Parallel Application of Levenshtein Distance to Establish Similarity Between Strings

Paweł Kobzdej[a], Dariusz Waligóra[a], Kinga Wielebińska[a], Marcin Paprzycki[ab*]

[a]*Department of Mathematics and Computer Science, Adam Mickiewicz University, ul. Umultowska, 61-614 Poznań, Poland, {d115627, d115713, d118985}@atos.wmid.amu.edu.pl*
[b]*Computer Science Department, Oklahoma State Uniwersity, Tulsa, OK 74106, USA, marcin@cs.okstate.edu*

Abstract
This paper presents results of parallelizing an algorithm to calculate the modified Levenshtein measure (edit distance) as a method of establishing similarity between two character strings. The proposed approach was implemented using the MPI communication library. Results illustrating efficiency of parallelization are presented and discussed.

1. Introduction

Determining the similarity of strings has many applications that include spell-checking [1], examining correctness of pronunciation and affinities between dialects [2], analyzing the DNA structure [3] or Web mining [4], to mention just a few. There exist many methods that allow establishing presence of a string inside of another string e.g. Knuth-Morris-Pratt [5] or Boyer-Moore algorithms [6]. However, what is often needed is not establishing a fact that one string is a substring of another, but rather a measure of similarity between them. A standard example of such a situation is the case of plagiarism detection, where the plagiarized text is likely to have some words changed, some deleted, some exchanged etc. In such a case algorithms like Rabin-Karp [7] will not produce needed results. Therefore, a different approach is required and some form of approximate string matching has to be applied. It is worth mentioning that there exist three related problems: (a) string matching with "don't care" symbols, (b) string matching with k mismatches and (c) string matching with k differences. The first involves a meta-character matching "anything," the second allows matches where up to k characters are different, the third requires that the pattern has an edit distance from the text less then or equal to k [8, 9, 10].

In our work we have decided, however, to experiment with the general problem of establishing a "similarity" between two strings (finding that in certain areas they closely resemble each other). For this purpose we have selected a modified Levenshtein measure, which allows to assess similarity between strings by considering how many changes must be introduced to one string in order to transform it into another. More precisely, we have used the *edit distance* which is an extension of the original string similarity measure proposed by Levenshtein [11]. He considered three basic text editing operations: *insertion of a letter*, *deletion of a letter* and *change of a letter*. Edit distance as described in [12], adds one more edit operation to the above mentioned set: *transposition of two adjacent letters*. Therefore, if we assume that s and w are words of the length i and j respectively and w^j is a substring of word w that ends at position j, and w_j – is the letter at position j, then the edit distance between s and w is described by the formula:

[*]Corresponding author. Work at Adam Mickiewicz University was sponsored by a scholarship from the Fulbright Commission. Computer time grant from the Poznań Supercomputing and Networking Center is kindly acknowledged.

$$ed(s^{i+1}, w^{j+1}) = \begin{cases} ed(s^i, w^j) & \text{(NO CHANGE)} \quad \text{if } s_{i+1} = w_{j+1} \\ 1+\min\{ed(s^{i-1}, w^{j-1}), & \text{(TRANSPOSITION)} \\ ed(s^{i+1}, w^j), & \text{(INSERTION)} \\ ed(s^i, w^{j+1})\} & \text{(DELETION)} \quad \text{if } s_i = w_{j+1} \text{ and } s_{i+1} = w_j \\ 1+\min\{ed(s^i, w^j), & \text{(CHANGE)} \\ ed(s^{i+1}, w^j), & \text{(INSERTION)} \\ ed(s^i, w^{j+1})\} & \text{(DELETION)} \quad \text{otherwise} \end{cases}$$

where:

$ed(s^0, w^j) = j$, for $0 \le j \le length(w)$,
$ed(s^i, w^0) = i$, for $0 \le i \le length(s)$,
$ed(s^{-1}, w^j) = ed(s^i, w^{-1}) = \max(length(w), length(s))$.

To explain how the edit distance is calculated let us consider an example:

s: *abcdef* length(*s*) = 6
w: *acb* length(*w*) = 3

			a	*ab*	*abc*	*abcd*	*abcde*	*abcdef*	
			6	6	6	6	6	6	-1
		0	1	2	3	4	5	6	0
a	6	1	0	1	2	3	4	5	1
ac	6	2	1	1	1	2	3	4	2
acb	6	3	2	1	1	2	3	4	3
	-1	0	1	2	3	4	5	6	

Table 1. Edit distances between various substrings of two strings.

Let us now look into calculations performed for a few selected cells in Table 1:

part of *w*	part of *s*	cell	Calculations	operation	condition
'*a*'	'*a*'	(1,1)	0 = 0	no change	if $s_{i+1} = w_{j+1}$
'*ac*'	'*a*'	(2,1)	1 + min(1,2,0) = 1	deletion	otherwise
'*ac*'	'*ab*'	(2,2)	1 + min(0,1,1) = 1	change	otherwise
'*acb*'	'*abc*'	(3,3)	1 + min(0,1,1) = 1	transposition	if $s_i = w_{j+1}$ and $s_{i+1} = w_j$

Naturally, if the value of *ed*(*s*, *w*) is small then the two strings are similar. We can see that strings *abc* and *acb* are similar (edit distance 1), while strings *abcdef* and *acb* are not (edit distance 4).

Obviously, when strings to be compared are large then a large amount of work is required (somewhat similarly to that required for the brute force string comparison algorithms [13]). More precisely, when strings of length *len*(*s*) and *len*(*w*) are considered, the total amount of work required to calculate their edit distance is equal to 4**len*(*s*)**len*(*w*). This being the case, one of the possible solutions is to apply parallel computers to reduce time of computations. Observe, however, that the process of establishing a "complete" edit distance between two

strings may not be exactly what is required. Often we are interested in finding **where** two strings are somewhat similar to each other (e.g. in the case of plagiarism detection). In this case, we would like to be able to establish a collection of similarity measures that describe the "localizable" similarity between two strings (see also Figure 1). This is the research problem pursued here. We assume that for two strings *w* and *s*, where *len*(*s*) >> *len*(*w*), we have to establish a semi-continuous localized measure of their similarity. We will therefore utilize the modified Levenshtein measure (edit distance) locally and report a collection of such measures to characterize the similarity between two strings. In the next section we discuss our approach used to parallelize computation of edit distance. In Section 3 we present and analyze results of our experiments on a 12 processor SGI Power Challenge computer and on a homogeneous cluster consisting of 23 PC's and on heterogeneous cluster consisting of 28 PC's.

2. Parallelization of computation of string similarity

For parallelization of the local-global edit distance computation we have selected the master-slave approach. As specified above, we assume here, that the length of *w* is substantially smaller than that of *s*. The master process(or) reads the two texts/strings (*s* and *w*) edit distance between which is to be computed. Next, it sends the shorter text *w* to all slave process(es/ors) (there must be at least one slave process since the master does not perform actual computations). Finally, the longer text *s* is divided by the master into parts of size:

$$len(w) + overlap$$

and successively sent to the slave processes (an *overlap* is usually about 30% of *len*(*w*), see below). This approach allows us to employ a simple form of dynamic load balancing, which is important particularly in the case when computations take place in a heterogeneous environment. Having received text *w* and "its" initial part of *s*, each slave process computes the edit distance and sends the result to the master process(or). In return it receives the next part of *s* to examine and continues to do so until all the remaining fragments have been checked. This process can be summarized in the following pseudo-code:

```
for each slave i
      master       → send an appropriate part of string s to slave i
      slave        → receive part of (s) and process it
while not end_of(s)
      slave j      → send an outcome of processing the part of (s) to master
      master       → receive an outcome from slave j
      master       → send a part of string s to slave j
      slave j      → receive part of (s) and process it
```

We have considered master process performing some calculations, but we came to the conclusion that it may aggravate potential load balancing problems as slave processes could have been forced to wait for the master to finish its work to be able to service their requests.

Let us now return to the description of the problem we are trying to address in this paper. We are not interested in calculating a complete measure between two strings (characterized by a single number, which due to the fact that the two strings considerably differ in length, would have to be large and would completely obscure the similarity characteristics of the two strings). Rather, we are interested in being able to represent similarity of two strings (to find out where, in the longer string, there exist parts similar to the shorter string). This being the case, beside strings *s* and *w*, we have to consider also the size of the *overlap* as well as the length of a step.

The size of the *overlap* determines the size of the examined fragment of *s*. For example, an overlap of 0.3 means that the length of a given fragments of *s* under consideration will be: length(*w*)*1.3. Observe that too small an overlap and thus too short fragments of *s* may lead to a situation in which the sought "similar" part of *s* will not fit into a single fragment and will

not be successfully located. Conversely, too big an overlap causes the differences between the edit distances for individual *s* fragments becoming negligible (just large numbers, see below – Figure 1 and discussion following it) thus hindering the interpretation of results.

The second parameter is the length of a step. It determines the distance between the beginnings of each pair of fragments. The smaller the step, the greater the accuracy of the results, but also the longer the calculation time. For instance, dividing the length of the step by two, we achieve twice as many fragments to compare. Our initial tests suggest that reasonable results are achieved for a step between 10% to 30% of *len*(*w*). Let us illustrate the above considerations by an example of partitioning of *s*:

s: I am just reading the best publication on comparing strings.
w: publish

partitioning of *s*:

I am just reading the best publication on comparing strings.
I am just reading the best publication on comparing strings.
I am just reading the best publication on comparing strings.
I am just reading the best publication on comparing strings.

and so on…

In this example we used an overlap in the size of two characters (~ 30% of *w*) and the step equal to three. Let us also consider a practical example of execution of our algorithm. Figure 1 depicts the result obtained for texts of *len*(*s*) = 2556 and *len*(*w*) = 397 characters.

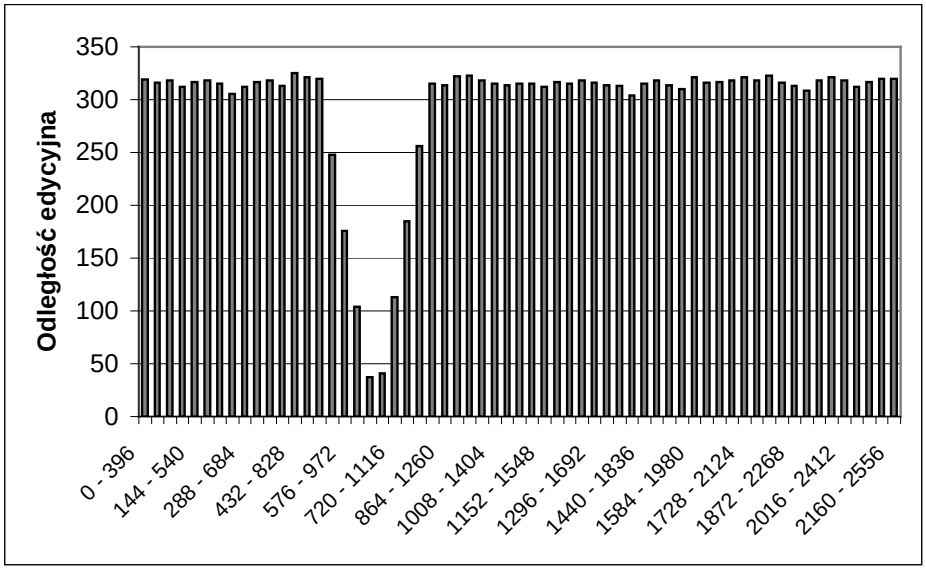

Figure 1. Graphical representation of string similarity

From Figure 1, it can be concluded that the fragment of *s* between 648th to 1044th character of *s* is very similar to *w*. It might seem strange that the smallest edit distance is 37. This follows from the assumed overlap as well as from the small differences between those texts. On the other hand, the biggest edit distance is not 397 and this is explained by the fact that there are always some common words or at least parts of them or groups of letters that appear in both texts. It follows from our tests that taking two random texts of the length *l*, the average value of edit distance would be approximately 80% of *l*.

In general, from the point of view of implementation, calculating edit distance for two given texts *w* and *s* consists in filling in an array of the size: length(*s*)*length(*w*). Examining the formula of *ed* (Section 1) we can see that each element depends on four others. As we are exclusively interested in the extreme bottom right corner value in the array, i.e. in our case the distance between the substring of *s* and the string *w*, and not the intermediate values, only three rows of the array have to be allocated and used cyclically. This leads to a considerable reduction in the amount of memory needed for the calculations.

As it should be obvious by now, the result of our program is an array containing data which reflects the edit distances between the shorter text *w* and the fragments of the longer text *s*. By choosing the smallest value (shortest distance), it can be determined whether and, if so where, *s* is most similar to *w* (the two texts are most similar).

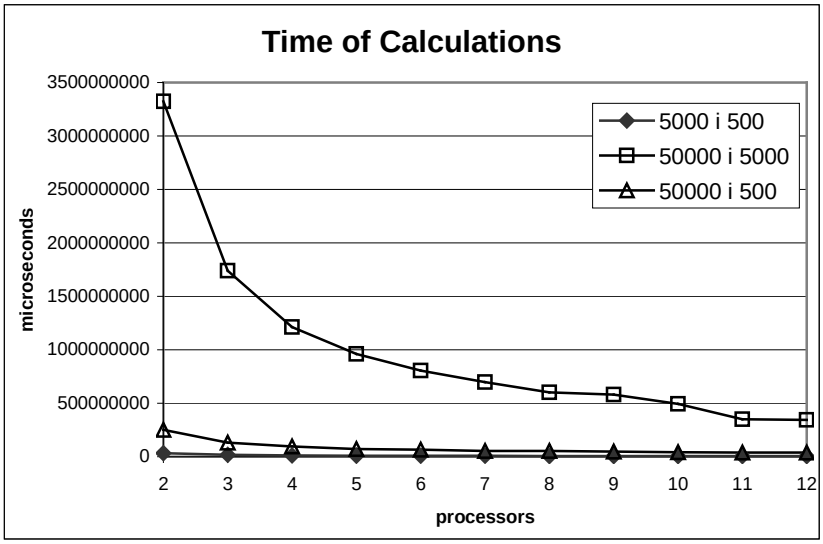

Figure 2. Performance on the SGI Power Challenge; computation time

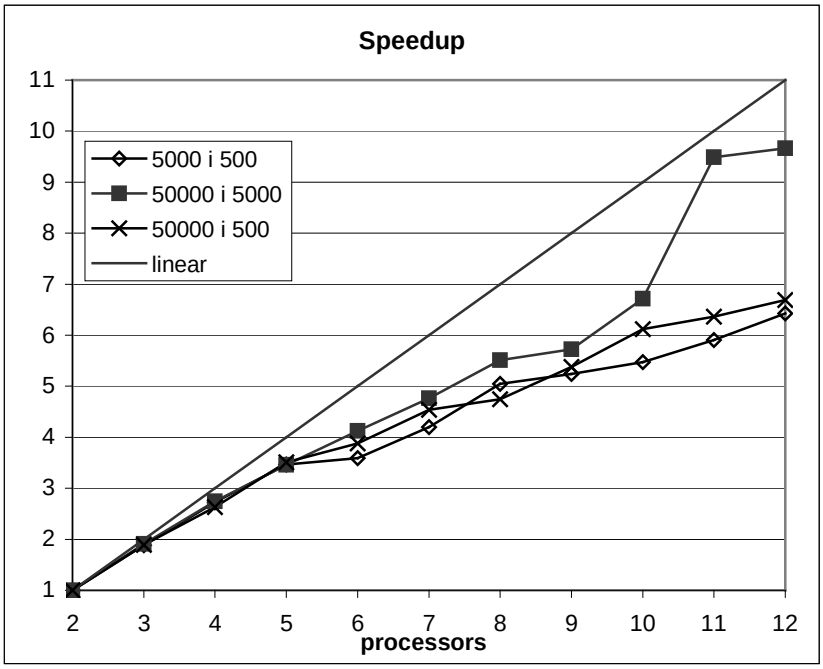

Figure 3. Performance on the SGI Power Challenge; speedup

3. Experimental data

The first series of experiments was performed on a 12 processor SGI Power Challenge XL. It is a 10 years old, shared memory machine, with 90MHz MIPS R8000 processors and 1GB of RAM. Code was implemented in C and the SGI C compiler was used with -64 -mips4 -r8000 -O3 options. Parallelization was achieved through the MPI communication library and SGI's native MPI was used. We have run three series of experiments for strings of length 5000 and 500, 50000 and 5000 and 50000 and 500. In Figure 2 we present computation time for all three cases for 1-11 slave processors, while in Figure 3 we present obtained speedup.

The second series of experiments was performed on a cluster of 23 1.5 GHz Pentium 4 based PCs, with 256 Mbytes of memory each. Computers were connected via a Catalyst 6500 switch (100mbit/s, full duplex). Figures 4 and 5 present the execution time and speedup obtained on that machine.

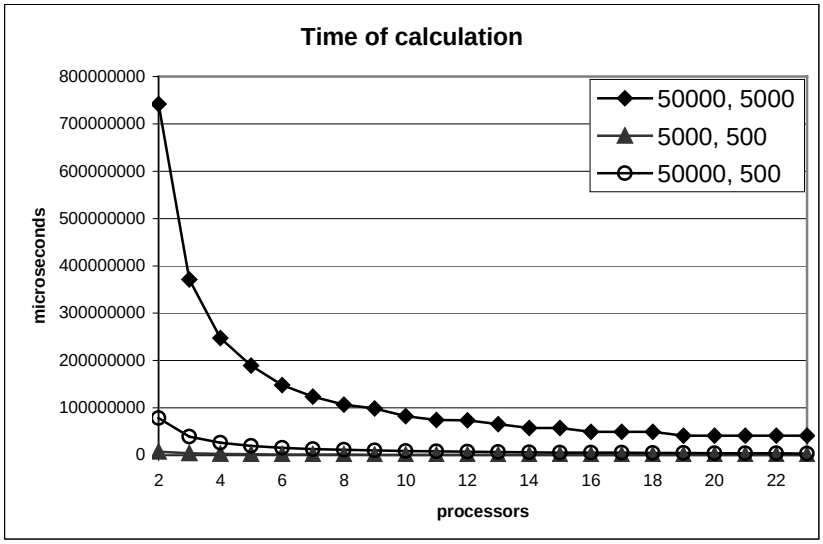

Figure 4. Performance on the homogeneous cluster; computation time

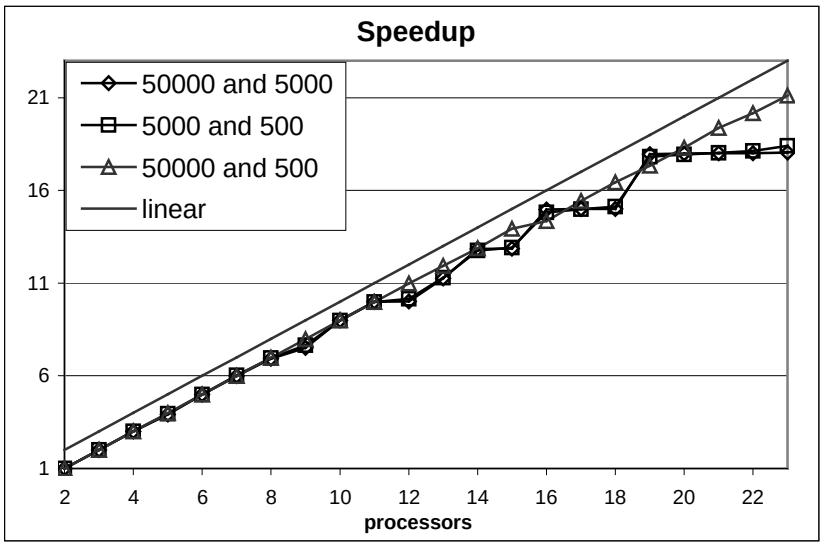

Figure 5. Performance on the homogeneous cluster; speedup

The final series of experiments was performed on a heterogeneous cluster of 18 computers with P4 1.5 GHz processor and 256 Mb of RAM and 10 computers with P4 1.8 GHz processor and 512 Mb of RAM. These computers were connected through a pair of switches similar to that used to set-up the heterogeneous cluster. Experiments were performed with the largest

test case (5000/50000) only. Figures 6 and 7 present the execution time and speedup obtained on that machine.

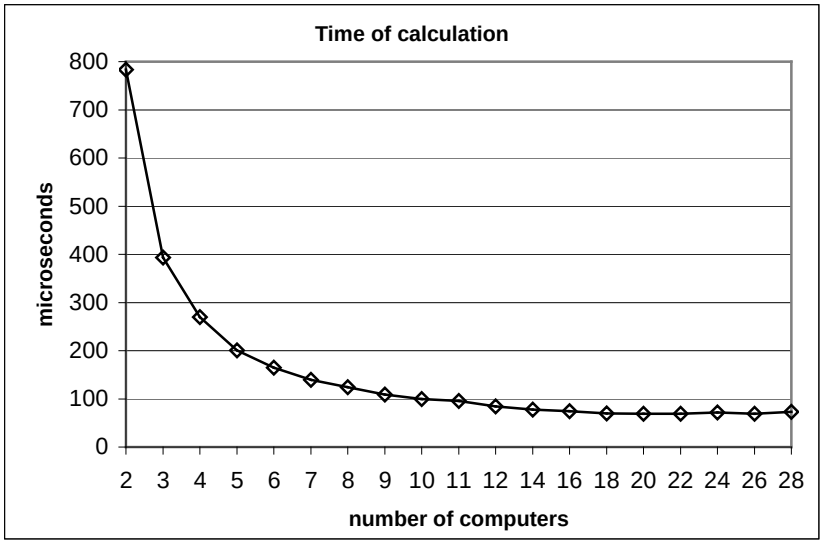

Figure 6. Performance on the heterogeneous cluster; computation time

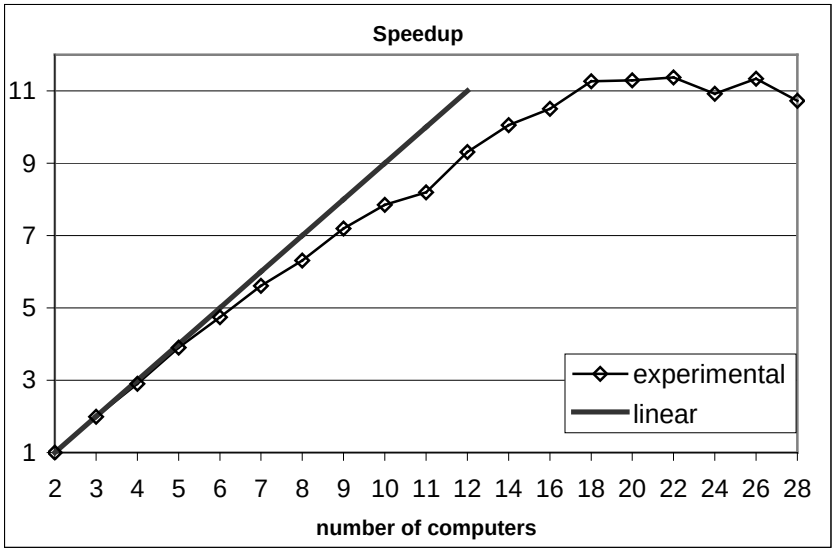

Figure 7. Performance on the heterogeneous cluster; speedup

In all of our tests we obtained a reasonable speedup. For small number of processors it is almost linear. The best results were obtained on the homogeneous cluster. It is worth mentioning, that these results were obtained late at night on an empty machine that was set-up for the purpose of our experiments. The results collected on the SGI Power Challenge have been collected on a relatively full machine. This may explain the relatively unimpressive speedup obtained for the larger number of processors. The only situation where the results are not particularly impressive is the case of the heterogeneous cluster. Here we had to deal with two levels of heterogeneity; first, two different types of PC's and, second, with a double switch providing the infrastructure for the cluster. Clearly, the results are acceptable for up to 12 computers. As the number of machines increases, we can see that the speedup stops increasing. This effect will require further studies. Furthermore, to improve the performance we need to consider a more refined approach to load balancing.

6. Concluding remarks

In this paper we have discussed how parallel computers can be utilized to establish similarities between two strings. The proposed approach was based on local utilization of modified Levenshtein measure and obtaining a semi-continuous picture of the relationship between two strings. We have implemented the proposed algorithm and experimented with it on three different parallel computers. In all cases we have obtained a reasonable efficiency.

In the near future we plan to pursue a number of additional questions. First, for the shared memory computer we plan to experiment with an OpenMP based implementation. Second, we need to refine the load balancing strategy to improve the performance of the algorithm on heterogeneous clusters. Third, we plan to study in detail the effects of the size of the overlap and the step in the case of large texts. Finally, we will try to apply computation of edit distance to finding themes in baroque fugues.

Bibliography

[1] M. Wypych (2002) *Stochastic Spelling Correction of Texts in Polish*, Institute of Linguistics, Adam Mickiewicz University, Poznań, Poland; "Speech and Language Technology. Volume 6", Poznań

[2] J. Nerbonne, W. J. Heeringa, P. Kleiweg (1999) *Edit Distance and Dialect Proximity*, in: D. Sankoff and J. Kruskal (eds.), *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequency Comparison*, CSLI, Standford, pp. v-xv.

[3] http://www2.toki.or.id/book/AlgDesignManual/BOOK/BOOK5/NODE204.HTM

[4] A. Scherbina, *Application of Levenstein Metric to Web Usage Mining*, Institute for System Programming, Russian Academy of Science, Proceedings of the 7th BIS Conference, Poznan University of Economics Press, to appear

[5] D. E. Knuth, J. H. Morris Jr., V. R. Pratt (1977) *Fast pattern matching in strings*, SIAM J. Comput., 6(1), 323-350

[6] R. S. Boyer, J. S. Moore (1977) A fast string searching algorithm, CACM, 20, 762-772

[7] R. M. Karp, M. O. Rabin (1987) *Efficient randomized pattern-matching algorithms*, IBM J. Res. Dev., 31(2), 249-260

[8] A. H. Wright, *Approximate String Matching using Within-Word Parallelism*, Computer Science University of Montana

[9] http://www.cs.mu.oz.au/~mjl/thesis/node17.html

[10] http://www2.toki.or.id/book/AlgDesignManual/BOOK/BOOK2/NODE46.HTM

[11] V. I. Levenshtein (1966) *Binary codes capable of correcting deletions, insertions and reversals*. Soviet Physics, Doklady 10(8), 707-710

[12] J. Daciuk (1998) *Incremental Construction of Finite-State Automata and Transducers, and their Use in Natural Language Processing*, Ph.D. Thesis, Technical University of Gdansk, Poland.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein (2001) *Introduction to Algorithms*, (second edition) MIT Press, Cambridge, MA