Problems regarding fuzzy string processing

by

Arevik Harmandayan

Bachelor of Science, Yerevan State University, 2016

A thesis submitted in partial satisfaction of

the requirements for the degree of

Master of Science

in

Computer & Information Science

in the

COLLEGE OF SCIENCE AND ENGINEERING

of the

AMERICAN UNIVERSITY OF ARMENIA

Supervisor: ___________________________

Signature: ___________________ Date:_____________

Committee Member: _____________________

Signature: ___________________ Date:_____________

Committee Member: _____________________

Signature: ___________________ Date:_____________

Committee Member: _____________________

Signature: ___________________ Date:_____________

Abstract

This thesis presents the solutions to the fuzzified dotted string matching and fuzzified string distance problems. These algorithms are modifications of known string matching and string distance algorithms.

For each of the problems two cases are presented. For the fuzzified dotted string matching problem the cases are matching a string with a fuzzy pattern and fuzzy matching of a string with a pattern. For the fuzzified string distance problem the cases are distance between a string and a fuzzy pattern and distance with fuzzy matching.

Licenses for Software and Content

Software Copyright License (to be distributed with software developed for masters project)

Content Copyright License (to be included with Technical Report)

Content

Introduction

The algorithms presented in this thesis use the dynamic programming approach.
The algorithms presented in this thesis can be viewed as approximate string matching algorithms. Even though well known approximate string matching algorithms are sometimes called fuzzy string matching algorithms, they work with exact strings and exact matching. The algorithms presented in this thesis work with fuzzy patterns (patterns consisting of values of linguistic variables) and fuzzy matching.
Some of the common applications of these kind of algorithms are text searching, text editing, spell checking, context search, querying specific rows from a database, data compression, spam filtering and others.
In recent years the importance and usability of algorithms like these grew in biology (matching of nucleotide sequences).

Chapter 1
Known algorithms

In this chapter we will discuss already well known exact string matching algorithms. Some of the fuzzy string matching and fuzzy string distance algorithms that will be presented in later chapters are modifications of these exact algorithms.

1.1 Longest common subsequence [1]

Given two sequences X and Y , we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y . For example, if X = ⟨A, B, C, B, D, A, B⟩ and Y = ⟨B, D, C, A, B, A⟩, the sequence ⟨B, C, A⟩ is a common subsequence of both X and Y . The sequence ⟨B, C, A⟩ is not a longest common subsequence (LCS) of X and Y , however, since it has length 3 and the sequence ⟨B, C, B, A⟩, which is also common to both X and Y , has length 4. The sequence ⟨B, C, B, A⟩ is an LCS of X and Y , as is the sequence ⟨B, D, A, B⟩, since X and Y have no common subsequence of length 5 or greater.
In the *longest-common-subsequence* problem, we are given two sequences $X=\langle x_1,x_2,...,x_m\rangle$ and $Y=\langle y_1,y_2,...,y_n\rangle$ and wish to find a maximum length common subsequence of X and Y.

Characterizing a longest common subsequence

The LCS problem has an optimal-substructure property, however, as the following theorem shows. As we shall see, the natural classes of subproblems correspond to pairs of “prefixes” of the two input sequences. To be precise, given a sequence $X=\langle x_1,x_2,...,x_m\rangle$, we define the ith *prefix* of X, for i = 0,1,...,m, as $X_i=\langle x_1,x_2,...,x_i\rangle$. For example, if X = ⟨A, B, C, B, D, A, B⟩ , then X_4 = ⟨A, B, C, B⟩ and X_0 is the empty sequence.

Theorem (Optimal substructure of an LCS)

Let $X=\langle x_1,x_2,...,x_m\rangle$ and $Y=\langle y_1,y_2,...,y_n\rangle$ be sequences, and let $Z=\langle z_1,z_2,...,z_k\rangle$ be any LCS of X and Y .

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

The way that Theorem characterizes longest common subsequences tells us that an LCS of two sequences contains within it an LCS of prefixes of the two sequences. Thus, the LCS problem has an optimal-substructure property. A recursive solution also has the overlapping-subproblems property.

A recursive solution

To find an LCS of X and Y , we may need to find the LCSs of X and Y_{n-1} and of X_{m-1} and Y . But each of these subproblems has the subsubproblem of finding an LCS of X_{m-1} and Y_{n-1}. Many other subproblems share subsubproblems.

As in the matrix-chain multiplication problem, our recursive solution to the LCS problem involves establishing a recurrence for the value of an optimal solution. Let us define c[i,j], j to be the length of an LCS of the sequences X_i and Y_j . If either i = 0 or j = 0, one of the sequences has length 0, and so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i = y_j \\ \max(c[i,j-1],c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i \neq y_j \end{cases} \qquad (1)$$

Computing the length of an LCS

Based on equation (1), we could easily write an exponential-time recursive algorithm to compute the length of an LCS of two sequences. Since the LCS problem has only $\mathbf{\Theta}(mn)$ distinct subproblems, however, we can use dynamic programming to compute the solutions bottom up.

Procedure LCS-LENGTH takes two sequences $X=\langle x_1,x_2,...,x_m\rangle$ and $Y=\langle y_1,y_2,...,y_n\rangle$ as inputs. It stores the c[i,j] values in a table c[0..m,0..n], and it computes the entries in *row-major* order. (That is, the procedure fills in the first row of c from left to right, then the second row, and so on.) The procedure also maintains the table b[1..m,1..n] to help us construct an optimal solution. Intuitively, b[i,j] points to the table entry corresponding to the optimal subproblem solution chosen when computing c[i,j] . The procedure returns the b and c tables; c[m,n] contains the length of an LCS of X and Y .

```
LCS-LENGTH(X, Y)
m=X.length
n=Y.length
let b[1..m,1..n] and c[0..m,0..n] be new tables
for i = 1 to m
     c[i,0] = 0
for j = 0 to n
     c[0,j] = 0
for i = 1 to m
     for j = 1 to n
         if $x_i$ == $y_j$
              c[i,j] = c[i-1,j-1] + 1
              b[i,j] = "↗"
         elseif c[i-1,j] $\geq$ c[i,j-1]
              c[i,j] = c[i-1,j]
```

```
            b[i,j] = "↑"
        else c[i,j] = c[i,j-1]
            b[i,j] = "←"
return c and b
```

Constructing an LCS

The b table returned by LCS-LENGTH enables us to quickly construct an LCS of $X=\langle x_1,x_2,...,x_m\rangle$ and $Y=\langle y_1,y_2,...,y_n\rangle$. We simply begin at b[m,n] and trace through the table by following the arrows. Whenever we encounter a "-" in entry b[i,j], it implies that $x_i = y_j$ is an element of the LCS that LCS-LENGTH found. With this method, we encounter the elements of this LCS in reverse order. The following recursive procedure prints out an LCS of X and Y in the proper, forward order. The initial call is PRINT-LCS(b, X, X.length, Y.length).

```
PRINT-LCS(b,X,i,j)
if i == 0 or j == 0
    return
if b[i,j] == "↗"
    PRINT-LCS(b,X,i-1,j-1)
    print $x_i$
elseif b[i,j] = "↑"
    PRINT-LCS(b,X,i-1,j)
else PRINT-LCS(b,X,i,j-1)
```

1.2 String Distance Algorithms [2]

The algorithm discussed below will apply uniformly to each of three kinds of distances: Levenshtein (d_L), edit (d_E) and weighted (d_W) distances, we will denote distance simply by d. Then we can express the properties of d in terms of edit operations performed on single (nonemply) letters λ and μ:

- (insert: replace ε by λ) $d(\varepsilon,\lambda) > 0$;
- (delete: replace λ by ε) $d(\lambda,\varepsilon) >0$;
- (substitute: replace λ by μ) $d(\lambda,\mu) > 0$ iff $\lambda \neq \mu$;

For $d = d_L$ or d_E, $d(\lambda,\varepsilon) = d(\varepsilon,\lambda) = 1$ for all λ; while for $d = d_L$, $d(\lambda,\varepsilon) = 2$ and for $d = d_E$, $d(\lambda,\mu) = 1$.

Observe that each single-letter distance may also be thought of as the *cost* of performing the corresponding edit operation: more generally, we think of the distance between two strings as the minimum cost of transforming one string into the other.

The algorithm presented here is a dynamic-programming algorithm, that is, it compares the input strings $x_1 = x_1[1..n_1]$ and $x_2 = x_2[1..n_2]$ from left to right, computing for each pair of positions i and j the minimum cost of transforming $x_1[1..i]$ to $x_2 = x_2[1..j]$ based on the already-computed minimum costs

- $d(x_1[1..i], x_2[1..j-1])$
- $d(x_1[1..i-1], x_2[1..j])$
- $d(x_1[1..i-1], x_2[1..j-1])$

This basic recurrence is explained below.

It is convenient to introduce two-dimensional *cost array* $c=c[0..n_1,0..n_2]$ in which

$$c[i,j] = d(x_1[1..i], x_2[1..j])$$

for every $i \in 0..n_1$, $j \in 0..n_2$ with initial values defined as follows:

- $c[0,0] = 0$, the minimum cost of transforming the empty string into itself;

- $C[0,j] = \sum_{1 \le h \le j} d(\varepsilon,x_2[h])$, the minimum cost of inserting the first j letters of x_2 into the empty string;
- $C[i,0] = \sum_{1 \le h \le i} d(x_1[h],\varepsilon)$, the minimum cost of deleting the first i letters of x_1 into the empty string.

Lemma

For every $i \in 0..n_1$, $j \in 0..n_2$

$$c[i,j] = \min\{ c[i-1,j] + d(x_1[i],\varepsilon), c[i,j-1] + d(\varepsilon,x_2[h]), c[i-1,j-1] + d(x_1[i],x_2[h]) \}$$

The critical factor in the proof of this lemma is the inserts and deletes can be performed in any order. This property of the inserts and deletes does not depend on the Symmetry Property of the metric; that is, it held even if, for some strings x_1 and x_2,

$$d(x_1,x_2) \neq d(x_2,x_1)$$

Consequently, Lemma holds even if the Symmetry Property does not. This observation makes it clear that cost need not be quite the same thing as distance: since we have defined cost in terms of transferring x_1 into x_2, it is possible to imagine that the cost of transferring x_2 into x_1 may not be the same.

Finally, we remark that the cost array and the recurrence scheme for computing it given by Lemma are very simple and natural ideas with widespread consequences: not
only does the recurrence form the basis of distance (hence LCS) calculations, it is also fundamental to almost all approaches to the calculation of the shortest common superstring of a collection of strings, and to the "best" alignment of a collection of strings.

Chapter 2
Fuzzy set theory. The Concept of a Linguistic Variable

2.1 Fuzzy set theory

Fuzzy logic

Fuzzy logic is a form of many-valued logic in which the truth values of the variables may be any real number between 0 and 1. It is employed to handle the concept of partial truth, where the truth value may range between completely true and completely false [5]. In Boolean logic, the truth value can be only completely true or completely false: 1 or 0.

Fuzzy Sets

A fuzzy set is a set whose elements have grades of membership. It is characterized by a membership function which assigns each element a grade of membership ranging between 0 and 1. [3]

A fuzzy subset *A* of a universe of discourse *U* (a finite or infinite set) is characterized by a *membership function* $\mu_A : U \to [0, 1]$ which associates with each element *u* of *U* a number , $\mu_A(u)$ in the interval [0,1], with $\mu_A(u)$ representing the *grade of membership* of *u* in *A*. The *support* of A is the set of points in *U* at which $\mu_A(u)$ is positive. The *height* of *A* is the supremum of $\mu_A(u)$ over *U*. A *crossover point* of *A* is a point in *U* whose grade of membership in *A* is 0.5. [4]

Example: Let the universe of discourse be the interval (-∞, ∞), with *u* interpreted as *temperature*. A fuzzy subset of *U* labeled *high* may be defined by a membership function such as

$$\mu_A(u) = 0 \qquad \text{for } u \leq 20$$

$$\mu_A(u) = (x - 20)/20 \qquad \text{for } 20 \leq u \leq 40$$

$$\mu_A(u) = 1 \qquad \text{for } u > 40$$

The support of high is [20,∞), the height is 1, the crossover point is 30.

Operations on fuzzy sets [4]

When the support of a fuzzy set is a continuum rather than a countable or a finite set, we shall write:

$$A = \int_U \mu_A(u)/u$$

with the understanding that $\mu A(u)$ is the grade of membership of u in *A*, and the integral denotes the union of the fuzzy singletons $\mu_A(u)/u$, $u \in U$.

There are many operations that can be done of fuzzy sets.We are going to discuss *compliment* and *fuzzification.*

Compliment

The *complement* of *A* is denoted by ¬*A* (or sometimes by *A'*) and is
defined by

$$\neg A = \int_U [1 - \mu_A(u)]/u$$

The operation of complementation corresponds to negation. Thus, if *A* is a label for a fuzzy set, then not A would be interpreted as ¬*A*.

Fuzzification

The operation of *fuzzification* has, in general, the effect of transforming a nonfuzzy set into a fuzzy set or increasing the fuzziness of a fuzzy set. Thus, a *fuzzifier F* applied to a fuzzy subset *A* of *U* yields a fuzzy subset *F(A;K)* which is expressed by

$$F(A;K) = \int_U \mu_A(u)K(u)$$

Where the fuzzy $K(u)$ is the *kernel* of *F*, that is, the result of applying *F* to a singleton 1/*u*:

$$K(u) = F(1/u; K);$$

$\mu_A(u)K(u)$ represents the product of a scalar $\mu_A(u)$ and the fuzzy set $K(u)$; and $\int$ is the union of the family of fuzzy sets $\mu_A(u)K(u)$, $u \in U$.

The operation of fuzzification plays an important role in the definition of linguistic hedges such as *more or less*, *slightly*, *somewhat*, *much*, etc.

2.2 Linguistic Variable

Linguistic variables are variables whose values are words or sentences in a natural or artificial language.[4]

For example, *Temperature* is a linguistic variable if its values are linguistic, i.e., high, very high, low, not low, extremely high, not very low, etc., rather than numeric, 40, 46, -10, 7, 78, 2, etc.

Linguistic variable is characterized by a quintuple (*L*, *T*(*L*), *U*, *G*, *M*) in which *L* is the name of the variable; *T*(*L*) is the term-set of *L* , that is, the collection of its linguistic values; *U* is a universe of discourse; *G* is a syntactic rule which generates the terms in *T*(*L*); and *M* is a semantic rule which associates with each linguistic value X its meaning, *M*(X), where *M*(X) denotes a fuzzy subset of *U*. The meaning of a linguistic value X is characterized by a compatibility function, $c : U \rightarrow [0, 1]$ which associates with each *u* in *U* its compatibility with X. [4] So the compatibility of 28 degrees with *high* might be 0,4, while that of 34 might 0,7.

The function of the semantic rule is to relate the compatibilities of the so-called *primary* terms in a composite linguistic value, e.g., *low* and *high* in *not very low* and *very high,* to the compatibility of the composite value. To this end, the hedges such as *very*, *quite*, *extremely*, etc., as well as the connectives *and* and *or* are treated as nonlinear operators which modify the meaning of their operands in a specified fashion.[4] Those hedges are sometimes also called *modifiers*

Chapter 3
Fuzzyfied string dotted matching

3.1 Matching a string with a fuzzy pattern

Problem definition: We have a string of length n and a pattern of length m. The pattern is comprised of linguistic variable values. We need to find if the sting is dotted matched to the pattern with a degree of membership that is greater than a given threshold T.

The degree of membership of string to the pattern is the multiplication of the degrees of membership of the matched elements from the string to the pattern.

We are given a sequences $A=\langle a_1,a_2,...,a_n\rangle$ and pattern of values of linguistic variables $B=\langle b_1,b_2,...,b_m\rangle$. The number of values of this linguistic variable is k.

This problem can be view as a special case of the longest common subsequence problem.

We take a matrix matrix[n][m]. Every element of the matrix holds a reference to an array. The length of the arrays are distributed like this :

1	1	1	...	1
1	2	2	...	2
1	2	3	...	3
1	2	3	...	4
:	:	:		
1	2	3	...	m

We can’t unequivocally say if an element should be in the final matching subsequence of A or not we need to keep history of different paths we can take. This history is kept in these arrays.

The elements of the array are a structure that hold weight (the degree of membership) and direction (to help us construct an optimal solution). The direction points to the table entry corresponding to the optimal subproblem solution chosen when computing the weight. So we keep the path of each choice.

This is of how the arrays are filled:

$$m[i][j][h].weight= \begin{cases} 1 & \text{if } i=0 \text{ and } h=0,\text{or } j=0 \text{ and } h=0 \\ m[i-1][j-1][h-1].w*weight(a[i],b[j]) & \text{if } i=j=h \text{ and } i,j>0 \\ \max(m[i-1][j][h].w,m[i][j-1][h].w,m[i-1][j-1][h-1].w*weight(a[i],b[j])) & \\ & \text{if } i<=j,h<j \text{ and } i,j<0 \\ & \text{or } i>=j,h<i \text{ and } i,j<0 \\ \max(m[i-1][j][h].w,m[i-1][j-1][h-1].w*weight(a[i],b[j])) & \\ & \text{if } i>j,h>=i \text{ and } i,j<0 \\ \max(m[i][j-1][h].w,m[i-1][j-1][h-1].w*weight(a[i],b[j])) & \\ & \text{if } i<j,h>=j \text{ and } i,j<0 \end{cases}$$

If we take m[i-1][j][h], m[i][j][h].dir = ‘ ↑ ’, if we take m[i][j-1][h] m[i][j][h].dir = ‘←’, else ‘↗’.
This way the array referenced in the element matrix[n][m] holds the max weights of substrings of each length : matrix[n][m][i] holds the max value of a subarray of length i, and the end of its path. As we need to have match with every element of the pattern, we will take the matrix[n][m][m] element of the matrix,that holds the degree of membership of the best matching substring of A to the pattern B, so if the T ≤ matrix[n][m][m], we say that the string A dotted matches the fuzzy pattern B, if T > matrix[n][m][m] we say that there is no match.

Example.

We have 2 sequences. The number sequence is 2,2,4,1.The pattern is small,large,small.

		small	large	small
	1	1	1	1
2	1	1 ↖ 0,75	1 ← 0,75	1 ↖ 0,75
2	1	1 ↖ 0,75	1 ← 0,75 2 ↖ 0,1875	1 ← 0,75 2 ↖ 0,5675
4	1	1 ↑ 0,75	1 ↖ 0,75 2 ↖ 0,5675	1 ← 0,75 2 ↑ 0,5675 3 ↖ 0,046875
1	1	1 ↖ 1	1 ← 1 2 ↑ 0,5675	1 ← 1 2 ↖ 0,75 3 ↖ 0,5675

In the first line and first column we have and 1 element array. In the matrix[1][1] we have [1 ↖ 0,75] .The 1 shows that the length of the subsequence is 1. ↖ is the direction and 0.75 is the degree of membership of 2 to small. Matrix[3][2] has an array with elements {[1 ← 0,75],[2 ↖ 0,5675]}. In [1 ← 0,75],1 is the length of the subsequence, 0,75 is from max(weight(4,large), matrix[3][1][1].weight, matrix[2][2][1].weight). The ↖ shows that we took the max(weight(4,large). In [2 ↖ 0,5675]: 2 is the length of the subsequence, 0,5675 is from max(weight(4,large)*matrix[2][1][1].weight, matrix[2][2][1].weight).
The ↖ shows that we took the weight(4,large)*matrix[2][1][1].weight.
The matrix[4][3] hold the reference to the array {[1 ← 1],[2 ↖ 0,75],[3 ↖ 0,5675]} which means that the maximum degree of membership of the 3 element subsequence is 0,5675, 2 element subsequence is 0,75 and of the 1 element subsequence is 1. You can see how we get the elements of the subsequences following the highlighted arrows, the blue for 3, green for 2 and red for 1 element subsequences. We follow the arrows and take the element that have the arrow ↖.When we have taken as many elements as the length of subsequence we are looking for.

3.2: Fuzzy matching of a string with a pattern

Problem definition: We are given the string $A=\langle a_1,a_2,...,a_n\rangle$ and the pattern $B=\langle b_1,b_2,...,b_m\rangle$ that have exact values. We need to mention that the elements of A and B must be from the the universe of discourse of the same linguistic variable: $a_i \in U$ for $\forall i = 1..n$ and $b_j \in U$ for $\forall j = 1..m$. Every element of U has a corresponding threshold. We need to find if the string A is dotted, fuzzy matched to the pattern B so that every element of A is matched to B with the threshold greater than or equal to the corresponding threshold T(a).

Now let's clarify how the fuzzy matching works:
We consider a_i and b_j a match, when a_i and b_j match the same value of the linguistic variable, and the difference in the degrees of membership of a_i and b_j to this value is smaller than or equal to the given corresponding threshold $T(a_i)$.

Example: We have $a_i = 2$, $b_j = 1$, $T(a_i) = 0,5$. The linguistic variable is *size*.
$\boldsymbol{\mu}_{small}(a_i)= 0.75$ $\boldsymbol{\mu}_{small}(b_j)= 1$ $|\boldsymbol{\mu}_{small}(a_i) - \boldsymbol{\mu}_{smalll}(b_j)| = 0,25 \leq 0,5 \Rightarrow$ a_i and b_j match.

This is also a special case of the longest common subsequence problem. The algorithm in this case is the same as in case of the general LCS, the only difference being the matching.

If in case of the general LCS we say that we have a match between i_{th} element of A and j_{th} element of B when $a_i == b_j$, in this case we have fuzzy matching.

```
FLCS-LENGTH(A, B)
1  n=A.length
2  m=B.length
3  let b[1..n,1..m] and c[0..n,0..m] be new tables
4  for i = 1 to n
5        c[i,0] = 0
6  for j = 0 to m
7        c[0,j] = 0
8  for i = 1 to n
9        for j = 1 to m
10             if FUZZY_MATCH($a_i$, $b_j$,T($a_i$))
```

```
11              c[i,j] = c[i-1,j-1] + 1
12              b[i,j] = "↗"
13          elseif c[i-1,j] ≥ c[i,j-1]
14              c[i,j] = c[i-1,j]
15              b[i,j] = "↑"
16          else c[i,j] = c[i,j-1]
17               b[i,j] = "←"
18   return c and b
```

U has n elements. The linguistic variable has k values.
The degrees of membership of each element of the universal set (U) to each value of the linguistic variable will be stores in a matrix dm[k][n]. If the elements of U are not integers we will also have an array of strings (elements of U) u[n], to know which value corresponds to which column in dm[k][n].

```
FUZZY_MATCH(x, y, T)
1  for i = 1 to n
2       if u[i] == x
3          nx = i
4       if u[i] == y
5          ny = i          \\ don't need 1 to 5 if the elements of U are integers
6  for i = 1 to k
7       if (absolute(dm[i][nx] - dm[i][ny]) < T)
8           return true
9  return false
```

Constructing the LCS will be the same as in the general LCS algorithm.

Chapter 4
Fuzzified string distance

4.1 Distance between a string and a fuzzy pattern

For this case the String Distance algorithm will be the same as the general String Distance algorithm, with one difference. We are given a threshold T and we need to set the distance between the linguistic variable value and the exact string element to 0 when the degree of the membership of the exact string element to the linguistic variable value is greater than or equal to the threshold.

So the only difference in the algorithm will be in the properties of d in terms of subtitute operations performed on single (nonemply) letters λ and τ:

- (insert: replace ε by λ) $d(\varepsilon,\lambda) > 0$;
- (delete: replace λ by ε) $d(\lambda,\varepsilon) >0$;
- (substitute: replace λ by τ) $d(\lambda,\tau) > 0$ iff $\boldsymbol{\mu}_{\tau}(\lambda) \geq T$

Every other part of the algorithm is the same as in the general string distance algorithm.

4.2 String Distance with fuzzy matching

In this case, again, the String Distance algorithm will be the same as the general String Distance algorithm, with one difference. We are given a threshold T(u) for every element of U (the universy of discourse both the string and the pattern belong to) and we need to set the distance between the exact string elements to 0 when between the 2 elements there is a fuzzy match. The FUZZY_MATCH(λ, μ, T(λ)) is the same as in the previous chapter.

So the only difference in the algorithm will be in the properties of d in terms of edit operations performed on single (nonemply) letters λ and μ:

- (insert: replace ε by λ) $d(\varepsilon,\lambda) > 0$;
- (delete: replace λ by ε) $d(\lambda,\varepsilon) >0$;
- (substitute: replace λ by μ) $d(\lambda,\mu) > 0$ iff FUZZY_MATCH(λ, μ, T(λ))

Every other part of the algorithm is the same as in the general string distance algorithm.

Conclusion

In this thesis four algorithms on two topics were presented. All of this algorithms were specific cases of approximate string matching problem.
The algorithms presented in this thesis can be viewed as modifications of the longest common subsequence and string distance algorithms using fuzzy set theory and the concept of linguistic variables. All of the algorithms were using the dynamic programming approach.

References

[1] Cormen, Thomas H., Charles Eric Leiserson, and Ronald L. Rivest. Introduction to Algorithms. 3rd ed. Cambridge, Mass. : MIT Press, 2009.

[2] Smyth W. Computing Patterns in Strings. Pearson/Addison-Wesley, 2003

[3] L. A. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, no. 3, pp. 338-353, June 1965.

[4] L. A. Zadeh, "The concept of a linguistic variable and its application to approximate reasoning - I," Information Sciences, vol. 8, no. 3, pp. 199-249, July 1975.

[5] Novák, V., Perfilieva, I. and Močkoř, J., "Mathematical Principles of Fuzzy Logic", Kluwer Academic Publishers 1999.Print

[6] A. Kostanyan. Fuzzy String Matching with Finite Automata, in Proceedings on 2017 IEEE Conference "2017 Computer Science and Information Technologies" (CSIT), Yerevan, Armenia, 25-29 Sep., 2017, IEEE Press, USA, 2018, pp. 9 - 11.