{k+1}$ is recognized in a state $\leq j$. But then $x \cdot min(j)$ is recognized in a state $n < j$. By viii., since $min(j) \leq_s x \cdot min(j)$, this means that $min(j)$ is recognized in state $s \leq n < j$ and we have a contradiction. Thus $k+1 \leq poccur(min(j) \cdot p_{k+1}, p) \leq l$ and therefore, since $l \leq k+1$ holds, $l = k+1$. In that case, $p_h...p_{k+1}$ is recognized in $l = k+1$ and $p_h...p_m$ is recognized in m. But then $k_h = m$, and we have a contradiction.

Thus, $k_{max} = k_h \leq k = k_i$ and iii. holds for the new automaton.

Ad iv: Let $s = min(j)$, $t = min(k+1)$ and $u = min(h)$ $(k+1 \leq h \leq m)$ respectively in the old automaton. Due to the proof of iii., $k = k_i \geq k_{max}$ and therefore a unique path between $k+1$ and h exists, labeled r, and—due to iv—$u \leq_s tr$.

If $|sp_{k+1}r| \geq |u|$, u remains the minimal length string recognized in state h. Since $s \leq_s p_i...p_k$, $sp_{k+1}r \leq_s p_i...p_{k+1}r$. Since $u \leq_s tr$, $tr \leq_s p_1...p_{k+1}r$ and $|sp_{k+1}r| \geq |u|$, $u \leq_s sp_{k+1}r$ and—due to iv.—$u \leq_s s'p_{k+1}r$ as well for any s' recognized in state j.

If $|sp_{k+1}r| < |u|$, $sp_{k+1}r$ is the new minimal length string recognized in state h. Since $s \leq_s p_i...p_k$, $sp_{k+1}r \leq_s p_i...p_{k+1}r$. Since $u \leq_s tr$, $tr \leq_s p_1...p_{k+1}r$ and $|sp_{k+1}r| < |u|$, $sp_{k+1}r \leq_s u$ and—due to iv.—$sp_{k+1}r \leq_s s'p_{k+1}r$ as well for any s' recognized in state j.

Since $p_i...p_{k+1}r$ was not recognized before, it is not a prefix of p, $p_2...p_m$, ..., $p_{i-1}...p_m$ (using ii.), hence $poccur(p_i...p_{k+1}r, p) = k+1+|r|$. Since $s \leq_s p_i...p_k$, $poccur(sp_{k+1}r, p) \leq k+1+|r|$. Assume that $poccur(sp_{k+1}r, p) < k+1+|r|$, then $p_i...p_{k+1}r = usp_{k+1}rv$ $(u, v \in V^*, v \neq \varepsilon, |u|$ minimal), since $sp_{k+1}r$ cannot start before

p_i because in that case it would have already been recognized by the old automaton. Factor us is recognized in state $g < j$ (using i.) and—since viii. holds—$s \leq_s us$ is recognized in a state $o \leq g < j$. This contradicts s being recognized in j. As a result $poccur(sp_{k+1}r, p) = k + 1 + |r|$.

Ad v: Any new factor of p recognized after creation of the transition from j to $k+1$ has the form $vp_{k+1}r$ and is recognized in $k+1+|r|$ with $v \in \mathbf{fact}(p)$ recognized in state j. Since $k + 1 + |r| = poccur(min(k + 1)r, p)$ (using iii., iv. holding for the new automaton plus the fact that k is the new k_{max}) and $min(k+1) \cdot r \leq_s vp_{k+1}r$ due to iv. holding for the new automaton, $k+1+|r| \leq poccur(vp_{k+1}r, p)$ using Property 5.

Ad vi: The states n we have to consider are $n = j$ and $n = h$ for $k + 1 \leq h \leq m$.

For $n = j$, a new transition to $k+1$ is created and by iv., $min(j) \leq_s p_i...p_k$., hence we have $min(j) \cdot p_{k+1} \leq_s p_i...p_{k+1}$, $p_{k+1-|min(j)|}...p_{k+1} = min(j) \cdot p_{k+1}$, $min(j) \cdot p_{k+1} \in \mathbf{fact}(p)$ and $poccur(min(j) \cdot p_{k+1}, p) \leq k+1$. Since $min(j) \cdot p_{k+1}$ is recognized in state $k + 1$, due to v. for the new automaton, $k + 1 \leq poccur(min(j) \cdot p_{k+1}, p)$. Therefore $k + 1 = poccur(min(j) \cdot p_{k+1}, p)$.

For $n = h$ with $k + 1 \leq h \leq m$, $min(h)$ changes to $sp_{k+1}r$ if and only if $|sp_{k+1}r| < |u|$ (with r, s, u as in the proof of iv.). We know that $ua \in \mathbf{fact}(p)$ and $q = poccur(ua, p)$. Since $sp_{k+1}r \leq_s u$, $sp_{k+1}ra \leq_s ua$, hence $sp_{k+1}ra \in \mathbf{fact}(p)$ as well and $poccur(sp_{k+1}ra, p) \leq poccur(ua, p) = q$, but due to v., $q \leq poccur(sp_{k+1}ra, p)$ hence $q = poccur(sp_{k+1}ra, p)$.

Ad vii: Assume $min(n) \leq_s min(q)$. We have $poccur(min(n), p) \leq poccur(min(q), p)$ due to Property 5, which according to iv. is equivalent to $n \leq q$.

Ad viii: By induction on $|w|$. It is true if $|w| = 0$ or $|w| = 1$. Assume that it is true for all strings x such that $|x| < |w|$. We will show that it is also true for w, recognized in n.

Let $w = xa$ ($x \neq \varepsilon$), x is recognized in h ($0 < h < n$). Consider a proper suffix of w, recognized in state q. It either equals ε and is recognized in state $0 \leq n$ or it can be written as va where $v <_s x$.

Suffix va of w is recognized, therefore suffix v of x is recognized and according to the induction hypothesis, v is recognized in state $l \leq h$. Let $\bar{x} = min(h)$ and $\bar{v} = min(l)$. Due to iv. for the new automaton, $\bar{x} \leq_s x$ and $\bar{v} \leq_s v$. We now prove that $\bar{v} \leq_s \bar{x}$. If $l = h$, then $\bar{v} = \bar{x}$. Now consider the case $l < h$. Since $v \leq_s x$ and $\bar{v} \leq_s v$, $\bar{v} \leq_s x$. Due to vii., $\bar{x} \not\leq_s \bar{v}$. Thus, since $\bar{v}$ and $\bar{x}$ both are suffixes of x, $\bar{v} \leq_s \bar{x}$. Since $\bar{x}$ is recognized in h and there is a transition by a from h to n, by vi. for the new automaton we have that $\bar{x}a \in \mathbf{fact}(p)$ and $n = poccur(\bar{x}a, p)$. Since $\bar{v}$ is recognized in l and there is a transition by a from l to q, $\bar{v}a \in \mathbf{fact}(p)$ and $q = poccur(\bar{v}a, p)$ due to vi. for the new automaton. Since $\bar{v}a \leq_s \bar{x}a$, $poccur(\bar{v}a, p) \leq poccur(\bar{x}a, p)$ due to Property 5 and hence $q \leq n$.

We have shown that the properties hold for every partial automaton during the construction. Consequently, they hold for the complete automaton Oracle(p). $\square$

Computing the Minimum k-Cover of a String

Richard Cole [1§], Costas S. Iliopoulos [2†], Manal Mohamed [2‡],
W. F. Smyth [3¶] and Lu Yang[4]

[1] Computer Science Department, Courant Institute of Mathematical Sciences,
New York University, New York, NY 10012-1185 U.S.A.
`cole@cs.nyu.edu`

[2] Algorithm Design Group, Department of Computer Science,
King's College London, London WC2R 2LS, England
`{csi,manal}@dcs.kcl.ac.uk`

[3] Algorithms Research Group, Department of Computing & Software,
McMaster University, Hamilton ON L8S 4K1, Canada &
School of Computing, Curtin University, Perth WA 6845, Australia
`smyth@mcmaster.ca`

[4] IBM Canada Limited, 8200 Warden Avenue, Markham ON L6G 1C7, Canada
`luyang@ca.ibm.com`

Abstract. We study the minimum k-cover problem. For a given string x of length n and an integer k, the minimum k-cover is the minimum set of k-substrings that covers x. We show that the on-line algorithm that has been proposed by Iliopoulos and Smyth [IS92] is not correct. We prove that the problem is in fact NP-hard. Furthermore, we propose two greedy algorithms that are implemented and tested on different kind of data.

Keywords: string algorithm, k-cover, data compression, NP-complete, greedy algorithm.

1 Introduction

The *minimum k-cover* problem is to compute, for a given string x and an integer $k < |x|$, a set $U = \{u_1, u_2, \ldots, u_m\}$ of substrings of x such that:

(i) every u_i is of length k;

(ii) the set U covers the string x;

(iii) the number $m = |U|$ of such substrings is the smallest possible.

§ Work supported in part by NSF grant CCR-0105678.
† Partially supported by a Marie Curie fellowship,Wellcome and Royal Society grants.
‡ Supported by an EPSRC studentship.
¶ Supported by a grant from the Natural Sciences & Engineering Research Council of Canada.

This problem was studied by Iliopoulos and Smyth [IS92], where they designed an $O(n^2(n-k))$ on-line algorithm. The idea of a k-cover is a generalization of the idea of a cover, where a string w is called a cover of a string x if x can be constructed by concatenations and superpositions of w. For example, if $x = ababaaba$, then aba and x are the covers of x. If $w \neq x$ covers x then w is called a *proper cover* of a *coverable* string x. The notion of a cover was introduced by Apostolico *et al.* [AFI91], where they gave a linear time algorithm for the shortest covers problem. Breslauer [B92] presented an on-line algorithm for the same problem. Moore and Smyth [MS94] presented a linear time algorithm to compute all the covers of every prefix of a string. An on-line algorithm for the same problem was developed by Li and Smyth [LS02]. Two $O(n \log n)$ algorithms for computing all maximal coverable substrings of a given string were also presented, one by Iliopoulos and Mouchard [IM93] and the other by Brodal and Pederson [BP00]. A lot of work has been done on parallel computation of covers; see for example [B94] and [IP94].

A minimum k-cover provides a theoretical classification of strings according to approximate periodicity. For every k, some strings have a minimum k-cover of cardinality 1, some a minimum k-cover of cardinality 2, and so on. Thus for a range of k, a minimum k-cover can provide a measure of how close to periodic every string x is. Practically, a minimum k-cover has a potential application in data compression of nonrandom strings. A minimum k-cover may also be useful in DNA sequence analysis. A DNA sequence is based on a four-letter *alphabet* for example $\{a, c, g, t\}$. Hence, finding the k-cover of a DNA sequence could be helpful for the analysis of its structure.

In this paper, we briefly present Iliopoulos and Smyth’s on-line algorithm. Their algorithm computes the minimum k-covers for all prefixes of a given string x in $O(n^2(n-k))$ time. We show why the algorithm does not work correctly (Section 3). In the rest of the paper we consider two closely-related problems:

(Problem 1) for given x, k and m, *decide* whether there exists a k-cover of x of cardinality m;

(Problem 2) *compute* a minimum k-cover of x.

For $m = 1$, Problem 1 can be solved in $\Theta(n)$ time simply by computing all the covers of x [MS94, MS95, LS02] while at the same time testing to determine whether or not each one is of length k. For $m > 1$ we show by reduction to 3-SAT that Problem 1 is NP-hard (Section 4). We then describe two efficient algorithms that yield approximate solutions to Problem 2 (Section 5). These approximation algorithms have been tested and shown to provide good results (Section 6). More approximation algorithms were proposed in [Y00].

2 Preliminaries

A *string* is a sequence of zero or more symbols drawn from an alphabet Σ. The set of all strings over Σ is denoted by Σ^*. The string of length zero is the *empty string* ϵ; a string x of length $n > 0$ is represented by $x_1 x_2 \cdots x_n$, where $x_i \in \Sigma$ for $1 \leq i \leq n$. A string w is a *substring* of x if $x = uwv$ for $u, v \in \Sigma^*$. More precisely, let $i \leq n$ and $j \leq n$ denote nonnegative integers: if $1 \leq i \leq j$, $x[i..j]$ denotes the substring of x

that starts at position i and has length $j - i + 1$; otherwise, $x[i..j] = \epsilon$. A string w is a *prefix* of x if $x = wu$ for some $u \in \Sigma^*$. Similarly, w is a *suffix* of x if $x = uw$ for some $u \in \Sigma^*$.

The string xy is a *concatenation* of two strings x and y. The concatenation of k copies of x is denoted by x^k. For two strings $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$ such that $x_{n-i+1} \cdots x_n = y_1 \cdots y_i$ for some $i \geq 1$ (that is, such that x has a suffix equal to a prefix of y), the string $x_1 \cdots x_n y_{i+1} \cdots y_m$ is said to be a *superposition* of x and y. Alternatively, we may say that x *overlaps* with y.

A substring w is said to be a *cover* of a given string x if every position of x lies within an occurrence of a string w within x. Additionally, if $|w| < |x|$ then w is called a *proper cover* of x. For example, x is always a cover of x, and $w = aba$ is a proper cover of $x = abaababa$.

For a given a nonempty string x of length n and a set

$$U = \{u_1, u_2, \ldots, u_m\}$$

of m strings each of length k, we say that U is a *k-cover* of x if and only if every position of x lies within an occurrence of some u_i, $1 \leq i \leq m$. If m is the minimum integer for which such a set U exists, then U is said to be a *minimum k-cover* of x. To avoid trivialities we suppose throughout that $1 < k < n/2$. Note that $1 \leq m \leq \lceil n/k \rceil$. Next we state some basic facts about the minimum k-cover.

Fact 1 The prefix $x[1..k]$ and the suffix $x[n-k+1..n]$ are both necessarily elements of every minimum k-cover of x.

Fact 2 The cardinality of a minimum k-cover of a string of length n is at most $\lceil n/k \rceil$.

Fact 3 A minimum k-cover of a string x is not unique.

For example, if $x = abcdefg$, then the sets

$$\{abc, bcd, efg\}, \{abc, cde, efg\}, \{abc, def, efg\}$$

are all minimum 3-covers of x.

In [IS92], the number of distinct minimum k-covers of a given string x of length n has been proved to be exponential in n. This is a major complicating factor in the design of polynomial time algorithm for computing the minimum k-covers of a given string.

3 Iliopoulos & Smyth On-Line Algorithm

Recall that in [IS92], Iliopoulos and Smyth designed an $O(n^2(n-k))$ time on-line algorithm for computing a minimum k-cover of a given string x of length n. Their algorithm scans a given string x from left to right and iteratively calculates a minimum k-cover for every prefix of x. The algorithm is based upon the following two main ideas:

1. Fact 1 states that a minimum k-cover of $x[1..i+1]$ must include the suffix $x[i-k+2..i+1]$. This is used as a yardstick to find a minimum k-cover.

2. For $i \geq k$, a minimum k-cover of $x[1..i+1]$ depends only on the minimum k-covers of the previous k positions; that is, the minimum k-cover of $x[1..i-k+1], \ldots, x[1..i-1], x[1..i]$.

To achieve efficiency, the algorithm stores for each positions i in x an array which identifies all the k-substrings that occur in *at least one* of the minimum k-covers. Let c_i be the cardinality of this set. At step $i+1$, the algorithm checks for each position $j \in i-k+1..i$, whether the current suffix $x[i-k+2..i+1]$ has already been included in the stored minimum k-cover of $x[1..j]$. If so then the set covers $x[1..i+1]$, otherwise the current suffix has to be added to the set. Among these k candidates, the algorithm chooses a set with the smallest cardinality as a minimum k-cover of $x[1..i+1]$. For more details see [IS92].

Lemma 3.1 For $i \geq 2k$ and $l, l' = 1, 2, \ldots$, let $U_{i,l}$ denotes the distinct minimum k-cover for $x[1..i]$. Then every minimum set $U_{i+1,l}$ is a superset of some minimum set $U_{j,l'}$, $i-k+1 \leq j \leq i$.

The above lemma is stated in [IS92] and it follows directly from the two ideas stated at the beginning of this section. The algorithm as we briefly described also relies on the correctness of the lemma. In the next example we will show that the lemma is not correct and consequentially nor is the algorithm. The following example illustrates just one of the situations where the algorithm fails to compute a minimum k-cover.

Example: If $x = bacaababbaaaccaabbabbbaaaac$ and $k = 3$ then when $i+1 = 27$, $j \in 24..26$, and position 27 should form its minimum k-cover from position 24 because $c_{24} = \min(c_j),\ j \in 24..27$. The minimum k-covers of position 24 are as follows:

$$U_{24,1} = \{bac, aab, abb, baa, cca\},$$

$$U_{24,2} = \{bac, aab, abb, baa, acc\}.$$

Neither of them contains the suffix aac, so we get $c_{27} = c_{24} + 1 = 6$, and accordingly the minimum k-covers of position 27 are as follows:

$$U_{27,1} = \{bac, aab, abb, baa, cca, aac\},$$

$$U_{27,2} = \{bac, aab, abb, baa, acc, aac\}.$$

But we can find at least one minimum k-cover that is different from $U_{27,1}$ and $U_{27,2}$; namely:

$$U_{27,3} = \{bac, aab, abb, baa, caa, aac\}.$$

$U_{27,3}$ is a k-cover of position 24, but not the minimum. However it will contribute to the minimum when position 27 is reached. There is a potential problem for future calculations if we lose $U_{27,3}$ at position 27; for example if we extend x by adding aa to the end. As we can see, $U_{27,3}$ can be a minimum k-cover of $x[1..29]$. Without keeping $U_{27,3}$, we shall get $c_{29} = 7$, one greater than the minimum.

The above suggests that in order to compute a minimum k-cover of the current position, we have to refer to every single k-cover of the previous positions. Since the number of minimum k-covers of a string may be exponential, we doubt that the problem of computing a minimum k-cover can be solved in polynomial time.

4 Problem 1 and NP-Completeness

The k-cover problem is to find a set cover of minimum size for a given string. Restating this optimization problem as a decision one, we wish to determine whether a given string has a k-cover of a given size m.
k_m-COVER $= \{\langle x, k, m\rangle$: string x has a k-cover of size $m\}$.

The following theorem shows that this problem is NP-complete.

Theorem 4.1 The k_m-COVER $\in$ NP.

Proof. To show that k_m-COVER $\in$ NP, for a given string x, we use the set U_m of m substrings all of length k as a certificate for x. Checking whether U_m is a k-cover can be accomplished in $O(n \log n)$ time by checking whether, for each position $1 \leq i \leq n$, i is covered by at least one of the k-substrings in U_m.

We next prove that 3-SAT $\leq_p$ k_m-COVER, which shows that a minimum k-cover problem is NP-hard. 3-SAT is well-known to be NP-complete [C71]. We transform 3-SAT to k_m-COVER. Let $V = \{v_1, v_2, \ldots, v_p\}$ be a set of variables, $C = \{c_1, c_2, \ldots, c_q\}$ be the set of clauses and $F = c_1 \wedge c_2 \wedge \ldots \wedge c_q$ be a 3-SAT formula with $c_i = \ell_1^i \vee \ell_2^i \vee \ell_3^i$, $1 \leq i \leq q$.

We shall show how to construct from F a string x such that x will have a k-cover of size m if and only if F is satisfiable. We choose $k = 3$ and note that there is an easy reduction to 2-CNF for $k = 2$. The string x is build of substrings separated by sequences of $sssss$; hence sss is one of the chosen covering k-strings, and thus we can focus on the individual substrings. The construction will be made up of truth-setting components, and satisfaction testing components.

Variable Choice

For each variable $v \in V$, we construct the following 6 substrings (each substring is proceeded and followed by $sssss$); each character is indexed by v:

(i) $\#_a\ r\ r\ \$\ v\ \phi\ \pi\ r\ r\ \#_a$	**(ii)**$\#_b\ t\ t\ \$\ \bar{v}\ \phi\ \pi\ t\ t\ \#_b$
(iii)$\#_a$	**(iv)** $\#_b$
(v)$\#_a\ \#_b$	**(vi)**$\#_b\ \#_a$

The only ways to cover the above strings with 9 or fewer length 3 strings, are one of the following (notice the uninteresting flexibility in (v) and (vi)):

1. $\{ss\#_a, rr\$, v\phi\pi, rr\#_a, \#_b tt, \$\bar{v}\phi, \pi tt, \#_b ss\}$ and one of $\{s\#_b\#_a, \#_b\#_a s\}$.

2. $\{\#_a rr, \$v\phi, \pi rr, \#_a ss, ss\#_b, tt\$, \bar{v}\phi\pi, tt\#_b\}$ and one of $\{s\#_a\#_b, \#_a\#_b s\}$.

To see this, consider covering string (iii). It can be done by one of $ss\#_a$, $\#_a ss$, $s\#_a s$, but only the first two could be used elsewhere, so one of them may as well be chosen. Clearly, 8 strings at least are needed to cover (i) and (ii) as they have no length 3 substring in common. Thus, to use only 1 additional string to cover (v) and (vi) we need to choose either $ss\#_a, \#_b ss$ or $\#_a ss, ss\#_b$.

The choice $v\phi\pi$ and $\$\bar{v}\phi$ (given by choosing $ss\#_a$) corresponds to $v = T$ while the choice $\bar{v}\phi\pi$ and $\$v\phi$ (given by choosing $\#_a ss$) corresponds to $v = F$.

Clause Satisfiability

For each clause $c \in C$, where $c = \ell_1 \vee \ell_2 \vee \ell_3$, the following substrings are created, again preceded and followed by $sssss$. The characters, except for $\$_i, \phi_i, \pi_i, \ell_i, i = 1, 2, 3$ are indexed by c also; $\$_i, \phi_i, \pi_i, \ell_i$ carry the index for the literal.

(i)$\$_1\ \ell_1\ \phi_1\ \pi_1\ h_1$	**(ii)** $\$_2\ \ell_2\ \phi_2\ \pi_2\ h_2$	**(iii)** $\$_3\ \ell_3\ \phi_3\ \pi_3\ h_3$
(iv)$\$_1$	**(v)**$\$_2$	**(vi)**$\$_3$
(vii)h_1	**(viii)**h_2	**(ix)**h_3
(x)$\phi_1\ \pi_1\ h_1\ d_1\ \phi_2\ \pi_2\ h_2$	**(xi)**$\phi_2\ \pi_2\ h_2\ d_2\ \phi_3\ \pi_3\ h_3$	**(xii)**$\phi_3\ \pi_3\ h_3\ d_3\ \phi_1\ \pi_1\ h_1$
(xiii)ϕ_1	**(xiv)**ϕ_2	**(xv)**ϕ_3

To cover (iv)-(ix) and (xiii)-(xv) we may as well choose $ss\$_i$, h_iss and $ss\phi_i$ as these are the only reusable substrings.

If ℓ_i is true, then $\ell_i\phi_i\pi_i$ was already chosen; otherwise $\$_i\ell_i\phi_i$ was chosen. Thus, if ℓ_i is false; in (i)-(iii), π_i remains to be covered. The only reusable covering string is $\phi_i\pi_i h_i$.

Consider strings (x)-(xii) and suppose at least one ℓ_i is true. Without loss of generality let it be ℓ_1. Then it is not hard to see that 5 more strings that include $\phi_2\pi_2h_2$ and $\phi_3\pi_3h_3$ thereby covering π_2 in (ii) and π_3 in (iii) suffice. We choose: $\phi_2\pi_2h_2, \phi_3\pi_3h_3, \pi_1h_1d_1, d_2\phi_3\pi_3$ and $d_3\phi_1\pi_1$. It is not hard to see that 5 covering strings are needed: 3 to cover d_1, d_2 and d_3, but this can only completely cover one of π_1, π_2 and π_3 as each occurs twice, and hence two more covering strings are needed for the remaining pair among π_1, π_2 and π_3.

If no ℓ_i is true, we are obliged to choose $\phi_1\pi_1h_1, \phi_2\pi_2h_2$ and $\phi_3\pi_3h_3$ as well as 3 strings to cover d_1, d_2 and d_3. At least 6 covering strings in all are needed. Thus, if F is satisfiable then the full string can be covered by

$$m = 9p + 6p + 3q + 5q + 1 = 15p + 8q + 1$$

covering strings, where p is the number of variables in F and q is the number of clauses. Otherwise, it needs at least $15p + 8q + 2$ covering strings. □

5 Approximate Minimum k-Cover

In this section we introduce two greedy algorithms to compute a minimum k-cover. The greedy method works by picking, at each stage, the k-substring which covers the greatest number of uncovered positions. The first algorithm works globally while the second algorithm follows a local strategy. To calculate all possible k-substrings in a given string x, both greedy algorithms use Crochemore's partitioning algorithm [C81] to preprocess the input string x.

Originally, Crochemore's algorithm was designed to compute the *repetitions* in a string in $O(n \log n)$ time. A string has a *repetition* when it has at least two consecutive equal substrings. For example, $abab$ is a repetition in $aababba = a(ab)^2ba$. We shall use the algorithm in another way — to find the sets of the starting positions of all the distinct substrings of length k in a given string x. This idea can be expressed more precisely as follows:

Given a string $x[1..n]$ and an integer k, Crochemore's algorithm is used to compute the equivalence classes of all equal substrings of length k in x. We denote these equivalence classes by $e_1, e_2, ..., e_m$, where the elements in e_i are sorted integers denoting starting positions of equal substrings, and m is the number of possible equivalence classes returned by the algorithm.

These elements are stored using a global array $L[1..n]$, such that $L[i]$ is the next position in the same equivalence class of equal substrings of length k. That is, $L[i] = j$ if $L[i..i+k-1] = x[j..j+k-1]$ and the circular sequence $i, L[i], L[L[i]], \ldots, L^{\ell}[i] = i$ identifies all ℓ k-substrings in x that are equal to $x[i..i+k-1]$.

For example, if $x = abaababaabaab$ and $k = 3$ then $e_1 = \{3, 8, 11\}$, $e_2 = \{1, 4, 6, 9\}$, $e_3 = \{2, 7, 10\}$, and $e_4 = \{5\}$ are the equivalence classes. Where aab, aba, baa, bab are the corresponding 3-substrings. Hence, the value of array L is as follows:

	1	2	3	4	5	6	7	8	9	10	11	12	13
$x =$	a	b	a	a	b	a	b	a	a	b	a	a	b
$L[i]$	4	7	8	6	5	9	10	11	1	2	3		
$Eid[i]$	2	3	1	2	4	2	3	1	2	3	1		

In the above, $Eid[i]$ identifies the equivalence class containing position i. In the following subsections, we shall present two approximation algorithms. We call the first Global-Uncovered and the second Local-Uncovered.

5.1 Global-Uncovered Algorithm

Recall that the greedy algorithm works by selecting one k-substring at a time that covers the most positions among the uncovered ones. Our greedy algorithm is comparable to the greedy one [J74] to construct the minimum set cover. The cost of a greedy solution is known to come always within a multiplicative factor of $\mathcal{H}(\max_j |EC_j|)$, where EC_j is the number of positions that could be covered by the k-substring j. Here, $\mathcal{H}(d) = \sum_{i=1}^{d} \frac{1}{i}$ is the dth harmonic number and is bounded by $1 + \log d$. This was shown by Johnson [J74] and Lovasz [L75] for the general SET COVER problem.

The key to Algorithm Global-Uncovered is finding the equivalence class which can cover the maximum number of so-far-uncovered positions efficiently. The details of the algorithm are provided in Figure 1. To achieve efficiency, the algorithm uses the following data structures:

1. An array $Ebucket[1..n]$ indexed by the number of so-far-uncovered positions that could be covered by a single equivalence class. Each element (bucket) of the array is doubly-linked list of the equivalence classes that could cover equal number of so-far-uncovered positions. Thus, every element of the doubly linked list contains an index of an equivalence class in addition to the *left* and the *right* pointers to the adjacent elements.

2. A two dimensional array $Eptr[1..m]$ indexed by the equivalence class j. Where $Eptr[j][bucket]$ identifies the bucket that includes j in its doubly linked list. In other words, equivalence class j could cover $Eptr[j][bucket]$ so-far-uncovered positions. Additionally $Eptr[j][ptr]$ is a pointer to the corresponding element of the doubly linked list $Ebucket[Eptr[j][bucket]]$. Thus, any elements of the doubly linked lists can be referenced in constant time by using $Eptr$.

Algorithm *Global-Uncovered*(x, k)

Input: A string x of length n, an integer $0 < k < n$

Output: An approximate minimum k-cover U_g

```
(L[1..n], Eid[1..n], start[1..m], m) ←CrochemorePar(x, k)
cover_so_far[1..n] ←F, F, ..., F
initialization:
U_g ←∅
for e ←1 to m do
        Euncov[e] ←0  **number of positions that could be covered by equivalence class e**
for i←1 to n − k + 1
        if i < L[i]
          then Euncov[Eid[i]] += min(k, L[i] − i)
          else Euncov[Eid[i]] += k
(Ebuckect, Eptr) ←Bucket-Sort(Euncov)
The algorithm:
k_prefix, k_suffix ←Eid[1], Eid[n − k + 1]
GU-Cover(k_prefix, Ebucket, Eptr)
Add(U_g, k_prefix)
if k_suffix ≠ k_prefix
   then GU-Cover(k_suffix, Ebucket, Eptr)
        Add(U_g, k_suffix)
e ←Head(Ebucket)
while e ≠ 0
       GU-Cover(e, Ebucket, Eptr)
       Add(U_g, e)
       e ←Head(Ebucket)
return U_g


Function GU-Cover(e, Ebucket, Eptr)
i ←start[e]   **the first element in the equivalence class e**
repeat
     for j ←1 to k do
             if cover_so_far[i + j − 1] = F then
                     cover_so_far[i + j − 1]←T
                     for every  l ∈ Eid[(i + j − 1) − k + 1], ... Eid[i + j − 1] do
                            Delete(Ebucket[Eptr[l][bucket]],Eptr[l][ptr])
                            if Eptr[l][bucket] ≠ 1
                              then Insert(Ebucket[Eptr[l][bucket − 1]],Eptr[l][ptr])
                            Eptr[l][bucket] ←Eptr[l][bucket] − 1
     i ←L[i]
until (i = start[e])
```

Figure 1: Global-Uncovered Algorithm.

Once *Ebucket* is established, the k-prefix and the k-suffix are the first elements to be included in the approximate minimum k-cover. The algorithm then iteratively choose a head element of *Ebucket* as an element of the approximate minimum k-cover. The head element is an equivalence class that covers the largest number of so far uncovered positions. Finding such equivalence classes costs $O(n)$ time throughout the calculations.

The algorithm requires $O(n \log n)$ time to run Crochemore's algorithm and an additional $O(n)$ time to construct and initialize *Ebucket* and *Eptr*. Note that a linear time Bucket-Sort has been used because the number of positions that could be covered by any equivalence class is bounded.

For each position i, $cover\_so\_far[i]$ is initialized to F and set to T once during the calculation. When $cover\_so\_far[i]$ is set from F to T, $O(k)$ elements in *Ebucket* may need to be deleted from the current bucket and inserted to the next bucket. Each rearrangement costs $O(1)$ time. Thus, the total time required to maintain the elements in *Ebucket* throughout the calculation is $O(kn)$. Summing the above gives the total running time: $O(n \log n) + O(n) + O(kn) = \max\{O(n \log n), O(kn)\}$ time, which for a fixed k, asymptotically approaches $O(n \log n)$ as n increases to ∞.

5.2 Local-Uncovered Algorithm

Algorithm Local-Uncovered chooses its candidate element, of the approximate minimum k-cover, in a range of $Eid[left\_uncover-k+1]..Eid[left\_uncover]$; the integer $left\_uncover$ keeps track of the leftmost so-far-uncovered position. The algorithm uses the array $uncover\_no$. The array $uncover\_no[1..m]$ is indexed by the equivalence classes, where $uncover\_no[j]$ is the number of positions corresponding to equivalence class j that have not been covered. Hence, the values of the array need to be updated dynamically during the computation. The details of the algorithm are provided in Figure 2.

The initialization is just the same as in Global-Uncovered. However, we need to update $uncover\_no$. As in Global-Uncovered, the k-prefix and the k-suffix are the first two elements to be included in the approximate minimum k-cover. The algorithm then tries to cover the leftmost uncovered position with the k-substring corresponding to the equivalence class which can cover the maximum number of uncovered positions. That is, let $j = left\_uncover$ if $j < n$, then the chosen k-substring is the one corresponding to equivalence class satisfying

$$max\{uncover\_no[Eid[j-k+1], uncover\_no[j-k+2], \ldots, uncover\_no[Eid[j]]\}.$$

A brief analysis of the algorithm shows that the algorithm requires:

- $O(n \log n)$: to run Crochemore's algorithm;
- $O(n)$: Step 2, the loop on (Steps 6-9), and the total time spent in Add();
- $O(k)$: the loop on (Steps 19-23);
- $O(kn)$: is the total time of the LU-Cover subroutine.

Summing the above gives the total running time $O(n \log n) + O(n) + O(k) + O(kn) = max\{O(n \log n), O(kn)\}$ time.

Algorithm *Local-Uncovered*(x, k)
Input: A string x of length n, an integer $0 < k < n$
Output: An approximate minimum k-cover U_l

```
(L[1..n], Eid[1..n], m) ←CrochemorePar(x, k)
cover_so_far[1..n] ←F, F, ..., F
initialization:
U_l ←∅
left_uncover ←1
for i←1 to n − k + 1 do
        if i < L[i]
          then uncover_no[Eid[i]] += min(k, L[i] − i)
          else uncover_no[Eid[i]] += k
The algorithm:
k_prefix, k_suffix ←Eid[1], Eid[n − k + 1]
LU-Cover(k_prefix, 1, uncover_no, left_uncover)
Add(U_l, k_prefix)
if k_suffix ≠ k_prefix then
        LU-Cover(k_suffix, n − k + 1, uncover_no, left_uncover)
        Add(U_l, k_suffix)
while left_uncover < n do
      max = 0
      for j ←1 to k do
            if uncover_no[Eid[left_uncover − j + 1]] > max then
                  max ←uncover_no[Eid[left_uncover − j + 1]]
                  e ←Eid[left_uncover − j + 1]
                  s ←left_uncover − j + 1
      LU-Cover(e, s, uncover_no, left_uncover)
      Add(U_l, e)
return U_l
```

```
Function LU-Cover(e, start, uncover_no, left_uncover)
i ←start
repeat
    for j ←1 to k do
          if cover_so_far[i + j − 1] = F then
                cover_so_far[i + j − 1]←T
                for every l ∈ Eid[(i + j − 1) − k + 1], ... Eid[i + j − 1] do
                      uncover_no[l] −= 1
    i ←L[i]
until (i = start)
while left_uncover ≤ n and cover_so_far[left_uncover] do
      left_uncover ++
```

Figure 2: Local-Uncovered Algorithm.

Length	$\|U_N\|$	$\|U_{GU}\|$	$\|U_{LU}\|$	$\|U_{best}\|$	α_N (%)	α_{GU} (%)	α_{LU} (%)
100	12	11	11	11	9.09	0	0
200	14	14	14	14	0	0	0
300	14	15	15	14	0	7.14	7.14
400	16	15	17	15	6.67	0	13.3
500	17	17	17	17	0	0	0
600	16	16	16	16	0	0	0
700	18	16	16	16	12.5	0	0
800	17	17	19	17	0	0	11.8
900	18	16	18	16	12.5	0	12.5
1000	18	17	16	16	12.5	6.25	0
Average (%)	/	/	/	/	5.33	1.34	4.47

Table 1: Pseudo-Random Strings on Alphabet $\{a,\ b,\ c\}$, and $k = 3$

6 Experimental Results

We used four types of strings: sturmian strings, pseudo random strings on the alphabets: $\{a, b\}$, $\{a, b, c\}$, $\{a, b, c, d\}$, DNA sequences*, and English text. In order to compare our approximate methods in term of effectiveness, we developed a naive algorithm based on the Iliopoulos and Smyth algorithm. This naive algorithm finds the minimum k-cover at position $i+1$ by testing each position $j \in i-k+1..i$ in the same way as in Iliopoulos and Smyth's. However, the key difference is that the algorithm stores not only the covers that are minimum but also those that are one more than minimum at every position. Thus, the aim here is to store as much information as possible taking into consideration the limitation of the computer's resources. The implementation results show that the naive algorithm does not always yield the best k-cover - in most cases the two approximate algorithms yield better results. Let U_{min} be the minimum k-cover of a string x, U_N be the result computed by our naive method, U_{GU} be the result computed by Global-Uncovered algorithm, and U_{LU} be the result computed by Local-Uncovered algorithm. Then the following simplifying assumption has been made:

$$|U_{min}| \leq |U_{best}| = min\{|U_N|, |U_{GU}|, |U_{LU}|\}$$

Table 1, 2, 3 show that Algorithm Global-Uncovered yields the best result in most cases, the naive algorithm never exceed a deviation of 7.83%, and Algorithm Local-Uncovered never exceed 6.24%. The following observations are also worth mentioning:

- The Sturmian strings are very well-structured. For the tested Sturmian strings, from length of 20 to 1000, for every $k \in 3, 4, 5$, $|U_{best}| = 2$.
- For the tested pseudo-random strings and DNA sequences, $|U_{best}|$ increases as the values of k, the length n, and the alphabet size are increasing.
- Let $|U_{best-DNA}|$ denotes the cardinality of the approximate minimum k-cover of DNA sequence and $|U_{best-abcd}|$ denotes the cardinality of the approximate

*excerpted from www.cbs.dtu.dk/databases/DNA2protSS/nucall.seq.

Length	$\|U_N\|$	$\|U_{GU}\|$	$\|U_{LU}\|$	$\|U_{best}\|$	α_N (%)	α_{GU} (%)	α_{LU} (%)
100	19	19	19	19	0	0	0
200	25	26	27	25	0	4.00	8.00
300	32	29	29	29	10.3	0	0
400	37	34	36	34	8.80	0	5.88
500	36	36	35	35	2.86	2.86	0
600	37	36	37	36	2.78	0	2.78
700	37	35	38	35	5.71	0	8.57
800	42	37	39	37	16.2	0	5.41
900	42	35	42	35	20	0	20
1000	42	38	39	38	10.5	0	2.63
Average (%)	/	/	/	/	7.71	0.68	5.32

Table 2: Pseudo-Random Strings on Alphabet $\{a,\ b,\ c,\ d\}$, and $k = 3$

Length	$\|U_N\|$	$\|U_{GU}\|$	$\|U_{LU}\|$	$\|U_{best}\|$	α_N (%)	α_{GU} (%)	α_{LU} (%)
60	13	13	13	13	0	0	0
126	21	22	23	21	0	4.76	9.52
171	23	22	23	22	4.54	0	4.54
234	25	24	26	24	4.17	0	8.33
312	32	29	30	29	10.3	0	3.45
432	26	27	29	26	0	3.85	11.5
591	34	31	35	31	9.68	0	12.9
771	40	34	36	34	17.6	0	5.89
1233	43	38	37	37	24.3	2.70	0
Average (%)	/	/	/	/	7.83	1.26	6.24

Table 3: DNA Sequences, and $k = 3$

minimum k-cover of pseudo-random strings on alphabet $\{a,\ b,\ c,\ d\}$. For the same value of k and n, $|U_{best-DNA}| < |U_{best-abcd}|$. We can make a conjecture that DNA sequences are better structured than pseudo-random strings on an alphabet of size 4.

Conclusions

We have shown that for $k \geq 2$, the k-cover problem (Problem1) is NP-Complete. We have then proposed two $O(n \log n)$ greedy algorithms that can be used to calculate an approximate minimum k-cover. The results obtained by the algorithms are believed to come within a multiplicative factor of the minimum. Prove this has been left as an open problem.

References

[AFI91] A. Apostolico, M. Farach & C. S. Iliopoulos, **Optimal superprimitivity testing for strings**, *Information Processing Letters 39-1* (1991) 17-20.

[B92] D. Breslauer, **An on-line string superprimitivity test**, *Information Processing Letters 44* (1992) 345-347.

[B94] D. Breslauer, **Testing string superprimitivity in parallel**, *Information Processing Letters 49-5* (1994) 235-241.

[BP00] G. S. Brodal & C. Pederson, **Finding maximal quasiperiodicities in strings**. *In Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM)* (2000) 397-411.

[C71] Stephen A. Cook, **The complexity of theorem-proving procedures**, *Proc. Third Annual ACM Symp. on Theory of Computing* (1971) 151-158.

[C81] M. Crochemore, **An optimal algorithm for computing all the repetitions in a word**, *Information Processing Letters 12-5* (1981) 244-248.

[IM93] C. S. liopoulos & L. Mouchard, **An $O(n \log n)$ algorithm for computing all maximal quasiperiodicities in strings**, *Theoratical Computer Science 119-2* (1993) 247-265.

[IP94] C. S. Iliopoulos & K. Park, **An optimal $O(\log \log n)$-time algorithm for parallel superprimitivity testing**, *Journal of the Korea Information Science Society 21-8* (1994) 1400-1404.

[IS92] C. S. Iliopoulos & W. F. Smyth, **An on-line algorithm of computing a minimum set of k-covers of a string**, *Proc. Ninth Australasian Workshop on Combinatorial Algorithms (AWOCA)*, (1998) 97-106.

[J74] D. S. Johnson, **Approximation algorithms for combinatorial problems**, *Journal of Computer and System Science 9* (1974) 256-278.

[MS94] D. Moore & W. F. Smyth, **An optimal algorithm to compute all the covers of a string**, *Information Processing Letters 50-5* (1994) 239-246.

[MS95] D. Moore & W. F. Smyth, **A correction to: An optimal algorithm to compute all the covers of a string**, *Information Processing Letters 54* (1995) 101-103.

[L75] L. Lovasz, **On the ratio of optimal integral and fractional covers**, *Discrete Mathematics 13* (1975) 383-390.

[LS02] Y. Li & W. F. Smyth, **Computing the cover array in linear time**, *Algorithmica 32-1*, (2002) 95-106.

[Y00] Lu Yang, **Computing the Minimum k-Cover of a String**, *M. Sc. thesis, McMaster University*, (2000).

Learning the Morphological Features of a Large Set of Words*

Abolfazl Fatholahzadeh

Supélec - Campus de Metz
2, rue Édouard Belin, 57078 Metz, France.

e-mail: Abolfazl.Fatholahzadeh@supelec.fr

Abstract. Given K - a large set of words - this paper presents a new method for learning the morphological features of K. The method, LMF, has two components : preprocessing and processing. The first component makes use of two separate methods, namely, refinement and time–space optimization. The former is a method that uses the closed world assumption of the default logic for partitioning K into a set of hierarchical languages. The latter is for efficiently learning the morphological features of each language outputted by the former method. Although, the finite-state transducers or the two-trie structure can be used to map a language onto a set of values, but we use our own competitor which has recently been proposed for such a mapping, consisting of associating a finite-state automaton accepting the input language with a decision tree (dt) representing the output values. The advantages of this approach are that it leads to more compact representations than transducers, and that decision trees can easily be synthesized by machine learning techniques.

In the processing phase, given an input string (x), thanks to the hierarchical languages establishing the preferency order for the utilization of the current automaton(g_i) among the multiple ones, if x can be spelled out using g_i, then the output is returned using its counterpart namely dt_i, otherwise, we inspect other alternative until an output or failure be done. LMF has learned good strategies for the large sets of the words which are consuming tasks form space and times point of views *e.g., all* the verbs in French, including all the conjugated forms of each verb.

Keywords: morphological features, automata, decision trees, learning.

1 Introduction

The morphological features (*i.e.,* mode, tense, person and gender) are supposed to be the important ingredients of the lexicons which are widely used in the process of determining for a word (*e.g.,* “livre”) its output values (*e.g.,* Verb+IND-PRES-1-SING, Verb+IND-PRES-3-SING, Verb+IMP-PRES-3-SING, Noun+MASC-SING and Noun+FEM-SING).

*This work is partially supported by le Conseil Régional de Lorrain.

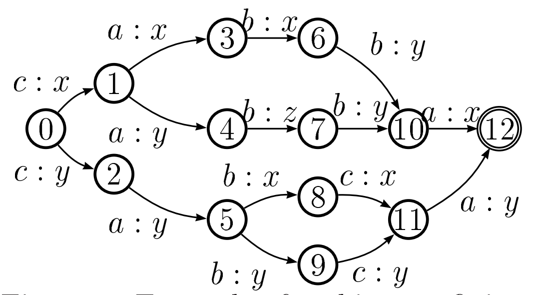


Figure 1: Example of ambiguous finite-state transducer shown by a (13,16) automaton [4, Page 158].

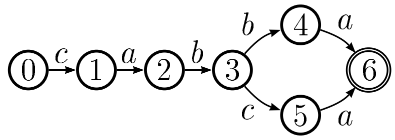


Figure 2: Our alternative - a (7,7) unlabeled automaton along with two decision rules. If b2 = ’b’ Then v1 = [xxxxx,xxyyx,xtzyx]. If b2 = ’c’ Then v2 = [yzxxy,yzyyy]. $\mathtt{b}_2$ stands for the second character from right to left of the input language.

An obvious solution to such a task is to store all the desired words along with their associated output values in a large-scale dictionary. But in this case two major problems have to be solved: *fast lookup* and *compact representation*. Two modern and efficient methods can achieve fast lookup by determination and compact representation by minimization. The first method is the technique of *two–tries* proposed by Aoe et al [1]. This method has the advantage of being applicable to a dynamic set of keys but unfortunately it has the disadvantage (Please refer to the page 488 of [1]) of containing more than states (hence the transitions) representing the data compared to its competitor, namely, the automata [13].

The second method is the *transducers* (*i.e.,* automata with outputs) [6, 8, 9] which have proved to be a very formal and robust execution framework for linguistic phenomena, but there are still some aspects that should be investigated. In particular, as shown in Figures 1, the transducers assign the unnecessary labels to some arcs of the graph representing the automaton. That is why, in our recent work, we have proposed a method to avoid such unnecessary labels (hence the states and the transitions) as pictured in Figure 2. Our solution for mapping a language onto a set of values is based on associating a finite-state automaton accepting the input language with a decision tree representing the output values. The advantages of this approach are that it leads to more compact representations than transducers, and that decision trees can easily be synthesized by machine learning techniques.

For the sake of clarity, we consider only the verbs in a given language and will show how our alternate approach can be combined with the closed world assumptions of the default reasoning. We show that the representation developed here provides a richer language for dealing with a set of strings where each of which is associated with one or more set of strings while keeping in the core of our system the two mentioned desiderata: compact representation and fast lookup. After presenting the default reasoning and its applicability to the morphology, we illustrate in Section 3 combining the automata and the decision tree. In Section 4 the refinement is described. The main algorithm of LMF along with examples in four languages closes: Azeri, English, French and Persian are described in Section 5. Finally, the concluding remarks close the paper.

2 Using Default Logic in Morphology

Default reasoning is a special but very important form of *non–monotonic* reasoning [5]. The term “default reasoning” is used to denote the process of arriving at conclusions

based upon patterns of inferences of the form "In the absence of any information to the contrary assume ..." (*e.g.,* if all elephants we have seen had a trunk, we might think that all elephants have a trunk). Of course, the possible circumstances in which any "presumed" correct line of reasoning can be defeated astound, and we are doomed to make mistakes when our experiences does not support the current situation. If we assume that the morphology world of the natural languages is closed one then there is a great chance that the rate of the classification noise be lower, even zero.
Example 1: *w.r.t.* the world of the verbs in French, even if there is no indications about the verb "zaper" in our system, LMF is able to learn 95 morphological features associated with the conjugated forms (*e.g.,* "zapons") of that verb.

Remark 1: The number 95 came from the fact that LMF is designed to learn the morphological features of all modes, namely indicative (IND), subjunctive (SUB), conditional (COND), imperative (IMP), infinitive(INF) and participate (PART). IND mode has 48 forms in eight tenses: present, imperfect, past, future, *etc.* Each of which allows to generate six forms according to: (1) gender (singular and plural); and (2) the person (1, 2, and 3). SUB mode has 24 forms in four tenses. COND mode has 24 forms in two tenses. IMP, INF modes has two and three forms, respectively. PART mode has usually three forms, two for some irregular verbs.

2.1 The Closed World Assumption

It seems not generally recognized that the reasoning components of many natural language understanding systems have default assumption built into them. The representation of knowledge upon which the reasoner computes does not explicitly indicate certain default assumptions. Rather, these default are realized as part of the code of the reasoner's process structure containing the hierarchies.

The starting point of the default reasoning is a set of *inference rules*(axioms) possibly along with some facts of the domain at hand collected in database which we call axiomal database (noted by G_{ax}). Given G_{ax}, the task based on the "specificity" and "inheritance" is to draw a plausible inference for the input. These can be illustrated by the classical Tweety example as follows: Consider the database containing four defaults: "penguins are birds", "penguins do not fly", "birds fly" and "birds have wings". "Specificity" tell us Tweety is a penguin, then Tweety doesn't fly because *penguin* is a more specific classification of Tweety than *bird*. "Inheritance" on the other hand, does equip Tweety with wings, by virtue of being a bird, albeit an exceptional bird *w.r.t.* flying ability.

From efficient implementation of the reasoner's process structure point of view, if the class "Specificity" lies "*above*" the generic class *i.e.,* there is some pointer leading from penguin's to node bird in G_{ax}, then given a particular penguin we can conclude that it doesn't fly. Notice that the reasoner's process structure of G_{ax} can be either a network - the graph of the taxonomy - or a set of first order formulae. The second option has been chosen to form G_{ax} of the morphology world in our work. In that option for fast inference purpose, G_{ax} is organized according to *priorities* which are given as ordering of predicates formulae, or default rules: in conflicting situations preference is given to item with high priority. That is to say, the data are added in G_{ax} in the following orders: (1) the facts of the exceptional data; (2) the facts associated with generic axioms; (3) the exceptional axioms describing the specificity; and finally (4) the generic axioms.

Example 2: *w.r.t.* Tweety the orders of G_{ax} is as follows: (1) Penguin(tweety); (2) Bird(tweety); (3) $(\forall x)Penguin(x) \rightarrow \neg Flies(x)$; (4) $(\forall x)Bird(x) \rightarrow Flies(x)$.

(3) can be paraphrased as "penguins usually cannot fly". If a particular penguin (say Foo) can fly, this is obviously a counter exceptional data (or insensitivity to specificity) *w.r.t.* to (3). Although, how the representation of the insensitivity to specificity can be done in the open world (*i.e.,* the data related to the exceptions and in particular those of the counter exceptions are not known in advance), but this is not a limitation for our work because the databases of LMF is composed only using three predicates : regular, exceptional and counter-exceptional. The selection of the counter exceptional data is based on the fast inference purpose.

The LMF policy for such above purpose is to take into account both the high priority of usage in the text of a given language (*e.g.,* the auxiliary verbs of a given language such as "avoir" - to have - or "être" - to be -) and the seldom of data *w.r.t.* exceptional data (*e.g.,* "aller" -to go - the only member of the class 22 of the irregular verbs) or its specificity *w.r.t.* the general data (*e.g.,* "Haïr" meaning to hate, which is also a unique member of the 20th class of the regular verb).

3 Combing the Automata and the Decision Trees

In what follows, we summarize our recent work [3] concerning the combination of the automata and the decision trees. We assume the reader to be familiar with both the theory of finite automaton and the decision tree learning as presented in standard books *e.g.,* [13] and [7], respectively. We refer to a *key* and a *value* denoted by k and kv, respectively, as a sequence of characters surrounded by empty spaces which may have one or more internal spaces. We may use key and word (including verbs), interchangeably, as well as, the value, key–value and the morphological features.

The input of our algorithm for such above combination is the following customary form: $f = \{(k_i, v_i)|i = 1, \ldots, n\}$ for representation and fast lookup. The point of our idea is as follows: If an input string(x) can be recognized using the *unlabeled* finite-state-automaton (g) associated with the keys (of f) - hence having less states and transitions compared to the transducer as shown in Figures 1 and 2 - then use the learn decision tree (dt) for outputting the value associated with x. Table 1 shows a simple decision tree (dt) of $f1 = \{(Iran, Tehran), (Iraq, Baghdad), (Ireland, Dublin)\}$. Note that the $\mathtt{dt}$ *w.r.t.* $\mathtt{f}_2 = \{$(Iran,Asia),(Iraq,Asia)$\}$ has a unique solution-path i.e. $(kvAsia)$ - no condition (*i.e.,* question) is required to discriminate the key-value.

3.1 Acyclic Finite-state Automaton

Recall that an acyclic finite-state automaton is a graph of the form $\mathtt{g} = (Q, \Sigma, \delta, q_0, F)$ where $\mathtt{Q}$ is a finite set of states, Σ is the alphabet, $\mathtt{q}_0$ is the start state, $F \subseteq Q$ is the accepting states. δ is a partial mapping $\delta : Q \times \Sigma \longrightarrow Q$ denoting *transition.* If $a \in \Sigma$, the notation $\delta(q, a) = \perp$ is used to mean that $\delta(\mathtt{q}, \mathtt{a})$ is undefined. Let $\Sigma^\star$ denotes the set containing all strings over Σ including zero-length string, called the

Table 1: Backward attribute-based Data and Decision Tree.

b_7	b_6	b_5	b_4	b_3	b_2	b_1	KV	Solution-Path	Question	KV
$\star$	$\star$	$\star$	I	r	a	n	Tehran	(b_1 n kv Tehran)	b_1 = n?	Tehran
$\star$	$\star$	$\star$	I	r	a	q	Baghdad	(b_1 q kv Baghdad)	b_1 = q?	Baghdad
I	r	e	l	a	n	d	Dublin	(b_1 d kv Dublin)	b_1 = d?	Dublin

Table 2: Ten keys of the same lengths along with associated values.

Key	onC	myC	mnH	onH	nnH	nnC	mnC	nyC	myH	oyC
Value	down	down	up	down	up	up	up	up	down	down

empty string ε. The extension of the partial δ mapping with $x \in \Sigma^\star$ is a function $\delta^\star : Q \times \Sigma^\star \longrightarrow Q$ and defined as follows:

$$\delta^\star(\mathtt{q}, \varepsilon) = \mathtt{q}$$

$$\delta^\star(\mathtt{q}, \mathtt{ax}) = \begin{cases} \delta^\star(\delta(q,a), x) & \text{if } \delta(q,a) \neq \perp \\ \perp & \text{otherwise.} \end{cases}$$

A finite automaton is said to be ($\mathtt{n}$,$\mathtt{m}$)–automaton if $|\mathtt{Q}| = n$ and $|\mathtt{E}| = m$ where $\mathtt{E}$ denotes the set of the edges (transitions) of $\mathtt{g}$. The property $\delta^\star$ allows fast retrieval for variable-length strings and quick unsuccessful search determination. The pessimistic time complexity of $\delta^\star$ is $\mathcal{O}(n)$ *w.r.t.* a string of length $\mathtt{n}$.

3.2 Decision Tree Learning

Decision tree learning is a method for approximating discrete–valued target functions, in which the learned function is represented by a decision tree (*dt*). Learned decision trees can also be re-represented as a set of if–then rules to improve human readability.

Example 3: Below we list the if–then rules representing the decision tree associated with data of Table 2.

If $f_1 =$ 'o' **Then** KV = 'down';
If $f_1 = 'm' \wedge f_2 =' y'$ **Then** KV = 'down';
If $f_1 = 'm' \wedge f_2 =' n'$ **Then** KV = 'up';
If $f_1 = 'n'$ **Then** KV = 'up';

where f_1 and f_2 denote first character and sceond character (of the key from left to right), respectively. Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instances. Each node in the tree specifies a test of some *attribute* (*e.g.,* $b1$ of Table 1) instance, and each branch descending from that node corresponds to one of the possible values for this attribute. An instance is classified by starting at the root of the tree, testing the attribute value by this node, then moving down the tree branch corresponding to the value of the attribute in the given example. This process is then repeated for the subtree rooted at the new node. Notice that the implementation of the decision tree is based on `m-array` tree rather than the binary one. The former allows to save the decision tree in a less space compared to the latter. Figure 4 shows such a learned tree representing the values of the keys of Table 2.

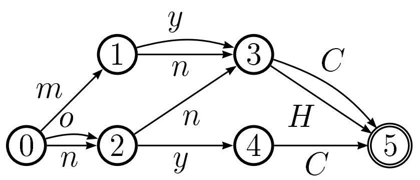


Figure 3: A (6,10) unlabeled automaton for recognizing the keys of Table 2.

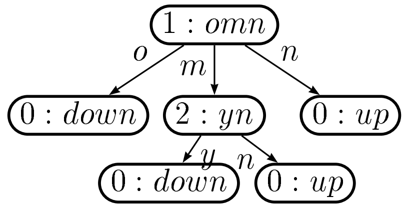


Figure 4: Learned decision tree for determining the value of any recognized key of Table 2.

Table 3: Distribution of French regular verbs according to the class and the frequency noted by C and F, respectively.

C	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
F	3875	156	165	342	69	114	19	12	9	254	26	49	2	302	1

4 Refinement

The refinement process has the following tasks to perform:

1. Transform the input of LMF, namely our input, namely $f = \{(k_i, v_i)|i = 1, \ldots, n\}$ into axiomal database D_{ax}, as described in Section 2.1.

2. Partition D_{ax} into the counter-exceptional, exceptional and general axioms.

The transformation is based on the closed world assumption of the morphology assuming that the set of the words of (f) noted by K can be divided into two subsets of so-called regular and irregular words. The regular forms follows the fact that their derivate/inflectional forms (each noted by d_k) can be generated using those axioms specified by the linguists which are usually further refined in a set of finer regular axioms (*axiom*). Using a root (of the word) each axiom allows to generate all d_ks of the word. The root is obtained by removing a particular substring of used *axiom*.

Example 4: The regular forms of the verbs in French is divided into the first group containing 13 classes (ranged from 6 to 18) and the second group which is composed of two classes (ranged from 19 to 20), where each number stands for an *axiom*. Below the repartition of 5189 infinitives (of the regular verbs) used in our experiment is shown in Table 3.

Remark 2: As appear from Table 3, 20th class has only one member, namely "Haïr". However, as we mentioned earlier, it is not considered is a a regular data. Indeed, *w.r.t.* to the inference process, it is wise to consider it as a counter-exceptional data. The reason is to speed up the inference processing by mentioning explicitly the data and axioms is the following order: counter-exceptional, exceptional and general. This process constitutes the well known practical trick of the default logic. So, 5188 (*i.e.,* 5189 -1) roots along with 19 classes will be used as the reservoir for learning the extended database of 492860 (*i.e.,* 5188 × 95) d_ks of the lexicographers expressed in a raw database.

An *axiom* can be described using a two dimensional vector of size r, where r stands for the number of morphological features in use. The first row of such a vector

Table 4: Information on size of 13943 verbs of the third group in French and morphological information along with the forest of the decision trees obtained by the partitive learning mode. Ent. refers to number of call to the entropy function.

Data		**Decision Tree**			**Gain**	
Len.	Freq.	Inodes	Leaves	Ent.	K%	V%
2	11	9	3	15	66%	19%
4	183	133	40	371	81%	23%
5	412	225	66	904	88%	44%
6	943	460	131	2149	91%	47%
7	1480	578	202	3388	93%	57%
8	2160	727	240	5065	94%	62%
9	2317	692	342	6664	95%	67%
10	2115	582	252	6531	96%	70%
11	1729	445	207	6361	96%	72%
12	1168	318	125	4980	97%	70%
13	733	164	69	3472	97%	75%
14	389	106	50	2620	97%	70%
15	183	59	22	1624	97%	68%
16	72	36	18	1063	95%	50%
17	25	9	4	288	97%	64%
18	7	3	2	83	96%	58%

is composed of r the values. The second row contain different substrings related to d_ks. Usually, the lexicographers are used to add the word in explicit database in which each entry is composed one d_k and a value. Since it may happen that for a d_k different values be associated with it (*e.g.,* aime IND-PRES-1-SING, IMP-PRES-3-SING, *etc.*) therefore, the learning process should assure to collect them into a set of morphological features representing a set of unique ambiguity class. In summary, the entire lexicon can viewed as follows. First on can form the the four following reservoir f_g, s_g, f_e and f_c representing: (1) f_g: Database related to the general axioms; (2) s_g: Database of suffixes of the regular (general) words; (3) f_e: Database of derivate forms expressed as the exceptional data; (4) f_c: Database of derivate forms based on the high priority relating the counter exceptional data. Notice that f_g along with s_g will be used to recognize the derivate forms of the words governed by the general axioms.

4.1 More Refinement: Learning by Partitive Mode

As we mentioned earlier, the input of decision tree learning is a fixed attributes the size of this table is $\ell + 1 \times n$, where ℓ denotes the length of the longest keys of f and n is the number of *keys*. Usually, we have to use the dummy characters (noted by $\star$ see Table 1). Using the dummy characters augment the size of the input table. Because of the very recursive nature of the learning process, including the characterization of the decision tree may be a time consuming task for the large data. An alternative to the a unique table is to employ multiple tables as follows. First f is divided into q

user-inputs (f_i) such that the length of the keys of each f_i be identical, then form the corresponding decision trees. So, in the partitive mode, we have to learn a *forest of the decision tress* : composed a vector of r positive integers. ith number is pointed to the ith decision tree.

Searching a value for an input string (x of length y) works as follows. If y belongs to the vector of above mentioned numbers, first we spell out x this time using the automaton associated with entire keys of K. If x spelled out correctly, then we use the y^{th} decision tree to output the value.

Example 5: The value of $x = ababab ad$ can not be learned *w.r.t.* current $f = \{(abc, 1), (ababbac, 2)(abababc, 3)\}$. We have length(x) = 8 which is not member of {3,5,7}. In the contrary, for $x = abc$ the value is 1 *i.e.,* (1) $length(x) \in \{3, 5, 7\}$, (2) x is recognized using the automaton associated with $K = \{abc, ababc, abababc\}$ and (3) no question is required for f_3 the value is 1. Table 4 shows the Information on size of 13943 verbs of the third group in French and morphological information along with the forest of the decision trees obtained by the partitive learning mode.

5 Main Algorithm

Below the algorithm for learning morphological features is given which is composed of two components: preprocessing and processing. In the first component four automata and two decision trees along with a forest decision trees containing r decision trees are formed, where r stands for the number of partitions of the exceptional data according to the same key-length criterion. In the second component, if an user-input (x) can be recognized by one of the four automata (see below for the order in use) then the corresponding decision tree will be inspected to output the value. The argument of main function are:

1. $f_g = \{(root_i, axiom_i)|i = 1 \ldots, n_1\}$ *i.e.,* Database related to the general axioms;
2. $s_g = \{(suf_i, mf_i|i = 1 \ldots, m_1\}$ *i.e.,* Database of suffixes of the regular (general) words; mf stands for a morphological features or a set of alternate morphological features;
3. $f_e = \{(d_i, mf_i)|i = 1 \ldots, n_2\}$ *i.e.,* Database of derivate forms expressed as the exceptional data; d_i refers to a derivate form of a base word (*e.g.,* infinitive);
4. $f_c = \{(d_i, mf_i)|i = 1 \ldots n_3\}$ *i.e.,* Database of derivate forms based on the high priority relating the counter exceptional data.

---

func LearningMorphologicalFeatures(f_g, s_g, f_e, f_c)
  $K_g \leftarrow CollectKeys(f_g)$. $K_c \leftarrow CollectKeys(f_c)$.
  $g_{kg} \leftarrow FormAutomaton(K_g)$; $g_{kc} \leftarrow FormAutomaton(K_c)$.
  ApplyPreprocessingPartitiveMode(f_e).
  $g_{Ke} \leftarrow FormAutomaton(K_e)$.
  $table_c \leftarrow FormInputForLearning(f_c)$.
  $t_c \leftarrow LearnDecisionTree(table_c)$.
  $t_s \leftarrow LearnDecesionTreeOfSuffixes(s_g)$.

ApplySearch(x).{Processing component, x is an input string.}
cnuf

The function $FormAutomaton()$ follows the elegant algorithms described in [2] for the incremental construction of minimal acyclic finite state automata and transducers from both sorted and unsorted data We adapted the former one such that the length of the longest key be calculated for being used later in the construction of suitable input for learning the `dt` of the counter exceptional data. Please refers to [3] for the description of the function $FormInputForLearning()$ and $LearnDecisionTree()$. The construction of the forest of the decision trees works as follows.

---

func ApplyPreprocessingPartitionMode(f_e)
  $\bigcup_{i=\ell_1}^{\ell_x} f_{ei} \leftarrow Partition(f_e)$
  for $i \in (\ell_1, \ldots \ell_x)$ **do**
    $K_{ei} \leftarrow CollectKeys(f_{ei})$; $g_{kei} \leftarrow FormAtuomaton(f_{ei})$.
    $Table_{ei} \leftarrow FormInputForLearning(f_{ei})$
    $t_{ei} \leftarrow LearnDecisionTree(Table_{ei})$.
  end for
cnuf

Since the search order is based on looking at the following order : (1) counter exceptional, (2) exceptional and general data, then processing component is as follows:

---

func ApplySearch(x)
  return(SearchValue(x, g_{kc}, t_c) OR SearchValueUsingPartitionMode(x, g_{ke}, $forest$) OR SearchByMismatch(x, g_{kg}, s_g, t_s)).
cnuf

For knowing how SearchValue() works, again consider Figure 4 where zero used in a node indicates that node is a leaf one. A positive integer number used in a node has its own meaning indicating the test to be done taking into account the content of the current node under inspection *e.g.,* “1:omn” means that if the first character of x is ’m’ then gets the value by descending in the sub-tree of first child. Since the sub-tree has only one node - a leaf - then value is ’down’. If the first character of x is ’m’ this time the value has to be selected using the sub-tree of the second child. Depending on the second character (“2:yn”) of x the output value is either “down” or “up”.

---

func SearchValue(x, g, dt)
  if $\delta^\star(q_0, x) = q$ such that $q \in F(ofg)$ **then**
    $kv \leftarrow$ GetValue(x,dt).
  else
    $kv \leftarrow nil$; {x is unknown w.r.t. the current g}
  end if
cnuf

The function SearchByMismatch() uses the automaton associated with the general data to know if the root of (the base) word can be recognized by that automaton. If the input string can be spelled out using a given position then there is a chance that the suffix of the input string be recognized using the automaton of the available suffixes (s_g), if so, then GetValue will be activated to output the output value.

---

func SearchByMismatch(x, g_{kg}, t_s)
  $pos \leftarrow MisMatchPosition(x, g_{kg})$; $s \leftarrow substr(x, pos)$. {s stands for the suffix}
  return(GetValue(s, t_s)).
cnuf

5.1 Examples

Below we illustrate the traces of LMF applied to the verbs in English and French, Azeri and Persian.

Example 6 (French): Let us consider the following phrase: "Il livre un livre." *i.e.,* He is providing a book. Suppose that we are interested in learning the morphological features of the word "livre". The current word cannot be spelled out neither using the automaton associated with the counter exceptional automaton nor with the exceptional automaton. Therefore, the automaton associated with f_g (database of regular roots in French corresponding to the first group) will be called to partially spell out the word "livre". Using function `SearchByMismatch` tell us to stop at the fourth character (from left to right). The remaining part of the current word - "e" - will then be used as the entry of the decision tree associated with the suffixes of f_g outputting the desired result: Verb+IND-PRES-1-SING, Verb+IND-PRES-3-SING, Verb+IMP-PRES-3-SING, Noun+MASC-SING and Noun+FEM-SING.

Remark 3: The reason for which it is preferable to divide the set of words (of a language) into several files, each of which containing the same syntactic category could better be illustrated using our previous example. Indeed, one could use the rules of local grammar *e.g.,* (1) pronoun+verb as in "il livre" and (2) determinant+noun, as in "un livre", for the efficient tagging purpose while learning the morphological and right features of used word in a text.

Example 7 (French): In the the following phrase: "Bush hait Saddam et vice-versa. *i.e.,* Bush hates Saddam and vice-versa." Learning the morphological features of the word "hait" is immediate because this word belongs to the exceptional data containing the verbs of 20th class.

Example 8 (English): The morphological features of the word "stood" in the following phrase: `''He stood the child''`, can also be learned immediately, because it belongs to the exceptional data *w.r.t.* the verbs in English.

Example 9 (Azeri): Like in Turkish, the order of constituents may change rather freely without affecting the grammaticality of a sentence. Due to various syntactic and pragmatic constraints, different orderings are not just stylistic variants of the canonical order. For instance, a constituent that is to be emphasized is generally placed immediately before the verb. This affects the places of all the constituents in a sentence except that of the verb:

Man	**oşhaxlara**	**ketabi**	**verdim.**	I gave the book to
I	children+DAT	book+ACC	give+P1S	the children.
Oşhaxlara	**<u>man</u>**	**ketabi**	**verdim.**	It was me who gave
children+DAT	I	book+ACC	give+P1S	the children the book.
Man	**ketabi**	**<u>oşhaxlara</u>**	**verdim.**	It was the children to
I	book+ACC	children+DAT	give+P1S	them I gave the book.

The first above sentence is an example of the canonical word order whereas in the second one the subject, `man`, is emphasized. Similarly, in the last one the direct object, `oşhaxlara`, is emphasized.

Remark 4: Although, Azeri has some similarity with old Turkish, but their structures differ in several aspects, notably *w.r.t.* new Turkish. This is particularly true for the the vocabularies and the morphology. All together, this makes the processing of Azeri different from Turkish, including our learning process.

Example 10 (Persian): If we concern ourselves with the unmarked order of constituents, like in Turkish and Azeri, Persian can be characterized as a subject-object-verb language: (a) "Man be baçeha ketab ra dadam." (*i.e.,* I gave the book to the children.) and (b) "Lazat bordand." (*i.e.,* (They) enjoyed). In (a) the morphological features of the verb "dadam" is determined by what we call the counter exceptional data whereas in (b) the segment "Lazat (adjective) bordan (verb)" have to be considered as a compound verb. So, the combination of the morphological features of two words would determine the morphological feature of the mentioned segment.

6 Concluding Remarks

LMF is written in C and applied for learning of the large set of the verbs in French and very limited ones in Persian and Azeri. The experiments show that combing the closed world assumption, the automata and the decision trees is a good approach since our tests provide the right results for more than half million verbs - including the conjugated form - in French. Note that the transducers [8], as the the best available method, have been used in the morphology world. However, the advantages of combing the automata with the decision trees are that it leads to compact representations than transducers, and the decision trees can easily synthesize by machine learning techniques. This is emphasized in this work by Figure 2.

It must be stressed that using automata is appropriate when there is `no need` for frequent updates of one or more databases. This is due to the fact that it is difficult to update quickly the automaton. However, *w.r.t.* our present work, this is not necessarily a limitation because we are dealing with static keys originated from the morphology world. From update viewpoint, using the two-trie structure of Aoe et al. [1] instead of the automata is preferred where there is the need for frequent updates. But in this case, the cost of space (number of states and transitions) is (slightly) expensive compared to the automaton.

An interesting extension is the question of addressing how to learn the regular and irregular data from pure Stringology viewpoint *i.e.,* without attaching a domain to the values of the keys. That is to say, we have to discover the axioms along with possible exceptional and/or counter exceptional ones.

Acknowledgments

I thank the anonymous referees for their constructive comments.

References

[1] Aoe, J–I., Morimoto, K., Shishibori, M., and Park, K. A trie compaction algorithm for a large set of keys. *IEEE Transaction on Knowledge and Data Engineering 8*, 3 (1996), 476–491.

[2] Daciuk, J., Mihov, S., Watson, B. W., and Watson, R. E. Incremental construction of finite-state automata. *Association for Computational Linguistics 26*, 1 (2000), 3–16.

[3] Fatholahzadeh, A. Implementation of dictionaries via automata and decision trees. Champarnaud J. M. and Maurel D. (eds.): Seventh International Conference on Implementation of Automata (CIAA02). In LCNS *Lecture notes on Computer Science*, vol. 2608. Springer, Berlin Heidelberg, (2003), 95–105.

[4] Kempe, A. Factorizations of ambiguous finite-state transducers. In *International Conference on Implementation and Application of Automata* (2000), Daley M., Eramian M., and Yu S. pre-proceeding (eds.), 157–164.

[5] McCarthy J., and Hayes, P.J. Some Philosophic problems from the standpoint of Artificial Intelligence. In *Machine Intelligence* (1969), vol. 4, Meltzer B. and Michie D. (eds), Edinburgh University Press, 463–502.

[6] Mihov, S., and Maurel, D. Direct construction of minimal acyclic sub-sequential transducers. In *International Conference on Implementation and Application of Automata* (2000), Daley M., Eramian E., and S.Yu pre-proceeding (eds.), 150–156.

[7] Mitchell, T. M. *Machine Learning*. Mc Graw-Hill, 1997.

[8] Mohri, M. On some application of finite-state automata theory to natural language. *Natural Language Engineering 2*, 1 (1996), 1–20.

[9] Mohri, M. Finite-state transducers in language and speech processing. *Computational Linguistics 23*, 2 (1997), 269–311.

[10] Mohri, M. Generic $\epsilon-$removal algorithm for weighted automata. In *International Conference on Implementation and Application of Automata* (2000), Daley M., Eramian E., and Yu S. pre-proceeding (eds.) 26–35.

[11] Quinlan, R. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[12] Reiter R. On reasoning by default. In *Reading in Knowledge Representation* (1985), Brachmann R.J. and Levesque H.J. (eds), Morghan Kaufmann, 402–410.

[13] Rozenberg G. and Salomaa A. (eds.) *Handbook of Formal Language*. Springer–Verlag, Berlin Heidelberg, 1997.

A Linear Algorithm for the Detection of Evolutive Tandem Repeats

Richard Groult*[1], Martine Léonard[1] and Laurent Mouchard†[2]

[1] LIFAR - ABISS, Faculté des Sciences, 76821 Mont Saint Aignan Cedex, France

[2] UMR 6037 - ABISS, Faculté des Sciences, 76821 Mont Saint Aignan Cedex, France and Dept. Computer Science, King's College London, London WC2R 2LS, England

e-mail: `{Richard.Groult,Martine.Leonard,Laurent.Mouchard}@univ-rouen.fr`

Abstract. We present here a linear algorithm for the detection of evolutive tandem repeats. An evolutive tandem repeat consists in a series of almost contiguous copies, every copy being similar (using Hamming distance in this article) to its predecessor and successor. From a global view point, evolutive tandem repeats extend the traditional approximate tandem repeat where each copy has to be in a neighborhood of a given model. Due to the lack of algorithms, these repeats have been discovered in genomic sequences only recently. In this article, we present a two-stage algorithm, where we first compute an array containing all the Hamming distances between candidates, then we visit this array to build a complete evolutive tandem repeat from insulated pairs of copies. Moreover, we explain how it is still consistent with the usual technique devoted to dynamic programming which consists in filling a comparison matrix and backtracking through it to find an optimal alignment.

Keywords: linear algorithm, evolutive tandem repeats, Hamming distance

1 Introduction

The notion of approximate tandem repeat is generally well-defined, from the formal view point [2, 12], it uses a consensus model, every copy participating to this repeat being very similar to the consensus. An *evolutive tandem repeat* has no need for a consensus model, the first and the last copies might be completely different but every time we are considering two successive copies participating to the repeat, they are very similar to each other: finding evolutive tandem repeats is obviously much more complicated than detecting generic tandem repeats for which usual well-known structures, such as suffix trees, can be used during a preprocessing stage [9].
Evolutive tandem repeats have been phrased by molecular biologists, for example in [4], and have been observed in real DNA sequences (see Appendix A for a complete example, detected in *A. thaliana*). In [5], we gave a formal definition of evolutive tandem repeats with jumps then we described a quadratic space and time algorithm

*Supported by a French Ministry of Research grant.

†Partially supported by Programme inter-EPST Bio-informatique and by GenoGRID (ACI GRID).

which detects all the maximal. Even if numerous models and algorithms searching for various kinds of repeats have been developed [1, 3, 10, 11, 8, 12], none of these algorithms are able to locate evolutive tandem repeats, as far as we know, we therefore designed a quadratic algorithm for their detection, it was based on the construction of two graphs and their visits.
Since we are looking for local repetitions having approximatively the average length of mini (or even micro) satellites and because we are also looking for a certain number of copies (having three or less copies in an evolutive tandem repeats is meaningless), we are here interested in searching for copies whose length may vary from 4 to 64 [6], that is usually thousands times less than the size of the sequences we are studying. We present in this article a $O((\ell_{max} - \ell_{min} + 1) \times (j_{max} - j_{min} + 1) \times |w|)$-time and $O(j_{max} - j_{min} + 1)$-space algorithm where and ℓ_{min} and ℓ_{max} (resp. j_{min} and j_{max}) are the minimal and maximal values of the length of the copies (resp. the jump between two copies) and w is the studied sequence. More precisely, since length and jump values are very small (with respect to the length of the sequence which can be counted in millions of base pairs), we still have an overall linear time-complexity. So in practice, the time complexity is in $O(C \times |w|)$, where $C \leq (61 \times (j_{max} - j_{min})$.
In section 2, we recall some basic definitions and introduce the evolutive tandem repeats. In section 3, we present the ideas of our algorithm. In section 4, we explain the connection with comparison matrices. In section 5, we present experimental results and finally, in section 6, we conclude.

2 Preliminaries

Let Σ be an alphabet and Σ^* its associated free monoid. A *word* (resp. *non empty word*) over Σ is an element of Σ^* (resp. Σ^+). The letter of a word w occurring at position i is denoted by w_i. The *length* $|w|$ of a word w is the number of letters of w, i.e. $w = w_1 \cdots w_{|w|}$. We will denote by Σ^ℓ the set of all possible words of length ℓ over Σ. We denote by $u.v$ (or simply uv) the concatenation of two words u and v. Consider $w = p.f.s$ for some $p, f, s \in \Sigma^*$. Such p, f, s are respectively *prefix*, *factor* and *suffix* of w. We denote $f = w[i, j] = w_i w_{i+1} \cdots w_{j-1} w_j$ for $1 \leq i \leq j \leq |w|$. The concatenation of n copies of u is denoted by u^n.
There exist several distances one can use for the analysis of genomic sequences. In this article, we will consider the *Hamming distance*: the Hamming distance between two words of equal length is the number of positions at which their corresponding letters differ: for $u, v \in \Sigma^\ell$, $d_H(u, v) = \text{Card}\{i \in \{1, \ldots, \ell\} \mid u_i \neq v_i\}$.

Definition 2.1 (Evolutive tandem repeat)
An *evolutive tandem repeat with jumps* (e.t.r. for short) is a tuple $(v, \varepsilon, (j_{min}, j_{max}), \ell, n, (p_i)_{1 \leq i \leq n})$ where v is a word, ε is the maximal number of errors between two consecutive copies, $[j_{min}, j_{max}]$ is the range of the length of a jump (overlap or gap between two consecutive copies) with $(j_{max} - j_{min} + 1) \leq \ell/2$, ℓ is the length of the copies, n is the number of copies, p_i are the starting positions of the copies $c_i = v[p_i, p_i + \ell - 1]$ and

$$\begin{cases} p_1 = 1,\ p_n + \ell - 1 = |v|, \\ j_{min} \leq p_{i+1} - (p_i + \ell) \leq j_{max},\ \forall i \in \{1, \ldots, n-1\}, \\ d_H(c_i, c_{i+1}) \leq \varepsilon,\ \forall i \in \{1, \ldots, n-1\}. \end{cases}$$

Example 2.1 Let consider the word $v = \underline{aaa}t\overline{aac}\underline{ag\overline{c}}\overline{gc}$.
$(v, 1, (-1, 1), 3, 4, (1, 5, 8, 10))$ is an e.t.r. with jumps: $p_1 = 1$, $p_2 = 5$ (gap), $p_3 = 8$ and $p_4 = 10$ (overlap) corresponding to $c_1 = aaa$, $c_2 = aac$, $c_3 = agc$ and $c_4 = cgc$ (see FIG. 1).

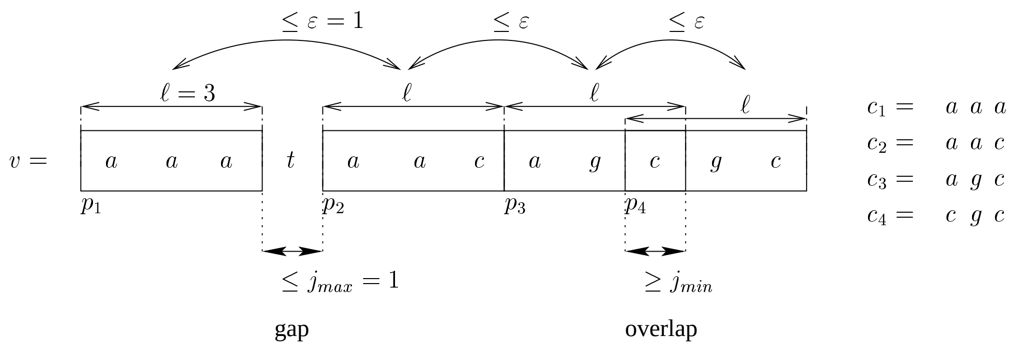


FIG. 1: Example of an evolutive tandem repeat with jumps

We will consider only in what follows maximal e.t.r., that is e.t.r. which is not embedded in a longer one: consider for example a word $w = gaaagacgaggcgg$ and $\ell = 3$. The e.t.r. $etr_1 = (\underline{aag}\overline{acg}\underline{agg}, 1, (-1, 1), 3, 3, (1, 4, 7))$ is not maximal in w since the repeat $etr_2 = (\underline{aag}\overline{acg}\underline{agg}\overline{cgg}, 1, (-1, 1), 3, 4, (1, 4, 7, 10))$ contains more copies. In this case, we say that etr_2 "contains" etr_1 and remark that etr_2 is a maximal e.t.r. in w.
In a previous article [5], we first considered all factors of w having the same length. For each factor, we computed the set of its starting positions using an equivalence relation on positions in w. Then, we built a graph for which nodes are these sets and there exists an edge between two nodes if the corresponding factors are slightly different in the meaning of the Hamming distance. Next, we computed a second graph namely the ℓ-position graph defined as follows:

Definition 2.2 (ℓ-position graph) Let w be a word and ε and $jump$ integers. The *ℓ-position graph* corresponding to w, ε and $jump$ is the oriented graph $PG_\ell(w, \varepsilon, jump) = (N, E)$ where

$$\begin{cases} N = \{1, ..., |w| - \ell + 1\} \text{ and} \\ E = \{(i, i', i' - (i + \ell)) \text{ for } (i, i') \in N \times N,\ i < i' \\ \qquad \text{such that } |i' - (i + \ell)| \leq jump, \\ \qquad d_H(w[i, i + \ell - 1], w[i', i' + \ell - 1]) \leq \varepsilon\}. \end{cases}$$

Nodes are labeled with all the positions $\{1, \ldots, |w| - \ell + 1\}$ of factors of length ℓ and there exists an edge labeled with d between two nodes if the corresponding positions are close in w and if the Hamming distance between their associated factors, denoted d is not greater than a given ε. We used a quadratic time but linear space algorithm to compute it. In what follows we denote by (i, i', d) an edge labeled d from the node i to the node i'.
Finally, we looked for all the longest paths in the ℓ-position graph to find maximal e.t.r.

3 A Linear – Time and Space – Algorithm

In a previous article [5], we described a quadratic space and time algorithm which detects all maximal e.t.r. in a word w. In what follows, we present a linear time and space algorithm that starts with the filling of a "position" array and follows on with the visit of this array in an attempt to find regularities. We will first draw the "big-picture" and will consolidate the description by explaining the structures we used and the strategies we developed.
The first important idea consists in considering every ℓ-mer (factor of length ℓ) as a sliding window. Since we have to compute the distances between pairs of factors, we have to use two sliding windows f and f' (see FIG. 2): one window, f', ending at position i will correspond to the right-most factor (moving sequentially from left to right, one position at a time) while the other window, f, will correspond to the candidates for a pair (ending at a position in the interval $[i-\ell-j_{max}, i-\ell-j_{min}]$). Therefore, we only have to consider $j_{max}-j_{min}+1$ possible positions for the left sliding window, for each given position of the right sliding window and focus on the computation of $(j_{max}-j_{min}+1)\times(|w|-\ell+1)$ distances, that is a linear-time and space construction of a "position" array (emulating the position graph we defined in [5]).

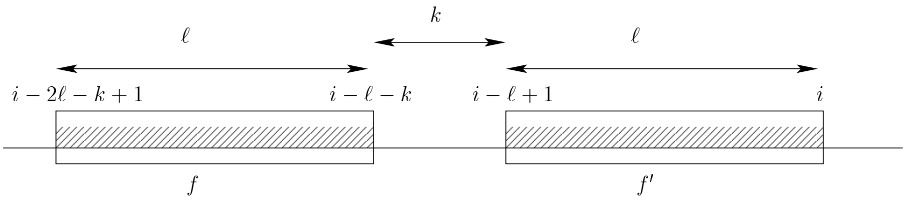


FIG. 2: The two sliding windows f and f'

The second important idea is the computation of the Hamming distance by itself: if the Hamming distance between the factors of length ℓ ending at position i and i' is known then the Hamming distance between the factors ending at position $i+1$ and $i'+1$ can be computed in $O(1)$-time because $(\ell-1)$ comparisons have already been done. It will speed up the filling of the position array (see FIG. 3).

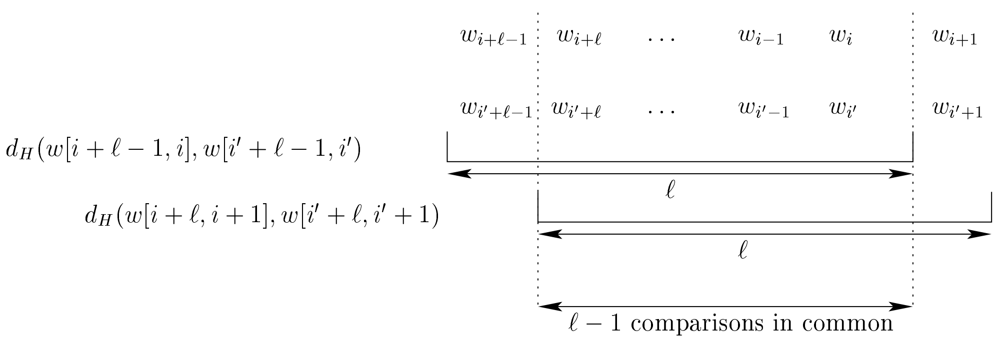


FIG. 3: Computing Hamming distance on incremental positions

Finally we only have to visit the position array and search for a series of acceptable values (smaller than ε) located at appropriate positions (the distance between two consecutive positions has to belong to $[\ell+j_{min}, \ell+j_{max}]$).

A Two-stage Algorithm

We first have to compute the Hamming distances between every possible pairs of candidates and fill the position array D that contains all these computations.

Definition 3.1 Let $w = w_1 \ldots w_n$ be a word over Σ, ℓ an integer and $k \in \{j_{min}, \ldots, j_{max}\}$. We define $D_k^{w,\ell}(i)$ by

$$D_k^{w,\ell}(i) = \begin{cases} 0, & \forall i \in \{1, \ldots, \ell + k\} \\ d_H(w[1, i-\ell-k], w[\ell+k+1, i]), & \forall i \in \{\ell+k+1, \ldots, 2\ell+k-1\} \\ d_H(w[i-2\ell-k+1, i-\ell-k], w[i-\ell+1, i]), & \forall i \in \{2\ell+k, \ldots, |w|\} \end{cases}$$

We assume now that $D_k^{w,\ell}(i-1)$ has been previously computed and we would like to compute $D_k^{w,\ell}(i)$, i.e we know $d_H(w[i-2\ell-k, i-\ell-k-1], w[i-\ell, i-1])$ and we would like to compute $d_H(w[i-2\ell-k+1, i-\ell-k], w[i-\ell+1, i])$.

We therefore define two additional functions:

- $\forall a, b \in \Sigma,\ \mathbb{1}_a(b) = 0$ if $b = a$, 1 otherwise;
- $\forall k \in \{j_{min}, \ldots, j_{max}\},\ E_k^{w,\ell}(i) = \mathbb{1}_{w_{i-\ell-k}}(w_i)$ if $i \in \{\ell+k+1, \ldots, |w|\}$, 0 otherwise.

Lemma 3.1 Let w be a word over Σ, ℓ an integer and $k \in \{j_{min}, \ldots, j_{max}\}$. We have:

$$D_k^{w,\ell}(i) = \begin{cases} 0, & \forall i \in \{1, \ldots, \ell+k\}, \\ D_k^{w,\ell}(i-1) + E_k^{w,\ell}(i), & \forall i \in \{\ell+k+1, \ldots, 2\ell+k-1\}, \\ D_k^{w,\ell}(i-1) + E_k^{w,\ell}(i) - E_k^{w,\ell}(i-\ell), & \forall i \in \{2\ell+k, \ldots, |w|\}. \end{cases}$$

Proof 1 Let $k \in \{j_{min}, \ldots, j_{max}\}$ and $i \in \{2\ell+k, \ldots, |w|\}$. If $i > 2\ell+k$ then $D_k^{w,\ell}(i-1) = d_H(w[i-2\ell-k, i-\ell-k-1], w[i-\ell, i-1])$ and therefore

$$\begin{aligned} D_k^{w,\ell}(i) & \\ &= d_H(w[i-2\ell-k+1, i-\ell-k], w[i-\ell+1, i]) \\ &= d_H(w[i-2\ell-k+1, i-\ell-k-1], w[i-\ell+1, i-1]) + \mathbb{1}_{w_{i-\ell-k}}(i) \\ &= d_H(w[i-2\ell-k, i-\ell-k-1], w[i-\ell, i-1]) - \mathbb{1}_{w_{i-2\ell-k}}(i-\ell) + \\ &\quad \mathbb{1}_{w_{i-\ell-k}}(i) \\ &= D_k^{w,\ell}(i-1) - E_k^{w,\ell}(i-\ell) + E_k^{w,\ell}(i). \end{aligned}$$

If $i = 2\ell+k$ then $D_k^{w,\ell}(i) = d_H(w[1, i-\ell-k], w[\ell+k+1, i]) = d_H(w[1, i-\ell-k-1], w[\ell+k+1, i-1]) + \mathbb{1}_{w_{i-\ell-k}}(w_i) = D_k^{w,\ell}(i-1) + E_k^{w,\ell}(i)$.
But we have $E_k^{w,\ell}(i-\ell) = E_k^{w,\ell}((2\ell+k)-\ell) = E_k^{w,\ell}(\ell+k) = 0$, so $D_k^{w,\ell}(i) = D_k^{w,\ell}(i-1) - E_k^{w,\ell}(i-\ell) + E_k^{w,\ell}(i)$.
We prove the other case in the same manner. □

The size of the arrays D (where $D[k][i] = D_k^{w,\ell}(i)$) and E (where $E[k][i] = E_k^{w,\ell}(i)$) is $(j_{max} - j_{min} + 1) \times |w|$. In order to fill these two arrays, we now use a $O((j_{max} - j_{min} + 1) \times |w|)$-time and space algorithm.

Example 3.1
This example (see FIG. 4) has been obtained with $w = aaataagttatcaatccaaatcgtgtca$, $\ell = 4$, $j_{min} = -1$, $j_{max} = 1$ and $\varepsilon = 2$:
For example $D_{-1}^{w,4}(7) = d_H(w[1,4], w[4,7]) = d_H(aaat,\ taag) = 2$, $D_0^{w,4}(17) = d_H(w[10,\ 13], w[14, 17]) = d_H(atca,\ atcc) = 1$ and $D_1^{w,4}(28) = d_H(w[20, 23], w[25, 28]) = d_H(atcg,\ gtca) = 2$.

a a a t / 1 0 0 1 = 2 / t a a g

a t c a / 0 0 0 1 = 1 / a t c c

a t c g / 1 0 0 1 = 2 / g t c a

	i	1				5					10					15					20					25			28
	w	a	a	a	t	a	a	g	t	t	a	t	c	a	a	t	c	c	a	a	a	t	c	g	t	g	t	c	a
overlap	$E_{-1}^{w,4}(i)$	0	0	0	1	0	0	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1
	$D_{-1}^{w,4}(i)$	0	0	0	1	1	1	2	2	3	4	3	3	2	2	3	3	4	4	4	4	4	4	4	3	3	3	3	4
concatenation	$E_{0}^{w,4}(i)$	0	0	0	0	0	0	1	0	1	0	1	1	1	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1
	$D_{0}^{w,4}(i)$	0	0	0	0	0	0	1	1	2	2	2	3	3	3	2	1	1	1	2	3	3	4	4	4	4	4	4	4
gap	$E_{1}^{w,4}(i)$	0	0	0	0	0	0	1	1	0	0	1	1	1	1	1	1	0	0	0	1	1	0	1	1	1	0	0	1
	$D_{1}^{w,4}(i)$	0	0	0	0	0	0	1	2	2	2	2	2	3	4	4	4	3	2	1	1	2	2	3	3	3	3	2	2

FIG. 4: D and E arrays

The space complexity can be improved as follows.
Since the values $E[k][i]$ are independent, we can decrease the space complexity by ignoring the filling of the array E and by computing $E[k][i]$ only when needed without increasing the time complexity.
Moreover, for a given ℓ, we only need the last value $D_k^{w,\ell}(i-1)$ in order to compute $D_k^{w,\ell}(i)$ (see Lemma 3.1), thus we will only store the last column of the array D. Finally (see FIG. 5), we obtain a $O((j_{max} - j_{min} + 1) \times |w|)$-time and $O(j_{max} - j_{min} + 1)$-space algorithm (D is an array of size $O(j_{max} - j_{min} + 1)$). If we are looking for all e.t.r. for copies of length $\ell \in [\ell_{min}, \ell_{max}]$,the complexity is $O((\ell_{max} - \ell_{min} + 1) \times (j_{max} - j_{min} + 1) \times |w|)$. From a practical point of view, $(\ell_{max} - \ell_{min} + 1) \leq 61$ is much lower than $|w|$ and the time complexity is still linear: $O(C \times |w|)$, where $C \leq 61 \times (j_{max} - j_{min})$.

Construction of the Longest Paths

The two arrays are compact representations of the graphs we depicted in [5], and if we refer to the traditional graph vocabulary, we can associate a cell in the position array and a node in the position graph.

CONSTRUCTION OF THE ARRAY CONTAINING THE LONGEST PATHS$(w, \ell, j_{min}, j_{max}, \varepsilon)$

for $\ell \leftarrow \ell_{min}$ **to** ℓ_{max} **do**
  for $i \leftarrow 1$ **to** $|w|$ **do**
    $C[i] \leftarrow -1$
    $L[i] \leftarrow 0$
    for $k \leftarrow j_{min}$ **to** j_{max} **do**
      if $(i \leq \ell + k)$ **then**
        $D[k] \leftarrow 0$
      elseif $(i \leq 2\ell + k)$ **then**
        $D[k] \leftarrow D[k] + \mathbb{1}_{w_{i-\ell-k}}(w_i)$
      else $D[k] \leftarrow D[k] + \mathbb{1}_{w_{i-\ell-k}}(w_i) - \mathbb{1}_{w_{i-2\ell-k}}(w_{i-\ell})$
      if $(i \geq 2\ell + k)$ and $(D[k] \leq \varepsilon)$ and $(L[i - 2\ell - k + 1] + 1 > L[i - \ell + 1])$ **then**
        $L[i - \ell + 1] \leftarrow L[i - 2\ell - k + 1] + 1$
        $C[i - \ell + 1] \leftarrow i - 2\ell - k + 1$
return (C, D)

FIG. 5: Construction of the array containing the longest paths

When $D_k^{w,\ell}(i) \leq \varepsilon$ and $i \geq 2\ell + k$, the arc between nodes $(i - 2\ell - k + 1)$ and $(i - \ell + 1)$ is added only if it creates a longest path to node $(i - \ell + 1)$, moreover the previously

existing, previously unique arc ending in $i-\ell+1$ is removed: let a path of length c ending in $(i-\ell+1)$, if the length of the path ending in $(i-2\ell-k+1)$ plus 1 is greater than c, then thearc ending in $(i-\ell+1)$ is removed and the arc from $(i-2\ell-k+1)$ to $(i-\ell+1)$ is created.
Finally each node i has at most one arc ending in i and therefore the ℓ-position graph is stored in an array C of integers, where $C[i]$ is the index of the head of the arc $(C[i], i)$, and -1 otherwise. We use an array L of integers, where $L[i]$ is the length of the longest path ending in i.
Let C and L be arrays of integers of size $|w|$ (see algorithm FIG. 5).
The determination of the longest paths, corresponding to the maximal e.t.r., uses the traditional algorithm.

Computation of the Distance between Two Factors of Length $\ell+1$

Lemma 3.2 (Computation of $D_k^{w,\ell+1}(i)$) Let ℓ, j_{min}, j_{max} and k be integers. We have $\forall k \in \{j_{min}, \dots, j_{max}\}$, $i \in \{2\ell+k, \dots, |w|\}$, $D_k^{w,\ell+1}(i) = D_{k+1}^{w,\ell}(i) + E_{k+1}^{w,\ell}(i-\ell)$, (see FIG. 6).

Proof 2 Let $\ell, j_{min}, j_{max}, i$ and k integers such that $k \in \{j_{min}, \dots, j_{max}\}$ and $i \in \{2\ell+k, \dots, |w|\}$. We have

$$\begin{aligned} D_k^{w,\ell+1}(i) &= d_H(w[i-2(\ell+1)-j+1, i-(\ell+1)-k], w[i-(\ell+1)+1, i]) \\ &= d_H(w[i-2\ell-k-1, i-\ell-k-1], w[i-\ell, i]) \\ &= d_H(w[i-2\ell-k, i-\ell-k-1], w[i-\ell+1, i]) + \mathbb{1}_{w_{i-2\ell-k-1}}(w_{i-\ell}) \\ &= d_H(w[i-2\ell-(k+1)+1, i-\ell-(k+1)], w[i-\ell+1, i]) + \\ &\qquad \mathbb{1}_{w_{i-2\ell-k-1}}(w_{i-\ell}) \\ &= D_{k+1}^{w,\ell}(i) + E_{k+1}^{w,\ell}(i-\ell). \end{aligned}$$

□

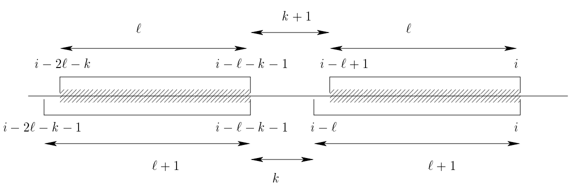


FIG. 6: Computation of $D_k^{w,\ell+1}(i)$

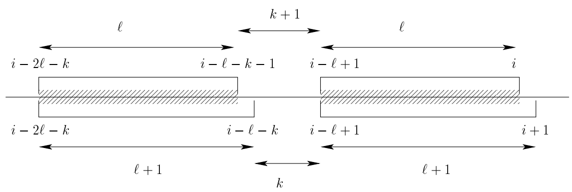


FIG. 7: Computation of $D_k^{w,\ell+1}(i+1)$

Lemma 3.3 (Computation of $D_k^{w,\ell+1}(i+1)$) Let ℓ, j_{min} and j_{max} be integers. We have $\forall k \in \{j_{min}, \dots, j_{max}\}$, $i \in \{2\ell+k, \dots, |w|\}$ $D_k^{w,\ell+1}(i+1) = D_{k+1}^{w,\ell}(i) + E_{k+1}^{w,\ell}(i+1)$, (see FIG. 7).

Proof 3 According to Lemma 3.2, $D_k^{w,\ell+1}(i+1) = D_{k+1}^{w,\ell}(i+1) + E_{k+1}^{w,\ell}(i-\ell+1)$ and by Definition 3.1, $D_{k+1}^{w,\ell}(i+1) = D_{k+1}^{w,\ell}(i) - E_{k+1}^{w,\ell}(i-\ell+1) + E_{k+1}^{w,\ell}(i+1)$, therefore, $D_k^{w,\ell+1}(i+1) = D_{k+1}^{w,\ell}(i) + E_{k+1}^{w,\ell}(i+1)$.

□

Lemma 3.4 (Computation of $D_k^{w,\ell+1}(i)$) Let ℓ, j_{min} and j_{max} be integers. We have $\forall k \in \{j_{min}, \ldots, j_{max}\}$, $i \in \{2\ell + k, \ldots, |w|\}$

$$\begin{aligned} D_k^{w,\ell+1}(i) &= D_{k+1}^{w,\ell}(i) + E_{k+1}^{w,\ell}(i-\ell) \\ &= D_{k+1}^{w,\ell}(i-1) + E_{k+1}^{w,\ell}(i). \end{aligned}$$

4 Evolutive Tandem Repeats and Comparison Matrices

Comparison Matrices

We will now explain the connection between the arrays we are computing and using, and well-known techniques used by several algorithms devoted to sequence comparison.

A traditional technique in sequence comparison consists in the construction and the visit of the two-dimension matrix, where a cell (i, i') contains the comparison score, i.e. the distance, between a factor ending at position i in one sequence and a factor ending at position i' in the other sequence.

Computing the positions of all the approximate repeats in one sequence can be carried out by comparing the sequence with itself, that is by constructing a specific symmetric square matrix, like the one we are presenting in Fig. 8. Note that Fig. 9 represents the arrays D and E corresponding to the three white diagonals of Fig. 8.

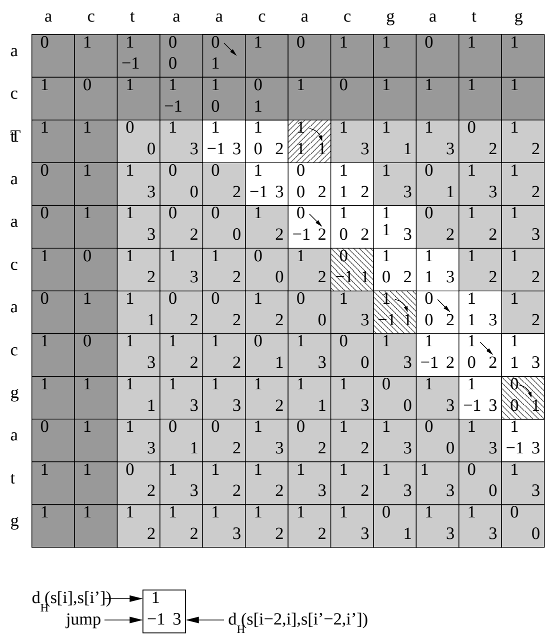


Fig. 8: Matrix and its diagonals for $\ell = 3$, $j_{min} = -1, j_{max} = 1$ and $\varepsilon = 1$

i	1				5					10		
w	a	c	t	a	a	c	a	c	g	a	t	g
$E_{-1}^{w,3}(i)$	0	0	1	1	1	1	0	0	1	1	1	1
$D_{-1}^{w,3}(i)$	0	0	1	2	3	3	2	1	1	2	3	3
$E_0^{w,3}(i)$	0	0	0	0	1	1	0	1	1	0	1	0
$D_0^{w,3}(i)$	0	0	0	0	1	2	2	2	2	2	2	1
$E_1^{w,3}(i)$	0	0	0	0	0	0	1	1	1	1	1	1
$D_1^{w,3}(i)$	0	0	0	0	0	0	1	2	3	3	3	3

Fig. 9: The arrays D and E corresponding to the three white diagonals

In this matrix, the content of a cell (i, i') contains informations corresponding to $d_H(w[i-2, i], w[i'-2, i'])$. One can observe four different kinds of cells: dark gray cells correspond to undefined distances ($i < \ell$ or $i' < \ell$, the factors are not long enough to compute $d_H(w[i-2,i], w[i'-2,i'])$, therefore only $d_H(w[i], w[i'])$ is reported in the upper left corner), light gray cells correspond to useless cells such that $i'-i < \ell+j_{min}$ or $i'-i > \ell+j_{max}$, white cells contain three values as expressed in FIG. 8 and are the only cells that are really needed and finally dashed cells tick copies participating to a potential e.t.r. (for example, the dashed cell $(3,7)$ states that $d_H(w[1,3], w[5,7]) \leq \varepsilon$, that is $d_H(act, aca) \leq 1$, which is correct).

Remark 4.1 Dashed cells contributing to a diagonal indicate a potential larger repeat: $(3,9)$ and $(4,10)$ (corresponding respectively to $d_H(act,\ acg) \leq 1$ and $d_H(cta,\ cga) \leq 1$) can establish the existence of a longer repeat (in this example $d_H(acta,\ acga) \leq 1$) but more generally, dashed cells (i,i') and $(i+1,i'+1)$, that is $d_H(w[i-2,i], w[i'-2,i']) \leq 1$ and $d_H(w[i-1,i+1], w[i'-1,i'+1]) \leq 1$, does not imply necessarily that $d_H(w[i-2,i+1], w[i'-2,i'+1]) \leq 1$ (consider $(6,8)$ and $(7,9)$ for example).

Assume now that we are searching for *approximate* tandem repeats of length $\ell = 3$, with an error rate $\varepsilon = 1$ and $j_{min} = -1, j_{max} = 1$, once we have built our matrix, the hunt for the repeats can be carried out by visiting one row at a time and reporting regions containing cells with a lower right value smaller than ε every at least $\ell+j_{min} = 3-1 = 2$ and at most $\ell+j_{max} = 3+1 = 4$ positions. In this matrix (see Fig. 10), if we consider the third row, one can find such cells in columns 3, 7 and 9 and therefore deduce that there exists an approximate repetition starting at position 1 and ending at position 9: as a matter of fact, *actaacacg* is an approximate tandem repeat with jumps, the letter a located at position 4 corresponds to a gap between copies $c_1 = act$ and $c_2 = aca$, the letter a locates at position 7 corresponds to an overlap between copies $c_2 = aca$ and $c_3 = acg$. This is more or less the concept Sagot and Myers used in [12] for finding microsatellites.

Evolutive Tandem Repeats

Finding evolutive tandem repeats with jumps is slightly different, the location of a copy participating to the e.t.r. depends only on the location of its predecessor, ℓ, the length of the copies and j_{min}, j_{max} the acceptable jump between two consecutive copies.
Consider a copy belonging to the e.t.r. that ends at position i, its successor must ends at a specific position (between $i+\ell+j_{min}$ and $i+\ell+j_{max}$) in the matrix, we therefore have to search for a dashed cell at positions (i,i') for $i+\ell+j_{min} \leq i' \leq i+\ell+j_{max}$. If there exists such a cell, it gives us a significant information about the way the copies are connected: if $i+\ell+j_{min} \leq i' \leq i+\ell-1$ there is an overlap of length $i+\ell-i'$ between the copies, if $i' = i+\ell$ the copies are contiguous, if $i+\ell+1 \leq i' \leq i+\ell+j_{max}$ there exists a gap of length $i'-i-\ell$ between the copies. Therefore, for every row i, we only have to consider $(j_{max} - j_{min}) + 1$ cells. In order to find e.t.r. we therefore have to compute and visit the diagonals starting in columns $i+\ell+j_{min}$ to $i+\ell+j_{max}$. That leads to computing and visiting only $O((j_{max} - j_{min} + 1) \times |w|)$ cells.
The left-most diagonal, starting in cell $(1, \ell+j_{min}+1)$, corresponds to the maximal authorized overlap, while the right-most diagonal, starting in cell $(1, \ell+j_{max}+1)$, corresponds to the maximal authorized gap. We can therefore build a matrix that sums up all these informations as depicted in FIG. 8. The three white diagonals are

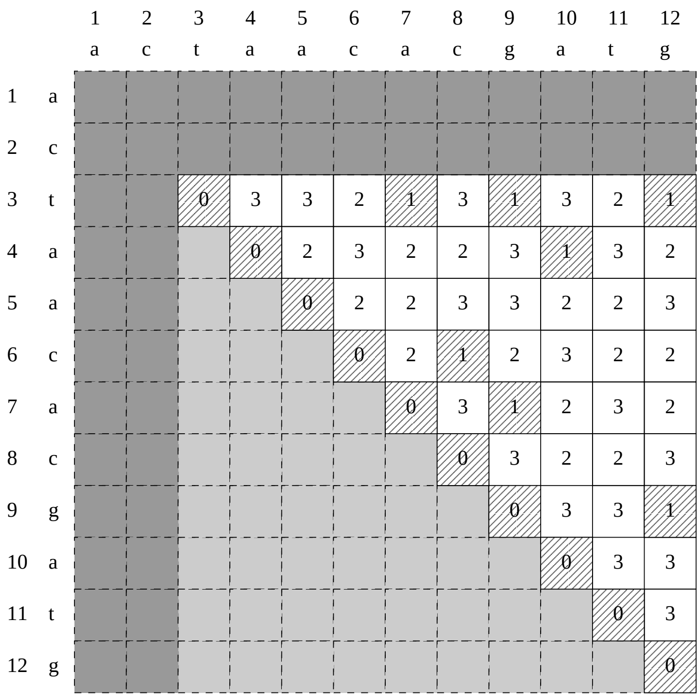


FIG. 10: Two dimension matrix corresponding to the comparison of *actaacacgatg* with itself, for $\ell = 3$ and $\varepsilon = 1$

the only ones that need to be computed (even if in this matrix, we show all the cells). Moreover, the computation of the three diagonals is equivalent to the computation of the D and E arrays.

5 Experimental Results

We have implemented and tested this algorithm on various sequences, we built random sequences over the alphabet $\{a, c, g, t\}$ and no e.t.r. has been detected (for the same rapameters as below), it appears that this kind of repetition is not an artifact. Moreover we focused on real sequences from *A. thaliana* and for testing purpose we used sequences with length varying from 10kb to 200kb (see FIG. 11).
The average behaviour of the timing curves corresponds to that we were expecting. Time and space consumptions enabled us to search for e.t.r. in whole chromosomes, we studied more specifically *A. thaliana* which possesses five chromosomes (their length varying from 17 to 29Mb) and an example is presented in Appendix A.

6 Conclusion and Perspectives

In this article, we presented a both space and time linear algorithm for the detection of evolutive tandem repeats. Furthermore, we implemented this approach, developed a web interface (see FIG. 12, `http://abiss.crihan.fr/~rgroult/index.php`) that

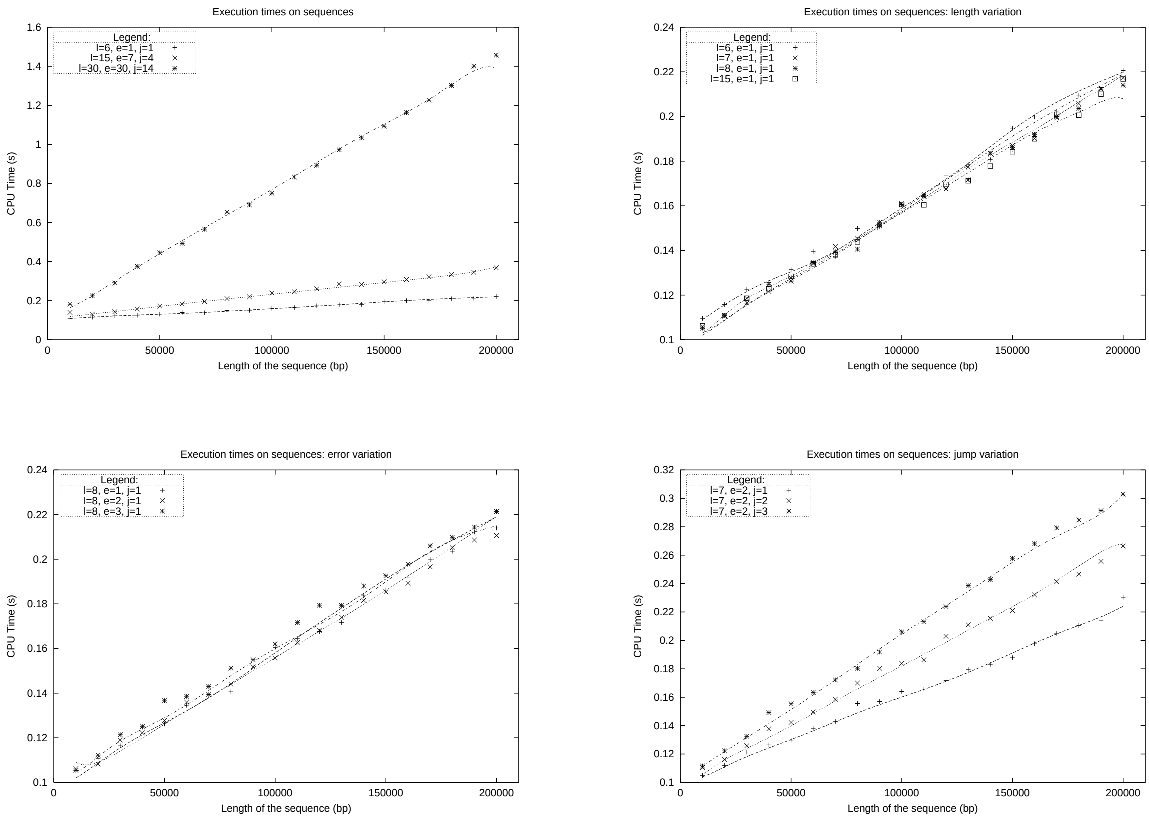


FIG. 11: Execution times on sequences, where l is the length of the copies, e is the maximal Hamming distance and j is the jump

presents the copies, the alterations and sums up informations relative to the repeats. We are now looking for this kind of repeats in complete genomes, we found several interesting e.t.r. that are not inherited from approximate tandem repeats. We are still in the process of studying the way it works, from the biologist viewpoint and we are trying to figure out their role, preferred location and number in different genomes. Since considering Hamming distance is somehow restrictive, we are moving forward by designing an algorithm that makes use of Levenshtein distance (which allows indels as well as substitution) instead of Hamming distance.

References

[1] G. Benson. An algorithm for finding tandem repeats of unspecified pattern size. In S. Istrail, P. Pevzner, and M. Waterman, editors, *Proceedings of the 2nd Annual International Conference on Computational Molecular Biology (RECOMB-98)*, pages 20–29, New York, Mar.22–25 1998. ACM Press.

[2] G. Benson. Tandem repeats finder: a program to analyze DNA sequences. *Nucleic Acids Res.*, 27(2):573–580, 1999.

[3] O. Elemento, O. Gascuel, and M.-P. Lefranc. Reconstructing the duplication history of tandemly repeated genes. *Molecular Biology and Evolution*, (19):278–288, 2002.

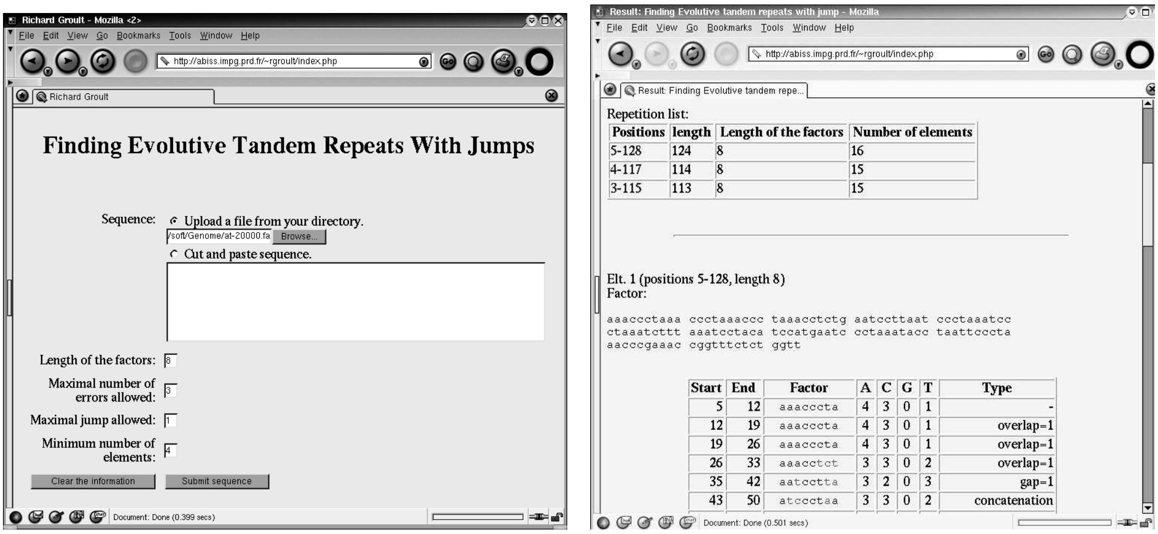


FIG. 12: HTML interface

[4] D. Golstein and C. Schlotterer. *Microsatellites: Evolution and Applications.* Oxford University Press, 1999.

[5] R. Groult, M. Léonard, and L. Mouchard. Evolutive tandem repeats using hamming distance. In *Proceedings of the 27th International Symposium on Mathematical Foundations of Computer Science*, pages 292–304, Warszawa - Otwock, Poland, Aug. 2002. Lecture Notes in Computer Science 2420, K. Diks, W. Rytter (Eds.), Springer.

[6] A. Jeffreys. Higly variable minisatellites and DNA fingerprints. *Biochem. Soc. Trans.*, 15:309–317, 1987.

[7] R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *IEEE Symposium on Foundations of Computer Science*, pages 596–604, 1999.

[8] R. M. Kolpakov and G. Kucherov. Finding approximate repetitions under hamming distance. In *Proceedings of the 9th European Symposium on Algorithms (ESA 2001)*, volume 2161 of *Lecture Notes in Computer Science*, pages 170–181, Aarhus, Denmark, 2001.

[9] S. Kurtz, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. G. Computation and visualization of degenerate repeats in complete genome. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology*, pages 228–238, La Jolla, California, 2000. The AAAI Press.

[10] S. Kurtz and C. Schleiermacher. Reputer - fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15(5), 1999.

[11] A. Lefebvre and T. 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.

[12] M. Sagot and E. W. Myers. Identifying satellites in nucleic acid sequences. In S. Istrail, P. Pevzner, and M. Waterman, editors, *Proceedings of the 2nd Annual International Conference on Computational Molecular Biology (RECOMB-98)*, pages 234–242, New York, Mar.22–25 1998. ACM Press.

A An Example of e.t.r. Occurring in *A. thaliana*, chr 4 (17Mb)

We found numerous e.t.r. in chr 4 (17Mb) of *A. thaliana*, here is an example appearing in an exon of the `AT4G38590.1` gene.

```
./evorep  -m11 -e3 -j1 -r4 -f ~/at4.fasta
->
  - number of e.t.r.: 662
  -            time: 0m38.758s

Example of found e.t.r.
#=================================================
# Parameters: length=11, error=3, jmin=-1, jmax=1, rMin=4
# Sequence: > at4.seq (17Mb)
# Execution time: 38 sec.

17245698 17245709 17245719 17245731 17245743 17245755 17245767
acaagatgagaagaagaagaaagaagataaagacgaagaggaagaggacgatgaagatgatgatgaagaagaag
                                                                   [ aagaag
17245698        acaagatgaga
17245709        agaagaagaaa
17245719        agaagataaag
17245731        cgaagaggaag
17245743        ggacgatgaag
17245755        tgatgatgaag
17245767        agaagaagaag
#=================================================
```

We investigated this sequence using “tandem repeat finder” [2] and “mreps” [7] and obtained:

```
->
Tandem Repeat Finder:
Indices Period Copy   Consensus Percent Percent Score A C G T Entropy(0-2)
        Size   Number Size Matches Indels
No Repeats Found!

->
./mreps  -err 3 -minp 2 -from 1 -exp 3.0
* Processing window [1 : 80] *

from -> to : size <per.> [exp.]  repetition
-------------------------------------------------
  1  -> 18 : 18   <5>    [3.60]  acaag atgag aagaa gaa
```

```
 5  -> 25 : 21   <6>    [3.50]  gatgag aagaag aagaaa gaa
 8  -> 40 : 33   <4>    [8.25]  gaga agaa gaag aaag aaga taaa gacg aaga g
10  -> 32 : 23   <7>    [3.29]  gaagaag aagaaag aagataa ag
11  -> 33 : 23   <5>    [4.60]  aagaa gaaga aagaa gataa aga
20  -> 80 : 61   <6>    [10.17] aaagaa gataaa gacgaa gaggaa gaggac gatgaa
                                            [ gatgat gatgaa gaagaa gaagaa g
30  -> 80 : 51   <9>    [5.67]  aagacgaag aggaagagg acgatgaag atgatgatg
                                            [ aagaagaag aagaag
30  -> 80 : 51   <12>   [4.25]  aagacgaagagg aagaggacgatg aagatgatgatg
                                            [ aagaagaagaag aag
36  -> 47 : 12   <4>    [3.00]  aaga ggaa gagg
60  -> 80 : 21   <4>    [5.25]  atga tgaa gaag aaga agaa g
----------------------------------------------------
RESULTS: There are 10 maximal repetitions in the segment processed
```

Computing the Repetitions in a Weighted Sequence

Costas S. Iliopoulos[1], Laurent Mouchard[2], Katerina Pedikuri[3,4] and Athanasios K. Tsakalidis[3,4]

[1] Department of Computer Science, King's College London Strand, London WC2R 2LS, England
e-mail: csi@dcs.kcl.ac.uk

[2] ABISS, Atelier Biology, Informatics, Statistics and Sociolinguistics, Université de Rouen, 76821 Mont Saint Aignan Cedex, France
e-mail: Laurent.Mouchard@univ-rouen.fr

[3] Research Academic Computer Technology Institute, 61 Riga Feraiou Str., 26221 Patras, Greece
e-mail: tsak@cti.gr

[4] Department of Computer Engineering and Informatics, University of Patras, 26500 Patras, Greece
e-mail: perdikur@ceid.upatras.gr

Abstract. We present an $O(n \log n)$ algorithm for computing the set of repetitions in a weighted sequence with probability of appearance larger than $1/k$, where k is a given constant.

1 Introduction

The key problem today in sequencing a large string of DNA is that only a small amount of DNA can be sequenced in a single read. That is, whether the sequencing is done by a fully automated machine or by a more manual method, the longest unbroken DNA substring that can be reliably determined in a single laboratory procedure is about 300 to 1000 (approximately 500) bases long [Celera1, Celera2]. A longer string can be used in the procedure but only the initial 500 bases will be determined. Hence to sequence long strings or an entire genome, the DNA must be divided into many short strings that are individually sequenced and then used to assemble the sequence of the full string. The critical distinction between different large-scale sequencing methods is how the task of sequencing the full DNA is divided into manageable subtasks, so that the original sequence can be reassembled from sequences of length 500.

Reassembling DNA substrings introduces a degree of uncertainty for various positions in a biosequence. This notion of uncertainness was initially expressed with the use of "don't care" characters denoted as "$*$". A *don't care* symbol has the property of matching with any symbol in the given alphabet. For example the string $p = AC*C*$ matches the pattern $q = A*DCT$. In some cases scientists determine the appearance of a symbol in a position of a sequence by assigning a probability of appearance for every symbol. In other words a *don't care* symbol is replaced by a list of probabilities of appearance for a set of characters. Such a sequence is called a weighted sequence. Other immediate applications in molecular biology include: using sequences containing degenerate bases, IUB codes [IUB], where a letter can replace several bases (for example, a B will represent a G, T or C and a H will represent A, T or C); using logo sequences [SS90] which are more or less related to consensus: either from assembly or from blocks obtained by a multiple alignment program.

In this paper we present an efficient algorithm for computing all possible repetitions of primitive words in a weighted sequence. The structure of the paper is as follows. In Section 2 we give all the basic definitions used in the rest of the paper, in Section 3 we present our algorithm while in Section 4 we give a brief time complexity analysis of the proposed method. Finally in Section 5 we conclude and discuss our research interest in open problems of the area.

2 Background

A lot of work has been done for identifying the repetitions in a word. In [Cro81], [Apo83], [Mai84] and [Sto98], authors have presented efficient methods that find occurrences of squares in a string of length n in time $O(nlogn)$ plus the time to report the detected squares. Moreover in [Kol99a] and [Kol99b] authors presented efficient algorithms to find maximal repetitions in a word. In the area of computational biology, algorithms for finding identical repetitions in biosequences are presented in [Kur99], [Tsu99] and [Mar83]. In this section we will give all the basic definitions used in the paper.

2.1 Basic Definitions

Let Σ be a finite alphabet which consists of a set of characters (or symbols). The cardinality of an alphabet denoted by $|\Sigma|$ expresses the number of distinct characters in the alphabet. A *string* or *word* is a sequence of zero or more characters drawn from an alphabet. The set of all words over the alphabet Σ is denoted by Σ^+. A word w of length n is represented by $w[1..n] = w[1]w[2]\cdots w[n]$, where $w[i] \in \Sigma$ for $1 \leq i \leq n$, and $n = |w|$ is the length of w. The empty word is the empty sequence (of zero length) and is denoted by ε; we write $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. Moreover a word is said to be *primitive* if it cannot be written as v^e with $v \in \Sigma^+$ and $e \geq 2$.

A factor f of length p is said to occur at position i in the word w if $f = w[i, \cdots i + p - 1]$. In other words f is a substring of length p occurring at position i in word w.

A word has a repetition when it has at least two consecutive equal factors. More precisely, a repetition in w is defined as a triple (i, p, e) so that $w[i, \cdots i+p-1]{=}w[i + p, \cdots i + 2 * p - 1] = \cdots = w[i + (e-1) * p, \cdots i + e * p - 1]$. The integers p and e are called respectively the *period* and *exponent* of the repetition.

In the case that for a given position of a word w we consider the presence of a set of characters with a given probability of appearance each we define the sense of a weighted word w, defined as follow:

Definition 1. *A weighted word* $w = s_1 s_2 \cdots s_n$ *is a continuous set of couples* $(s, \pi_i(s))$, *where* $\pi_i(s)$ *is the probability of having the character* s *at position* i. *For every position* $1 \leq i \leq n$, $\Sigma\pi_i(s) = 1$.

For example, if we consider the DNA alphabet $\Sigma = \{A, C, G, T\}$ the word w= [(A,0.5),(C,0.25),(G,0.25),(T,0)] [(A,0),(C,1),(G,0),(T,0)] [(A,1),(C,0),(G,0),(T,0)],

represents a word having three letters: the first one is either A,C,G with respective probabilities 0.5, 0.25 and 0.25, the second one is always a C, while the third letter is necessarily an A, since its probability of presence is 1. That means that in a given biological sequence one of the following words: ACA, CCA, GCA might appear with probability 0.5, 0.25 and 0.25 each. We observe that the probability of presence of a word is the cumulative probability which is calculated by multiplying the relative probabilities of appearance of each character in every position. For the above example the probability of the word ACA to appear in positions 1 to 3 can be analyzed as follows: $\pi(ACA) = \pi_1(A) * \pi_2(C) * \pi_3(A)$=0.5*1*1=0.5. The definition of a weighted factor can be easily extended.

A weighted sequence has a repetition when it has at least two identical occurrences of a factor (weighted or not). The probability of appearance of the factor may vary according to the position it appears. In biological problems scientists are interested in discovering all the repetitions of all possible words having a probability of appearance larger than a predefined constant.

2.2 Equivalent Classes of Repetitions

In our methodology, in order to record the repetitions of all possible words we use a list $(L_p)_{p\geq 1}$ of equivalent repetitions of length p on the positions of a weighted sequence, defined as follows:

Definition 2. *Let x be a weighted sequence of length $|x|$=n; then $(i, j) \in L_p$ iff $i+p \leq n$, $j+p \leq n$ and $x_i \cdots x_{i+p-1} = x_j \cdots x_{j+p-1}$, while $\pi(x_i \cdots x_{i+p-1}) \geq 1/k$ and $\pi(x_j \cdots x_{j+p-1}) \geq 1/k$.*

So, two positions in x are equivalent when the factors of x of length p starting at i and j respectively are equal although the respective probabilities of appearance can vary. The positions of appearance of the factors as well as the respective probabilities are stored in a set of classes C^p.

Definition 3. *Let x be a weighted sequence of length $|x|$=n; then the (C_f^p) class is the ordered list of at least 2 couples $(i_f, \pi_i(f))$, which includes all positions of appearance of the factor f of length p in the weighted sequence. We exclude all couples with probability less than 1/k .*

Moreover we also define a function on the positions of x, which gives for every position the next position in the same equivalence class.

Definition 4. *$D_p(i)$ = the least integer $k > 0$, so that $(i, i + k) \in L_p$.* (If there is no such k the function is not defined).

One can easily check that any list L_{p+1} is a refinement of L_p ($L_{p+1} \leq L_p$), since list L_{p+1} contains all possible repetitions of length p that can be extended by one character. Furthermore there clearly exists a smallest integer N, $1 \leq N \leq n$, so that $L_1 \geq L_2 \cdots \geq L_N$. Thus the computation of the equivalences L_p can be done using the values of L_{p-1}, the respective classes C^{p-1} and a proper $choicefunction\ f$.

Definition 5. *A $choicefunction\ f$ is a function*

$f : \{C_1', \cdots, C_k'\} \longrightarrow \{C_1, \cdots, C_k\}$, with the properties: for any $C' \in \{C_1', \cdots, C_k'\}$ $[f(C') \subset C'$ and for any $C \in \{C_1, \cdots, C_k\} C \subset C' \Longrightarrow |C| \leq |f(C')|]$,

where $\{C_1', \cdots, C_k'\}$ and $\{C_1, \cdots, C_k\}$ the equivalence classes of L_{p-1} and L_p respectively.

So f associates to each E_{p-1} – *class* one of its E_p – *subclasses* of maximal size. Given a choice function f, each L_p class $f(C')$ is called a *big_class*; the others are called *small_classes*. By definition, all the L_1-classes are small.

Now we define a new sequence $(S_p)_{p\geq 1}$ of equivalences on the positions of x as follows:

Definition 6. $(i,j) \in S_p$ *iff for any small class* L_p*-class* C^p, $i \in C^p$ *iff* $j \in C^p$.

Lemma. For any $p \geq 1$, $(i,j) \in L_{p+1}$ iff $(i,j) \in L_p$ and $(i+1, j+1) \in S_p$.

For more information on the proof of the Lemma the reader can refer to [Cro81].

3 Computing the Repetitions

In this paper we address the problem of computing the set of repetitions in a weighted biological sequence. More formally the problem can be stated as follows:

Problem *Given a weighted sequence* X *and an integer* k *find all the repetitions of all possible words having a probability of appearance larger than* $1/k$.

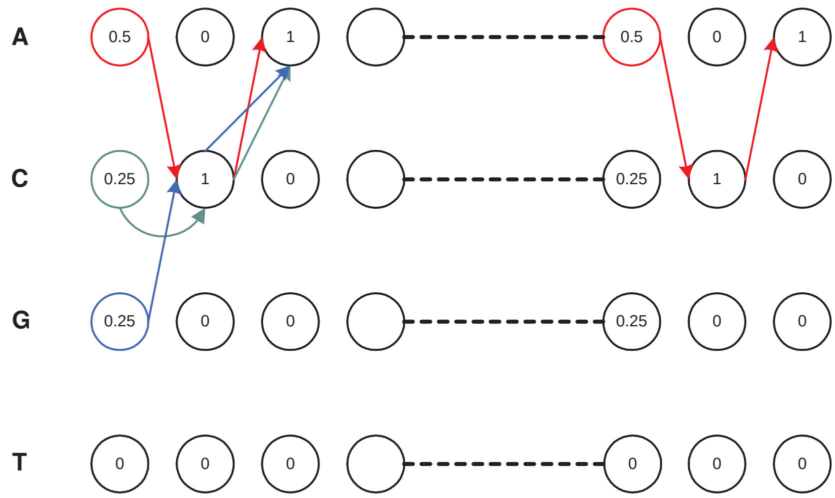


Figure 1: Graphical approach of the problem.

In a graphical approach the problem can be represented as in the Figure 1. For each position of the weighted sequence we write down the probability of appearance of each character of the alphabet. For the DNA alphabet which constitutes of 4 characters we write down 4 respective probabilities. The probability of appearance of a word is the cumulative probability calculated following the respective directed path.When the probability is larger than $1/k$, the directed path is a schema that can be extended by one character, in the following step and graphically we search for a repeated schema. In the above Figure the *red directed path* has a probability of appearance larger than $1/2$, (k=2) thus we search for such repeated schemas.

Solution. *For every character* s *in the alphabet we define a class* C^1 *as the ordered list of couples* $(i_s, \pi_i(s))$, *which includes all equivalent positions of appearance of the character* s *in the weighted sequence. We exclude all couples with probability less than* $1/k$. *The set of* C^1 *classes forms the* L_1 *list for all possible repetitions of length one. We continue by computing* D_1 *for each position in the sequence.All* L_1*-classes are small. The process is continued by computing all* C^p *classes for* $p \geq 2$

and updating L_p thus forming D_p. The process stops when we reach the maximal (in length) repeated words with probability of appearance larger than $1/k$.

The above solution uses ideas from the algorithm presented by Crochemore (see [Cro81]). The major difference is the *choice function* that we have used in order to incorporate the notion of probability of appearance in repetitions. A schema of the algorithm is presented below.

FIND-WEIGHTED REPETITIONS(X,k)
Compute all possible repetitions of any length with probability larger than $1/k$
FOR all $s \in \Sigma$ **DO**
create the small classes C^1 of couples $(s, \pi_i(s))$,
where $\pi_i(s)$ is the probability of having the character s at position i.
IF $\pi_i(s) \leq 1/k$ exclude it from the respective class
Compute for $p = 1$ L_p and D_p;
WHILE $\bigcup small\_classes \neq 0$, **DO**
report the repetitions of period p.
$p \longleftarrow p + 1$; if $p > |x|/2$ return repetitions;
$L_p \longleftarrow L_p \cap S_p$; update D_p;
$small\_classes \longleftarrow$ {indices of small $L_p - classes$}
END FIND-WEIGHTED REPETITIONS

Example Suppose we want to find all repetitions of the weighted sequence: $X=$ ACTT[(A,0.5),(C,0.5)]TC[(A,0.5),(C,0.3),(T,0.2)]TTT, with probability larger than 1/4. We will illustrate the steps following the above presented algorithm.

1. For all characters $s \in \Sigma_{DNA} = \{A, C, G, T\}$ create the C^1 classes.
$C^1_A = (1_A, 1)(5_A, 0.5)(8_A, 0.5)$.
$C^1_C = (2_C, 1)(5_C, 0.5)(7_C, 1)(8_C, 0.3)$.
$C^1_G = empty$.
$C^1_T = (3_T, 1)(4_T, 1)(6_T, 1)(9_T, 1)(10_T, 1)(11_T, 1)$.

2. Define L_1 class as the union of C^1 classes and the values D_1.
$L_1 = C^1_A \cup C^1_C \cup C^1_T$.
$D_1 = \{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1\}$.

3. Since $\bigcup small\_classes \neq 0$ we will compute all possible repetitions of length $p \geq 2$, using the lemma we presented in subsection 2.2.
$C^2_{AT} = (5_{AT}, 0.5)(8_{AT}, 0.5)$.
$C^2_{CT} = (2_{CT}, 1)(5_{CT}, 0.5)(8_{CT}, 0.3)$.
$C^2_{TC} = (4_{TC}, 0.5)(6_{TC}, 1)$.
$C^2_{TT} = (3_{TT}, 1)(9_{TT}, 1)(10_{TT}, 1)$.

4. Define L_2 class as the union of C^2 classes and the values D_2.
$L_2 = C^2_{AT} \cup C^2_{CT} \cup C^2_{TC} \cup C^2_{TT}$.
$D_2 =$ {not defined, 1, 1, 1, 1, 2, not defined, 1, 1, not defined, not defined}.

5. Following the above procedure we conclude that the repetitions with probability larger than 1/k are:.
$L_3 = C^3_{CTT} = (2_{CTT}, 1)(8_{CTT}, 0.3)$

Theorem The above algorithm computes all repetitions in a weighted sequence X of length $|n|$.

Proof. It is easy to see that the algorithm stops. The length of L_1 in the algorithm is bounded by $O(|\Sigma|^{|X|})$. As far as it concerns the values of the list L_p for $p \geq 2$, are computed using the Lemma in subsection 2.2 and the values of L_{p-1} list. Each list of repetitions $p+1$ is at most half the size of the list of repetitions of length p.

4 Time Complexity Analysis

The time complexity analysis of our algorithm is based on the combination of the following two facts:

1. The well known "smaller-half trick" used also in [Cro81], [Apo83], [Sto98], for finding tandem repeats. According to the "smaller-half trick" each list of repetitions of length $p+1$ is at most half the size of the list of repetitions of length p.

2. The probability of existence of a factor f in a weighted sequence X is the cumulative probability which is calculated by multiplying the relative probabilities of appearance of each character/symbol in every position. Note that we interested in repetitions with probability greater than $1/k$. It is not difficult to see that given a position i of x, then there is only a constant number of different substrings that can occur at position i with probability greater than $1/k$. (The proof follows).

   For every weighted sequence w of length n, $w[1..n] = w[1]w[2]\cdots w[n]$, each position $w[i]$ for $1 \leq i \leq n$, is the starting position of a weighted $factor$ iff the respective character s has $\pi(s_i) \geq 1/k$. Therefore the maximum probability of appearance for the rest of the characters in position i is bounded by $p = 1 - 1/k$. Assume that the number of starting positions inside a weighted factor, produced from position i is l. In order this $factor$ to be interesting its probability of appearance must be grater than $1/k$. This is mathematically formulated as follows:

   $p^l \geq 1/k \longrightarrow l \leq log_p(k)$.

   That means that the number of weighted positions inside a *weighted factor* is bounded by a constant and thus the number of different substrings that can occur at position i with probability greater than $1/k$ is also a constant number.

Based on the above two facts the time complexity of our algorithm for computing the set of repetitions in a weighted sequence with probability of appearance larger than $1/k$ is $O(n \log n)$.

5 Conclusions

Our future direction is focused on defining the notion of borders for a weighted sequence and developing efficient algorithms for computing the covers and the seeds of weighted sequences.

Moreover we are studying the same problem using the suffix tree as the fundamental data structure. The basic idea behind this approach is to incorporate the notion of probability of appearance in the path labels and in the leaves in the suffix tree of a weighted sequence [Ili03].

Another potential application of our algorithm is in defining a basis for the repeated motifs of a weighted sequence. In our algorithm we create in an exhaustive way all possible repetitions with probability larger than $1/k$. We can use all primitive repetitions and a set of allowed operations in order to define a basis that efficiently produces all repeated motifs. As any repeated word can be expressed as an array of primitive repetitions, it is often desirable to find only primitive repetitions.

References

[Cro81] Crochemore, M.: An Optimal Algorithm for Computing the Repetitions in a Word. Information Processing Letters, Vol.12 (5), (1981) 244-250.

[Celera1] Celera Genomics: The Genome Sequence of Drosophila melanogaster, Science 287, (2000) 2185-2195

[Celera2] Celera Genomics: The Sequence of the Human Genome, Science 291, (2001) 1304-1351.

[IUB] Nomenclature Committee of the International Union of Biochemistry (NC-IUB). Nomenclature for incompletely specified bases in nucleic acid sequences, Eur. J. Biochem. 150(1985) 1-5.

[SS90] Schneider T. D., Stephens R. M.: Sequence Logos: A New Way to Display Consensus Sequences, Nucleic Acids Res. 18, (1990) 6097-6100.

[Knu77] Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings, SIAM J. Comput., (6), (1977) 322-350.

[Apo83] Apostolico, A., Preparata, F.P.,: Optimal off-line detection of repetitions in a string. Theoretical Computer Science, (22), (1983) 297-315.

[Mai84] Main, M.G., Lorentz, R.J.,: An $O(nlogn)$ algorithm for finding all repetitions in a string. Journal of Algorithms, (5), (1984) 422-432.

[Sto98] Stoye, J., Gusfield, D.,: Simple and flexible detection of contiguous repeats using a suffix tree. In proceedings of the 9th Annual Symposium on Combinatorial Pattern matching (CPM), volume 1448 of Lecture Notes in Computer Science, (1998) 140-152.

[Kol99a] Kolpakov, R., Kucherov, G.,: Finding maximal repetitions in a word in linear time. Proceedings of IEEE Foundations of Computer Science, (1999).

[Kol99b] Kolpakov, R., Kucherov, G.,: On maximal repetitions in words. Proceedings of Foundamentals of Computation Theory, (1999) 374-385.

[Mar83] Martinez, H.,: An Efficient Method for Finding Repeats in Molecular Sequences. Nucleic Acid Research, (11), (1983) 4626-4634.

[Tsu99] Tsunoda, T., Fukagawa, M., Takagi, T.,: Time and memory efficient algorithm for extracting palindromic and repetitive subsequences in nucleic acid sequences. Pacific Symposium on Biocomputing, (4), (1999) 202-213.

[Kur99] Kurtz, S., Schleiermacher, C.,: REPuter: fast computation of maximal repetas in complete genomes. Bioinformatics, (15), (1999) 426-427.

[Ili03] Iliopoulos, C., Makris, Ch., Panagis, I., Perdikuri, K., Theodoridis, E., Tsakalidis, A.,: Computing the Repetitions in a Weighted Sequence using Weighted Suffix Trees. European Conference On Computational Biology (ECCB 2003), (accepted).

Matching Numeric Strings under Noise

Veli Mäkinen[1*], Gonzalo Navarro[2†], and Esko Ukkonen[1*]

[1] Department of Computer Science, P.O Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland.
e-mail: {vmakinen,ukkonen}@cs.helsinki.fi

[2] Center for Web Research, Department of Computer Science, University of Chile
Blanco Encalada 2120, Santiago, Chile.
e-mail: gnavarro@dcc.uchile.cl

Abstract. *Numeric string* is a sequence of symbols from an alphabet $\Sigma \subseteq \mathbb{U}$, where $\mathbb{U}$ is some numerical universe closed under addition and subtraction. Given two numeric strings $A = a_1 \cdots a_m$ and $B = b_1 \cdots b_n$ and a distance function $d(A, B)$ that gives the score of the best (partial) matching of A and B, the *transposition invariant distance* is $\min_{t \in \mathbb{U}}\{d(A + t, B)\}$, where $A + t = (a_1 + t)(a_2 + t) \ldots (a_m + t)$. The corresponding *matching* problem is to find occurrences j of A in B where $d(A+t, B_{j' \ldots j})$ is smaller than some given threshold and $B_{j' \ldots j}$ is a substring of B. In this paper, we give efficient algorithms for matching numeric strings — with and without transposition invariance — *under noise*; we consider distance functions $d(A, B)$ such that symbols $a \in A$ and $b \in B$ can be matched if $|b - a| \leq \delta$, or the κ largest differences $|b - a|$ can be discarded.

Keywords: approximate matching, transposition invariance, (δ, γ)–matching

1 Introduction

Transposition invariant string matching is the problem of matching two strings when all the characters of either of them can be "shifted" by some amount t. By "shifting" we mean that the strings are sequences of numbers and we add number t to each character of one of them.

Interest in transposition invariant string matching problems has recently arisen in the field of music information retrieval (MIR) [CIR98, LT00, LU00]. In music analysis and retrieval, one often wants to compare two music pieces to test how similar they are. A reasonable way of modeling music is to consider the pitches and durations of the notes. Often the durations are omitted, too, since it is usually possible to recognize the melody from a sequence of pitches. Hence, our focus is on distance measures for pitch sequences (of monophonic music) and their computation.

We studied the computation of edit distances under transposition invariance in [MNU03]. We noticed that sparse dynamic programming is useful in transposition

*Supported by the Academy of Finland under grant 22584.
†Supported by Millenium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile.

invariant matching, and obtained e.g. an $O(mn \log\log m)$ algorithm for transposition invariant longest common subsequence problem.

In this paper, we complement our earlier results by studying "non-gapped" distance measures for numeric strings. That is, we study measures where the ith symbol of the source is matched with the ith symbol of the target. To allow some noise in the values to be compared, we study measures that either allow matching symbols that approximately match (i.e. values are within δ distance apart), or allow discarding some amount (κ) of largest differences. We show how to compute the transposition invariant *Hamming distance* under noise in $O(m \log m)$ time, and transposition invariant sum of absolute differences (SAD) and maximum absolute difference (MAD) distances under noise in $O(m + \kappa \log \kappa)$ time, where m is the length of both strings to be compared.

For the corresponding search problems we only give the trivial algorithm that repeats the distance computation at each of the n text positions. However, the upper bound obtained this way for SAD distance is in fact the same as what is known *without* transposition invariance (see [Mut95], "weighted k–mismatches problem"). We also consider the combined search problem with SAD and MAD distances, known as the (δ, γ)*–matching* problem; we give an $O(mn)$ algorithm for the transposition invariant case of this problem. Again the best known upper bound for (δ, γ)–matching without transpositions is $O(mn)$ (because of the SAD distance).

In addition to the distance-specific results we introduce a more general approach to tackle with noise; many distance measures that allow matching two characters a and b for free when $|b-a| \leq \delta$ can be computed easily once the *set of possible matches* $|\mathbb{M}^\delta| = |\mathbb{M}^\delta|(A, B) = \{(i,j) \mid |b_j - a_i| \leq \delta, a_i \in A, b_j \in B\}$ has been computed. We show how to construct this set in $O(m \log|\Sigma| + n \log|\Sigma| + |\mathbb{M}^\delta| \min(\log(\delta + 2), \log\log m))$ time, where Σ is the alphabet of the two strings to be compared. After the set $\mathbb{M}^\delta$ is constructed, Hamming and MAD distances and (δ, γ)–matching under noise can be computed in time linear in the size of the set.

In the transposition invariant case, the construction of the sets of possible matches for all relevant transpositions is useful as well (e.g. for edit distance under noise). We show how to do this in linear time in the overall size of these sets (plus some additive factors of m,n, and $\log|\Sigma|$).

Some of the results of this paper appear in a technical report [MNU02].

2 Definitions

Let Σ be a finite numerical alphabet, which is a subset of some universe $\mathbb{U}$ that is closed under addition and subtraction. Let $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ be two *numeric strings* over Σ^*, i.e. the *symbols* (*characters*) a_i, b_j of the two strings are in Σ for all $1 \leq i \leq m, 1 \leq j \leq n$. We will assume w.l.o.g. that $m \leq n$. String A' is a *substring* of A if $A' = A_{i \ldots j} = a_i \ldots a_j$ for some $1 \leq i \leq j \leq m$. String A'' is a *subsequence* of A, denoted by $A'' \sqsubseteq A$, if $A'' = a_{i_1} a_{i_2} \ldots a_{i_{|A''|}}$ for some indexes $1 \leq i_1 < i_2 < \cdots < i_{|A''|} \leq m$.

When $m = n$, the following distances can be defined. The *Hamming distance* d_{H} between strings A and B is $d_{\mathrm{H}}(A, B) = m - |\{(i,i) \mid a_i = b_i, 1 \leq i \leq m\}|$. The *maximum absolute difference distance* d_{MAD} between A and B is $d_{\mathrm{MAD}}(A, B) = \max_{1 \leq i \leq m}\{|a_i - b_i| \mid 1 \leq i \leq m\}$. The *sum of absolute differences distance* d_{SAD} between A and B is $d_{\mathrm{SAD}}(A, B) = \sum_{i=1}^{m} |a_i - b_i|$. Note that d_{MAD} is in fact the maximum metric (l_∞ norm) and d_{SAD} the Manhattan metric (l_1 norm) when we interprete A and B as points in m dimensional Euclidean space.

String A is a *transposed copy* of B (denoted by $A =^t B$) if $B = (a_1 + t)(a_2 + t) \cdots (a_m + t) = A + t$ for some $t \in \mathbb{U}$. The transposition invariant versions of the above distance measures d_* where $* \in \{\mathrm{H}, \mathrm{MAD}, \mathrm{SAD}\}$ can now be stated as $d^{\mathrm{t}}_*(A, B) = \min_{t \in \mathbb{U}} d_*(A + t, B)$.

So far our definitions allow either only exact (transposition invariant) matches between some characters ($d^{\mathrm{t}}_{\mathrm{H}}$) or approximate match between all characters ($d^{\mathrm{t}}_{\mathrm{MAD}}$ and $d^{\mathrm{t}}_{\mathrm{SAD}}$). To relax these conditions, we introduce a constant $\delta > 0$. We write $a =^\delta b$ when $|a - b| \leq \delta$, $a, b \in \Sigma$. By replacing the equality $a = b$ with $a =^\delta b$ in the definition of $d^{\mathrm{t}}_{\mathrm{H}}$, we get a more error-tolerant version of the distance; let us denote the new distance $d^{\mathrm{t},\delta}_{\mathrm{H}}$. Similarly, by introducing another constant $\kappa > 0$, we can define distances $d^{\mathrm{t},\kappa}_{\mathrm{MAD}}, d^{\mathrm{t},\kappa}_{\mathrm{SAD}}$ such that the κ largest differences $|a_i - b_i|$ are discarded.

The *approximate string matching problem*, based on the above distance functions, is to find the minimum distance between A and any substring of B. In this case we call A the *pattern* and denote it $P_{1 \ldots m} = p_1 p_2 \cdots p_m$, and call B the *text* and denote it $T_{1 \ldots n} = t_1 t_2 \cdots t_n$, and usually assume that $m << n$. A closely related problem is the *thresholded search problem* where, given P, T, and a threshold value $k \geq 0$, one wants to find all the text positions j such that $d(P, T_{j' \ldots j}) \leq k$ for some j'. We will refer collectively to these two closely related problems as the *search problem*.

Notice that searching under Hamming distance is known as the *k–mismatches problem* [Abr87, ALP01, BYG94, BYP96, GG86, LB86]. Also, a search problem related to distances d_{MAD} and d_{SAD} is known as the (δ, γ)–matching problem [CCIMP99, CILP01, CILPR02] in which occurrences j are searched for such that $d_{\mathrm{MAD}}(P, T_{j' \ldots j}) \leq \delta$ and $d_{\mathrm{SAD}}(P, T_{j' \ldots j}) \leq \gamma$.

Our complexity results are different depending on the form of the alphabet Σ. We will distinguish two cases. An *integer* alphabet is any alphabet $\Sigma \subset \mathbb{Z}$. For integer alphabets, $|\Sigma|$ will denote $\max(\Sigma) - \min(\Sigma) + 1$. A *real* alphabet will be any other $\Sigma \subset \mathbb{R}$, and then $|\Sigma|$ denotes the cardinality of Σ. For any string $A = a_1 \ldots a_m$, we will call $\Sigma_A = \{a_i \mid 1 \leq i \leq m\}$ the alphabet of A.

Last, we will need some orders for a set of pairs $P = \{(i, j)\}$, where $a_i \in A$ and $b_j \in B$. The *row order* of P is such that P is sorted first by i (in increasing order) and secondary by j (in increasing order). In *column order* P is sorted first by j and secondary by i. In *diagonal order* P is sorted first by $j - i$ and secondary by i.

3 Matching under Noise without Transposition Invariance

We will now present a general and efficient method that can be used with little modifications for solving both the k–mismatches problem and the (δ, γ)–matching problem. The time complexities will depend on the number of possible matches between pattern and text characters. A similar approach will also be used later in the transposition invariant case.

Let $\mathbb{M}^\delta(P, T) = \mathbb{M}^\delta = \{(i, j) \mid |p_i - t_j| \leq \delta\}$ be the *set of possible matches*. Let us assume that we are given $\mathbb{M}^\delta$ in diagonal order. By one traversal over $\mathbb{M}^\delta$ one can easily compute values $S(d)$ and $N(d)$ for each diagonal d, where $S(d) = \sum\{|p_i - t_j| \mid (i,j) \in \mathbb{M}^\delta, j - i = d\}$ and $N(d) = |\{(i,j) \mid (i,j) \in \mathbb{M}^\delta, d = j - i\}|$.

Given the arrays $S(0 \ldots n-m)$ and $N(0 \ldots n-m)$, one can solve various problems. For example, all values d such that $S(d) \leq \gamma$ and $N(d) = m$, correspond to a (δ,γ)–match starting at position $d+1$ of the text. Similarly, if $N(d) \geq m - k$ when computed for $\mathbb{M}^0$, then there is an occurrence starting at position $d+1$ of the text for the k–mismatches problem.

Thus we have an $O(|\mathbb{M}^\delta| + n)$ algorithm for several problems, if we just manage to construct $\mathbb{M}^\delta$ in linear time in its size.

Theorem 1 Given numeric strings P (pattern) and T (text) of lengths m and n ($m << n$), the set of possible matches $\mathbb{M}^\delta(P,T) = \{(i,j) \mid |p_i - t_j| \leq \delta\}$ can be constructed in time $O(|\Sigma| + m + n + |\mathbb{M}^\delta| \min(\log(\delta+2), \log\log m))$ on an integer alphabet, and in time $O(m \log|\Sigma| + n \log|\Sigma| + |\mathbb{M}^\delta| \min(\log(\delta+2), \log\log m))$ on a real alphabet. Within the same bounds, the set $\mathbb{M}^\delta$ can be constructed in row, column, or diagonal order.

Proof. Let us first consider the integer alphabet with $\delta = 0$. We construct an array $L(1 \ldots |\Sigma|)$, where each entry $L(c)$ stores an increasing list of all positions of P, where character c occurs. Array L can obviously be constructed by one traversal over P in $O(|\Sigma| + m)$ time. The set $\mathbb{M}^0$ can then be constructed in column order in one traversal over T by concatenating lists $L(t_1), L(t_2), \ldots L(t_n)$. The running time is $O(m + n + |\Sigma| + |\mathbb{M}^0|)$.

For $\delta > 0$, we construct the array L as above but the traversal over T is now more complicated. To construct the column j of $\mathbb{M}^\delta$ we need to merge the $2\delta+1$ lists $L(t_j - \delta), \ldots, L(t_j + \delta)$ into a single list. This merging can be done using a priority queue $\mathcal{P}$ as follows. Add the first element, say i, of each list $L(c)$ into $\mathcal{P}$ by using i as the priority and c as the key. Then repeat the following until all lists are empty: Take the element with minimum priority, say (i,c), from $\mathcal{P}$, and add the next element from list $L(c)$ into $\mathcal{P}$. Column j of $\mathbb{M}^\delta$ is constructed by inserting pair (i,j) at the end of $\mathbb{M}^\delta$ at each step. The operations on a priority queue can be supported in $O(\log(\delta+2))$ time by using some standard implementation.

Since the priority values that need to be stored are in the range $[1,m]$, we can implement the priority queue more efficiently using a data structure of van Emde Boas [vEB77]. It supports, among other operations, retrieving the smallest value, inserting a new value, and deleting the smallest value, in $O(\log\log m)$ amortized time on values in the range $[1,m]$. We can store the values i using this data structure. Then we can repeat retrieving and deleting the smallest value i until the structure is empty, adding (i,j) at the end of $\mathbb{M}^\delta$ at each step. Thus the claimed bound on the integer alphabet follows.

When the alphabet is real, we can use exactly the same procedure, expect that the array L needs to be replaced by a binary search tree. It takes $O(m \log|\Sigma|)$ time to construct this search tree. For each character of T we need to do a range query on this tree to retrieve the lists of positions that correspond to characters in range $[t_j - \delta, t_j + \delta]$. This will take $O(n \log|\Sigma|)$ time. Merging can be done similarly as in the case of an integer alphabet, so the claimed bound follows.

Finally, the set is in column order after the above construction. Other orders can be constructed easily from the column order in time $O(m + n + |\mathbb{M}^\delta|)$. $\square$

The above theorem gives e.g. an $O(|\Sigma| + m + n + |\mathbb{M}^0|)$ time solution for the k–mismatches problem on an integer alphabet. This can be $\Theta(mn)$, but in the expected case it is much smaller. An expected bound $\Theta(mn/|\Sigma|)$ is easy to prove; see e.g. [BYP96], where the above algorithm was originally proposed for the k–mismatches problem.

4 Matching under Noise and Transposition Invariance

For this section, let $\mathbb{T} = \{t_i = b_i - a_i \mid 1 \leq i \leq m\} = \{t_i\}$ be the set of transpositions that make some characters a_i and b_i match. Note that the optimal transposition does not need, in principle, to be included in $\mathbb{T}$, but we will show that this is the case for $d_{\mathrm{H}}^{\mathrm{t}}$ and $d_{\mathrm{SAD}}^{\mathrm{t},\kappa}$. Note also that $|\mathbb{T}| = O(|\Sigma|)$ on an integer alphabet and $|\mathbb{T}| = O(m)$ in any case.

4.1 Hamming Distance

Let $A = a_1 \ldots a_m$ and $B = b_1 \ldots b_m$, where $a_i, b_i \in \Sigma$ for $1 \leq i \leq m$. We consider the computation of transposition invariant Hamming distance $d_{\mathrm{H}}^{\mathrm{t},\delta}(A, B)$. That is, we search for a transposition t maximizing the size of set $\{i \mid |b_i - (a_i + t)| \leq \delta, 1 \leq i \leq m\}$.

Theorem 2 Given two numeric strings A and B, both of length m, there is an algorithm for computing distance $d_{\mathrm{H}}^{\mathrm{t},\delta}(A, B)$ in $O(|\Sigma| + m)$ time on an integer alphabet, or in $O(m \log m)$ time on a general alphabet.

Proof. It is clear that the Hamming distance is minimized for the transposition in $\mathbb{T}$ that makes the maximum number of characters match. What follows is a simple voting scheme, where the most voted transposition wins. Since we allow a tolerance δ in the matched values, t_i votes for range $[t_i - \delta, t_i + \delta]$. Construct sets $S = \{(t_i - \delta, \text{“open”}) \mid 1 \leq i \leq m\}$ and $E = \{(t_i + \delta, \text{“close”}) \mid 1 \leq i \leq m\}$. Sort $S \cup E$ into a list I using order

$$(x', y') <^{\mathrm{H}} (x, y) \quad \text{if} \quad x' < x \text{ or } (x' = x \text{ and } y' < y),$$

where “open”<“close”. Initialize variable *count* $= 0$. Do for $i = 1$ to $|I|$ if $I(i) = (x, \text{“open”})$ then *count* $=$ *count*$+1$ else *count* $=$ *count*-1. Let *maxcount* be the largest value of *count* in the above algorithm. Then clearly $d_{\mathrm{H}}^{\mathrm{t},\delta}(A, B) = m - maxcount$, and the optimal transposition is any value in the range $[x_i, x_{i+1}]$, where $I(i) = (x_i, *)$, for any i where *maxcount* is reached. The complexity of the algorithm is $O(m \log m)$. Sorting can be replaced by array lookup when Σ is an integer alphabet, which gives the bound $O(|\Sigma| + m)$ for that case. □

4.2 Sum of Absolute Differences Distance

We shall first look at the basic case where $\kappa = 0$. That is, we search for a transposition t minimizing $d_{\mathrm{SAD}}(A + t, B) = \sum_{i=1}^{m} |b_i - (a_i + t)|$.

Theorem 3 Given two numeric strings A and B, both of length m, there is an algorithm for computing distance $d^{\mathrm{t}}_{\mathrm{SAD}}(A,B)$ in $O(m)$ time on both integer and general alphabets.

Proof. Let us consider $\mathbb{T}$ as a multiset, where the same element can repeat multiple times. Then $|\mathbb{T}| = m$, since there is one element in $\mathbb{T}$ for each $b_i - a_i$, where $1 \le i \le m$. Sorting $\mathbb{T}$ in ascending order gives a sequence $t_{i_1} \le t_{i_2} \le \ldots \le t_{i_m}$. Let t_{opt} be the optimal transposition. We will prove by induction that $t_{opt} = t_{i_{\lfloor m/2 \rfloor +1}}$, that is, the optimal transposition is the median transposition in $\mathbb{T}$.

To start the induction we claim that $t_{i_1} \le t_{opt} \le t_{i_m}$. To see this, notice that $d_{\mathrm{SAD}}(A+(t_{i_1}-\epsilon),B) = d_{\mathrm{SAD}}(A+t_{i_1},B)+m\epsilon$, and $d_{\mathrm{SAD}}(A+(t_{i_m}+\epsilon),B) = d_{\mathrm{SAD}}(A+t_{i_m},B)+m\epsilon$, for any $\epsilon \ge 0$.

Our induction assumption is that $t_{i_k} \le t_{opt} \le t_{i_{m-k+1}}$ for some k. We may assume that $t_{i_{k+1}} \le t_{i_{m-k}}$, since otherwise the result follows anyway. First notice that, independently of the value of t_{opt} in the above interval, the cost $\sum_{l=1}^{k} |b_{i_l} - (a_{i_l} + t_{opt})| + \sum_{l=m-k+1}^{m} |b_{i_l} - (a_{i_l} + t_{opt})|$ will be the same. Then notice that $\sum_{l=k+1}^{m-k} |b_{i_l} - (a_{i_l} + t_{i_{k+1}} - \epsilon)| = \sum_{l=k+1}^{m-k} |b_{i_l} - (a_{i_l} + t_{i_{k+1}})| + (m-2k)\epsilon$, and $\sum_{l=k+1}^{m-k} |b_{i_l} - (a_{i_l} + t_{i_{m-k}} + \epsilon)| = \sum_{l=k+1}^{m-k} |b_{i_l} - (a_{i_l} + t_{i_{m-k}})| + (m-2k)\epsilon$. This completes the induction, since we showed that $t_{i_{k+1}} \le t_{opt} \le t_{i_{m-k}}$.

The consequence is that $t_{i_k} \le t_{opt} \le t_{i_{m-k+1}}$ for maximal k such that $t_{i_k} \le t_{i_{m-k+1}}$, that is, $k = \lceil m/2 \rceil$. When m is odd, it holds $m-k+1 = k$ and there is only one optimal transposition, $t_{i_{\lceil m/2 \rceil}}$. When m is even, one easily notices that all transpositions t_{opt}, $t_{i_{m/2}} \le t_{opt} \le t_{i_{m/2+1}}$, are equally good. Finally, the median can be found in linear time [BFPRT72]. □

To get a fast algorithm for $d^{\mathrm{t},\kappa}_{\mathrm{SAD}}$ when $\kappa > 0$ largest differences can be discarded, we need a lemma that shows that the distance computation can be incrementalized from one transposition to another. Let $t_{i_1}, t_{i_2}, \ldots, t_{i_m}$ be the sorted sequence of $\mathbb{T}$.

Lemma 4 Once values S_j and L_j such that $d_{\mathrm{SAD}}(A+t_{i_j},B) = S_j + L_j$, $S_j = \sum_{j'=1}^{j-1} t_{i_j} - t_{i_{j'}}$, and $L_j = \sum_{j'=j+1}^{m} t_{i_{j'}} - t_{i_j}$, are computed, the values of S_{j+1} and L_{j+1} can be computed in $O(1)$ time.

Proof. Value S_{j+1} can be written as

$$S_{j+1} = \sum_{j'=1}^{j} t_{i_{j+1}} - t_{i_{j'}} = \sum_{j'=1}^{j} t_{i_{j+1}} - t_{i_j} + t_{i_j} - t_{i_{j'}} = j(t_{i_{j+1}} - t_{i_j}) + S_j.$$

Similar rearranging gives

$$L_{j+1} = \sum_{j'=j+2}^{m} t_{i_{j'}} - t_{i_{j+1}} = (m-j)(t_{i_j} - t_{i_{j+1}}) + L_j.$$

Thus both values can be computed in constant time given the values of S_j and L_j (and $t_{i_{j+1}}$). □

Theorem 5 Given two numeric strings A and B both of length m, there is an algorithm for computing distance $d^{\mathrm{t},\kappa}_{\mathrm{SAD}}(A,B)$ in $O(m+\kappa\log\kappa)$ time on both integer and general alphabets. On integer alphabets, time $O(|\Sigma|+m+\kappa)$ can also be obtained.

Proof. Consider the sorted sequence $t_{i_1}, t_{i_2}, \ldots, t_{i_m}$ as in the proof of Theorem 3. Clearly the candidates for the κ outliers (largest differences) are $M(k', k'') = \{t_{i_1}, \ldots, t_{i_{k'}}, t_{i_{m-k''+1}}, \ldots t_{i_m}\}$ for some $k' + k'' = \kappa$. The naive algorithm is then to compute the distance in all these $\kappa+1$ cases: Compute the median of $\mathbb{T} \setminus M(k', k'')$ for each $k' + k'' = \kappa$ and choose the minimum distance induced by these medians. These $\kappa + 1$ medians can be found as follows: First select values $t_{\kappa+1}$ and $t_{m-\kappa}$ using the linear time selection algorithm [BFPRT72]. Then collect and sort all values smaller than $t_{\kappa+1}$ or larger than $t_{m-\kappa}$. After selecting the median $m_{0,\kappa}$ of $\mathbb{T} \setminus M(0, \kappa)$ and $m_{\kappa,0}$ of $\mathbb{T} \setminus M(\kappa, 0)$, one can collect all medians $m_{k',k''}$ of $\mathbb{T} \setminus M(k', k'')$ for $k' + k'' = \kappa$, since the $m_{k',k''}$ values are those between $m_{0,\kappa}$ and $m_{\kappa,0}$. The $\kappa + 1$ medians can thus be collected and sorted in $O(m + \kappa \log \kappa)$ time, and the additional time to compute the distances for all of these $\kappa + 1$ medians is $O(\kappa m)$. However, the computation of distances given by consecutive transpositions can be incrementalized using Lemma 4. First one has to compute the distance for the median of $\mathbb{T} \setminus M(0, \kappa)$, $d_{\mathrm{SAD}}(A + m_{0,\kappa}, B)$, and then continue incrementally from $d_{\mathrm{SAD}}(A + m_{k',k''}, B)$ to $d_{\mathrm{SAD}}(A + m_{k'+1,k''-1}, B)$, until we reach the median of $\mathbb{T} \setminus M(\kappa, 0)$, $d_{\mathrm{SAD}}(A + m_{\kappa,0}, B)$ (this is where we need the medians sorted). Since the set of outliers changes when moving from one median to another, one has to add value $t_{i_{k'}} - t_{i_m}$ to S_m and value $t_{i_m} - t_{i_{k''}}$ to L_m, where S_m and L_m are the values given by Lemma 4 (here we need the outliers sorted). The time complexity of the whole algorithm is $O(m + \kappa \log \kappa)$. On an integer alphabet, sorting can be replaced by array lookup to yield $O(|\Sigma| + m + \kappa)$. $\square$

4.3 Maximum Absolute Difference Distance

We consider now how $d^{t,\kappa}_{\mathrm{MAD}}$ can be computed. In case $\kappa = 0$, we search for a transposition t minimizing $d_{\mathrm{MAD}}(A + t, B) = \max_{i=1}^{m} |b_i - (a_i + t)|$. In case $\kappa > 0$, we are allowed to discard the k largest differences $|b_i - (a_i + t)|$.

Theorem 6 Given two numeric strings A and B both of length m, there is an algorithm for computing distance $d^{t,\kappa}_{\mathrm{MAD}}(A, B)$ in $O(m + \kappa \log \kappa)$ time on both integer and general alphabets. On integer alphabets, time $O(|\Sigma| + m + \kappa)$ can also be obtained.

Proof. When $\kappa = 0$ the distance is clearly $d^{t}_{\mathrm{MAD}}(A, B) = (\max_i\{t_i\} - \min_i\{t_i\})/2$, and the transposition giving this distance is $(\max_i\{t_i\} + \min_i\{t_i\})/2$. When $\kappa > 0$, consider again the sorted sequence $t_{i_1}, t_{i_2}, \ldots, t_{i_m}$ as in the proof of Theorem 3. Again the κ outliers are $M(k', k'')$ for some $k' + k'' = \kappa$ in the optimal transposition. The optimal transposition is then the value $(t_{i_{m-k''}} + t_{i_{k'+1}})/2$ that minimizes $(t_{i_{m-k''}} - t_{i_{k'+1}})/2$, where $k' + k'' = \kappa$. The minimum value can be computed in $O(\kappa)$ time, once the $\kappa + 1$ smallest and largest t_i values are sorted. These values can be selected in $O(m)$ time and then sorted in $O(\kappa \log \kappa)$ time, or $O(|\Sigma| + \kappa)$ on integer alphabets. $\square$

4.4 Searching

Up to now we have considered distance computation. Any algorithm to compute the distance between A and B can be trivially converted into a search algorithm for P in T by comparing P against every text window of the form $T_{j-m+1\ldots j}$. Actually, we do not have any search algorithm better than this.

Lemma 7 For distances $d_{\mathrm{H}}^{\mathrm{t},\delta}$, $d_{\mathrm{SAD}}^{\mathrm{t},\kappa}$, and $d_{\mathrm{MAD}}^{\mathrm{t},\kappa}$, if the distance can be evaluated in $O(f(m))$ time, then the corresponding search problem can be solved in $O(f(m)n)$ time.

On the other hand, it is not immediate how to perform transposition invariant (δ,γ)–matching. We show how the above results can be applied to this case.

Note that one can find in $O(mn)$ time all the occurrences $\{j\}$ such that $d_{\mathrm{MAD}}^{\mathrm{t}}(P, T_{j-m+1\ldots j}) \leq \delta$, and all the occurrences $\{j'\}$ where $d_{\mathrm{SAD}}^{\mathrm{t}}(P, T_{j'-m+1\ldots j'}) \leq \gamma$. The (δ,γ)–matches are a subset of $\{j\} \cap \{j'\}$, but identity does not necessarily hold. This is because the optimal transposition can be different for $d_{\mathrm{MAD}}^{\mathrm{t}}$ and $d_{\mathrm{SAD}}^{\mathrm{t}}$.

What we need to do is to verify this set of possible occurrences $\{j\} \cap \{j'\}$. This can be done as follows. For each possible match $j'' \in \{j\} \cap \{j'\}$ one can compute limits s and l such that $d_{\mathrm{MAD}}(P+t, T_{j''-m+1\ldots j''}) \leq \delta$ for all $s \leq t \leq l$: If the distance $d = d_{\mathrm{MAD}}(P+t_{opt}, T_{j''-m+1\ldots j''})$ is given, then $s = t_{opt} - (\delta - d)$ and $l = t_{opt} + (\delta - d)$. On the other hand, note that $d_{\mathrm{SAD}}(P+t, T_{j''\ldots j''+m-1})$, as a function of t, is decreasing until t reaches the median of the transpositions, and then increasing. Thus, depending on the relative order of the median of the transpositions with respect to s and l, we only need to compute distance $d_{\mathrm{SAD}}(P+t, T_{j''-m+1\ldots j''})$ in one of them ($t = s$, $t = l$, or $t = t_{\lceil m/2 \rceil}$). This gives the minimum value for d_{SAD} in the range $[s, l]$. If this value is $\leq \gamma$, we have found a match.

One can see that using the results of Theorems 3 and 6 with $\kappa = 0$, the above procedures can be implemented so that only $O(m)$ time at each possible occurrence is needed. There are at most n occurrences to test.

Theorem 8 Given two numeric strings P (pattern) and T (text) of lengths m and n, there is an algorithm for finding all the transposition invariant (δ,γ)–occurrences of P in T in $O(mn)$ time on both integer and general alphabets.

4.5 Set of Possible Matches Revisited

Recall that an edit distance between two strings A and B is the cost of single symbol insertions, deletions, and substitutions to convert A into B. The unit cost or *Levenshtein distance* [Lev66] assigns cost 1 to each operation. If substitutions are forbidden and other operations have cost 1 the resulting distance is related to the longest common subsequence (LCS) of A and B. See e.g. [MNU03] and the references therein (like [Sel80]) for an introduction and formal definition of these edit distances.

For the sequel, it is only necessary to know the fact [MNU03] that the above edit distances can be computed efficiently once the set of possible matches $\mathbb{M} = \{(i,j) \mid a_i = b_j, a_i \in A, b_j \in B\}$ is given. Since we gave an efficient algorithm in Sect. 3 for constructing $\mathbb{M}^\delta = \{(i,j) \mid |b_j - a_i| \leq \delta\}$ we immediatedly have algorithms for edit distances under noise; just use the *sparse dynamic programming* algorithms of [MNU03] (or others' cited therein) on $\mathbb{M}^\delta$ instead of on $\mathbb{M}$. The effect of parameter δ is that two symbols can be matched if their values are close enough. For example, the method sketched above can be used to compute the *longest approximately common subsequence* of two numeric strings.

Now we focus on the transposition invariant edit distances under noise. Let us denote the size of $\mathbb{M}^\delta$ as $r = r(A,B,\delta) = |\mathbb{M}^\delta(A,B)|$. Let us redefine $\mathbb{T}$ in this section to be the set of those transpositions that make some characters between A and B

exactly δ apart, that is $\mathbb{T} = \{b_j - a_i \pm \delta \mid 1 \leq i \leq m, 1 \leq j \leq n\}$. The match set corresponding to a transposition $t \in \mathbb{T}$ is $\mathbb{M}_t^\delta = \{(i,j) \mid |b_j - a_i - t| \leq \delta\}$. Notice that there is always some $t \in \mathbb{T}$ whose match set $\mathbb{M}_t^\delta$ is equal to $\mathbb{M}_{t'}^\delta$, where $t' \in \mathbb{U}$. For most edit distances (like Levensthtein distance or LCS) same match set means that the distance will also be the same.

As noticed in [MNU03] (in the case $\delta = 0$) one could compute the above edit distances by running the basic dynamic programming algorithms [Sel80] over all pairs $(A+t, B)$, where $t \in \mathbb{T}$. In case $\delta > 0$, one would just interpret symbols a be b the same when $|b - a| \leq \delta$. One can obtain a more efficient method using advanced algorithms at each transposition. Let us first assume that $\delta = 0$ and let $r(A,B) = r(A,B,0)$. The following connection was shown in [MNU03]:

Lemma 9 ([MNU03]) If an algorithm computes a distance $d(A,B)$ in $O(r(A,B)f(m,n))$ time, then there is an algorithm that computes the transposition invariant distance $d^{\mathrm{t}}(A,B) = \min_{t\in\mathbb{T}} d(A+t,B)$ in $O(mnf(m,n))$ time.

As a consequence of the above lemma, we have $O(mn\,\mathrm{polylog}(n))$ time algorithms for different edit distances, since we manage to construct the match sets for all transpositions in $O(mn\,\mathrm{polylog}(n))$ time [MNU03]. In our noisy case, the above lemma extends to giving an $O(\sum_{t\in\mathbb{T}} |\mathbb{M}_t^\delta| f(m,n))$ algorithm, which equals $O(mn\,\mathrm{polylog}(n))$ when $\delta = 0$. To achieve total running time $O(\sum_{t\in\mathbb{T}} |\mathbb{M}_t^\delta| f(m,n))$, we still need to show that the sets $\mathbb{M}_t^\delta$ can be constructed in linear time in their overall size.

Theorem 10 The *match sets* $\mathbb{M}_t^\delta = \{(i,j) \mid a_i + t = b_j\}$, each sorted in the column order, for all transpositions $t \in \mathbb{T}$, can be constructed in time $O(|\Sigma| + \delta mn)$ on an integer alphabet, and in time $O(m \log |\Sigma_A| + n \log |\Sigma_B| + |\Sigma_A||\Sigma_B| \log(\min(|\Sigma_A|, |\Sigma_B|)) + \sum_{t\in\mathbb{T}} |\mathbb{M}_t^\delta|)$ on a real alphabet.

Proof. (We extend the proof given in [MNU03] for the case $\delta = 0$.) On an integer alphabet we can proceed naively to obtain $O(|\Sigma| + mn)$ time using array lookup to get the transposition where each pair (i,j) has to be added. For $\delta > 0$ each pair (i,j) is added to entries from $b_j - a_i - \delta$ to $b_j - a_i + \delta$, in $O(|\Sigma| + \delta mn)$ time.

The case of real alphabets is solved as follows. Let us first consider the case $\delta = 0$. Create a balanced tree $\mathcal{T}_A$ where every character $a = a_i$ of A is inserted, maintaining for each such $a \in \Sigma_A$ a list $\mathcal{L}_a$ of the positions i of A, in increasing order, such that $a = a_i$. Do the same for B and $\mathcal{T}_B$. This costs $O(m \log |\Sigma_A| + n \log |\Sigma_B|)$. Now, create an array $R(1 \ldots |\Sigma_A||\Sigma_B|)$, where each $R(k)$ stores the subset of the match set $\mathbb{M}_{t_k}$ (in column order), where $t_k = b - a$, $b_j = b$, and $a_i = a$ for all $(i,j) \in R(k)$. There is an entry in R for each possible pair (a,b), where $a \in \Sigma_A$, $b \in \Sigma_B$. Clearly R can be constructed in $O(mn)$ time once $\mathcal{T}_A$, $\mathcal{T}_B$, and the associated lists $\mathcal{L}$ are given. However, many pairs can produce the same transposition, thus we have to (i) sort R based on values t_k and (ii) merge the partial match sets that correspond to the same transposition. Phase (i) can be implemented to run in $O(|\Sigma_A||\Sigma_B| \log(\min(|\Sigma_A|, |\Sigma_B|)))$ time; consider w.l.o.g. that $|\Sigma_A| \leq |\Sigma_B|$. For fixed $a \in \Sigma_A$, we can get the $|\Sigma_B|$ transpositions $b - a$, $b \in \Sigma_B$, in increasing order by a depth-first search on $\mathcal{T}_B$. Thus we have $|\Sigma_A|$ lists, each containing $|\Sigma_B|$ transpositions already in order. Merging these lists (using standard techniques) takes $O(|\Sigma_A||\Sigma_B| \log |\Sigma_A|)$ time. Phase (ii) can be implemented to run in $O(mn)$ time; we can traverse through B and for each b_j add a new column to each $\mathbb{M}_t$, where $b_j - a = t$, $a \in \Sigma_A$. The correct set $\mathbb{M}_t$ can be found in constant time since we can maintain suitable pointers when sorting R in phase (i).

Finally, let us consider the case where $\delta > 0$. As discussed earlier, each pair (a, b) produces two relevant transpositions, $b - a - \delta$ and $b - a + \delta$. We proceed as before until array R is constructed and sorted. Consider sliding a window of length 2δ over the transpositions t_k in R. Let the middle point of current window be at t. Clearly, the pairs that are included in the current window produce the whole match set for transposition t. That is, partial match sets $R(l), R(l+1), \ldots, R(r)$ are merged into match set $\mathbb{M}_t^\delta$, where $t_l = b_j - a_i \geq t - \delta$ for (all) $(i, j) \in R(l)$, $t_r = b_{j'} - a_{i'} \leq t + \delta$ for (all) $(i', j') \in R(r)$, and $[l, r]$ is maximal range of R where this holds. The match sets change only when the middle points of the sliding window are from set $\mathbb{T} = \{b - a \pm \delta \mid a \in \Sigma_A, b \in \Sigma_B\}$. We can construct this set in $O(|\Sigma_A||\Sigma_B|)$ time. After sorting it, we can slide the window of length 2δ stopping at each middle point $t \in \mathcal{T}$, and construct each match set $\mathbb{M}_t^\delta$ by merging the match sets in the entries of R that are covered by the current window.

What is left is to consider how the merging can be done efficiently. Notice that the match sets corresponding to consecutive transpositions share a lot in common; the merging does not have to be done by brute force. We have two cases depending on whether the consecutive match sets differ (i) only by one entry of R, or (ii) by several entries. In case (i), the range $[l, r]$ of R is changed either to $[l + 1, r]$ or to $[l, r + 1]$. Both situations can be handled by one traversal over match set corresponding to $[l, r]$ and in the latter case also over $R(r + 1)$. In case (ii), the range $[l, r]$ of R is changed either to $[l + k, r]$ or to $[l, r + k]$ for some k (by definition both ranges can not change at the same time). Let us consider the latter situation, since the first is analogous. It follows that $t_{r+1} = \cdots = t_{r+k}$, since otherwise there would be a relevant transposition $t_{r+k'} - \delta$, for some $1 < k' < k$, in between $t_r - \delta$ and $t_{r+k} - \delta$, which conflicts the fact that we are moving from one relevant transposition to the next. What follows is that we can preprocess R just like in the case when $\delta = 0$, merging consecutive entries of R having exactly the same transposition in $O(mn)$ time. After this is done, case (ii) can be handled just like case (i). The time complexity of this merging phase is bounded by $\sum_{t \in \mathbb{T}} |\mathbb{M}_t^\delta|$. □

Notice that $\sum_{t \in \mathbb{T}} |\mathbb{M}_t^\delta| \leq \delta mn$ on an integer alphabet. So the bound on a real alphabet is analogous to the bound on an integer alphabet.

5 Concluding Remarks

The motivation to study transposition invariant distances comes from music information retrieval. However, there are also other applications where these distance measures are useful. For example, in image comparison one could use the transposition invariant SAD distance to search for the occurrences of a small template inside a large image. With gray-level images the search would then be "lighting invariant". Combining other invariances, such as rotation or scaling invariance, with transposition invariance in a search algorithm, is a major challenge.

References

[Abr87] K. Abrahamson. Generalized string matching. *SIAM J. Computing*, 16(6):1039–1051, 1987.

[ALP01] A. Amir, M. Lewenstein, and E. Porat. Approximate Subset Matching with Don't Cares. In *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'01)*, pp. 279–288, 2001.

[BYG94] R. Baeza-Yates and G. Gonnet. Fast string matching with mismatches. *Information and Computation*, 108(2):187–199, 1994.

[BYP96] R. Baeza-Yates and C. Perleberg. Fast and Practical Approximate String Matching. *Information Processing Letters*, 59:21–27, 1996.

[BFPRT72] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *J. Computer and System Sciences*, 7:448–461, 1972.

[CCIMP99] E. Cambouropoulos, M. Crochemore, C.S. Iliopoulos, L. Mouchard, and Yoan J. Pinzón. Algorithms for computing approximate repetitions in musical sequences. In *Proc. 10th Australian Workshop on Combinatorial Algorithms, AWOCA'99*, R. Raman and J. Simpson, eds., Curtin University of Technology, Perth, Western Australia, pp. 129–144, 1999.

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

[CILP01] M. Crochemore, C.S. Iliopoulos, T. Lecroq, and Y.J. Pinzón. Approximate string matching in musical sequences. In *Proc. Prague Stringoly Club (PSC 2001)*, M. Baliik and M. Simanek, eds, Czech Technical University of Prague, pp. 26–36, DC-2001-06, 2001.

[CILPR02] M. Crochemore, C.S. Iliopoulos, T. Lecroq, W. Plandowski, and W. Rytter. Three Heuristics for δ–Matching: δ–BM Algorithms. In *Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM'02)*, Springer-Verlag LNCS 2373, pp. 178–189, 2002.

[GG86] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17:52–54, 1986.

[LT00] K. Lemström and J. Tarhio. Searching monophonic patterns within polyphonic sources. In *Proc. RIAO 2000*, pp. 1261–1279 (vol 2), 2000.

[LU00] K. Lemström and E. Ukkonen. Including interval encoding into edit distance based music comparison and retrieval. In *Proc. AISB 2000*, pp. 53–60, 2000.

[Lev66] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady* 6:707–710, 1966.

[LB86] G. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

[Mut95] S. Muthukrishnan. New results and open problems related to non-standard stringology. In *Proc. 6th Annual Symposium on Combinatorial Pattern Matching (CPM'95)*, LNCS 937, pp. 298–317, 1995.

[MNU02] V. Mäkinen, G. Navarro, and E. Ukkonen. Algorithms for Transposition Invariant String Matching. *TR/DCC-2002-5*, Dept. of CS, Univ. Chile, July 2002,
"ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/ti_matching.ps.gz"

[MNU03] V. Mäkinen, G. Navarro and E. Ukkonen. Algorithms for Transposition Invariant String Matching (Extended Abstract). In *Proc. 20th International Symposium on Theoretical Aspects of Computer Science (STACS 2003)*, Springer-Verlag LNCS 2607, pp. 191–202, Berlin, February, 2003.

[Sel80] P. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. of Algorithms*, 1(4):359–373, 1980.

[vEB77] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Proc. Letters* 6(3):80–82, 1977.

Operation L-INSERT on Factor Automaton*

Bořivoj Melichar and Milan Šimánek

Department of Computer Science & Engineering
Faculty of Electrical Engineering
Czech Technical University Prague

e-mail: `melichar@fel.cvut.cz`, `simanek@fel.cvut.cz`

Abstract. The factor automaton is used for time-optimal searching for substrings in text. In general, if the text is changed the new factor automaton has to be constructed. When the text change is simple enough we can change the original factor automaton to reflect the changes of the text and save the time of the new factor automaton construction.

This paper deals with operation L-INSERT and describes the algorithm modifying the factor automaton when a new symbol is prepended to the text. This algorithm can be also used for on-line backward construction of factor automaton.

Keywords: factor automaton, DAWG, operation on factor automaton, construction of factor automaton, finite automaton.

1 Introduction

The factor automaton is a finite automaton accepting the set of all factors (substrings) of the given text (string) T. The factor automaton can be constructed for arbitrary text by one of the common construction algorithms. The time complexity of the construction is linear to the size of the text T, while pattern matching for pattern P is linear to the size of the pattern P and is independent of the size of text T. So, in the most common case the factor automaton is once constructed and many time used for pattern matching. However, when we change the text T the factor automaton must be dropped and new factor automaton has to be constructed.

If the changes in the text are simple enough then we can find an algorithm modifying the original factor automaton according text T. The time complexity of this algorithm is often better then the complete construction of the new factor automaton for the changed text.

A nice example of such algorithm is the APPEND algorithm described in [1, Chapter 6.3], which can modify given factor automaton when a new symbol is appended to the text T. The authors use this algorithm as a part of their on-line factor automaton construction algorithm for text $T = t_1 t_2 \cdots t_n$: they start with one-node factor

*This research has been partialy supported by the Ministry of Education, Youth, and Sports of the Czech Republic under research program No. J04/98:212300014 (Research in the area of information technologies and cummunications) and by Grant Agency of Czech Republic grant No. 201/01/1433.

automaton for empty text ε and compute successively factor automata for texts t_1, t_1t_2, $t_1t_2t_3$, $\cdots$, $t_1t_2\cdots t_n$.

Another known factor automaton modifying algorithm is the L-DELETE algorithm [2]. It can make desired changes to the factor automaton when the text T is reduced by deleting the leftmost symbol. The L-DELETE algorithm can be used in conjunction with the APPEND algorithm to implement fast substring matching in sliding window data compression method.

This paper describes an L-INSERT algorithm modifying the factor automaton when the text T is prepended by a new symbol. Like the APPEND algorithm, this algorithm can also be used for the construction of the factor automaton. The well-known construction using operation APPEND creates the factor automaton by appending symbols of the text T from left to right. On the contrary, the construction based on L-INSERT creates the factor automaton starting with the rightmost symbol to the left.

2 Basic Definitions

The factor automaton for text T is defined as a finite automaton M accepting the language $L(M) = Fac(T)$ of all factors of T. There is an infinite number of such automata, hence we select one with very regular structure of its transition diagram (Figure 1). All its states are both initial and final.

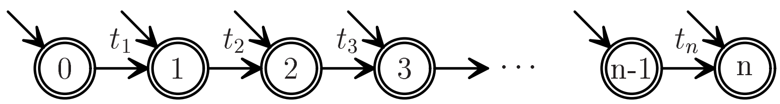


Figure 1: Canonical nondeterministic factor automaton (CNFA)

Definition 2.1 — Canonical nondeterministic factor automaton (CNFA)
Canonical nondeterministic factor automaton CNFA for text $T = t_1t_2t_3\cdots t_n$ is a nondeterministic finite automaton $M = (Q, \mathcal{A}, \delta, I, F)$ which satisfies:

1. $Q = \{q_0, q_1, q_2, \cdots q_n\}$
2. $\forall q_i \in Q, a \in \mathcal{A}: \quad \delta(q_i, a) = \begin{cases} \{q_{i+1}\} & \forall i < n, a = t_{i+1} \\ \emptyset & \text{in other cases} \end{cases}$
3. $I = Q$
4. $F = Q$

We cannot directly use CNFA because of a nondeterminism. Each nondeterministic finite automaton can be transformed to deterministic one accepting the same language. The transformation can be done by subset construction [3]. We use the variant of the transformation which does not insert inaccesible states into the resulting DFA [4, algorithm 3.6] and we denote it as the standard determinization method.

The standard determinization method is based on the following state-sets construction: For each nondeterministic finite automaton $M = (Q, \mathcal{A}, \delta, I, F)$ we can find a deterministic finite automaton $\hat{M} = (\hat{Q}, \mathcal{A}, \hat{\delta}, \hat{q}_0, \hat{F})$ accepting the same language satisfying the following conditions:

- $\hat{Q} \subseteq \mathcal{P}(Q)$ such that $\hat{Q} = \{\hat{q} : \hat{q} = \delta^*(I, w); w \in \mathcal{A}^*\}$
- $\hat{\delta}$ is a mapping $\hat{\delta} : \hat{Q} \times \mathcal{A} \mapsto \hat{Q}$
  $\forall \hat{q} \in Q, a \in \mathcal{A} : \quad \hat{\delta}(\hat{q}, a) = \bigcup_{q \in \hat{q}} \delta(q, a)$,
- $\hat{q}_0 \in \hat{Q} \quad \hat{q}_0 = I$,
- $\hat{F} \subset \hat{Q} \quad \hat{F} = \{\hat{q} \in \hat{Q} : \hat{q} \cap F \neq \emptyset\}$.

We use the hat accent to denote deterministic automaton, its states and transition function. States of CDFA are sets of CNFA. Note, that that CDFA contains only reachable states.

Definition 2.2 — Canonical deterministic factor automaton (CDFA)
Canonical deterministic factor automaton CDFA for text T is a deterministic automaton given as the result of the standard determinization of the canonical nondeterministic factor automaton for the same text T.

The L-INSERT algorithm modifying CNFA is very simple (it just inserts a new state and one transition). We use that algorithm and the standard determinization to find L-INSERT algorithm modifying CDFA. To keep the relationship between states of CNFA and CDFA automata we use several adjacent data structures.

3 Adjacent Data Structures

To enable efficient algorithm modifying CDFA we extend CDFA by following additional information:

- suffix links,
- text pointers,
- in-degree of nodes.

3.1 Suffix Links

Each state $\hat{q}$ of the CDFA represents a set of active states of the CNFA – after accepting any string w the active state $\hat{q}_w = \hat{\delta}^*(\hat{q}_0, w)$ of CDFA represents a set of active states $Q_w = \delta^*(I, w)$ of CNFA, formally $\hat{q}_w = Q_w$.

Lemma 3.1 If two states $\hat{q}_u, \hat{q}_w \in \hat{Q}$ have nonempty intersection, $\hat{q}_u \cap \hat{q}_w \neq \emptyset$, then one of them is a subset of the other ($\hat{q}_w \subset \hat{q}_u$).

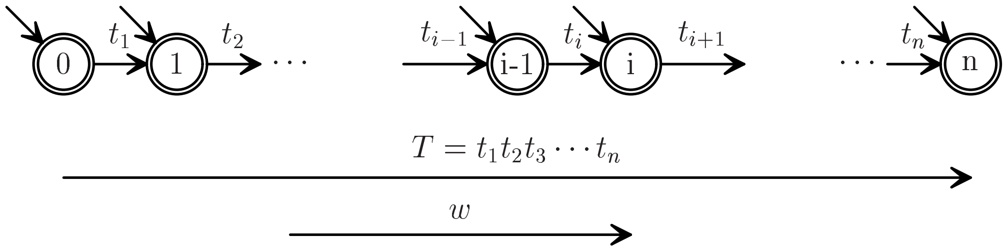


Figure 2: If state $\hat{q}_w = \hat{\delta}^*(\hat{q}_0, w)$ contains a state q_i then string w ends at position i

Proof:

If both two states $\hat{q}_u$ and $\hat{q}_w$ contain state q_i then both represent the CNFA active state q_i. Because of very regular structure of CNFA the state q_i becomes active only if the accepted string is a factor of the text T ending at position i (see Figure 2). It means that both strings u and w (leading to states $\hat{q}_u$ and $\hat{q}_w$) are factors of the text T ending on the same position i. Therefore one of them must be a suffix of the other (Figure 3). Let

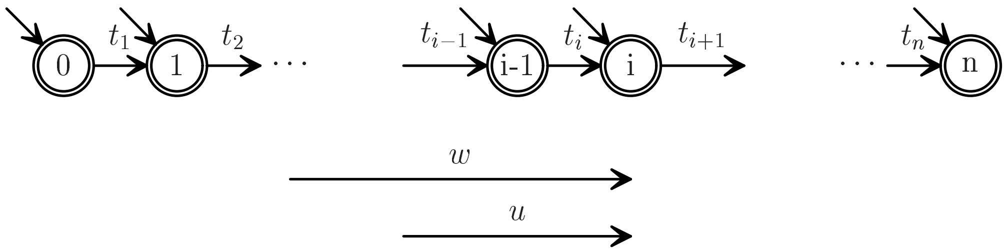


Figure 3: Strings u and w end in the same position.

u be a suffix of w. The state $\hat{q}_w$ represents states $\hat{q}_w = \{q_{j_1}, q_{j_2}, q_{j_3}, \cdots\}$ where j_k are ending positions of all occurrences of the string w in the text. The string u is a suffix of w so that it occurs at least on the same ending positions, therefore $\hat{q}_w \subset \hat{q}_u$ (Figure 4). □

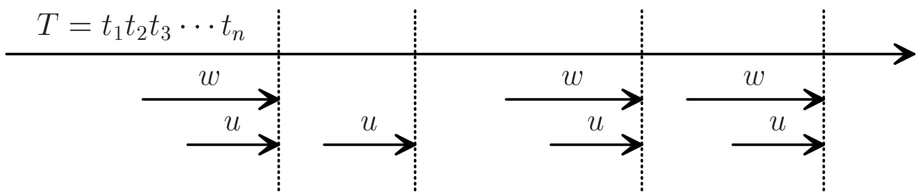
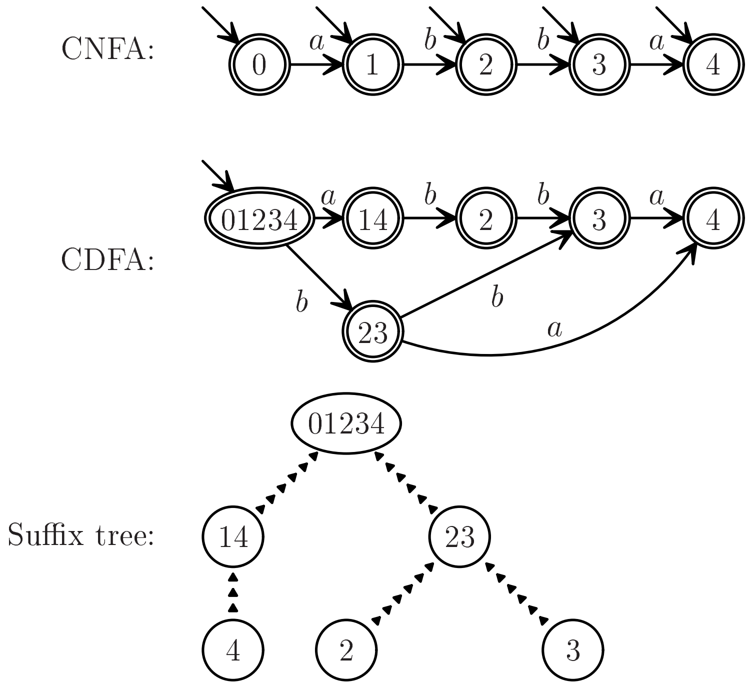


Figure 4: String u ends at least on the same ending positions as string w.

From the lemma above, any pair of CDFA states containing any common CNFA state q_i are ordered by set inclusion. Therefore all CDFA states representing any CNFA state q_i create ordered set (chain of states). The initial state $\hat{q}_0 = I = Q = \{q_0, q_1, \cdots q_n\}$ containing all CNFA states is a superset of any set of CNFA states and it is the biggest set of any chain of sets. We can say that all states of CDFA are ordered in a rooted tree with the root $\hat{q}_0$. The common name for such tree is **suffix tree**.



Positions in text T: ${}_0a_1b_2b_3a_4$

state	words	ending pos.
$\hat{q}_{\{q_0,q_1,q_2,q_3,q_4\}}$	ε	0, 1, 2, 3, 4
$\hat{q}_{\{q_1,q_4\}}$	a	1, 4
$\hat{q}_{\{q_2,q_3\}}$	b	2, 3
$\hat{q}_{\{q_2\}}$	ab	2
$\hat{q}_{\{q_3\}}$	bb abb	3
$\hat{q}_{\{q_4\}}$	ba bba $abba$	4

Figure 5: An example of suffix tree for $T = abba$

This suffix tree (as a data structure) can be implemented by pointers from each state $\hat{q} \in \hat{Q}$ to its parent $\hat{p}$ in the suffix tree. We call such pointer as **suffix link** and denote $\hat{p} = suf[\hat{q}]$. The state $suf^k[\hat{q}]$ means $k^{\underline{th}}$ iteration of suffix link and $suf^*[\hat{q}]$ (transitive closure) denotes a set of all iterations of suffix link of the state $\hat{q}$.

$$suf^*[\hat{q}] = \{\hat{q}, suf[\hat{q}], suf^2[\hat{q}], suf^3[\hat{q}], \cdots\}$$

Lemma 3.2 If two nonequal states $\hat{p}, \hat{q} \in \hat{Q}$ differ by a one state $q \in Q$ i.e. $\hat{p} = \hat{q} \cup \{q\}$ then there exists a direct suffix link between them: $\hat{p} = suf[\hat{q}]$.

Proof:

Any two states $\hat{p}, \hat{q} \in \hat{Q}$ where $\hat{q}$ is a proper subset of $\hat{p}$ ($\emptyset \subset \hat{q} \subset \hat{p}$) are connected by a suffix link iff there does not exist another state r such that $\hat{q} \subset \hat{r} \subset \hat{p}$. As states $\hat{p}$ and $\hat{q}$ differ only by one state, no such state $\hat{r}$ may exist. □

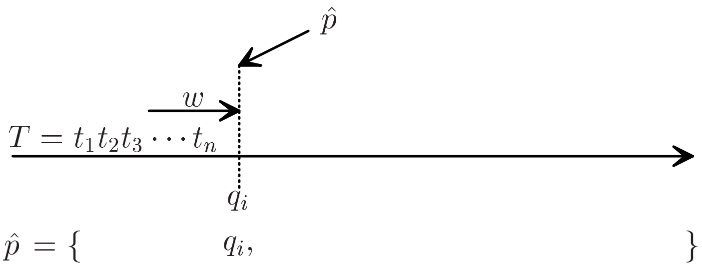


Figure 6: The state $\hat{p}$ has no incoming suffix link iff it contain only one state

Lemma 3.3 State $\hat{p} \in \hat{Q}$ has no incoming suffix link if and only if the set $\hat{q}$ contains exactly one state $q \in Q$.

Proof:

We divide the proof of equivalence to proofs of the both implications. The proof of the first implication (the state $\hat{p}$ has no incoming suffix link $\Longrightarrow$ the set $\hat{p}$ contains only one state) follows from this contradiction:

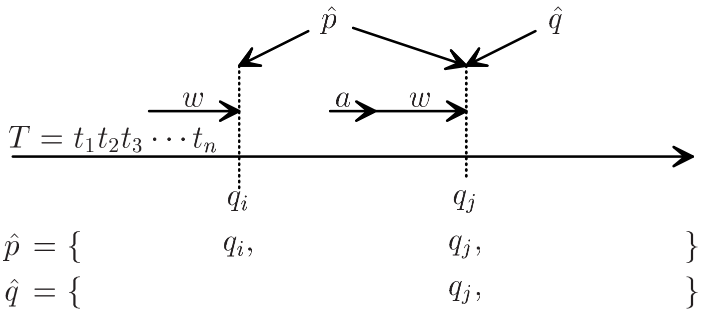


Figure 7: If the state $\hat{p}$ contains two states then it has incoming suffix link.

If the set $\hat{p}$ would contain more than one state (see Figure 7) then there would exist the longest factor w of the text T, which would end at ending positions represented by members of $\hat{p}$. Not all occurrences of string w are preceded by the same symbol (because w is the longest string with these endings) and therefore there would exist a string aw which is a factor of the text T and would end at positions $\hat{q}$ where $\hat{q} \subset \hat{p}$. Due to this inclusion both states $\hat{p}$ and $\hat{q}$ would share the same branch of suffix tree which would lead from $\hat{q}$ to $\hat{p}$. The state $\hat{p}$ would have at least one incoming suffix link, which gives the contradiction.

The second part, the proof of backward implication (the set $\hat{p}$ contains only one state $\Longrightarrow$ the state $\hat{p}$ has no incoming suffix link) is trivial because a suffix link can lead only from a subset to a superset and a set with just only one state has no regular subsets. □

Lemma 3.4 If a state $\hat{p} \in \hat{Q}$ has just one incoming suffix link and w is the longest string leading to this state $\hat{p} = \hat{\delta}^*(\hat{q}_0, w)$ (see Figure 8) then

a) there are at least two occurrences of the string w in the text T,

b) the string w is a prefix of the text T,

c) all occurrences of w in T except the very first one (the prefix of T) are preceded by the same symbol.

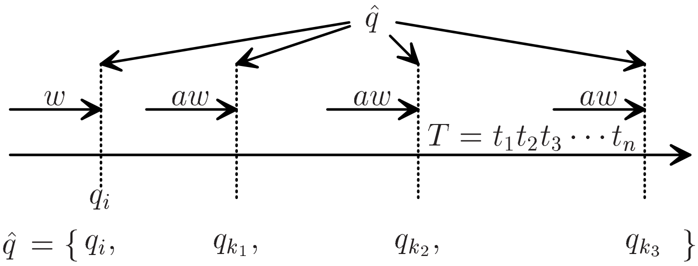


Figure 8: The only one incoming suffix link leads to a state $\hat{p}$.

Proof:

The proof of part a) follows from the Lemma 3.3.

There are no couple of occurrences of string w following two different symbols. If two strings aw and bw (where $a \neq b$) would occur in text T then both states $\hat{q}_{aw}$ and $\hat{q}_{bw}$ would be disjunct subsets of $\hat{p}$ and their suffix links would lead to state $\hat{p}$. At least one occurrence of w must not be preceded by the same symbol as others because w is the longest string leading to state $\hat{p}$. Therefore w occurs at the beginning of T and all next occurrences are preceded by the same symbol. w is a prefix of T. This proves parts b) and c). □

Lemma 3.5 If a suffix link $suf[\hat{q}] = \hat{p}$ is the only suffix link leading to state $\hat{p}$ then set $\hat{p}$ is larger then $\hat{q}$ by just one state q_i (i.e. $\hat{p} = \{q_i\} \cup \hat{q}$).

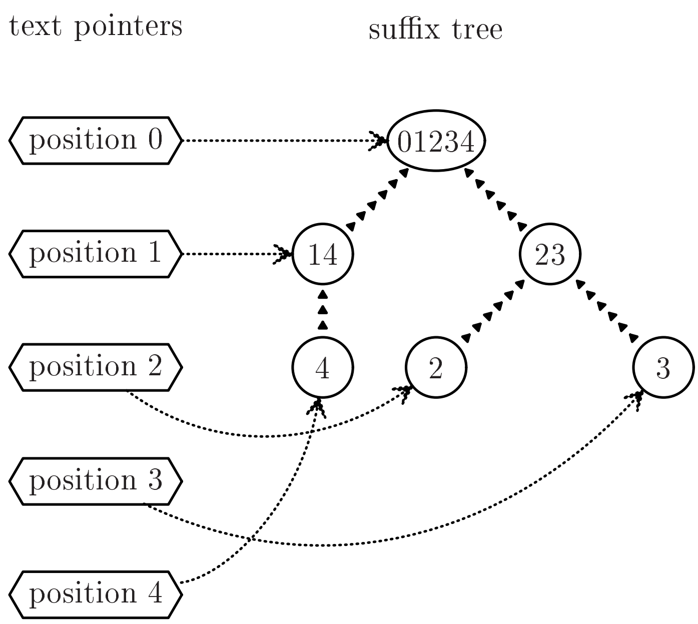
Proof:

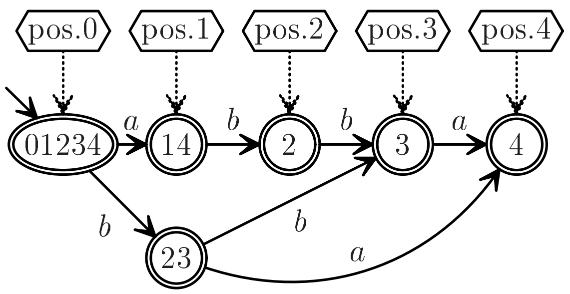
Let w be a string leading to the state $\hat{p} = \hat{\delta}^*(\hat{q}_0, w)$ (see Figure 8). Due to Lemma 3.4, string w is a prefix of the text T and all other occurrences of w in the text T are preceded by the same symbol a. The string aw occurs at the same ending positions as string w except the very first one (w is a prefix of T). We can divide the set $\hat{p}$ into the first occurrence (the state q_i) and the rest (occurrences of aw): $\hat{p} = \{q_i\} \cup \hat{\delta}^*(\hat{q}_0,\text{aw})$. Due to Lemma 3.2 it holds $\hat{p} = suf[\hat{\delta}^*(\hat{q}_0,\text{aw})]$. There is only one suffix link leading to $\hat{p}$ so that states $\hat{\delta}^*(\hat{q}_0,\text{aw})$ and $\hat{q}$ are identical and we can write $\hat{p} = \{q_i\} \cup \hat{q}$.

□

3.2 Text Pointers

Most of algorithms operating on factor automaton need to resolve which states of CDFA represent given state q of CNFA. Since all relevant CDFA states contain q they create a separate branch in the suffix tree. We can store only the starting state of the branch and continue over the suffix tree to its root. Text pointers is a data structure which keeps the information about the starting state. It can be implemented as an array $TextPos[i]$ of CDFA states indexed by position i in text. In factor automata it holds $TextPos[i] = \hat{\delta}^*(\hat{q}_0, t_1 t_2 \cdots t_i)$. An example of text pointers array for $T = abba$ is on Figure 9.





Text positions: $T = {}_0a_1b_2b_3a_4$
text pointers table

position	state
0	$\hat{q}_{\{q_0,q_1,q_2,q_3,q_4\}}$
1	$\hat{q}_{\{q_1,q_4\}}$
2	$\hat{q}_{\{q_2\}}$
3	$\hat{q}_{\{q_3\}}$
4	$\hat{q}_{\{q_4\}}$

Figure 9: An example of the suffix tree and the automaton with text pointers for $T = abba$.

Note that the number of states is often larger then the number of positions in the text. Therefore, there exist states which are not the value of any $TextPos$. An example of that is on Figure 9. Although the state $\hat{q}_{\{q_2,q_3\}}$ represents ending positions 2 and 3 for string b, it is neither a value of $TextPos[2]$ nor $TextPos[3]$. We can get all states representing the ending position 2 by inspecting the whole branch of suffix tree (a sequence of suffix links) from the state $\hat{q}_{\{q_2\}} = TextPos[2]$.

3.3 Node In-degree

We use the number of transitions leading to this state (incoming transitions) as a reference counter for detecting unreachable states. If the automaton has unreachable states then one of them must have in-degree equal to zero because the CDFA has no loops. After its removing it holds that either another unreachable state becomes zero in-degree or we are sure there are no unreachable states in the automaton.

3.4 Operation L-INSERT

The canonical nondeterministic factor automata (CNFA) for the texts $T = t_1t_2t_3 \cdots t_n$ and $aT = at_1t_2t_3 \cdots t_n$ are shown on the Figure 10.

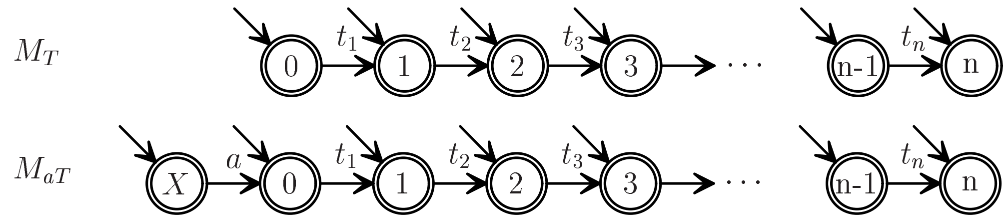


Figure 10: The change in CNFA when a new symbol is prepended.

The operation L-INSERT creates a new state q_X, which is both initial and final and a new transition from the state q_X into the state q_0.

The algorithm modifying CDFA follows from the relationship between nondeterministic and deterministic factor automaton.

When the new initial state q_X is created, CDFA's initial state $\hat{q}_0$ – see Figure 11 (step 1) – is changed to the new state $\hat{q}'_0 = \hat{q}_0 \cup \{q_X\}$. The outgoing transitions from this state are still the same as from $\hat{q}_0$ (step 2). Now, we create a new transition in CNFA leading from q_X to q_0 for symbol a. In the CDFA, we should redirect the transition leading from $\hat{q}'_0$ labeled by a symbol a to another state which contains similar set of states extended by the state q_0, because $q_0 = \delta(q_X, a)$ is the new transition (step 3).

The algorithm is based on the recursive function `GetExtendedState`$(\hat{q}, i)$, which takes the set of states $\hat{q}$ and integer i as arguments, and finds a state $\hat{q}' = \hat{q} \cup \{q_i\}$. If there is no such state in the automaton, it is created by the function. The value of the function is the state $\hat{q}'$ (Figure 12).

Using this function the whole algorithm can be written in five steps:

1. create a new state $\hat{q}'_0$ with the same outgoing transitions as $\hat{q}_0$,

2. get the old target of the first transition: $\hat{q} = \hat{\delta}(\hat{q}'_0, a)$,

3. compute new state for that transition: $\hat{q}' =$ `GetExtendedState`$(\hat{q}, 0)$,

4. redirect the transition: $\hat{\delta}(\hat{q}'_0, a) = \hat{q}'$,

5. change the initial state to $\hat{q}'_0$.

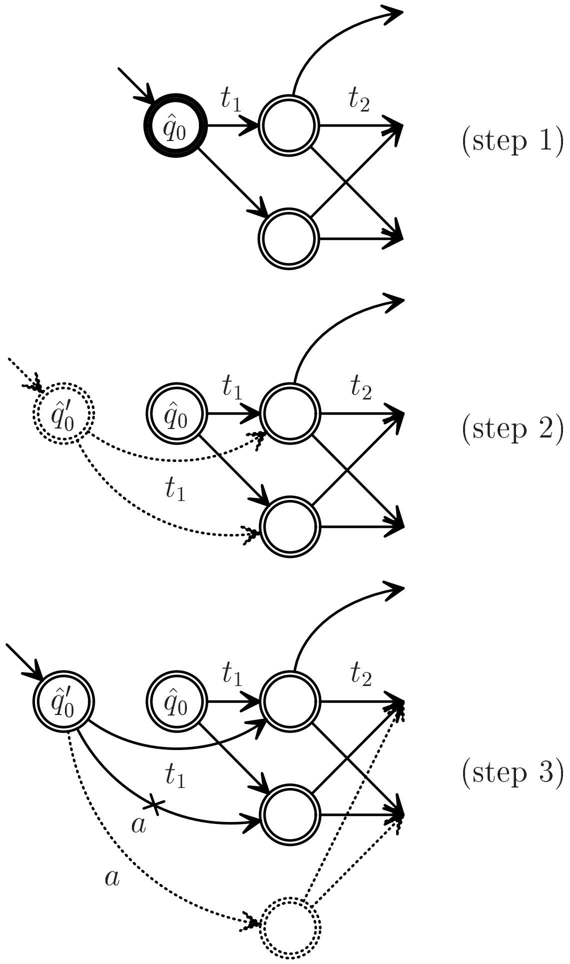


Figure 11: The change in CDFA when symbol a is inserted.

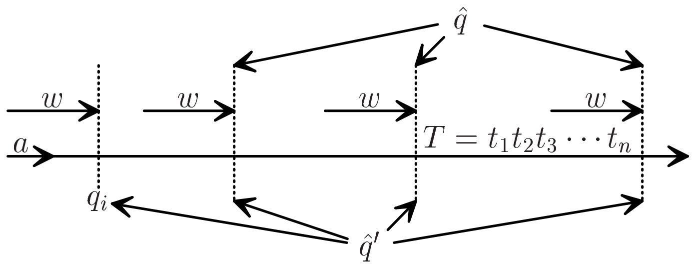


Figure 12: The state $\hat{q}'$ contains state q_i and all states from $\hat{q}$

We assume any unreachable state is removed as soon as it looses the last incoming transition (or the last reference).

Let us concern the function `GetExtendedState`$(\hat{q}, i)$. It assumes that the string $w = at_1t_2t_3 \cdots t_i$ leads to the state $\hat{q}$ (i.e. $\hat{q} = \hat{\delta}^*(\hat{q}_0, w)$). It is the shortest string leading to this state because the text shorter by the first symbol a would be a prefix of T an would occur in advance at ending position i.

Note that the string $w = at_1t_2t_3 \cdots t_i$ may not be a factor of the text T. In this case the state $\hat{q}$ may be $\hat{q} = \{\} = \emptyset$. In such case, the solution is a state $\hat{q}' = \{q_i\}$. Of course, this state may or may not be present in the current automaton. We can find it by inspecting the text pointer at position i. The value of $TextPos[i]$ may be the required state $\hat{q}' = \{q_i\}$ or its superset. According to Lemma 3.3: if there is no suffix link leading to this state then it contains only one CNFA state $\{q_i\}$ and it is the result value of the function `GetExtendedState` (Figure 13). If there exists a

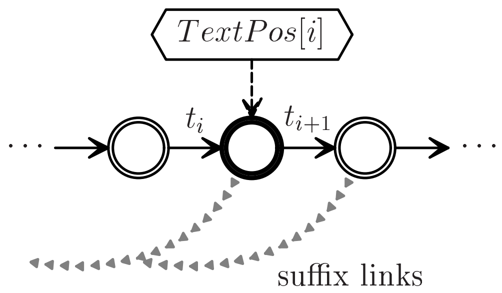


Figure 13: The focused state has no incoming suffix links therefore it contains only one state q_i

suffix link leading to this state then we must create a new state $\hat{q}' = \{q_i\}$ and set its outgoing transitions. In this case the state $\hat{q}'$ will have only one outgoing transition for the symbol t_i leading to state $\{q_{i+1}\}$ (which can be obtained by recursive calling the function `GetExtendedState`$(nil, i + 1)$). In addition, we should set up the suffix link of this state to lead to $TextPos[i]$ and update $TextPos[i]$ to new value – state $\hat{q}'$. (See Figure 14).

Now, we concern the case when $\hat{q}$ is an already existing state of CDFA. The function `GetExtendedState` should locate the state representing the set $\hat{q} \cup \{q_i\}$. If there is no such state, it should be created. Due to the Lemma 3.2 if there exists such state it must be the target of the suffix link from state $\hat{q}$. But the suffix parent $\hat{p} = suf(\hat{q})$ of the state $\hat{q}$ may not be the required state in any case, of course. We can test it by inspecting the number of suffix links leading to it. There are two disjunct cases:

- only one suffix link leads to state $\hat{p}$,
- the state $\hat{p}$ is a target of more suffix links.

At first we assume the suffix link from the state $\hat{q}$ to the state $\hat{p}$ is the only link leading to $\hat{p}$ (Figure 15). As the string $w = at_1t_2t_3 \cdots t_i$ is the shortest string leading to $\hat{q}$ then the first suffix – string $u = t_1t_2t_3 \cdots t_i$ leads to state $suf(\hat{q}) = \hat{p}$. We are sure that string $t_1t_2t_3 \cdots t_i$ occurs at position i and therefore $\hat{p}$ contain the required

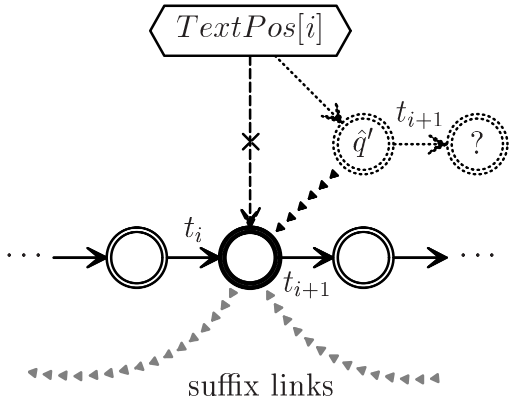


Figure 14: If any suffix link leads to the state found by $TextPtr[i]$ then we have to create a new state $\hat{q}'$, connect its suffix link, outgoing transition and redirect $TextPtr[i]$

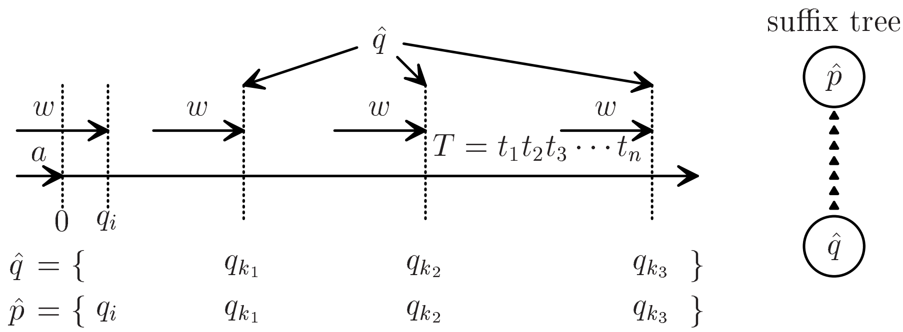


Figure 15: $\hat{q} \mapsto \hat{p}$ is the only suffix link leading to $\hat{p}$ therefore $\hat{p} = \hat{q} \cup \{q_i\} = \hat{p}'$

state $\hat{q}_i$. On the other side, the state $\hat{p}$ does not contain any other state then $\{q_i\}$ or $\hat{q}$ (see Lemma 3.5) therefore state $\hat{p}$ is the value of the function `GetExtendedState` .

Now, assume there exist at least two suffix links leading to the state $\hat{p}$. One of them is the link from $\hat{q}$ and let another one lead from a state $\hat{q}_q$ (Figure 16). The

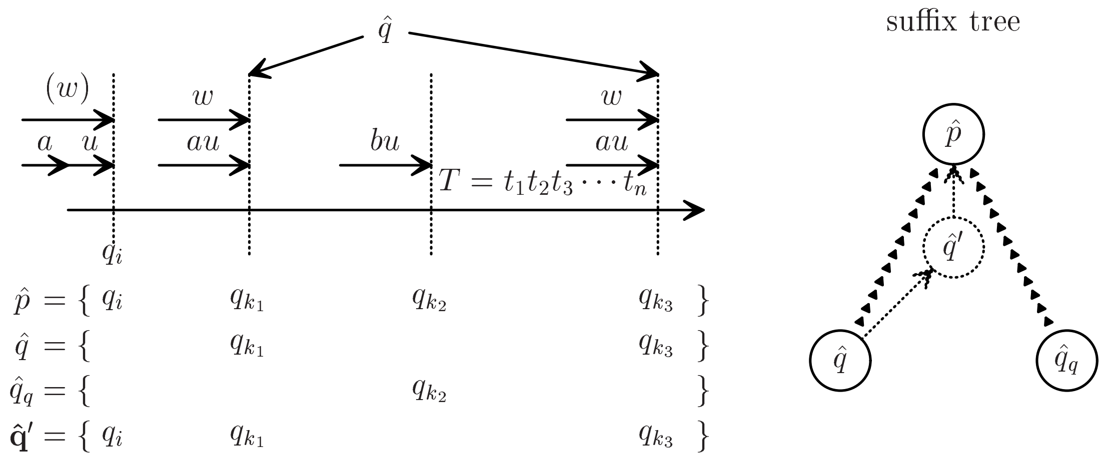


Figure 16: If the state $\hat{p}$ receives more suffix links then it is unusable. A new state $\hat{q}'$ has to be created.

sets $\hat{q}$ and $\hat{q}_q$ are disjunct because they are in the different branches of the suffix tree. The state $\hat{p}$ is the superset of both sets. Therefore, the set $\hat{p}$ contains more states then $\hat{q} \cup \{q_i\}$ and will be unusable for us. The resulting state is still not in the set of states of the automaton and we have to create it.

We create a new state $\hat{q}'$ which should represent the set $\hat{q} \cup \{q_i\}$ and therefore it inherits the same outgoing transition as $\hat{q}$. However the transition for the symbol t_{i+1} should be redirected to the state (the set of CNFA states) extended by the state q_{i+1}. We can lookup this state using the function `GetExtendedState` in recursion. The redirection is made by assigning $\hat{\delta}(\hat{q}', t_{i+1}) =$ `GetExtendedState`$(\hat{\delta}(\hat{q}, i), i+1)$. Finally, we should update suffix links. The new state $\hat{q}'$ is a subset of $\hat{p}$ and a superset of $\hat{q}$ therefore we include it between states $\hat{p}$ and $\hat{q}$: $suf[\hat{q}'] = \hat{p}$ and $suf[\hat{q}] = \hat{q}'$.

Algorithm 3.1 — Operation L-INSERT using function `GetExtendedState`

INPUT: CDFA automaton $\hat{M} = (\hat{Q}, \mathcal{A}, \hat{\delta}, \hat{q}_0, \hat{F})$ with suffix links, text T and text pointers
symbol a

OUTPUT: CDFA automaton $\hat{M}$ with suffix links, text T and text pointers

LOCAL: integer n
state $\hat{p}$
state $\hat{q}'$
state $\hat{q}'_0$
state $\hat{t}$

REQUIRE: $\hat{M}$ accepts factors of $T = t_1t_2t_3 \cdots t_n$

ENSURE: $\hat{M}$ will accept factors of $T = at_1t_2t_3 \cdots t_n$

function `GetExtendedState`(*state* $\hat{q}$, *integer* i)
if $(\hat{q} ==$ nil$)$ **then**

```
    t̂ = TextPtr[i]
    n = |suf^{-1}(t̂)|  { the number of suffix links incomming to t̂ }
    if (n == 0) then
        q̂' = t̂
        return q̂'
    else
        q̂' =new state
        δ̂(q̂', a) = GetExtendedState(nil, i + 1)
        suf[q̂'] = t̂
        return q̂'
    end if
else
    p̂ = suf[q̂]
    n = |suf^{-1}(p̂)|
    if (n == 1) then
        q̂' = p̂
        return q̂'
    else
        q̂' = duplicate(q̂)
        δ̂(q̂', t_{i+1}) = GetExtendedState(δ̂(q̂, t_{i+1}), i + 1)
        suf[q̂'] = p̂
        suf[q̂] = q̂'
        return q̂'
    end if
end if
endfunction
q̂'_0 = duplicate(q̂_0)
δ̂(q̂'_0, a) = GetExtendedState(δ̂(q̂_0, a), 0)
SetInitialState(q̂'_0)
```

4 Efficiency of the Algorithm

4.1 Time Complexity

The best case from the time complexity point of view appears when the new inserted symbol a is equal to each symbol in the text: $T = a^n$. In such case, the recursive function `GetExtendedState` is called only once. Neither this function nor the main algorithm contain loop, therefore the time complexity is constant $O(1)$ – independent on the size of the text T.

The worst case occurs if all symbols in text T are the same but different from the new inserted symbol a: $T = b^n$. In such case, the original automaton has $n+1$ states and the new automaton will have $2n - 1$ states, and so the algorithm have to create $n - 2$ states and it has asymptotically time complexity linear $O(n)$ with respect to the size of the text T.

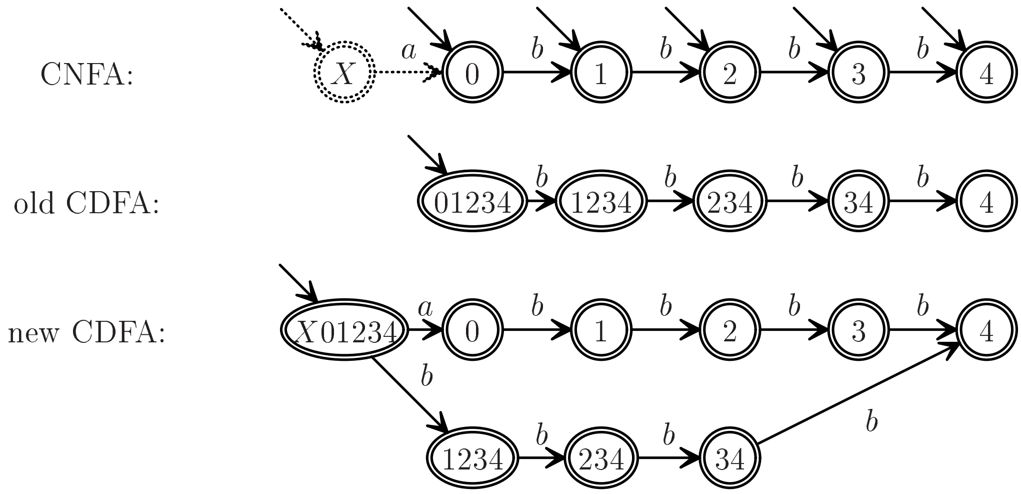


Figure 17: The worst case

4.2 Space Complexity

The algorithm requires extra space for following data structures:

- text pointers,
- suffix links,
- states,
- transitions,
- stack for recursion.

Text pointers is an array indexed by the position in text T. The size of the array is linear to the size of text T. Text pointers are more useful for other operations with factor automata. In the case of L-INSERT algorithm, text pointers can be substituted by text T, because we need successively the values $TextPos[0], TextPos[1], TextPos[2], ...$ and $TextPos[i] = \hat{\delta}(TextPos[i-1], t_i)$ while $TextPos[0] = \hat{q}_0$. So that we could compute the values of $TextPos$ during recursion of the function `GetExtendedState`.

Both suffix links and states take the same space complexity because there is just one outgoing suffix link per a state. The number of states is at most $2n$ (proved in [1]).

The number of transitions in the factor automaton is less than $3n$ (proved in [1]).

The size of the stack required for the recursion is limited by the number of recursive calls. As a new states is created before any recursive call, the total number of recursive calls is limited by the number of inserted states. Moreover, the recursion function `GetExtendedState` can be transformed into an iteration loop without a need of an extra data space.

As the all data structures require space at most linear to the size of the automaton, we can say the L-INSERT algorithm is space-linear.

5 Conclusion

This paper deals with the factor automaton and its modifications when the text often changes. We discuss several operations on the text and cite algorithms reflecting these operations into the factor automaton. Moreover we describe some adjacent data structures (suffix links and text pointers) used in algorithms modifying the factor automaton. We present a new algorithm of operation L-INSERT. The algorithm can efficiently modify a factor automaton when a new symbol is inserted before the first symbol of the text. This algorithm can be also used for on-line backward construction of the factor automata. This means that the text grows from right to left while constructing the automaton. Finally, the time and space complexity of the L-INSERT algorithm is also discussed.

References

[1] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.

[2] M. Šimánek. The factor automaton. In J. Holub and M. Šimánek, editors, *Proceedings of the Prague Stringologic Club Workshop '98*, pages 102–106, Czech Technical University, Prague, Czech Republic, 1998. Collaborative Report DC–98–06.

[3] J. E. Hopcroft and J. D. Ullman. *Introduction to automata, languages and computations*. Addison-Wesley, Reading, MA, 1979.

[4] J. Holub. Simulation of nondeterministic finite automata in approximate string and sequence matching. Technical Report DC–98–04, Department of Computer Science and Engineering, Czech Technical University, Prague, Czech Republic, 1998.

An Efficient Mapping for Score of String Matching

Tetsuya Nakatoh[1], Kensuke Baba[2], Daisuke Ikeda[1], Yasuhiro Yamada[3], and Sachio Hirokawa[1]

[1] Computing and Communications Center, Kyushu University
Hakozaki 6-10-1, Higashi-ku, Fukuoka 812-8581, Japan
e-mail: {nakatoh,daisuke,hirokawa}@cc.kyushu-u.ac.jp

[2] PRESTO, Japan Science and Technology Corporation
Honcho 4-1-8, Kawaguchi City, Saitama 332-0012, Japan
e-mail: baba@i.kyushu-u.ac.jp

[3] Graduate School of Information Science and Electrical Engineering
Kyushu University, Hakozaki 6-10-1, Higashi-ku, Fukuoka 812-8581, Japan
e-mail: yshiro@cc.kyushu-u.ac.jp

Abstract. This paper proposes an efficient algorithm to solve the problem of *string matching with mismatches*. For a text of length n and a pattern of length m over an alphabet Σ, the problem is known to be solved in $O(|\Sigma| n \log m)$ time by computing a score by the fast Fourier transformation (FFT). Atallah et al. introduced a randomized algorithm in which the time complexity can be decreased by the trade-off with the accuracy of the estimates for the score. The algorithm in the present paper yields an estimate with smaller variance compared to that the algorithm by Atallah et al., moreover, and computes the exact score in $O(|\Sigma| n \log m)$ time. The present paper also gives two methods to improve the algorithm and an exact estimation of the variance of the estimates for the score.

Keywords: string matching with mismatches, FFT, convolution, deterministic algorithm, randomized algorithm.

1 Introduction

String matching [4, 5] is the problem to obtain all the occurrences of a (short) string called a *pattern* in a (long) string called a *text*. We consider *string matching with mismatches* which allows inexact match introduced by substitution. Let Σ be an alphabet and δ the Kronecker function from $\Sigma \times \Sigma$ to $\{0, 1\}$, that is, for $a, b \in \Sigma$, $\delta(a, b)$ is 1 if $a = b$, 0 otherwise. The problem with mismatches is generally solved by computing the *score vector* $C(T, P)$ between a text $T = t_1 \cdots t_n$ and a pattern $P = p_1 \cdots p_m$ as follows:

$$C(T, P) = (c_1, \ldots, c_i, \ldots, c_{n-m+1}), \quad \text{where} \quad c_i = \sum_{j=1}^{m} \delta(t_{i+j-1}, p_j).$$

We can compute the score vector using the fast Fourier transform (FFT) in $O(n \log m)$ time, if the score vector is represented as a convolution, that is, if the Kronecker function is expressed by a product of two mappings from Σ to a set of numbers. This approach was developed by Fischer and Paterson [6] and is simply summarized in Gusfield [7]. However, practically, the time complexity of the algorithm depends on the number of alphabets. One of the reason for the difficulties is that the Kronecker function can not be written as a product of mappings directly. For example, if $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, the generalized algorithm in [7] needs three mappings ϕ_1, ϕ_2, and ϕ_3 which convert symbols into $\{1, 0\}$ as the following table.

	ϕ_1	ϕ_2	ϕ_3
a	1	0	0
b	0	1	0
c	0	0	1

Then, we have $\delta(a, b) = \sum_{\ell=1}^{3} \phi_\ell(a) \cdot \phi_\ell(b)$ and the score vector is obtained by computing the convolution $\sum_{j=1}^{m} \phi_\ell(t_{i+j-1}) \cdot \phi_\ell(p_j)$ for $1 \le i \le n$ three times.

Atallah et al. [1] introduced a randomized algorithm where the time complexity has a trade-off with the accuracy of the estimates for the score vector. In this algorithm, symbols are converted into complex numbers with a primitive σ-th root ω of unity and the Hermitian inner product is used for the convolution. Then, the score vector is obtained as the average of the results of convolutions with respect to all possible mappings φ_ℓ from Σ to $\{0, \ldots, |\Sigma| - 1\}$, that is,

$$c_i = \frac{1}{|\Phi|} \sum_{\ell=1}^{|\Phi|} \sum_{j=1}^{m} \omega^{\varphi_\ell(t_{i+j-1}) - \varphi_\ell(p_j)},$$

where Φ is the set of all mappings ϕ_ℓ. (A deterministic algorithm constructed by those mappings requires the computation of the convolution $|\Sigma|^{|\Sigma|}$ times.) An estimate for the score vector is the average of the results with respect to some mappings chosen independently and uniformly from Φ. Let k be the number of randomly chosen samples. Then, the time complexity is $O(kn \log m)$. They showed that the expectation of the estimates equals to the score vector and the variance is bounded by $(m - c_i)^2/k$. Baba et al. [2] improved this algorithm by simplifying the mappings which converts the strings into numbers. The codomain of the mappings is the set $\{-1, 1\}$ instead of the set of complex numbers. Then, the score vector is

$$c_i = \frac{1}{|\Phi|} \sum_{\ell=1}^{|\Phi|} \sum_{j=1}^{m} \phi_\ell(t_{i+j-1}) \cdot \phi_\ell(p_j).$$

Baba et al. [3] pointed out that the algorithms which compute the score vector by FFT are distinguished by the mappings which convert strings into numbers in each algorithm, and the exact score is obtained by repeating the $O(n \log m)$ operation $|\Phi|$ times.

In this paper, we propose an efficient algorithm to solve string matching in which the variance of the estimates is not greater than $(m - c_i)^2/k$. Moreover, the exact score vector is computed in $O(|\Sigma| n \log m)$ time. We also give a strict evaluation of the variance and introduce two methods to improve our algorithm.

2 Efficient Algorithm

We propose an efficient algorithm for string matching with mismatches. The time complexity of a deterministic algorithm and the variance of the estimates for the score vector are obtained by analyzing the mappings which convert the symbols to the numbers. Let p be the smallest prime number which is greater than or equal to the cardinality $|\Sigma|$ of the alphabet. The codomain of the mappings is the p-adic number field Z_p. Since such a prime number is less than $2|\Sigma| - 2$ (Chebyshev's theorem), a deterministic algorithm with this mappings computes the score vector between a text of length n and a pattern of length m in $O(|\Sigma| n \log m)$ time. Moreover, in the same way as the algorithm by Atallah et al, we can construct a randomized algorithm in which the variance of the estimates for the score vector is independent to $|\Sigma|$.

2.1 Efficient Mapping

Let φ be a bijective mapping from Σ to $\{0, 1, \cdots |\Sigma| - 1\}$. For $0 \leq x \leq p-1$ and $a \in \Sigma$, we define a mapping ϕ_x as

$$\phi_x(a) = \omega^{x \cdot \varphi(a)}, \tag{1}$$

where ω is a primitive p-th root of unity. Then, we have the following lemma.

Lemma 1 For any $a, b \in \Sigma$,

$$\delta(a, b) = \frac{1}{p} \sum_{x=0}^{p-1} \phi_x(a) \cdot \overline{\phi_x(b)},$$

where $\overline{\omega^y} = \omega^{-y}$.

Proof. If $a = b$, we have $\phi_x(a) \cdot \overline{\phi_x(b)} = \omega^0 = 1$ for any $0 \leq x \leq p-1$. Hence, the right side of the equation is equal to 1. If $a \neq b$, the difference $\varphi(a) - \varphi(b)$ is an element of $Z_p \backslash \{0\}$. Therefore, $x \cdot (\varphi(a) - \varphi(b))$ is valued $0, \ldots, p-1$ modulo p for $0 \leq x \leq p-1$. Thus, we have $\sum_{x=0}^{p-1} \phi_x(a) \cdot \overline{\phi_x(b)} = \sum_{x=0}^{p-1} \omega^{x \cdot (\varphi(a) - \varphi(b))} = 0$. □

Lemma 2 By using the mapping ϕ_x, the score vector between a text of length n and a pattern of length m over an alphabet Σ can be computed in $O(|\Sigma| n \log m)$ time.

Proof. By the definition of the score vector and Lemma 1, the score vector is

$$c_i = \frac{1}{p} \sum_{x=0}^{p-1} \sum_{j=1}^{m} \phi_x(t_{i+j-1}) \cdot \overline{\phi_x(p_j)}. \tag{2}$$

Therefore, the score vector is obtained by computing the convolution

$$f(i) = \sum_{j=1}^{m} \phi_x(t_{i+j-1}) \cdot \overline{\phi_x(p_j)} \quad (1 \leq i \leq n)$$

p times. Since $p = O(|\Sigma|)$, we have the lemma. □

2.2 Analysis of Variance

In the same way as the algorithm by Atallah et al. [1], we can construct a randomized algorithm in which an estimate for the score vector is obtained by choosing some mappings from Φ. We define a *sample* s_i of an element c_i of the score vector to be

$$s_i = \sum_{j=1}^{m} \phi_{x(\ell)}(t_{i+j-1}) \cdot \overline{\phi_{x(\ell)}(p_j)}.$$

Let k be the number of chosen samples. Then, an *estimate* $\hat{s}_i$ for the element c_i of the score vector is defined by

$$\hat{s}_i = \frac{1}{k} \sum_{\ell=1}^{k} s_i.$$

By Eq. (2), it is clear that the mean of the estimates is equal to c_i. The following lemma gives the upper-bound of the variance of the estimates.

Lemma 3 In a randomized algorithm constructed with the mapping ϕ_x, the variance of the estimates for the score vector is bounded by $(m - c_i)^2/k$.

Proof. We denote by $V(X)$ the variance of a random variable X. By the definition of the estimate and the basic property of variance, we have $V(\hat{s}_i) = V(s_i)/k$. Since $\phi_{x(\ell)}(a) \cdot \overline{\phi_{x(\ell)}(a)} = 1$ and $|\phi_{x(\ell)}(a) \cdot \overline{\phi_{x(\ell)}(b)}| = 1$ for any $1 \leq \ell \leq |\Phi|$ and any $a, b \in \Sigma$, the variance of the samples is $V(s_i) = \sum_{\ell=1}^{|\Phi|}(\sum_{j=1}^{m} \phi_{x(\ell)}(t_{i+j-1}) \cdot \overline{\phi_{x(\ell)}(p_j)} - c_i)^2/|\Phi| \leq (m - c_i)^2$. $\square$

2.3 Description of Algorithm

We describe the algorithm which uses the mapping ϕ_x in detail. The input is a text string $T = t_1 \cdots t_n$, a pattern string $P = p_1 \cdots p_m$ over an alphabet Σ, and a number k of iterations in this algorithm. The output is an estimate for the score vector $C(T, P)$ if $k < p$, the exact score vector if $k = p$, where p is the smallest prime number such that $|\Sigma| \leq p$. By the standard technique [4] of partitioning the text, we can assume $n = (1 + \alpha)m$ for $\alpha = O(m)$. The algorithm is constructed by iterations of the following operations.

- convert the text into a numerical sequences $\phi_x(T) = \omega^{\varphi_x(t_1)} \cdots \omega^{\varphi_x(t_{(1+\alpha)m})}$ by the mapping ϕ_x from Σ to $\{\omega^0, \ldots, \omega^{p-1}\}$;

- convert the pattern into $\overline{\phi_x(P)} = \omega^{-\varphi_x(p_1)} \cdots \omega^{-\varphi_x(p_m)}$ by ϕ_x and pad with αm zeros;

- compute the sample s_i for $1 \leq i \leq (1 + \alpha)m$ as the convolution of $\phi_x(T)$ and the reverse of the padded $\overline{\phi_x(P)}$ by FFT.

The output is computed as the average of the results of the convolution for $1 \leq x \leq k$. If $k = p$, by Lemma 2, the output is equal to the score vector. If $k < p$, the output is regarded as an estimate for the score vector obtained by a randomized algorithm with "sampling without replacement". Therefore, by Lemma 3 the variance of the estimates is $((p - k)/(p - 1)) \cdot (V(s_i)/k)$.

Theorem 1 By the algorithm with the mapping ϕ_x, the exact score between a text of length n and a pattern of length m over an alphabet Σ is computed in $O(|\Sigma| n \log m)$ time. Moreover, an estimate for the score vector is computed in $O(kn \log m)$ time, where k is the number of iterations in the algorithm and the variance of the estimates is bounded by $(p-k)(m-c_i)^2/(p-1)k$.

In generally, the variance of the estimates obtained by sampling without replacement is

$$\frac{|\Phi| - k}{|\Phi| - 1} \cdot V(\hat{s}_i)$$

where Φ is the set of all mappings which convert symbols into numbers. The cardinality $|\Phi|$ of the set is $|\Sigma|^{|\Sigma|}$ in the algorithm by Atallah et al [1]. and $2^{|\Sigma|}$ in one by Baba et al [2]. Hence, the finite-size correction term $(|\Phi| - k)/(|\Phi| - 1)$ is not so effective.

A key distinguishing feature of our algorithm is that the exact score can be computed in a practical time. Since $|\Phi|$ is large in the two randomized algorithms, their deterministic versions constructed in a similar way as our algorithm are not practical for a large alphabet. Although the deterministic algorithm generalized by Gusfield [7] can be extended to a randomized algorithm in the same way as our algorithm, the variance of the estimates depends on the number of alphabets.

3 Improvement of Algorithm

We propose two techniques to improve the algorithm in the previous section.

3.1 Removal of Defective Mapping

Our mappings convert the different symbols to the distinct numerical values. But only the mapping ϕ_0 converts all symbols to 0. Therefore, we remove the mapping ϕ_0 from the set Φ. That is possible without computing convolution.

By Eq. (1), $\delta(a,b) = \frac{1}{p}\sum_{x=0}^{p-1} \phi_x(a) \cdot \overline{\phi_x(b)} = \frac{1}{p}(\sum_{x=1}^{p-1} \phi_x(a) \cdot \overline{\phi_x(b)} + \phi_0(a) \cdot \overline{\phi_0(b)}) = \frac{1}{p}(\sum_{x=1}^{p-1} \phi_x(a) \cdot \overline{\phi_x(b)} + 1)$. Therefore, the score vector is $c_i = \sum_{j=1}^{m} \frac{1}{p}(\sum_{x=1}^{p-1} \phi(t_{i+j-1}) \cdot \overline{\phi(p_j)} + 1) = \frac{1}{p}\sum_{x=1}^{p-1}\sum_{j=1}^{m} \phi_x(t_{i+j-1}) \cdot \overline{\phi_x(p_j)} + \frac{m}{p}$. To randomize the computation of c_i, we define c'_i as follows: $c'_i = \frac{1}{p-1}\sum_{x=1}^{p-1}\sum_{j=1}^{m} \phi_x(t_{i+j-1}) \cdot \overline{\phi_x(p_j)}$. Hence, $c_i = \frac{p-1}{p}c'_i + \frac{m}{p}$.

We define a sample s'_i of an element c'_i to be

$$s'_i = \sum_{j=1}^{m} \phi_x(t_{i+j-1}) \cdot \overline{\phi_x(p_j)}.$$

And an estimate $\hat{s'_i}$ is defined by

$$\hat{s'_i} = \frac{1}{k}\sum_{\ell=1}^{k}\sum_{j=1}^{m} \phi_x(t_{i+j-1}) \cdot \overline{\phi_x(p_j)}$$

where $1 \le k \le p-1$.

And an estimate $\hat{s}_i$ for the element c_i of the score vector is defined by

$$\hat{s}_i = \frac{p-1}{p}\frac{1}{k}\sum_{\ell=1}^{k}\sum_{j=1}^{m}\phi_x(t_{i+j-1})\cdot\overline{\phi_x(p_j)} + \frac{m}{p} \tag{3}$$

where $1 \le k \le p-1$.

By the difinition of a variance, $V(s_i) = \frac{(p-1)^2}{p^2}V(s_i')$. Moreover, because the number of mappings decrease by one, the variance in consideration of that is bounded by

$$\frac{(p-1)^2}{p^2}\cdot\frac{p-1-k}{p-2}\cdot\frac{(m-c_i)^2}{k}. \tag{4}$$

3.2 Removal of Imaginary Part

The magnitude of $\phi_x(a)\cdot\overline{\phi_x(b)}$ in Eq. (1) is 1. We used this magnitude for the analysis of the variance until this point. However, the real part is independent of the imaginary part. Therefore, those parts of Eq. (1) can be computed separately.

Let $\Re(v)$ be a real part of a complex number v. By Lemma 1, $\frac{1}{p}\sum_{x=0}^{p-1}\phi_x(a)\cdot\overline{\phi_x(b)}$ returns 0 or 1. Therefore, we can remove the imaginary part. Then, $\delta(a,b) = \Re(\frac{1}{p}\sum_{x=0}^{p-1}\phi_x(a)\cdot\overline{\phi_x(b)})$ for any $a,b \in \Sigma$. By the definition of the score, $c_i = \sum_{j=1}^{m}\Re(\frac{1}{p}\sum_{x=0}^{p-1}\phi_x(t_{i+j-1})\cdot\overline{\phi_x(p_j)})$. Since the order of addition is not restricted, the score vector is

$$c_i = \frac{1}{p}\sum_{x=0}^{p-1}\Re(\sum_{j=1}^{m}\phi_x(t_{i+j-1})\cdot\overline{\phi_x(p_j)}).$$

The computation of the complex number is necessary to compute convolution with FFT. We only have to omit the imaginary part after the computation of FFT. By this omission, the computation of both the sum of the imaginary part and the magnitude of complex number become unnecessary.

The variance is the poorest when inconsistent $m-c$ characters are each a kind of symbol on the text and the pattern. In such a case, $\phi_\ell(a)\cdot\overline{\phi_\ell(b)}$ is fixed without influence of j. By Eq. (1), $\Re(\phi_x(a)\cdot\overline{\phi_x(b)}) = \cos\theta_\ell$, where $\theta_\ell = \frac{2\pi x\cdot(\varphi(a)-\varphi(b))}{p}$. Then, the random variable s_i is following.

$$s_i = \sum_{j=1}^{m}\Re(\phi_\ell(a)\cdot\overline{\phi_\ell(b)}) = \sum_{j=1}^{m}\cos\theta_\ell = c_i\cos 0 + (m-c_i)\cos\theta_\ell = c_i + (m-c_i)\cos\theta_\ell.$$

The variance $V(s_i)$ of this random variable s_i are followings.

$$\begin{aligned} V(s_i) &= \sum_{\ell=1}^{p}(c_i + (m-c_i)\cos\theta_\ell - c_i)^2\cdot\frac{1}{p} \\ &= \frac{1}{p}\sum_{\ell=1}^{p}((m-c_i)\cos\theta_\ell)^2 \\ &= \frac{1}{p}(m-c_i)^2\sum_{\ell=1}^{p}\cos^2\theta_\ell \\ &= \frac{(m-c_i)^2}{p}\sum_{\ell=1}^{p}\frac{1+\cos\theta_\ell}{2} \end{aligned}$$

$$
\begin{aligned}
&= \frac{(m-c_i)^2}{2p}\left(\sum_{\ell=1}^{p} 1 + \sum_{\ell=1}^{p} \cos\theta_\ell\right) \\
&= \frac{(m-c_i)^2}{2p}(p+0) \\
&= \frac{(m-c_i)^2}{2} \qquad (5)
\end{aligned}
$$

By $V(\hat{s}_i) = V(s_i)/k$, the variance of the estimates $\hat{s}_i$ is bounded by

$$\frac{(m-c_i)^2}{2k}. \qquad (6)$$

3.3 Variance of Improved Algorithm

We showed two improvement points. That both can be applied to the basic algorithm at a time.

Now, the change point of the algorithm from the basis one shown in Subsection 2.3 is showed in the followings.

- We remove ϕ_0, and choose a sample from the remaining mappings.
- An estimate $\hat{s}'_i$ is computed using that samples.
- Only a real part is used for a computation of an estimate from the result of FFT.
- We compute $\hat{s}_i$ by Eq. (3), and make it the estimate of c_i.

When these improvements are applied, by Eq. (4) and Eq. (6), the variance of the estimates is bounded by

$$\frac{(p-1)^2}{p^2} \cdot \frac{p-1-k}{p-2} \cdot \frac{(m-c_i)^2}{2k}.$$

It is smaller than one in the algorithm of Section 2.

4 Exact Estimation of Variance

Atallah et al. presented an upper bound of the variance of the estimates for the score in their algorithm as $(m-c_i)^2/k$. The reason for this variance is that their set of mappings includes many mappings which convert some different symbols into same numerical value. One of the features of our mappings is that it does not convert some different symbols into same numerical value because a single exceptional mapping was removed in Subsection 3.1. Using this feature, we give an exact estimation of the variance based on our mappings.

Let a, b be symbols in Σ. If a product $\phi(a) \cdot \overline{\phi(b)}$ in one position is independent of it in other position, the estimate of $\sum_j^{(m-c_i)} \phi_x(t_j) \cdot \overline{\phi_x(p_j)}$ is 0. The two following conditions must be satisfied for that. One of those conditions is that a symbol in one position is independent of symbols in other positions. In this paper, we suppose that condition. The independence can not be expected in the general English text much. But, we expect high independence about the comparison of the product $\phi(a) \cdot \overline{\phi(b)}$.*

*In this paper, we did not get to the verification of that point. It is a future work.

Another condition is the following lemma.

Lemma 4 If all mappings convert different symbols into distinct numerical values, then the product $\phi(a) \cdot \overline{\phi(b)}$ in one position is independent of that in other position.

Proof. Let t_1, t_2, p_1, p_2 be symbols in Σ, x a value which can be returned by mappings and r the number of kinds of x. Let Φ_x be a set of the mappings which convert more than one of some symbols into x, and Φ_{xy} denotes $\Phi_x \cap \Phi_y$. We define D_x as the difference between the number of x which the mappings convert a given symbol into and the number of mappings used for it. The number of certain value x which a certain symbol a convert to is $\frac{|\Phi|}{r}$ because $\sum_{\ell=1}^{|\Phi|} \phi_\ell(a) = 0$. Then, the number of certain value x which all the symbols convert to is Φ. Therefore, $|\Phi_x| = |\Phi| - D_x$. In the mapping that converts the different symbols to the distinct numerical values, Φ_x equal to Φ.

$\Pr(X)$ denotes the probability of event X. Let A be the event $\phi(t_1) \cdot \overline{\phi(p_1)} = x$ and B the event $\phi(t_2) \cdot \overline{\phi(p_2)} = x$. And let A' be the event $\phi(t_1) = d_1$, A'' the event $\phi(p_1) = d_2$, B' the event $\overline{\phi(t_2)} = d_3$, and B'' the event $\overline{\phi(p_2)} = d_4$.

If a certain event occurred, that a result of a mapping was value x, the mapping in the next event is restricted to mappings which return value x. After the event A, a set of mappings is $\Phi_{d_1 d_2}$ because the mapping returned d_1 and d_2 were used in the event A. A probability that a mapping return a value x is (the number of combinations of the mapping and the symbol which can return x)/(the product of the number of mappings and the number of symbols). Then we have

$$\begin{aligned}
\Pr(B') &= \frac{\frac{1}{r} \cdot |\Phi| \cdot |\Sigma|}{|\Phi| \cdot |\Sigma|} = \frac{1}{r}, \\
\Pr(B''|B') &= \frac{\frac{1}{r} \cdot |\Phi| \cdot |\Sigma|}{|\Phi_{d_3}| \cdot |\Sigma|} = \frac{|\Phi|}{r \cdot |\Phi_{d_3}|}, \\
\Pr(B) &= \sum_{d_3=0}^{r-1} \Pr(B') \Pr(B''|B') = \sum_{d_3=0}^{r-1} (\frac{1}{r} \cdot \frac{|\Phi|}{r \cdot |\Phi_{d_3}|}) = \frac{1}{r^2} \sum_{d_3=0}^{r-1} (\frac{|\Phi|}{|\Phi_{d_3}|}),
\end{aligned}$$

and

$$\Pr(B|A) = \sum_{d_3=0}^{r-1} (\frac{|\Phi|}{r \cdot |\Phi_{d_1 d_2}|} \cdot \frac{|\Phi|}{r \cdot |\Phi_{d_1 d_2 d_3}|}) = \frac{1}{r^2} \sum_{d_3=0}^{r-1} (\frac{|\Phi|^2}{|\Phi_{d_1 d_2}| \cdot |\Phi_{d_1 d_2 d_3}|}).$$

We get $\Pr(B|A) \neq \Pr(B)$, hence $\phi(t_1) \cdot \overline{\phi(p_1)}$ is not independent of $\phi(t_2) \cdot \overline{\phi(p_2)}$. However, if $\Phi = \Phi_{d_1 d_2} = \Phi_{d_1 d_2 d_3}$, then $\Pr(B|A) = \Pr(B)$. This condition is satisfied only when all mappings should convert different symbols into distinct numerical values. $\square$

Other two mappings can not satisfy the condition of Lemma 4 while only our mappings can satisfy it in case of $|\Sigma| = p$. Therefore, we add a dummy symbol in case of $|\Sigma| < p$. Then we can correct a sampling bias because we can know that by the dummy symbol in advance.

When ϕ_ℓ is drawn uniformly randomly from Φ, the random variable $\hat{s}$ is $\hat{s} = \frac{1}{k} \sum_{\ell=1}^{k} \sum_{j=1}^{m} \phi_\ell(t_j) \cdot \overline{\phi_\ell(p_j)}$.

Then, we get the following lemma.

Lemma 5 Given that the product $\phi(a)\cdot\overline{\phi(b)}$ in one position is independent of that in other position. When c symbols align in the m symbols, the variance $V(\hat{s})$ of random variable s are

$$V(\hat{s}) = \frac{m - c_i}{k}.$$

Proof. Let s_j be the random variable as $\phi_\ell(t_j)\cdot\overline{\phi_\ell(p_j)}$, then $s_j = \phi_\ell(t_j)\cdot\overline{\phi_\ell(p_j)} = \omega^{d(t_j,p_j)}$ where $d(t_j,p_j) = x\cdot(\psi(t_j) - \psi(p_j))$. $s_{(t_j=p_j)}$ denotes that s in $t_j = p_j$ and $s_{(t_j\neq p_j)}$ denotes that s in $t_j \neq p_j$.

If $t_j = p_j$, $s_j = 1$. If $t_j \neq p_j$, $s_j = \omega^{d(t_j,p_j)}$. Then, those means are $E(s_{(t_j=p_j)}) = 1, E(s_{(t_j\neq p_j)}) = \sum_{x=0}^{p-1}\omega^{d(t_j,p_j)}\cdot\frac{1}{p} = 0$. And those variance are $V(s_{(t_j=p_j)}) = (s_{(t_j=p_j)} - E(s_{(t_j=p_j)}))^2\cdot 1 = (1-1)^2\cdot 1 = 0, V(s_{(t_j\neq p_j)}) = \sum_{x=0}^{p-1}(s_{(t_j\neq p_j)} - E(s_{(t_j\neq p_j)}))^2\cdot\frac{1}{p} = \frac{1}{p}\sum_{x=0}^{p-1}(|\omega^{d(t_j,p_j)}|)^2 = \frac{1}{p}\sum_{x=0}^{p-1}1 = 1$.

Because we assume that the product $\phi(a)\cdot\overline{\phi(b)}$ in one position is independent of that in other position, a variance $V(s)$ of s are the simple total of a variance of every position. Then, $V(s) = \sum^{c}V(s_{(t_j=p_j)}) + \sum^{m-c_i}V(s_{(t_j\neq p_j)}) = \sum^{c}0 + \sum^{m-c_i}1 = m - c_i$.

Using k samples s, a variance $V(\hat{s})$ of the estimate s is $V(\hat{s}) = \frac{1}{k}V(s)$. Then

$$V(\hat{s}) \;=\; \frac{m - c_i}{k}.$$

□

This analysis can be applied to the algorithm which improvement in Section 3 was added to.

Then Eq. (5) changes as follow,

$$\begin{aligned} V(s_{j(t_j\neq p_j)}) &= \sum_{x=0}^{p-1}(s_{j(t_j\neq p_j)} - E(s_{j(t_j\neq p_j)}))^2\frac{1}{p} \\ &= \frac{1}{p}\sum_{x=0}^{p-1}(\cos\frac{2\pi g(a,b)}{p} - 0)^2 \\ &= \frac{1}{p}\sum_{x=0}^{p-1}\cos^2\frac{2\pi g(a,b)}{p} \\ &= \frac{1}{p}\sum_{x=0}^{p-1}\frac{1+\cos\frac{2\pi g(a,b)}{p}}{2} \\ &= \frac{1}{2p}(\sum_{x=0}^{p-1}1 + \sum_{x=0}^{p-1}\cos\frac{2\pi g(a,b)}{p}) \\ &= \frac{1}{2} \end{aligned} \tag{7}$$

By Eq. (7), we analyze the variance as the proof of Lemma 5.

$$V(\hat{s}) = \frac{m - c_i}{2k}. \tag{8}$$

By Eq. (4) and Eq. (8), we get the following theorem.

Theorem 2 The variance of the estimates for the score in our algorithm is

$$V(\hat{s}) = \frac{(p-1)^2}{p^2}\cdot\frac{p-1-k}{p-2}\cdot\frac{m - c_i}{2k}.$$

Conclusion

We gave an efficient randomized algorithm for string matching with mismatches. This randomized algorithm uses convolution with FFT, like that proposed by Atallah et al. and Baba et al. We used the mappings which convert the symbols to the p-adic number field. One of the features of our mappings is that it does not convert some different symbols into same numerical value. By that feature, the variance of the estimate of the score vector is smaller. The other feature of our mappings is that there are not so many mappings. The number of mapping is $p-1$ where $|\Sigma| \leq p < 2|\Sigma|-2$.

We analyzed the variance of the estimates for the score in this algorithm. And it is very small as compared to the randomized algorithms proposed in the past. The variance in this algorithm is $\frac{(p-1)^2}{p^2} \cdot \frac{p-1-k}{p-2} \cdot \frac{m-c_i}{2k}$. Its time complexity is $O(kn \log m)$ where k is the number of samples, and the upper bound of k is $p-1$. When k is $p-1$, this algorithm is deterministic, and the estimate becomes the real value.

Experiments with read texts and the evaluation of computation time are future work. We have a plan to apply the method for pattern extraction from Web pages [8].

References

[1] Atallah, M. J., Chyzak, F., and Dumas, P.: A Randomized Algorithm for Approximate String Matching. Algorithmica 29, 468-486. 2001.

[2] Baba, K., Shinohara, A., Takeda, M., Inenaga, S., and Arikawa, S.: A Note on Randomized Algorithm for String Matching with Mismatches. Nordic Journal of Computing 10, 2-12. 2003.

[3] Baba, K., Tanaka, Y., Nakatoh, T., Shinohara, A.: A Unification of FFT Algorithm for String Matching. Proc. International Symposium on Information Science and Electrical Engineering 2003, to appear.

[4] Crochemore, M. and Rytter, W.: Text Algorithms. Oxford University Press, New York. 1994.

[5] Crochemore, M. and Rytter, W.: Jewels of Stringology. World Scientific. 2003.

[6] Fischer, M. J. and Paterson, M. S.: String-matching and other products. In Complexity of Computation (Proceedings of the SIAM-AMS Applied Mathematics Symposium, New York, 1973), 113-125. 1974.

[7] Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, New York. 1997.

[8] Taguchi, T., Koga, Y. and Hirokawa, S.: Integration of Search Sites of the World Wide Web. Proc. of International Forum cum Conference on Information Technology and Communication, Vol. 2, pp. 25-32, 2000.