A comparison of two algorithms for discovering repeated word sequences

R. Tesar[1], D. Fiala[1,2], F. Rousselot[2], K. Jezek[1]
[1]Department of Computer Science and Engineering, University of West Bohemia in Plzen, Czech Republic.
[2]Laboratory of Computer Science and Artificial Intelligence, University Louis Pasteur – Strasbourg I, France.

Abstract

We will make the readers of this paper familiar with two basic approaches to repeated sequences extraction – a suffix tree based method and an inverted list based method. The first algorithm (ST) makes use of a tree data structure known from suffix tree clustering (STC) where each node represents one word and the root represents the null word. Thus, each path from the root is a phrase occurring somewhere in the corpus. The second applies an inverted index (broadly used in information retrieval), more specifically its hash table implementation (HT). Occurrences of each word in this index are employed to construct lists of words following the current word in different places of the corpus. These lists are then modified in a recursive fashion to satisfy the constraints of repeated segments. We will introduce both methods and compare them in terms of their time and space complexity, efficiency, effectiveness and results yielded with some sample input data. We will conclude that whereas the ST-algorithm is better as to time cost, it is outperformed by the HT-approach with respect to space. Finally, we will suggest several possible applications of repeated sequences.

Keywords: repeated sequences, repeated segments, frequent phrases, suffix tree, algorithms, comparison.

1 Introduction

Repeated word sequences, sometimes referred to as repeated segments, phrases, co-occurrences, or collocations, are simply sequences of words that occur more than once in a text or corpus. Whether all words in the text are taken into account or some of them are omitted (via stoplists) is implementation dependent. More generally, we may think of symbols instead of words. Then, repeated sequences of symbols are patterns representing the original data from which they were obtained. In this general sense, repeated segments extraction and application reach far beyond the scope of computational linguistics and text mining. They are an important means of clustering, classification, topic detection and other machine learning and artificial intelligence techniques.

2 Suffix tree

A suffix tree (ST hereafter) is a data structure that allows many problems on strings to be solved easily and quickly. It can be used to solve a large number of string issues that occur in text editing, searching and other application areas. The suffix tree originates in the 70s (see Weiner [5]), but only in the 90s it was modified for ST-phrases (repeated sequences) retrieval (see Zamir [3]).

Since this method is natural language independent, there is a possibility to manipulate with documents in many languages simultaneously (see Grolmus [1]). Other advantages are the document processing order independency and the possibility to regulate the length of retrieved phrases that is straight dependent on the adjusted suffix tree level. There exist many implementations (see Sandeep [6], Ukkonen [7] or Andersson [8]) suitable for various purposes. We had been inspired by Zamir [3], Sandeep [6], Ukkonen [7] and Andersson [8] when we implemented the procedure of creating a ST-structure we present below.

The general algorithm for the suffix tree structure construction convenient for discovering frequent phrases is similar to the algorithm depicted in Zamir [3].

In its simplest version, the suffix tree algorithm creates a tree structure that contains words in the order corresponding to their positions in the input documents. Each node of the suffix tree represents one word and the root represents the null word. Thus each path from the root represents a phrase containing the words labeling the nodes traversed. The suffix tree is a rooted and directed tree. For example, we want to create a suffix tree structure using the following sentences

"can drive trucks safely. men drive cars safely. men can drive trucks"

and we define the maximum tree height (thus the maximum phrases length) $m = 3$. The structure created in this way is shown in Figure 1.

When the suffix tree structure is created, the simplest way of obtaining single frequent phrases is to use a recursive procedure traversing the tree from the root node to leaf nodes. A non-recursive implementation is a relatively complex problem and there is no guarantee that we can achieve a better efficiency (while having the same complexity), because it depends on the current compiler implementation.

The frequency of phrases is determined by the node that represents the last word in a particular phrase. From the set of nodes representing a particular phrase, this node is always situated at the bottom level of the suffix tree structure (when we consider the root as being at the top).

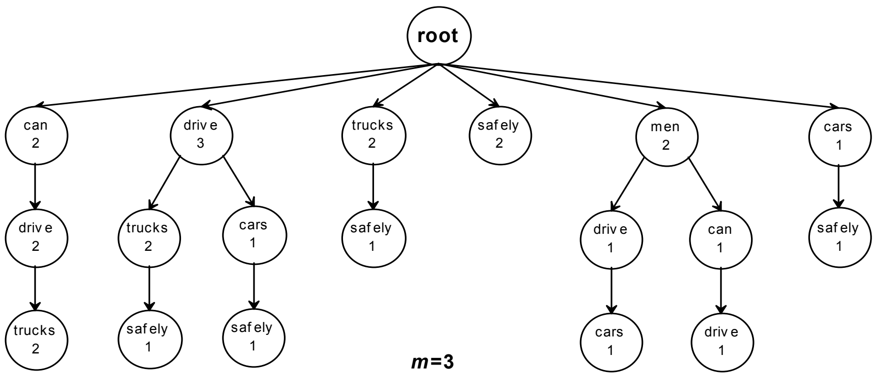


Figure 1: Suffix tree structure example

2.1 A detailed description of the implemented ST-algorithm

The implemented suffix tree algorithm pseudo code is presented in Figure 2. Time complexities for each partial task are shown as well.

At the beginning, an input corpus is tokenized and stored to *split list* in such a manner that particular words are stored consecutively and sentences (or their parts bordered by punctuation etc.) are delimited by space (see Figure 3). To do this, we used regular expressions which allow defining words and sentences delimiters. The *split list* is then used for the suffix tree structure construction.

First of all, a root node has to be created. The root node is special, because it does not represent any word. Then a word from the *split list*'s first position is read and added to a hashtable of unique words in order to accelerate access to stored words.

```
create a root node of ST-structure;                                  O(1)
while (input is not empty)
{
clear parent list;                                                   O(N)
while (get next input word != sentence delimiter)
{
    add word to hash table;                                          O(N)
    add word node to root node;                                      O(N)
    add reference to parent list;                                    O(N)

    for each node from parent list (except the last one added)
    {
        add word node to current node;                               O(N*m)
        add reference to parent list;                                O(N*m)
        remove currently processed node from parent list;            O(N*m)
    }
}
}

{
traverse created ST-structure recursively to obtain ST-phrases       O(N)
}
```

Figure 2: Pseudo code of Suffix Tree algorithm

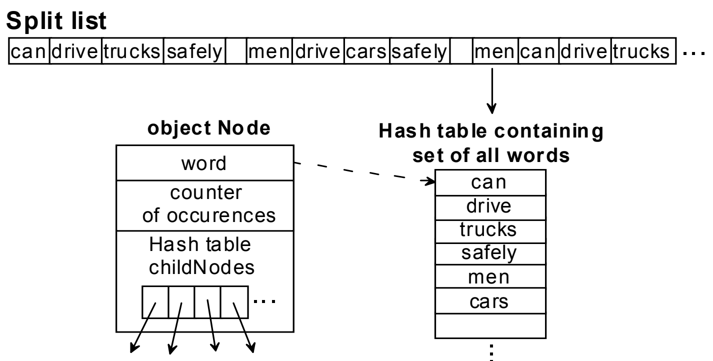


Figure 3: Structure of a Node object

Each of the suffix tree nodes represents just one input corpus word in form of a reference to the hash table, which is created in parallel. In this way we can considerably reduce the total memory requirements since single words at a suffix tree level deeper than one are repetitious. Nodes also contain information on how many times the phrase they represent occurs in the input corpus and they provide a list of children *childNodes* (see Figure 3).

So if a word from the *split list* is read and added to the hash table, the root node has to be examined whether it contains a node that represents the current word because it is a parent node every time. If there is such a node, the counter of occurences increases. Otherwise, a new node representing the current word is created and added to the list of children of the root node. In both cases, the node representing the current word is added to the end of the parent list.

Now we process the *parent list* which contains nodes added in the previous step (see Figure 2). Because these nodes (except the last added node - it is ready

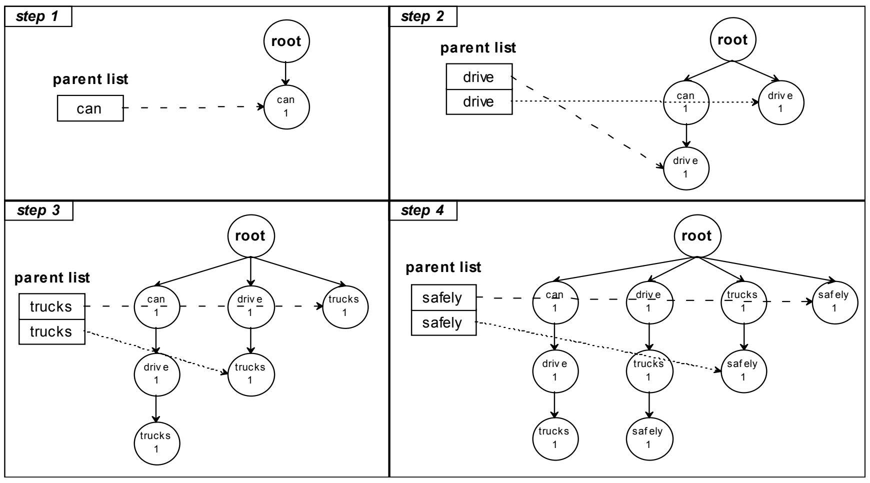


Figure 4: Suffix tree structure construction example using *parent list* (m = 3)

for the next step) are parents of the current word (that means they represent the word that occurs before the current word in the input corpus), we can simply do the same we did in the case of the root node. The parent nodes processed are removed from the *parent list*. Naturally, a node representing the current word cannot be added to the *parent list* if its level is equal to the maximum suffix tree level.

After that, a next word is read and the procedure described above repeats until the end of the *split list* is reached. Figure 4 depicts the first four suffix tree construction steps using the sentence "can drive trucks safely" with the maximum level m set to three.

2.2 Complexity

As we can infer from Figure 2, where N is the number of all words in the input corpus and m is the maximum suffix tree level, the time complexity is O(N + N*m + N).

The space complexity of storing the tokenized text to memory is O(N), the hash table with unique words O(K), where K is number of unique words in the input corpus, all the suffix tree nodes O(N*m) in the worst case, and all the ST-phrases obtained O(N*m), in general.

Thus, the total time and space complexities are both linear (see Figure 8), which is conforming to Zamir [3].

3 Repeated segments

Various authors used the term "repeated segments" in their papers among which we shall mention Justeson and Katz [9] and Lebart and Salem [10]. Oueslati [11] employed repeated segments to acquire knowledge from a corpus. In those works, however, repeated segments had a more specific meaning and they were limited in definition by several constraints. They were thought of as term candidates for discovering new concepts and domains in text corpora. Thus, each term candidate had to contain a noun. Whether or not to include articles, prepositions, etc. in the term candidate was already application dependant. Of course, to detect nouns and exclude verbs, for instance, we need some external resources such as different filters, etc. Another problem with repeated segments employed for concepts discovery is that they don't work for all languages. For instance, the German language, which creates new words by simply concatenating existing words, doesn't enable finding repeated segments in the strict sense. The newly created word can occur no matter how many times in the corpus but, as it is a single word, it is never considered a concept.

A tool that was developed on the basis of Oueslati's work is called LIKES [12]. It is designed to allow for a number of linguistic tasks. To refine the discovery of term candidates (repeated segments) it makes use of a couple of filters such as a cutting filter with a list of words that can't be part of a term candidate (mostly verbs and conjunctions), a grammatical filter with words that tend to introduce a segment (articles and pronouns) and so forth. These filters, of course, are language dependant and they must exist for each language in the corpus. LIKES itself is somewhat heavy weighted; it builds a tree of objects (paragraphs, sentences, words) for the whole corpus. However, it keeps track of segment contexts and relations. For instance, it is possible to find out where in the corpus (which text, paragraph and sentence) a particular segment is located. In the text below, repeated segments will mean pure repeated sequences without the limitations mentioned in the previous paragraph.

3.1 Algorithm

The algorithm pseudo code is presented in Figure 5. Time complexities for each partial task are shown as well. Explanation will be given further.

Our algorithm for discovering repeated segments doesn't have the necessity to take account of various filters because it considers each token in the corpus (here the tokens are words but they might be whatever symbols as well) as equal in terms of their morpho-syntactical meaning. Figure 6 shows the most significant stages of the algorithm. We will follow the steps in the algorithm and give an example of its application. Consider the following input text (corpus):

Personal construct psychology. Personal construct psychology. Personal construct theory. Personal construct technology.

Pre-processing, i.e. tokenization, normalization, etc. of the input corpus is performed outside the algorithm, so the actual input is then an array of tokens (words). We create an inverted index by adding each word to a hash table along with a list of its occurrences, i.e. with references to the corresponding places in the corpus where those words occur. Afterwards, with each word in the index we do the same operations. For instance, for the word personal, we look up its occurrences and create a list of words which follow this word at different places (see phase 1 in Figure 6). The list will be sorted and only those words will remain that are part of sequences that occur two or more times. The other words are removed (see phase 2 in Figure 6). Next, duplicate sequences are removed and the unique ones are counted in phase 3. Left subsequences of all sequences

```
// reads tokenized text and updates inverted index
foreach word in text
    add to hash table;              O(N)

foreach key in hash table           O(K)
{
    // creates list of words following each occurrence of key word
    get following words;            O(V)
    // modifies list so as only those sequences remain that occur twice at least
    reduce words list;              O(V + VlogV)
    // counts newly discovered segments and adds them
    remove empty columns;           O(V)
    remove duplicates;              O(V)
    add inclusions;                 O(V + VlogV)
    add segments;                   O(V)
}
```

Figure 5: Repeated segments – algorithm pseudo code

are added to the list in phase 4 and the same as in phase 3 is done in phase 5. Finally, the newly discovered segments are added to the list of repeated segments (phase 6). The whole process is repeated for another word in the inverted index. The sequence *construct psychology*, for example, will be found when processing the word construct. In theory, the maximum length of a repeated segment could be a half of that of the whole corpus. However, it is advisable to set it to a reasonable constant (in the example above the constant is three). Segments longer than this constant will be ignored.

3.2 Complexity

Partial time complexities of individual operations are already noted in algorithm pseudo code in Figure 5 where N is the number of words in the corpus, K is the number of unique words in the corpus (i.e. items in the hashtable) and V is the average number of occurrences of a word in the corpus. The evident time complexity of the whole algorithm in terms of these three variables is

$$O(N + K(V \log V + V)) \tag{1}$$

Obviously, V is N/K. Thus, substituting in (1) we get

$$O(N + K((N/K)\log(N/K) + N/K)) \tag{2}$$

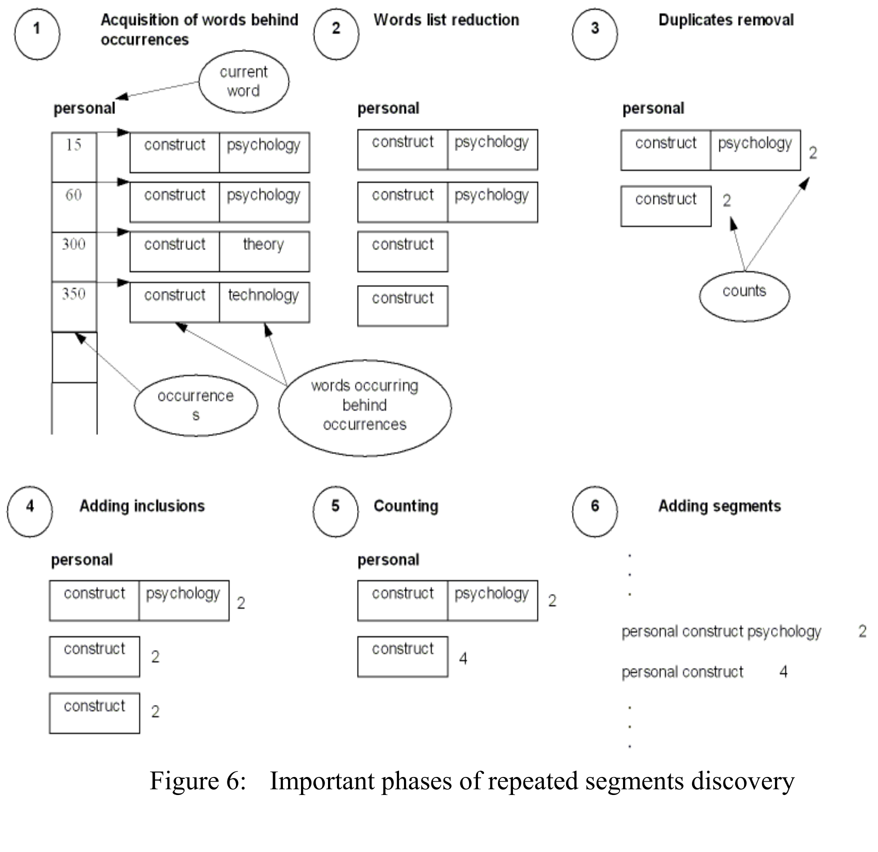


Figure 6: Important phases of repeated segments discovery

and further

$$O(N + N + N\log(N/K))\,. \tag{3}$$

If K was N (i.e. each word in the corpus was different), the simplification to O(N) would be straightforward. But as K rather tends to be logN (see Figure 7 and Zipf's law [14]), the total time complexity is supralinear; that means

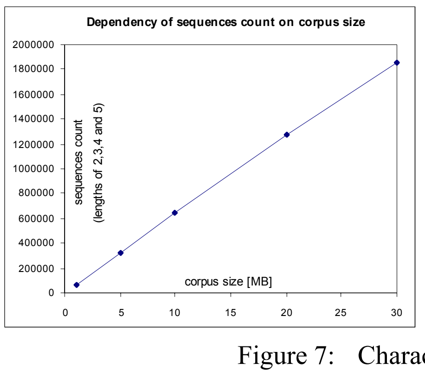
$$O(N\log N)\,. \tag{4}$$

If we focus our attention on the space complexity, we observe that keeping the tokenized text in memory is O(N), the hash table with occurrences, i.e. the inverted index, is O(N) (the number of all occurrences can never be larger than N) and the list of segments is O(N). Thus, for the overall space complexity we get O(N). We can conclude saying that the algorithm for discovering repeated segments presented above is supralinear in time and linear in space.

4 Results and conclusions

We ran each algorithm on five different corpora whose size ranged from 1 MB to 30 MB approximately, in terms of the total number of words up to several million words. We were interested in sequences of maximum lengths of three, four and five and in time and memory consumed by each algorithm. Both algorithms discovered all word sequences occurring in every single corpus, thus the sequences found were identical in either case. See Figure 7 for the characteristics of used corpora. Charts in Figure 8 confirm the theoretical assumptions we made about the algorithms' complexities. ST algorithm is linear in time as well as in space, whereas RS algorithm is supralinear in time and linear in space. However, the space consumption by ST algorithm is about twice as high as that by RS algorithm. (We implemented both algorithms in C#, compiled them on .NET Framework 1.1 and ran them on AMD Opteron 1.6 GHz with 2 GB RAM.)

In this paper we showed two algorithms for finding repeated sequences of words and there are many good reasons why we should be interested in those repeated sequences (and not always of words). For instance, they may be useful for classification and clustering. We might apply a standard method (Han [2]) but as a similarity of two objects we would use the ratio of common repeated



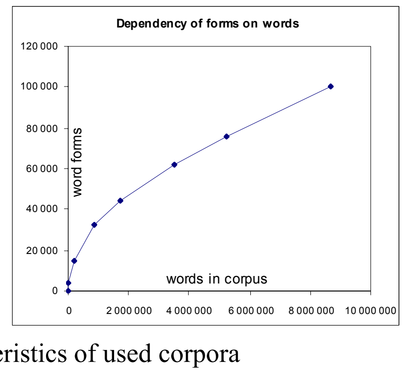


Figure 7: Characteristics of used corpora

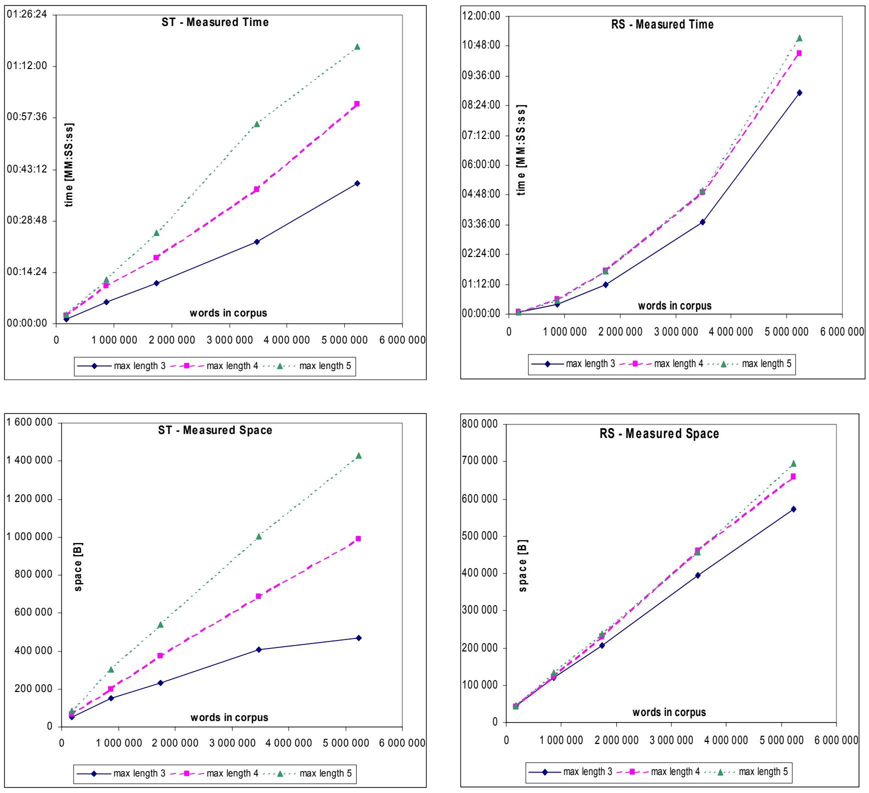


Figure 8: Results achieved with both algorithms on sample corpora

sequences. Also, repeated sequences are usable for topic detection in sets of text, topic evolution and other text mining tasks (e.g. Feldman [13]). Obtaining repeated sequences as a by-product of a compression algorithm called SEQUITUR is described in Nevill-Manning [15]. Possible applications include spam filtering [16], Web users profiles generation [17], compression [18], document indexing [19], statistical linguistics [20] and others.

Experiments with both of the methods described in this article underlined the need for a general tool for linguistic tasks suitable for very large text corpora, which we would be happy to deal with in the future.

This work was supported by the Ministry of Education of the Czech Republic – project FR VS 1347/2005/G1.

References

[1] Grolmus P., Hynek J., Jezek K.: User Profile Identification Based on Text Mining. *Proc. of 6th Int. Conf. ISIM'03*, MARQ Ostrava, pp. 109-118, ISBN 80-85988-84-4, 2003.

[2] Han J., Kamber M.: *Data Mining: Concepts and Techniques*, Morgan Kaufmann Publishers, 550 pages. ISBN 1-55860-489-8, August 2000

[3] Zamir O., Etzioni O.: Web document clustering: A feasibility demonstration. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (*SIGIR '98*, Melbourne, Australia, Aug. 24–28), W. B. Croft, A. Moffat, C. J. van Rijsbergen, R., Wilkinson, and J. Zobel, Chairs. ACM Press, New York, NY, 46–54, 1998.

[4] Zamir O., Etzioni O.: A dynamic clustering interface to web search results. In *Proceedings of WWW8* (Toronto, Canada), 31 (11-16): 1361-1374, 1999

[5] Weiner P.: Linear pattern matching algorithm, In *Proceedings of 14th IEEE Symposium on Switching and Automata Theory*, 1-11, 1973.

[6] Sandeep T., Hankins R. A., Patel J. M.: Practical Suffix Tree Construction. *VLDB 2004*: 36-47, 2004.

[7] Ukkonen E.: On-line construction of suffix-trees. *Algorithmica 1995*; 14(3):249–260, 1995.

[8] Andersson A., Nilsson S.: Efficient Implementation of Suffix Trees. *Software–Practice and Experience (SPE)*, 25(2):129–141, 1995.

[9] Justeson J., Katz M.: "Technical terminology: some linguistic properties and an algorithm for identification in text", *Natural Language Engineering*,. Vol. 1, No. 1, pp. 9-27, 1995.

[10] Lebart L., Salem A.: "Statistique textuelle", Dunod, 1994.

[11] Oueslati R., Frath P., Rousselot F.: "Term Identification and Knowledge Extraction", *International Conference on Applied Natural Language and Artificial Intelligence*, Moncton, New Brunswick, Canada , 1996.

[12] L'outil de traitement de corpus LIKES, http://www-ensais.u-strasbg.fr/liia/LIIA_Products_Installers/install.htm

[13] Feldman R., Fresko M., Kinar Y., Lindell Y., Liphstat O., Rajman M., Schler Y., Zamir O.: "Text Mining at the Term Level", *Proceedings of Principles of Data Mining and Knowledge Discovery*, pp. 65 – 73, 1998.

[14] Zipf G. K.: "Human Behaviour and the Principle of Least-Effort", Addison-Wesley, Cambridge MA, 1949.

[15] Nevill-Manning C. G., Witten I. H.: "Compression and Explanation Using Hierarchical Grammars", *The Computer Journal*, Vol. 40, No. 2/3, pp. 103 – 116, 1997.

[16] Corvigo, http://www.corvigo.com/media/news/20040102.html

[17] Grolmus P., Hynek J., Jezek K.: "Finding frequent phrases for user profiles generation" (in Czech), Proceedings of ITAT 2003, pp. 100 – 108, 2003.

[18] Grabowski S.: "Text preprocessing for Burrows-Wheeler block sorting compression", In *Sieci i Systemy Informatycne - teoria, projekty, wdrozenia*, Lodz, 1999.

[19] Hammouda K. M., Kamel M. S.: "Efficient Phrase-Based Document Indexing for Web Document Clustering", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 16, No. 10, pp. 1279 - 1296, 2004.

[20] Stubbs M.: "On very frequent phrases in English: distributions, functions and structures", *25th anniversary meeting of ICAME*, Verona, 2004.