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 **f** = {f,d, g, r, t, c, v}. The sequence of sets **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 $A = a_1 a_2 \ldots a_m, a_i \in \Omega, 1 \leq i \leq m$ be a string over an alphabet Ω, where $|A| = m$ is the length of the string A.

Let $\mu_A(a_i) \in (0, 1\rangle, 1 \leq i \leq m$, 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 $C \in P(\Omega), C = c_1 \ldots c_p$ is *a subsequence* of the string $A = a_1 \ldots a_m$, if a monotonous increasing sequence of natural numbers $i_1 < \ldots < i_p$ 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, $MCCS((A, \mu_A), (B, \mu_B)) = Val(CCS((A, \mu_A), (B, \mu_B)))$. $\diamond$

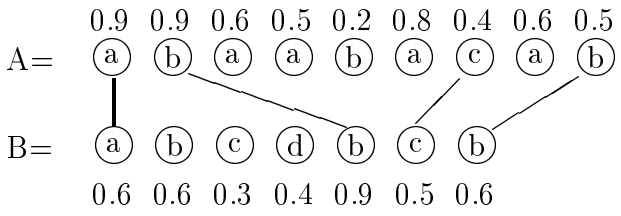

Figure 1. The closest common subsequence of two cf-strings A and B.

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, $\mathrm{h^A} = h_0^A h_1^A h_2^A \ldots h_{k^A}^A, 0 = h_0^A < h_1^A < h_2^A < \ldots < h_{k^A}^A = m, 1 \leq k^A \leq m$ is *a partition of the string* (A, μ_A).

The sequence $\mathrm{h^A}$ divides the string (A, μ_A) in the following way:
$A = |a_1 a_2 \ldots a_{h_1^A}|a_{h_1^A+1} \ldots a_{h_2^A}| \ldots |a_{h_{k-1}^A+1} \ldots a_{h_{kA}^A}| = subst_1^A subst_2^A \ldots subst_{k^A}^A$, where $subst_i^A = a_{h_{i-1}^A+1} \ldots a_{h_i^A}, 1 \leq i \leq k^A$. $[(A, \mu_A), h^A]$ 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 $C = c_1 c_2 \ldots c_p, 1 \leq p \leq m$ is *a restricted subsequence* of the cf-string with the partition $[(A, \mu_A), \mathrm{h^A}]$, if and only if

1. there exists a sequence of indices $1 \leq i_1 < i_2 < \ldots < i_p \leq m$ 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 $[(A, \mu_A), \mathrm{h^A}]$ and $[(B, \mu_B), \mathrm{h^B}]$ if (C, μ_C) is a restricted cf-subsequence of $[(A, \mu_A), \mathrm{h^A}]$ and (C, μ_C) is a restricted cf-subsequence of $[(B, \mu_B), \mathrm{h^B}]$ at once.

The string (C, μ_C) is *a closest common restricted cf-subsequence* of two cf-strings with partition $[(A, \mu_A), \mathrm{h^A}]$ and $[(B, \mu_B), \mathrm{h^B}]$ 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, $MCCRS([(A, \mu_A), h^A], [(B, \mu_B), h^B]) = Val(CCRS([(A, \mu_A), h^A], [(B, \mu_B), h^B]))$. ◇

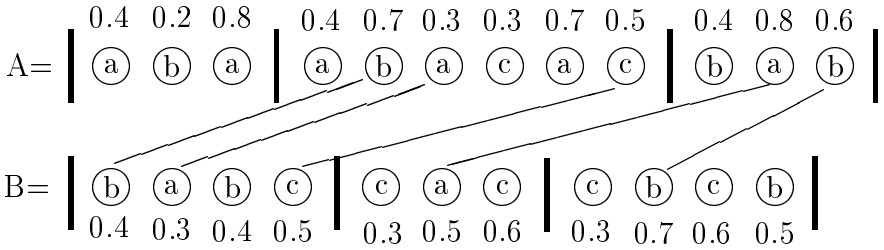

Figure 2. Closest common restricted subsequence of two strings A and B.

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

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

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 $MSCCS((A, \mu_A), (\mathcal{B}, \mu_{\mathcal{B}})) = Val(SCCS((A, \mu_A), (\mathcal{B}, \mu_{\mathcal{B}})))$, $\diamond$

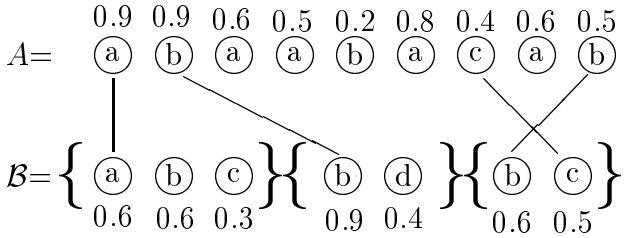

Figure 3. The closest common subsequence of two strings A and $\mathcal{B}$.

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 $\mathcal{A} = A^1 \ldots A^m, \mathcal{B} = B^1 \ldots B^n, 1 \leq m \leq n$, 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, $MSSCCS((\mathcal{A},\mu_{\mathcal{A}}),(\mathcal{B},\mu_{\mathcal{B}})) = Val(SSCCS((\mathcal{A},\mu_{\mathcal{A}}),(\mathcal{B},\mu_{\mathcal{B}})))$, $\diamond$

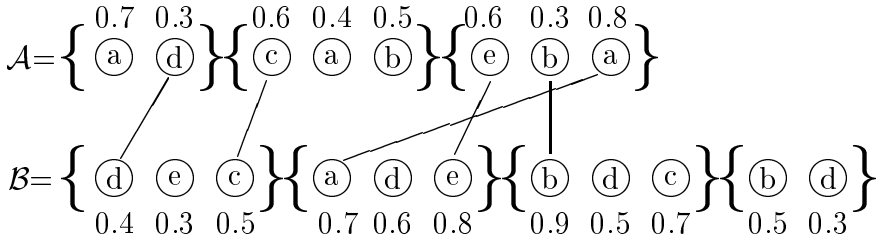

Figure 4. The closest common subsequence of two set-strings $\mathcal{A}$ and $\mathcal{B}$.

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:

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

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

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

It means

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

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

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

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

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

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 $1 \leq i \leq k \leq m$,
- $MM[m,n] = MCCS((A,\mu_A),(B,\mu_B))$,
- $MM[i,j] = MCCS((A[1..i],\mu_A),(B[1..j],\mu_B))$.

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

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

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 $m \leq n$. 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	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											
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.

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

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

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

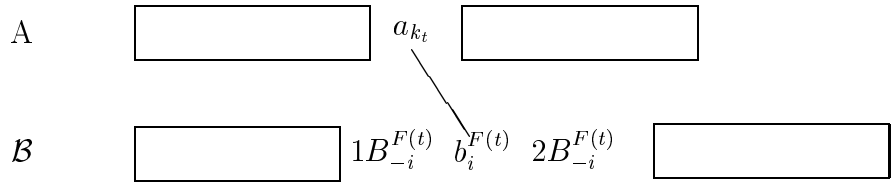

Figure 5. The idea for the construction of algorithm

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

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

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:

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

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.