Text analysis with sequence matching

Marko Ferme, Milan Ojsteršek

***Abstract*—** This article describes some common problems faced in natural language processing. The main problem consist of a user given sentence, which has to be matched against an existing knowledge base, consisting of semantically described words or phrases. Some main problems in this process are outlined and the most common solutions used in natural language processing are overviewed. A sequence matching algorithm is introduced as an alternative solution and its advantages over the existing approaches are explained. The algorithm is explained in detail where the longest subsequences discovery algorithm is explained first. Then the major components of the similarity measure are defined and the computation of concurrence and dispersion measure is presented. Results of the algorithms performance on a test set are then shown and different implementations of algorithm usage are discussed. The work is concluded with some ideas for the future and some examples where our approach can be practically used.

***Keywords*—**Sequence matching, subsequence analysis, similarity measure, fuzzy string search, phrase detection

I. INTRODUCTION

IN natural language processing, user input is usually being matched against a knowledge base, which consist of a finite collection of semantically described words or phrases. While trying to classify parts of the user input, a comparison between those two must eventually be made. In this article we will be dealing with the user input in the form of text, which is given in a form one or more sentences.

Because of the structure of the knowledge base, which consists of words and phrases, the first task for a successful comparison is to break the sentences into these smaller parts. The most common approach is to break the sentence into words using word breakers. Word breakers are characters such as space, coma, or period, which delimit words in text. Determining which words form a phrase is a much more difficult task, which is usually solved after the comparison with the knowledge base has been made. For that to be possible, the knowledge base must consist only out of single words that can then later be linked together to a meaning of phrase. Detection is then possible in a later stage of text processing.

Because of user input being an infinite set, either a reduction of input words or expansion of the knowledge base must be made. Popular approaches that solve this problem include stemming, lemmatization and various distance functions. Stemming is based upon a set of rules, which determine word morphing, and is therefore limited to weakly inflected languages, where such rule collections exist. Lemmatization is used in conjunction with large language specific dictionaries, which are used to expand the knowledge base dictionary. This information is then used to derive morphed words into their lemma. Both approaches are intolerable to user input errors and have a finite set of either rules or words. Distance functions such as Levenshtein distance [1], which are based on the number of changes required to transform one sequence into another, are used to address this issue. While they do offer some level of "fuzzy" sequence matching they lack the information and depth of analysis to determine sufficient sequence similarity. As already mentioned, all of the above methods require text segmentation to a smallest free form of a language (word), which implies the usage of advanced matching algorithms when dealing with phrases or multiple word entities.

In our work we were building a question answering system[2] in which we were focusing on matching words or phrases from our knowledge base against an user given sentence, which was a perquisite for future semantic processing. Therefore we present an alternative approach for determining sequence similarity based on subsequence analysis which is language invariant, error tolerable, does not require additional rules or language dictionaries and can be used with same efficiency on single words, phrases or even larger texts. Moreover it can also be used for phrase extraction or detection in sentences or larger quantities of text. In chapter II we describe the problems we have encountered while trying to deploy the more commonly used solutions for text processing described above. We also describe the proposed solution to this problems and define the necessary attributes for determining similarity between two sequences and a similarity metric. In chapter III an algorithm for determining the longest subsequences from a sequence pair is described. In chapter IV we describe the computation of similarity. We provide the measured similarity results on a test set in chapter V and describe some ideas on how to improve the algorithm accuracy when choosing the most similar sequence. In chapter VI, we describe the algorithm application in order to detect phrases and provide some ideas on how to solve phrase or word overlapping. We conclude our paper with some future work ideas in chapter VII.

II. USER INPUT MATCHING

The problem of user input manipulation, to better match the existing knowledge, is one being solved in different manners, mostly depending on the language in which the knowledge is contained and the user input is submitted. As mentioned in the introduction the first step in text processing is text

segmentation. The most simple way to split text into words is to use word breaking lists. These contain a list of literals, which delimit words. Such list can be found in various tools and offers a fast and reliable way of simple text segmentation. The problem of such text segmentation is, that it cannot determine phrases, should they occur in text and requires our knowledge base to consist of only single words. We must also prepare special structures in which we store information about word order, should we later want to perform phrase detection. After we split the text into words, we can process them with one of the common natural language processing methods:

A. Stemming

One of the most common approaches in natural language processing is stemming. It is very effective in languages that are weakly inflected like modern English, Swedish, Norwegian and Danish and struggles with languages that are moderately inflected such as Spanish, Italian, French, Portuguese and Romanian. Languages being highly inflected are considered inappropriate for efficient stemming [3]. Such languages include all the Slavic languages. Stemmers are also intolerable to user input errors (misspelling), which are bound to occur during natural text processing.

B. Lemmatization

In heavily inflected languages the use of lemmatization is preferred. It offers a fast and accurate way of matching user input to morphed instances of a headword but requires exact dictionaries, which have to be build by language experts. A major problem in the process of lemmatization is disambiguation, which occurs when a word or phrase can be transformed into two or headwords. It is most widely being solved with the usage of tree taggers which require large training corpuses and use probability to determine the most suitable headword, which we call a lemma. Building such large collections is very time consuming and requires the aid of language experts. Such corpuses can also suffer from domain specific language usage and can cripple the tree tagger results on a different domain. A very large portion of misses in lemmatization, when being used on heavily inflected languages, is produced from unknown words [4], such as names, surnames and geographical locations. Those are mostly excluded from dictionaries and tagged corpuses, which makes them nigh on impossible to convert to a lemma. Input error (misspelling) intolerance during lemmatization or tree tagging is in most cases also unaddressed.

C. Edit distance functions

Edit distance function where designed to counter the occurrence of misspelling in sequence matching, which both of the above described approaches cannot resolve. The basic idea is to determine the number of transforms needed to convert one sequence to another. Different variations of edit distance are known and used, one of the most popular being the Levenshtein distance [1]. It is defined as the number of insertions, deletions and substitutions needed to align two sequences, where weights for each transformation can be defined. Its most commonly used in spellchecking and is implemented in various software that contains text processing.

D. Sequence matching

Algorithms for sequence matching exist but are more popular in DNA sequence matching then in string matching. They try to find global [5] or local [6] subsequence alignments and mostly use expert defined tables which determine the distance between single sequence elements. Because our goal was to find a measure of similarity expressed in a way, that would allow effective comparison between sequences and allowed us to find the most similar one, the above described sequence matching algorithms were not sufficient.

After we successfully determined which words are contained in the user given text, the next step would be linking the words in possible phrases. For this step sufficient information must exists in our knowledge base otherwise the phrase detection is impossible.

We propose a new algorithm that discovers the longest subsequences contained in a pair of sequences and then evaluates them. By doing so we measure similarity instead of difference. We define three main decision factors, that will help us calculate a similarity measure:

- total length of common subsequences
- dispersion of common subsequences
- order of common subsequences

Based on this decision factors, similarity between two sequences is calculated and present as a value between zero, meaning that the sequences are disjoint or completely scrambled, and one, meaning that the sequences are identical.

III. SUBSEQUENCE DISCOVERY

In order to find similarity between two sequences we pursued an idea of finding the longest common subsequences which do not overlap. Our first step was to find all the subsequences contained in a sequence pair. We derived an algorithm from the solution to the longest common substring problem. In Fig 1. we can see how algorithm transverses through the matrix, building a list of subsequences candidates.

			d	e	l	i	v	e	r	y		d	a	y
		i	0	1	2	3	4	5	6	7	8	9	10	11
	j	0	0	0	0	0	0	0	0	0	0	0	0	0
d	0	0	1	0	0	0	0	0	0	0	0	1	0	0
a	1	0	0	0	0	0	0	0	0	0	0	0	2	0
t	2	0	0	0	0	0	0	0	0	0	0	0	0	0
e	3	0	0	1	0	0	0	1	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0	1	0	0	0
o	5	0	0	0	0	0	0	0	0	0	0	0	0	0
f	6	0	0	0	0	0	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	0	0	0	1	0	0	0
d	8	0	1	0	0	0	0	0	0	0	0	2	0	0
e	9	0	0	2	0	0	0	1	0	0	0	0	0	0
l	10	0	0	0	3	0	0	0	0	0	0	0	0	0
i	11	0	0	0	0	4	0	0	0	0	0	0	0	0
v	12	0	0	0	0	0	5	0	0	0	0	0	0	0
e	13	0	0	1	0	0	0	6	0	0	0	0	0	0
r	14	0	0	0	0	0	0	0	7	0	0	0	0	0
y	15	0	0	0	0	0	0	0	0	8	0	0	0	1

Fig. 1 Subsequences discovery

We start the procedure by ordering the subsequences candidates list according to their length. The longest subsequence is added to the list of final longest subsequences and removed from the list of subsequence candidates. The rest of the subsequence candidates is then filtered for overlapping. Based on the longest subsequence list start and end coordinates, three zones are constructed. If we name the longest subsequence l we can describe its start (1) and end (2) coordinates as a coordinate pair (i, j), where the meaning of i and j is shown in the matrix in Fig 1.

$$(si_l, sj_l) \quad (1)$$

$$(ei_l, ej_l) = (si_l + len(l) - 1, sj_l + len(l) - 1) \quad (2)$$

The longest subsequence l then splits the entire matrix into three zones (3), (4), and (5) as shown by example in Fig 2.

$$Z_I(i,j) = \begin{Bmatrix} 1, if\ si_l \le i \le ei_l\ AND\ sj_l \le j \le ej_l \\ 0, else \end{Bmatrix} \quad (3)$$

$$Z_N(i,j) = \begin{Bmatrix} 1, if\ i < si_l\ AND\ j < sj_l \\ 1, if\ i < si_l\ AND\ j > ej_l \\ 1, if\ i > ei_l\ AND\ j < sj_l \\ 1, if\ i > ei_l\ AND\ j > ej_l \\ 0, else \end{Bmatrix} \quad (4)$$

$$Z_{NI}(i,j) = \begin{Bmatrix} 1, if\ si_l \le i \le ei_l\ AND\ j < sj_l \\ 1, if\ si_l \le i \le ei_l\ AND\ j > ej_l \\ 1, if\ i < si_l\ AND\ sj_l \le j \le ej_l \\ 1, if\ i > ei_l\ AND\ sj_l \le j \le ej_l \\ 0, else \end{Bmatrix} \quad (5)$$

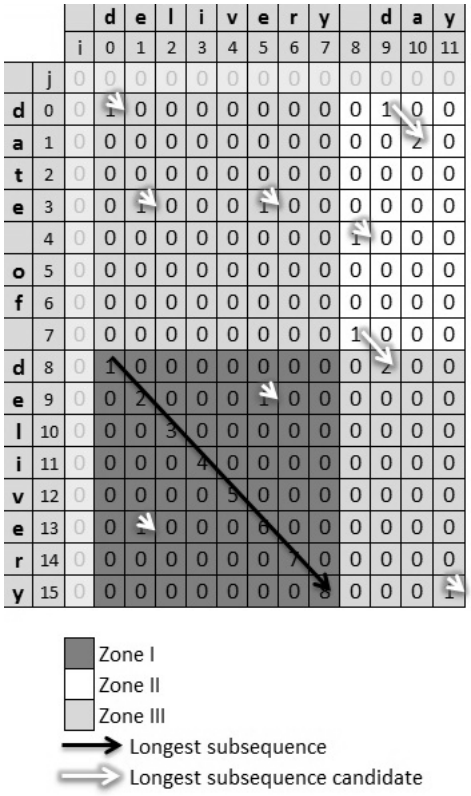

Fig. 2 Matrix division into zones for determining the longest subsequences

Every remaining subsequence is then processed depending on the zone it is residing in. For each subsequence the start and end coordinates are compared with the conditions shown in (3), (4) and (5) and a fitting zone is assigned. Three outcomes are possible:

1. If the subsequence candidate start and end coordinates are both located in zone I or in zone III, we remove that subsequence from the list of subsequences candidates, because it is already contained in the subsequence just added to the final list.
2. If the subsequence candidate start and end coordinates are located in zone II, we leave the subsequence unaltered, because it does not overlap with the subsequence just added to the final list and is still a potential candidate for the final subsequence list.
3. If the subsequence candidate start or end coordinates are located in zone III, subsequence overlapping must be solved. The subsequence cannot start and end in zone III, since that would mean that it is longer than the subsequence we just added to the final list, which cannot be true, since we ordered the list descending by sequence length. We then must either remove the leading or the trailing members of the subsequence depending on where the subsequence starts. If it starts inside zone I, we must remove the leading elements until we move its starting coordinates into zone II. If it starts inside zone III we must remove its trailing

elements until we move its end coordinates into zone II. That way we removed the possible overlapping by favoring the longest sequence to retain all its member elements and removing the overlapped elements from the shorter subsequence candidates.

The candidate list is again ordered according to the subsequences length, the longest subsequence added to the final subsequences list and the candidates list is checked for overlapping again. This process is repeated until the candidates list is empty and we are left with the final list of longest subsequences, which do not overlap.

IV. MEASURING SEQUENCE SIMILARITY

The similarity measure between two sequences is composed out of the following three parts:

- concurrence measure,
- dispersion measure and
- ordering measure.

The first part measures the subsequence concurrence in both sequences being matched and is calculated as shown in (6).

$$f = \frac{\sum_{i=0}^{n} \overline{p_i}}{\max(\bar{a},\bar{b})} \quad (6)$$

$\bar{a}$ - length of sequence a

$\bar{b}$ - length of sequence b

p - longest subsequences list

$\overline{p_i}$ - length of the i-th subsequence in the longest subsequences list

This part of comparison basically represents the share of elements contained in both sequences. If the sequences are identical the result is one. If the sets are disjoint the result is zero.

The second part measures the subsequences dispersion and is calculated as shown in (7).

$$g = 1 - \left(\frac{N_p - 1}{\min(\bar{a},\bar{b}) - 1}\right)^{\bar{p}_{max}/4} \quad (7)$$

N_p-number of longest subsequences found

$\bar{a}$ - length of sequence a

$\bar{b}$ - length of sequence b

$\bar{p}_{max}$ - length of the longest subsequence

Dispersion measure is designed around the fact that good similarity can be achieved only if there is a small number of found subsequences. Two identical sequences should only have one found subsequence, where the dispersion measure would be one. On the other hand, two sequences that might have the same elements, but their order is completely scrambled, would have n subsequences, where n is the length of the shorter sequence. In that case the dispersion similarity would be zero. The equation constructs a curve between these two values where its shape is affected by the length of the longest subsequence found. The division of the length of the longest subsequence by 4 makes this function either a convex, concave or a linear function. The higher the value the slower the curve will decline from the value 1, which means that it will be more tolerable to the higher number of subsequences, and the other way around. The value 4 which makes the curve linear was chosen for our test sets under the assumption that when comparing sequence and finding the longest subsequence with its length less than 4 should decline more rapidly. For other test or working sets a different value can be chosen as the convex/concave limit. Fig 3. shows the dispersion measure behavior for different function parameters based on the comparison of two sequences, which contained 15 elements. Axis Z represents the calculated dispersion measure (g), axis X represents the length of the longest subsequence found ($\bar{p}_{max}$)and axis Y represents the number of subsequences found (N_p). We can see that in cases of longer sequences the dispersion measure is very high when the number of found subsequences (Y) is low. The value declines more rapidly if the maximum subsequence length (X) is low.

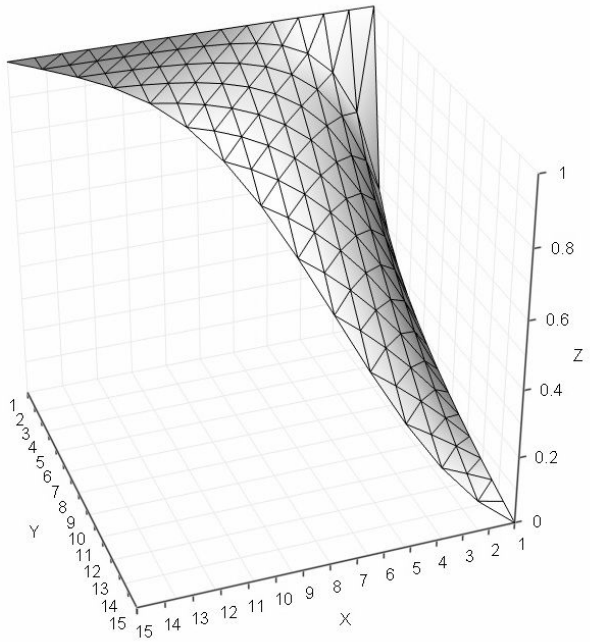

Fig. 3 Dispersion function graph for $\min(\bar{a},\bar{b})$=15

The measure of ordering is the last measure included in the measure of similarity. We do recognize its importance in sequence analysis but since our original problem was a sequence set, where ordering was not so important for discovering similarity (our sequence domain was a flexible word order language), we did not developed it further. So in our current solution we used the maximum value of order measure for each case of comparison, which is one. We do plan to address and solve this problem in our future work.

Since all the single measures are normalized the final measure of similarity S (8) is computed as a product of sub measures.

$$S = fgh \quad (8)$$

f - concurrence measure
g - dispersion measure
h - ordering measure (has value of 1)

The result of such a product is also normalized and can be used for simple comparison to find the most similar sequence.

V. RESULTS

For measuring the results we had to create a test set, which would allow us to compare sequences against each other and would at the same time contain the information about the closest match. Therefore we chose to use a Slovene language dictionary, which contains a large number of words in all morphed forms and their relation to the lemma as our test set. We suspected that our algorithm could have troubles on such a set, since it contains a lot of sequences that are very similar, where their distinction could prove to be difficult. The dictionary contains over 3 million distinct words which are linked to over 250 thousand lemmas. We then randomly chose one thousand words out of the whole dictionary creating a test set with an average sequence length of 10. We used the algorithm to compare each of the chosen sequences against the entire lemma collection in order to find the most similar match. That way we calculated all possible similarity measures and also got a lot of additional information, which can help us make the right decision. The most obvious choice now would be to choose the lemma with the highest similarity measure, which is what we did in our test. We also noted the similarity measure between the random word and its true lemma. If the lemma with the highest similarity measure is the same as the sequence true lemma, our algorithm performed well.

Fig 4. shows the measured result of our comparison. We can see, that the highest similarity between a sequence and its lemma was achieved 80,5 % of the time. In that cases the average similarity was 86,4%. In cases where the algorithm missed the lemma we can see, that the average similarity was 70,5%. From the collection of the missed cases we see that there were in majority considerably different as their lemma. Based on our result we can conclude that this algorithm is successful and suits the needs of natural language processing. Its success rate is slightly worse than those of stemmers and tree taggers when compared on a set that is strictly build out of the dictionary.

Similarity results	Total	Percentage	Sequence matching average
Correct prime forms	805	80,5%	86,4%
False prime forms	195	19,5%	70,5%
All	1000	100,0%	83,3%

Fig. 4 Algorithm results for detecting prime forms(lemmas)

In an absence of a test set, against which the most similar word is compared, the decision if a word is similar enough must also be made. It can easily happen, that the word we are comparing against a knowledge base is not contained in it. In this case we must use additional approaches to avoid suggesting a not so similar word, just because it is the most similar of all the words contained in the knowledge base. Here a simple threshold can be used, where only the words with similarity above it are considered. From our test we can see, that the average similarity measure was 86.5% when finding the correct form and it was 70,5% when finding a false one. So if we would approximate a threshold of a minimum 75% similarity measure to mark a word as the most similar one, we would greatly improved our false positives results. By doing so our success rate dropped to 72,7% but instead of giving 19,5% of false answers, we only selected the wrong word 7,2% of the time. Using such a threshold when applying the algorithm in a real application would greatly improve its precision by not giving the wrong answers and just marking the words below the threshold as unknown. In some systems selecting a wrong word may cause more problems and provide worse results as not recognizing a word at all. We are also aware that the measurements taken in our test set do not represent a real environment and that further research is needed to obtain better information on determining the right threshold values for specific languages or specific domains.

Another problem that occurs with our algorithm is when high similarities measures are computed for more sequences contained in the knowledge base. If the sequences are very similar to each other, it can easily happen that we will get a lot of sequences with high similarity measures with very similar values. The decision to pick the word with the highest similarity measure is highly doubtful in that case, since the difference between the first and the second candidate is very small. We developed an approach based on the relative distance between the calculated similarity measures, which helps us determine when such a case has occurred. In our test case the average difference between the word with the highest similarity measure and the similarity measure of the words lemma, when we detected a false lemma was 8,9%. That tells us, that in case our algorithm missed, the word's lemma and some other lemma were quite similar. This was an expected result, since we took a full dictionary as our test set, but it also showed us the potential problems that could occur when using this algorithm. When other means of solving disambiguation are applied, selecting more words as potential candidates is not a problem, but when relaying solely on this algorithm, selecting a wrong word should be avoided if possible. Applying a relative distance threshold can greatly help to reduce suggesting wrong lemmas when faced with the situation described above. When other words have the similarity measure inside the relative threshold of the highest rated word, the algorithm should mark the word as unknown, since it cannot decide on the correct one with high enough probability. Determining the value of the relative threshold can be a difficult task, dependent on the language or the domain the algorithm is being used in. Selecting a too high value can result in low precision.

The relative similarity distance can also be used as a test measure between the elements in the knowledge base or the collection against which the comparison will be made. The

first step would be creating a test sample out of the full set, and comparing it against the rest of the set, similar as we did in our test. By determining the average relative distances the suitability of this algorithm can be determined. If the relative similarity distance between the words is not to low, our algorithm will perform well. Otherwise the use of a different approach is advised.

Spelling mistakes or flexible sequence order were not taken into account in our test, since they are harder to produce and measure. Unfortunately our test set also did not include phrases where our algorithm would excel. As described in our plans for future work, we attempt to compare our algorithm against other natural language processing techniques on a test set that would include both regular and irregular spelled words and would also include phrases. We would also like to test different approaches for determining the algorithm threshold and maximum variance. Based on our test result, we conclude that the use of our algorithm is best suited in a environment where words or sequences are in general not to similar to each other. When dealing with such cases similarity threshold and relative similarity distance can be used to improve the algorithm accuracy.

VI. Phrase Detection

As we explained in our introduction, the first step in natural language processing, when dealing with natural text, is text segmentation. We explained how such segmentation could generate words from user given sentences and how extended knowledge bases could be used to detect phrases. When developing our algorithm we realized that with slight modification our algorithm could be successfully used to detect phrases in sentences given a knowledge base that would contain such phrases.

The modification is done in the first part of the similarity measure, where we need to change the way that the concurrence measure is calculated. We derive two concurrence measures from the basic formula (6), which only differ in the denominator. Instead of using a maximum length of the both sequences, we calculate the concurrence measure against each length as shown in (9) and (10).

$$f_a = \frac{\sum_{i=0}^{n} \overline{p_i}}{\bar{a}} \tag{9}$$

$$f_b = \frac{\sum_{i=0}^{n} \overline{p_i}}{\bar{b}} \tag{10}$$

We use these two concurrence measures to calculate two similarity measures S_a and S_b (11), where the first one measures how much sequence B resembles sequence A, and the second one measure how much the sequence A resembles sequence B. The dispersion measure is calculated only once in the same manner than in the original algorithm and its used for both similarity measure calculations. The same goes for the ordering measure.

$$S_a = f_a g h$$
$$S_b = f_b g h \tag{11}$$

f_a, f_b - concurrence measures
g - dispersion measure
h - ordering measure (has value of 1)

That way the similarity measures actually measures how much a sequence is contained in another sequence. If the similarity measure reaches its maximum value of one, then the sequence is a subsequence. We make use of this algorithm property for phrase detection. It allows us to compare whole phrases against whole sentences without the need to segment the sentence into words. The way the algorithm works, it is also very tolerable against slight changes in phrases caused by inflection and can, like in our case, to some degree ignore word order. The time complexity of using this approach is the same as using the original algorithm, since all the needed data is computed before calculating the concurrency measure. The only addition is one operation of division.

In our application of the algorithm in a question answering system[9], we had to detect phrases from a sentence. We had a knowledge base that contained single words as well as phrases and had to compare them against text in form of sentences given by a user. We compared each of the sequences from the knowledge base as sequence A against the entire sentence as sequence B and tried to detect which of them are contained in the sentence. As a result we got a table, an example is shown in Fig 5., where each of the entries in the knowledge base had both similarity measures calculated and also had the remainder of the comparison, which, as mentioned above, is a side product of the longest subsequences discovery algorithm. That way we had enough information to decide which of the words or phrases are actually contained in the sentence.

Sequence B: Kam javim zaporo ceste na javni cesti			
Phrase (Sequence A)	Sa	Sb	Remainder
ceste	100,00%	13,51%	Kam javim zaporo na javni cesti
na	100,00%	5,12%	Kam javim zaporo ceste javni cesti
Kam	100,00%	8,11%	javim zaporo ceste na javni cesti
zapora ceste na javni cesti	92,46%	67,47%	Kam javim o
Nina	88,89%	9,61%	Kam javim zaporo ceste jav cesti
zapora javne ceste	87,66%	42,64%	Kam javim o ni cesti
zapora ceste	82,64%	26,80%	Kam javim o na javni cesti
zapora ceste na javni poti	80,25%	56,39%	Kam javim o ces

Fig. 5 Phrase detection example

The easiest way to determine the phrases contained in the sentence is to order them first according to the similarity measure S_a. High S_a values presented the words or phrases, which are most likely to be contained in the user given sentence.

Two major problems arise from such solution. First we must determine how many of the found words or phrases should we consider as the ones contained in the sentence. A very simple solution, which we used in our application at first,

is to use a threshold method suggested in chapter V. All the phrases above the threshold are considered to be similar enough and are marked as words or phrases contained in the sentence. In our example we chose a threshold of 80%, but as explained in chapter V. the threshold must be chosen very thoughtfully. This approach works very well if the application the algorithm is being used in has other means to solve disambiguation. Setting the threshold to low can result in large amounts of words or phrases being marked as contained in the user sentence, which can create problems for the disambiguation process. The main reason for a large number of found word or phrases lies in overlapping, which we can also see in our example. If the knowledge base contains complex phrases the overlapping between them and simple words can occur easily. The same happens if the base contains some very short or simple words like in, or, at, etc. In our example we have a complex phrase "zapora ceste na javni cesti" which contains a simple word "cesta" and our base has a very short word "na" which can easily be contained in other words or phrases. In order to reduce this number an approach with the remainder can be used. After applying the threshold method, the table is sorted according to the value of S_b. The first word is automatically chosen as the correct one. The next word, which is treated as sequence A, is then compared using our algorithm against the remainder of the first word, which is treated as sequence B. If the newly calculated value S_a is lower as the one previously calculated, overlapping must have occurred, and the word is removed from the table. If the value S_a remains the same, the newly acquired remainder becomes sequence B and the next sequence in the order table becomes sequence A. The process is repeated for all the words in the table as shown on our previous example in Fig 6.

Sequence B: Kam javim zaporo ceste na javni cesti			
Phrase (Sequence A)	Old Sa	Sb	Remainder(Sequence B)
zapora ceste na javni cesti	92,46%	67,47%	Kam javim o
zapora ceste na javni poti	80,25%	56,39%	Kam javim o
zapora javne ceste	87,66%	42,64%	Kam javim o
zapora ceste	82,64%	26,80%	Kam javim o
ceste	100,00%	13,51%	Kam javim o
Nina	88,89%	9,61%	Kam javim o
Kam	100,00%	8,11%	javim o
na	100,00%	5,12%	javim o

Deleted sequence
Accepted sequence, new remainder

Fig. 6 Detected words or phrases reduction using the remainder

Using this method we can be more exact and can avoid running into problems when using additional disambiguation methods. As shown, the algorithm can perform phrase detection inside sentences and can also solve disambiguation to some degree.

VII. Future Work

The major unaddressed issue in our work is time complexity, which we feel can be improved. Because we developed our algorithm based on a premise of comparing two sequences, we ended up with time complexity $O(n^2)$. When comparing a sequence to a collection of other sequences, trying to find the most similar one, this time complexity can become an issue. In our future work, we propose building a special case of suffix trees best suited for subsequence discovery. Such trees would reduce the time complexity of single sequence comparison against a sequence collection and would allow the development of special algorithm designed to find the most similar match. In course of this work we also plan to evaluate most commonly found subsequences and equip them with statistics and semantics. That way we could further improve our similarity matching algorithm to either exclude certain subsequences or increase their importance in a sequence according to statistics or semantic annotation. We also plan to test our algorithm against other approaches used in natural language processing. We plan to construct a more suitable test set, which would mimic user input as close as possible. Two major cases that we would like to include in our test set are phrases and spelling errors. We plan to extract phrases out of openly available thesauruses such as EuroVoc[7] and add them to our test set. We also plan to insert custom spelling errors and change the word order in phrases to make the test set more representative. With the help of such a test set we plan to improve our algorithms effectiveness by trying out different algorithm parameter values. We also want to test different ways of determining the best sequence match. In our work we simply chose the sequence with the highest similarity measure, but as explained we think that other factors can have a large effect on the most relevant sequence found. We plan to run additional test and measure the effects of using the approaches of relative distance and thresholds into account. We also plan to perform further testing on other corpuses to determine ideal threshold and relative distance values. With such measurements we hope to be able to predict good values for this parameters in a real environment. We also plan to apply external knowledge, which holds information about the sequence space we are in, to the similarity calculation. With such information we hope to lower the impact of some subsequences that hold no information or to raise the requirements on the similarity of specific words, such as names, surnames or places.

VIII. Conclusion

In our work, we presented an alternative approach to string matching. We based our work on sequence analysis and developed measures to evaluate similarity between two sequences. We preformed a test in which we tried to determine the accuracy of our algorithm and also provided some ideas on how to improve the best match selection. We presented the test results which helped us determine the values of some parameters in our algorithm for future implementation. We also presented a way to detect phrases from sentences with a slight algorithm modification. We shown the approach we used in our natural language processing application. We also pointed out some problems and gave some ideas for our future work. Our current work can be used in natural language processing either as a standalone metric to find the most appropriate matches in corpuses or as an addition to existing comparison techniques in a similar way that spellcheckers are

used. It can also be used to perform fuzzy string matching, either in full text searches across databases or as a text search in larger documents. It can also be used to extract phrases or even to compare documents. All of the above can be done with small modifications to the original algorithm and with smart parameters choice. We successfully deployed our algorithm into the advanced search in DKUM [8] making it a fuzzy full text search. We also used the algorithm in the upgraded version of the question answering system described in [10] as a process for detecting entities for solving disambiguation. We plan to use our algorithm in our next applicative projects and with its help plan to increase our matching precision and with that improve our users experience.

REFERENCES

[1] V. Levenshtein, Binary codes capable of correcting spurious insertions and deletions of ones, Probl. Inf. Transmission 1, 8–17, 1965.
[2] I. Čeh, M. Ojsteršek, *Slovene Language Question Answering System*, Proceedings of the 13th WSEAS International Conference on COMPUTERS.
[3] M. Popovic, P. Willett, "The effectiveness of stemming for natural language access to Slovene textual data", Journal of the American Society for Information Science, 43(5), 384–390, 1992.
[4] T. Erjavec, S. Džeroski, "Machine learning of morphosyntactic structure: Lemmatizing unknown Slovene words", Applied Artificial Intelligence: An International Journal, 18(1), 17-41, 2004
[5] S. B. Needleman, C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins", Journal of Molecular Biology 48 (3): 443–53, 1970.
[6] T. F. Smith, M. S. Waterman, "Identification of Common Molecular Subsequences", Journal of Molecular Biology, 1981.
[7] "Eurovoc thesaurus", http://europa.eu/eurovoc/, visited on September 2010
[8] J. Brezovnik, M. Ojsteršek, "Digital library of University of Maribor (more than just a bunch of documents)", Proceedings of the International Conference on Applied Computer Science, 2010.
[9] I. Čeh, M. Ojsteršek, "Developing a Question Answering System for the Slovene Language", WSEAS Transaction on Information science and applications, Issue 9, Vol. 6, 2009.
[10] B. Gorenjak, M. Ferme, M. Ojsteršek, *Ontology-Driven Question Answering System with Semantic Web Services Support*, Proceedings of the European Conference on Advances in Communications, Computers, Systems, Circuits and Devices (ECCS'10)

M. Ferme is a teaching assistant at University of Maribor, Faculty of Electrical Engineering and Computer Science. He graduated in 2008 at Faculty of Electrical Engineering and Computer Science at University of Maribor. His research interests are natural language processing, Question-answering systems, ontologies and semantic web. He has been involved in several research and commercial projects on question-answering systems.

M. Ojsteršek M. Ojsteršek is an associate professor at University of Maribor, Faculty of Electrical Engineering and Computer Science. His research is focused on heterogeneous computing systems, semantic web, service-oriented architecture, natural language processing and dialog systems. He has been involved in several research and commercial projects on question-answering systems