Collaborative Report DC–99–05

Proceedings of the Prague Stringology Club Workshop '99

Edited by Jan Holub and Milan Šimánek

July 1999

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, Bruce W. Watson

Organizing Committee

Miroslav Balík, Martin Bloch, Jan Holub, Martin Rýzl, Milan Šimánek, Zdeněk Troníček

Table of contents

Preface

This collaborative report contains the proceedings of the Prague Stringology Club Workshop '99 (PSCW'99), held at the Department of Computer Science and Engineering of Czech Technical University in Prague on July 8–9, 1999. The workshop was preceded by PSCW'96 which was the first action of the Prague Stringology Club, by PSCW'97 and by PSCW'98. The proceedings of PSCW'96, PSCW'97 and PSCW'98 were published as collaborative reports DC–96–10, DC–97–03 and DC–98–06, respectively, of Department of Computer Science and Engineering and are also available in the postscript form at Web site with URL: `http://cs.felk.cvut.cz/psc`. While the papers of PSCW'96 were invited papers, the papers of PSCW'97 and PSCW'98 were selected from the papers submitted as a response to a call for papers. The papers in this proceedings are alphabetically ordered by the authors.

The PSCW aims at strengthening the connection between stringology (the computer science on strings and sequences) and finite automata theory. The automata theory has been developed and successfully used in the field of compiler construction and can be very useful in the field of stringology too. The automata theory can facilitate the understanding of existing algorithms and the developing of new algorithms.

Jan Holub and Milan Šimánek, editors

The Closest Common Subsequence Problems[1]

Gabriela Andrejková

Department of Computer Science, Faculty of Science, P. J. Šafárik University,
Jesenná 5, 041 54 Košice, Slovakia

e-mail: `andrejk@kosice.upjs.sk`

Abstract. Efficient algorithms are presented that solve general cases of the Common Subsequence Problems, in which both input strings contain symbols with *competence values* or sets of symbols with competence values. These problems arise from a searching of the sets of most similar strings.

Key words: Subsequence, common subsequence, measure of the string, dynamic programming, design and analysis of algorithms.

1 Introduction

The motivation to the CCS Problems can be found in the typing of a text on the keyboard. The following mistakes can be made in typing some string:

1. Typing a different character, usually from the neighbour area of the given character.
2. Inserting a single character into the source string.
3. Omiting (skipping) any single source character.

In the most frequent mistakes, a character from the area on the keyboard adjacent to the required character was typed instead of the required character. For example, the neighborhood of the character *f* is the set $\mathbf{f} = \{$f,d, g, r, t, c, v$\}$. The sequence of sets $\mathbf{A} =$ **f, r, e, s, c, o** belongs to the word *fresco*. In this case (typing mistakes) let us assign *competence value (c.v.)* to each element of the neighborhood in such way that the character itself has c.v. 1 and the c.v.'s of "more erroneous" character are smaller than those of the "better one". For example, for set **f** we have $\mu(f) = 1, \mu(d) = 0.4, \mu(g) = 0.4, \mu(r) = 0.2, \mu(t) = 0.4, \mu(c) = 0.3, \mu(v) = 0.3$. We consider that in the text, it is necessary to find the words which are very close to the word *fresco*. We consider the sum of c.v.'s of a given string as a measure of its similarity of the string to the given word *fresco*. The lengths of the found words can be different to the length of the given word *fresco*. For example, if the word *fresco* is found in the text then the measure of the similarity to the given word *fresco* is the length of the word *fresco* (6), if the word *tresc* is found then the measure of the similarity is 4.4 because the symbol t is very close to the symbol f and symbol o is omitted.

[1] This research was partially supported by Slovak Grant Agency for Science VEGA, project No. 1/4375/97

It is possible to consider the described problem as the *closest common subsequence problem* of the two similar strings and its repetition for text of strings.

The common subsequence problem of two strings is to determine one of the subsequences that can be obtained by deleting zero or more symbols from each of the given strings. It is possible to demand some additional properties for the common subsequence. Usually, it is the greatest length of the common subsequence, but we can consider some different measures for the common subsequence.

The longest common subsequence problem (*LCS Problem*) of two strings is to determine the common subsequence with the maximal length. For example, the string *AGI* is a common subsequence and the string *ALGI* is the longest common subsequence of the strings *ALGORITHM* and *ALLEGATION*. Algorithms for this problem can be used in chemical and genetic applications and in many problems concerning data and text processing [15], [12], [3]. Further applications include the string-to-string correction problem [12] and determining the measure of differences between text files [3]. The length of the longest common subsequence (*LLCS Problem*) can determine the measure of differences (or similarities) of text files. The simulation method for the approximate strings and sequence matching using the Levenstein metric can be found in J. Holub [9] and the algorithm for the searching of the subsequences is in Z. Troníček and B. Melichar [16].

D. S. Hirschberg and L. L. Larmore [7] have discussed a generalization of LCS Problem, which is called Set LCS Problem (*SLCS Problem*) of two strings where however the strings are not of the same type. The first string is a sequence of symbols and the second string is a sequence of subsets over an alphabet Ω. The elements of each subset can be used as an arbitrary permutation of elements in the subset. The longest common subsequence in this case is a sequence of symbols with maximal length. The SLCS Problem has an application to problems in computer driven music [7]. D. S. Hirschberg and L.L. Larmore have presented $O(m \cdot n)$-time and $O(m+n)$-space algorithm, m, n are the lengths of the strings. The Set-Set LCS Problem (*SSLCS Problem*) is discussed by D. S. Hirschberg and L. L. Larmore [8]. In this case both strings are strings of subsets over an alphabet Ω. In the paper [8] is presented the $O(m \cdot n)$-time algorithm for the general SSLCS Problem.

In this paper we present algorithms for general cases of the Common Subsequence Problem, it means Closest Common Subsequence Problems: *CCS Problem* (for two strings of symbols), *CCRS Problem* (for two strings of symbols with restricted using of the symbols), *SCCS Problem* (for one string of symbols and second string of symbol sets) and *SSCCS Problem* (for two strings of symbol sets).

2 Basic Definitions

In this section, some basic definitions and results concerning to CCS Problem, SCCS and SSCCS Problem are presented.

Let Ω be a finite alphabet, $|\Omega| = s, P(\Omega)$ the set of all subsets of $\Omega, |P(\Omega)| = 2^s$.

Let $A = a_1 a_2 \dots a_m, a_i \in \Omega, 1 \le i \le m$ be a string over an alphabet Ω, where $|A| = m$ is the length of the string A.

Let $\mu_A(a_i) \in (0,1\rangle, 1 \le i \le m$, be some competence (membership) values of elements in the string A.

The pair (A, μ_A) is *the string A with the competence function* μ_A, *cf-string* (A, μ_A) for short. $Val(A, \mu_A)$ is a measure of (A, μ_A) defined by the (1).

$$Val(A, \mu_A) = \Sigma_{i=1}^{m} \mu_A(a_i) \tag{1}$$

The string $C \in P(\Omega), C = c_1 \dots c_p$ is *a subsequence* of the string $A = a_1 \dots a_m$, if a monotonous increasing sequence of natural numbers $i_1 < \dots < i_p$ exists such that $c_j = a_{i_j}, 1 \leq j \leq p$. The string C is *a common subsequence* of two strings A, B if C is a subsequence of A and C is a subsequence of B. $|C|$ is the length of the common subsequence. The classical problem to find the longest common subsequence is defined and solved in Hirschberg [5].

The string (C, μ_C) is *a subsequence with the competence function* μ_C, *cf-subsequence for short* of the cf-string (A, μ_A) if C is a subsequence of the string A and $0 < \mu_C(c_t) \leq \mu_A(a_{i_t})$, for $1 \leq t \leq p$. The cf-subsequence (C, μ_C) is *a closest cf-subsequence* if $Val(C, \mu_C) = \Sigma_{j=1}^{p} \mu_C(c_j) = \Sigma_{j=1}^{p} \mu_A(a_{i_j})$.

The string (C, μ_C) is *a common cf-subsequence* of two cf-strings (A, μ_A) and (B, μ_B) if (C, μ_C) is a cf-subsequence of (A, μ_A) and (C, μ_C) is a cf-subsequence of (B, μ_B).

The string (C, μ_C) is a *closest common cf-subsequence* of the cf-strings (A, μ_A) and (B, μ_B) if (C, μ_C) is a common cf-subsequence with the maximal value $Val(C, \mu_C)$. It means, if (D, μ_D) is a common cf-subsequence of the strings (A, μ_A) and (B, μ_B) then $Val(D, \mu_D) \leq Val(C, \mu_C)$.

If (C, μ_C) is a closest common cf-subsequence of the cf-strings, (A, μ_A) and (B, μ_B) then $\mu_C(c_t) = min\{\mu_A(a_{k_t}), \mu_B(b_{l_t})\}$, for $1 \leq t \leq p$.

The CCS Problem: Let (A, μ_A) and (B, μ_B) be cf-strings. To find a closest common subsequence of the cf-strings (A, μ_A) and (B, μ_B), $CCS((A, \mu_A), (B, \mu_B))$ for short.

The MCCS Problem is to find the measure of CCS cf-string, $MCCS$ for short. It means, $MCCS((A, \mu_A), (B, \mu_B)) = Val(CCS((A, \mu_A), (B, \mu_B)))$. $\diamond$

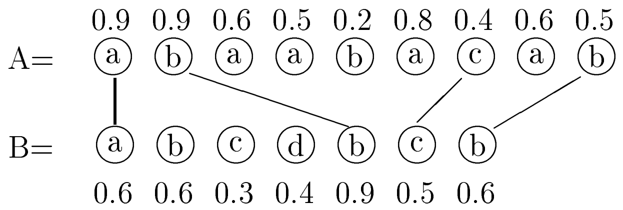

Figure 1. The closest common subsequence of two cf-strings A and B.

Example 1. $\Omega = \{a, b, c\}$, $A = abaabacab$, $m = 9$, $B = abcdbcb$, $n = 7$, $\mu_A = (0.9, 0.9, 0.6, 0.5, 0.2, 0.8, 0.4, 0.6, 0.5)$, $\mu_B = (0.6, 0.6, 0.3, 0.4, 0.9, 0.5, 0.6)$. The string $C = abcb$ is a subsequence, $C' = abbcb$ is the longest common subsequence of the strings A and B, and $(C", \mu_{C"})$, $C" = abcb, \mu_{C"} = (0.6, 0.9, 0.4, 0.5)$ is the closest common subsequence of the cf-strings (A, μ_A) and $(B, \mu_B), Val(C", \mu_{C"}) = MCCS((A, \mu_A), (B, \mu_B)) = 2.4$ as it is shown in the Figure 1.

Let (A, μ_A) be the string A with the competence function μ_A. A sequence of indices, $\mathrm{h}^A = h_0^A h_1^A h_2^A \dots h_{k^A}^A, 0 = h_0^A < h_1^A < h_2^A < \dots < h_{k^A}^A = m, 1 \leq k^A \leq m$ is *a partition of the string* (A, μ_A).

The sequence h^{A} divides the string (A, μ_A) in the following way:

$A = |a_1a_2 \dots a_{h_1^A}|a_{h_1^A+1} \dots a_{h_2^A}| \dots |a_{h_{k-1}^A+1} \dots a_{h_{k^A}^A}| = subst_1^A subst_2^A \dots subst_{k^A}^A$, where $subst_i^A = a_{h_{i-1}^A+1} \dots a_{h_i^A}, 1 \leq i \leq k^A$. $[(A, \mu_A), h^A]$ is called *the cf-string with the partition.*

For example, $\Omega = \{a, b, c\}$, $A = |aba|abacac|bab|, m = 12$, $\mu_A = (0.4, .2, .8, .4, .7, .3, .3, .7, .5, .4, .8, .6)$, $h^A = 0, 3, 9, 12$; $subst_1^A = aba$, $subst_2^A = abacac$, $subst_3^A = bab$.

A string $C = c_1c_2 \dots c_p, 1 \leq p \leq m$ is *a restricted subsequence* of the cf-string with the partition $[(A, \mu_A), \mathrm{h}^{\mathrm{A}}]$, if and only if

1. there exists a sequence of indices $1 \leq i_1 < i_2 < \dots < i_p \leq m$ such that $a_{i_t} = c_t, 1 \leq t \leq p$, and

2. if $h_{r-1}^A < i_u, i_v \leq h_r^A$ then $c_u \neq c_v$, for all r, $1 \leq r \leq k^A$,
(each element of an alphabet $\Omega(subst_r^A)$ can be used in C once at most).

The string (C, μ_C) is *a common restricted cf-subsequence* of two cf-strings with partition $[(A, \mu_A), \mathrm{h}^{\mathrm{A}}]$ and $[(B, \mu_B), \mathrm{h}^{\mathrm{B}}]$ if (C, μ_C) is a restricted cf-subsequence of $[(A, \mu_A), \mathrm{h}^{\mathrm{A}}]$ and (C, μ_C) is a restricted cf-subsequence of $[(B, \mu_B), \mathrm{h}^{\mathrm{B}}]$ at once.

The string (C, μ_C) is *a closest common restricted cf-subsequence* of two cf-strings with partition $[(A, \mu_A), \mathrm{h}^{\mathrm{A}}]$ and $[(B, \mu_B), \mathrm{h}^{\mathrm{B}}]$ if (C, μ_C) is a common restricted cf-subsequence with maximal value defined by (1).

The CCRS Problem: Let $[(A, \mu_A), h^A]$ and $[(B, \mu_B), h^B]$ be the cf-strings. To find the closest common subsequence of the cf-strings $[(A, \mu_A), h^A]$ and $[(B, \mu_B), h^B]$, $CCRS([(A, \mu_A), h^A], [(B, \mu_B), h^B])$ for short.

The MCCRS Problem is to find the measure of CCRS cf-string, $MCCRS$ for short. It means, $MCCRS([(A, \mu_A), h^A], [(B, \mu_B), h^B]) = Val(CCRS([(A, \mu_A), h^A], [(B, \mu_B), h^B]))$. $\diamond$

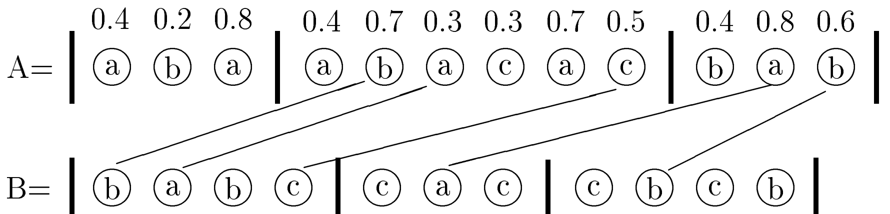

Figure 2. Closest common restricted subsequence of two strings A and B.

Example 2. $\Omega = \{a, b, c\}$, $A = |aba|abacac|bab|, m = 12$, $\mu_A = (0.4, 0.2, 0.8, 0.4, 0.7, 0.3, 0.3, 0.7, 0.5, 0.4, 0.8, 0.6)$, $h^A = 0, 3, 9, 12$; $B = |babc|cac|cbcb|, n = 11$, $\mu_B = (0.4, 0.3, 0.4, 0.5, 0.3, 0.5, 0.6, 0.3, 0.7, 0.6, 0.5)$. The string $C = bacb$ is a restricted subsequence, $C' = bacab$ is the closest restricted common subsequence with measure 2.3 as it can be seen in Figure 2. The string $C" = babcacbb$ is the longest common subsequence of the strings $A = abaabacacbab$ and $B = babccaccbcb$ if the partition does not matter.

A string of sets, set-string for short, $\mathcal{B}$ over an alphabet Ω is any finite sequence of sets from $P(\Omega)$. Formally, $\mathcal{B} = B^1B^2 \dots B^n, B^i \in P(\Omega), 1 \leq i \leq n$, n is the number of sets in $\mathcal{B}$. The length of the symbol string described by $\mathcal{B}$ is $N = \Sigma_{i=1}^{n}|B^i|$. The pair $(\mathcal{B}, \mu_{\mathcal{B}})$ is the set-string $\mathcal{B}$ with the competence functions $\mu_{\mathcal{B}}$, set-cf-string for short.

A string of symbols $C = c_1c_2\dots c_p, c_i \in \Omega, 1 \le i \le p$, is a *subsequence of symbols* (subsequence, for short) of the set-string $\mathcal{B}$ if there is a nonincreasing mapping $F : \{1, 2, \dots, p\} \to \{1, 2, \dots, n\}$, such that

1. if $F(i) = k$ then $c_i \in B_k$, for $i = 1, 2, \dots, p$
2. if $F(i) = k$ and $F(j) = k$ and $i \ne j$ then $c_i \ne c_j$.

The combination of a string and a set-string and the finding of their closest common cf-subsequence leads to the solution of problems in above motivation.

Let (A, μ_A), be cf-string over Ω and $(\mathcal{B}, \mu_{\mathcal{B}})$ be a set-cf-string over $P(\Omega)$. The cf-string (C, μ_C) is *a common cf-subsequence* of (A, μ_A) and $(\mathcal{B}, \mu_{\mathcal{B}})$ if (C, μ_C) is a cf-subsequence of A and (C, μ_C) is a cf-subsequence of the set-string $\mathcal{B}$. A *closest common cf-subsequence* of the cf-string (A, μ_A) and the set-cf-string $(\mathcal{B}, \mu_{\mathcal{B}})$, SCCS$((A, \mu_A), (\mathcal{B}, \mu_{\mathcal{B}}))$ is a common cf-subsequence (C, μ_C) with the maximal value $Val(C, \mu_C)$. Note that (C, μ_C) is not unique in general way.

The SCCS Problem: The Set closest Common Subsequence problem of the cf-string (A, μ_A) and the set-cf-string $(\mathcal{B}, \mu_{\mathcal{B}})$, SCCS$((A, \mu_A), (\mathcal{B}, \mu_{\mathcal{B}}))$ for short, consists of finding a closest common cf-subsequence (C, μ_C).

The MSCCS Problem consists of finding the measure of $SCCS$ cf-string, $MSCCS$ for short.
This means that $MSCCS((A, \mu_A), (\mathcal{B}, \mu_{\mathcal{B}})) = Val(SCCS((A, \mu_A), (\mathcal{B}, \mu_{\mathcal{B}})))$, $\diamond$

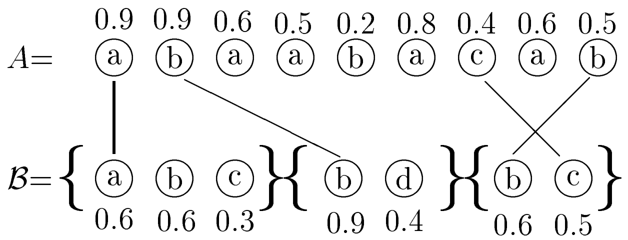

Figure 3. The closest common subsequence of two strings A and $\mathcal{B}$.

Example 3. Let $A = abaabacab, \mu_A = (0.9, 0.9, 0.6, 0.5, 0.2, 0.8, 0.4, 0.6, 0.5)$, $\mathcal{B} = \{a, b, c\}\{b, d\}\{b, c\}$, $\mu_{B^1}(a) = 0.6, \mu_{B^1}(b) = 0.6, \mu_{B^1}(c) = 0.3, \mu_{B^2}(b) = 0.9, \mu_{B^2}(d) = 0.4, \mu_{B^3}(b) = 0.6, \mu_{B^3}(c) = 0.5$. Then $MSCCS((A, \mu_A), (\mathcal{B}, \mu_{\mathcal{B}})) = 2.4$ as it is shown in the Figure 3.

Let $\mathcal{A} = A^1 \dots A^m, \mathcal{B} = B^1 \dots B^n, 1 \le m \le n$, be two set-strings of sets over an alphabet Ω. The string of symbols C is *a common subsequence of symbols* of $\mathcal{A}$ and $\mathcal{B}$ is C a subsequence of symbols of $\mathcal{A}$ and C is a subsequence of symbols of the set-string $\mathcal{B}$. *The longest common subsequence problem* of the set-strings $\mathcal{A}$ and $\mathcal{B}$ (SSLCS$(\mathcal{A}, \mathcal{B})$ consists of finding a common subsequence of symbols C of the maximal length. Note that C is not in general unique.

The SSCCS Problem: Let $(\mathcal{A}, \mu_{\mathcal{A}}), (\mathcal{B}, \mu_{\mathcal{B}})$ be two set-cf-string.
The Set-Set Closest Common Subsequence problem of the set-cf-strings $(\mathcal{A}, \mu_{\mathcal{A}})$ and $(\mathcal{B}, \mu_{\mathcal{B}})$, (SSCCS$((\mathcal{A}, \mu_{\mathcal{A}}), (\mathcal{B}, \mu_{\mathcal{B}}))$ for short, consists of finding a closest common cf-subsequence (C, μ_C).

The MSSCCS Problem consists of finding the measure of SSCCS set-cf-string, $MSSCCS$ for short.
It means, $MSSCCS((\mathcal{A},\mu_{\mathcal{A}}),(\mathcal{B},\mu_{\mathcal{B}})) = Val(SSCCS((\mathcal{A},\mu_{\mathcal{A}_t}),(\mathcal{B},\mu_{\mathcal{B}}))),\quad \diamond$

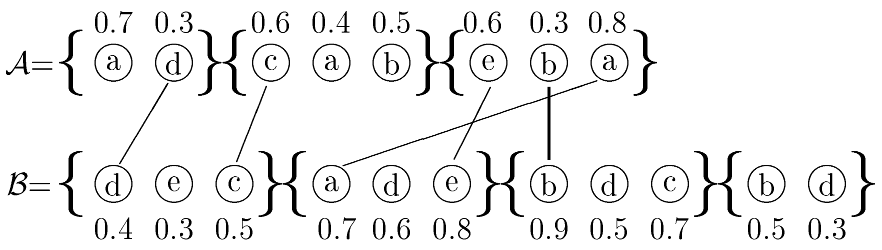

Figure 4. The closest common subsequence of two set-strings $\mathcal{A}$ and $\mathcal{B}$.

Example 4. Let $\mathcal{A} = \{a,d\},\{c,a,d\},\{e,b,a\}, m = 3$, $\mu_{A^1} = (0.7,0.3), \mu_{A^2} = (0.6,0.4,0.5), \mu_{A^3} = (0.6,0.3,0.8)$, $\mathcal{B} = \{d,e,c\},\{a,d,e\},\{b,d,c\},\{b,d\}, n = 4$. $\mu_{B^1} = (0.4,0.3,0.5), \mu_{B^2} = (0.7,0.6,0.8), \mu_{B^3} = (0.9,0.5,0.7), \mu_{B^4} = (0.5,0.3)$. The competence values are described according to the named order in the set. For example, $\mu_{A^1}(a) = 0.7, \mu_{A^1}(d) = 0.3$.

Then $MSSCCS((\mathcal{A},\mu_{\mathcal{A}}),(\mathcal{B},\mu_{\mathcal{B}})) = 2.4$ as it is shown in the Figure 4.

3 Algorithm for MCCS Problem

From the definition of $MSSC$ Problem it follows:

$$\begin{aligned} MCCS((A,\mu_A),(B,\mu_B)) = \;& max_{(C,\mu_C)}\{Val(C,\mu_C) : (C,\mu_C)\, is\; the\; common \\ & cf - subsequence\; of\; (A,\mu_A)\, and\, (B,\mu_B)\} \end{aligned} \tag{2}$$

The expression (2) can be written in the following way

$$\begin{aligned} &= max_{(C,\mu_C)}\{\Sigma_{t=1}^{p}\mu_C(c_t) : c_t = a_{k_t} = b_{l_t}, 1 \le t \le p, 1 \le k_1 < \ldots < k_p \le m, \\ &\quad 1 \le l_1 < \ldots < l_p \le n\}\, and\, 0 < \mu_C(c_t) = min\{\mu_a(a_{k_t}),\mu_B(b_{l_t})\}. \end{aligned} \tag{3}$$

It means

$$\begin{aligned} MCCS((A,\mu_A),(B,\mu_B)) = max\{\Sigma_{t=1}^{p} min\{\mu_A(a_{k_t}),\mu_B(b_{l_t})\} : a_{k_t} = b_{l_t}, \\ 1 \le t \le p, 1 \le k_1 < \ldots < k_p \le m, 1 \le l_1 < \ldots < l_p \le n\} \end{aligned} \tag{4}$$

Let M_{min} be a matrix defined as follows:

$$M_{min}[i,j] = \begin{cases} min\{\mu_A(a_i),\mu_B(b_j),\}, & \text{if } a_i = b_j \\ 0, & \text{otherwise.} \end{cases} \tag{5}$$

The expression (4) is the basis for the following algorithm and it should be written now in the following way:

$$\begin{aligned} MCCS((A,\mu_A),(B,\mu_B)) = max\{\Sigma_{t=1}^{p} M_{min}[k_t,l_t] : \\ k_1 < \ldots < k_p \le m, 1 \le l_1 < \ldots < l_p \le n\} \end{aligned} \tag{6}$$

The expression (6) can be used in the recursive algorithm or nonrecursive algorithm using the method of dynamic programming.

Designation.

- $A[i..k] = a_i a_{i+1} \dots a_k$, for $1 \le i \le k \le m$,
- $MM[m,n] = MCCS((A,\mu_A),(B,\mu_B))$,
- $MM[i,j] = MCCS((A[1..i],\mu_A),(B[1..j],\mu_B))$.

Recursive version of the algorithm is constructed according to the following idea:
If an element c_t is in the $CCS((A,\mu_A),(B,\mu_B))$ then the strings can be split into two parts and

$$\begin{aligned} MCCS((A,\mu_A),(B,\mu_B)) = \mu(c_t) \quad &+MCCS((A[1..k_{t-1}],\mu_A),(B[1..l_{t-1}],\mu_B)) \\ &+MCCS((A[k_{t+1}..m],\mu_A),(B[l_{t+1}..n],\mu_B)) \quad (7) \end{aligned}$$

The recursive version of the algorithm has exponential time complexity. Some computations are repeated and it means in the algorithm, it is possible to use the dynamic programming method to compute the partial values $MM[i,j]$ once only and to use them in the following computations.

In the algorithm, two functions are used: The function *Minim* computes minimum of two values, the function *Maxim* computes maximum of three values. The $i-$th line of the matrix MM is computed from two lines $(i-1)-$th and the already computed part of $i-$th column. It means that the space complexity of the algorithm can be reduced to $O(n)$, for $m \le n$. The algorithm works in the $O(m*n)$ time. It can be written in the following simple form (without the construction of the matrix M_{min}):

Algorithm MCCS:

```
for i:=0 to m do MM[i,0]:=0;
for j:=1 to n do MM[0,j]:=0;

for i:=1 to m do
   for j:=1 to n do
   begin
      if a[i]=b[j] then help:=MM[i-1,j-1] + Minim(miA[i],miB[j])
                   else help:=0;
      MM[i,j]:= Maxim(MM[i-1,j], help, MM[i, j-1]);
   end;
```

Example 5. The computation of $MCCS((A,\mu_A),(B,\mu_B))$ for the strings in Example 1 according to the algorithm MCCS.

```
           B=  0.6    0.6    0.3    0.4    0.9    0.5    0.6
                a      b      c      d      b      c      b
     A=    -------------------------------------------------
    a 0.9 |  0.6    0.6    0.6    0.6    0.6    0.6    0.6
    b 0.9 |  0.6    1.2    1.2    1.2    1.5    1.5    1.5
    a 0.6 |  0.6    1.2    1.2    1.2    1.5    1.5    1.5
```

a	0.5	0.6	1.2	1.2	1.2	1.5	1.5	1.5
b	0.2	0.6	1.2	1.2	1.2	1.5	1.5	1.7
a	0.8	0.6	1.2	1.2	1.2	1.5	1.5	1.7
c	0.4	0.6	1.2	1.5	1.5	1.5	1.9	1.9
a	0.6	0.6	1.2	1.5	1.5	1.5	1.9	1.9
b	0.5	0.6	1.2	1.5	1.5	2.0	2.0	2.4

4 Algorithm for MCCRS Problem

The basic idea to the solution can be found in [1]. The algorithm for LRCS Problem have to be modified in the computation of the the measure of closest common restricted subsequence. In the algorithm, the Boolean function *Candidate* gives the value *true* if the pair $(a_i, \mu(a_i)), (b_j, \mu(b_j))$ is a potential candidate to increase the closest common subsequence, *false* otherwise. The function *Candidate* is used in the same form as in [1]. The main part of the modification is designed in the program text. It could be proved (similar as for LRCS Algorithm in [1]) that the modified algorithm computes correctly the closest common restricted subsequence of two cf-strings and it works in $O(m \cdot n \cdot p)$-time and $O(n + r)$-space, where $r = |\{\langle i, j\rangle : a_i = b_j, 1 \leq i \leq m, 1 \leq j \leq n\}|$ and $p \leq min\{m, n\}$ is the number of elements in closest common restricted subsequence.

The following dynamic data structures are used in the algorithm:

```
type vertex=record
                x, y: indices;
                p: pointer;
            end;
     pointerv=^vertex;
     genseq=record
                val: real;
                pt:pointer;
            end;
```

The main phase of the algorithm is the following:

```
{Omega is an alphabet of strings}
{Input: [(A, mvA), hA], [(B,mvB), hB] - two cf-strings of symbols
                           with partitions over alphabet;
                           mvA, mvB - competence functions of A and B}
{Output: pptr is the pointer to the closest common restricted
                           subsequence of A and B;}
{Variables: Arrays C, D[0..m] of the type genseq.}
{C[1..i], D[1..i] contain pointers to the closest common
                           subsequences of A(1..i) and B(1..j);}
{hA[1..kA], hB[1..kB] - arrays of partitions of the strings A and B;}
{uA, uB - upper bounds of intervals in the partitions for current
                           positions i, j: uA\leq i, uB\leq j.}
{dA, dB - the numbers of intervals in the partitions,}
{pp - a pointer to the vertex.}
```

```
Method:
begin
        for j:=0  to n do
        begin D[j].pt:=nil; D[j].val:=0; end;
        C[0].pt:=nil; C[0].val:=0;
        dA:=1; uA:=1;
        for i:=1 to m do
        begin if i>hA[dA] then begin inc(dA); uA:=hA[dA-1]+1 end;
           dB:=1; uB:=1;
           for j:=1 to n do
           begin if j>hB[dB] then begin inc(dB); uB:=hB[dB-1]+1 end;
              if a[i].el=b[j].el then
                         q:=Candidate(D[j-1].pt,a[i],uA,uB)
                               else q:=false;
              if q then {***modified part***}
              begin if a[i].mv<=b[j].mv then min:=a[i].mv
                                   else min:=b[j].mv;
                 help:=D[j-1].val+min;
                 if (help>D[j].val) and (help>C[j-1].val) then
                 begin new(pp); pp^.p:=D[j-1].pt; pp^.x:=i; pp^.y:=j;
                    C[j].pt:=pp; C[j].val:=D[j-1].val+min;
                 end    {***end of the modified part***}
               end  else
               if D[j].val>=C[j-1].val then C[j]:=D[j]
                                     else C[j]:=C[j-1];
               {Invariant1}
            end; {Invariant2}
            for j:=1 to n do D[j]:=C[j];
         end;
         value := C[n].val; pptr:= C[n].pt;
{"value" contains the value of the closest common restricted
subsequence and C[n].pt contains pointer to the CCRS(A,B)}
end;
```

Example 6. The computation of $MCCRS([(A,\mu_A),h^A],[(B,\mu_B),h^B])$ for the strings in Example 2 according to the algorithm MCCRS.

B / A	0.4 b	0.3 a	0.4 b	0.5 c	0.3 c	0.5 a	0.6 c	0.3 c	0.7 b	0.6 c	0.5 b
a 0.4	0.0	0.3	0.3	0.3	0.3	0.4	0.4	0.4	0.4	0.4	0.4
b 0.2	0.2	0.3	0.5	0.5	0.5	0.5	0.5	0.5	0.6	0.6	0.4
_a_0.8_	0.2	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.6	0.6	0.6
a 0.4	0.2	0.5	0.5	0.5	0.5	0.9	0.9	0.9	0.9	0.9	0.9
b 0.7	0.4	0.5	0.5	0.5	0.5	0.9	0.9	0.9	1.6	1.6	0.9
a 0.3	0.4	0.7	0.7	0.7	0.7	0.9	0.9	0.9	1.6	1.6	1.6
c 0.3	0.4	0.7	0.7	1.0	0.7	0.9	1.2	0.9	1.6	1.9	1.9

a 0.7 |	0.4	0.7	0.7	1.0	1.0	1.0	1.2	1.2	1.6	1.9	1.9
_c_0.5_|	0.4	0.7	0.7	1.2	1.2	1.2	1.2	1.2	1.6	2.1	2.1
b 0.4 |	0.4	0.7	0.7	1.2	1.2	1.2	1.2	1.2	1.6	2.1	2.1
a 0.8 |	0.4	0.7	0.7	1.2	1.2	1.7	1.7	1.7	1.7	2.1	2.1
_b_0.6_|	0.4	0.7	0.7	1.2	1.2	1.7	1.7	1.7	2.3	2.3	2.3

5 Algorithm for MSCCS Problem

The basic idea of the algorithm starts from the definition of the $MSCCS$ Problem.

$$MSCCS((A,\mu_A),(\mathcal{B},\mu_{\mathcal{B}})) = max_{(C,\mu_C)}\{Val(C,\mu_C) : (C,\mu_C) \text{ is the common cf-subsequence of } (A,\mu_A)\, and\, (\mathcal{B},\mu_{\mathcal{B}})\} = \tag{8}$$

$$max_p\{\Sigma_{t=1}^{p}\mu_C(c_t) : c_t = a_{k_t} = b_i^{F(t)}\, and\, 0 < \mu_C(c_t) = min\{\mu_A(a_{k_t}),\mu_{\mathcal{B}}(b_i^{F(t)})\}, 1 \le t \le p, 1 \le k_1 < \ldots < k_p \le m, 1 \le i \le n_{F(t)}, 1 \le F(1) \le \ldots \le F(p) \le n\} \tag{9}$$

The recursive version of the algorithm is constructed according to the following idea (Figure 5.):

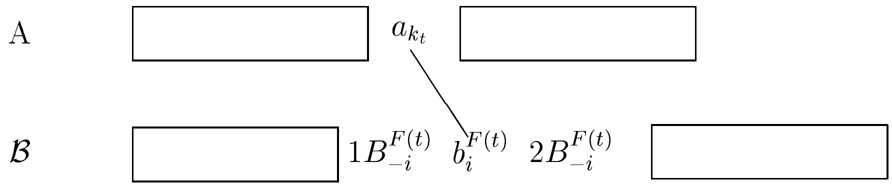

Figure 5. The idea for the construction of algorithm

Designation.

- $A = a_1 \ldots a_m, m \ge 1, \mathcal{B} = B^1 \ldots B^n, n \ge 1, B^l = \{b_1^l, b_2^l, \ldots, b_{n_l}^l\}$,
- $MM[m,n] = MSCCS((A,\mu_A),(\mathcal{B},\mu_{\mathcal{B}}))$,
- $MM[i,j] = MSCCS((A[1..i],\mu_A),(\mathcal{B}[1..j],\mu_{\mathcal{B}}))$.

If an element c_t is in the $SGCD((A,\mu_A),(\mathcal{B},\mu_{\mathcal{B}}))$ then

$$MSCCS((A,\mu_A),(\mathcal{B},\mu_{\mathcal{B}})) = \mu(c_t) + max\{MSCCS((A[1..k_{t-1}],\mu_A),(\mathcal{B}[1..F(t-1)]1B_{-i}^{F(t)},\mu_B)) + MCCS((A[k_{t+1}..m],\mu_A),(\mathcal{B}[F(t+1)..n]2B_{-i}^{F(t)},\mu_B))\} \tag{10}$$

where $1B_{-i}^{F(t)} = (B^{F(t)} - \{b_i^{F(t)}\})^1$ and $2B_{-i}^{F(t)} = (B^{F(t)} - \{b_i^{F(t)}\})^2$ are the disjoint subsets $1B_{-i}^{F(t)}$ and $2B_{-i}^{F(t)}$ of the set $(B^{F(t)} - \{b_i^{F(t)}\}) = 1B_{-i}^{F(t)} \cup 2B_{-i}^{F(t)}$ and the maximum is the maximal value over all disjoint partitions. The idea is shown in the Figure 6. The time complexity of the recursive version is exponential.

A *flattening* of a sequence of sets is defined as a concatenation, in order of the sequence, of strings formed by some permutation of individual elements of the sets in the sequence. For example, the flattening of the set-string $\mathcal{A}$ in example 3 is *dadacabe* and so is *adadceba*.

The very simple algorithm for MSCCS Problem can use Algorithm for MCCS Problem for all pairs of the cf-string A and the flattening of the set-cf-string $\mathcal{B}$. The algorithm have to compute and compare results of $\Pi_{j=1}^{n}|B_j|$ pairs.

It is possible to represent the sets in the string $\mathcal{B}$ as the strings of symbols with all permutations of elements (the method will be applied in the MSSCCS Algorithm). Each element of the string of symbols has the competence value the same as it has in the set. Then it is possible to apply the algorithm for common subsequence with a restricted use of elements [1].

The nonrecursive algorithm is constructed by the dynamic programming method and it has the following idea:

$$\begin{aligned} MM[i,j] = max\{ \quad & MM[k-1,j-1] + Val(SCCS((A[k..i],\mu_A),(B^j,\mu_{B^j}))), \\ & MM[k,j-1], k = 1,2,\ldots,i\}. \end{aligned} \qquad (11)$$

The values of the matrix $MM[*,*]$ can be computed according to columns, the input for j-th column is the matrix $(j-1)$-th column. The set B^j can match better some elements in the string A than the sets $B^1,\ldots,B^{j-1}$ and it is necessary to compute these matching values and to find the maximal value.

The following algorithm has a motivation in Hirschberg's and Larmore's method [7] for SLCS Problem. We use the a data structure U, which is called *unique stack* (for control of elements from the sets), but our unique stack works in a different way. It has the condition that no member can occur twice or more in the stack. When *Push(U, x, k)* is executed for some element x, x is first compared to the elements in the stack. If x is in the stack in the position l then the competence values of the both occurrences are compared. If the competence value of the element x in the position l is greater than the competence value of the new element x then the unique stack is not modified else the element in the position l is deleted and the new element x is pushed on the top of the unique stack. In the stack are the elements of the string A which have best matching to the some set in the string of sets $\mathcal{B}$.

```
procedure Push(var U:Ustack; x:Element; k:integer);
{Push the element x on the top of the unique stack U;
 k is the index of x in the string A;
 Competence values are less than Maxi1000;}
var Upom: Ustack;
    tophlp: integer;
    kk: integer;
begin
       kk:=top;
       tophlp:=0;
       Maxi:=Max1000;
       while kk>=1 do
       begin if (x.p<>U[kk].p) then
             begin inc(tophlp); Uhlp[tophlp]:=U[kk];
             end else begin
```

```
                           Maximum:=U[kk].mi;
                           if Maximum<x.mi then Maximum:= x.mi;
                           if Maximum>x.mi then
                           begin inc(tophlp);
                                Uhlp[tophlp]:=U[kk];
                                Maxi:=Maximum;
                           end;
                       end;
             dec(kk);
       end;
       top:=0;
       for kk:=tophlp downto 1 do
       begin inc(top); U[top]:=Uhlp[kk]; end;
       if (Maxi<x.mi) or (Maxi=Max1000) then
       begin inc(top); U[top]:= x; best[x.p]:=k;
       end;
end; {Push}
```

The procedure *Findpeaks* searches for the values $peak[k], \ldots, peak[0]$ which can represent measures of the new candidates for $SCCS$. In *Findpeak*, as k decreases, U is the list of all elements in B_j which are found in the substring $A[k+1..m]$ in the order in which they first occur and according to their competence function. For any $x \in U$, $first[x]$ is the index of that best occurrence.

```
procedure Findpeak(j: integer);

{ j - index of j-th set in the set-string B;
  m - the length of the symbol string A;
 top- global variable for the top of Unique stack.}

begin
     top:=0;
     for k:=m downto 0 do
     begin measure:=Mi[k,j-1];
          peak[k]:=measure;
          for x:=top downto 1 do
          begin xx:=U[x].p;
               Minimum:= Minim(U[x],B[j]);
               measure:=measure+Minimum;
               peak[best[xx]]:= Maxim{measure,peak[best[xx]]};
          end;
          if k>0 then
               if A[k].p in B[j].pp then Push(U,A[k],k);
     end;
end;
```

The main algorithm has the following form:

Algorithm MSCCS:

```
for i:=0 to m do MM[i,j]:=0;
for j:=1 to n do
     begin Findpeak(j);
          MM[0,j]:=0;
          for i:=1 to m do
               MM[i,j]:= Maxim{peak[i],MM[i-1,j]};
     end;
```

Example 7. Let $A = abaabacab, \mu_A = (0.9, 0.9, 0.6, 0.5, 0.2, 0.8, 0.4, 0.6, 0.5)$, $\mathcal{B} = \{a, b, c\}\{bd\}\{bc\}$, $\mu_{B^1}(a) = 0.6, \mu_{B^1}(b) = 0.6, \mu_{B^1}(c) = 0.3, \mu_{B^2}(b) = 0.9, \mu_{B^2}(d) = 0.4, \mu_{B^3}(b) = 0.6, \mu_{B^3}(c) = 0.5$ then $MCCS(A, \mathcal{B}) = 2.4$ as it is computed in the following matrix.

```
          B      B1     B2     B3
                a 0.6
                b 0.6  b 0.9  b 0.6
      A         c 0.3  d 0.4  c 0.5
                ---------------------
    a 0.9 |      0.6    0.6    0.6
    b 0.9 |      1.2    1.5    1.5
    a 0.6 |      1.2    1.5    1.5
    a 0.5 |      1.2    1.5    1.5
    b 0.2 |      1.2    1.5    1.5
    a 0.8 |      1.2    1.5    1.5
    c 0.4 |      1.5    1.5    1.9
    a 0.6 |      1.5    1.5    1.9
    b 0.5 |      1.5    2.0    2.4
```

The subsequence can be recovered after the algorithm is finished if an array of a backpointers to the best matching elements is maintained. *Correctness* of the algorithm follows from the following invariants:

(1) After the j-th iteration of main algorithm all values $MM[i, j], 0 \leq i \leq m$ are computed. After the n-th iteration we have all values $MM[i, n], 0 \leq i \leq m$ and $MM[m, n] = MCCS((A, \mu_A), (\mathcal{B}, \mu_{calB}))$.

(2) *Findpeak(j)* computes the best matching of the j-th set B^j, *peak*$[j] \leq MM[i, j]$ and there exist some $j_0 \leq j$ such that *peak*$[j_0] \geq MM[i, j]$.

Time complexity. The main algorithm has the cycle for i and the call of procedure *Findpeak* inside of the cycle for j. It means $O(m \cdot n \cdot N)$-time complexity, where $N = \Sigma_{j=1}^{n}, |B^j|$.

Space complexity. The presented algorithm requires $O(m \cdot n)$-space for the array MM and $O(m)$-space for the unique stack.

6 Algorithm for MSSCCS Problem

The basic idea of the algorithm is very similar to the previous algorithm for MSCCS. It starts from the definition of $MSSCCS$ Problem.

$$MSCCS((\mathcal{A}, \mu_{\mathcal{A}}), (\mathcal{B}, \mu_{\mathcal{B}})) = max_{(C, \mu_C)}\{Val(C, \mu_C) : (C, \mu_C)\, is\, the\, common$$

$$cf - subsequence\, of\, (\mathcal{A}, \mu_{\mathcal{A}})\, and\, (\mathcal{B}, \mu_{\mathcal{B}})\} \tag{12}$$

If we have some flattenings of both set-strings then it is possible to apply the $MCCS$ algorithm. It is necessary to compute $MCCS$ values of all pairs of all flattenings both set-strings but that is too time consuming.

If we have the flattening of one set-string and the second is as set-string then it is possible to use the $MSCCS$ algorithms. But it is necessary to compute $MSCCS$ value for all flattenings of one string. This is also too time consuming. Both algorithms have exponential time complexity.

It is possible to use the following algorithm of polynomial time complexity. The algorithm works in two steps:

1. to create the string of symbols for each of set-string; each set can be encoded as the string of all permutations of its elements (the length of such string is $k^2 - 2 \cdot k + 4$, k is the number of elements in set [13]);

2. to apply the MCCRS algorithm for the two constructed strings (each element of the set can be used once at most);

The algorithm works in polynomial time: $O(M^2 \cdot N^2 \cdot K)$, where $M = \Sigma_{i=1}^{m} |A^i|, N = \Sigma_{j=1}^{n} |B^j|$, and K is the number of elements in closest common restricted subsequence.

7 Concluding Remarks

Polynomial algorithms for the solutions of the MCCS Problem, MCCRS Problem and MSCCS Problem with a competence functions have been presented. The MSSCCS Problem was formulated and the polynomial time algorithm for its solution was developed. However, we are convinced of the existence of an algorithm with better time complexity.

References

[1] Andrejková, G.: *The longest restricted common subsequence problem.* Proceedings of the Prague Stringology Club Workshop'98, Prague, 1998, p. 14-25.

[2] Dewar, R. B., Merritt, S. M., Sharir, M.: *Some modified algorithms for Dijkstra's longest common subsequence problem.* Acta Informatica 18, 1982, p. 1–15.

[3] Heckel, P.: *A technique for isolating differences between files.* Comm. ACM 21, 4 (Apr. 1978), p. 264–268.

[4] Hirschberg, D. S.: *A linear space algorithms for computing maximal common subsequences.* Comm. ACM 18, 6 (June 1975), p. 341–343.

[5] Hirschberg, D. S.: *Algorithms for longest common subsequence problem.* Journal ACM 24, 4 (Oct 1977), p. 664–675.

[6] Hirschberg, D. S.: *The least weight subsequence problem.* Symp. on FCT, October, 1985, p. 137–143.

[7] Hirschberg, D. S., Larmore, L. L.: *The Set LCS Problem.* Algorithmica 2 (1987), p. 91–95.

[8] Hirschberg, D. S., Larmore, L. L.: *Set-Set LCS Problem.* Algorithmica 4 (1989), p. 503–510.

[9] Holub, J.: *Dynamic Programming for Reduced NFAs for Approximate String and Sequence Matching.* Proceedings of the Prague Stringology Club Workshop'98, Prague, 1998, p. 73-82.

[10] Huang, S. S., Asuri, S. H.: *Algorithms for the Set-LCS and Set-Set-LCS Problems.* Tech. Report No. UH-CS-89-09, University of Houston, March, 1989.

[11] Hunt, J. W., Szymanski, T. G.: *A fast algorithm for computing longest common subsequences.* Comm. ACM 20, 5 (May 1977), p. 350–351.

[12] Lowrance, R., Wagner, R. A.: *An extension of the string-to-string correction problems.* Journal ACM 22, 2 (Apr. 1975), p. 177–183.

[13] Mohanty, S. P.: *Shortest string containing all permutations.* Discrete Mathematics 31, 1980, p. 91–95.

[14] Nakatsu, N., Kombayashi, Y., Yajima, S.: *A longest common subsequence algorithm suitable for similar text strings.* Acta Informatica 18, 1982, p. 171–179.

[15] Needleman, S. B., Wunsch, Ch. D.: *A general method applicable to the search for similarities in the amino acid sequence of two proteins.* Journal Mol. Biol. 48, 1970, p. 443–453.

[16] Troníček, Z., Melichar, B.: *Directed Acyclic Subsequence Graph.* Proceedings of the Prague Stringology Club Workshop'98, Prague, 1998, p. 107-118.

A Fast String Matching Algorithm and Experimental Results

T. Berry and S. Ravindran

Department of Computer Science
Liverpool John Moores University
Liverpool L3 3AF
United Kingdom

e-mail: {T.BERRY,S.RAVINDRAN}@livjm.ac.uk

Abstract. In this paper we present experimental results for string matching algorithms which have a competitive theoretical worst case run time complexity. Of these algorithms a few are already famous for their speed in practice, such as the Boyer-Moore and its derivatives. We chose to evaluate the algorithms by counting the number of comparisons made and by timing how long they took to complete a given search. Using the experimental results we were able to introduce a new string matching algorithm and compared it with the existing algorithms by experimentation. These experimental results clearly show that the new algorithm is more efficient than the existing algorithms for our chosen data sets. Using the chosen data sets over 1,500,000 separate tests were conducted to determine the most efficient algorithm.

Key words: string matching, pattern matching, algorithms on words.

1 Introduction

Many promising data structures and algorithms discovered by the theoretical community are never implemented or tested at all. Moreover, theoretical analysis (asymptotic worst-case running time) will show only how algorithms are likely to perform in practice, but they are not sufficiently accurate to predict actual performance. In this paper we show that by considerable experimentation and fine-tuning of the algorithms we can get the most out of a theoretical idea.

The string matching problem [CR94] has attracted a lot of interest throughout the history of computer science, and is crucial to the computing industry. String matching is finding an occurrence of a pattern string in a larger string of text. This problem arises in many computer packages in the form of spell checkers, search engines on the internet, find utilities on various machines, matching of DNA strands and so on.

Section 2 describes string matching algorithms which are known to be fast. Section 3 gives experimental results for these algorithms. From the findings of the experimental results discussed in Section 3, we identify two fast algorithms to produce a new algorithm. The new algorithm is described in Section 4. In Section 5 we compare the new algorithm with the existing algorithms.

2 The String Matching Algorithms

String matching algorithms work as follows. First the pattern of length m, $P[1..m]$, is aligned with the extreme left of the text of length n, $T[1..n]$. Then the pattern characters are compared with the text characters. The algorithms can vary in the order in which the comparisons are made. After a mismatch is found the pattern is shifted to the right and the distance the pattern can be shifted is determined by the algorithm that is being used. It is this shifting procedure and the speed at which a mismatch is found which is the main difference between the string matching algorithms.

In the Naive or Brute Force (BF) algorithm, the pattern is aligned with the extreme left of the text characters and corresponding pairs of characters are compared from left to right. This process continues until either the pattern is exhausted or a mismatch is found. Then the pattern is shifted one place to the right and the pattern characters are again compared with the corresponding text characters from left to right until either the text is exhausted or a full match is obtained. This algorithm can be very slow. Consider the worst case when both pattern and text are all a's followed by a b. The total number of comparisons in the worst case is $O(nm)$. However, this worst case example is not one that occurs often in natural language text.

An improved version of the BF algorithm, the Not So Naive (NSN) algorithm [HA93], changes the order of the comparisons. Suppose the pattern is aligned with the text characters, first the second pattern character is compared with the corresponding text character followed by comparisons of the rest of the pattern with corresponding text characters, and then the last characters to be compared are the first character of the pattern and the text character it is aligned with. A shift of two is made if a mismatch is made with the second character of the pattern and the first two characters of the pattern are the same, or if the second character of the pattern matches the text but a mismatch occurs and the first two characters are not equal.

The number of comparisons can be reduced by moving the pattern to the right by more than one position when a mismatch is found. This is the idea behind the Knuth-Morris-Pratt (KMP) algorithm [KMP77]. The KMP algorithm starts and compares the characters from left to right the same as the BF algorithm. When a mismatch occurs the KMP algorithm moves the pattern to the right by maintaining the longest overlap of a prefix of the pattern with a suffix of the part of the text that has matched the pattern so far. After a shift, the pattern character compared against the mismatched text character has to be different from the character that mismatched. The KMP algorithm takes at most $2n$ character comparisons. The KMP algorithm does $O(m+n)$ operations in the worst case.

The Colussi (COL) [CO91] algorithm is an improvement of the KMP algorithm. The number of character comparisons is $1.5n$ in the worst case. The set of pattern positions is divided into two disjoint subsets due to the combinatorial properties of their positions. First the comparisons are performed from left to right for the characters at positions in the first set. If there is no mismatch, the characters at positions in the second set are compared from right to left. This strategy reduces the number of comparisons.

Galil and Giancarlo (GG) [GG92] improved the COL algorithm by reducing the number of character comparisons in the worst case to $\frac{4}{3}n$. In these algorithms the preprocessing takes $O(m)$ time.

The Boyer-Moore (BM) algorithm [BM77] differs in one main feature from the algorithms already discussed. Instead of the characters being compared from left to right, in the BM algorithm the characters are compared from right to left starting with the rightmost character of the pattern. In a case of mismatch it uses two functions, last occurrence function and good suffix function and shifts the pattern by the maximum number of positions computed by these functions. The good suffix function returns the number of positions for moving the pattern to the right by the least amount, so as to align the already matched characters with any other substring in the pattern that are identical. The number of positions returned by the last occurence function determines the rightmost occurrence of the mismatched text character in the pattern. If the text character does not appear in the pattern then the last occurence function returns m. The worst case running time of the BM algorithm is $O(mn)$.

The Turbo Boyer-Moore (TBM) algorithm [CC94] and the Apostolico-Giancarlo (AG) algorithm [AG86] are ameliorations of the BM algorithm. When a partial match is made between the pattern and the text these algorithms remember the characters that matched and do not compare them again with the text. The TBM algorithm and the Apostolico-Giancarlo algorithm perform in the worst case at most $2n$ and $1.5n$ character comparisons respectively [CL97b].

The Horspool (HOR) algorithm [HO80] is a simplification of the BM algorithm. It does not use the good suffix function, but uses a modified version of the last occurrence function. The modified last occurrence function determines the right most occurrence of the $(k+m)$th text character, $T[k+m]$ in the pattern, if a mismatch occurs when a pattern is aligned with $T[k..k+m]$. This algorithm changes the order in which characters of the pattern are compared with the text. It compares the rightmost character in the pattern first then compares the leftmost character, then all the other characters in ascending order from the second position to the $m-1$th position.

The Raita (RAI) algorithm [RA92] again changes the order in which characters of the pattern are compared with the text. The process used to compare the rightmost character of the pattern, then the leftmost character, then the middle character and then the rest of the characters from the second to the $(m-1)$th position. If at any time during the procedure a mismatch occurs then it performs the shift as in the HOR algorithm.

The Quicksearch (QS) algorithm [SU90] is similar to the HOR algorithm and the RAI algorithm. It does not use the good suffix function to compute the shifts. It uses a modified version of the last occurrence function. Assume that a pattern is aligned with the text characters $T[k..k+m]$. After a mismatch the length of the shift is at least one. So, the character at the next position in the text after the alignment ($T[k+m+1]$) is necessarily involved in the next attempt. The last occurrence function determines the right most occurrence of $T[k+m+1]$ in the pattern. If $T[k+m+1]$ is not in the pattern the pattern can be shifted by $m+1$ positions. The comparisons between text and pattern characters during each attempt can be done in any order.

The Maximal Shift (MS) algorithm [SU90] is another variant of the QS algorithm. The algorithm is designed in such a way that the pattern characters are compared in the order which will give the maximum shift if a mismatch occurs.

The Smith (SMI) algorithm [SM91] uses HOR and Quick Search last occurrence functions. When a mismatch occurs, it takes the maximum values between these functions.

The Zhu and Takaoka (ZT) algorithm [ZT87] is another variant of the BM algorithm. The comparisons are done in the same way as BM (i.e. from right to left) and it uses the good suffix function. If a mismatch occurs at $T[i]$, the last occurrence function determines the right most occurrence of $T[i-1..i]$ in the pattern. If the substring is in the pattern, the pattern and text are aligned at these two characters for the next attempt. The shift is m, if the two character substring is not in the pattern.

Searching can be done in $O(n)$ time using a minimal Deterministic Finite Automaton (DFA) [SI93]. This algorithm uses $O(\sigma m)$ space and $O(\sigma + m)$ pre-processing time, where σ is the size of the alphabet. The Simon (SIM) algorithm [SI93] reduces the pre-processing time and the space to $O(m)$.

The pre-processing is needed for the algorithm to calculate the relevant shifts upon a mismatch/match except for the BF algorithm which has no pre-processing. The pre-processing cost of the algorithms does not effect the efficiency of the algorithms as they are relatively very small and all are approximately the same.

3 Experimental Results of the Existing Algorithms

Monitoring the number of comparisons performed by each algorithm was chosen as a way to compare the algorithms. All the algorithms were coded in C and their C code are taken from [CL97a] and animations of the algorithms can be found at [CL98]. This collection of string matching algorithms were easy to implement as functions into our main control program. The algorithms were coded as their authors had devised them in their papers. The main control program read in the text and pattern and had one of the algorithms to be tested inserted into it for the searching process. The main control program was the same for each algorithm and so did not affect the performance of the algorithms. Each algorithm had an integer counter inserted into it, to count the number of comparisons made between the pattern and the text. The counter was incremented by one each time a comparison was made.

A random text of 200,000 words from the UNIX English dictionary was used for the first set of experiments. The random text was constructed so as to simulate an actual English text. All the letters in the UNIX dictionary were made lower case to increase the probability of a match. In English text roughly only every 1 in 10 words begin with a capital letter. We decided to number each of the words in UNIX dictionary from 1 to 25,000. Then we used a pseudo random number generator to pick words from the UNIX dictionary and place them in the random text. Separating each word by a space character. Punctuation was also removed as we were concerned with finding words and the punctuation would not effect the results obtained. The reason for using a large text (200,000 words) was to ensure that as many of the 25,000 words in the UNIX English dictionary occurred somewhere in the random text generated. For each pattern in the dictionary, we searched the text (of 200,000 words) for the first occurrence of the pattern.

The text was searched for each word in the UNIX dictionary and the results are given in Table 1. The first column in Table 1 is the length of the pattern. The second column is the number of words of that length in the UNIX English dictionary. For example, for a pattern length of 7, 4042 test cases were carried out and the average number of character comparisons made by the KMP algorithm was 197,000 (to the nearest 1000). The average was calculated by taking the total number of comparisons performed to find all 4042 cases and dividing this number by 4042. These columns are arranged in descending order of the average of the total number of comparisons of the algorithms. An interesting observation is that for (almost) each row the values are in descending order except for the last two columns.

p. len	num.	BF	KMP	DFA	SIM	NSN	COL	GG	BM	AG	HOR	RAI	TBM	MS	QS	ZT	SMI
2	133	7	7	7	7	6	6	6	3	3	3	3	3	2	2	3	2
3	765	38	38	37	37	37	37	37	13	13	13	13	13	11	10	13	10
4	2178	82	82	80	80	80	79	79	23	23	23	23	22	19	19	22	18
5	3146	151	150	145	145	145	145	145	34	34	34	34	34	30	30	32	28
6	3852	186	185	179	179	179	178	178	36	36	36	36	36	33	32	33	30
7	4042	198	197	191	191	191	190	190	34	34	34	34	34	32	31	30	28
8	3607	205	204	197	197	197	196	196	32	32	31	32	31	30	29	27	26
9	3088	212	211	204	204	204	203	203	30	30	30	30	30	29	28	25	24
10	1971	220	219	212	212	212	210	210	29	29	29	29	29	28	27	24	23
11	1120	209	207	201	201	200	198	198	26	26	26	26	25	25	24	21	21
12	593	218	217	210	210	209	207	207	25	25	25	25	25	24	24	21	20
13	279	224	222	215	215	213	212	212	24	24	24	24	24	23	23	19	19
14	116	228	227	220	220	219	217	217	23	23	23	23	23	23	23	19	19
15	44	151	150	144	144	143	142	142	15	15	15	15	14	14	14	11	12
16	17	227	225	217	217	215	214	214	20	21	21	21	20	20	20	18	16
17	7	233	231	222	222	221	218	218	20	20	20	20	19	19	20	15	16
18	4	236	234	225	225	223	221	221	19	20	20	20	19	19	20	14	16
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	1	132	131	122	122	121	119	119	10	10	10	10	10	10	10	7	8
21	2	311	309	295	295	290	288	288	24	24	25	25	23	23	24	15	18
22	1	491	486	455	455	451	445	445	33	33	33	33	33	31	34	22	27
total	24966	180	179	174	174	173	172	172	31	31	30	30	30	28	28	27	25

Table 1: Results of searching a text of 200,000 words for each word in the UNIX dictionary.

The algorithm with the largest number of comparisons is the BF algorithm. This is because the algorithm shifts the pattern by one place to the right when a mismatch occurs, no matter how much of a partial/full match has been made. This algorithm has a quadratic worst case time complexity. But the KMP algorithm which has a linear worst case time complexity, does roughly the same number of comparisons as the BF algorithm. The reason for this is that in a natural language a multiple occurrence of a substring in a word is not common. For the same reason, the KMP variants, COL and GG algorithms have only a small improvement over the KMP algorithm. Other linear time algorithms, DFA and SIM, also have roughly the same number of comparisons as the BF algorithm. We will see below that the other quadratic worst case time complexity algorithms perform much better than these linear worst case time algorithms. This is a good example showing that asymptotic worst-case running time analysis can be indicative of how algorithms are likely to perform in practice, but they are not sufficiently accurate to predict actual performance.

The BM algorithm uses the good suffix function to calculate the shift which depends on a reoccurrence of a substring in a word. But, it also uses the last occurrence function. It is this last occurrence function that reduces the number of comparisons significantly. In practice, on an English text, the BM algorithm is three or more times faster than the KMP algorithm [SG82]. ¿From Table 1 one can see that the KMP algorithm is takes six times more comparisons than the BM algorithm on average. The other algorithms, TBM, AG, HOR, RAI, QS, MS, SMI and ZT, are variants of the BM algorithm. The number of comparisons for these algorithms is roughly the same number as in the BM algorithm.

The SMI algorithm and the ZT algorithm do the least number of comparisons for pattern lengths less than or equal to twelve and greater than twelve respectively.

4 The New Algorithm - the BR algorithm

¿From the findings of the experimental results discussed in section 3, it is clear that the SMI and ZT algorithms have the lowest number of comparisons among the others. We combined the calculations of a valid shift in SMI and ZT algorithms to produce a more efficient algorithm. If a mismatch occurs when the pattern $P[1..m]$ is aligned with the text $T[k+1..k+m]$, the shift is calculated by the rightmost occurrence of the substring $T[k+m+1..k+m+2]$ in the pattern. If the substring is in the pattern then the pattern and text are aligned at this substring for the next attempt. This can be done shifting the pattern as shown in the table below. Let $*$ be a wildcard character that is any character in the ASCII set. Note that if $T[k+m+1..k+m+2]$ is not in the pattern, the pattern is shifted by $m+2$ positions. The total number of comparisons in the worst case is $O(nm)$.

$T[k+m+1]$	$T[k+m+2]$	*Shift*
$*$	$P[1]$	$m+1$
$P[i]$	$P[i+1]$	$m-i+1,\ 1 \leq i \leq m-1$
$P[m]$	$*$	1
Otherwise		$m+2$

For example, the following shifts would be associated with the pattern, onion.

$T[k+m+1]$	$T[k+m+2]$	*Shift*
$*$	o	6
o	n	5
n	i	4
i	o	3
o	n	2
n	$*$	1
Otherwise		7

After a mismatch the calculation of a shift in our BR algorithm takes $O(1)$ time. Note that for the substrings ni and n* have a value of 4 and 1 respectively. This ambiguity can be solved by the higher shift value being overwritten with the lower value. We will explain this later in this section. For a given pattern $P[1..m]$ the preprocessing is done as follows, and it takes $O(\sigma^2)$ time.

There are 128 characters in the ASCII set and $(128)2 = 16384$ distinct pairs. We define an array Shift Array (SA) of length 16384 and initialise it to $m+2$. Using a hash function we insert the values for each pair and the hash function we use is:

$T[m+k+1] \times 127 + T[m+k+2]$ where for $P[m+k+1]$ and $P[m+k+2]$ we use their ASCII values. This gives each pair of character a distinct value in SA and we insert into the SA the shift for the pair. If the same pair occurs more than once then the lower shift value overwrites the higher value. So for example for the pair $[i][o]$ we would insert the value 3 at the $[105 \times 127] + 111 = 13446th$ position in SA.

$[wildcard][o] = 6$ all array positions that satisfy $x[0] mod 127 = 111 mod 127 = 6$
$[o][n] = 5$ position $111 \times 127 + 110 = 14207$
$[n][i] = 4$ position $110 \times 127 + 105 = 14075$
$[i][o] = 3$ position $105 \times 127 + 111 = 13446$
$[o][n] = 2$ position $111 \times 127 + 110 = 14207$
$[n][wildcard] = 1$ position $110 \times 127 + 0..127 = 13970..14097$

The order of performing the steps is important in ensuring the correct values appear in the array. Note that the higher values have been over written by the lower values.

In the RAI algorithm the first and last characters of the pattern are made variables. This cuts down the number of array look ups performed during a search. We adapted this idea to our algorithm and compared the least frequent pattern character with its corresponding text character. We then repeated the process for the second least frequent character and then the rest of the characters in order from right to left.

The UNIX dictionary used in the tests was used to see how many times each letter occurred in the dictionary. The frequency of each letter is given in the following chart.

letter	frequency	ranking	letter	frequency	ranking	letter	frequency	ranking
a	16395	25	j	432	3	s	10167	19
b	4110	10	k	1923	6	t	12789	22
c	8209	17	l	10013	18	u	6476	16
d	5763	14	m	5822	15	v	1890	5
e	20083	26	n	12062	20	w	1950	7
f	2660	8	o	12696	21	x	616	4
g	4125	11	p	5514	13	y	3618	9
h	5179	12	q	377	1	z	429	2
i	13963	24	r	13409	23			

Note that we choose the characters in the pattern that have the lowest ranking. If the character is not in the pattern then it has a ranking of 0 and is chosen as the least frequent character.

We now give an example of our BR algorithm in action to find the pattern onion. The SA array for the pattern onion were used to calculate the shift after a mismatch. $P[2]$ is the least frequent and $P[5]$ is the next least frequent character.

w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
	$\neq$																								
o	n	i	o	n																					

mismatch shift on $SA([n][t]) = 110 \times 127 + 116 = SA[14086] = 1$

w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
		$\neq$																							
	o	n	i	o	n																				

mismatch shift on $SA([t][]) = 116 \times 127 + 32 = SA[14764] = 7$.

w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
									$\neq$																
								o	n	i	o	n													

mismatch shift on $SA([s][t]) = 115 \times 127 + 116 = SA[14721] = 7$

w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
																$\neq$									
															o	n	i	o	n						

mismatch shift on $SA([][o]) = 32 \times 127 + 111 = SA[4175] = 6$.

w	e		w	a	n	t		t	o		t	e	s	t		w	i	t	h		o	n	i	o	n
																					=	=	=	=	=
																					5	1	4	3	2
																					o	n	i	o	n

So the word onion is found in 9 comparisons in a text of length 26. On the above full match the order in which the comparisons are conducted is shown on the third row.

5 Experimental Results and Comparisons with the BR Algorithm

We select the best nine algorithms from the results in Table 1 and the KMP algorithm, and compare with our BR algorithm. Experiments were carried out for different random texts as described in Section 3. The texts were constructed by randomly choosing words from the UNIX English dictionary. There were 2 different texts of 10,000 words, a text of 50,000 words and a text of 100,000 words. The results are described in Tables 3-6 (see appendix) respectively. Tables 3-6 (which can be found in the appendix at the back of this paper) show the average number of comparisons required for a search for the given pattern length. They are based on taking the total number of comparisons for the search for all the patterns of a length and dividing the number by the number of patterns of that size to give the average. So for example, in Table 3 the BM algorithm takes 12,000 comparisons (to the nearest thousand) on average if the pattern length is 7. From these tables one can observe that the relative order of their performance is the same as in Table 1. The main observation is that the BR algorithm performs better than the other algorithms for all pattern lengths and for all texts used in the experiments.

p. len.	num.	KMP	AG	BM	HOR	RAI	TBM	MS	QS	ZT	SMI
2	133	199.98	93.96	93.96	94.00	93.96	93.89	35.94	32.92	93.96	31.48
3	765	366.02	64.09	64.18	64.20	64.19	63.70	28.78	28.21	60.03	24.93
4	2178	449.02	50.97	51.11	50.86	50.90	50.77	28.25	25.77	43.19	19.73
5	3146	540.11	44.91	45.02	44.58	44.46	44.72	28.33	26.47	33.91	18.13
6	3852	626.30	42.58	42.42	41.83	41.68	41.91	30.02	27.32	27.71	16.42
7	4042	719.01	42.07	41.38	40.92	41.00	40.72	31.49	28.83	24.94	16.08
8	3607	807.61	40.76	40.58	40.28	40.35	39.95	32.27	30.10	21.67	15.49
9	3088	896.18	41.85	41.52	40.92	40.84	40.69	34.75	32.19	19.29	15.45
10	1971	982.63	42.38	42.19	41.69	41.79	41.16	36.62	34.37	17.75	15.64
11	1120	1067.87	44.91	44.14	43.67	43.79	42.97	38.57	37.18	17.06	16.32
12	593	1164.14	45.36	45.28	44.58	44.68	44.20	40.06	39.28	16.14	17.34
13	279	1245.53	48.85	47.88	47.22	47.32	46.36	42.26	41.61	12.65	17.54
14	116	1322.70	46.46	46.74	46.46	46.60	45.16	42.62	42.26	11.32	17.03
15	44	1426.02	50.78	51.20	51.51	51.59	49.23	44.73	45.29	8.72	19.00
16	17	1527.28	48.99	49.34	50.44	50.60	47.37	46.60	49.06	24.80	20.02
17	7	1598.50	45.09	45.29	44.51	44.58	43.42	40.22	45.01	6.72	16.95
18	4	1700.81	50.34	50.58	53.96	54.06	48.54	50.12	53.59	6.09	22.21
19	0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
20	1	1948.74	58.37	58.37	58.12	58.07	58.37	52.25	63.51	3.01	29.43
21	2	1947.96	58.13	57.38	63.98	63.99	56.32	57.59	57.50	2.22	21.84
22	1	2129.14	50.97	50.97	49.87	49.89	50.97	45.07	55.43	1.04	25.09
total	24992	737.56	43.54	43.29	42.83	42.82	42.65	32.00	29.72	26.09	16.66

Table 2: The average difference between each of the existing algorithms and our BR algorithm as a percentage.

Table 2 summarises the results of Tables 3-6. The entries in Table 2 are in percentage form and describe how many fewer comparisons our BR algorithm uses, when compared with the existing algorithms. The figures are an average of the four different texts used. To calculate the difference as a percentage between our BR algorithm and the existing algorithms we used the following formula. The average number of comparisons was taken from the relevant cell in Tables 3-6 and divided by the value for that pattern length for our BR algorithm. This value was then deducted by 1 and multiplied by 100 to give the percentage difference between the two algorithms. An interesting observation of the existing algorithms when compared with the BR algorithm, is that for each individual text the percentages were within 1% for each specific algorithm. Each value in Table 2 is calculated by taking the difference as a percentage between each algorithm and our BR algorithm for each pattern length, adding them together and dividing by 4. For example, for a pattern length of 4 the BM algorithm takes on average 51.11% more comparisons than our BR algorithm.

The result of a full search for the dictionary over all four texts is given in the last row of Table 2. From this we can see that the BM algorithm took on average 43.54% more comparisons than our BR algorithm (see 5th column, last row) for a complete search for all the words in the dictionary.

Further to counting the number of comparisons we time the algorithms. The saving in the number of comparisons may be paid for by extra overhead due to accessing the precomputed shift table. We timed the search of the medium text of 50,000 words for all occurrences of the words in the UNIX dictionary. We used a 486-DX66 with 8 megabytes of RAM and a 100 megabyte hard drive running SUSE 5.2. In Table 7, the total number of comparisons for the search are given along with the time taken by each algorithm for the search, including any preprocessing performed by the algorithm. The number of comparisons are reduced by a factor of 1000. i.e. for BF 10911786 means 10911786000 comparisons.

	medium1			book1			book2		papers	
	number	time	% dif BR	num. comp.	time sec.	% dif. BR	time	% dif. BR	time	% dif. BR
BF	10911786	1315m 13s	528.54							
KMP	10433340	1341m 25s	541.06							
DFA	10433340	892m 59s	326.75							
SIM	10433340	1688m 18	706.83							
NSN	10482487	777m 52s	271.74							
BM	2002822	371m 51s	77.71	3602739	674m	79.73	663s	69.57	264s	58.08
AG	2005310	972m 10s	364.60							
HOR	1985219	244m 41s	16.93	3580863	442m	17.87	446s	14.07	249s	49.10
RAI	1998657	238m 27s	13.95	3601251	431m	14.93	434s	11.00	173s	3.59
MS	1815486	318m 49s	52.36							
QS	1785730	245m 58s	17.55	3189368	444m	18.40	452s	15.60	180s	7.78
ZT	1761716	420m 55s	101.15							
TBM	1683516	1166m 4s	457.26							
SMI	1621591	280m 41s	34.14	2930285	513m	36.80	514s	31.46	207s	23.95
BR	1489839	209m 15s	n/a	2682916	375m	n/a	391s	n/a	167s	n/a

Table 7: Timing for a complete search for the dictionary in the given texts.

¿From this table we can see that the algorithms that take a high number of comparisons are quite slow as well. The lower the number of comparisons the better the time. Although putting the algorithms in order of how many comparisons they take from highest to lowest starting at the BM we get the list: BM, RAI, AG, HOR, MS, QS, ZT, TBM, SMI and the BR. If we do the same for the timings we get ZT, BM, MS, SMI, QS, RAI and the BR. The reason for the difference in the lists is due to overheads in traversing the data structures which are present in the algorithms for the calculation of the correct shift value. Also the pre-processing of some of the algorithms are expensive. So we can not assume that because an algorithm takes a fewer number of comparisons that it will be more efficient than another.

We can also save time by performing the comparisons as in the RAI algorithm. This is done by making the least and second least likely characters variables instead having to look them up in the pattern array. Although counting the comparisons is a good estimate of which algorithm is the best to use we have to actual time the algorithms to find the best algorithm for the task of string matching.

We repeated the tests for the medium text for the book1 text for the 5 algorithms with the best times and our BR algorithm. From Table 7 we can see that our BR algorithm is still the quickest and the other algorithms are still over 14% more time than our algorithm. So our findings for a random text hold for this real English text. We then considered two other texts, book2 and the six papers concatenated together from the Calgary corpus [CAL]. We searched for 500 random words from the UNIX dictionary again for the best 5 algorithms and our BR algorithm. The results documented in Table 7 show that algorithm is the fastest algorithm for these tests. The main reason for the speed of our BR algorithm is the improved maximum shift of $m + 2$ and the searching on the least likely to occur characters.

Conclusions

The experimental results show that the BR algorithm is more efficient than the existing algorithms in practice for our chosen data sets. Over our 4 random texts and 3 real texts where the BR algorithm is compared to the existing algorithms, our algorithm is comfortably more efficient over each text. With the addition of punctuation and capital letters it does not affect the BR algorithm. If the pattern to be searched for began with a capital letter then this would make the capital letter the least frequent character and so it would be searched for first. We would expect the probability of a mismatch to rise and so the algorithm would speed up considerably. So in the real world we would expect our savings to remain and make our BR algorithm competitive with the existing algorithms. It is also possible to apply some of our finding to what makes a fast algorithm to the existing algorithms. This may make them faster but we were concerned with the original algorithms that were devised by their authors.

Acknowledgments

We wish to thank Carl Bamford for comments and suggestions made to us during the writing of this paper.

References

[AG86] Apostolico A., Giancarlo R., *"The Boyer-Moore-Galil string strategies revisited"*, SIAM Journal of Computing, 15(1), pages 98-105, 1986.

[BM77] Boyer R. S., Moore J. S., *"A fast string searching algorithm"*, Communications of the ACM, 23(5), pages 1075-1091, 1977.

[CAL] Calgary Corpus available at:
ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/

[CL97a] Charras C., Lecroq T., Exact string matching available at:
HTTP://www.dir.univ-rouen.fr/ lecroq/string.ps, 1997.

[CL98] Charras C., Lecroq T., Exact string matching animation in JAVA available at: HTTP://www. dir.univ-rouen.fr/ charras/string/, 1998.

[CO91] Colussi L., *"Correctness and efficiency of the pattern matching algorithms"*, Information Computing, 95(2), pages 225-251, 1991.

[CC94] Crochemore M., Czumaj A., Gąsieniec L., Jarominek T., Lecroq T., Plandowski W., Rytter W., *"Speeding up two string matching algorithms"*, Algorithmica, 12(4), pages 247-267, 1994.

[CL97b] Crochemore M., Lecroq T., *"Tight bounds on the complexity of the Apostolico-Giancarlo algorithm"*, Information Processing Letters, 63(4), pages 195-203, 1997.

[CR94] Crochemore M., Rytter W., *"Text algorithms"*, Oxford University Press, 1994.

[GG92] Galil Z., Giancarlo R., *"On the exact complexity of string matching: upper bounds"*, SIAM Journal of Computing, 21(3), pages 407-437, 1992.

[HA93] Hancart C., *"Analyse exacte et en moyenne d'algorithmes de recherche d'un motif dans un texte"*. Thése de doctorat de l'Université de Paris 7, France, 1993.

[HO80] Horspool R. N., *"Practical fast searching in strings"*. Software Practice and Experience. 10(6), pages 501-506, 1980.

[KMP77] Knuth D. E., Morris Jr J. H., Pratt V. R., *"Fast pattern matching in strings"*, SIAM Journal of Computing, 6(1), pages 323-350, 1977.

[RA92] Raita T., *"Tuning the Boyer-Moore-Horspool string searching algorithm"*, Software Practice and Experience, 22(10), pages 879-884, 1992.

[SI93] Simon I., *"String matching algorithms and automata"*, First American Workshop on String Processing, ed. Baeza-Yates and Ziviani, pages 151-157. Universidade Federal de Minas Gerais, 1993.

[SM91] Smith P. D., *"Experiments with a very fast substring search algorithm"*, Software Practice and Experience, 21(10), pages 1065-1074, 1991.

[SG82] de Smit G. V., *"A Comparison of Three String Matching Algorithms"*, Software Practice and Experience, 12(1), pages 57-66, 1982.

[SU90] Sunday D. M., *"A very fast substring search algorithm"*, Communications of the ACM, 33(8), pages 132-142, 1990.

[ZT87] Zhu R. F., Takaoka T., *"On improving the average case of the Boyer-Moore string matching algorithm"*, Journal of Information Processing, 10(3), pages 173-177, 1987.

Appendix

p len	num	KMP	AG	BM	HOR	RAI	TBM	MS	QS	ZT	SMI	BR
2	133	6	3	3	3	3	3	2	2	3	2	2
3	765	20	7	7	7	7	7	6	6	7	5	4
4	2178	41	11	11	11	11	11	10	10	11	9	7
5	3146	60	14	14	13	13	13	12	12	12	11	9
6	3852	67	13	13	13	13	13	12	12	12	11	9
7	4042	68	12	12	12	12	12	11	11	10	10	8
8	3607	69	11	11	11	11	11	10	10	9	9	7
9	3088	70	10	10	10	10	10	9	9	8	8	7
10	1971	71	9	9	9	9	9	9	9	8	8	6
11	1120	70	9	9	9	9	9	8	8	7	7	6
12	593	70	8	8	8	8	8	8	8	6	7	5
13	279	72	8	8	8	8	8	8	8	6	6	5
14	116	69	7	7	7	7	7	7	7	5	6	5
15	44	72	7	7	7	7	7	7	7	5	6	5
16	17	70	6	6	6	6	6	6	6	5	5	4
17	7	75	7	7	7	7	6	6	6	5	5	4
18	4	87	7	7	7	7	7	7	7	5	6	5
19	0	0	0	0	0	0	0	0	0	0	0	0
20	1	89	7	7	7	7	7	7	7	4	5	4
21	2	88	7	7	7	7	7	6	7	4	5	4
22	1	89	6	6	6	6	6	6	6	4	5	4
total	24966	64	11	11	11	11	11	10	10	10	9	7

Table 3: Averages for random TEXT A of 10,000 words

p len	num	KMP	AG	BM	HOR	RAI	TBM	MS	QS	ZT	SMI	BR
2	133	6	3	3	3	3	3	2	2	3	2	2
3	765	21	7	7	7	7	7	6	6	7	6	4
4	2178	42	12	12	12	12	12	10	10	11	9	8
5	3146	59	13	13	13	13	13	12	12	12	11	9
6	3852	66	13	13	13	13	13	12	12	11	11	9
7	4042	68	12	12	12	12	12	11	11	10	10	8
8	3607	69	11	11	11	11	11	10	10	9	9	8
9	3088	70	10	10	10	10	10	9	9	8	8	7
10	1971	71	9	9	9	9	9	9	9	8	8	7
11	1120	70	9	9	9	9	9	8	8	7	7	6
12	593	71	8	8	8	8	8	8	8	6	7	6
13	279	71	8	8	8	8	8	8	7	6	6	5
14	116	70	7	7	7	7	7	7	7	6	6	5
15	44	64	6	6	6	6	6	6	6	5	5	4
16	17	74	7	7	7	7	7	7	7	5	5	5
17	7	64	6	6	6	6	6	5	6	4	4	4
18	4	87	7	7	7	7	7	7	7	5	6	5
19	0	0	0	0	0	0	0	0	0	0	0	0
20	1	41	3	3	3	3	3	3	3	2	3	2
21	2	72	5	5	6	6	5	5	5	4	4	3
22	1	89	6	6	6	6	6	6	6	4	5	4
total	24966	63	11	11	11	11	11	10	10	10	9	8

Table 4: Averages for random TEXT B of 10,000 words

p len	num	KMP	AG	BM	HOR	RAI	TBM	MS	QS	ZT	SMI	BR
2	133	9	6	6	6	6	6	4	4	6	4	3
3	765	37	13	13	13	13	13	10	10	13	10	8
4	2178	77	21	21	21	21	21	18	18	20	17	13
5	3146	133	30	30	30	30	30	27	26	28	25	20
6	3852	159	31	31	31	31	31	29	28	28	26	21
7	4042	170	29	29	29	29	29	27	27	26	24	20
8	3607	176	27	27	27	27	27	26	25	24	22	19
9	3088	181	26	26	26	26	26	25	24	22	21	18
10	1971	185	24	24	24	24	24	23	23	20	20	17
11	1120	184	23	23	23	23	23	22	22	18	18	15
12	593	186	21	21	21	21	21	21	20	17	17	14
13	279	183	20	20	20	20	20	19	19	15	16	13
14	116	194	20	20	20	20	20	19	19	15	16	13
15	44	164	16	16	16	16	16	16	16	12	13	10
16	17	217	20	20	20	20	20	20	20	17	16	13
17	7	172	15	15	15	15	14	14	15	11	12	10
18	4	147	12	12	13	13	12	12	13	9	10	8
19	0	0	0	0	0	0	0	0	0	0	0	0
20	1	41	3	3	3	3	3	3	3	2	3	2
21	2	221	17	17	18	18	17	17	17	11	13	10
22	1	397	27	27	27	27	27	26	28	18	22	17
total	24966	155	27	27	26	26	26	24	24	23	22	18

Table 5: Averages for random text of 50,000 words

p len	num	KMP	AG	BM	HOR	RAI	TBM	MS	QS	ZT	SMI	BR
2	133	13	7	7	7	7	7	5	5	7	5	3
3	765	37	13	13	13	13	13	10	10	13	10	8
4	2178	80	22	22	22	22	22	19	18	21	17	15
5	3146	149	34	34	34	34	34	30	29	31	28	23
6	3852	182	36	36	36	36	36	33	32	33	29	25
7	4042	193	33	33	33	33	33	31	30	29	27	24
8	3607	201	31	31	31	31	31	29	29	27	26	22
9	3088	198	28	28	28	28	28	27	26	24	23	20
10	1971	198	26	26	26	26	26	25	25	22	21	18
11	1120	199	25	25	25	24	24	24	23	20	20	17
12	593	217	25	25	25	25	25	24	24	20	20	17
13	279	207	23	23	23	23	22	22	22	18	18	15
14	116	180	20	19	19	19	19	18	18	14	15	13
15	44	218	22	22	22	22	21	21	21	17	17	14
16	17	162	15	15	15	15	15	15	15	12	12	10
17	7	220	20	20	20	20	19	19	19	14	15	13
18	4	208	17	17	17	17	17	17	18	12	14	11
19	0	0	0	0	0	0	0	0	0	0	0	0
20	1	157	12	12	12	12	12	12	13	8	10	8
21	2	89	7	7	7	7	7	7	7	11	5	4
22	1	315	21	21	21	21	21	20	22	14	18	14
total	24966	173	30	30	30	30	29	27	27	26	24	21

Table 6: Averages for random text of 100,000 words

On Procedures for Multiple-string Match with Respect to Two Sets[1]

Weiler A. Finamore, Rafael D. de Azevedo
& Marcelo da Silva Pinho

Center for Telecommunications Studies (CETUC)
Catholic University of Rio de Janeiro
Marqus de S. Vicente, 225
22453-900, RIO DE JANEIRO, RJ
Brazil

e-mail: weiler@cetuc.puc-rio.br

Abstract. String match procedures with respect to two sets are investigated. The procedures traditionally used for data compression are based on single-string match with respect to a single set [LZ78, W84]. Some recent work broadened this view by presenting procedures for multiple-string match with respect to a single set [FPC98, PFP99] with improved performance as compared to the single-match versions. In this work an algorithm based on double-match with respect to two sets is stated. We do conjecture that multiple-string match procedures with respect to two sets can achieve even better performance. A preliminary analysis corroborating this conjecture with some evidence is reported in this work.

Key words: Multiple-string match, Lempel-Ziv algorithm, Data compression.

1 Introduction

The procedure proposed by Lempel and Ziv in 1978 [LZ78] for lossless data compression is a rather simple and elegant string-match based algorithm. Its low complexity and implementation simplicity has turned it into a very popular algorithm which is used for instance in the *compress* program of UNIX operational system.

By selecting diferent combinations of the basic parameters of this algorithm many variations can be established. In the result published in [FPC98] a version that searches for double-string matches instead of the usual single-match is stated — an improved performance was obtained. Extension to multiple string-match was proposed in [PFP99]. Similar results were reported by Hartman and Rodeh in [HR85].

In this work the two most popular Lempel-Ziv variations, LZ78 and LZW [LZ78, W84], has been cast in the framework of string-match with respect to two sets. We also propose two new variations (designated lg-LZ and dt-LZ), which are inspired and discussed in this new framework. Although the ultimate goal of finding new

[1]This work was supported by grant **CNPq**-502235/91-8(NV) and **AEB/PR**-004/97.

algorithms with improved is a motivation behind the algorithms proposed, the imediate objective is to expand the ways of looking at the string matches algorithms and hopefully to find better procedures.

This work is organized as follows: in Section 3, we present the idea of string match with respect to two sets and establish a motivation by discussing two well-known algorithms in the framework of matching with respect to two sets. A new algorithm (lg-LZ) which is a simple variation of the Lempel-Ziv algorithm is also proposed in this section. In Section 4 a version of double-match/double-tree algorithm is introduced. Results obtained by computer simulation are presented in Section 5. Our conclusion is then summarized in Section 5.

2 Notations

We establish the following notation for use in this work.

1. $x_i^j = x_i x_{i+1} \dots x_j$, $j > i$ denotes a finite sequence of symbols x_k, $i \leq k \leq j$, that take their values in a given set $\mathcal{A} = \{a_0, a_1, \cdot, a_{|\mathcal{A}|-1}\}$ of cardinality $|\mathcal{A}|$. If $j = i$, this is the single symbol string x_i and if $i > j$ we will assume that x_i^j is the empty string.

2. $|\alpha|$ denotes the length, if α is a sequence, or the cardinality, if α is a set.

3. λ denotes the null-length string, i.e. $|\lambda| = 0$.

4. $\mathbf{s}_i \circ \mathbf{s}_j$ denotes the concatenation of the strings $\mathbf{s}_i$ and $\mathbf{s}_j$. (the result of the concatenation will also be indicated by $\mathbf{s}_i\mathbf{s}_j$ or $\mathbf{s}_i$,$\mathbf{s}_j$)

5. When $\mathbf{s}_1, \mathbf{s}_2, \cdots, \mathbf{s}_k \in \mathcal{A}^*$ are strings of symbols of lengths $|\mathbf{s}_1|, |\mathbf{s}_2|, \cdots, |\mathbf{s}_k|$ respectively, the notation $\mathbf{s}_1^k$ represents the string of length $|\mathbf{s}_1| + |\mathbf{s}_2| + \cdots + |\mathbf{s}_k|$ formed by the concatenation of strings $\mathbf{s}_1 \circ \mathbf{s}_2 \circ \cdots \circ \mathbf{s}_k$.

6. The concatenation of the string $\ell \in \mathcal{L} = \{\ell_0, \cdots, \ell_{|\mathcal{L}|-1}\}$ and the set $\mathcal{M} = \{\mathbf{m}_0, \cdots, \mathbf{m}_{|\mathcal{M}|-1}\}$ is the set
$$\ell \circ \mathcal{M} = \bigcup_{i=0}^{|\mathcal{M}|-1} \{\ell \circ \mathbf{m}_i\}$$

7. Let $\mathcal{L} = \{\ell_0, \cdots, \ell_{|\mathcal{L}|-1}\}$ and $\mathcal{M} = \{\mathbf{m}_0, \cdots, \mathbf{m}_{|\mathcal{M}|-1}\}$. We define the concatenation of these two sets by
$$\mathcal{L} \circ \mathcal{M} = \bigcup_{i=0}^{|\mathcal{L}|-1} \{\ell_i \circ \mathcal{M}\}$$

8. $\lceil x \rceil$ denotes the smallest integer greater than or equal to number x.

9. $\Pi[\mathbf{z}|\mathcal{L}]$, for $|\mathcal{L}| > 0$, is the longest string $\ell_i \in \mathcal{L} = \{\ell_0, \cdots, \ell_{|\mathcal{L}|-1}\}$ which is a prefix of $\mathbf{z}$.

10. $\mathcal{X}[\mathbf{s}|\mathcal{L}]$ is the unique integer index i that identify the member $\ell_i \in \mathcal{L}$ such that $\ell_i = \mathbf{s}$.

11. $\mathbf{z} - \mathbf{y}$, when $\mathbf{z} = x_i x_{i+1} \cdots x_j$ and $\mathbf{y} = x_i x_{i+1} \cdots x_k$ is a prefix of $\mathbf{z}$, represents the string $x_{k+1} \cdots x_j$.

12. $\mathcal{F}[\mathbf{z}]$ is the length 1 prefix of $\mathbf{z}$, if $|\mathbf{z}| > 0$ else it is the empty string.

13. $\mathcal{S}[\mathbf{z}]$ is the length $|\mathbf{z}| - 1$ prefix of $\mathbf{z}$.

14. $\phi_k[J]$, $k \geq \log J$ (base 2 logarithm) is the trivial k-bit binary representation of the integer J.

3 The Idea of String Match Algorithm with Respect to Two Sets

To establish the framework and the rationale behind our discussion, the well-known string-match procedure proposed by Ziv and Lempel [LZ78] for data compression will be presented, in the context of string match with respect to two sets. We will undistinguishably refer to this as a double-tree string match context since the sets we will be dealing with are tree-structured.

3.1 Lempel-Ziv Algorithm (LZ78)

Let us consider that $\mathbf{z}_0 = x_0^{N-1}$ is the sequence of N symbols generated by the information source which is to be encoded (each source symbol x_i belongs to the source alphabet $\mathcal{A}$, of dyadic cardinallity for simplicity). Generally speaking the Lempel-Ziv algorithm (LZ78) [LZ78] can be envisioned as divided in three tasks: The first task, (parsing), which yields the unique parsing

$$x_0^{N-1} = (\ell_0 \circ \mathbf{m}_0), (\ell_1 \circ \mathbf{m}_1), \cdots, (\ell_t \circ \mathbf{m}_t)$$

of the source sequence in $t+1$ phrases. The next task, (map to integers), assign each phrase $\mathbf{s_i} = (\ell_i \circ \mathbf{m}_i)$ to a unique pair of integers (J_i, K_i) which are then, in the task that follows (integer code), replaced (or encoded) by a binary representation according to some rule to encode integer numbers into binary.

Specifically, the algorithm LZ78 [LZ78] can be stated using the double-tree framework by initially setting $\mathcal{L}_0 = \{\lambda, x_0\}$, $\mathcal{M}_0 = \mathcal{A}$ and $\mathbf{s}_0 = (\ell_0 \circ \mathbf{m}_0) = (\lambda \circ x_0) = x_0$. At a general step i, the sets $\mathcal{L}_{i-1}$ and $\mathcal{M}_{i-1}$ are known, the source string has been parsed in i phrases $\mathbf{s}_0, \cdots, \mathbf{s}_{i-1}$ and there is a remaining unparsed string which will be denoted by $\mathbf{z}_i$. The algorithm is described next.

Algorithm LZ78

$i = 0$

$\mathbf{z}_0 = x_0^{N-1}$
$\mathcal{L}_0 = \{\lambda\}$, $\mathcal{M}_0 = \mathcal{A}$
$\mathbf{s}_0 = \ell_0 \circ \mathbf{m}_0$ with $\ell_0 = \lambda$ and $\mathbf{m}_0 = x_0$.

$1 \leq i \leq t$

1. Update unparsed string:
   $\mathbf{z}_i = \mathbf{z}_{i-1} - (\ell_{i-1} \circ \mathbf{m}_{i-1})$
2. Find longest match $\mathbf{s}_i$ with respect to $\mathcal{D}_i = \mathcal{L}_{i-1} \circ \mathcal{M}_{i-1}$:
   $\mathbf{s}_i = \Pi[\mathbf{z}_i|\mathcal{D}_i] = \ell_i \circ \mathbf{m}_i$,
   with $\ell_i = \Pi[\mathbf{z}_i|\mathcal{L}_{i-1}]$, and $\mathbf{m}_i = \Pi[(\mathbf{z}_i - \ell_i)|\mathcal{M}_{i-1}]$.
3. $(J_i, K_i) = (\mathcal{X}[\ell_i|\mathcal{L}_{i-1}], \mathcal{X}[\mathbf{m}_i|\mathcal{M}_{i-1}])$
4. Update $\mathcal{L}$-tree:
   $\mathcal{L}_i = \mathcal{L}_{i-1} \cup \{\ell_i \circ \mathcal{F}[\mathbf{m}_i]\}$
   $\mathcal{M}_i = \mathcal{A}$
5. $(B_i, C_i) = (\phi_{\lceil \log |\mathcal{L}_{i-1}| \rceil}[J_i], \phi_{\lceil \log |\mathcal{M}_{i-1}| \rceil}[K_i])$

The efficiency of a string match algorithm is closely related to the number $t+1$ of phrases parsed off from the source string and to the rate of growth of the sets $\mathcal{L}$ and $\mathcal{M}$. In the present case, LZ78, $t+1$ phrases are generated and the N source symbols will be represented by L binary symbols,

$$L = \sum_{i=0}^{t} (|B_i| + |C_i|) = (t+1)\log_2 |\mathcal{A}| + \sum_{i=0}^{t} |B_i|,$$

rendering a $\rho = L/N$ compression rate. If the source symbols are drawn from an stationary source, the compression rate provedly [LZ78] converges to the entropy of the source. The interplay between these two parameters is quite envolved [S97] and is not our main concern. It is worth mentioning that Integer Codes more efficient than the one used to produce the binary block (B_i, C_i) could be used. An improvement in the above code, for instance, can be introduced simply by noticing that the phrase $\mathbf{s}_i$ which is parsed off at the i-th step, actually belongs to a set $\mathcal{D}_i$ (called dictionary or codebook)

$$\mathcal{D}_i = \mathcal{L}_{i-1} \circ \mathcal{M}_{i-1}$$

with some elements (or codewords) on it, which are not able to be selected as a match to $\mathbf{s}_i$ — the enumeration reserved for these are therefore a waste of bits. This is of little concern to us at this point and the Integer Code as it is will be used with the other algorithm versions discussed in the entire work.

The important point to be stressed in relation to the LZ78 is that no matter the value of i, the associated tree $\mathcal{M}_i$ is kept fixed, equal to $\mathcal{A}$. Whether there are procedures which performs more efficiently, by allowing $\mathcal{M}_i$, the second dictionary tree, to grow rather than be fixed, is a conjecture naturally raised. This issue is examined on the next section. A variation of the LZ78 which constructs the dictionary $\mathcal{D}_i$ in a sligthly different manner and which, for this reason, has a slightly better performance will be presented. Example I ilustrates the workings of LZ78.

Example I

Let the sample string to be compressed be

$$\texttt{Sample0} = x_0^{33} = \texttt{aacabadababaacadabacabadadabababaaaba}$$

The quaternary source alphabet is $\mathcal{A} = \{\texttt{a,b,c,d}\}$. The sequence $\{\mathcal{L}_i : i = 0, 14\}$ of sets obtained with the LZ78 procedure, the corresponding phrases and binary codewords obtained are next presented.

Step $i = 0$
$\mathbf{z}_0 = \texttt{aacabadababaacadabacabadadabababaaaba}$
$\mathcal{L}_0 = \{\lambda\}$, $\mathcal{M}_0 = \mathcal{A}$
$\ell_0 = \lambda$, $\mathbf{m}_0 = \texttt{a}$
$\mathbf{s}_0 = \ell_0 \circ \mathbf{m}_0 = \texttt{a}$, $W_0 = 00$
Step $i = 1$
$\mathbf{s}_0, \mathbf{z}_1 = \texttt{a,acabadababaacadabacabadadabababaaaba}$
$\mathcal{L}_1 = \{\texttt{a}\}$
$\ell_0 = \texttt{a}$, $\mathbf{m}_0 = \texttt{c}$
$\mathbf{s}_1 = \ell_1 \circ \mathbf{m}_1 = \texttt{a} \circ \texttt{c}$, $W_1 = 1\ 10$
Keep going like this will take us to
$\mathbf{s}_0^{13}\mathbf{z}_{14}$ =a,ac,ab,ad,aba,b,aa,c,ada,ba,ca,bad,adab,abaa,aba
$\mathcal{L}_{14} = \{$ a, ac, ab, ad, aba, b, aa, c, ada, ba, ca, bad, adab, abaa $\}$
$\mathbf{s}_{14} = \texttt{aba}\ \lambda$, $W_{14} = 0101\ \lambda$

3.2 A Less Greedy LZ78

We observe, in the plain LZ78 discussed on Section 3.1, that the set $\mathcal{L}_i$ is increasead by one element at each step i, i.e., $|\mathcal{L}_i| = |\mathcal{L}_{i-1}| + 1$. The dictionary $\mathcal{D}_i$ is built by transforming the tree corresponding to $\mathcal{L}_{i-1}$ into a complete tree having only terminal nodes and nodes with exactly $|\mathcal{A}|$ branches stemming from them. This greedy expansion of the set $\mathcal{L}_{i-1}$ seems to be one reason for the degraded performance of the LZ78 algorithm, as compared to other variations, such as LZW for instance. The variation introduced in this section (lg-LZ, in short), allows for a less-greedy expansion in order to get the dictionary $\mathcal{D}_i$. The longest string match is not found this time (lg-LZ), with respect to the dictionary $\mathcal{D}_i = \mathcal{L}_{i-1} \circ \mathcal{M}_{i-1}$ but, instead, with respect to the dictionary

$$\mathcal{D}_i = \mathcal{L}_{i-1} \cup \{\mathbf{s}_i \circ \mathcal{A}\}.$$

The dictionary $\mathcal{D}_i$ is now built by expanding the $\mathcal{L}_{i-1}$ tree by appending to the node corresponding to the path just selected as a longest match, the tree corresponding to the alphabet $\mathcal{A}$. The algorithm is stated next.

Algorithm lg-LZ

$i = 0$

$\mathbf{z}_0 = x_0^{N-1}$
$\mathcal{L}_0 = \mathcal{A}$, $\mathcal{M}_0 = \mathcal{A}$
$\mathbf{s}_0 = x_0$
$J_0 = \mathcal{X}[\mathbf{s}_0|\mathcal{A}]$, $B_0 = \phi_{\lceil \log|\mathcal{A}| \rceil}[J_0]$

$1 \leq i \leq t$

1. Update unparsed string
   $\mathbf{z}_i = \mathbf{z}_{i-1} - \mathbf{s}_{i-1}$
2. Find longest match $\mathbf{s}_i$ with respect to $\mathcal{D}_i = \mathcal{L}_{i-1} \cup \{\mathbf{s}_{i-1} \circ \mathcal{M}_{i-1}\}$
   $\mathbf{s}_i = \Pi[\mathbf{z}_i|\mathcal{D}_i]$,
3. $J_i = \mathcal{X}[\mathbf{s}_i|\mathcal{D}_i])$
4. $B_i = \phi_{\lceil \log|\mathcal{D}_i| \rceil}[J_i]$
5. Updating tree
   $\ell_{new} = \mathbf{s}_{i-1} \circ \mathcal{F}[\mathbf{s}_i]$
   if $|\ell_{new}| = |\mathbf{s}_i|$ and $\mathbf{s}_i \notin \mathcal{L}_{i-1}$ then $\ell_{new} = \mathbf{s}_i$
   $\mathcal{L}_i = \mathcal{L}_{i-1} \cup \{\ell_{new}\}$
   $\mathcal{M}_i = \mathcal{A}$

Also here we have $\mathbf{s}_i = \ell_i \circ \mathbf{m}_i$ with, possibly, $\mathbf{m}_i = \lambda$. The performances displayed on Table 2, obtained by computer simulation show instances where the lg-LZ performs better when compared to its counterpart LZW. The example presented next ilustrate the workings of the lg-LZ.

Example II

Let x_0^{33} = `aacabadababaacadabacabadadababaaaba`. $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. The parsing that the procedure lg-LZ yields is

`a, ac, a, b, a, d, ab, aba, aca, da, ba, c, aba, da, dab, abaa, aba`

The compressed representation of x_0^{33} is a binary string with 72 bits — compression rate of 0.257

3.3 Lempel-Ziv-Welch Algorithm

The Lempel-Ziv-Welch procedure, popularly called LZW, is known to have a performance on the average 10% better then the plain LZ78 version. One aspect that makes the LZW different from LZ78 is that it works with a rule that build the dictionary $\mathcal{D}_i$ by appending only one node to the corresponding tree $\mathcal{L}_{i-1}$.

The following would be the description of the LZW algorithm.

Algorithm LZW

$i = 0$

$\mathbf{z}_0 = x_0^{N-1}$ $\mathcal{L}_0 = \mathcal{A}$,
$\mathcal{M}_0 = \{\lambda\}$ and
$\mathbf{s}_0 = x_0$. $\ell_0 = x_0$

$1 \leq i \leq t$

1. $\mathbf{z}_i = \mathbf{z}_{i-1} - \mathbf{s}_{i-1}$
2. Find longest match with respect to $\mathcal{D}_i = \mathcal{L}_{i-1} \cup \ell_{i-1} \circ \mathcal{M}_{i-1}$
   $\ell_i = \Pi[\mathbf{z}_i|\mathcal{L}_{i-1}]$,
   $\mathbf{s}_i = \Pi[\mathbf{z}_i|\mathcal{D}_i]$,
3. $J_i = \mathcal{X}[\mathbf{s}_i|\mathcal{D}_i]$
4. $\mathcal{L}_i = \mathcal{D}_i$
   $\mathcal{M}_i = \{\mathcal{F}[\mathbf{z}_i - \mathbf{s}_i]\}$
5. $B_i{=}\phi_{\lceil \log|\mathcal{D}_i|\rceil}[J_i]$

Example III

Consider again `Sample0` $= x_0^{33} =$ `aacabadababaacadabacabadadababaaaba` with $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. This sequence is parsed into 20 phrases as follows

```
a, a, c, a, b, a, d, ab, aba, ac, ad, aba, ca, ba, da, da,
ba, ba, aa, ba
```

and its compressed representation is a binary string with 81 bits — a compression rate of 0.289

4 Description of Double-tree Algorithms

In the previous section two known agorithms (LZ78 and LZW) and a simple variation of the former (lg-LZ) were stated within the framework of a double-tree string match. Each one of the algorithms produce a sequence of trees $\{\mathcal{L}_i\}_{i=0,t}$ and corresponding sequence of dictionaries $\{\mathcal{D}_i\}_{i=0,t}$ with a string match done with respect to each dictionary. The basic difference among the three algorithms relies in the manner in which the tree $\mathcal{L}_{i-1}$ is concatenated with the corresponding $\mathcal{M}_{i-1}$, to build the dictionary $\mathcal{D}_i$. Table 1 summarizes this aspect.

$$
\begin{array}{lrcl}
\text{LZ78:} & |\mathcal{D}_i| & = & |\mathcal{L}_{i-1} \circ \mathcal{M}_{i-1}| \\
 & & \leq & |\mathcal{L}_{i-1}||\mathcal{M}_{i-1}| \\
\text{lg-LZ:} & |\mathcal{D}_i| & = & |\mathcal{L}_{i-1} \cup \{\ell_i \circ \mathcal{A}\}| \\
 & & = & |\mathcal{L}_{i-1}| + |\mathcal{A}| \\
\text{LZW:} & |\mathcal{D}_i| & = & |\mathcal{L}_{i-1} \cup \ell_{i-1} \circ \mathcal{M}_{i-1}| \\
 & & = & |\mathcal{L}_{i-1}| + 1
\end{array}
$$

Table 1: Length of the dictionaries

A point which is common to the three algorithms so far discussed is that they all concatenate the set $\mathcal{L}_{i-1}$ with a depth one tree in order to build their dictionaries. It is quite natural at this point to ask whether there are procedures which performs more efficiently when the second dictionary tree is allowed to have depth greater than one. A double-tree string match algorithm, with a second tree having a more general structure is stated in this section. Allowing a more general structure for the second tree $\mathcal{M}_{i-1}$, enlarge the number of algorithm variations that can be stated. The search for string matches are now searches for double-matches — this imply that more general ways to search are possible and that the longest-match is not necessarily a concatenation of a string ℓ_i (which is the longest match with respect to the tree $\mathcal{L}_{i-1}$) with the string $\mathbf{m}_i$ (which is the longest match with respect to the tree $\mathcal{M}_{i-1}$). Now, in order to optimize the number $t+1$ of parses, the best strategy is to search for a concatenation $(\ell_i \circ \mathbf{m}_i)$ which among all double-matches, have the largest size $|\ell_i|+|\mathbf{m}_i|$. We have implemented one version of a double-match/double-tree procedure and analysed their performance by computer simulations. The algorithm, which will be, abreviatedly, referred to as dt-LZ, is presented next.

Algorithm dt-LZ

$i = 0$ (Initialization step)

- $\mathbf{z}_0 = x_0^{N-1}$
- $\mathcal{L}_0 = \mathcal{M}_0 = \mathcal{A}$
- $\mathbf{m}_0 = \Pi[\mathbf{z}_0|\mathcal{M}_0]$,
- $K_0 = \mathcal{X}[\mathbf{m}_0|\mathcal{M}_0]$;
- $C_0 = \phi_{\lceil \log |\mathcal{M}_0| \rceil}[K_0]$
- $\mathbf{z}_1 = \mathbf{z}_0 - \mathbf{m}_0$;
- $\mathcal{M}_0 = \mathcal{M}_0 \cup \{\mathbf{m}_0 \circ \mathcal{F}[\mathbf{z}_1]\}$

$1 \leq i \leq t$ (Generic step)

1. Segmentation:
   (a) $\ell_i = \Pi[\mathbf{z}_i|\mathcal{L}_{i-1}]$,
   $\mathbf{z}_{temp} = \mathbf{z}_i - \ell_i$,
   $\mathbf{m}_i = \Pi[\mathbf{z}_{temp}|\mathcal{M}_{i-1}]$,
   $\tau = |\ell_i| + |\mathbf{m}_i|$,
   $\mathbf{u} = \ell_i$.
   (b) i. $\mathbf{u} = \mathcal{S}[\mathbf{u}]$
   $\mathbf{z}_{temp} = \mathbf{z}_i - \mathbf{u}$
   $\mathbf{v} = \Pi[\mathbf{z}_{temp}|\mathcal{M}_{i-1}]$.
   ii. If $(|\mathbf{u}| + |\mathbf{v}| \geq \tau)$: $(\ell_i, \mathbf{m}_i) = (\mathbf{u}, \mathbf{v})$, $\tau = |\ell_i| + |\mathbf{m}_i|$.
   iii. If $|\mathbf{u}| > 0$ return to step (i).
   (c) $\mathbf{z}_i = (\mathbf{z}_i - \ell_i) - \mathbf{m}_i$
2. Update Dictionaries:
   $\mathcal{L}_i = \mathcal{L}_{i-1} \cup \{\ell_i \circ \mathcal{F}[\mathbf{m}_i]\}$
   $\mathcal{M}_i = \mathcal{M}_{i-1} \cup \{\mathbf{m}_i \circ \mathcal{F}[\mathbf{z}_i]\}$
3. Map to Integer
   $(J_i, K_i) = (\mathcal{X}[\ell_i|\mathcal{L}_{i-1}],\ \mathcal{X}[\mathbf{m}_i|\mathcal{M}_{i-1}])$

4. Integer Code:
$$(B_i, C_i) = (\phi_{\lceil \log |\mathcal{L}_{i-1}| \rceil}[J_i],\ \phi_{\lceil \log |\mathcal{M}_{i-1}| \rceil}[K_i])$$

Example IV

Let x_0^{33} = `aacabadababaacadabacabadadababaaaba`. $\mathcal{A} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. The parsing for the procedure dt-LZ yields is
`(-,a), (a,c), (a,b), (a,d), (a,ba), (b,aa), (c,a), (d,a), (ba,ca), (ba,da), (da,bab), (a,aa),(b,a).`
where we show the double-matches displayed in parenthesis.

5 Some Computer Simulation Results

The algorithms discussed have been implemented as computer programs which were used to compress some sample sequences. Although the performance of all these algorithms are optimum in the sense that their compression rate asymptotically converges to the entropy of the information source or to the Lempel-Ziv complexity of the individual sequence, they perform quite differently when finite sequences and the rate of convergence to the asymptotic optimum are considered. Table 2 displays some of the simulation results exhibiting the performance of the algorithms. We have not

Sequence (size)	LZW (size)	lg-LZ (size)	dt-LZ (size)
`Sample0` (280)	.289 (81)	.257 (72)	.311 (87)
`Sample1` (576)	.089 (51)	.099 (57)	.097 (56)
`Sample2` (544)	.077 (42)	.086 (47)	.103 (56)
`Sample3` (672)	.357 (240)	.371 (249)	.335 (225)
`Sample4` (256)	.258 (66)	.113 (29)	.320 (82)

Table 2: Compression rate of algorithms LZW, lg-LZ and dt-LZ (all sequence sizes, in parenthesis, are in bits)

presented results for the LZ78 algorithm. As the other versions this algorithm is asymptotically optimum but has an inferior perfomance as compared to the LZW. As it can be noticed from the results presented in Table 2 the behavior of the algorithms are sequence dependent. For some sequences the LZW can achieve a better result than the lg-LZ — this gain is basically due to the penalty paid by the lg-LZ for expanding the first tree with $\mathcal{A}$ nodes to build the dictionary, instead of the one node expansion done by the LZW. This gain in performance tend to disappear as the sequence length grows larger. Examining the line on Table 2 corresponding to `Sample4` one can see that the performance of lg-LZ can converge considerably fast to the optimum, as compared to LZW, for certain types of sequences. These are sequences constructed to benefit the performance of lg-LZ (no such construction can be done, we conjecture, to benefit LZW).

Conclusion

We have proposed algorithms which are based on the idea of string matches with respect to two sets or, equivalently, string match with respect to two trees. Many implementations variations of these algorithms are possible — a double-string match with respect to two trees version (called dt-LZ) was implemented.

In our preliminary investigation we exam the behavior of these algorithms and analyse its performance by computer simulation. Also we stated the well known LZ78 algorithm [LZ78] in the framework of string match with respect to two trees, as well as the LZW [W84]. A simple modification of the LZ78 was also proposed (this was called lg-LZ).

It is our expectation that higher compression can be achieved with double-string match with respect to two trees procedures. This is based on the argument that the use of two trees allows the construction of concatenated trees with more general structures, leaving more room for optimizing the search. It is also based on results we have obtained with multiple-string matches algorithms [PFP99] — which achieve a better compression than single-matches ones. These multiple-string match algorithms are based on the double-tree idea yet the two trees involved in the process are kept equal.

The results presented in this work do not single out a definite better double-match/double-tree algorithm — if one can be found — but bring to our attention that there are many variations. Our investigations will be further pursued by examining other double-match/double-tree implementations. An extension of the multiple-match described in [PFP99] will also be sought.

References

[LZ78] Ziv, J., Lempel, A., "Compression of individual sequences via variable-rate coding," IEEE Trans. Inform. Theory, vol. IT-24, pp.530-536, Sep. 1978.

[W84] Welch, T. A., "A technique for high-performance data compression," Computer, vol. 17, pp.8-19, Jun. 1984.

[FPC98] Finamore, W. A., Pinho, M. S., Craizer, M., "A multi-string match algorithm for lossless data compression," Abstracts of Invited Lectures and Short Communications, 7^{th} International Colloquium on Numerical Analysis and Computer Sciences with Applications, p.39, Plovdiv, Bulgaria, Aug. 1998.

[PFP99] Pinho, M. S., Finamore, W. A., Pearlman, W. A., "Fast multi-match Lempel-Ziv," Proc. of IEEE Data Compression Conference, Snowbird, UT, April 1999.

[HR85] Hartman, A., Rodeh, M., "Optimal Parsing of Strings," Combinatorial Algorithms on Words, Springer-Verlag, A. Apostolico & Z. Galil, editors, pp. 155-167, 1985.

[S97] Savari, S. A., "Redundancy of Lempel-Ziv incremental parsing rule," IEEE Trans. Inform. Theory, vol. IT-43, pp.9-21, Jan. 1997.

A New Practical Linear Space Algorithm for the Longest Common Subsequence Problem*

H. Goeman, M. Clausen

Institut für Informatik V
Universität Bonn
Römerstraße 164
D–53117 Bonn
Germany

e-mail: {goeman,clausen}@cs.uni-bonn.de

Abstract. This paper deals with a new practical method for solving the longest common subsequence (LCS) problem. Given two strings of lengths m and n, $n \geq m$, on an alphabet of size s, we first present an algorithm which determines the length p of an LCS in $O(ns + \min\{mp, p(n-p)\})$ time and $O(ns)$ space. This result has been achieved before [Ric94, Ric95], but our algorithm is significantly faster than previous methods. We also provide a second algorithm which generates an LCS in $O(ns + \min\{mp, m \log m + p(n-p)\})$ time while preserving the linear space bound, thus solving the problem posed in [Ric94, Ric95]. Experimental results confirm the efficiency of our method.

Key words: Design and analysis of algorithms, edit distance, longest common subsequence.

1 Introduction

Let $x = x_1 \ldots x_m$ and $y = y_1 \ldots y_n$, $n \geq m$, be two strings over an alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_s\}$ of size s. A *subsequence* of x is a sequence of symbols obtained by deleting zero or more characters from x. The *Longest Common Subsequence* (LCS) *Problem* is to find a common subsequence of x and y which is of greatest possible length.

It will be convenient to describe the problem in another way. An ordered pair (k, ℓ), $1 \leq k \leq m$, $1 \leq \ell \leq n$, is called a *match* if $x_k = y_\ell$. The set M of all matches can be identified with a *matching matrix* of size $m \times n$ in which each match is marked with a dot. For example, if $x = abacbcba$ and $y = cbabbacac$, then M is as shown in Fig. 1 (a). Define a partial order $\ll$ on $\mathbb{N} \times \mathbb{N}$ by establishing $(k, \ell) \ll (k', \ell')$ iff both $k < k'$ and $\ell < \ell'$. A *chain* $C \subseteq M$ is a set of points which are pairwise comparable, i.e., for any two distinct $p_1, p_2 \in C$, either $p_1 \ll p_2$ or $p_1 \gg p_2$, where $p_1 \gg p_2$ means $p_2 \ll p_1$. Then the LCS problem can be viewed as finding a chain of maximal cardinality in M. One such chain is indicated as a path in Fig. 1 (b).

Finding an LCS is closely related with the computation of string edit distances [LW75, MP80, Wag75, WC76] and shortest common supersequences [GMS80]. It was

*Research supported by Deutsche Forschungsgemeinschaft, Grant CL 64/3-1

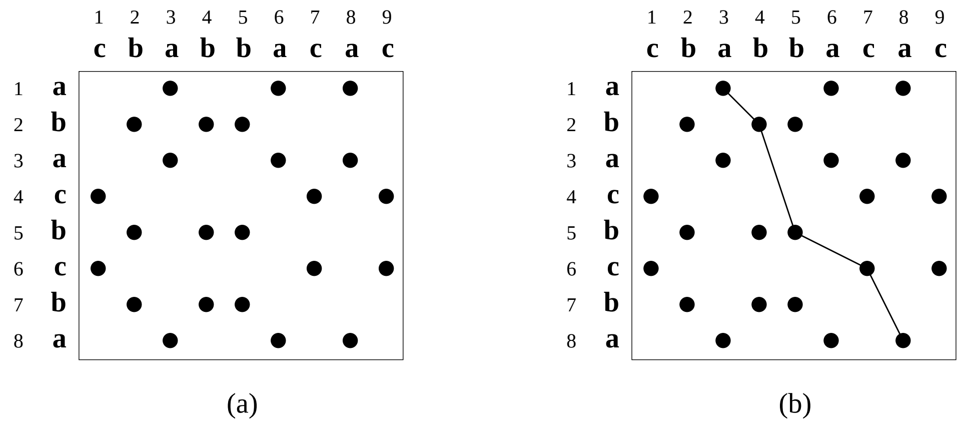

Figure 1: (a) matching matrix, (b) path representing an LCS.

first used by molecular biologists to study similar amino acids [Day65, Day69, NW70, SC73]. Other applications are in data compression [AHU76, GMS80, Mai78] and pattern recognition [FB73, LF78].

The LCS problem can be solved in $O(mn)$ time by a dynamic programming approach [SK83, WF74], while the asymptotically fastest general solution uses the "four russians" trick and takes $O(nm/\log n)$ time [MP80]. A lot of other algorithms have also been developed which are sensitive to other problem parameters, e.g., the length p of an LCS. They usually perform much better than the latter algorithms, although they all have a worst case time complexity at least of $\Omega(mn)$. For example, Hunt and Szymanski [HS77] have presented an $O((r+n)\log n)$ algorithm, where $r := |M|$. Thus their approch is fast when r is small, e.g., $r = O(n)$, but its worst–case time complexity is $O(n^2 \log n)$. Later, this has been improved to $O(mn)$ [Apo86]. There are also several routines which run in $O(n(n+1-p))$ or $O(n(m+1-p))$ time, and thus are efficient when an LCS is expected to be long [Mye86, NKY82, Ukk85, WMM90]. Other algorithms have running times $O(n(p+1))$ or $O(m(p+1))$ and should be used for short LCS [Apo87, AG87, Hir77, HD84]. However, it might be very difficult to *a priori* select a good strategy because in general the length p cannot be easily estimated. Also, when having a small alphabet, we can expect p to be of intermediate size, e.g., for $s = 4$, the average length of an LCS is bounded between $0.54 \cdot m \leq p \leq 0.71 \cdot m$ [CS75, DP94, Dek79, PD94, SK83]. Then none of the above methods performs well. Therefore recent research has been concentrated on more flexible algorithms which are efficient for short, intermediate, and long LCS, such as the method proposed by Chin/Poon [CP94]. Another approach from Rick [Ric94, Ric95] with running time $O(ns + \min\{mp, p(n-p)\})$ has been widely accepted as the fastest algorithm for the general LCS problem.

In this paper, we shall develop a new algorithm which is based on a kind of dualization of Rick's method. A detailed description of the theoretical background will be given in Sect. 2 and 3. Our idea does not improve the $O(ns + \min\{mp, p(n-p)\})$ time bound, but two important advantages are obtained. First, the number of matches processed while computing the length of an LCS is significantly decreased, resulting in a faster execution speed. The corresponding algorithm will be presented in Sect. 4. Second, when generating an LCS, we can achieve linear space through a divide–and–conquer scheme similar to that of other (but slower) algorithms [ABG92, Hir75, KR87]. This will be explained in Sect. 5. The methods mentioned before all need at least $\Omega(nm/\log n)$ space in their worst cases (see [PD94] for a survey), and most of them, including Rick's approach, cannot be combined with the divide–and–conquer technique. The open problem of a linear space implementation of Rick's algorithm [Ric95] is hereby solved. Experimental results presented in Sect. 6 confirm the efficiency of our method.

2 A New Approach to the LCS Problem

As already mentioned in the introduction, the LCS problem is equivalent to finding a chain of maximum cardinality in M. Dilworth's fundamental theorem [Dil50] states that this cardinality equals the minimum number of disjoint *antichains* into which M can be decomposed (an antichain of M consists of matches which are pairwise incomparable). In our example, this number (called the *Sperner number* of M) equals five. A suitable decomposition is shown in Fig. 2 (f). To find such a minimum decomposition, we first split $[1:m] \times [1:n]$ into subsets denoted by T^i, L^i, B^i, and R^i, where

$$\begin{aligned} T^i &:= \{i\} \times [i : n+1-i] \\ L^i &:= [i+1 : m+1-i] \times \{i\} \\ B^i &:= \{m+1-i\} \times [i+1 : n+1-i] \\ R^i &:= [i+1 : m-i] \times \{n+1-i\} \end{aligned}$$

and $1 \leq i \leq \lceil m/2 \rceil$ (see Fig. 2 (a) for an illustration). Additionally, let

$$T^{\leq i} := \bigcup_{j \leq i} T^j, \quad L^{\leq i} := \bigcup_{j \leq i} L^j, \quad B^{\leq i} := \bigcup_{j \leq i} B^j, \quad R^{\leq i} := \bigcup_{j \leq i} R^j \ .$$

Now for $i = 1, 2, \ldots, \lceil m/2 \rceil$, we construct four sets of antichains $A^{T,i}$, $A^{L,i}$, $A^{B,i}$, and $A^{R,i}$ which decompose (a suitable subset of) $T^{\leq i}$, $L^{\leq i}$, $B^{\leq i}$, and $R^{\leq i}$, respectively. The decompositions are generated by updating the previous sets, using the matches found in T^i, L^i, B^i, and R^i (details are given below). We use $A_u^{T,i}$ to denote an antichain in $A^{T,i}$, where u is an index between 1 and the size $e^{T,i} := |A^{T,i}|$ of $A^{T,i}$. Therefore $e^{T,i}$ is also called the *end index* of $A^{T,i}$. For $A^{L,i}$, $A^{B,i}$, and $A^{R,i}$, we introduce analogous notations. Furthermore, there are two *start indices* $s^{TL,i}$ and $s^{BR,i}$. The first one is used to split both $A^{T,i}$ and $A^{L,i}$ into two parts. One part contains all antichains with indices less than $s^{TL,i}$, and the other part consists of the rest. Only the latter part will be used for the updating process, whereas the former one will be copied to $A^{T,i+1}$ resp. $A^{L,i+1}$ without change. $s^{BR,i}$ similarly splits $A^{B,i}$ and $A^{R,i}$.

Fig. 2 (b), (c), (d), and (e) give a preview of the construction in the sample matching matrix after step $i = 1$, 2, 3, and 4, respectively. The centered grey box represents the remaining part of M which has not been processed so far. By our construction, with each step, it shrinks by two rows and columns.

We need the following terminology for the description of the construction process. For two antichains $C, D \subseteq M$ the set

$$IP(C, D) := \{p_1 \in C \mid \forall\, p_2 \in D\colon\ \neg(p_1 \ll p_2 \vee p_1 \gg p_2)\}$$

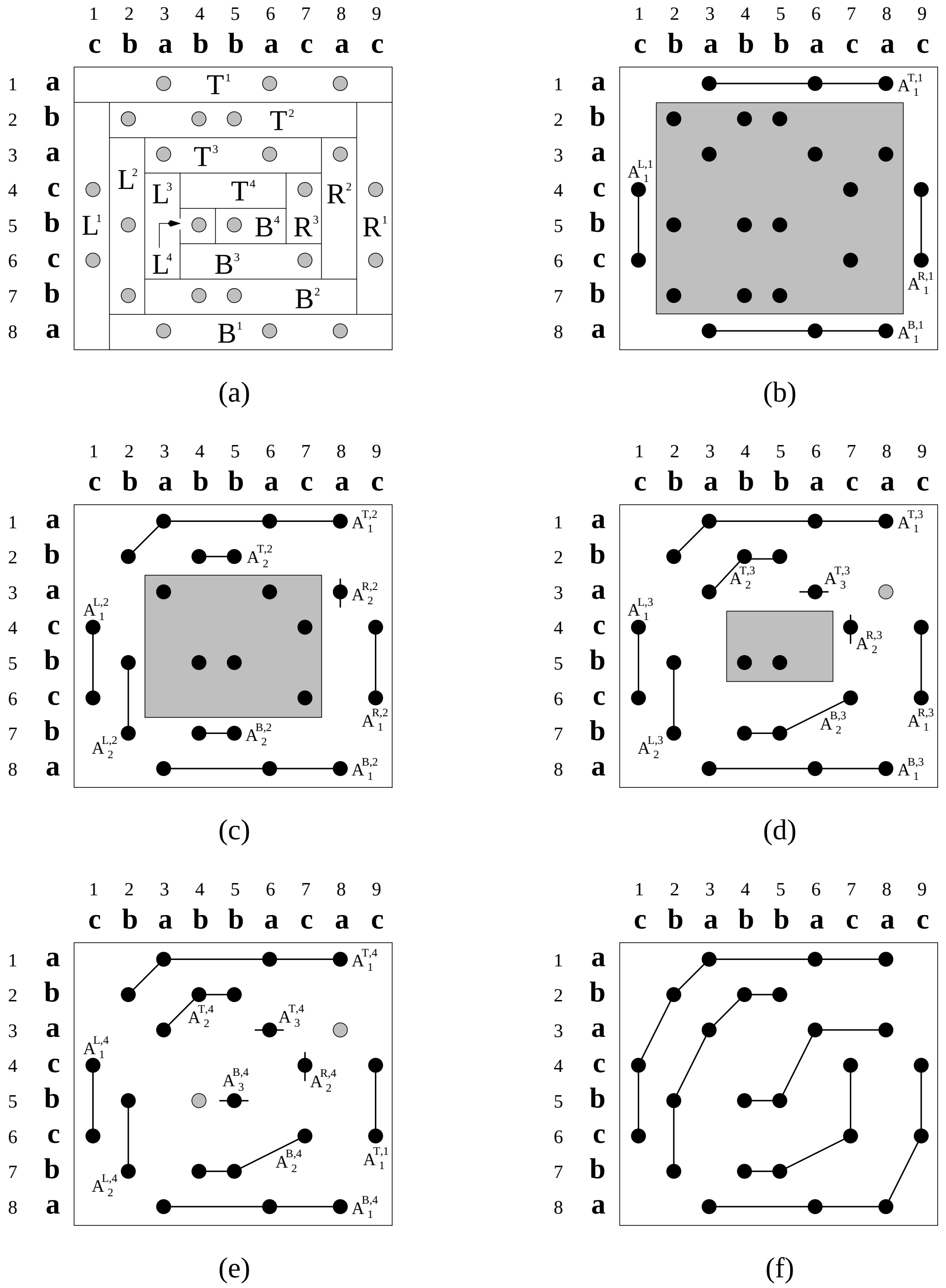

Figure 2: (a) splitting of M, (b)–(e) construction of antichains, (f) final decomposition.

is called the *incomparable part* of C relative to D. Clearly, $IP(C, D) \cup D$ is the greatest antichain above D contained in $C \cup D$. We say C is *incomparable* to D if $IP(C, D) = C$, and a single match $p_1 \in M$ is *incomparable* to D if $IP(\{p_1\}, D) = \{p_1\}$.

We are now prepared to discuss the generation of the antichains in more detail. Initially, there are no antichains, i.e., we have $A^{T,0} = A^{L,0} = A^{B,0} = A^{R,0} = \emptyset$ by initializing each start and end index to 1 and 0, respectively. Then, for each step $i = 1, \ldots, \lceil m/2 \rceil$, we start with T^i to determine $A^{T,i}$ from $A^{T,i-1}$. Let $s := s^{TL,i-1}$ and $e := e^{T,i-1}$. The first $s - 1$ antichains remain unchanged and are simply copied from $A^{T,i-1}$ to $A^{T,i}$. Now define $A_s^{T,i}$ as $A_s^{T,i-1} \cup IP(T^i \cap M, A_s^{T,i-1})$. For example, when processing T^2 in Fig. 2 (b), $IP(T^2 \cap M, A_1^{T,1}) = \{(2, 2)\}$, and thus the match $(2, 2)$

combined with $A_1^{T,1}$ makes up $A_1^{T,2}$ as shown in Fig. 2 (c). Next, for $u = s+1, \ldots, e$, the antichain $A_u^{T,i-1}$ is handled in the same way to set up $A_u^{T,i}$, but only those matches in T^i not belonging to $A_s^{T,i}, \ldots, A_{u-1}^{T,i}$ are considered. Finally, we establish $s^{TL,i} := s$ and, if there are no matches left, $e^{T,i} := e$. Otherwise, we set $e^{T,i}$ to $e+1$ and collect all remaining matches in a new antichain $A_{e+1}^{T,i}$. Also, if $A^{R,i-1} \neq \emptyset$, we check whether its last antichain $A_{\tilde{e}}^{R,i-1}$, $\tilde{e} := e^{R,i-1}$, is incomparable to $A_{e+1}^{T,i}$. In this case we say $A_{\tilde{e}}^{R,i-1}$ is *inactivated* by $A_{e+1}^{T,i}$, and we remove $A_{\tilde{e}}^{R,i-1}$ from $A^{R,i}$ by setting $e^{R,i} := e^{R,i-1}$. Continuing our example with T^2 in Fig. 2 (b), we see there are two matches $(2,4)$ and $(2,5)$ left after processing $A_1^{T,2}$. Therefore a new antichain $A_2^{T,2}$ is created, but $A_1^{R,1}$ remains unchanged because, for example, $(2,4) \ll (4,9)$. The final set $A^{T,2}$ is shown in Fig. 2 (c) (the modifications to the other antichains are described below). Now let us consider the work involved with T^3. The match $(3,3)$ cannot be put into $A_1^{T,3}$, but into $A_2^{T,3}$, and the other match $(3,6)$ makes up the new antichain $A_3^{T,3}$. This time $(3,6)$ inactivates $(3,8)$, and thus $A_2^{R,2}$ is removed. The result is illustrated in Fig. 2 (d) (all matches located in deleted antichains are indicated by grey dots).

```
S := T^i ∩ M;     (* Determine A^{T,i} *)
For u := s^{TL,i-1} To e^{T,i-1} Do {
  A_u^{T,i} := A_u^{T,i-1} ∪ IP(S, A_u^{T,i-1});
  S := S \ IP(S, A_u^{T,i-1});
};
If S ≠ ∅ Then {
  e^{T,i} := e^{T,i-1} + 1; e := e^{T,i}; A_e^{T,i} := S;
  e^{R,i} := e^{R,i-1}; ẽ := e^{R,i};
  If s^{BR,i-1} ≤ e^{R,i-1} Then {
    If IP(A_ẽ^{R,i-1}, A_e^{T,i}) = A_ẽ^{R,i-1} Then {
      D^{TR} := D^{TR} ∪ A_ẽ^{R,i-1};
      e^{R,i} := ẽ - 1;
    };
  };
} Else { e^{T,i} := e^{T,i-1}; e^{R,i} := e^{R,i-1} };
For u := 1 To s^{TL,i-1} - 1 Do A_u^{T,i} := A_u^{T,i-1};

S := L^i ∩ M;     (* Determine A^{L,i} *)
For u := s^{TL,i-1} To e^{L,i-1} Do {
  A_u^{L,i} := A_u^{L,i-1} ∪ IP(S, A_u^{L,i-1});
  S := S \ IP(S, A_u^{L,i-1});
};
If S ≠ ∅ Then {
  e^{L,i} := e^{L,i-1} + 1; e := e^{L,i}; A_e^{L,i} := S;
  e^{B,i} := e^{B,i-1}; ẽ := e^{B,i};
  If s^{BR,i-1} ≤ e^{B,i-1} Then {
    If IP(A_ẽ^{B,i-1}, A_e^{L,i}) = A_ẽ^{B,i-1} Then {
      D^{BL} := D^{BL} ∪ A_ẽ^{B,i-1};
      e^{B,i} := ẽ - 1;
    };
  };
} Else { e^{L,i} := e^{L,i-1}; e^{B,i} := e^{B,i-1} };
For u := 1 To s^{TL,i-1} - 1 Do A_u^{L,i} := A_u^{L,i-1};
s^{TL,i} := s^{TL,i-1};
```

(a)

```
S := B^i ∩ M;     (* Determine A^{B,i} *)
For u := s^{BR,i-1} To e^{B,i} Do {
  A_u^{B,i} := A_u^{B,i-1} ∪ IP(S, A_u^{B,i-1});
  S := S \ IP(S, A_u^{B,i-1});
};
If S ≠ ∅ Then {
  e^{B,i} := e^{B,i} + 1; e := e^{B,i}; A_e^{B,i} := S;
  If s^{TL,i} ≤ e^{L,i} Then {
    ẽ := e^{L,i};
    If IP(A_ẽ^{L,i}, A_e^{B,i}) = A_ẽ^{L,i} Then {
      D^{BL} := D^{BL} ∪ A_ẽ^{L,i};
      e^{L,i} := ẽ - 1;
    };
  };
};
For u := 1 To s^{BR,i-1} - 1 Do A_u^{B,i} := A_u^{B,i-1};

S := R^i ∩ M;     (* Determine A^{R,i} *)
For u := s^{BR,i-1} To e^{R,i} Do {
  A_u^{R,i} := A_u^{R,i-1} ∪ IP(S, A_u^{R,i-1});
  S := S \ IP(S, A_u^{R,i-1});
};
If S ≠ ∅ Then {
  e^{R,i} := e^{R,i} + 1; e := e^{R,i}; A_e^{R,i} := S;
  If s^{TL,i} ≤ e^{T,i} Then {
    ẽ := e^{T,i};
    If IP(A_ẽ^{T,i}, A_e^{R,i}) = A_ẽ^{T,i} Then {
      D^{TR} := D^{TR} ∪ A_ẽ^{T,i};
      e^{T,i} := ẽ - 1;
    };
  };
};
For u := 1 To s^{BR,i-1} - 1 Do A_u^{R,i} := A_u^{R,i-1};
s^{BR,i} := s^{BR,i-1};
```

(b)

Figure 3: The algorithms for generating $A^{T,i}$ & $A^{L,i}$ (a), and $A^{B,i}$ & $A^{R,i}$ (b).

Having determined $A^{T,i}$, we continue with the necessary calculations for $A^{L,i}$ which are very similar. The first $s-1$ antichains are copied and then, for $u = s, \ldots, e^{L,i-1}$, $A_u^{L,i}$ is defined as the union of $A_u^{L,i-1}$ and the incomparable part of L^i relative to $A_u^{L,i-1}$, where only those matches are considered which have not already been used. Remaining matches form a new antichain and, if they are incomparable to the last antichain in $A^{B,i-1}$, we decrease $e^{B,i}$ by one. The corresponding algorithm in Fig. 3 (a) also introduces two additional sets D^{TR} and D^{BL} which contain all deleted matches. Details will be given in the next section.

Before processing $A^{B,i-1}$ and $A^{R,i-1}$ in an analogous way, we first check whether the first antichain in $A^{T,i}$ or $A^{L,i}$ is *TL–complete*, i.e., whether one of them contains a match (k,ℓ) such that $1 \leq k,\ell \leq i$. For example, in the configuration shown in Fig. 2 (c), $A_1^{T,2}$ is TL–complete due to the match $(2,2)$. As soon as $A_s^{T,i}$ is detected to be TL–complete, $s^{TL,i}$ is increased by one, thus the first antichains in both corresponding sets which are checked for additional matches remain unchanged from now on. If there is no such antichain in $A^{L,i}$ (i.e. $s > e^{L,i}$), but $s^{BR,i-1} \leq e^{B,i}$, then we additionally test whether $A_s^{T,i}$ is incomparable to the last antichain in $A^{B,i-1}$ and, should this situation arise, delete this antichain from $A^{B,i}$ by decreasing $e^{B,i}$.

Now assume $A_s^{L,i}$ is TL–complete. Then, as shown in Fig. 4 (a), we also increase $s^{TL,i}$, and similarly, if $s > e^{T,i}$ and $s^{BR,i-1} \leq e^{R,i}$, we decrease $e^{R,i}$ if $A_s^{L,i}$ inactivates the last antichain in $A^{R,i}$.

```
(* Check $A^{T,i}$ for TL–completeness *)
If $s^{TL,i} \leq e^{T,i}$ Then {
  $s := s^{TL,i}$;
  If $\exists (k,\ell) \in A_s^{T,i}: k,\ell \leq i$ Then {
    If $s > e^{L,i}$ Then {
      If $s^{BR,i-1} \leq e^{B,i}$ Then {
        $\tilde{e} := e^{B,i}$;
        If $IP(A_{\tilde{e}}^{B,i-1}, A_s^{T,i}) = A_{\tilde{e}}^{B,i-1}$ Then {
          $D^{BL,i} := D^{BL,i} \cup A_{\tilde{e}}^{B,i-1}$;
          $e^{B,i} := \tilde{e} - 1$;
        };
      };
      $e^{L,i} := s$; $A_s^{L,i} := \emptyset$;
    };
    $s^{TL,i} := s + 1$;
  };
};

(* Check $A^{L,i}$ for TL–completeness *)
If $s^{TL,i} \leq e^{L,i}$ Then {
  $s := s^{TL,i}$;
  If $\exists (k,\ell) \in A_s^{L,i}: 1 \leq k,\ell \leq i$ Then {
    If $s > e^{T,i}$ Then {
      If $s^{BR,i-1} \leq e^{R,i}$ Then {
        $\tilde{e} := e^{R,i}$;
        If $IP(A_{\tilde{e}}^{R,i-1}, A_s^{L,i}) = A_{\tilde{e}}^{R,i-1}$ Then {
          $D^{TR,i} := D^{TR,i} \cup A_{\tilde{e}}^{R,i-1}$;
          $e^{R,i} := \tilde{e} - 1$;
        };
      };
      $e^{T,i} := s$; $A_s^{T,i} := \emptyset$;
    };
    $s^{TL,i} := s + 1$;
  };
};
```

(a)

```
(* Check $A^{B,i}$ for BR–completeness *)
If $s^{BR,i} \leq e^{B,i}$ Then {
  $s := s^{BR,i}$;
  If $\exists (k,\ell) \in A_s^{B,i}: k > m - i \wedge \ell > n - i$ Then {
    If $s > e^{R,i}$ Then {
      If $s^{TL,i} \leq e^{T,i}$ Then {
        $\tilde{e} := e^{T,i}$;
        If $IP(A_{\tilde{e}}^{T,i}, A_s^{R,i}) = A_{\tilde{e}}^{T,i}$ Then {
          $D^{TR,i} := D^{TR,i} \cup A_{\tilde{e}}^{T,i}$;
          $e^{T,i} := \tilde{e} - 1$;
        };
      };
      $e^{R,i} := s$; $A_s^{R,i} := \emptyset$;
    };
    $s^{BR,i} := s + 1$;
  };
};

(* Check $A^{R,i}$ for BR–completeness *)
If $s^{BR,i} \leq e^{R,i}$ Then {
  $s := s^{BR,i}$;
  If $\exists (k,\ell) \in A_s^{R,i}: k > m - i \wedge \ell > n - i$ Then {
    If $s > e^{B,i}$ Then {
      If $s^{TL,i} \leq e^{L,i}$ Then {
        $\tilde{e} := e^{L,i}$;
        If $IP(A_{\tilde{e}}^{L,i}, A_s^{B,i}) = A_{\tilde{e}}^{L,i}$ Then {
          $D^{BL,i} := D^{BL,i} \cup A_{\tilde{e}}^{L,i}$;
          $e^{L,i} := \tilde{e} - 1$;
        };
      };
      $e^{B,i} := s$; $A_s^{B,i} := \emptyset$;
    };
    $s^{BR,i} := s + 1$;
  };
};
```

(b)

Figure 4: The algorithms for handling complete antichains in $A^{T,i}$ & $A^{L,i}$ (a), and in $A^{B,i}$ & $A^{R,i}$ (b).

The remaining work in step i concerns with the analogous construction of $A^{B,i}$ and $A^{R,i}$. (The analogue of TL–completeness is called *BR–completeness*. An antichain is BR–complete if it contains a match (k,ℓ) with $m - i < k \leq m$ and $n - i < \ell \leq n$.) Details are available from the algorithms shown in Fig. 3 (b) and Fig. 4 (b).

The main program shown in Fig. 5 is straightforward. Our next task is to elaborate the connection between the generated antichains and a minimal decomposition of M. This is done in the next section.

$i := 1$; (∗ Initialization ∗)
$s^{T,0} := 1$; $s^{L,0} := 1$; $s^{B,0} := 1$; $s^{R,0} := 1$;
$e^{T,0} := 0$; $e^{L,0} := 0$; $e^{B,0} := 0$; $e^{R,0} := 0$;
For $i := 0$ **To** $\lceil m/2 \rceil$ **Do** $D^{TL,i} := \emptyset$;
For $i := 0$ **To** $\lfloor m/2 \rfloor$ **Do** $D^{BR,i} := \emptyset$;

While $i \leq \lfloor m/2 \rfloor$ **Do** { (∗ Main loop ∗)
  Determine $A^{T,i}$ and $A^{L,i}$; (∗ see Fig. 3 (a) ∗)
  Look for TL-complete antichains in $A^{T,i}$ and $A^{L,i}$; (∗ see Fig. 4 (a) ∗)
  Determine $A^{B,i}$ and $A^{R,i}$; (∗ see Fig. 3 (b) ∗)
  Look for BR-complete antichains in $A^{B,i}$ and $A^{R,i}$; (∗ see Fig. 4 (b) ∗)
  $i := i + 1$;
};

If Odd(m) **Then** {
  Determine $A^{T,\lceil m/2 \rceil}$ and $A^{L,\lceil m/2 \rceil}$; (∗ see Fig. 3 (a) ∗)
  Look for TL-complete antichains in $A^{T,\lceil m/2 \rceil}$ and $A^{L,\lceil m/2 \rceil}$; (∗ see Fig. 4 (a) ∗)
};

Figure 5: The main program for decomposing M

3 Analysis of the Construction

In this section, we study how to combine the antichains into larger ones such that a minimal decomposition of M is obtained. We further establish some results which later help us to construct an LCS in linear space.

Let us assume m is odd, and let $i = \lceil m/2 \rceil$. For technical reasons, we then put $A_u^{B,i} := A_u^{B,i-1}$ and $A_u^{R,i} := A_u^{R,i-1}$ for all $1 \leq u \leq e^{B,i-1}$ and $1 \leq u \leq e^{R,i-1}$. We also set $s^{BR,i} := s^{BR,i-1}$, $e^{B,i} := e^{B,i-1}$, and $e^{R,i} := e^{R,i-1}$. Furthermore, for $0 \leq i \leq \lceil m/2 \rceil$, we define $A_u^{T,i} := \emptyset$, $A_u^{L,i} := \emptyset$, $A_u^{B,i} := \emptyset$, and $A_u^{R,i} := \emptyset$ for $u > e^{T,i}$, $u > e^{L,i}$, $u > e^{B,i}$, and $u > e^{R,i}$, respectively.

Lemma 3.1 *Let $1 \leq i \leq \lceil m/2 \rceil$. Then the following holds:*

a) $\forall\, s^{TL,i-1} \leq u < v \leq e^{T,i}\ \forall p_1 \in A_v^{T,i}\ \exists p_2 \in A_u^{T,i}\colon\ p_1 \gg p_2$.

b) $\forall\, s^{TL,i-1} \leq u < v \leq e^{L,i}\ \forall p_1 \in A_v^{L,i}\ \exists p_2 \in A_u^{L,i}\colon\ p_1 \gg p_2$.

c) $\forall\, s^{BR,i-1} \leq u < v \leq e^{B,i}\ \forall p_1 \in A_v^{B,i}\ \exists p_2 \in A_u^{B,i}\colon\ p_1 \ll p_2$.

d) $\forall\, s^{BR,i-1} \leq u < v \leq e^{R,i}\ \forall p_1 \in A_v^{R,i}\ \exists p_2 \in A_u^{R,i}\colon\ p_1 \ll p_2$.

Proof. We only show the first claim, the other proofs are similar. Let $p_1 = (k, \ell)$. Since $A_v^{T,i} \subseteq T^{\leq \lceil m/2 \rceil}$, p_1 has been added to $A_v^{T,k}$ while processing T^k in step k, and $k \leq i$. Clearly, from the way S is handled in lines 1–5 of Fig. 3 (a), $p_1 \notin \mathit{IP}(T^k \cap M, A_j^{T,k-1})$, for $s^{TL,k-1} \leq j < v$. Hence, since $s^{TL,k-1} \leq s^{TL,i-1} \leq u < v$, there is some $p_2 \in A_u^{T,k-1}$ such that $p_1 \gg p_2$ or $p_1 \ll p_2$. But the second case would imply $p_2 \in T^{k'}$ for some $k' > k$ which is impossible during the first k steps of our construction. Finally observe that the algorithm never removes matches while updating an antichain, thus p_2 is still present in $A_u^{T,i}$. □

Lemma 3.2 *The following holds:*

a) $\forall\, 1 \leq i \leq \lceil m/2 \rceil\ \forall\, v\colon\ v < s^{TL,i} \iff A_v^{T,i}$ *or* $A_v^{L,i}$ *is TL–complete .*

b) $\forall\, 1 \leq i \leq \lceil m/2 \rceil\ \forall\, v\colon\ v < s^{BR,i} \iff A_v^{B,i}$ *or* $A_v^{R,i}$ *is BR–complete .*

Proof. We only prove the first claim, the other one is similar.
If. By contradiction, let i be the first step such that $A_v^{T,i}$ or $A_v^{L,i}$ is TL–complete, but $v \geq s^{TL,i}$. Clearly $v \neq s^{TL,i-1}$, otherwise the TL–completeness would have been detected by the algorithm shown in Fig. 4 (a), and thus, contradicting the property of v, we would have $v < s^{TL,i} = s^{TL,i-1}+1$. Hence $v > s^{TL,i-1}$. By the TL–completeness, there is some match $(k,\ell) \in A_v^{T,i} \cup A_v^{L,i}$ such that $1 \leq k, \ell \leq i$. Furthermore, by Lemma 3.1, there exists some match $(k',\ell') \in A_{v-1}^{T,i} \cup A_{v-1}^{L,i}$ such that $(k',\ell') \ll (k,l)$. But then $1 \leq k', \ell' < i$, and therefore either $A_{v-1}^{T,i}$ or $A_{v-1}^{L,i}$ would be TL-complete after step $i-1$, a contradiction to the choice of i.
Only if. Obvious from the management of the start indices. □

Lemma 3.3 *For all i, u define $A_u^{TL,i} := A_u^{T,i} \cup A_u^{L,i}$ and $A_u^{BR,i} := A_u^{B,i} \cup A_u^{R,i}$. Then*

a) $\forall\, 0 \leq i \leq \lceil m/2 \rceil\ \forall\, 1 \leq u \leq \min\{e^{T,i}, e^{L,i}\}\colon\ A_u^{TL,i}$ *is an antichain .*

b) $\forall\, 0 \leq i \leq \lceil m/2 \rceil\ \forall\, 1 \leq u \leq \min\{e^{B,i}, e^{R,i}\}\colon\ A_u^{BR,i}$ *is an antichain .*

Proof. We prove the first claim by induction on i. The base $i = 0$ it trivial because $A^{T,0} = A^{L,0} = \emptyset$. For the induction step $i-1 \to i$, we consider three different cases.
Case a: $1 \leq u < s^{TL,i-1}$. Then $A_u^{T,i} = A_u^{T,i-1}$ and $A_u^{L,i} = A_u^{L,i-1}$ (see lines 15 and 30 in Fig. 3 (a), respectively). Thus, by the induction hypothesis, $A_u^{TL,i}$ is an antichain.
Case b: $s^{TL,i-1} \leq u \leq \min\{e^{T,i-1}, e^{L,i-1}\}$. By definition the set $T := IP(S, A_u^{T,i-1})$ added to $A_u^{T,i}$ in line 3 (Fig. 3 (a)) is incomparable to $A_u^{T,i-1}$, but it is also incomparable to $A_u^{L,i}$ as we now demonstrate. Let $(k,\ell) \in IP(S, A_u^{T,i-1})$ and $(k',\ell') \in A_u^{L,i}$. Observe $k = i$ and $\ell \geq i$. Also note that $k' > \ell'$ and $\ell' \leq i$ because $A_u^{L,i} \subseteq L^{\leq i}$. Thus $(k,\ell) \ll (k',\ell')$ would contradict $\ell \geq i \geq \ell'$. Furthermore, $(k',\ell') \ll (k,\ell)$ would imply $\ell' < k' < k = i$, i.e., $A_u^{L,i-1}$ would be TL-complete, a contradiction to Lemma 3.2 and the choice of u. Similar arguments can be used for the set $L := IP(S, A_u^{L,i-1})$ added to $A_u^{L,i}$ in line 19. Finally note that $T \subseteq T^i$ and $L \subseteq L^i$ are also incomparable.
Case c: $\min\{e^{T,i-1}, e^{L,i-1}\} < u \leq \min\{e^{T,i}, e^{L,i}\}$. Clearly, this case is only possible if $u = e^{T,i} = e^{T,i-1}+1$ or $u = e^{L,i} = e^{L,i-1}+1$. If both conditions hold, then $A_u^{T,i} \subseteq T^i \cap M$ (lines 1 and 7) and $A_u^{L,i} \subseteq L^i \cap M$ (lines 17 and 23), thus their union obviously makes up an antichain. Otherwise, only one new antichain is generated whereas the other one is updated, and we can argument as in the second case to show that both antichains are incomparable.

The proof of the second claim is similar. □

Lemma 3.4 *Let $1 \leq i \leq \lceil m/2 \rceil$. Then the following holds:*

a) $\forall\, j \leq \max\{e^{T,i}, e^{L,i}\}\ \forall\, p_j \in A_j^{TL,i}\ \exists\, p_1 \in A_1^{TL,i}, \ldots, p_{j-1} \in A_{j-1}^{TL,i}$:
$p_1 \ll \ldots \ll p_j$.

b) $\forall j \le \max\{e^{B,i}, e^{R,i}\}\ \forall p_j \in A_j^{BR,i}\ \exists p_1 \in A_1^{BR,i}, \ldots, p_{j-1} \in A_{j-1}^{BR,i}$:
$p_1 \gg \ldots \gg p_j$.

Proof. We prove the first claim by choosing p_v for $v = j-1, \ldots, 1$.

Consider step $j' \le i$ when p_{v+1} was added to $A_{v+1}^{TL,j'} \subseteq A_{v+1}^{TL,i}$. Then Lemma 3.1 implies the existence of p_v if $v \ge s^{TL,j'-1}$. Otherwise, by Lemma 3.2, $A_v^{T,j'-1}$ or $A_v^{L,j'-1}$ has been detected to be TL–complete before step j', i.e., $A_v^{TL,j'-1}$ contains a match (k', ℓ') such that $k', \ell' < j'$. But p_{v+1} is of the form (k, ℓ) with $k, \ell \ge j'$, thus we can choose $p_v := (k', \ell')$.

Similar arguments can be used for the second claim. □

Lemma 3.5 *For $0 \le i \le \lceil m/2 \rceil$, there are two chains*

$$C^{TR,i}, C^{BL,i} \subseteq T^{\le i} \cup L^{\le i} \cup B^{\le i} \cup R^{\le i}$$

of length $e^{T,i} + e^{R,i}$ and $e^{B,i} + e^{L,i}$, respectively.

Proof. We prove the existence of the first chain $C^{TR,i}$ by induction on i. The base $i = 0$ is trivial. For the induction step $(i-1) \to i$, we have to analyse the situations which cause $e^{T,i} + e^{R,i}$ to be greater than $e^{T,i-1} + e^{R,i-1}$. One such situation is given in lines 7–14 of Fig. 3 (a) if the condition in line 10 is not satisfied because then $e := e^{T,i} = e^{T,i-1} + 1$ and $\tilde{e} := e^{R,i} = e^{R,i-1}$. But since $IP(A_{\tilde{e}}^{R,i-1}, A_e^{T,i}) \ne A_{\tilde{e}}^{R,i-1}$ there exist two comparable matches $c^T \in A_e^{T,i}$ and $c^R \in A_{\tilde{e}}^{R,i-1}$. More precisely, since $c^T \in T^i$ and $c^R \in R^{\le i-1}$, we must have $(k, \ell) \ll (k', \ell')$. Thus, by Lemma 3.4, we can construct a chain

$$p_1 \ll \ldots \ll p_{e-1} \ll c^T \ll c^R \ll p'_{\tilde{e}-1} \ll \ldots \ll p'_1$$

of length $e + \tilde{e}$.

Similar arguments can be used for the remaining situations and for the other chain. □

Our next task is to reveal the structure in D^{TR} and D^{BL}. We shall show that for each deleted match there always is some antichain which is incomparable to this match. In order to prove this property, we keep track of each deleted match by *assigning* it to some antichain during the construction process. More precisely, whenever an antichain A is removed due to the existence of some other antichain B which inactivates it, all matches in A are assigned to B, e.g., considering the situation in Fig. 2 (d), the match $(3, 8)$ is assigned to $A_3^{T,3}$. Furthermore, all previously deleted matches assigned to A now also belong to B. The assigned matches are inherited when an antichain is updated, e.g., in Fig. 2 (e), $(3, 8)$ also belongs to $A_3^{T,4}$. These rules guarantee that after step i, each deleted match is assigned to exactly one antichain in $A^{T,i} \cup A^{L,i} \cup A^{B,i} \cup A^{R,i}$. We write $D(A)$ to denote the set of matches assigned to an antichain A.

Lemma 3.6 *Let $1 \le i \le \lceil m/2 \rceil$, and assume $(k, \ell) \in D(A)$ for some antichain A in $A^{T,i}$, $A^{L,i}$, $A^{B,i}$, or $A^{R,i}$. Then*

a) $(k, \ell) \in D^{TR} \implies \forall (k', \ell') \in A\colon\ k \le k' \wedge \ell \ge \ell'$.

b) $(k,\ell) \in D^{BL} \implies \forall\,(k',\ell') \in A\colon\ k \geq k' \wedge \ell \leq \ell'$.

Proof. For the first claim, let us assume (k,ℓ) was assigned to A while executing line 11 in Fig. 3 (a) during step $j \leq i$ (the following arguments can analogously be applied to the other instructions which modify D^{TR}). Thus $A = A_e^{T,i}$, where $e = e^{T,j}$. Now we consider two cases concerning the status of (k,ℓ) before step j.
Case a: $(k,\ell) \in A_{\tilde{e}}^{R,j-1} \subseteq R^{\leq j-1}$, $\tilde{e} = e^{R,j-1}$. Then $\ell > n-j+1$. From lines 1, 6, 7, and 10 we see that (k,ℓ) is incomparable to any match (k'',ℓ'') in $A_e^{T,j}$. But $A_e^{T,j} \subseteq T^j$, thus $k'' = j$ and $\ell'' \leq n-j+1$. Hence, the incomparability implies $k \leq j$. Now observe that $A_e^{T,j}$ is the first constructed part of $A_e^{T,i}$, later extensions are taken from $T^{j+1},\ldots,T^i$. Thus every match $(k',\ell') \in A_e^{T,i}$ fulfills $k' \geq j$ and $\ell' \leq n-j+1$, and the claim follows.
Case b: (k,ℓ) is assigned to $A_{\tilde{e}}^{R,j-1}$. We can inductively assume

$$\forall\,(k'',\ell'') \in A_{\tilde{e}}^{R,j-1}\colon\ k \leq k'' \wedge \ell \geq \ell''$$

Deleted matches are never assigned to empty antichains. Thus there is at least one match $(k'',\ell'') \in A_{\tilde{e}}^{R,j-1}$, and we can prove as in the first case that $k'' \leq k'$ and $\ell'' \geq \ell'$. Hence we have $k \leq k'$ and $\ell \geq \ell'$.

The proof of the second claim follows similar arguments and is therefore omitted. □

Lemma 3.7 *Let $1 \leq i \leq \lceil m/2 \rceil$. Then the following holds:*

a) $\forall\,1 \leq u \leq e^{T,i}\colon\ D^{BL} \cap D(A_u^{T,i}) \neq \emptyset \implies A_u^{L,i} = \emptyset \wedge A_u^{T,i}$ *is TL–complete .*

b) $\forall\,1 \leq u \leq e^{L,i}\colon\ D^{TR} \cap D(A_u^{L,i}) \neq \emptyset \implies A_u^{T,i} = \emptyset \wedge A_u^{L,i}$ *is TL–complete .*

c) $\forall\,1 \leq u \leq e^{B,i}\colon\ D^{TR} \cap D(A_u^{B,i}) \neq \emptyset \implies A_u^{R,i} = \emptyset \wedge A_u^{B,i}$ *is BR–complete .*

d) $\forall\,1 \leq u \leq e^{R,i}\colon\ D^{BL} \cap D(A_u^{R,i}) \neq \emptyset \implies A_u^{B,i} = \emptyset \wedge A_u^{R,i}$ *is BR–complete .*

Proof. We again only show the first claim. From lines 10 and 11 in Fig. 3 (a), we see that all matches assigned there to $A_u^{T,i}$ are either placed into D^{TR}, or they have been assigned before to some non–complete antichain in $A^{R,i-1}$. But concerning the latter case, we see from lines 26 and 27 in Fig. 3 (b) that any such match has been put into D^{TR} as well, or again belongs to some non–complete antichain in $A^{T,j}$, $j < i$. Repeating this argument, we conclude that all matches assigned to $A^{T,i}$ are contained in D^{TR}. The only exception is given by lines 8 and 9 in Fig. 4 (a), where deleted matches are assigned to $A_u^{T,i}$, but added to D^{BL}. But then, from lines 3, 4, and 13, the claim follows. □

Lemma 3.8 *All matches assigned to an antichain A are pairwise incomparable, thus by Lemma 3.6, they extend the antichain to a larger one.*

Proof. Whenever a match is deleted, the algorithm always removes a complete antichain. By induction, this antichain B together with its assigned matches forms a larger antichain C. If there already is a set of matches D assigned to A (which is only possible when A is detected to be complete), then, following the arguments given in the proof of Lemma 3.7, $C \subseteq D^{BL}$ and $D \subseteq D^{TR}$ or vice versa, and Lemma 3.6 immediately implies that B and D are pairwise incomparable. $\square$

We are now prepared to construct a minimal decomposition of M. We start by decomposing $M \setminus (D^{TR} \cup D^{BL})$, the deleted matches are later considered in Thm. 3.9 below. The construction is as follows. Using Lemma 3.3, we combine the first $e^{TL} := \min\{e^{T,\lceil m/2\rceil}, e^{L,\lceil m/2\rceil}\}$ antichains in $A^{T,\lceil m/2\rceil}$ and $A^{L,\lceil m/2\rceil}$ to larger ones. We also connect the first $e^{BR} := \min\{e^{B,\lceil m/2\rceil}, e^{R,\lceil m/2\rceil}\}$ antichains in $A^{B,\lceil m/2\rceil}$ to the corresponding ones in $A^{R,\lceil m/2\rceil}$. For example, in Fig. 2 (e), we have $e^{T,\lceil m/2\rceil} = e^{B,\lceil m/2\rceil} = 3$ and $e^{L,\lceil m/2\rceil} = e^{R,\lceil m/2\rceil} = 2$, thus this generates four combined antichains. Concerning the remaining antichains we consider four different cases.

Case a: $e^{T,\lceil m/2\rceil} \leq e^{L,\lceil m/2\rceil}$ and $e^{B,\lceil m/2\rceil} \geq e^{R,\lceil m/2\rceil}$. Then we leave the remaining antichains as they are and have $p := e^{L,\lceil m/2\rceil} + e^{B,\lceil m/2\rceil}$ antichains in total. But by Lemma 3.5, there also exists a chain of this length. Thus, by Dilworth's theorem, the decomposition is minimal.

Case b: $e^{T,\lceil m/2\rceil} > e^{L,\lceil m/2\rceil}$ and $e^{B,\lceil m/2\rceil} \leq e^{R,\lceil m/2\rceil}$. Similar to the first case we have $p := e^{T,\lceil m/2\rceil} + e^{R,\lceil m/2\rceil}$ antichains, and also a chain of this length.

Case c: $e^{T,\lceil m/2\rceil} \leq e^{L,\lceil m/2\rceil}$ and $e^{B,\lceil m/2\rceil} < e^{R,\lceil m/2\rceil}$. From the management of the start and end indices, we have $e^{T,\lceil m/2\rceil} \geq s^{TL,\lceil m/2\rceil} - 1$. Thus, by Lemma 3.2, $A_u^{L,\lceil m/2\rceil}$ is not TL–complete for $u > e^{T,\lceil m/2\rceil}$. This implies $k > \lceil m/2\rceil$ and $\ell \leq \lceil m/2\rceil$ for any match $(k,\ell) \in A_u^{L,\lceil m/2\rceil} \subseteq L^{\leq\lceil m/2\rceil}$. For all $v > e^{B,\lceil m/2\rceil}$ and $(k',\ell') \in A_v^{R,\lceil m/2\rceil}$ we similarly have $k' \leq \lceil m/2\rceil$ and $\ell' > n - \lfloor m/2\rfloor \geq \lceil m/2\rceil$. Thus $A_u^{L,\lceil m/2\rceil}$ and $A_v^{R,\lceil m/2\rceil}$ are incomparable. Now assume $e^{L,\lceil m/2\rceil} \geq e^{R,\lceil m/2\rceil}$. Then we can connect all remaining antichains in $A^{R,\lceil m/2\rceil}$ to corresponding ones in $A^{L,\lceil m/2\rceil}$ and obtain $p := e^{L,\lceil m/2\rceil} + e^{B,\lceil m/2\rceil}$ antichains in total, thus again a minimal decomposition. If $e^{L,\lceil m/2\rceil} < e^{R,\lceil m/2\rceil}$, then similarly $p := e^{T,\lceil m/2\rceil} + e^{R,\lceil m/2\rceil}$ is the optimal length of a chain in $M \setminus (D^{TR} \cup D^{BL})$.

Case d: $e^{T,\lceil m/2\rceil} > e^{L,\lceil m/2\rceil}$ and $e^{B,\lceil m/2\rceil} > e^{R,\lceil m/2\rceil}$. Finding a minimal decomposition is slightly more complicated in this case. Consider the following algorithm. Starting with $u := e^{T,\lceil m/2\rceil}$ and $v := e^{R,\lceil m/2\rceil} + 1$, we check whether $A_u^{T,\lceil m/2\rceil}$ and $A_v^{B,\lceil m/2\rceil}$ are incomparable. If they are not, then we backup u and v in $\tilde{u}$ and $\tilde{v}$, respectively, and increase v by one. Otherwise the antichains are connected, u is set to $u-1$, and v is set to $v+1$. We repeat this until all remaining antichains in either $A^{T,\lceil m/2\rceil}$ or $A^{B,\lceil m/2\rceil}$ have been used, i.e., $u = e^{L,\lceil m/2\rceil}$ or $v > e^{B,\lceil m/2\rceil}$. Then the total number of antichains is $p := u + e^{B,\lceil m/2\rceil}$. Thus, if $u = e^{L,\lceil m/2\rceil}$, we have $p = e^{L,\lceil m/2\rceil} + e^{B,\lceil m/2\rceil}$, and the decomposition is optimal. Now assume $u > e^{L,\lceil m/2\rceil}$. If $\tilde{u}$ and $\tilde{v}$ are unused, then all remaining antichains in $A^{B,\lceil m/2\rceil}$ have been connected to corresponding antichains in $A^{T,\lceil m/2\rceil}$, and we have $p = e^{T,\lceil m/2\rceil} + e^{R,\lceil m/2\rceil}$. Hence, in this case the decomposition is also a minimal one. Finally assume that $\tilde{u}$ and $\tilde{v}$ have been used for saving u and v at least once. Then for $j = \tilde{v}+1, \ldots, e^{B,\lceil m/2\rceil}$, $A_j^{B,\lceil m/2\rceil}$ has been connected to $A_{\tilde{u}+\tilde{v}-j}^{T,\lceil m/2\rceil}$, and we have $u = \tilde{u} - (e^{B,\lceil m/2\rceil} - \tilde{v})$. Thus $p = \tilde{u} - (e^{B,\lceil m/2\rceil} - \tilde{v}) + e^{B,\lceil m/2\rceil} = \tilde{u} + \tilde{v}$. But from the properties of $\tilde{u}$ and $\tilde{v}$, it can be shown (similar to the proof of Lemma 3.5) that there is a chain of length $\tilde{u} + \tilde{v}$ which contains two matches $p_1 \in A_{\tilde{u}}^{T,\lceil m/2\rceil}$ and $p_2 \in A_{\tilde{v}}^{B,\lceil m/2\rceil}$. Hence, the constructed decomposition is optimal.

Let us consider our example. Case d applies to the situation in Fig. 2 (e), and $A_3^{T,4}$ is compared with $A_3^{B,4}$. Since these antichains are incomparable, they are connected, and we obtain a decomposition consisting of 5 antichains in total.

Theorem 3.9 *The length of an LCS in M equals p as defined in the four cases above.*

Proof. Consider a combined antichain A of the decomposition. Assume an antichain $A_u^{T,\lceil m/2\rceil} \in A^{T,\lceil m/2\rceil}$ is one component of it (otherwise, we can handle the following construction in a similar way).
Case a: $A_u^{T,\lceil m/2\rceil}$ is the only component of A. Then we extend A with the set B of deleted matches assigned to $A_u^{T,\lceil m/2\rceil}$. Lemma 3.8 guarantees that the result is still an antichain.
Case b: $A_u^{T,\lceil m/2\rceil}$ has been combined with $A_u^{L,\lceil m/2\rceil}$. By Lemma 3.7, $B \subseteq D^{TR}$. Let $(k,\ell) \in A_u^{L,\lceil m/2\rceil}$ and $(k',\ell') \in A_u^{T,\lceil m/2\rceil}$. From $(k,\ell) \in L^{\lceil m/2\rceil}$, $(k',\ell') \in T^{\lceil m/2\rceil}$, and the incomparability of (k,ℓ) and (k',ℓ'), we have $k \geq k' \wedge \ell \leq \ell'$. Now consider a match $(k'',\ell'') \in B$. By Lemma 3.6, we have $k \geq k' \geq k''$ and $\ell \leq \ell' \leq \ell''$. Hence, $A_u^{L,\lceil m/2\rceil}$ is incomparable to B. We can use a similar way to show that the set C of deleted matches assigned to $A_u^{L,\lceil m/2\rceil}$ is a subset of D^{BL} and incomparable to $A_u^{T,\lceil m/2\rceil}$. Finally, B and C are clearly incomparable as well. Thus $A_u^{T,\lceil m/2\rceil} \cup A_u^{L,\lceil m/2\rceil} \cup B \cup C$ is still an antichain.
Case c: $A_u^{T,\lceil m/2\rceil}$ has been combined with some other antichain $D \in A^{B,i}$. Then, similar to the proof of the second case, we can show that the union of A and the two corresponding sets of assigned matches still make up an antichain.

By handling each combined antichain in this way, we can construct a decomposition of M without generating any additional antichains. The proof is complete. □

Fig. 2 (f) illustrates the corresponding decomposition for our example.

4 Implementation

We now describe an efficient implementation for the given algorithm and analyse its time and space complexity.

All new antichains created in step i are extensions from antichains generated during step $i-1$. Furthermore, the only antichains used for decomposing M are from the last step. Thus for the implementation it is sufficient to update the antichains of interest. The same is true for the start and end indices, and we thus sometimes drop the index i from now on. The necessary information for each actual antichain can be kept in one single number as follows. Let $1 \leq i \leq \lceil m/2\rceil$ and $1 \leq u \leq e^{T,i}$. We define *ThreshT*$[u]$ as the leftmost column used by some match in $A_u^{T,i}$, i.e.,

$$\mathit{ThreshT}[u] := \min\{\ell \mid \exists\, k\colon\ (k,\ell) \in A_u^{T,i}\}\ .$$

For example, in Fig. 2 (b), *ThreshT*$[1] = 3$, and in Fig. 2 (d), *Top-Thresh*$[1] = 2$, *ThreshT*$[2] = 3$, and *ThreshT*$[3] = 6$. To update this array in each step, we use an auxiliary array *LeftPos* on $\Sigma \times [1 : n+1]$ given by

$$\mathit{LeftPos}[c,\ell] := \min(\{n+1\} \cup \{j \mid \ell \leq j \leq n \wedge y_\ell = c\})\ ,$$

i.e., *LeftPos*$[a_i,\ell]$ equals the column number of the leftmost occurence of a match in row i located right to column ℓ, and equals $n+1$ if there is no such match. In our example ($y = cbabbacac$), we obtain the following values:

a	3	3	3	6	6	6	8	8	10	10
b	2	2	4	4	5	10	10	10	10	10
c	1	7	7	7	7	7	7	9	9	10

Now it is not difficult to see that the following routine correctly updates *ThreshT* when processing T^i, representing lines 1–7 in Fig. 3 (a). (Similar procedures are used in [AG87, Ric94, Ric95] to determine *contours* which correspond to the antichains used here.)

```
k := LeftPos[a_i, i];
For u := s^TL To e^T Do {
   j := ThreshT[u];
   If k ≤ j And k ≤ n − i + 1 Then {
      ThreshT[u] := k; k := LeftPos[a_i, j + 1];
   };
};
If k ≤ n − i + 1 Then { e^T := e^T + 1; ThreshT[e^T] := k };
```

For $A^{L,i}$, $A^{B,i}$, and $A^{R,i}$ we introduce additional arrays *ThreshL*, *ThreshB*, and *ThreshR* which similarly store the topmost rows, rightmost columns, and bottommost rows used by the corresponding antichains. To handle them analogously to *ThreshT*, we also need three more auxiliary arrays given by

$$\begin{array}{lll} \mathit{TopPos}[c,k] & := \min(\{m+1\} \cup \{j \mid k \le j \le m \wedge x_j = c\}) \ , & (1 \le k \le m+1) \ , \\ \mathit{RightPos}[c,\ell] & := \max(\{0\} \cup \{j \mid 1 \le j \le \ell \wedge y_\ell = c\}) \ , & (0 \le \ell \le n) \ , \\ \mathit{BottomPos}[c,k] & := \max(\{0\} \cup \{j \mid 1 \le j \le k \wedge x_j = c\}) \ , & (0 \le k \le m) \ . \end{array}$$

Note that in Fig. 3 and Fig. 4, each test for the incomparability of two antichains can be replaced by a rather simple conditional statement. For example, considering line 10 in Fig. 3 (a), we know that all matches in T^i are located to the left of any match in $R^{\le i-1}$. Thus, with $e := e^{T,i}$ and $\tilde{e} := e^{R,i}$, A^T_e and $A^R_{\tilde{e}}$ are incomparable if and only if $A^R_{\tilde{e}}$ is also completely contained in the first i rows, i.e., $\mathit{ThreshR}[\tilde{e}] \le i$. The algorithm presented in Fig. 6 shows how the other situations are handled. It also makes use of some special implementation details which cannot be discussed here, e.g., the construction starts with the bottommost row instead of the topmost one when m is even. In Fig. 6 some lines are marked with a dot (•) on their left sides. These lines are used for the construction of an LCS and should be ignored for the moment.

The complexity of the algorithm may be deduced as follows. The four auxiliary arrays can be easily preprocessed in $O(ns)$ time and space, where $s = |\Sigma|$. Clearly, during one of the $\lceil m/2 \rceil$ iterations of the main loop, none of the four inner **While**–loops takes more than $O(p)$ time, and when determining p, at most $\lceil m/2 \rceil$ pairs of antichains have to be compared. Thus the algorithm takes at most $O(ns+mp)$ time. Furthermore, observe that the j–th antichain in A^T (which is added to A^T during some step $i \ge j$) must contain a match (k,ℓ) with $\ell \le n-(p-j)$, otherwise it would be impossible to construct a chain of length p. But then this antichain is detected to be TL–complete after step $n-(p-j)$, therefore it is only considered for at most $n-(p-j)-i \le n-p$ times in the corresponding **While**–loop (lines 59–65). Similar arguments can be given for antichains in A^L, A^B, and A^R. Hence, we have shown the following theorem.

```
Determine TopPos and LeftPos;
Determine BottomPos and RightPos;
For u := 0 To ⌈m/2⌉ Do {
  ThreshT[u] := 0; ThreshL[u] := 0;
};
For u := 0 To ⌊m/2⌋ Do {
  ThreshB[u] := n + 1; ThreshR[u] := m + 1;
};
t := 1; ℓ := 1; b := m; r := n;
s^TL := 1; e^T := 0; e^L := 0;
s^BR := 1; e^B := 0; e^R := 0;

If Odd(m) Then Goto Line 57;

While t ≤ b Do {        (* Main loop *)

  k := RightPos[x_b, r];        (* Update A^B *)
  u := s^BR;
  While u ≤ e^B Do {
    j := ThreshB[u];
    If k ≥ j Then {
      ThreshB[u] := k; k := RightPos[x_b, j − 1];
    };
    u := u + 1;
  };
  If k ≥ ℓ Then {
    e^B := u; ThreshB[e^B] := k;
    If ThreshL[e^L] ≥ b Then e^L := e^L − 1
      Else Update c^B, c^L, ℓ^BL;
  };

  k := BottomPos[y_r, b − 1];        (* Update A^R *)
  u := s^BR;
  While u ≤ e^R Do {
    j := ThreshR[u];
    If k ≥ j Then {
      ThreshR[u] := k; k := BottomPos[y_r, j − 1];
    };
    u := u + 1;
  };
  If k ≥ t Then {
    e^R := u; ThreshR[e^R] := k;
    If ThreshT[e^T] ≥ r Then e^T := e^T − 1
      Else Update c^T, c^R, ℓ^TR;
  };

  (* Check for BR–complete antichains *)
  If ThreshB[s^BR] = r Then {
    If s^BR > e^R Then {
      If ThreshT[e^T] ≥ r Then e^T := e^T − 1
        Else Update c^T, c^R, ℓ^TR;
    };
    s^BR := s^BR + 1;
  } Else If ThreshR[s^BR] = b Then {
    If s^BR > e^B Then {
      If ThreshL[e^L] ≥ b Then e^L := e^L − 1
        Else Update c^B, c^L, ℓ^BL;
    };
    s^BR := s^BR + 1;
  };
  t := t + 1; ℓ := ℓ + 1;
  k := LeftPos[x_t, ℓ];        (* Update A^T *)
  u := s^TL;
  While u ≤ e^T Do {
    j := ThreshT[u];
    If k ≤ j Then {
      ThreshT[u] := k; k := LeftPos[x_t, j + 1];
    };
    u := u + 1;
  };
  If k ≤ r Then {
    e^T := u; ThreshT[e^T] := k;
    If ThreshR[e^R] ≤ t Then e^R := e^R − 1
      Else Update c^T, c^R, ℓ^TR;
  };

  k := TopPos[y_l, t];        (* Update A^L *)
  u := s^TL;
  While u ≤ e^L Do {
    j := ThreshL[u];
    If k ≤ j Then {
      ThreshL[u] := k; k := TopPos[y_l, j + 1];
    };
    u := u + 1;
  };
  If k ≤ b Then {
    e^L := u; ThreshL[e^L] := k;
    If ThreshB[e^B] ≤ ℓ Then e^B := e^B − 1
      Else Update c^B, c^L, ℓ^BL;
  };

  (* Check for TL–complete antichains *)
  If ThreshT[s^TL] = ℓ Then {
    If s^TL > e^L Then {
      If ThreshB[e^B] ≤ ℓ Then e^B := e^B − 1
        Else Update c^B, c^L, ℓ^BL;
    };
    s^TL := s^TL + 1;
  } Else If ThreshL[s^TL] = t Then {
    If s^TL > e^T Then {
      If ThreshR[e^R] ≤ t Then e^R := e^R − 1
        Else Update c^T, c^R, ℓ^TR;
    };
    s^TL := s^TL + 1;
  };
  b := b − 1; r := r − 1;
};

(* Determine length p of an LCS *)
If e^T > e^L And e^B > e^R Then {
  If s^TL ≤ e^L Then s^TL := e^L + 1;
  If s^BR ≤ e^R Then s^BR := e^R + 1;
  u := e^T; v := s^BR;
  While u ≥ s^TL And v ≤ e^B Do {
    If ThreshT[u] ≥ ThreshB[v]
      Then u := u − 1
      Else { ũ := u; ṽ := v };
    v := v + 1;
  };
  p := u + e^B;
} Else p := max{e^L + e^B, e^T + e^R};
```

Figure 6: The $O(ns + \min\{mp, p(n-p)\})$ algorithm for determining the length p of an LCS.

Theorem 4.1 *The length p of an LCS can be computed in $O(ns+\min\{mp, p(n-p)\})$ time and $O(ns)$ space.*

This result has been achieved before by Rick [Ric94, Ric95], and in fact, the algorithm presented here is some kind of dualization of Rick's method, but our algorithm is significantly faster as we shall show in Sect. 6.

5 Construction of an LCS in Linear Space

This section deals with the generation of an LCS. The idea is to apply the divide–and–conquer scheme [ABG92, Hir75, KR87] which first identifies at least one point of an LCS such that this LCS is splitted into two parts of roughly the same size. Then the remainder is computed by recursive calls. The method presented here usually determines two LCS–neighbouring matches c^{TL} and c^{BR} which are located in $T^{\le\lceil m/2\rceil} \cup L^{\le\lceil m/2\rceil}$ and $B^{\le\lceil m/2\rceil} \cup R^{\le\lceil m/2\rceil}$, respectively. This is accomplished as follows.

In each step i of the construction described in Sect. 2, we subsequently update the following variables:

- p^{TL} is the match which caused $A_s^{T,i}$ or $A_s^{L,i}$ to become TL–complete, $s = s^{TL,i}-1$. For example, in Fig. 2 (c), $p^{TL} = (2,2)$, and in Fig. 2 (d) and (e), $p^{TL} = (3,3)$.
- p^{BR} has a corresponding meaning for the last BR–complete antichain in $A^{B,i}$ and $A^{R,i}$, e.g., in Fig. 2 (d), $p^{BR} = (6,7)$.
- c^T and c^R are the two matches introduced in the proof of Lemma 3.5. They both lie in $C^{TR,i}$ and are neighbours in this chain. Furthermore, c^T and c^R are always located in the first i topmost rows and i rightmost columns of M, respectively.
- c^B and c^L have analogous properties for $C^{BL,i}$.
- ℓ^{TR} and ℓ^{BL} is the position of c^T in $C^{TR,i}$ and of c^L in $C^{BL,i}$, respectively. Also, $\ell^{TR}+1$ and $\ell^{BL}+1$ is the position of c^R in $C^{TR,i}$ and of c^B in $C^{BL,i}$, respectively.

p^{TL} and p^{BR} can be easily updated. For example, consider lines 85–98 in Fig. 6 where new TL–complete antichains are handled. Let $p^{TL} = (u,v)$. If the condition in line 86 is satisfied, then we know p^{TL} has to be set to the bottommost match located in the first t rows and column ℓ. Therefore two additional statements can be inserted between lines 86 and 87 such that u is set to $BottomPos[y_\ell, t]$ and v is set to ℓ. Similar statements apply for the situation in lines 92–98, and this completes the description of the management for p^{TL}. p^{BR} can be handled in a similar way.

c^T, c^R, and ℓ^{TR} must be updated whenever the length of $C^{TR,i}$ increases. These situations are indicated in lines 40, 46, 69, and 95 in Fig. 6, and here we only sketch how to manage them. By arguments analogous to the ones given in the proof of Lemma 3.4, we have to distinguish two cases when updating c^T. If $s^{TL,i} > e^{T,i}$, then c^T is set to p^{TL}, otherwise c^T can be determined by some additional statements which are similar to the ones used for updating p^{TL}. In either case, we set ℓ^{TR} to $e^{T,i}$ because $e^{T,i}$ is the position of c^T in $C^{TR,i}$, as seen in the proof of Lemma 3.5. The management of c^B, c^L, and ℓ^{BL} is similar.

Now let us review the construction of the final decomposition given in the end of Sect. 3. If p is set to $e^{T,\lceil m/2\rceil} + e^{R,\lceil m/2\rceil}$, then we can use c^T and c^R as the appropriate matches for c^{TL} and c^{BR}. Similarly, if $p = e^{B,\lceil m/2\rceil} + e^{L,\lceil m/2\rceil}$, we establish $c^{TL} = c^L$ and $c^{BR} = c^B$. Finally, if a longest chain is determined by the algorithm described in case d of the construction (corresponding to lines 103–112 in Fig. 6), and p is not set to one of the above values, then we can use the backup values $\tilde{u}$ and $\tilde{v}$ to determine $c^{TL} := (BottomPos[y_{\hat{u}}, b], y_{\hat{u}})$ and $c^{BR} := (TopPos[y_{\hat{v}}, t], y_{\hat{v}})$, where $\hat{u} := ThreshT[\tilde{u}]$ and $\hat{v} := ThreshB[\tilde{v}]$.

Before recursively calling the algorithm for the remaining parts of the LCS, we see it is necessary for our routine to not only work on the complete matrix of size $[1 : m] \times [1 : n]$, but also on any subarea $[k_1 : k_2] \times [\ell_1 : \ell_2]$. The necessary changes are quite straightforward, and we do not provide any details here. Moreover, it might be impossible to locate both c^{TL} and c^{BR} (e.g., when $|M| = 1$), but then one recursive call can simply be skipped.

Theorem 5.1 *An LCS can be constructed in $O(ns + \min\{mp, m \log m + p(n-p)\})$ time and $O(ns)$ space.*

Proof. Clearly, for the top–level call, the additional overhead needed to keep track of the new variables is bounded by $O(m)$. Thus, not taking into account the time consumed by preprocessing or any recursive calls, we can assume the number of elementary operations to be bounded by $d(m+\min\{mp, p(n-p)\})$, for some appropriate constant d. We first examine the bound $d(m+mp)$. Let $c^{TL} = (k, \ell)$ and $c^{BR} = (k', \ell')$ (if only one match has been determined, the analysis is similar). Consider the two first–level recursive calls concerning the areas $M_1 := [1 : k-1] \times [1 : \ell - 1]$ and $M_2 := [k'+1 : m] \times [\ell'+1 : n]$. Let p_1 and p_2 denote the length of an LCS in M_1 and M_2, respectively, i.e., $p_1 + p_2 = p - 2$. Recall that c^{TL} is located in the first $\lceil m/2 \rceil$ rows and columns, i.e., the length of one side of M_1 is bounded by $\lceil m/2 \rceil - 1$. The same is true for M_2, and thus the number of operations taken for both first–level calls is bounded by

$$d(\lceil m/2 \rceil - 1)(p_1 + 1) + d(\lceil m/2 \rceil - 1)(p_2 + 1) \leq dp\frac{m}{2}$$

Repeating this argument, we obtain a $dmp/2^i$ bound for the at most 2^i ith–level recursive calls. Since recursion ends at level $\lceil \log(m/2) \rceil$, this sums up to at most $2 \cdot dmp$ for the complete algorithm.

For the other bound $d(m + p(n-p))$, let $g := (\sqrt{5} - 1)/2 \approx 0.618$ and consider the following two cases.

Case a: $p \leq gm$. Then

$$2 \cdot dmp \leq \frac{2}{1-g} d(1-g)mp = \frac{2}{1-g} d(m - gm)p \leq \frac{2}{1-g} d(m-p)p \leq \frac{2}{1-g} d(n-p)p$$

Case b: $p > gm$. Let $h := \max\{k-1, \ell-1\}$ and $h' := \max\{m-k', n-\ell'\}$. Clearly $h + h' \leq n - 2$. Also note that $p_1, p_2 \leq \lceil m/2 \rceil - 1$ because an LCS cannot exceed the length of any side of M_1 and M_2. But then the two first–level recursive calls use at most

$$\begin{aligned} &d(\lceil m/2 \rceil - 1 + p_1(h - p_1)) + d(\lceil m/2 \rceil - 1 + p_2(h' - p_2)) \\ &\leq d(m + p_1(h-p_1) + p_2(h'-p_2)) \leq d(m + (\lceil m/2 \rceil - 1)(h - p_1 + h' - p_2)) \\ &\leq d(m + (\lceil m/2 \rceil - 1)(n-p)) \leq d(m + \tfrac{1}{2g}p(n-p)) \end{aligned}$$

operations. Similarly, all ith–level recursive calls together use at most

$$d(m + p(n-p)/(2g)^i)$$

operations. This sums up to

$$d(m \log m + \frac{1}{1-1/(2g)}p(n-p)) = d(m \log m + \frac{2}{1-g}p(n-p)) \ .$$

Both cases imply that the algorithm takes at most $O(ns+\min\{mp, m\log m+p(n-p)\})$ time, and the worst case overhead factor can be expected to be $2/(1-g) < 5.25$. Furthermore, when comparing the divide–and–conquer routine with the algorithm which determines the length p of an LCS, we only need $O(\log m)$ additional stack space, and thus the $O(ns)$ space bound is still valid. □

6 Experimental Results

We compared our routine with the algorithm proposed by Rick [Ric94, Ric95] which clearly outperforms any other method when constructing longest common subsequences of intermediate lengths. Rick's algorithm is also a flexible one, being very efficient for short and long LCS as well. It uses a strategy similar to the one presented here, but only constructs antichains (or *contours*) from the top and left side of M. While this substantially simplifies the implementation and also the preprocessing phase (i.e., we only have to compute *LeftPos* and *TopPos*), there are two severe drawbacks. First, in order to recover an LCS after determining its length, the so–called *dominant matches* must be saved during the construction of the contours, and this might take $\Omega(mn)$ space. Second, the number of checks of *Thresh*–values is significantly increased when decomposing M from only two sides. For an alphabet of size 8, Table 1 shows some sample results when determining p for different settings of m, n, and p.

Table 1: Frequency of checks of *Thresh*–values

m	n	p	Rick [Ric95]	New method
500	500	100	16864	14983
500	500	200	28962	23078
500	500	300	33276	23394
500	500	400	20384	13276

m	n	p	Rick [Ric95]	New method
1500	1500	300	145129	126796
1500	1500	600	265107	216845
1500	1500	900	280026	207000
1500	1500	1200	172846	121516

The corresponding running times are presented in Table 2. Both algorithms were programmed in a straightforward way, using no special optimizations, and were tested on an Intel Pentium II at 300 MHz. It can be seen that our algorithm only takes about 70% of the time needed by Rick's method when computing the length of an LCS which is of intermediate length. For very short or very long LCS our method slightly suffers from the additional overhead during the preprocessing phase, but is still very efficient.

Finally, we checked the running times and the consumed space when generating an LCS. Table 3 shows that in spite of the linear space restriction, our algorithm

Table 2: Running times in microseconds for determining the length p of an LCS.

m	n	p	Rick [Ric95]	New method
500	500	100	3352	3626
500	500	200	5659	4725
500	500	300	6978	4890
500	500	400	5000	3516

m	n	p	Rick [Ric95]	New method
1500	1500	300	24451	21868
1500	1500	600	46099	34835
1500	1500	900	54176	33791
1500	1500	1200	38791	22308

sometimes runs more than twice as fast as Rick's method. This is due to the significant overhead in Rick's routine which is caused by the additional statements responsible for saving the contours in memory. Furthermore, the worst case factor 5.25 calculated in the proof of Thm. 5.1 is much too pessimistic in practical situations. Instead, a comparison with Table 2 shows that it roughly equals 2.

Table 3: Running times in microseconds for constructing an LCS of length p.

m	n	p	Rick [Ric95]	New method
500	500	100	6319	6044
500	500	200	14341	9066
500	500	300	19505	9890
500	500	400	15769	7802

m	n	p	Rick [Ric95]	New method
750	750	250	23132	16374
750	750	400	39835	20495
750	750	550	38516	16758
750	750	700	16319	9945

Table 4: Allocated space in bytes for constructing an LCS of length p.

m	n	p	Rick [Ric95]	New method
500	500	100	64284	34072
500	500	200	143820	34072
500	500	300	199464	34072
500	500	400	176328	34072

m	n	p	Rick [Ric95]	New method
750	750	250	219244	51072
750	750	400	390172	51072
750	750	550	396136	51072
750	750	700	193780	51072

Conclusions

We have investigated a new algorithm for the Longest Common Subsequence Problem. In spite of the quite complicated technical details necessary for the construction and analysis, the final routines proved to be extremely practical. More precisely, we have shown three results. First, we have presented a new fast method for determining the length of an LCS. Second, we have developed a linear space algorithm for constructing an LCS in $O(ns + \min\{mp, m\log m + p(n-p)\})$ time, thus solving a previously open problem. And third, we have shown by some experimental results that this algorithm is by far the fastest one when dealing with usual applications.

Acknowledgement. We would like to thank Dr. F. Kurth for helpful comments.

References

[AHU76] Aho, A.V., Hirschberg, D.S., Ullman, J.D.: Bounds on the complexity of the longest common subsequence problem. J. ACM **23**(1), 1976, 1–12.

[Apo86] Apostolico, A.: Improving the worst–case performance of the Hunt–Szymanski strategy for the longest common subsequence of two strings. Inform. Process. Lett. **23**, 1986, 63–69.

[Apo87] Apostolico, A.: Remarks on the Hsu–Du new algorithm for the longest common subsequence problem. Inform. Process. Lett. **25**, 1987, 235–236.

[AG87] Apostolico, A., Guerra, C.: The longest common subsequence problem revisited. Algorithmica **2**, 1987, 315–336.

[ABG92] Apostolico, A., Browne, S., Guerra, C.: Fast linear–space computations of longest common subsequences. Theoret. Comput. Sci. **92**, 1992, 3–17.

[CP94] Chin, F.Y.L., Poon, C.K.: A fast algorithm for computing longest common subsequences of small alphabet size. J. Inform. Process. **13**(4), 1990, 463–469.

[CS75] Chvátal, V., Sankoff, D.: Longest common subsequences of two random strings. J. Appl. Prob. **12**, 1975, 306–315.

[DP94] Dančík, V., Paterson, M.: Upper bounds for the expected length of a longest common subsequence of two binary sequences. Proceedings, 11th Annual Symp. on Theoretical Aspects of Computer Science, LNCS **775**, 1994, 669–678.

[Day65] Dayhoff, M.O.: Computer aids to protein sequence determination. J. Theoret. Biol. **8**, 1965, 97–112.

[Day69] Dayhoff, M.O.: Computer analysis of protein evolution. Sci. Amer. **221**(1), 1969, 86–95.

[Dek79] Deken, J.G.: Some limit results for longest common subsequences. Discr. Math. **26**, 1979, 17–31.

[Dil50] Dilworth, R.P.: A decomposition theorem for partially ordered sets. Annals Math. **51**, 1950, 161–166.

[FB73] Fu, K.S., Bhargava, B.K.: Tree systems for syntactic pattern recognition. IEEE Trans. Comput. **C–22**(12), 1973, 1087–1099.

[GMS80] Gallant, J., Maier, D., Storer, J.A.: On finding minimal length superstrings. J. Comput. System Sci. **20**, 1980, 50–58.

[Hir75] Hirschberg, D.S.: A linear space algorithm for computing maximal common subsequences. Comm. ACM **18**(6), 1975, 341–343.

[Hir77] Hirschberg, D.S.: Algorithms for the longest common subsequence problem. J. ACM **24**(4), 1977, 664–675.

[HD84] Hsu, W.J., Du, M.W.: New algorithms for the LCS problem. J. Comput. System Sci. **29**, 1984, 133–152.

[HS77] Hunt, J.W., Szymanski, T.G.: A fast algorithm for computing longest common subsequences. Comm. ACM **20**(5), 1977, 350–353.

[KR87] Kumar, S.K., Rangan, C.P.: A linear space algorithm for the LCS problem. Acta Inform. **24**, 1987, 353–363.

[LW75] Lowrance, R., Wagner, R.A.: An extension of the string–to–string correction problem. J. ACM **22**(2), 1975, 177–183.

[LF78] Lu, S.Y., Fu, K.S.: A sentence–to–sentence clustering procedure for pattern analysis. IEEE Trans. Syst. Man. Cybernet. **SMC–8**(5), 1978, 381–389.

[Mai78] Maier, D.: The complexity of some problem on subsequences and supersequences. J. ACM **25**(2), 1978, 322–336.

[MP80] Masek, W.J., Paterson, M.S.: A faster algorithm for computing string edit distances. J. Comput. System Sci. **20**(1) 1980, 18–31.

[Mye86] Myers, E.W.: An $O(ND)$ difference algorithm and its variations. Algorithmica **1** 1986, 251–266.

[NKY82] Nakatsu, N., Kambayashi, Y., Yajima, S.: A longest common subsequence algorithm suitable for similar text strings. Acta Inform. **18**, 1982, 171–179.

[NW70] Needleman, S.B., Wunsch, C.S.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Molecular Biol. **48**, 1970, 443–453.

[PD94] Paterson, M., Dančík, V.: Longest common subsequences. Proceedings, 19th Intern. Symp. on Mathematical Foundations of Computer Science, LNCS **841**, 1994, 127–142.

[Ric94] Rick, C.: New algorithms for the longest common subsequence problem. Research report no. 85123–CS, Department of Computer Science, University of Bonn, Germany, 1994.

[Ric95] Rick, C.: A new flexible algorithm for the longest common subsequence problem. Proceedings, 6th Annual Symp. on Combinatorial Pattern Matching, LNCS **937**, 1995, 340–351.

[SC73] Sankoff, D., Cedergren, R.J.: A test for nucleotide sequence homology. J. Molecular Biol. **77**, 1973, 159–164.

[SK83] Sankoff, D., Kruskal, J.B.: Time Warps, String Edits, and Macromolecules: The Theory And Practice of Sequence Comparison. Addison–Wesley, Reading, MA, 1983.

[Ukk85] Ukkonen, E.: Algorithms for approximate string matching. Inform. and Control **64**, 1985, 100–118.

[Wag75] Wagner, R.A.: On the complexity of the extended string–to–string correction problem. Proceedings, 7th Ann. ACM Sympos. on Theory of Comput. 1975, 218–223.

[WF74] Wagner, R.A., Fischer, M.J.: The string–to–string correction problem. J. ACM **21**(1), 1974, 168–173.

[WC76] Wong, C.K., Chandra, A.K.: Bounds for the string editing problem. J. ACM **28**(1), 1976, 13–18.

[WMM90] Wu, S., Manber, U., Myers, G., Miller, W.: An $O(NP)$ sequence comparison algorithm. Inform. Process. Lett. **35**, 1990, 317–323.

Centroid Trees with Application to String Processing

Fei Shi and Dong-Guk Shin

Dept. Math and Computer Science
Suffolk University
Boston, MA 02114-428, USA
shi@cas.suffolk.edu
and
Dept. Computer Science and Engineering
The University of Connecticut
Storrs, CT 06269-3155, USA
shin@eng2.uconn.edu

e-mail: {goeman,clausen}@cs.uni-bonn.de

Abstract. A centroid of a tree T is a node v which minimizes over all nodes the largest connected component of T induced by removing v from T. A centroid tree U of another tree T is defined on the same set of nodes of T: the root v of U is a centroid of T and the subtrees of v (in U) are the centroid trees of the connected components of $T - v$. We describe some interesting properties of the centroid and of the centroid tree. Our linear algorithm to find a centroid of a tree improves on the previously known algorithms either in terms of space requirement or in terms of time requirement. From the algorithm for finding a centroid it is easy to obtain an $O(n \log n)$ time algorithm to construct a centroid tree of a given tree with n nodes. However, we do not know whether this is the best that one can achieve. By exploiting the properties of the centroid tree, we devise an efficient algorithm for the longest common substring problem (LCS). Given two strings S (the text) of length n and P (the pattern) of length m, the LCS problem is to find the longest substring that appears in both the text and the pattern. Our algorithm requires $O(n \log n)$ time and $O(n)$ space to preprocess the text. After preprocessing of the text, the algorithm takes $O(m \log n)$ time using $O(m)$ extra space to find the solution. The algorithm may be used in the DNA applications in which the text is very large and fixed and is to be searched with many different patterns ($n \gg m$).

Key words: balanced trees, centroid of trees, string pattern matching, the longest common substring problem

1 Introduction

Let T be an arbitrary tree and let V denote the set of nodes in T. Let $v \in V$ and let $T_1, T_2, \cdots, T_d$ be the connected components of T induced by removing v from T (denoted by $T - v$). Let $|T|$ denote the number of nodes in T. Define

$$N(v) = \max_{1 \le i \le d} \{|T_i|\}.$$

A *centroid* of the tree T [Har69] is a node v_c which minimizes $N(v)$ over all nodes v, i.e.,v_c satisfies

$$N(v_c) = \min_{v \in V}\{N(v)\}.$$

It can be shown that every tree has either one centroid or two. This fact has been extensively applied (see, for examples, [Gol71], [KH79], [FJ80], [MTZC81], [Sla82]). Goldman [Gol71] and Megiddo et al. [MTZC81] proposed linear algorithms for finding the centroid of a tree. All algorithms known to us that make use of a centroid finding algorithm call either Goldman's algorithm or Megiddo's algorithm as a subroutine.

Goldman's algorithm requires a copy of the original tree T as an auxiliary tree on which it works. Therefore, $O(n)$ extra space is needed. While Megiddo's algorithm does not need any extra space, it has to visit each node *at least* once. In this paper, we present an algorithm, which might be viewed as a combination of Goldman's algorithm and Megiddo's algorithm. Our algorithm improves on the mentioned two algorithms either in terms of space or in terms of time. Specifically, our algorithm does not need an extra copy of the original tree; at the same time, it does not need more time than Goldman's algorithm. Our algorithm visits each node of the tree at *most* once; at the same time, it does not need more space than Megiddo's algorithm. Of course, one cannot improve the complexity order of the two mentioned algorithms since both are asymptotically optimal in terms of space and time.

The notion of the centroid of a tree inspired the notion of the centroid tree. A centroid tree U of another tree T has the same set of nodes as T. U's root v is a centroid of T and v's children (in U) are the centroid trees of the connected components of $T - v$. A nice property of the centroid tree is that its height is $\log n$. It is easy to obtain an $O(n \log n)$ time algorithm to construct a centroid tree from the algorithm for finding a centroid of a tree. However, it is unknown whether this is the best time complexity that one can achieve.

By exploiting the properties of the centroid tree, we are able to give an efficient algorithm for the longest common substring (LCS) problem. Given two strings S (the text) of length n and P (the pattern) of length m, the LCS problem is to find the longest substring that appears in both the text and the pattern. An efficient solution to the problem can be useful for homology searching in nucleotide/protein sequence databases [Wat89], in the editing distance problem, in the multiple pattern searching problem, etc. Our algorithm requires $O(n \log n)$ time and $O(n)$ space to preprocess the text. After the preprocessing, a query can be answered in $O(m \log n)$ time. The algorithm is probabilistic and there is a small chance of error. That is, it may claim that a substring of the pattern is identical to a substring of the text while they are not really identical. This is called a "false match". However, the probability of a false match can be made arbitrarily (inverse-polynomially) small within the above time bounds. Our algorithm has obvious advantages over the previously known algorithms and is particularly useful for the DNA applications in which the text is very large and fixed ($n \gg m$) and in which one wishes to search the text with many different patterns (For example, the DNA sequence of a human being may have up to 3×10^9 nucleotides and a typical pattern sequence may have a few hundreds to thousands nucleotides).

The rest of the paper is organized as follows. In Section 2 we present our algorithm for finding a centroid of a tree. We address the problem of constructing a centroid tree in Section 3. In Section 4 we devise an algorithm for the LCS problem applying the results presented in Sections 2 and 3. We then conclude the paper by discussing some open problems in Section 5.

2 Finding centroid

Lemma 2.1 ([Har69]) *Every tree has either one centroid or two. In the later case, the two centroids are connected by an edge.*

If i and j are two neighboring nodes of the tree T, then by removing the edge $(i,\, j)$ two connected components $C(i,\, j)$ and $C(j,\, i)$ are induced: $C(i,\, j)$ is the component which contains node i and $C(j,\, i)$ is the component which contains node j (Note that C is defined on ordered pairs of neighboring nodes). Let u be a node of T and let $x_1, \cdots, x_d$ be all neighbors of u. Then $C(x_1,\, u), \cdots, C(x_d,\, u)$ are all the connected components of $T - u$. In the following we sometimes simply use $C(i,\, j)$ to refer to $|C(i,\, j)|$ (i.e., the number of nodes in $C(i,\, j)$) when no ambiguity would likely occur.

The following lemma is crucial for our algorithm to find a centroid of a tree correctly.

Lemma 2.2 *A node v is a centroid of the tree T if and only if*

$$N(v) \leq n/2.$$

Proof We first prove the necessary condition of the lemma. Let v be a centroid of the tree T. Suppose $N(v) > n/2$. Let $x_1, \cdots, x_d$ be all neighboring nodes of v. Then by the definition of a centroid, there must exist a neighboring node, say x_{i_0}, of v such that $C(x_{i_0},\, v) > n/2$. Let $y_1, \cdots, y_{k-1}, y_k = v$ be all neighboring nodes of x_{i_0}. Then,

$$N(x_{i_0}) = \max\{C(y_1,\, x_{i_0}),\, C(y_2,\, x_{i_0}),\, \cdots,\, C(y_{k-1},\, x_{i_0}),\, C(v,\, x_{i_0})\} \tag{13}$$

We then have

$$\begin{array}{ll} N(x_{i_0}) & = C(v,\, x_{i_0}) \text{ if } C(v,\, x_{i_0}) \geq C(y_j,\, x_{i_0})(j = 1, \ldots, k-1) \\ N(x_{i_0}) & < C(x_{i_0},\, v) \text{ otherwise .} \end{array} \tag{14}$$

Since $C(x_{i_0},\, v) > n/2$, $C(v,\, x_{i_0}) < n/2$. It then follows that

$$N(x_{i_0}) < C(x_{i_0},\, v) = N(v).$$

So by the definition of a centroid of a tree, v cannot be a centroid of the tree T. This contradicts the assumption that v is a centroid of the tree T and therefore establishes the necessary condition of the lemma.

We now turn to prove the sufficient condition of the lemma. Suppose v is a node of the tree T satisfying $N(v) \leq n/2$. Let u be a centroid of T. If $u = v$, the sufficient condition is proved. We thus consider the case in which $u \neq v$. Let $x_1, \cdots, x_d$ be all neighboring nodes of u. v must be in one of the connected components of $T - u$, say $C(x_{i_0},\, u)$. Let $y_1, \cdots, y_k = v$ be all neighboring nodes of v. Let $y_{j_0} \neq v$ be the neighboring node of v on the path from x_{i_0} to v. From $N(v) \leq n/2$, we know that $C(y_{j_0},\, v) \leq n/2$, and then $C(v,\, y_{j_0}) \geq n/2$. Because $C(v,\, y_{j_0})$ is a subtree of the component $C(x_{i_0},\, u)$, we know that $C(x_{i_0},\, u) \geq n/2$. Thus, $N(u) \geq n/2$. Thus,

$N(u) \geq N(v)$. Therefore, since u is a centroid of T, v must also be a centroid of T and $N(u) = N(v)$. This completes the proof of the sufficient condition of the lemma. □

Lemma 2.2 says a node v is a centroid of T if no connected component induced by removing v from T contains more than $n/2$ nodes.

We now describe the algorithm. Without loss of generality, we let the tree T be rooted at an arbitrary node r. We denote by $K(i)$ the number of nodes in the subtree rooted at i. Then it is easy to see the following:

1. $K(i) = 1$ if i is a leaf, and
2. $K(i) = \sum_{\text{c: child of i}} K(c) + 1$ if i is not a leaf.

The algorithm computes $K(i)$s by proceeding from the leaves of the tree towards the root. One may start from any leaf. But by rule, one is only allowed to use rules (1) and (2) to compute $K(i)$s (This is called the bottom-up manner).

The algorithm

Compute the $K(i)$s in the above defined bottom-up manner until a node v is reached such that $K(v) \geq n/2$. Node v is a centroid of T. If $K(v) = n/2$, v's father is another centroid of T.

The cost

We assume that the representation of the tree allows us to access each leaf of the tree in constant time and any node can be reached from any of it's children in constant time. We note that it is easy to build a linked representation of the tree that will have these desired properties in linear time and space. Then in the worst case, the algorithm needs to visit each node of the tree just once. *The worst case occurs only when the sole centroid of the tree is also the root of the tree.*

We could use, for instance, the most common *left-child, right-brother* representation of a tree. In this representation, each node x of the tree contains three pointers: 1. *parent*$[x]$ points to the parent of node x, 2. *left-child*$[x]$ points to the leftmost child of x, and 3. *right-brother*$[x]$ points to the brother of x immediately to the right. Under this representation, the algorithm will enter each node x at most twice: 1. either from its father or from its left-brother, and 2. (when x is a nonterminal node) from one of its children. So if the *left-child, right-brother* representation of a tree is used, the algorithm needs at most $2n - f$ node visits where f denotes the number of leaves of the tree. Note that this implementation of the algorithm does not make use of the assumption that at any point one can access the leaves of the tree in constant time. This is why this implementation may visit some nodes of the tree more than once (but at most twice). If we augment the *left-child, right-brother* representation of a tree with an array of pointers each pointing to a leaf node of the tree, the above algorithms only needs to visit each node of the tree *at most* once.

Megiddo's algorithm needs first to traverse the tree to compute some function whose definition is similar to that of $K(i)$ for each node i, then looks for the centroid along a "right" path of the tree. That is, it need at least $2n - f$ steps if the *left-child, right-brother* representation of the tree is used. While the idea of Goldman's algorithm is similar to that of ours, Goldman's algorithm requires an extra copy of the tree to work on. It deletes in some way the nodes of the extra tree until there is only one node left; this remaining node is then a centroid of the tree (see [KH79] for another version of Goldman's algorithm).

The correctness of the algorithm

If v is the first node we encountered in the course of computing the $k(i)$'s in the bottom-up manner such that $k(v) \geq n/2$, then $N(v) \leq n/2$. The correctness of the algorithm then follows from Lemma 2.2.

3 Centroid trees

Definition 3.1 (centroid tree) *A centroid tree U of another tree T is defined on the same set of nodes of T: the root v of U is a centroid of T, and the subtrees of v (in U) are the centroid trees of the connected components of $T - v$ and v (in U) is connected to the roots of these (sub-)centroid trees.*

We sometimes use $U(T)$ to denote a centroid tree of another tree T. Note that $U(T) \neq T$ in general but $U(U(T)) = U(T)$. So different trees may have the same centroid tree. Lemma 3.2 shows a nice property of the centroid tree, which motivated our work of searching for efficient methods for constructing centroid trees.

Lemma 3.2 *For any tree T with n nodes, the height of its centroid tree U is $O(\log n)$.*

Proof Each node except the leaves in U has at least two children; by Lemma 2.2 the number of nodes in any branch at any node v in U is no more than half the number of nodes in the subtree rooted at v in T. So the height of U cannot exceed the height of a complete binary tree with the same number of nodes, which is $\lfloor \log_2 n \rfloor$. □

A straightforward approach to the construction of a centroid tree is to repeatedly call the centroid finding algorithm discussed in the previous section. This approach requires $O(n \log n)$ time. There are many ways to speed up this approach. However, it is not clear whether it is possible to asymptotically improve the time complexity of this naive approach. Let's call this simple approach Algorithm *Naive*.

The following simple observations may help us to gain more insight into the centroid tree construction problem.

Lemma 3.3 *Let u be any node of the tree T. If the sizes of all connected components of $T - u$ are less than or equal to $n/2$, then u is a centroid of T. Otherwise, the centroid of T must be in the maximal component of $T - u$.*

Proof The correctness follows from Lemma 2.2. □

Lemma 3.4 *If a centroid v of the tree T is in a subtree S of T, then v must lie on the path $s, \cdots, u$ or lie on the path $s, \cdots, u, u'$ where s denotes the root of S, u is a centroid of S and u' is a child of u (with respect to the root s). In the latter case, both u and u' are the centroids of T.*

Proof Let v be a centroid of T. Suppose that v is not on the path $s, \cdots, u$. Then there are two cases to consider.

In the first case, v's father f $(v \neq f)$ is on a path $f', \cdots, f, v$ such that f' is on the path $s, \cdots, u$ (it is possible that $f = f'$). Since v is a centroid of T, $|C(f, v)| \leq n/2$. Thus $C(v, f)$ (the subtree rooted at v) contains at least $n/2$ nodes. Then the connected component of S that consists of $C(v, f)$ and the path $s, \cdots, f$ contains at least $n/2 + 1$ nodes. Therefore, by Lemma 2.2, u cannot be a centroid of the subtree S, which leads to a contradiction.

In the second case, v is a descendant of u' and $v \neq u'$ where u' is a child of u (it is possible that u'=u). Since v is a centroid of T, $C(v, u')$ (the subtree rooted at v) has at least $n/2$ nodes. We need to consider two subcases: **a.** $C(v, u')$ has exactly $n/2$ nodes. Then by Lemma 2.2, u' is another centroid of T. It is easy to see that $u = u'$. Otherwise, the subtree rooted at u' contains at least $n/2+1$ nodes and therefore, u is not a centroid of the subtree S, which is a contradiction. **b.** $C(v, u')$ has more than $n/2$ nodes. This means a branch of u that contains u' has more than $n/2 + 1$ nodes. Thus, u cannot be a centroid of the subtree S, which is also a contradiction.

This completes the proof of Lemma 3.4. □

Lemma 3.5 *Let s_1 and s_2 be any two neighboring nodes of the tree T with $|C(s_1, s_2)| = n_1$, $|C(s_2, s_1)| = n_2$ and $n_2 > n_1$. Let u be a centroid of $C(s_2, s_1)$ and let n_3 denote the number of nodes of the subtree rooted at u of T. If the $K(i)$s of all nodes i of T are known, we need at most* $\min(n/2 - n_1, n/2 - n_3)$ *steps each of which takes constant time to find a centroid of the entire tree T.*

Proof Let v be a centroid of the tree T. By Lemma 3.4, v must lie on the path $s_2, \cdots, u$. We can check the nodes on the path one by one until we finally reach a centroid of T. The connected component of $T - v$ that contains s_2 has at most $n/2$ nodes; so if we proceed from s_2 towards u we need at most $n/2 - n_1$ steps before we reach a centroid of T. The connected component of $T - v$ that contains u has at most $n/2$ nodes; so if we proceed from u towards s_2 we need at most $n/2 - n_3$ steps. In either of these two directions, each step takes constant time because the $K(i)$s of all nodes i of T are known. □

We have modified Algorithm *Naive* by making use of Lemmas 3.3, 3.4 and 3.5. The resulting algorithm is called Algorithm *Heuristic*. We have applied Algorithm *Heuristic* to several random trees. The preliminary experimental results showed that Algorithm *Heuristic* constructed a centroid tree for a given random tree in time proportional to the number of nodes in the tree on the average. However, at we are unable to prove this behavior of Algorithm *Heuristic* rigorously.

4 Application to string processing

In this section we make use of the properties of the centroid tree to solve *the longest common substring (LCS)* problem. The problem is, given a string S (the text) of n characters and a string P (the pattern) of m characters over some finite alphabet Σ, to find the longest substring which occurs in both of the two strings. An efficient solution to the problem can be useful for homology searching in nucleotide/protein sequence databases [Wat89], in the editing distance problem, in the multiple pattern searching problem [Per93], etc. We are particularly interested in the case of the problem in which the text is given in advance and is fixed, and many queries with different patterns will be made later.

Three algorithms for the LCS problem are previously known (named algorithms *P1*, *P2*, and *P3* respectively) [Per93]. It is also possible to solve the problem by constructing a suffix tree for the concatenation of the two strings and then marking each node of the suffix tree that has leaves from both of the two strings in its subtree. Let's name this algorithm *Cat*. In the following we will propose a new algorithm for the problem. Table 1 shows the time and space bounds of the previously known algorithms compared with this new algorithm (named Algorithm *New*).

Table 1: Comparison of the LCS algorithms

Algorithm	Preprocessing		Searching time	
	space	time	worst case	average
P1	$m\,\lvert\Sigma\rvert$	$m\,\lvert\Sigma\rvert + m^2$	n	
P2	$\lvert\Sigma\rvert$	$\lvert\Sigma\rvert + m$	$m\,n$	$n \log n$
P3	$m + \lvert\Sigma\rvert$	$2\,m + \lvert\Sigma\rvert$	$m\,n$	$(1 + \frac{m}{\lvert\Sigma\rvert})\,n$
Cat			$m + n$	
New	n	$n \log n$	$m \log n$	

A weakness of Algorithm *P1* is that it requires large amounts of space and preprocessing time for large alphabets and/or patterns. Algorithm *P2* requires that the size of the pattern be no more than the size of a word of the machine on which the algorithm is executed. When the size of the underlying alphabet is quite small, e.g., $|\Sigma| = 4$ in the case of DNA applications, the average-case performance of Algorithm *P3* deteriorates to its worst-case performance. While Algorithm *Cat* runs in $O(n+m)$ time, it is not proper for applications in which the text is very large and fixed and one wishes to search the text with many different shorter patterns ($n \gg m$). This is because although the text is fixed and static for many queries, for each new query (new pattern) Algorithm *Cat* has to rebuild a suffix tree for the text and the pattern which takes as much as $O(n + m)$ time. For example, a DNA sequence of a human being may have up to 3×10^9 nucleotides and a typical pattern sequence may have a few hundreds to thousands nucleotides. In such cases, $m + n \gg m \log n$, the time needed by our new algorithm to answer a query.

The new algorithm finds the longest prefix of each of the suffixes of the pattern P in the text S. Note that P has m suffixes and therefore there are at most m longest prefixes (of the suffixes) that appear in T. The algorithm then simply choose the longest one from these prefixes found as an answer to the LCS problem. It requires $O(n \log n)$ time and $O(n)$ space to preprocess the text. After the preprocessing, a query can be answered in $O(m \log n)$ time. An advantage of this approach is that in cases where the text is large (e.g., $n > m \log n$) and static for many queries, we only have to preprocess the text once; after the text has been preprocessed, a query can be answered quickly. It is a probabilistic algorithm and there is a small chance of error. That is, the algorithm may claim that a substring of the pattern is equal to a substring of the text while they are not equal at all (This is called a "false match"). However, as will be seen later, the probability of a false match can be made arbitrarily (inverse-polynomially) small.

The general structure of the algorithm is as follows:

- Preprocessing stage
  - construct a suffix tree T for the text S
  - construct a centroid tree U for the suffix tree T
- Searching stage
  - search the centroid tree U for locations of the longest prefix of each of the suffixes of the pattern P in the text T

Now, we describe the algorithm in detail. Since algorithms for building suffix trees in linear time and space are known in the literature [Wei73, McC76, Ukk95] and we have already presented an algorithm for building the centroid tree (in Section 3), we will concentrate on the searching stage of the algorithm.

Let the text be $S = S[1] \cdots S[n]$ and let the pattern be $P = P[1] \cdots P[m]$. We use a suffix tree to represent the text. Assuming that the suffix tree T of the text S and a centroid tree U of T are already available, our search algorithm searches the trees for the occurrences of the pattern in the text.

Let w be the end node of the path that the pattern P determines in T. If P is not a substring of S, then we define the end node w to be the node that corresponds to the longest prefix of P that is a substring of S. Our goal is to find w.

We maintain the following variables:

- v, the current node in U; v is a centroid of some connected component C of T.
- u, the topmost node of C (in T); the substring corresponding to u is the longest substring of S found so far that is a prefix of P.
- i, an index to P such that $P[1], \cdots, P[i]$ determines the path from the root to u.
- j, the length of the substring determined by the path from u to v.
- k, a pointer to S that corresponds to the end position of the substring determined by the path from the root to v.

Furthermore, let x be any node of T. We denote by $x.length$ the length of the substring determined by the path from the root to node x and denote by $x.end$ an index to S that corresponds to the end position of this substring in S. Note that by assumption, $x.length$ and $x.end$ are already stored in each node x on construction of the suffix tree.

Given u and v computing j and k is easy:

$$\begin{aligned} j &:= v.length - u.length; \\ k &:= v.end. \end{aligned} \tag{15}$$

Initially, $u :=$ the root of T; $v :=$ the root of U; $i := 0$; and j and k are computed by (15).

In order to find w efficiently we need to find a way to decide quickly which of the connected components induced by removing v from T contains w. There are two possibilities: w is in the component that is "above" v or w is in one of the components that are "below" v. We notice that w is a descendant of v if and only if $S[k-j+1]\cdots S[k] = P[i+1]\cdots P[i+j]$. If w is in the component "above" v, we assign the centroid of that component to v and u is unchanged; if we know which of the components "below" v contains w, we assign the root of that component to u and assign the centroid of that component to v. The above ideas are precisely presented in **procedure** *search* in Figure 1. **procedure** *search* finds and stores w in its variable v and stores the index to P referring to the end position of the longest prefix of P that is equal to a substring of S in its variable i when executed with u being initialized to be the root of T, v being initialized to be the root of U and i being initialized to be 0.

The question that is crucial to implement **procedure** *search* efficiently is: Given a substring $S[k]\cdots S[k+j]$ of S and a substring $P[i]\cdots P[i+j]$ of P, how can we answer quickly whether they are equal or not? There is a probabilistic method [Nao91, KR87] which, after preprocessing the strings S and P in linear time and space, can test whether a substring of S is equal to a substring of P in constant time. There is a probability of error (a false match) in any test. But the probability of a false match can be made arbitrarily (inverse polynomially) small.

The method needs a prime q which is chosen at random from a set of primes smaller than M (to be stated soon). It is this prime q that may lead to a false match. By Theorem 3 of [KR87] the probability of a false match is less than $\pi(n)/\pi(M)$ where $\pi(n)$ denotes the number of primes smaller than n. By Lemma 2 of [KR87] $\frac{u}{\ln u} \leq \pi(u) \leq 1.25506\frac{u}{\ln u}$. Thus, for example, if we choose M to be $n^3 \log n$, the probability of a false match is (asymptotically) $1/n^2 \log n$.

We now look at the complexity of **procedure** *search*. Note that at each step v is assigned to one of its children (in U). By Lemma 3.2 the height of U is $O(\log n)$. So **procedure** *search* requires $O(\log n)$ steps. From the above discussion, each step takes constant time. So **procedure** *search* needs $O(\log n)$ time to find the longest prefix of P that appears in S.

To solve the whole LCS problem, for every suffix $P_i = P[i]\cdots P[m]$ $(i = 1, \cdots, m)$ we find the longest prefix of P_i that appears in S with **procedure** *search*. From among all these (locally) longest prefixes found, we choose the (globally) longest one as an answer to the LCS problem. All this takes $O(m \log n)$ time.

To summarize, our algorithm for the LCS problem consists of:

- Preprocessing the text

```
procedure search(node: u, v; integer: i);
integer: j, k;
begin
  j := v.length − u.length;
  k := v.end;
  if S[k − j + 1] . . . S[k] = P[i + 1] . . . P[i + j] then /* j may be 0 */
    if i + j = m then i := m; stop
    /* P is equal to the substring of S corresponding to node v */
    else
      if there exists a child c of v in T corresponding to the symbol
       P[i + j + 1] then
        if the substring S[k + 1] . . . S[k + l] of S corresponding to the edge
        (v, c) is equal to a substring of P starting at P[i + j + 1] then
          u := c;
          v := v's child in U corresponding to the subtree rooted at c in T;
          i := i + j + l;
          search(u, v, i)
        else
          /* Let L denote the maximal x in [1, l] such that
          S[k + 1] . . . S[k + x] = P[i + j + 1] . . . P[i + j + x] */
          find L with binary search supported with the substring equality
          testing technique;
          i := i + j + L; stop
          /* P[1]...P[i] is the longest prefix of P that is equal to a
          substring of S; this substring is the concatenation of the
          substring corresponding to node v and S[k + 1] . . . S[k + L] */
        end
      else
        i := i + j; stop
        /* P[1] . . . P[i] is the longest prefix of P that is equal to a
        substring of S; this substring corresponds to node v */
      end
    end
  else
    if there exists a child of u in T corresponding to P[i + 1] then
      v := v's child in U corresponding to the component "above" v;
      search(u, v, i)
    else
      v := u; stop
      /* P[1] . . . P[i] is the longest prefix of P that is equal to a
        substring of S; this substring corresponds to node v */
  end
end
```

Figure 1: Search for end node of path determined by pattern.

  - construct a suffix tree T for the text S in $O(n)$ time and space.
  - construct a centroid tree U for the suffix tree T in $O(n \log n)$ time and using $O(n)$ space.
  - process the text S in order to be able to check quickly the substring equality. This takes $O(n)$ time and space.

- Searching for the pattern
  - process the pattern P in order to be able to check quickly the substring equality. This takes $O(m)$ time and space.
  - search the centroid tree U for locations of the longest prefixes of all the suffixes of the pattern P in the text S in $O(m \log n)$ time and $O(1)$ space.

That is, the preprocessing takes $O(n \log n)$ time and $O(n)$ space and the searching takes $O(m \log n)$ time and $O(m)$ extra space.

To make this algorithm error free, we can add a step that checks whether a claimed match is *true* or *false*. If the claimed longest match is false, we discard it and check the second longest match, and so on, until we reach a *true* match. Since the probability of a false match can be made arbitrarily (inverse-polynomially) small without asymptotically increasing the time and space requirements of the algorithm, the chance of using this checking step can be made arbitrarily inverse-polynomially small as well.

5 Open questions

It is of considerable interest to either establish that there exists a non-linear lower bound on the run time of all algorithms which construct a centroid tree for any given tree, or to exhibit an algorithm whose run time is $O(n)$.

It is also interesting, at least from a practical point of view, to find centroid tree construction algorithms that run in linear time on the average and require linear space even if their worst-case behavior could be much worse. Are there any deterministic algorithms to do the search (as discussed in Section 4) using the same order of time as the probabilistic one does?

References

[FJ80] G.N. Frederichson and D.B. Johnson, Generating and searching sets induced by networks, *Proc. of the 7th International Colloquium on Automata, Languages and Programming*, LNCS 85, July 1980.

[Gol71] A.J. Goldman, Optimal center location in simple networks, *Trans. Sci.*, 3(1971).

[Har69] F. Harary, *Graph Theory*, Addison-Wesley, Mass., 1969.

[KH79] O. Kariv and S.L. Hakim, An algorithmic approach to network location problems. I: The p-centers, *SIAM J. appl. Math.*, Vol 37, No.3, Dec. 1979.

[Knu73] D.E. Knuth, *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*. Addison-Wesley, Reading, Mass, 1973, Ch. 6.3, pp. 490-493.

[KR87] R. Karp and M. Rabin, Efficient Randomized Pattern Matching Algorithms. *IBM J. Res. Develop.*, Vol. 31, No. 2, March 1987.

[McC76] E.M. McCreight, A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM* 23 (1976), 262-272.

[MTZC81] N. Megiddo, A. Tamir, E. Zemel and R. Chandrasekaran, An $O(n \log^2 n)$ algorithm for the kth longest path in a tree with applications to location problems, *SIAM J. Comput.*, Vol.10, No.2, May 1981.

[Nao91] M. Naor, String matching with preprocessing of text and pattern, *Proc. of the 18th International Colloquium on Automata, Languages and Programming*, Madrid, July 1991, pp.739-750.

[Per93] C.H. Perleberg, Three Longest Substring Algorithms, *Proc. First South American Workshop on Strong Processing*, Belo Horizonte, Brazil, 1993, eds. R. Baeza-Yates and N. Ziviani.

[Sla82] P.J. Slater, Locating Central Paths in a graph,*Trans. Sci.*, Vol.16, No.1, Feb. 1982.

[Ukk95] E. Ukkonen, On-line construction of suffix-trees. *Algorithmica* (1995) 14: 249-260, Springer-Verlag, New York.

[Wat89] M.S. Waterman (ed.), *Mathematical Methods for DNA Sequences*, CRC Press 1989, Boca Raton, Florida.

[Wei73] P. Weiner, Linear pattern matching algorithm, *Proc. 14th IEEE Symp. on Switching and Automata Theory*, 1973, pp. 1-11.