Comenius University
Faculty of Mathematics, Physics and Informatics
Department of Computer Science

The Distances on Words

(Master's Thesis)

Tomáš Kulich

Advisor: Prof. Branislav Rovan, PhD. Bratislava, 2006

I hereby declare that the work presented in this thesis is my own, written only with help of referenced literature, under the careful supervision of my thesis advisor.

. . . .

Acknowledgements

I would like to thank prof. Rovan for self-sacrificing guidance and many valuable suggestions.

I would also like to thank my friend Michal Pokorný who provided me with advices how to keep with my time-plan.

Abstract

Lot of research was done on the string similarity, due to many practical applications (similarities of DNA/RNA structures, etc). In the first part of this work, we shall introduce an abstract model of the distance of words, from which the known distances such as edit-distance (Levenshtein distance) can be derived. We shall present some other instances of this model. We shall discuss what derivations can/cannot be effectively computed. In the second part of this work we shall extend the notion of the distance on words to the distance of a word and a language. We present algorithm for computing Δ-similarity of a word and a context-free language.

Contents

Chapter 1

Similarities and distances

1.1 Introduction

The problem of determining, whether two strings are similar or not, is as old as the mistakes done during some transcription process. Today, similarity of strings is important mainly in:

(i) Similarity of DNA / RNA structures

(ii) Searching engines (Google, Amazon, AltaVista, etc.)

(iii) Correcting mistakes done during typing text

(iv) Protection from plagiarism

This wide area of applicability indicates that there can hardly exist one universal measure which would satisfy all these needs. This is exactly the reason, why one can invest an endless effort to invent special distance functions that would be fitted for very special sets of problems (and would be very accurate for them). As a result of this effort we have many measures (and implemented algorithms) and often one is not sure which algorithm is the best one to use. For example in (i) only skilled molecular biologists know, which algorithm will do the job best.

In this work, we shall present some frequently used distances and also some distances that were not studied so well yet, or were not studied at all. We shall also define a new abstract model can be defined, which generalizes many distances to one abstract distance and thus it shows their common base. In the second part of this thesis we shall investigate the distance between a word and a language. We shall present known algorithm for edit-distance between word and context-free language and also a new algorithm for Δ-similarity between word and context-free language.

1.2 Similarities and distances

In this chapter, we shall introduce some similarities and distances we can compute on finite words. We shall recall the edit-distance, which is a standard and well studied distance on finite words, then Δ-similarity which was introduced in [1]. We shall be interested in some modifications of the edit-distance and then, some new distances will be introduced, which, were not investigated so far. Let $D : \Sigma^* \times \Sigma^* \to \mathbb{R}$ be a distance function, measuring the distance between two given words. Usually we require D to have these properties:

$D(w, w) = 0$ (reflexivity)

$D(u, w) = D(w, u)$ (symmetry)

$D(u, w) + D(w, v) \geq D(u, v)$ (triangle inequality)

However, we shall also consider other measures, e.g., measures that are not symmetric (fragment distance). When defining a new distance measure, it is important to know, which of these properties it has. Let us now consider requirements a similarity function S should satisfy. First, $S(v, w)$ should be a real number between 0 and 1 (values 0 and 1 are allowed). The reason is simple. We expect $S(v, w)$ to provide the percentage of how similar the two words are. There are, again three properties, that S should satisfy:

$S(w, w) = 1$ (reflexivity)

$S(u, w) = S(w, u)$ (symmetry)

1.3 Abstract model for distances and its instances

1.3.1 Basic edit-distance

Edit-distance is a common way of measuring distances between words. It is motivated by editing text on a computer. Suppose we have a word u written in a editor and we need to rewrite it to w. To transform u to w we can use one of the following operations:

(i) Delete any letter from the word. Letters to the right of the deleted letter will be shifted left, so that no space is created.

(ii) Insert any letter to some position in the word. Letters to the right of this position are shifted to the right.

(iii) Replace any letter by another one (substitution).

One can see that inserting and deleting is used in the usual way, as we know it from typing the texts. Now, edit-distance is the minimal number of operations which allows us to transform one word into another. This definition is quite simple. We shall work with a more abstract definition, that will allow us to treat other distances, using a similar principle. Many distance functions can be described by a triple (O, T, C). O is a universum of all possible operations that can be performed on words. T is a translation function defined as follows:

$$T : O \times \{\Sigma^* \cup \{\bot\}\} \to \Sigma^* \cup \{\bot\}$$

C is a cost function, $C : O \to \mathbb{R}$. By a multi-operation we shall mean some finite sequence of members of O. Universum of all these sequences will be denoted by O^*. We shall treat the multi-operations as words and we shall use standard word-operations to manipulate them. The translation function can be extended also for multi-operations in this way: Let $T(\epsilon, w) = w$ and for $o_1, \ldots, o_n \in O$

$$T(o_1 \ldots o_n, w) = T(o_n, T(o_{n-1}, (\ldots T(o_1, w) \ldots)$$

If $u = T(o, w)$ for some words u, v and a (multi-)operation o we say, that o transforms u into v. Symbol $\bot$ means, that the result of the translation is not defined and we define $T(o, \bot) = \bot$. Also the cost function can be extended from operations to multi-operations. There are more possibilities how to do this. The following definition suffices for the needs of this thesis: [1]

$$C(o_1 \ldots o_n) = C(o_1) + \ldots + C(o_n)$$

Now we can finally define the distance between two words.

Definition 1.3.1. *Given a triple (O, T, C) described above, the edit-distance between words u, v is defined to be the cost of the cheapest multi-operation, that transforms u into v. This distance of words u and v will be denoted as $d_{OTC}(u, v)$ (when it is clear what triple (O, T, C) we are using, we can omit the subscript OTC).*

[1] Another approach is shown in [4], where authors show stochastic approach to edit-distance.

We shall now use this framework for defining the "standard" edit-distance, mentioned earlier, in a formal way. However, we shall not allow substitution as an operation. The reason for this is simple, the substitution can be done using insertion and deletion, so it is redundant. Of course, it has its meaning when considering costs of operations, but when looking for the simplest measure, we do not want such redundancy. So, let us define an instance of edit-distance which we shall call the basic edit-distance.

Definition 1.3.2. The basic edit-distance is the edit-distance defined using the triple (O, T, C), where:
O consists of pairs (a, n) $\ a \in \Sigma \cup \{\epsilon\}$, $n \in \mathbb{N}$. Operations of type (ϵ, n) will be called deletions, the other operations will be called insertions.
The translation is defined as follows:

$$(T((\epsilon, n), w) = u) \leftrightarrow ((\exists p, s \in \Sigma^*)\ (\exists a \in \Sigma)\ w = pas\ \wedge\ u = ps\ \wedge\ |p| = n)$$

If $a \in \Sigma$ then

$$(T((a, n), w) = u) \leftrightarrow ((\exists p, s \in \Sigma^*)\ w = ps\ \wedge\ u = pas\ \wedge\ |p| = n)$$

In both definitions at most one u satisfies the conditions, so they are correct. If there is no u satisfying the condition (such a, p, s, resp. p, s do not exist), then the translation result is undefined.
The cost function is defined by: $C(a, n) = 1$ for all values of n, and $a \in \Sigma \cup \{\epsilon\}$.

This is one instance of the edit-distance. Varying O, T and C we can modify the edit-distance, so it will have many useful properties. Before we look at these modifications, let us first see some properties of the basic edit-distance and consider the way to compute the basic edit-distance.

Theorem 1.3.1. *The basic edit-distance is reflexive, symmetric and it satisfies triangle inequality.*

Proof: The reflexiveness is trivial, we do not need any operation to transform a word to itself. Consider words u, v and a multi-operation o which transforms u into v. When replacing all insertions by deletions and vice versa we can contruct the multi-operation o' having the same cost, that transforms v into u. This means, that the basic edit-distance is symmetric. When transforming u into w, we can first transform u into some v and then this v into w. The multi-operation that does this is not necessarily the cheapest one, but it is some way of transforming u into w, so $d(u, w) \leq d(u, v) + d(v, w)$. $\square$

Definition 1.3.3. *Let d_{OTC} be a distance measure, and $A \subseteq O$. By C_A we denote such cost function, that counts the cost only of those operations, that are member of A. Formally:*

$$C_A(o_1 \dots o_n) = \sum_{o_i \in A} C(o_i)$$

$C_A(o)$ will be called a partional cost of o.

Definition 1.3.4. *Let d_{OTC} be a distance measure, $A_1, \dots, A_n \subseteq O$ and let for any $i \neq j$ the condition $A_i \cap A_j = \emptyset$ holds. Let o be any multi-operation. We say that o can be aligned in order $A_1, \dots, A_n$, if there exists a multioperation $o' = o'_1 \dots o'_n$ such that these four conditions hold:*

(i) $(\forall u \in \Sigma^*)\ T(o, u) = T(o', u)$

(ii) $C(o') \leq C(o)$

(iii) $(\forall i \in \mathbb{N})\ C_{A_i}(o') \leq C_{A_i}(o)$

(iv) $(\forall i, j \in \mathbb{N})\ ((i \leq j \wedge o'_i \in A_k \wedge o'_j \in A_l) \Rightarrow k \leq l)$

Speaking informally, o can be aligned in order $A_1, \dots, A_n$ if we can find o' that realizes the same translation, it is not more expensive and even its "partial" costs are not bigger and o' has structure, that operations from A_1 go first, then A_2, etc..

Lemma 1.3.1. *Considering the basic edit-distance, any multi-operation o can be aligned in order: deletions, insertions. It can also be aligned in order: insertions, deletions.*

Proof: Consider two words u, v and a multi-operation o that transforms u into v. We can make o cheaper (and even partially cheaper) if we eliminate of all such pairs insertion, deletion, where the deletion deletes the same letter that the insertion inserted. Clearly the multi-operation o' obtained in this way will have the same translation meaning as o. The deletions of o' can delete only the original letters of u, so they can happen at the begining of o' and the insertions can be done afterwards. The proof, that also alignment in the order insertions, deletions is also possible is similar. $\square$

Notation 1.3.1. *Let u be a word and i be an integer. Let us denote by u_i the i-th letter of word u. Also, let us denote by $[u]_i$ the prefix of u of length i.*

Notation 1.3.2. *Let us denote by $\mathbb{N}_k^l$ the set of all integers i, $k \leq i \leq l$. By $\mathbb{N}^+$ we denote all non-negative integers, e.g.,* $0, 1, 2, \ldots$

Definition 1.3.5. *The longest common subsequence of words u, v is the word w with maximal length $|w| = n$ such that there exist $k_1, \ldots, k_n$ and $l_1, \ldots, l_n$ such that $\forall i, j\ i < j \Rightarrow k_i < k_j \wedge l_i < l_j$ and $\forall i \leq n\ u_{k_i} = v_{l_i} = w_i$. We shall denote the longest common subsequence of words u and v by $LCS(u, v)$.*

We shall use also another look on LCS. Let $f : \mathbb{N}^+ \to \{\mathbb{N}^+ \cup \{\perp\}\}$ be an increasing function such that for every i either $f(i) = \perp$ or $u_i = v_{f(i)}$ holds. If we take such f that has minimal number of $\perp$ values (for $i = 0, \ldots, |u|-1$), then the leters on positions in which f is defined form the LCS. We shall also say that f defines partial mapping between letters of the words u, v.

Theorem 1.3.2. *Let d denote the basic edit-distance. Then the following equation holds:*

$$(\forall u, v \in \Sigma^*)\ d(u, v) = |u| + |v| - 2.|LCS(u, v)|$$

Proof: First, we shall prove that we can transform u into v using $|u| + |v| - 2.|LCS(u, v)|$ operations. First we take u and delete all letters that are not in the longest common subsequence. We can do this using a multi-operation with cost $|u| - |LCS(u, v)|$. Next, we shall add those letters that are in v and are not in the longest common subsequence. This can be done using a multi-operation with cost $|v| - |LCS(u, v)|$. The total cost of these two multi-operations is exactly the right side of our equation, so we prove that

$$(\forall u, v \in \Sigma^*)\ d(u, v) \leq |u| + |v| - 2.|LCS(u, v)|$$

Let us prove the other inequality. Using Lemma 1.3.1 we know, that for any words u, v there exists multi-operation o such that $T(o, u) = v$, $C(o) = d(u, v)$ and $o = o_1.o_2$, where o_1, o_2 are multi-operations consisting only from deletions and insertions respectively. Let $w = T(o_1, u)$. It is easy to see that $T(o_2, w) = v$ so there exists a multi-operation o_3 consisting only of deletions, such that $T(o_3, v) = w$ and $C(o_3) = C(o_2)$. Length of w is equal to $|u| - C(o_1)$ and also $|v| - C(o_3)$. Combining these two equations we have $2|w| = |u| - C(o_1) + |v| - C(o_3) = |u| + |v| - C(o)$. w represents some common subsequence of words u and v [2] so its lenght cannot be greater than $|LCS(u, v)|$. So, we have: $2.|LCS(u, v)| \geq |v| + |u| + C(o) \geq |v| + |u| + d(u, v)$. From this we easily obtain $d(u, v) \geq |v| + |u| - 2.|LCS(u, v)|$. $\Box$

[2] In fact, the words w and $LCS(u, v)$ are identical

This theorem gives us a direct method for calculating the basic edit-distance. All we have to do is to determine the length of LCS which can be done, e.g., by the well-known algorithm based on the dynamic programming. We shall use another approach that will be usefull later in this thesis. We shall directly compute the basic edit-distance by a similar algorithm, also based on dynamic programming.

Algorithm 1.3.1.

Input: *words* $u, v \in \Sigma^*$

Output: *the basic edit-distance* $d(u, v)$

We shall use an auxiliary function $A : \mathbb{N}_0^{|u|} \times \mathbb{N}_0^{|v|} \rightarrow \mathbb{R}$ such that $A(i, j) = d([u]_i, [v]_j)$ (Recall that $[u]_i$ stands for the prefix of the word u of length i). [3] Of course, $A(|u|, |v|)$ is the value we need to compute, but in our algorithm, we shall compute every $A(i, j)$. In the beginning, we can easily initialize the values $A(i, 0) = i$ and also $A(0, j) = j$. We now show how to compute $A(i, j)$. Let $o = o_1..o_n$ be multi-operation which transforms $[u]_i$ into $[v]_j$. If the last letters of $[u]_i$ and $[v]_j$ are the same, then $A(i, j) = A(i - 1, j - 1)$. If the last letters are different, then o must somehow achieve that the last letters of $T(o, [u]_i)$ and $[v]_j$ are the same. (Of course, these two words must be identical, but let us focus just on the last letters.) This can happen in two ways. Either $v_j = u_k$ for some k. Then o can simply delete all letters beyond the k-th position and the last letters will be the same. In this case $A(i, j) = A(i - 1, j) + 1$ where 1 is the cost of the deletion. Or, the letter v_j is just inserted to the end of $T(o_1..o_k, [u]_i)$ for some k (it does not make sense not to insert it at the end). In this case $A(i, j) = A(i, j - 1) + 1$, where 1 is the cost of the insertion. So, we have the two equations which correspond to the two ways of transforming $[u]_i$ into $[v]_j$. The distance will be the smaller one of these values. If we summarize all this we can write the final formula:

$$A(i, j) = \min\{A(i - 1, j - 1) + d(u_i, v_j), A(i - 1, j) + 1, A(i, j - 1) + 1\} \quad (1.1)$$

In this formula $d(u_i, v_j)$ is 0 if $u_i = v_j$ and 2 if not. It holds that $A(i - 1, j) \leq A(i-1, j-1)+1$, so if $u_i \neq v_j$, then $A(i-1, j-1)+2$ will be greater then other two values and therefore not picked up by the minimization. Equation 1.1 gives us a direct way of computing $A(i, j)$. All we have to do is to start with small arguments of the function A and remember these results for further computation. If $i > 0, j > 0$ then $A(i, j)$ can be computed in constant time using the values $A(i, j), A(i - 1, j), A(i, j - 1)$. If $i = 0$ or $j = 0$, the value of $A(i, j)$ is known from the initial phase.

[3] Thus, we can visualize A as a table of size $|u| \times |v|$.

From this it is easy to see, that the time complexity of the algorithm is $O(|u|.|v|)$, which is exactly the number of operations needed to fill the table of size $|u| \times |v|$ with values of the function A. The space complexity of the algorithm described above is also $O(|u|.|v|)$, but it can be easily reduced to $O(\min\{|u|, |v|\})$. This is so because we do not need to remember all values of $A(i, j)$; what we really need is just one row or one column of these values. Using this row (column) and the initial values (no need to keep them in the memory) we can compute the next row or the next column.

1.3.2 Computing the edit-distance effectively

From our definition of the edit-distance is clear, that there are many edit-distances that can not be computed even on Turing machine with no time limit. It is natural to try to find some conditions, under what edit-distance will be computed effectively. [4]

Definition 1.3.6. *A set of operations O is called $k-$bounded if*

$$\begin{aligned}
&(\forall o \in O)\ (\forall u, u' \in \Sigma^*)\ T(o, u) = u' \Rightarrow \\
&(\exists p, w, w', s \in \Sigma^*\ |w'| \leq k\ \wedge\ |w| \leq k\ \wedge\ u = pws\ \wedge\ u' = pw's\ \wedge \\
&(\forall p', s' \in \Sigma^*\ \exists o' \in O\ T(o', p'ws') = p'w's'\ \wedge\ C(o') = C(o)))
\end{aligned}$$

Speaking informally, O is $k-$bounded if all operation it contains are modifying only some sub-word of length k and this modification does not depend on the context surrounding this sub-word. If $o \in O$ does some modification to some sub-word then for every context surrounding this sub-word must exist an operation, that transforms the sub-word in the same way and with the same cost.

Let us now focus on the translation function. It is clear, that if we want to compute the edit-distance effectively, the translation function must be computed quickly. What it means is shown it the next definition:

Definition 1.3.7. *Let $f : \mathbb{N}^+ \to \mathbb{R}^+$. We say that an operation o is f-computable, if for all $u \in \Sigma^*$ and for all $i \leq |u|$ we can compute the i-th letter of $T(o, u)$ in $f(|u|)$ time. A set of operations O is f-computable if all of its members are f-computable.*

Next problem, which can occur in computing edit-distance is due to big number of operations, that can be performed on words. Let us focus on the basic edit-distance, as it was defined earlier. The set O consists of just

[4]Effectively means in polynomial time, but we shall analyze complexities of our algorithms more precisely

two types of operations, but in fact it is infinite. In spite of this we can compute the basic edit-distance effectively, because for a given word only a finite subset of operations in O have their result of translation defined. For every word we can find this subset quickly. If O is $k-$bounded, then we can focus only on operations that translate words of length k. Trivially, there can be only a finite number of operations that differ from each other in their translation meaning. Problem is, if we have an infinite number of operations that are doing 'the same'. In this case, computation can be impossible, because we must search an infinite set of operations just to make sure, they are really identical. How can it possibly happen that there are infinitely many operations with the same meaning? No 'reasonable' definition of O and T leads to this, so we can solve our problem simply by forbidding these abnormal cases. In further text, when talking about $k-$bounded set of operations, we shall automatically assume, that for every word there exist only finite number of operations that have the same translation meaning.

Now, when talking about $k-$bounded computable set O, we avoided all evident reasons, why we cannot compute generalized edit-distance effectively, so we could have optimistic anticipation that we shall be able to compute this edit-distance effectively. Following theorem shows, that in spite of these restrictions problem still remains quite hard.

Theorem 1.3.3. *Let O be a k-bounded set of operations that is $O(1)$-computable. Let d be an edit-distance defined by a triple (O, T, C). The problem of determining, whether the distance of two words is smaller than some given constant c is NP-complete.*

Proof: First, we can solve this problem by a non-deterministic polynomial algorithm by guessing some multi-operation and verifying, whether it really transforms one word to another and also verifying, if its cost is smaller than c. The proof of NP-completeness will be done by a reduction to problem, whether given turing machine M accepts given word u on less than d steps, where d is given constant. There are several possibilities for definying a configuration of a Turing machine. We shall use a definition, where the configuration is given by the contents of the tape and the head position is marked within this contents by a special symbol representing the current of the finite control of the machine. For example $abbabq_1bab$ means a configuration, where $abbabbab$ is the content of the tape, machine is in state q_1 and the head is located on the third letter from the right.

We shall construct a Turing machine M' which accepts the word u, if and only if M accepts u in less than d steps. We can clearly assume that the unique accepting configuration of M' on the word u is the configuration $z^{|u|}q_f$ where z is a special symbol and q_f is unique accepting state of M'.

Furthermore, M' has ability to write blank symbols to the some position of tape, if adjacent position already contains blank symbol. Note that this "ability" does not mess with definition of configuration of the machine.

Now, we reduce the problem whether M' accepts a word u to computation of the distance between $v = Bq_0uB$ and $w = Bz^{|u|}q_fB$, where q_0 and q_f is the initial and the final state of M', respectively, the letter B stands for the blank symbol. Words will be constructed over the alphabet Σ which is union of the tape alphabet of M' and the symbols for states of M' (to prevent confusion, these must be different from letters of tape alphabet). We allow only one type of operations. Let $t_1a_1qa_2t_2$ be a word, where t_1 and t_2 are some words, a_1, a_2 are letters from $\Sigma - B$ (they are letters, not blank symbols) and q is a state symbol in M'. Now we can transform $t_1a_1qa_2t_2$ into:

(i) $t_1pa_1a_3t_2$ if $(p, a_3, -1) \in \delta(q, a_2)$

(ii) $t_1a_1pa_3t_2$ if $(p, a_3, 0) \in \delta(q, a_2)$

(iii) $t_1a_1a_3pt_2$ if $(p, a_3, 1) \in \delta(q, a_2)$

Using these operations we are trying to simulate the computation of M' on the word u. The last thing we must solve is the problem with the end of the tape. Turing machine has infinite number of blank symbols at each end. We have only one blank symbol at each end, which can lead to problems. But compared to Turing machine we have one "special ability", we can write many (two) symbols at once. Using this, we can manage the ends of the tape as follows: If the word looks like $t_1a_1q_iB$ we can transform it into:

(i) $t_1pa_1a_2B$ if $(p, a_2, -1) \in \delta(q, B)$

(ii) $t_1a_1pa_2B$ if $(p, a_2, 0) \in \delta(q, B)$

(iii) $t_1a_1a_2pB$ if $(p, a_2, 1) \in \delta(q, B)$

for left end of the tape we have similar three rules. Cost of all operations can be arbitrary. Using these operations, we can direct simulate the computation of M' on the word u. If M' accepts u then the distance of v and w will be finite, else undefined $\perp$-value. The set of operations is $2-$bounded. $\Box$

So, we see that even with these restrictions we cannot compute the generalized edit-distance effectively. On the other hand, as we shall see, we can compute the basic edit-distance even with some added operations.

1.3.3 Modifications of the basic edit-distance

We have an algorithm to compute the basic edit-distance which is just one particular instance of our abstract definition of the edit-distance. Now we shall be interested in the question, what other instances of our abstract definition of the edit-distance can be effectively computed. First, we can have different costs for insertions and deletions. This just changes 1's in Formula 1.1 to actual values of insertions or deletions. We can also have special costs for the insertion of every letter. To do the same with deletions, we need to modify the definition of deletion. Instead of a pair (ϵ, n) it will be a triple (a, ϵ, n) where $a \in \Sigma$. The meaning of ϵ is just to distinguish deletions from insertions. The translation function on these triples will be defined naturally - if $n+1$-th letter of a given word is a then it will delete it, else the result is undefined. Now we can formulate a simple formula similar to 1.1:

$$\begin{aligned} A(i,j) = \min\{&A(i-1,j-1) + d(u_i, v_j),\ A(i-1,j) + d(u_i,\epsilon), \\ &A(i,j-1) + d([u]_i, [u]_i.v_j)\} \end{aligned}$$

or, when we want to avoid trivial distances in this formula, we can re-write it to the form:

$$\begin{aligned} A(i,j) = \min\{&(A(i-1,j-1) + 2.(u_i == v_j)), A(i-1,j) + C(u_i,\epsilon,i-1)), \\ &A(i,j-1) + C(v_j, i)\} \end{aligned}$$

where $u_i == v_j$ stands for 0 if $u_i = v_j$ and ∞ otherwise.

It is also common to add some new operations to the set O. The most common operation added is substitution, which is in fact a multi-operation $(\epsilon, n).(a, n)$. In most of the related work the edit-distance is automatically taken with substitution included. It is wise to treat the substitution as a single operation, because we can have special cost for it (different from multi-operation written above). In our setting the substitution will be a triple (a, b, n) where a is a letter which is deleted, b is the inserted letter and n specifies the position where this happens. Formally, $T((a,b,n),u) = pbs$ where $u = pas$ and $p = [u]_n$ (If such p, s do not exist, the result is undefined), $a, b \in \Sigma$ (so they can not be ϵ).

Another very useful operation is the transposition of two adjacent letters. The transposition of two adjacent letters is a common error which can occur when someone is typing a text, so it is good that we can compute edit-distance with transpositions effectively. According to the definition of the basic edit-distance with different cost for deletions (deletion is a triple), transposition is an operation $(trans, a, b, n)$ with translation meaning the same as the multi-operation

$$(a,\epsilon,n).(b,\epsilon,n).(a,n).(b,n)$$

Now we show, how we can compute edit-distance with a transposition effectively. First we shall analyze the problem, when cost of all operations is equal to 1. Later, we shall show how we can deal with different costs. We shall use similar approach as when computing the basic edit-distance. For two words u, v we shall compute the function $A : \mathbb{N}_0^{|u|} \times \mathbb{N}_0^{|v|} \times \Sigma \cup \{\epsilon\} \to \mathbb{R}$ with the meaning: $A(i, j, a)$ is a distance between words $[u]_i.a$ and $[v]_j$. It is obvious, that $A(i, j, \epsilon) = A(i-1, j, u_i)$. Now, let us focus on a problem, how it can happen that the last letters of these two words are identical. This can happen in the following ways:

(i) $a = v_j$ already, we have no problem,
$A(i, j, a) = A(i, j-1, \epsilon) = A(i-1, j-1, u_i)$

(ii) v_j letter is added to the end of $[u]_i.a$, $A(i, j, a) = A(i, j-1, a) + 1$

(iii) v_j is somewhere inside $[u]_i.a$, but it is not the last or the last but one letter. Now the last or the last but one letter must be deleted (we shall see further why), so we have: $A(i, j, a) = A(i, j, \epsilon) + 1 = A(i-1, j, u_i) + 1$ when deleting the last letter or $A(i, j, a) = A(i-1, j, a) + 1$ when deleting the last but one.

(iv) v_j is identical with u_i. Now we can use the transposition, to move it to the end. $A(i, j, a) = A(i-1, j-1, a) + 1$

Now it is no problem to write a formula for $A(i, j, a)$, similar to 1.1. To show its correctness we shall prove:

Theorem 1.3.4. *For $A(i, j, a)$ at least one equation from four cases (i)-(iv) written above holds.*

Proof: We shall use following lemma:

Lemma 1.3.2. *Considering the basic edit-distance with transpositions, any multioperation o can be aligned in order deletions, transpositions, insertions. This holds for arbitrary costs of operations.*

Proof: We use a similar reasoning as in Lemma 1.3.1. Any multi-operation can be converted to a (partially) cheaper one, in which no letter that was inserted will later be deleted and also, if transpositions will transpose only those original letters, that are not going to be deleted. Now it is clear, that we can make deletions at the begining, then transpositions and insertions at the end. □

Now to the proof of Theorem 1.3.4. First, we want to show, why this proof is not as easy as it was in Algorithm 1.3.1. Cases (i)-(iv) correspond to four possibilities, how can v_j appear at the end of the other word. If v_j is somewhere inside the other string and we want to get it to the end of it, we can do it using (iii) and (iv), but for example, we did not discuss the case when v_j comes to the end by several transpositions. Furthermore, we can use a multi-operation which is made of some deletions and some transpositions. These possibilities are not considered in our algorithm.

We shall solve this problem by showing that there always exists cheapest multi-operation found by our algorithm. Let us consider a cheapest multi-operation o that transforms the word u into $v.a$ where u, v are words and a is a letter. Using Lemma 1.3.2 we can assume that o has the following structure: first deletions, then transpositions, then insertions. Moreover, we shall assume that from all multi-operations satisfying these conditions, o has minimal partial cost of transpositions. Let $o = o_1 \dots o_n$ and let o_i, o_j $(i < j)$ be some transpositions that are touching (i.e. transposing) some letter z and no transposition between them is touching z. There are two posibilities, how o_i, o_j can look like:

(i) $o_i = (trans, z, b, n)$ and $o_j = (trans, z, c, n+1)$ (for some $b, c \in \Sigma, n \in \mathbb{N}$)

(ii) $o_i = (trans, z, b, n)$ and $o_j = (trans, c, z, n)$ (for some $b, c \in \Sigma, n \in \mathbb{N}$)

When (i) happens, then we can replace o_i by the deletion (z, ϵ, n) and o_j by the insertion $(z, n + 2)$. The translation meaning and also the cost remains the same. If (ii) happens, we can replace o_i by the deletion (z, ϵ, n) and o_j by the insertion (z, n). Once again, neither the translation meaning, nor the cost changed. Both cases lead to a contradiction with the assumption on the minimality of the partial cost of transpositions of o, so such o_i, o_j cannot exist. This proves that each letter is touched by at most one transposition.

Now it is clear, that the cases (i)-(iv) exhausted all the possibilities how v_j can appear at the end of $u.a$, either $a = v_j$ (i), or it is inserted there (ii), or it is the last but one letter and we need exactly one transposition (iv); or it is somewhere inside, and then at most one transposition can touch it. From this we know, that all letters right to it except one must be deleted and therefore the last or the last but one letter of $u.a$ must be deleted. And this is exactly what (iii) is counting with. □

In the previous algorithm we were significantly using the particular definition of the costs of the operations. Situation will be slightly more difficult, when trying to compute the edit-distance with transposition with generalized costs. Now, we cannot easily replace two adjacent transposition by deletion and insertion, but fortunately, there exists a finite number of adjacent transpositions, that can be replaced (without increasing the total cost) by insertion and deletion. Let p be the maximal cost of inserting plus the cost of deleting for any letter of the alphabet and let r be the minimal cost of transposition. Now, minimal number of adjacent transpositions k that we can replace by insertion and deletion is the minimal integer k such $k.r \geq p$. Thus $k = \lceil \frac{p}{r} \rceil$. If $r \neq 0$ we have a straight-forward way to compute the distance. We know that if a letter a contained in a word u is to be moved to the end of this word, it can be touched by at most k transpositions. That is why we must "widen" function A to $\mathbb{N}_0^{|u|} \times \mathbb{N}_0^{|v|} \times (\Sigma \cup \{\epsilon\})^k \to \mathbb{R}$. The meaning of A stays the same. When we construct A for words u and v then $A(i, j, w)$ means distance between words $[u]_i.w$ and $[v]_j$.

Lemma 1.3.3. *Let u, v be words and $o = o_1 \dots o_n$ is the cheapest multi-operation that consists just of transpositions such that $T(u, o) = v$. Let a be the last letter of v and let it be unique letter in it. Every transposition o_i that touches a, moves it forward (i.e. o_i is of a type $(trans, a, b, l)$).*

Proof: For contradiction, let there be an operation $o_i = (trans, b, a, l)$. Now, letter b gets to the right of a. At the end of transforming u by o, the letter a must be at the end, so a must be transposed with b once more using transposition $o_j = (trans, a, b, k)$. When both o_i, o_j are transforming the "actual version" of u, the letters a and b must be adjacent. From this we know that between transposition o_i and o_j the letters a and b must be transposed with the same set of letters. So, if we insert o_j right after o_i ($j = i + 1$), we can permute the rest of o in the way that neither translation meaning, nor cost is changed. But now, these two transpositions negate themselves and can be omitted, what is a contradiction with the fact that o is the cheapest. $\square$

Lemma 1.3.4. *Let w_1, w_2 be words and a be the last letter of w_2. Let o be the cheapest multi-operation such that $T(o, w_1) = w_2$. Let o_1 be the cheapest multi-operation aligned in order: deletions, transpositions, insertions, such that there exist multi-operation o_2 such that $T(o_1 o_2, w_1) = w_2$, $C(o_1 o_2) = C(o)$ and the last letter of $T(o_1, w_1)$ is a. There are only following cases how o_1 can look like:*

(a) $o_1 = \epsilon$

(b) *o_1 is one insertion of a to the end of w_1*

(c) *Let us focus on last (most right) occurence of a in w_1. Now o_1 consists of some deletions on the right of this occurrence followed by at most k transpositions that move a forward (transpositions of the type $(trans, a, b, l)$). Furthermore, $o_1 \neq \epsilon$.*

Proof: The case (a) happens, if and only if the last letter of w_1 is a. If o_1 contains one insertion, then from minimality of the cost of o_1 it must be exactly the insertion of a to the end of w_1. Furthermore, no other operation can be included in o_1. If o_1 is not empty and does not contain any insertion, then it consists only of transpositions and deletions. After the deletions take place, w_1 is transformed to some word w_3 which is transformed by transpositions to w_2. For w_3 and w_2 we can use Lemma 1.3.3. Furthermore, from minimality of the cost of o_1 it is clear that every used transposition must touch a and also that maximal number of transpositions is k. $\square$

Now, let us modify cases from Theorem 1.3.4 for this case. We can repeat the same arguments when considering (i), (ii) (this follows also from Lemma 1.3.4). When analysing how letter a from somewhere inside of the word can get to its end, we shall also use Lemma 1.3.4. We know that k is the maximal number of transpositions that is "allowed" to touch a. If a is not placed in the suffix of length k, then at least one letter from this suffix must be deleted. When all costs were equal to 1, we had to distinguish the cases whether the last or the last but one letter is deleted. Now we have to distinguish exactly k cases. So, instead of previous (iii) we shall have:

(iii) in case the l-th letter is deleted from the suffix of length k the following equation holds: $A(i, j, w) = A(i, j, w_1.w_2..w_{l-1}.w_{l+1}..w_k) + c$ where c is the cost of deletion of l-th letter

We must modify also the case (iv). Here we suppose that a is located in the suffix of length k and gets to its place by some transpositions (all deletions were done in (iii)). From Lemma 1.3.4 we know that all transpositions move only the last occurrence of a and they move it in forward direction, so we can write:

(iv) $A(i, j, s_1.a.b.s_2) = A(i, j, s_1.b.a.s_2) + c$ where $s1, s2 \in \Sigma^*$, $a = v_j$, s_2 does not contain a and c is the cost of used transposition

Last problem we have to solve is to find some order in which values of A can be computed. Similar to former algorithms, when having words of length n, m, we must compute $O(n.m)$ values of function A (third argument of A is the word of length k over finite alphabet and so it has only finite number of possibilities what it can be). In former algorithms values of $A(i, j, \mathit{anything})$ were computed using values $A(i', j', \mathit{anything})$, such that $i' \leq i \wedge j' < j \vee i' < i \wedge j' \leq j$. Now we compute $A(i, j, w)$ using $A(i, j, w')$ which can raise questions, whether our computation is finite. Now we show that there exists an ordering relation $\preceq$ (relation that is reflexive, anti-symmetric and transitive) on the set $\mathbb{N}_0^{|u|} \times \mathbb{N}_0^{|v|} \times (\Sigma \cup \{\epsilon\})^k$ with the following property: If we know values A on all triples that are smaller than some (i, j, w) (in the sense of $\preceq$) then we can also compute $A(i, j, w)$ (using (i)-(iv)).

Definition 1.3.8. *Let us have a function A computed for two words u, v as defined above. We define the relation $\preceq$: $(i, j, w) \preceq (i', j', w')$ if and only if one of these two conditions holds:*

(i) $i + j + |w| < i' + j' + |w'|$

(ii) $i + j + |w| = i' + j' + |w'| \ \wedge \ j = j' \ \wedge \ \mathit{Last}(v_j, [u]_i.w) \leq \mathit{Last}(v_j, [u]'_i.w')$

where the function $\mathit{Last}(a, u)$ returns the distance between the last occurrence of letter a and the end of the word, formally: $\mathit{Last}(a, u) = |u| - \max\{i \mid u_i = a\}$ if u contains at least one letter a; otherwise $\mathit{Last}(a, u) = |u|$.

Now, we have all we need. When we are computing the distance using (iii) or (iv) we are always referring to the values of A which are computed on smaller triples. Consider the time complexity of the algorithm. We must compute $|u|.|v|.|\Sigma|^k$ values of the function A. In spite of the fact, that $|\Sigma|^k$ can be quite large, it is still a constant, so is time for one of these values. For the total complexity we have $O(|u|.|v|)$.

After success with transpositions, we could think that we can compute the edit-distance with many other operations, that are k-bounded and that are just permuting the letters in some sub-word of a given word.

Definition 1.3.9. *Operation o is called permuting, if for all $u \in \Sigma^*$ it holds that $T(o, u)$ is either undefined, or it is some word, that is a permutation of the word u. The set of operations O is called permuting if for every $o \in O$ it holds that either o is an insertion, a deletion, or it is a permuting operation.*

Sadly, once more, we cannot compute this edit distance effectively.

Theorem 1.3.5. *Let O be k-bounded permuting set of operations, that is $O(1)$-computable. Let d be an edit-distance defined by triple (O, T, C). Problem of determining, if a distance of two words is lower than some value c is NP-complete.*

Proof: We shall only modify Theorem 1.3.3. Some differences will be done in constructing M' from M. M' will have blank symbols only on one side of the input word. Moreover, the number of these blank symbols is limited to number n. The initial configuration (for word u) is $Z_b q_0 u B^n Z_e$, where Z_b, Z_e are endmarkers, (such as endmarkers in the linearly bounded automaton, i.e. head of the machine can not go beyond them and they cannot be replaced with other characters). We can clearly assume that only the accepting configuration of M' on the word u is the configuration $Z_b z^{n+|u|} q_f Z_e$ where z is a special symbol and q_f is the unique accepting state of M'. If M' does not have enough place to do its computation, then it will not accept the word, otherwise it will accept those words that are accepted by M on less than d steps (see Theorem 1.3.3). For finite Σ (which is union of the tape alphabet of M' and letters representing the states of M') there exist an integer k, such that every member of Σ can be uniquely encoded to some member of set W, which must safisfy:

(i) W is sub-set of $\{0,1\}^{2k}$.

(ii) For every $w \in W$ holds that $\#_0(w) = \#_1(w) = k$, where $\#_a(w)$ means number of occurences of letter a in word w

(iii) There exists small integer $2 < l < \frac{n}{4}$ (for example we can take $l = 4$) such that $(\forall w \in W)$ $([w]_l = 1^l)$

(iv) For every $w \in W$, the prefix of w of length l is the only sub-word of w that is equal to 1^l

(v) For every $w \in W$, the last letter of w is 0

(vi) $|W| \geq |\Sigma|$

When encoding the members of Σ as the members of W, from any contiguous part of the tape of the machine M' we can unambiguously decide, where are the members of W located (those, that are whole included in the part). Now, all operations defined in Theorem 1.3.3 that are not dealing with the end of the tape (not widening the tape) are permuting and $4k$-bounded. We do not need other operations (neither insertions and deletions).

The only problem is that we do not know exactly, how much place M' needs for accepting the word u. To solve this, we start with computing distances between words $Z_b q_0 u B^n Z_e$ and $Z_b z^{n+|u|} q_f Z_e$ for $n = 1, 2, 3, \ldots$ If M accepts u in less than d steps, then an integer n is found in polynomial time, such that the distance between these two words will be a finite number, otherwise it will be still $\perp$ value. $\square$

What is important, the hardness of this problem does not lie in the abstractness of this model, but problem is hard even for some concrete operations. For example, in [3] it is shown, that solving even an easier problem (edit-distance on permutations) is NP-hard once we are considering such operations as a reverse of sub-string.

1.4 Other measures

1.4.1 Fragment distance

Let us consider the following situation: We are given a long string v (long word) and a "query" u. Our task is to decide, whether u contains patterns similar to v. Let us look for some instance of the edit-distance that could help us to solve this properly. Once v has patterns similar to u (for example, v can contain u as a sub-word) we want the distance to be constant. For example, the distance between u and $v.u$ should be some low value independent of v. This can be achieved by calculating an edit-distance $d(u.v, u)$ if we have zero cost of deletions. But once we have zero cost deletions we can obtain anything from sufficiently long and "rich" text, which destroys any structure of v.

Notation 1.4.1. *Let $u \in \Sigma^*$. By $[u]_i^j$ we denote the sub-word of a word u that starts at the position i and ends at the position $j-1$, i.e. $u_i.u_{i+1} \dots u_{j-1}$. If $i \geq j$ then $[u]_i^j$ is ϵ. Since in many related articles the notion of a sub-word is used in the same meaning as LCS, i.e., as "scattered" sub-word. For our contiguous sub-word we shall use the term "fragment".*

Using this notation we shall easily define fragment distance.

Definition 1.4.1. *Let $u, v \in \Sigma^*$. $D_f(u, v)$ means a fragment distance of u from v and let it be equal to the minimal n, for which there exist indeces $a_i, b_i, 0 \leq i \leq n-1$ such that $u = v_{a_0}^{b_0}..v_{a_{n-1}}^{b_{n-1}}$. If such n does not exist then the fragment distance of u from v is ∞.*

Our first observation is that if u contains some letters that are not included in v, then the fragment distance is automatically ∞. If this case does not happen, then we can construct u using up to $|u|$ fragments (in the worst case, each letter is one fragment). But even when we compute the fragment distance of a random query from a random text that is much longer than the query, we can expect distance much lower than $|u|$. Now, we present an algorithm for computing the fragment distance. We start by an easy lemma:

Lemma 1.4.1. *Let $v, u, w \in \Sigma^*$ be words such that u is a prefix (or a suffix) of w. Then $D_f(u, v) \leq D_f(w, v)$*

Proof: If we can assemble w using some number of fragments, then we can surely assemble any prefix (or suffix) with at most the same number of fragments. $\square$
Now, we present a greedy algorithm that computes the fragment distance between words u, v.

Algorithm 1.4.1.

Input: *words* $u, v \in \Sigma^*$

Output: *the fragment distance* $D_f(u, v)$

(i) Find such i, j that $[v]_i^j$ is a prefix of u and $j - i$ is maximal.

(ii) If $[v]_i^j = \epsilon$ return ∞ else take such $s \in \Sigma^*$ that $[v]_i^j.s = u$ and return $1 + D_f(s, v)$

Proof: When computing $D_f(u, v)$ we are trying to assemble the word u from the fragments of v. If u_p is the first fragment used to build u and $u = u_p.u_s$ then $D_f(u, v) = 1 + D_f(u_s, v)$. Lemma 1.4.1 clearly shows, that we can use the greedy approach and take u_p as long as possible. □

When computing $D_f(u, v)$ we must find a partitioning of u into $u_p.u_s$ such that u_p is fragment of v. The first approach can be as follows: We start with $i = 1$. Now we shall be increasing the value of i until $[u]_i$ is a fragment of v. After finding the maximal prefix, we shall remove it from u and repeat the whole process with u_p. This approach is quite inefficient. Let us solve another task: finding out whether the whole u is a fragment of v. This can be done in $O(|v|)$ time, using Knuth-Morris-Pratt algorithm. The point is to construct a deterministic finite automaton A that accepts all words which contain u as a sub-word. We can construct this automaton in $O(|u|)$ time. For example, in the Figure 1.1 we can see the deterministic finite automaton that accepts all words over the alphabet $\{a, b\}$ that contain sub-word *abaabab*.

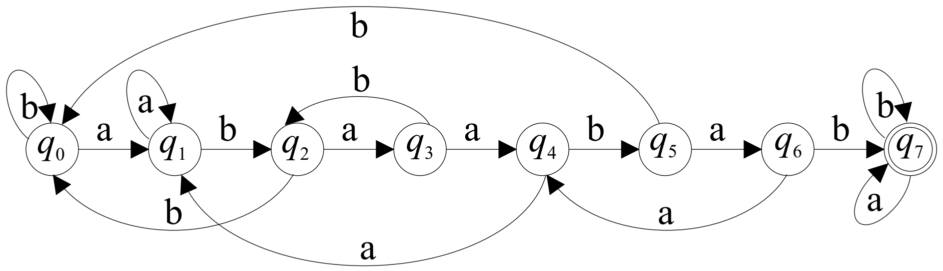

Figure 1.1: Deterministic automaton for string matching problem

If such automaton accepts v then the whole u is included in v and we have no problem. Let us assume that we have already read i letters from v and (after processing the i-th letter) we are in the j-th state of A (states are enumerated in the same way as in the example in Figure 1.1). That means, $[v]_{i-j-1}^i = [u]_{j-1}$. This helps us to find the longest prefix of u that is a sub-word of v. We shall construct the automaton A and try if it accepts

v. During the computation, we shall remember the maximal state (the state with the maximal index) that is reached and also, what letter (its position in v) was processed as the last when getting into this state. After (accepting or non-accepting) computation of A on the word v we know the longest prefix of u that is a sub-word of v which is exactly what we wanted. For the time complexity of computing $D_f(u, v)$ we have: we must search $D_f(u, v)$ times whether some suffix of u is a sub-word of w which can take $O(|v|)$. So, the whole computation can take $O(|v|.D_f(u, v))$. This form is not quite good, because we are expressing the time complexity by the result of our computation. The only way we can eliminate $D_f(u, v)$ in this formula is to approximate it by $|u|$. Now the formula for time complexity is $O(|v|.|u|)$. It can be quite surprising that we can compute this distance as fast as the basic edit-distance.

1.4.2 Fragment distance with costs

When computing the fragment distance, we are interested only in the minimal number of fragments needed to assemble the word that we need. It is natural to count not only the number of these fragments, but also the sum of the costs of them. In this thesis, the cost function $C : \mathbb{N}^+ \to \mathbb{R}^+$ will be the function of the length of the fragments only. Before we define fragment distance with costs, let us see some auxiliary definitions first.

Notation 1.4.2. *Let $f : \mathbb{N} \to \mathbb{R}$ be a function. By $\Delta(f)$ we denote the forward difference of f, i.e. $\Delta f(x) = f(x + 1) - f(x)$.*

Definition 1.4.2. *Let $f : \mathbb{N} \to \mathbb{R}$ be a function. We call the function f convex (concave, non-convex, non-concave) if Δf is increasing (decreasing, non-increasing, non-decreasing) on the whole interval of definition.* [5]

Now, the fragment distance with costs can be defined as follows:

Definition 1.4.3. *Let $u, v \in \Sigma^*$ and $C : \mathbb{N}^+ \to \mathbb{R}^+$ be the cost function which satisfies:*

(i) *C is non-decreasing*

(ii) *$C(0) = 0$*

(iii) *C is non-convex function*

[5]Note that likewise the non-decreasing function does not mean not decreasing function, the non-concave function does not mean function that is not concave.

Fragment distance with costs of u from v is equal to the minimal S, such that there exists a non-negative integer n and indices $k_i, l_i, 0 \leq i \leq n-1$ such that $u = v_{k_0}^{l_0}..v_{k_{n-1}}^{l_{n-1}}$ and $S = \sum_{i=0}^{n-1} C(l_i - k_i)$. If such n, k_i, l_i do not exist then the fragment distance with costs of u from v is ∞.

Now we want to explain, why applying conditions (i)-(iii) to the function C is necessary. All these conditions are necessary if we want to get reasonable results. It would be quite strange if using of shorter fragment was more expensive than using some long fragment, so (i) must hold. Also (ii) is a reasonable condition, using ϵ does not change the assembled word, so it should bear not cost. We can also argue, that the function C should be non-convex (iii). Let us discuss what would happen, if C was convex. Let n be length of one fragment that was used to assemble u. Cost of using this fragment is $C(n)$ and recalling (ii) we can write:

$$C(n) = (C(1) - C(0)) + (C(2) - C(1)) + \cdots + (C(n) - C(n-1)) =$$
$$\Delta C(0) + \Delta C(1) + \cdots + \Delta C(n-1) > n\Delta C(0) = n.C(1)$$

From this it is clear, that using the fragment of length 1 n times is more profitable than using a big fragment with length n. This will lead to the fact that $D_f(u, v)$ can be equal to only two values, either ∞ (if u can not be assembled) or $|u|.C(1)$ if we use fragments of length 1. We certainly do not want the fragment distance to have such property.

On the other hand, concave functions work quite well. Concaveness of C causes that the fragment distance works in a way we want: It tends to search for long common fragments and use them. When two fragments concatenated are equal to u (so the sum of their lengths is $|u|$), then one segment tends to be as long as possible and the other one does the rest. Good (concave) cost function is for example $C(x) = \sqrt{x}$.

Now we may try to adjust the greedy algorithm for computing the fragment distance to compute the fragment distance with costs. Following the analysis we have done, we may think that when we find the longest common fragment, we have fragment that is surely used when assembling u. If this was true, we could delete this fragment from u and repeat the search for the longest common fragment, so we could write very effective greedy algorithm. Sadly, as the next example shows, the longest common fragment is not necessarily used.

Example 1.4.1. *$u = abcdefghijkl$ $v = v_1.v_4.v_2.v_5.v_3.v_6$, where*
$v_1 = abcde$, $v_2 = fg$, $v_3 = hijkl$
$v_4 = abc$, $v_5 = defghi$, $v_6 = jkl$
Now, it is clear, we have only two ways, how to choose fragments, so that

after concatenation they will be equal to u. It holds, that $u = v_1.v_2.v_3$ and $v = v_4.v_5.v_6$.
Let us assume, that $C(x) = \sqrt{x}$, which is an increasing concave function with $C(0) = 0$. The cost of $u = v_1.v_2.v_3$ is approximately 5.06, the cost of the second alternative is approximately 5.91 which is larger despite the fact that only this alternative contains the longest common fragment (v_5).

So, as this example illustrates, when computing the fragment distance with costs, we cannot use the greedy algorithm - or at least not such a simple greedy algorithm. Once again, the dynamic programming will help us.

Algorithm 1.4.2.

Input: *words $u, v \in \Sigma^*$*

Output: *the fragment distance with costs $D_f(u, v)$*

We shall construct the function $A : \mathbb{N}^* \to \mathbb{R}^+$ so that

$$A(i) = D_f([u]_{|u|+1-i}^{|u|+1}, v)$$

In other words, $A(i)$ tells us, how far is suffix of length i of u from v. Trivially $A(0) = 0$. The value $A(i)$ can be easily computed for $i > 0$ using the values $A(j), j < i$. We can compute it using the formula

$$A(i) = \min_{\substack{0<l\leq|u| \\ i-l\geq 0}} \{C(l) + A(i-l) \mid$$
$$(\exists k \in \mathbb{N})\ ((0 \leq k < k + l \leq |v|) \wedge ([u]_i^{i+l} = [v]_k^{k+l}))\}$$

Correctness of this algorithm is quite clear and can be easily proved by complete induction on the function A. When computing $A(i)$ we assume that all values $A(j)$ for $j < i$ are computed correctly. Now, the first letter of $[u]_{|u|-i}^{|u|}$ must get to its place in a fragment of some length. We try all possibilities and take one with the minimal cost, so we surely compute $A(i)$ correctly, which is the proof of the induction step in our proof. □
Now, let us think about the time complexity of Algorithm 1.4.2. When computing the fragment distance (without costs) we showed, how we can find the longest prefix of u that is also the fragment of v. Once we have found it (with length l), then we have automatically found the prefixes (and fragments simultaneously) of all lengths that are smaller than l. So, this is the way, how the minimization can be done in $O(\min\{|u|, |v|\})$ time, which is not more than time $O(|v|)$ (needed for Knuth-Morris-Pratt algorithm). So, the total time complexity is $O(|v|.|u|)$.

1.4.3 Fragment distance with editing

The fragment distance has some inconvenient features. When computing $D_f(u, v)$, the first problem is, when u contains letters that are not included in v. This problem can be solved when instead of $D_f(u, v)$ we would be computing $D_f(u^v, v) + |u| - |u^v|$ where u^v is the word u without all letters that are not in v. After computing the fragment distance of the modified u and v, the number of these letters is added to the result as some penalization (we can estabilish some penalty function here). Although this technique solves our problem, it is not a natural way how to cope with this problem. The following example reveals more problems: Let us have two words u, v. Suppose the whole u is a fragment of v, so $D_f(u, v) = 1$. Now, we add some letters inside v, so u will not be the fragment of v any more. In the worst case, adding n letters forces us to use $n + 1$ fragments, which is quite a lot. A similar problem occures, when we delete n letters. In these cases, we use many fragments just to simulate basic edit-operations such as insertions and deletions. But according to our intuition, using long fragment is a much more "significant" operation than just the insertion (deletion) of one letter.

Definition 1.4.4. *Let u, v be words in Σ^*. By the fragment distance with editing of u from v we shall mean the cost of the cheapest multi-operation, that transforms ϵ into u. To do so, we can use operations of two types:*

(i) *Inserting or deleting a letter with cost c.*

(ii) *Inserting some fragment v_f of v to any place of the assembled word, with cost $C(|v_f|)$, where C is a cost function, satisfying the three conditions from Definition 1.4.3*

The fragment distance with editing will be denoted by $D_{fe}(u, v)$

In spite of the fact, that this distance is not an instance of edit-distance (we are transforming ϵ to u, not u to v), we can naturally widen Definition 1.3.4 also to this case. Now we can see that:

Lemma 1.4.2. *Considering the fragment distance with editing, any multi-operation o can be aligned in order: insertions of fragments, deletions, insertions (of letters).*

Proof: We can make o (partially) cheaper if it will delete only letters that were inserted as a part of the bigger fragment, so deletions can be done before insertions. Insertion of fragments has no effect on possibility of inserting or deleting some letter, so this can be done at the beginning. □
This gives us another way, how to define the fragment distance with editing.

Theorem 1.4.1. *The following equation holds for the fragment distance with editing :*

$$D_{fe}(u,v) = \min\{c|(\exists w \in \Sigma^*)\ c = D_f(w,v) + D_e(w,u)\}$$

where D_f is the fragment distance with the same cost function C as we are using when computing D_{fe}, and D_e is such an edit-distance, where insertions and deletions have the same cost as insertions and deletions in D_{fe}.

Proof: From Lemma 1.4.2 we know that any multi-operation o can be aligned in order: insertions of fragments, deletions, insertions (of letters). Let o' be a multi-operation that contains only insertions of fragments from o. Let $w = T(o', \epsilon)$. Now $D_{fe}(u,v) = D_f(w,v) + D_e(w,u)$. Word w we found is clearly the one, for which $D_f(w,v) + D_e(w,u)$ is minimal. $\square$

Although this theorem is interesting, it does not lead us to an effective algorithm to compute this distance. We shall again use the dynamic programming algorithm:

Algorithm 1.4.3.

Input: *words $u, v \in \Sigma^*$*

Output: *the fragment distance with editing $D_{fe}(u,v)$*

First, we shall construct the function $E : \mathbb{N}_0^{|u|} \times \mathbb{N}_0^{|u|} \times \mathbb{N}_0^{|v|} \times \mathbb{N}_0^{|v|} \to \mathbb{R}$, such that $E(i,j,k,l) = D_e([u]_i^j, [w]_k^l)$, where D_e is the same distance as it was in Theorem 1.4.1, so we already know, how to compute it. Now, we can construct the function $A : \mathbb{N}_0^{|u|} \times \mathbb{N}_0^{|u|} \to \mathbb{R}$, such that $A(i,j) = D_{fe}([u]_i^j)$. First, it is clear, that $\forall i\ A(i,i) = 0$, because distance ϵ from anything is trivially 0. Now, we have:

$$\begin{aligned} A(i,j) = \min\{c + A(i+1,j)\} \cup \{t|\ \exists\ k,l,m \in \mathbb{N}\ i < m \leq j \\ \wedge\ t = C(k-l) + E(i,m,k,l) + A(m,j)\} \end{aligned} \quad (1.2)$$

Having computed A, we have $D_{fe}(u,v) = A(0, |u|+1)$.

We shall prove the correctness of the algorithm by induction on $j - i$. The trivial case $j = i$ is evidently well assigned, so let us focus on the recursive relation for $A(i,j)$. Let us focus on how the letter u_i gets to its place. The first possibility is, that this letter is inserted. Now, the rest of the sub-word we want to assemble ($[u]_{i+1}^j$) must be assembled as cheaply as possible - From the induction hypothesis, this is exactly the value $A(i+1,j)$. The second possibility is, that u_i comes in some larger block and it is at the end of it.

Now, this block is inserted and edited, so it gets the whole sub-word $[u]_i^m$ to its place. Once again, the rest of $[u]_i^j$ (it actually is $[u]_m^j$) must be assembled as cheaply as possible. We discussed all possibilities, how u_i can get to its place and we are taking the cheapest one, so this proves the induction step. $\square$

Now to the time complexity of the algorithm. When computing the function E we must compute $O(|u|^2.|v|^2)$ times the edit-distance of two substrings. The lengths of these substrings are at most $|u|$ and $|v|$, so for the total complexity of computing E we have

$$O(|u|.|v|).O(|u|^2.|v|^2) = O(|u|^3.|v|^3)$$

Next, we must compute $O(|u|^2)$ values of the function A. To detain one value, we must try all possibilities for k, l, m in equation 1.2. k, l are referring to some positions in the word v, so we have $O(|v|^2)$ possibilities how to choose them. For k we have $j - i$ possibilities, what is less then $|u|$. Putting all this together, for complexity of computing A we have

$$O(|u|^2).O(|u|.|v|^2) = O(|u|^3.|v|^2)$$

Note, that the complexity of computing E is higher, so we can say, that the total complexity of the algorithm is the same as the complexity of computing E which is $O(|u|^3.|v|^3)$. Note also, that it is good to compute all values of E in the beginning. If we computed a value of E every time we need it (and forget it afterwards), the time complexity will be even higher.

The complexity of Algorithm 1.4.3 is quite high. It is hard to imagine, how to use an algorithm with time complexity $O(n^6)$ (for $|u| = |v| = n$), when even the basic edit-distance with $O(n^2)$ is too slow in some applications. Now we present an idea, how this algorithm can be improved. First, when talking about fragment distance, we say that we treat u as not a very long word and we try to assemble it from fragments of some longer text v. This asymmetry was not very important until now. Of course, D_f or D_fe are non-symmetric functions, but we did not try to minimize the computation complexity due to the parameter $|v|$ more than due parameter $|u|$. Situation will be quite different now. We shall try to minimize the exponent of $|v|$ more than the exponent of $|u|$, for example the complexity $O(|u|.|v|)$ will be treated as much worse complexity, as $O(|u|^2)$. This view gives us some possibilities, how to make the algorithm work more effectively. When u is small comparing to v, then it is sure, that only small fragments (maximal length can be roughly estimated as length of u) will be used for assembling u. In the following text, we shall try to find a maximal length of a fragment that we are using.

Lemma 1.4.3. *Let $u, w \in \Sigma^*$, $|w| > |u|$. Consider the longest common subsequence of words u and w. There exists a fragment of w of length at least $\lceil \frac{|w|-|u|}{|u|+1} \rceil$ which does not contain any occurence of a letter from the longest common subsequence.*

Proof: The length of the longest common subsequence (LCS) is at most $|u|$, so in w there must be at least $|w| - |u|$ letters, that are not included in LCS. Letters of LCS divide w to at most $|u| + 1$ sub-words (they may be empty). The statement in the lemma clearly follows using the Dirichlet's principle. $\Box$

Now, we shall be using the properties of the function C.

Lemma 1.4.4. *Let C be a non-decreasing non-convex function with $C(0) = 0$. Then*

$$\lim_{n\to\infty} \frac{C(n) - 2.C(\lceil \frac{n}{2} \rceil)}{n} = 0$$

or equivalently,

$$C(n) - 2.C(\lceil \frac{n}{2} \rceil \in o(n)$$

Proof: Let $\Delta C(n) = C(n+1) - C(n)$. C is non-decreasing, so $\Delta C(n) \geq 0$, but C is non-convex, so $\Delta C(n)$ is non-increasing. Let $U(n) = \frac{C(n)}{n}$ and $L = \lim\limits_{n\to\infty} U(n)$. The question is, whether this limit exists. We know, that the function U is positive and as we shall see it is also non-increasing. This two properties are enough to say that L must exist. To verify that U is non-increasing we can write:

$$U(n+1) - U(n) = \frac{C(n+1)}{n+1} - \frac{C(n)}{n} = \frac{C(n+1).(n) - C(n).(n+1)}{n.(n+1)}$$

For $n > 0$ the denominator of this fraction is positive, so it suffices to analyze the sign of the numerator:

$$\begin{aligned}
&C(n+1).(n) - C(n).(n+1) = n.(C(n+1) - C(n)) - C(n) = \\
&n.\Delta C(n) - \Delta C(0) - \Delta C(1) - \cdots - \Delta C(n-1) = \\
&(\Delta C(n) - \Delta C(1)) + (\Delta C(n) - \Delta C(2)) + \cdots + (\Delta C(n) - \Delta C(n-1))
\end{aligned}$$

From the fact, that ΔC is non-increasing it follows, that all addends are non-positive, so the difference of U is non-positive.

Once we know, that L exists, we can split the proof into two cases:
$L = 0$. Now,

$$\lim_{n\to\infty} \frac{2.C(\lceil \frac{n}{2} \rceil)}{n} = L = 0$$

This automatically proves what we wanted.

Now to the case $L \neq 0$. Let $C(n) = R(n) + n.L$. Now, R has some properties: Surely, $R(0) = 0$. Also,

$$\Delta R(n) = R(n+1) - R(n) = \Delta C(n) + L$$

From this we see, that if ΔC is a non-increasing function, then ΔR is also a non-increasing function. Moreover,

$$\lim_{n\to\infty} \Delta R(n) = \lim_{n\to\infty} \Delta C(n) - L = 0$$

Now we have:

$$\frac{2.C(\lceil\frac{n}{2}\rceil) - C(n)}{n} = 2.R(\lceil\frac{n}{2}\rceil) + 2.\frac{n}{2} - R(n) - n = 2.R(\lceil\frac{n}{2}\rceil) - R(n)$$

for R holds that

$$\lim_{n\to\infty} \frac{R(n)}{n} = \lim_{n\to\infty} \frac{C(n) - L.n}{n} = 0$$

so we can finish the proof in similar way as the case before. $\square$

In the next theorem we shall see the reason for having the above two lemmas:

Theorem 1.4.2. *There exist an integer k such that all fragments used by Algorithm 1.4.3 to assemble the word u have length at most $k.|u|$.*

Proof: Let us assume we are computing the distance of words u and v and w is some fragment of v we want to use. If w is too long, then there must exist some letters that are deleted and so they are not used in assembling u. Now from Lemma 1.4.3 we know that, in w there exists a fragment with length $\Theta(\frac{|w|}{|u|})$. Let us denote this fragment as f and let w_p, w_s be the prefix and the suffix of w such that $w = w_p.f.w_s$. If we want to use the fragment w we must first insert it with cost $C(|w|)$ and then delete (at least) the fragment f so the cost of the whole operation will be at least $C(|w|) + |f|.c$ where c is the cost for insertion or deletion. However, the same effect can be achieved, when w_p is inserted first and then w_s with total cost $C(w_p) + C(w_s)$. Now we know that $|w_p| + |w_s| \leq |w|$ and also from non-convexness of C we know that $C(w_p) + C(w_s) \leq 2.C(\lceil\frac{|w|}{2}\rceil)$. So, the difference between costs of these two possibilities is less than $2.C(\lceil\frac{|w|}{2}\rceil) - C(|w|) + |f|.c$ When we use Lemma 1.4.4 we have that $2.C(\lceil\frac{|w|}{2}\rceil) - C(|w|) \in o(1)$ and using Lemma 1.4.3 we have $|f| = \Theta(\frac{|w|}{|u|})$. So the difference of costs is

$$DC(|w|, |u|) = o(1) - \Theta(\frac{|w|}{|u|}).c = o(1) - \Theta(\frac{|w|}{|u|})$$

If the value of DC is positive, then inserting w as a whole is preferable, if the value is negative, then surely the second possibility (inserting w_p and then w_s) is better. From the last equation we know that there must exist positive constants c_1, c_2 such that $DC(|w|, |u|) \leq c_1 - c_2.\frac{|w|}{|u|}$. This expression will be negative, if $|w| \geq \frac{c_1.|u|}{c_2}$, so $k = \lceil \frac{c_1}{c_2} \rceil$ is exactly the constant we are looking for. □

From the last theorem it is clear, that we can improve the first part of Algorithm 1.4.3 to work in time $O(|v|.|u|^5)$ and the second part to $O(|u|^4.|v|)$, so the total complexity is $O(|v|.|u|^5)$. The time complexity of the algorithm is very sensitive to the length of the word we want to assemble. In case this word is not very long, we can use the algorithm to effectively find the distance from long strings.

1.4.4 Δ-similarity

This similarity measure was introduced in [1]. Motivation for defining Δ-similarity is obvious - if $d(u, v) = 2$ (d stands for the basic edit-distane), and $|u| = |v| = 2$, then these words do not seem very similar. On the other hand, if $|u| = |v| = 1000$, then u, v look quite similar. In [1], Δ-similarity was defined in a similar way like Definition 1.4.5. The original definition is not included here, because it contained small inaccuracies. Our definition is here:

Definition 1.4.5. *Δ-similarity is a function* $\Sigma^* \times \Sigma^* \rightarrow < 0, 1 >$*, such that:*

$$
\begin{aligned}
&(\forall\ a \in \Sigma \cup \epsilon)\ \Delta(a, a) = 1 && (1.3)\\
&(\forall\ u, v \in \Sigma^*)\ (u = \epsilon\ XOR\ v = \epsilon) \Rightarrow \Delta(u, v) = 0 && (1.4)\\
&(\forall\ a, b \in \Sigma)\ (\forall\ u, v \in \Sigma^*)\ \Delta(au, bv) = \frac{1}{|aubv|} \cdot \max\{ && \\
&\Delta(a, b) + \Delta(u, v).|uv|, && (1.5)\\
&\Delta(au, v).|auv|, && (1.6)\\
&\Delta(u, bv).|ubv|\} && (1.7)
\end{aligned}
$$

Although this definition was used in [1], we shall use an equivalent definition, that is more practical.

Definition 1.4.6.

$$(\forall u, v \in \Sigma^*)\ \Delta(u, v) = \frac{2 \cdot |LCS(u, v)|}{|u| + |v|}$$

where LCS means the longest common subsequence (see Definition 1.3.5).

Theorem 1.4.3. *Definitions 1.4.5 and 1.4.6 are equivalent.*

Proof can be easily done by induction. When $|u| = 0$, $|v| = 0$ or $|v| = |u| = 1$, both two definitions do trivially the same (in Definition 1.4.5 we use only the first two formulas (1.3 and 1.4) to compute the result). Now let us assume that definitions do the same for all u, v such that $|u| + |v| < n$. Now we prove that definitions do the same also for all u, v such that $|u| + |v| = n$. If u or v is ϵ, then it is a trivial case and we have already proven it. If not, we can write $u = a.u'$, $v = b.v'$ and use 1.5, 1.6 and 1.7 for computation of $\Delta(u, v)$. Now, $|u'| + |v'| + 2 = |u| + |v|$ so for all Δ expressions in the first definition in 1.5, 1.6 and 1.7 we can use Definition 1.4.6. Now we can write:

$$\frac{\Delta(au', bv') \cdot |au'bv'|}{2} = \frac{1}{2}\max\{2 \cdot \Delta(a, b) + 2 \cdot |LCS(u', v')|,$$
$$2 \cdot |LCS(au', v)|, \;\; 2 \cdot |LCS(u, bv'|\} =$$
$$\max\{\Delta(a, b) + |LCS(u', v')|, \;\; |LCS(au', v)|, \;\; |LCS(u, bv'|\}$$

Now we see that $\frac{\Delta(au',bv') \cdot |au'bv'|}{2}$ exactly matches the formula for computing $|LCS(au', bv')|$ so we can write

$$\frac{\Delta(au', bv') \cdot |au'bv'|}{2} = |LCS(au', bv')| \Rightarrow \Delta(u, v) = \frac{2 \cdot |LCS(u, v)|}{|uv|}$$

which proves the induction step of our proof. □

The following lemmas were proved in [1]. We do not need them directly, they are presented here to illustrate how easy they can be proven using the equivalent Definition 1.4.6

Lemma 1.4.5. *Given $x, y, u, v \in \Sigma^*$ the following inequality holds:*

$$\Delta(xu, yv) \geq \frac{\Delta(x, y)|xy| + \Delta(u, v)|uv|}{|xyuv|}$$

Proof: We use Definition 1.4.6 and we get an equivalent inequality:

$$|LCS(xu, yv)| \geq |LCS(x, y) + LCS(u, v)|$$

which holds trivially. □

Lemma 1.4.6.

$$(\forall u_1, u_2, v \in \Sigma) \; \Delta(u_1u_2, v) = max_{v=v_1v_2}\{\Delta(u_1, v_1)|u_1v_1| + \Delta(u_2, v_2)\}$$

Proof: Using Definition 1.4.6 we can re-write the lemma to its equivalent form:

$$(\forall u_1, u_2, v \in \Sigma)\ LCS(u_1u_2, v) = max_{v=v_1v_2}\{LCS(u_1, v_1) + LCS(u_2, v_2)\}$$

Now, this is a trivial property of LCS. □

Lemma 1.4.7.

$$(\forall a \in \Sigma\ \forall u, v \in \Sigma^*)\ \Delta(au, av) = \frac{2 + \Delta(u, v)|uv|}{|auav|}$$

Proof: Once again, in the notation of LCS we are saying only that

$$LCS(au, av) = 1 + LCS(u, v)$$

which is not surprising at all. □

Lemma 1.4.8.

$$(\forall x, u, v \in \Sigma^*)\ \Delta(xu, xv) = \frac{2|x| + \Delta(u, v)|uv|}{|xuxv|}$$

Proof: Same as in Lemma 1.4.7

$$LCS(xu, xv) = |x| + LCS(u, v)$$

□

Definition 1.4.7. *Let $u, u_1, u_2 \ldots, u_n \in \Sigma^*$ be words. We call $\{u_i\}_{i=1}^n$ a partition of u iff $u = u_1 \ldots u_n$.*

Definition 1.4.8. *Let $\{u_i\}_{i=1}^n$, $\{v_i\}_{i=1}^n$ be partitions of words u, v respectively. We call $\{v_i\}_{i=1}^n$ Δ-corresponding partition to partition $\{u_i\}_{i=1}^n$ iff:*

$$\Delta(u, v) = \frac{\sum_{i=1}^n \Delta(u_i, v_i).|u_iv_i|}{|uv|}$$

Lemma 1.4.9. *Let $u, v \in \Sigma^*$ be words and $\{u_i\}_{i=1}^n$ be partitions of word u. Then there exist a partition of v Δ-corresponding to $\{u_i\}_{i=1}^n$.*

Proof: Using Definition 1.4.6 we have that $\{u_i\}_{i=1}^n, \{v_i\}_{i=1}^n$ are Δ-corresponding iff

$$LCS(u, v) = \sum_{i=1}^{n} LCS(u_i, v_i)$$

Let us choose the partition $\{v_i\}_{i=1}^n$ such that v_i contains all letters that are corresponding with u_i due to $LCS(u, v)$ (all letters from u_i used in $LCS(u, v)$ have their pairs in v_i and vice versa). From this it is immediately clear, that

$$\Delta(u, v) \leq \frac{\sum_{i=1}^{n} \Delta(u_i, v_i).|u_i v_i|}{|uv|}$$

Moreover, $LCS(u, v)$ indicates existence of some common subsequences in any two sub-words of u, v. If we look at the common subsequence induced by $LCS(u, v)$ in words v_i, u_i, this must be the longest one, if not, we shall be able to use it for constructing a longer common subsequence of u, v than $LCS(u, v)$ is. □.

Lemma 1.4.10.

$$(\forall u, v \in \Sigma^*)\ \Delta(u, v) = \Delta(u^R, v^R)$$

Proof: Lemma is straight-forward corollary of the fact $|LCS(u, v)| = |LCS(u^R, v^R)|$ □

Lemma 1.4.11. *For Δ similarity it holds:*

(i) *Reflexivity:* $(\forall u \in \Sigma^*)\ \Delta(u, u) = 1$

(ii) *Symmetry:* $(\forall u, v \in \Sigma^*)\ \Delta(u, v) = \Delta(v, u)$

Proof: First part is a consequence of $|LCS(u, u)| = |u|$, the second part is clear from $|LCS(u, v)| = |LCS(v, u)|$. □

1.4.5 Δ-similarity on random words

In this sub-section, we shall be interested in a question, how large is the Δ-similarity between two random words. Motivation for the interest in this question is quite clear: Assume, that having two words u, v of length 1000 over the alphabet $\{a, b\}$ we know that $\Delta(u, v) = 0.8$. At a first sight, words with similarity 0.8 should be really similar. But in fact, long random words upon alphabet of size 2 have their Δ-similarity about 0.8. With this fact, words u, v do not look similar at all. So when talking about Δ-similarity of two words, it is good to know what is the average Δ-similarity between two random words of a given length. Once we know that, we can make conclusions, how similar these two words are. Whether we are interested in Δ-similarity or the basic edit-distance, we can reduce our questions to the expected value of $|LCS|$ between two random words.

This problem was first studied in [7] and then in many other publications, from which many are citing [7]. In these publications many interesting results can be found, mostly based on non-trivial combinatorical reasonings. For these reasons we present only few estimations in this work, that are interesting for a reason, that we do not need to build such complex machinery as in [7] or [6]. We focus on alphabet of size 2 and words of equal length.

Definition 1.4.9. *Let $u, v \in \{a, b\}^*$, $|u| = |v| = n$, be two random words. Let $E(x)$ mean the expected value of x. Now*

$$\gamma_n = E(\Delta(u, v))$$

Using Definition 1.4.6 we can write

$$\gamma_n = E\Big(\frac{|LCS(u, v)|}{n}\Big)$$

Theorem 1.4.4. *There exist limit $\gamma = \lim_{n\to\infty} \gamma_n$. Moreover*

$$\gamma = \sup\{\gamma_n\}$$

Proof can be found in [6]. $\square$ We shall be interested in some estimations of value γ. We can immediately say that

Theorem 1.4.5.

$$\gamma \geq \frac{1}{2}$$

Proof: For every $0 \leq i \leq n-1$ the equation $u_i = v_i$ (u_i is the i-th letter of u) holds with probability of $\frac{1}{2}$. If we reduce function f which determine LCS (see the note after Definition 1.3.5) only to $f(i) = i$ or $f(i) = \perp$, we automaticaly have common subsequence with average length $\frac{n}{2}$. $\square$

We shall improve this estimate in the following theorem:

Theorem 1.4.6.

$$\gamma \geq \frac{3}{4}$$

Proof: We shall prove this theorem by finding an algorithm that constructs a common subsequence that takes $\frac{3}{4}$ of the whole length of a word.

Algorithm 1.4.4.

Input: *Words* $u, v \in \{a,b\}^*$, $|u| = |v| = n$

Output: *A common subsequence with average length* $\frac{3}{4}$ *of word length (for large n)*

(i) Set $i = 0$, $j = 0$

(ii) If $i > n$ or $j > n$ then terminate the algorithm.

(iii) If $u_i = v_j$ then add letters u_i, v_j to LCS, set $i = i+1$, $j = j+1$ and go to step (ii)

(iv) If $u_i \neq v_j$ and $u_i = v_{j+1}$ then add the letters u_i to LCS, set $i = i+1$, $j = j+2$ and go to step (ii)

(v) Set $i = i+1$, go to step (ii)

Now let us prove, that the common subsequence found by this algorithm will be as long as we want it to be. When the algorithm comes to step (iii) then the current prefix of LCS increases by 1 with probability $\frac{1}{2}$. If this does not happen (probability $\frac{1}{2}$), we shall try to skip one letter in v and find a match there (probability $\frac{1}{2}$). So this happens with total probability $\frac{1}{4}$. With probability $\frac{1}{4}$ none of this happens, we increase i (step (v)) and we do not increase the current prefix of LCS. When we look at the average case, between two consecutive executions of step (ii) we increase the i by $\frac{1}{2}\cdot 1 + \frac{1}{4}\cdot 1 + \frac{1}{4}\cdot 1 = 1$ and j by $\frac{1}{2}\cdot 1 + \frac{1}{4}\cdot 2 + \frac{1}{4}\cdot 0 = 1$. Size of the current prefix of LCS is increased by expected value $\frac{1}{2}\cdot 1 + \frac{1}{4}\cdot 1 + \frac{1}{4}\cdot 0 = \frac{3}{4}$. This proves our goal, but we must solve some technical issues first.

The problem occures, if one of the values i, j, reaches end of the word and the other word is not processed yet. If u was infinitely long, then the common subsequence constructed by our algorithm is of the length x, where x is a random value with the expected value $E(x) = \frac{3}{4}n$ and normal ditribution (for large n). If v was infinitely long, then the common subsequence is of length y, where y is a random value from the same distribution as x. One word (the one that is not used to its full length) looks to us as if it was infinite, so we can express the expected $|LCS|$ by $z = \min\{x,y\}$. We can express z as $z = x - (x - \min\{x,y\})$. We know, that $E(x - \min\{x,y\})$ is linearly dependent on $\sqrt{D}$ where D is a dispersion of the distribution, which is linearly dependent on n, so the whole expression is linearly dependent on $\sqrt{n}$ and thus $\lim\limits_{n\to\infty} \frac{x - \min\{x,y\}}{n} = 0$. $\Box$.
This algorithm proves our goal. $\Box$

What is the real value of γ? The exact value is not known, but it is no problem, to turn on computer, let it generate many strings with the length for example one milion and compute γ approximately. In [6] the estimation $\gamma \in (0.8120, 0.8125)$ can be found.

Chapter 2

Projection of words to languages

2.1 Some easy but important algorithms

In this subsection we would like to present some algorithms that are not very hard to invent (and surely they already are in some literature, at least in form of exercises for readers). They will be quite important in the next subsections but not presenting them until then could harm the integrity of the text for a bit.

Definition 2.1.1. *Consider a context-free grammar $G = (N, T, P, S)$. Let A be any non-terminal from G. By G_A we shall mean a context-free grammar $G_A = (N, T, P, A)$ and let $L(G_A)$ be the language generated by this grammar. Trivially $L(G) = L(G_S)$. When no confusion is possible, we shall use just A instead of G_A or $L(G_A)$ to make our notations shorter.*

Now, we shall be interested in the length of the shortest word from some non-empty context-free language L. We shall call this value $Sht(L)$. Formally $Sht(L) = \min\{|w| | w \in L\}$. To make our notations shorter, we shall use $Sht(A)$ instead of $Sht(L(G_A))$ when it is clear what grammar G are we referring to.

We shall need to compute every $Sht(L(G_A))$ for every non-terminal $A \in N$. For this purpose, we shall use this algorithm:

Algorithm 2.1.1.

Input: *context-free grammar $G = (N, T, P, S)$ in Chomsky normal form*

Output: *$Sht(L(G_A))$ for every non-terminal $A \in N$*

The algorithm will be similar to the Dijkstra algorithm for finding the shortest path in graphs. We shall use some auxiliary set of non-terminals called H which will satisfy: In some stage of computing, for every non-terminal A it holds $A \in H \Leftrightarrow$ we already know the value of $Sht(A)$. The set H will have another useful property: If A and B are non-terminals and $A \in H \wedge B \notin H$ then $Sht(A) \leq Sht(B)$.

Initialization: We shall try to find every non-terminal A such that $Sht(A) \leq 1$. First find those non-terminals for which it holds $Sht(A) = 0$. Those non-terminals can be found by this sub-algorithm.

(init 1): set $H = \{A | A \to \epsilon \in P\}$.

(init 2): Construct the set H' by the rule

$$H' = \{A |\ \exists B, C\ A \to BC \wedge B, C \in H \wedge A \notin H\}$$

(init 3): construct the new set $H := H \cup H'$. If H did not change (i.e. H' is empty) end, else go to step init 2

Those non-terminals A for which $Sht(A) = 1$ can be found in a similar way. Once we have them, the initialization phase is over and we can do the algorithm itself.

(1) set $H = 0$

(2) For every non-terminal A from set $N - H$ find the value $Sht'(A)$ by the rule:

$$Sht'(A) = \min\{l |\ (\exists B, C \in N)\ A \to BC \wedge l = Sht(B) + Sht(C)\}$$

If the minimizing set is empty, than $Sht'(A) = \infty$. Now, let A be any non-terminal from set $N - H$ with minimum $Sht'(A)$. For this non-terminal it holds that $Sht(A) = Sht'(A)$.

(3) construct new set $H := H \cup \{A\}$. If $H = N$ end, else go to step (1)

Proof of correctness: First, we shall prove the correctness of the initialization phase. When repeating step (init 2) for the i-th time H' contains all non-terminals, such that for every non-terminal $A \in H'$ there exists a derivation tree with root A such that ϵ is derived and the height of this tree is exactly i. It is obvious, if $Sht(A) = 0$ then for a non-terminal A there must exist such tree with height at most $|N|$. If not, consider such a tree with minimal height. Then, some non-terminal is derived from itself, therefore this sequence can be omitted from the tree. Repeating of this step will lead to a contradiction with the minimal height of the tree. Therefore, our algorithm will find every non-terminals with the property $Sht(A) = 0$. When finding non-terminals with $Sht(A) = 1$, algorithm and also proof will be very similar, so we can omit them.

The proof of body of algorithm itself is also not hard. We must prove, that for a non-terminal which we find in (2) it really holds $Sht(A) = Sht'(A)$ and also, that this non-terminal is one with the minimal Sht value not known yet (i.e. it does not belong to H yet). After the initialization, these two properties hold. Now we shall discuss the induction step due to the number of repetitions of the step (2). Suppose, there is some non-terminal B for which $Sht(B) < Sht'(A)$ (actually, B and A can be the same, so this case includes all possibilities, why induction step can be wrong). Suppose that Sht value of B is minimal. Now, take the derivation tree of this B. Let $B \to CD$ be the rule which is used in this tree for B. Now, without loss of generality, two things can happen: Either $Sht(B) = Sht(D) \wedge Sht(C) = 0$ or $1 \leq Sht(C) < Sht(B) \wedge 1 \leq Sht(D) < Sht(B)$ If the first possibility occurs, we can repeat this with the non-terminal D until the second possibility occurs. If the second case occurs, let us look on C, D. If one of these non-terminals does not belong to H, then B was not a non-terminal with minimal Sht value, which is a contradiction. If B and C belongs to H, then the value $Sht(B) = Sht(C) + Sht(D) \geq Sht(A)$, which is a contradiction with the way, how we found B. $\square$

The time complexity of the algorithm can be measured according to many parameters, for example $|T|$, $|N|$ or $|P|$. We shall use the parameter $|G|$, which we define as a length of the description of G that is done in some "standard" encoding. If G does not use "useless" (non-)terminals (such (non-)terminals that are not contained in any rule, then $|G| = \Theta(|P|)$. Let us return to the former question - what is the complexity of the algorithm. First, consider the initialization. The most time-consuming step is the step (init 2). One iteration of this step takes time of $O(|G|)$. We shall repeat it $O(|G|)$ times, so its complexity is equal to $O(|G|^2)$. Finding non-terminals with $Sht(A) = 1$ is as hard as for the zero value. Now, for the algorithm itself. In (1) we must compute $Sht'(A)$ for $O(|G|)$ non-terminals. Every computation takes $O(|G|)$. This step is repeated $O(|G|)$ times, so the whole complexity is $O(|G|^3)$.

Now we shall try to solve another problem: Compute, how long is the shortest word from some language L that contains some letter $a \in \Sigma$. Although this problem can look quite unimportant, we shall need this result further. So let $Shl(L, a)$ be the value corresponding to the language L and the letter a. We shall be often in case $L = L(A)$ so we shall use also $Shl(A, a)$.

Algorithm 2.1.2.

Input: *Context-free grammar $G = (N, T, P, S)$ in Chomsky normal form*

Output: *$Shl(L(A), a)$ for every non-terminal $A \in N$ and every $a \in \Sigma$*

Initialization: We calculate *Sht* value for every non-terminal of G. Similarly like in the former algorithm, we shall use the set H that will store all pairs (A, a) for which we already known the $Shl(A, a)$ value. This set will have the property similar to the former one:

$$(A, a) \in H \wedge (B, b) \notin H \Rightarrow Shl(A, a) \leq Shl(B, b)$$

After the initialization phase, H should contain all pairs (A, a) such that $Shl(A, a) = 1$. Proof of this is easy, we can immediately see that such (A, a) that $(A \to \ a) \in P$ has this property. Then also any pair (B, a) which satisfies $B \Rightarrow^* A$ will have this property. Therefore, set of all pairs which will have this property can be constructed by an iterative approach similar to the initialization phase when finding *Sht*. Let us look directly at the algorithm

(1) For all pairs $(A, a) \notin H$ calculate $Shl'(A, a)$ by the rule

$$Shl'(A, a) = \min\{l|\ (\exists B, C)(A \to BC) \in P \wedge \\ \wedge(Shl(B, a) + Sht(C) = l \vee Shl(C, a) + Sht(B) = l)\}$$

If the minimizing set is empty then $Shl(A, a) = \infty$. Let (A, a) be any pair not included in H with minimal *Shl* value. Then $Shl'(A, a) = Shl(A, a)$.

(2) Construct the new set H by rule $H := H \cup (A, a)$. If H contains all possible pairs terminate, else go to step (1)

Proof of the algorithm is similar to the proof of Algorithm 2.1.1, so we can omit it. □

Also, calculating the complexity of this algorithm is similar to Algorithm 2.1.1. One complication is in the fact, that instead of non-terminals, we are working with pairs (A, a) the number of which is $O(|G|^2)$. Therefore, the complexity of this algorithm is $O(|G|^6)$

2.2 Algorithms for projections

2.2.1 Basic edit-distance and context-free languages

Edit-distance and Δ similarity have nice properties such as symmetry and reflexivity. This encourages us to ask some more difficult questions about it. In geometry it is common to define distance of two objects as a distance of two nearest points A, B, belonging to the first and the second object, respectively. We shall take some inspiration from this and we shall define distance between a word and a language.

Definition 2.2.1. *Let $d : \Sigma^* \times \Sigma^* \to \mathbb{R}^+$ be a similarity or distance measure. Now we can extend the definition of d so that for any word u and any language L we define*

$$d(u, L) = \inf\{d(u, v) | v \in L\}$$

for distance functions and

$$d(u, L) = \sup\{d(u, v) | v \in L\}$$

for similarity functions.

In this definition we are using supremum and infimum, because minimal and maximal value of given sets do not have to exist, but in all cases we shall assume, they do, so we can use minimal and maximal values instead.
Now, we shall be interested in the question: How big is distance between a word and a language L, when considered distance is the basic edit-distance (only insertions and deletions with cost 1 are allowed) and for L we take a context-free language. First thing helping us is that the basic edit-distance d has nice properties, when considering the distance between a word and $L = L_1 \cdot L_2$. This is formulated in the following lemma.

Theorem 2.2.1. *Let $u \in \Sigma^*$ be a word, L_1, L_2 be languages and $L = L_1 \cdot L_2$. Then the following equation holds:*

$$d(u, L) = \min_{u=u_1u_2} \{d(u_1, L_1) + d(u_2, L_2)\}$$

Proof: First, we can use this lemma:

Lemma 2.2.1.

$$\begin{aligned} d(u, v) = \min\{&|u_1| + \cdots + |u_n| + |v_1| + \cdots + |v_n| | \\ &\exists v_1 \ldots v_n, u_1 \ldots u_n \in \Sigma^*, \exists a_1, \ldots, a_{n-1} \in \Sigma^* \\ &u = u_1a_1 \ldots u_{n-1}a_{n-1}u_n \ \wedge \ v = v_1a_1v_2a_2 \ldots v_{n-1}a_{n-1}v_n\} \end{aligned}$$

For purposes of this lemma, any decomposition of a pair (u, v) as was used above will be called a partitioning, the sum we are minimizing will be called a cost of partitioning and partitioning found by minimization will be called optimal.

This lemma is corrollary of 1.3.2 and properties of LCS. □

Let us continue the proof of Theorem 2.2.1. Let $v \in L$ be a word for which it holds: $d(u, L) = d(u, v)$. Clearly v is a concatenation of two words $v_1 \in L_1$ and $v_2 \in L_2$. Now, we consider the optimal partitioning of u and v (in the sense of the above lemma). This partitioning is inducing a decomposition of u to $u = u_1u_2$ and some partitionings on pairs (u_1, v_1) and (u_2, v_2). The sum of the costs of these two partitionings is as large as the cost of the optimal partitioning on $(u,\ v)$, so this proves that $d(u, v)$ is greater or equal than the right side of the equation.

Moreover, taking any decomposition $u = u_1u_2$, $v = v_1v_2$ and transforming u_1 to be equal to v_1 and u_2 to be equal to v_2 is some (not necessarily optimal) method of transforming u to be equal to v. Thus $d(u, v)$ is less than or equal to the right side of the equation. □

Now, we shall see, how this theorem helps us to find distance between a word and a context-free language.

Consider a context free grammar $G = (N, T, P, S)$ in Chomsky normal form. When computing $d(u, G_A)$ $(A \in N)$, we can use:

$$L_A = \bigcup_{A->BC}^{B,C \in N} L_B L_C \ \cup \bigcup_{A->a}^{a \in T} a$$

and for $d(u, BC)$ we can use the method that is implied by Theorem 2.2.1 - we try all partitionings $u = u_1u_2$, recursively compute the basic edit-distance u_1 from $L(G_B)$ and u_2 from $L(G_C)$ and we find the minimum. This method can be easily implemented, but it has one little problem. Suppose we are computing $d(u, BC)$. We know there exists $u = u_1u_2$ such that $d(u, BC) = d(u_1, B) + d(u_2, C)$. If $u_1 = \epsilon$, we must compute $d(u, C)$. But the problem will remain, computation of $d(u, C)$ will lead to a computation of $d(u, D)$ etc., so the computation will be infinite. We must find some way, how to eliminate these problems. To make the situation more clear, we can visualize the computation of $d(u, A)$ as follows:

$$d(u, A) = d(u_1, B) + d(u_2, C) = d(u_{11}, E) + d(u_{12}, F) + d(u_{21}, G) + d(u_{22}, H) = \dots$$

where some oracle advises us, how the words should be split and which rules for non-terminals should be used. Our problem with never-ending computation looks like:

$$d(u, A) = d(u, B) + d(\epsilon, C) = d(u, E) + \dots \dots$$

We shall call such part of a computation an ϵ part. Whether $d(\epsilon, C)$ is equal to zero or not, we can assume that the computation does not contain an ϵ part of length more than $|N|$. If so, some non-terminal must appear more than once in it, so the whole computation between these two occurrences can be omitted. This will make the ϵ part shorter (and if some $d(\epsilon, C)$ which arises somewhere between two occurrences of the same non-terminal is non-zero, then it is in direct contradiction with the minimal computation we are considering).

So, we shall be counting, how long is the actual ϵ-part of the computation. If this length is more than $|N|$, we know, that all other ϵ steps are in vain and we can force the computation to do some non-ϵ step. We write these thoughts formally in this algorithm:

Algorithm 2.2.1.

Input: *Context-free grammar $G = (N, T, P, S)$ in Chomsky normal form, word $u \in \Sigma^*$*

Output: *The basic edit-distance between word u and language $L(G)$*

For purposes of the computation of $d(u, L(G))$ where $G = (N, T, P, S)$ we define the function $D : \mathbb{N}_0^{|u|} \times \mathbb{N}_0^{|u|} \times N \times \mathbb{N} \to \mathbb{R}$ in such way, that

$$D(i, j, A, k) = d([u]_i^j, L(G_A))$$

where k is the length of the last ϵ part in the computation. Now for D it holds that:

(i) $D(i, i, A, m) = d(\epsilon, A) = Sht(A)$

(ii) $D(i, j, A, 0) = \min\{\alpha_1, \alpha_2\}$ where $i < j$, and:

$$\alpha_1 = \min_{\substack{w \in T \\ A \to w}} \{d([u]_i^j, w)\}$$

$$\alpha_2 = \min_{\substack{i<k<j \\ A \to BC}} \{D(i, k, B, |N|) + D(k, j, C, |N|)\}$$

(iii) $D(i, j, A, m) = \min\{\beta_1, \beta_2, \beta_3\}$ where $m > 0$, and:

$$\beta_1 = \min_{A \to BC} \{D(i, j, B, m-1) + Sht(C)\}$$

$$\beta_2 = \min_{A \to BC} \{D(i, j, C, m-1) + Sht(B)\}$$

$$\beta_3 = D(i, j, A, m-1)$$

Step (i) solves the trivial case, when looking for distance between ϵ and the language $L(G_A)$ which we can easily compute using Algorithm 2.1.1. Step (iii) allows the computation to create an ϵ part of length at most $|N|$, but it is not necessary to create it (β_1). Step (ii) is the one, where we force the computation to do a non-ϵ step (therefore $i < k < j$).

Now to the time-complexity of the algorithm. We must compute $O(|u|^3.|N|)$ values of the function D. If we compute $Sht(A)$ for all A and remember it, then the step (i) is fast. So is the step (iii). The step (ii) takes most of the time. By this step also most of the values of D are computed. One step (ii) takes $O(|G|.|u|)$, so for the total complexity we have $O(|u|^4.|G|^2)$.

2.2.2 Δ-similarity and context-free languages

In this subsection we shall compute the Δ similarity of a word and a context-free language. Similar to problems with basic edit-distance and context-free languages, also this problem was solved in [1], where author presented an algorithm for finding a lower bound for this value. In this subsection an algorithm for exact value will be presented. First, we shall see, that it is necessary to do more, than just modify the algorithm for the basic edit-distance. The following lemma shows where the problem is:

Lemma 2.2.2. *There exist languages L_1, L_2 and a word u such that:*

$$(\forall u_1, u_2 \in \Sigma^*)(\exists v_1 \in L_1, v_2 \in L_2)((u = u_1u_2 \ \wedge \ \Delta(u_1, L_1) = \Delta(u_1, v_1) \ \wedge$$
$$\Delta(u_2, L_2) = \Delta(u_2, v_2)) \Rightarrow \Delta(u, v_1v_2) \neq \Delta(u, L))$$

Proof: In other words, this lemma says, that such property like Theorem 2.2.1 does not hold. We illustrate this fact on the following example that is also used in [1]

Example 2.2.1. $L_1 = \{aa\}, L_2 = \{abccc, ac\}, u = aaab$. For concatenation of L_1 and L_2 we have $L_1L_2 = aaabccc, aaac$. Now, $\frac{8}{11} = \Delta(aaab, aaabccc) < \Delta(aaab, aaac) = \frac{3}{4}$, so $v = aaac$ and $\Delta(u, L) = \Delta(u, v)$. Now we can try all partitionings $u = u_1u_2$ to verify that there always exist v_1, v_2 such that $\Delta(u_1, L_1) = \Delta(u_1, v_1)$ and $\Delta(u_2, L_2) = \Delta(u_2, v_2)$ and $v \neq v_1.v_2$

(i) $u_1 = \epsilon \;\; u_2 = aaab$. Now $v_1 = aa$ and $v_2 = abccc$. $v \neq v_1v_2$

(ii) $u_1 = a \;\; u_2 = aab$. Now $v_1 = aa$ and $v_2 = abccc$. $v \neq v_1v_2$

(iii) $u_1 = aa \;\; u_2 = ab$. Now $v_1 = aa$ and $v_2 = abccc$. $v \neq v_1v_2$

(iv) $u_1 = aaa \;\; u_2 = b$. Now $v_1 = aa$ and $v_2 = abccc$. $v \neq v_1v_2$

(v) $u_1 = aaab \;\; u_2 = \epsilon$. Now $v_1 = aa$ and v_2 can be either $abccc$ or ac. The second possibility satisfies $v = v_1 v_2$, but the first does not.

This example proves our lemma. □

Lemma 2.2.2 shows, that we cannot just modify Algorithm 2.2.1 to get an algorithm for Δ-similarity and context free language. Although, we can modify Algorithm 2.2.1 to find some estimation for a value we are looking for. This approach was used in [1].

We started by Definition 1.4.6 that gives us an idea to compute LCS instead of Δ-similarity. We can naturally extend the definition of LCS in such a way that it will also define the LCS of word and language. Given a word u and a context-free language L we compute words $v_1, \ldots, v_n$, where $(n = |u|)$ such that v_i is the shortest word for which $|LCS(v_i, L)| = i$. Now we know that

$$(\exists j \in \mathbb{N})\ \Delta(u, L) = \Delta(u, v_i)$$

The only thing we have to do is to try all v_i's and find the maximum of $\frac{i}{|uv_i|}$. Now we show, how we are going to find the v_i's. We use a property of LCS, that Δ-similarity does not posess. This property is shown in the following theorem:

Theorem 2.2.2. *For LCS, word u and languages L_1, L_2 it holds that:*

$$|LCS(u, L_1L_2)| = \max_{u=u_1u_2} \{|LCS(u_1, L_1)| + |LCS(u_2, L_2)|\}$$

Furthermore it holds that:

$$|LCS(u, L_1 \cup L_2)| = \max \{|LCS(u, L_1)|, |LCS(u, L_2)|\}$$

Proof: The second property is a direct consequence of the extended definition of LCS to LCS between a word and a language. For the first property let v denote a word for which $LCS(u, L) = LCS(u, v)$. The word v is a concatenation of some $v_1 \in L_1$ and $v_2 \in L_2$. Now v_1 and v_2 gives some partitioning of u into u_1u_2. This partitioning is the one that 'wins' the maximization (if not, it is a contradiction with the maximal length of $LCS(u, L_1L_2)$) and it has exactly the same value as the left side of the proven equation. □

We are now ready to present our algorithm:

Algorithm 2.2.2.

Input: *Context-free grammar $G = (N, T, P, S)$ in Chomsky normal form, word $u \in \Sigma^*$*

Output: *Δ-similarity between word u and language $L(G)$*

We construct words $v_1, \dots, v_n$ ($n = |u|$). For the construction of v_p we do the following: We define a function $D_p : \mathbb{N}_0^{|u|} \times \mathbb{N}_0^{|u|} \times N \times \mathbb{N} \to \mathbb{R}$ such that

$$D_p(i,j,A,m) = \min\{|v| : v \in L(G_A) \ \wedge \ |LCS([u]_i^j, L(G_A))| = p\}$$

Now, we can construct the function D_k using the following rules:

(i) $D_0(i,j,A,m) = d(\epsilon, A) = Sht(A)$

(ii) $D_1(i,j,A,m) = \min\limits_{i \le k < j} (\{Shl(u_k, A)\} \cup \{\infty\})$

(iii) Computation of $D_p(i,j,A,m)$ for $p > 1$:

(a) $D_p(i,j,A,0) = \min\limits_{\substack{i<k<j-1 \\ A \to BC \\ p=l+r \\ r>0,\ l>0}} \{D_l(i,k,B,|N|) + D_r(k,j,C,|N|)\}$

(b) $D_p(i,j,A,m) = \min\{\beta_1, \beta_2, \beta_3\}$ where $m > 0$ and:

$$\beta_1 = \min_{A \to BC}\{D_p(i,j,B,m-1) + Sht(C)\}$$
$$\beta_2 = \min_{A \to BC}\{D_p(i,j,C,m-1) + Sht(B)\}$$
$$\beta_3 = D_p(i,j,A,m-1)$$

Having D computed, return

$$\max_i \left\{ \frac{2D_i(0,|u|,S,|N|)}{i + |u|} \right\}$$

To make the algorithm easier to understand, let us make some notes. First, the meaning of m, the last argument of the function D is the same as the meaning of the last argument of D in Algorithm 2.2.1. Our computation can do ϵ-steps (included in β_1, β_2). Using the same argumentation as in Algorithm 2.2.1 we find out, that it makes no sense to make more than $|N|$ ϵ-steps. Thus we can force the algorithm to do non-ϵ-step, this is done in (iii-a), where we insist on $i < k < j-1$ and also on $r > 0, l > 0$ (without these conditions, the algorithm could perform never-ending computations).

Let us consider the time complexity of the algorithm. We must compute $O(|u|^3.|N|)$ values of the function D. If we compute $Sht(A)$ and $Shl(a,A)$ for all A and all A, a and remember the results, then steps (i) and (ii) are fast. Analysing the step (iii): Most of the time is taken by the step (a), where we must try all rules for the non-terminal A ($|N|$), all positions of k and l (resp. r), so one step takes $O(|G|.|u|^2)$ values. The total complexity of the algorithm is thus $O(|u|^5.|G|^2)$.

Chapter 3

Conclusion

There are several questions, which have arisen from this work that were not well answered here.

First, it is the problem: At what circumstances can the generalized edit-distance be computed effectively? We have found some conditions that are rather restricting for the edit-distance, but this only resulted in NP-completeness of the problem. On the other hand, there exist some instances such as the basic edit-distance that are computable effectively. Question is whether there exist some other restrictions, under which the edit-distance becomes computable in polynomial time.

The second problem is related to similarities on random words. Δ-similarity of two long random words over binary alphabet is about 0.8 (80%). This means that Δ-similarity in some way does not fit our intuition, because two words that are not related with each other seem very alike according to their Δ-similarity.

The third question is connected with extending the distance function to languages. It would be interesting, if distance of two languages could be defined using only a distance function that defines the distance of words.

Bibliography

[1] Štangel, M.: Finite Approximations and Similarity of Languages. Master's Thesis, 2001

[2] Jánošík, J.: Konečné aproximácie jazykov. Master's Thesis, 1999

[3] A. Caprara. Sorting permutations by reversals and eulerian cycle decompositions. SIAM Journal on Discrete Mathematics, 12(1):91 110, 1999.

[4] E. S. Ristad, and P. N. Yianilos, , Learning String Edit Distance. Tech. Rep. CS-TR-532-96, Department of Computer Science, Princeton University, Princeton, N.J.October 1996. Revised November 1997.

[5] J.E. Hopcroft and J.D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979.

[6] V. Dančík, Expected Length of Longest Common Subsequences, PhD thesis, University of Warwick, 1994.

[7] V. Chvatal and D. Sankoff, Longest Common Subsequences of two Random Sequences, Journal of Applied Probability, 12 (1975), pp. 306-315.

Resumé

Táto diplomová práca je venovaná rôznym spôsobom určovania vzdialenosti konečných slov. V tejto oblasti sú už dosiahnuté mnohé výsledky. Hnacím motorom sú potreby praxe (najmä biológia). Je známych viacero viac či menej použiteľných funkcií na určovanie vzdialenosti. Ak definujeme nejakú vzdialenostnú funkciu, môžeme sa posunúť "o úroveň ďalej" a začať si klásť otázku, ako definovať vzdialenosť slova a jazyka. V druhej časti práce sa venujeme práve tomuto problému.

Zaujímavým problémom je tiež skúmať priemernú vzdialenosť dvoch slov ktoré boli vygenerované istým spôsobom. Tejto problematike sa však práca venuje iba okrajovo.

Hlavnými výsledkami tejto práce sú:

- Abstraktná definícia edit-distance, predstavenie niekoľkých jej inštancií.
- Niekoľko tvrdení, ktoré sú výsledkom snahy zistiť, kedy sa konkrétna inštancia edit-distance dá rátať v polynomiálnom čase.
- Definícia fragment-distance, návod na jej rátanie a niekoľlko vlastností.
- Zjednodušená definícia Δ-podobnosti, definovanej v [1] a niekoľko dôkazov, ktoré sa vďaka tejto definícii zjednodušili.
- Algoritmus na rátanie Δ-podobnosti slova od bezkontextového jazyka.