Ternary Tree Optimalization for n-gram Indexing

Daniel Robenek, Jan Platoš, Václav Snášel

Department of Computer Science, FEI, VSB – Technical University of Ostrava,
17. listopadu 15, 708 33, Ostrava-Poruba, Czech Republic
{daniel.robenek, jan.platos, vaclav.snasel}@vsb.cz

Abstract. N-gram indexing is used in many practical applications. Spam detection, plagiarism detection or comparison of DNA reads. There are many data structures that can be used for this purpose, each with different characteristics. In this article the ternary search tree data structure is used. One improvement of ternary tree that can save up to 43% of required memory is introduced. In the second part new data structure, named ternary forest, is proposed. Efficiency of ternary forest is tested and compared to ternary search tree and two-level indexing ternary search tree.

Keywords: n-gram, ternary tree, ternary forest, inverted index

1 Introduction

Efficient indexing and searching in huge amount of data is big issue in computer science. For example finding plagiarisms, spam detection or comparison of DNA sequences are topics, where efficient indexing is a key element of fast software.

The piece of data in these problems can be called n-gram. For DNA sequences n-grams are nucleotides in the sequence read. For plagiarism and spam detection n-grams are words in sentences or characters in the words.

First problem is to efficiently extract these n-grams. There are many specific and optimized algorithms for this purpose. Next is necessary to perform the indexing.

Many data structures for this purpose, with different efficiency in search, insertion or memory requirements are known. Indexing can be divided into two main categories, depending on available memory or amount of data. First, when amount of the data exceeds available memory, the data are stored on hard drive. Data structures like B+ tree are optimized for this purpose.

Second category is in-memory based indexing, which expects sufficient amount of memory for this purpose. The article is mainly focused to second category, specifically to ternary tree optimization for n-gram indexing.

J. Pokorný, K. Richta, V. Snášel (Eds.): Dateso 2014, pp. 47–58, ISBN 978-80-01-05482-6.

2 Related Work

The path from plain text documents to ready-made search engine is long and difficult. The first problem is to extract required n-grams out of documents into required format. In case of spam detection or plagiarism detection we are interested in n-grams that occur at last m-times [8]. It is because we are comparing similarity of documents, respectively the most repeated parts of them.

In case of extremely large amount of documents the common data structures like hash tables or trees are not suitable, because they would probably not fit in the memory. Therefore another approach is inevitable. By using sophisticated algorithms and hard drive as a temporary storage high efficiency can be achieved without necessity of having large amount of RAM [4]. For smaller sets of data the utilized data structures can be used [6].

A number of data structures have been proposed for text search and inverted index is the most used one [1]. For example inverted index is used to evaluate queries in many search engines [7]. The optimization and compression can be used on inverted index data structures in order to speed up the search and reduce memory requirements [10,5].

There are many data structures that can do n-gram indexing in memory [9]. One opportunity how to create in memory inverted index is to use ternary search tree in which every node stores information about one n-gram character. As it was shown by tests on collections Google WebIT and English Gigaword corpus, the data structure is fast enough [3].

However storing the whole n-gram into single data structure as-is may not be optimal. Repeated words should not be stored many times. Redundant presence of words causes excessive memory consumption. The idea is to create two data structures where the words in n-grams are at first converted to unique numbers and only after that the numbers are processed by data structure [6,7].

Using two-level inverted index can considerably decrease memory consumption [9,2].

One of the requirements for these types of data structures is the opportunity to use wildcard placeholders. This is used when is necessary to look for particular similarity. When using two-level inverted index it is necessary to find range of words on the first index. However this is only efficient when indexes are sorted with the words. When this request is fulfilled, it is easy to look up for these words using data structures like B+ tree [2].

3 Tree Compression

When using general binary or ternary tree, every unigram is contained in separate part of tree named node. The node contains necessary tree parts, parts that are specific for different kinds of each type of the tree and the unigram key and n-gram value.

Unigram key can be represented by *char* data type in case of indexing words. The pointer can be also used when needed. Situation is similar for n-gram value. When we

are not interested in value, for example when use tree as a data set structure, the value can be omitted.

Data are partly specific to different tree implementations. Red-black tree needs to contain color value and each node of AVL tree should contain depth information. These information are necessary for self-balancing efficiency.

Last parts of tree node are references to next tree nodes. For binary tree, two pointers are needed. For ternary tree, one more pointer is needed. The question is, if these references are necessary. Binary tree nodes can be divided into two types - internal nodes and external nodes (leaf nodes). Internal nodes are those having at last one child. On the other hand, external nodes have no children.

In following sections, the ternary red-black tree will be used. New method for unused ternary tree references removal will be introduced and results of test will be shown.

3.1. Binary Tree

For memory saving computation is necessary to exactly define how both internal and external node should look like. Comparison is shown in Table 1. For computations the key variable is 8-bit character type and the value is 32-bit integer. References are also 32-bit integers.

Table 1. Composition of binary tree nodes

Variable	Internal node	External node
Left reference	4 bytes	0 bytes
Right reference	4 bytes	0 bytes
Color of node	1 byte	0 bytes
Key	1 byte	1 byte
Value	4 bytes	4 bytes
Sum of size	14 bytes	5 bytes
Real size	16 bytes	8 bytes

The size of external node in extreme case is only ~36% of internal node. The real test showed different values. Because of memory alignment, size of internal node is 16 bytes and external node size is 8 bytes. But the 50% is still large difference. The real size may vary, depending on the *Key* and *Value* variable sizes.

To compute memory saves of tree is necessary to know amount of internal and external nodes. Lets think about ideal binary tree, where every node has zero or two child nodes. The relative number of internal is:

$$c_{ne} = 2^{h-1}$$

$$c_{ni} = 2^{h-1} - 1$$

$$c_{ne} = c_{ni} + 1$$

Where c_{ne} is amount of external nodes, c_{ni} is amount of internal nodes and h is height of tree. If we consider only large trees, we can assume that

$$c_{ne} = c_{ni}$$

with negligible error. Now, for this type of tree and sizes of nodes mentioned in Table 1, we can compute memory saving for whole tree. Without optimization, the size of the node is 14 bytes. With optimization, the size of average node is ~9.5 bytes. It means ~32% of saved memory.

If we consider also memory alignment, the results are little different. Node size is 16 bytes and with optimization, average node size is 12 bytes. It results in 25% less memory usage.

This was one, ideal type of binary tree. But we can generalize these results for every binary tree. The amount of used references in binary tree is always same, no matter of tree arrangement. This amount is exactly same as number of nodes minus one, because root node has no reference to its parent node.

If we remove every unnecessary reference in the tree, the real saved memory would be about 25%, depending on *Key* and *Value* data type.

3.2. Ternary Tree

On Table 2 we can see node sizes of ternary search tree.

Table 2. Composition of ternary tree nodes

Variable	Internal node	External node
Left reference	4 bytes	0 bytes
Right reference	4 bytes	0 bytes
Middle reference	0 / 4 bytes	0 / 4 bytes
Color of node	1 byte	0 bytes
Key	1 byte	1 byte
Value	4 bytes	4 bytes
Sum of size	14 / 18 bytes	5 / 9 bytes
Real size	16 / 20 bytes	8 / 12 bytes

The difference is middle reference, which can enlarge node size of 4 bytes. Therefore minimum size of node is 5 bytes for leaf with no middle reference, and maximum is 16 bytes for internal node with middle reference or 8 bytes and 20 bytes for real allocated size.

If root node reference is omitted, there are 4 bytes per node for reference and 6 bytes for data. Therefore average amount of memory for one node of ternary tree is 10 bytes. This is ~44% less memory usage compared to tree created of nodes with complete references.

The memory saving can be increased by removing unused value variable.

3.3. Ternary Tree Tests

To prove these computations and to get time requirements of this optimization the tests were performed. The tests were performed on two data structures, which are previously mentioned compressed red-black ternary tree and ordinary red-black ternary tree. Every data structure was tested with predefined set of n-grams, from 1,000,000 to 100,000,000 each. These n-grams have average length of 11 characters. Moreover the efficiency of these structures was tested on different size of n-grams. The tests were performed on computer with 84xE5-4610@2,4GHz processor with 1 Tb of RAM. N-grams were extracted from Web 1T 5-gram, 10 European Languages Version 1 collection.

To simplify implementation and not to slow down search and insertion too much the implementation for tests counts only with two types of nodes. They were performed with common internal node and with external node without left and right reference. Each of these nodes has its own type definition in code and the resolution of the node type is done by the node index. As an alternative, one bit identifier may be also used. Transformation from red-black tree into compressed red-black tree is made by post processing.

3.3.1. Search Time

On the Figure 1, there is comparison between search times of compressed red-black tree and common red-black tree. Graph shows slightly decreasing trend with average slowdown of 3% amount. This slowdown is caused by type check of node on access. This amount of slowdown seems to be acceptable in comparison of theoretical memory saving.

Note that amount of n-grams on Figure 1 do not increase linearly.

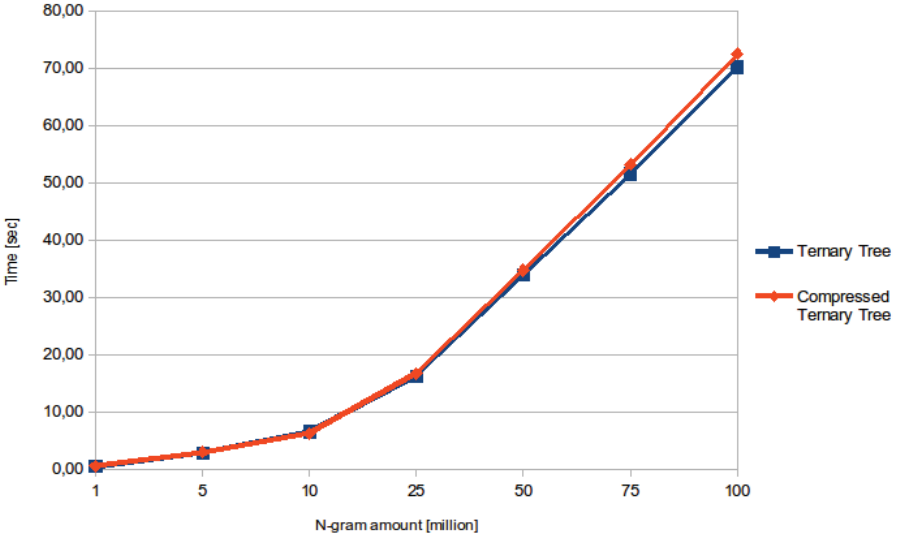

Figure 1. Search time comparison

3.3.2. Insertion Time

Comparison of insertion time is shown on Figure 2. This graph shows also slightly decreasing trend. The average slowdown is 16%. This number may be too high, but for many applications is more relevant search time or memory usage.

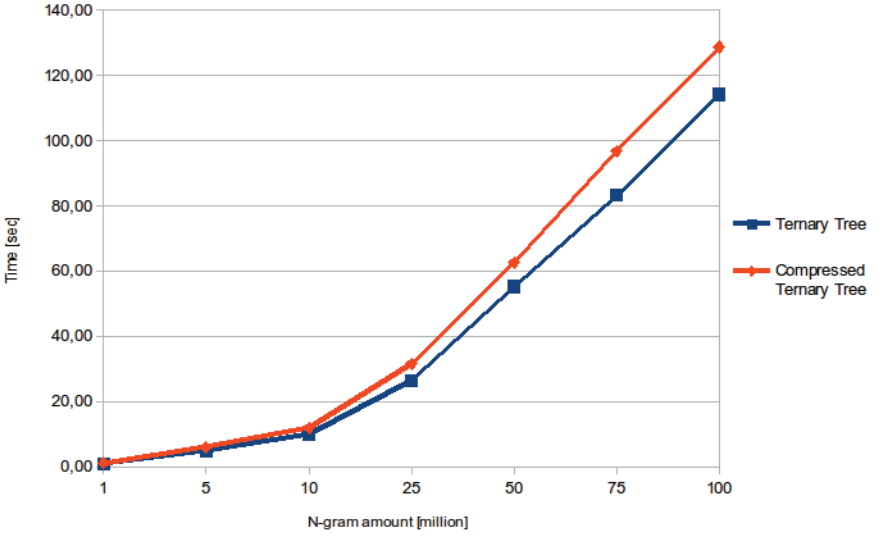

Figure 2. Insertion time comparison

3.3.3. Memory Usage

The Figure 3 shows memory usage of compared tree structures. Memory savings seems to be stable, about 35%. This amount of saved memory is high when we consider that the implementation does not remove middle reference, single left, or single right reference.

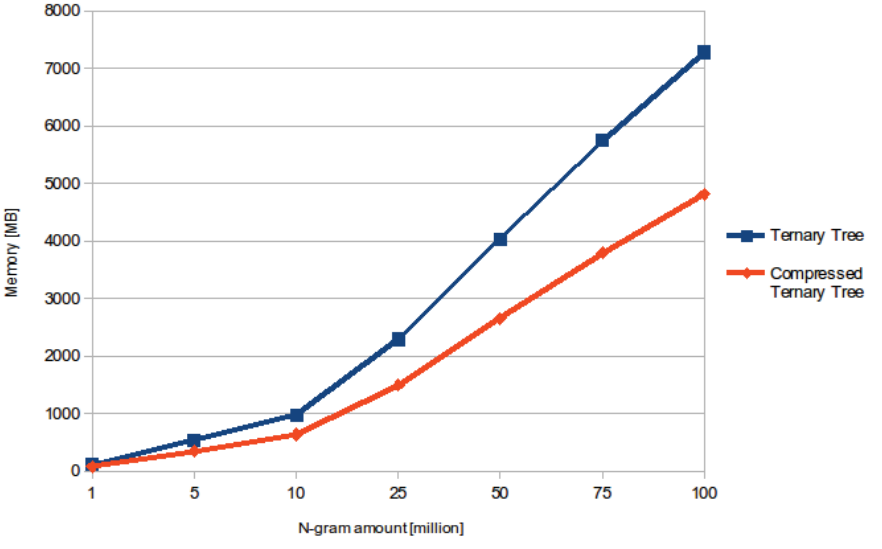

Figure 3. Memory consumption comparison

To explain this behavior is necessary to count amount of internal and external nodes of tested tree. Results show that almost 90% of all nodes are external nodes, nodes with no left or right child. The amount of nodes without middle reference is about 17%. This seems to be negligible for removal.

The amount of nodes with value reference is about 82%. Therefore removal of this type of reference can improve memory saving even more, especially when variable size is larger than 4 bytes. By modifying external node implementation to have no left reference, right reference and value variable the memory saving raises to approx. 43% without substantial slowdown.

3.4. Tests with n-gram Size

In previous chapters the behavior of compressed red-black tree depending on amount of n-grams was tested. The question is how the size of n-grams affects compressed n-gram tree performance.

Figure 4 shows search time with different size of n-grams from ~12 to ~24 characters. The amount of n-grams is 25,000,000. Results shows only small differences compared with common red-black ternary tree, with better results on greater n-gram size.

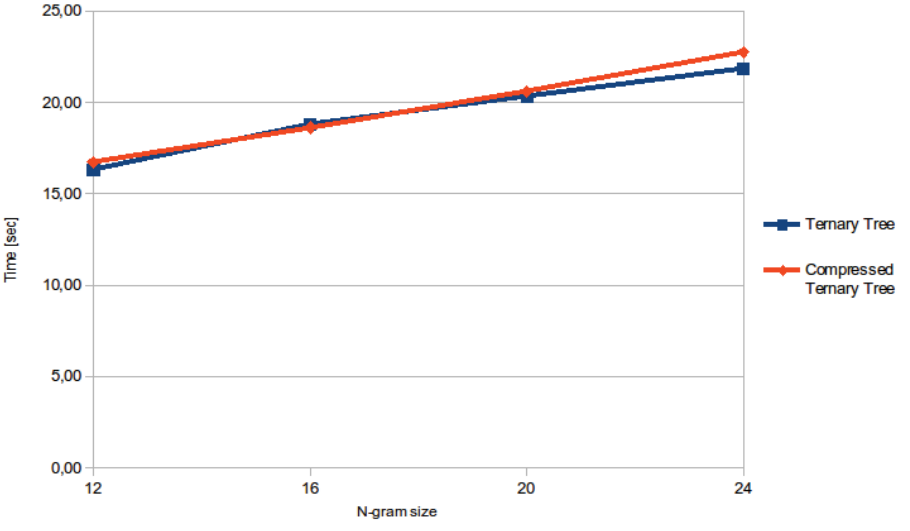

Figure 4. Search time comparison

Differences in insertion time show Figure 5. The time necessary to create and fill compressed ternary tree rises with n-gram size. This behavior may be caused by recursive algorithm used in tree compression.

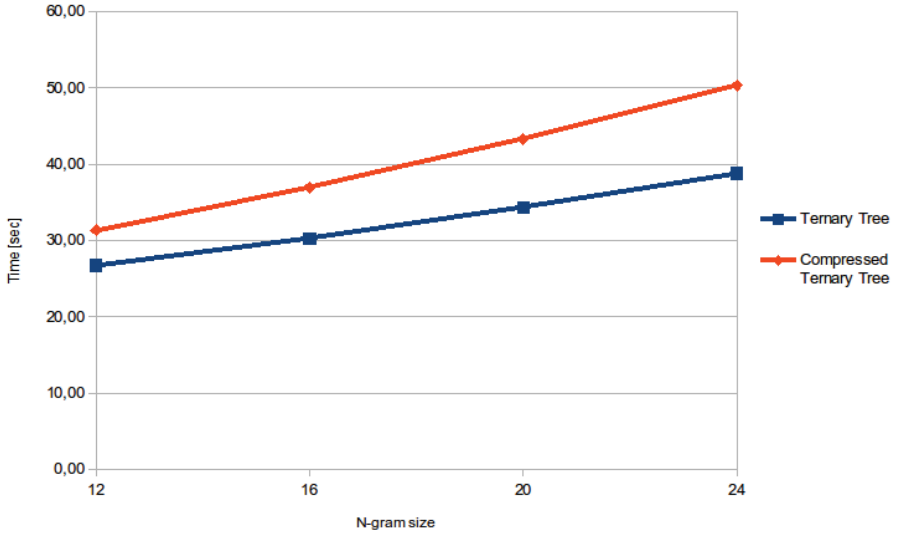

Figure 5. Insertion time comparison

Figure 6 shows memory requirements. Compressed red-black ternary tree has greater memory saving on longer n-grams. Increasing amount of single node binary trees in ternary tree causes this.

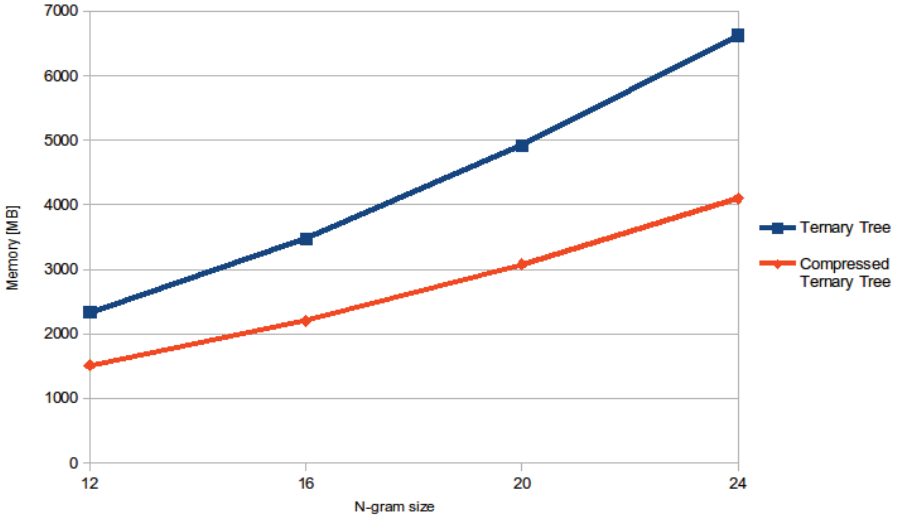

Figure 6. Memory consumption comparison

4 Ternary Forest

For sentence indexing is appropriate to use two-level (double) indexing. This approach saves a lot of computer memory, because words in all sentences are many times repeated.

Common ternary tree n-gram indexing and n-gram double indexing using ternary trees are two borderline cases. Single ternary tree is much faster in search and insertion time. This is mainly caused by low height of binary trees deeper in the ternary tree. Unfortunately this approach requires lot of computer memory.

Double indexing can save large amount of required memory, because it reduces duplicities in the tree. Disadvantage of this approach is slowdown in both search and insertion time, because deeper binary trees takes more time to be searched. But can we combine both approaches to get more balanced data structure?

One approach can be ternary forest. Ternary forest is created from two types of ternary search trees. First, *word* tree is indexing characters of words and second, *n-gram tree* is indexing whole words. Every of the second trees are connected to the last node of the first tree.

On Figure 7 is shown how three words “AB AC AB” can be indexed. In the *word tree*, node *A* is first binary tree. Second binary tree consists of nodes *B* and *C*. Second part of structure *n-gram tree* consists of two more single node binary trees, these are with keys *3* and *2*.

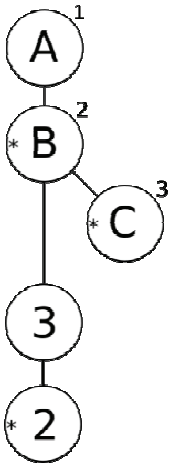

Figure 7. Ternary forest example

To search words "AB AC AB" is necessary to find first word in first part of data structure named *word tree*. When the first word is found, reference to the second part of the data structure *n-gram tree* is stored. Then the second word "AC" is found in the *word tree* with result *3*. The stored root index of *n-gram tree* is used to found node with index 3. Search is done again in the *word tree* with index *2* and the last node in the *n-gram tree* is found.

Advantage of this approach is that indexing trees are not separated, but second n-gram tree is directly connected to first *word* tree. This little difference from common double indexing may look negligibly.

The test was performed to show depth of binary trees in n-gram tree. The set of 10,000,000 5-grams was used. The results shown that over 90% of trees are single node trees. But more important is, that root tree has depth of size 32. Using ternary forest instead of double indexing can rapidly reduce this size.

Moreover, sequence amount of words in the word tree can differ. On Figure 7 word tree covers single word. But this may not be optimal for all purposes.

4.1. Ternary Forest Tests

Figure 8 shows behavior of insertion time, search time and memory requirements of data structures. First data structure is double indexing ternary search tree. Second is ternary forest, and the last one is common red-black ternary tree.

Data used for tests was 5-grams with average length of 24 characters. Ternary forest is used in four tests. In each test the ternary forest has different amount of words sequentially stored in the word tree in a row.

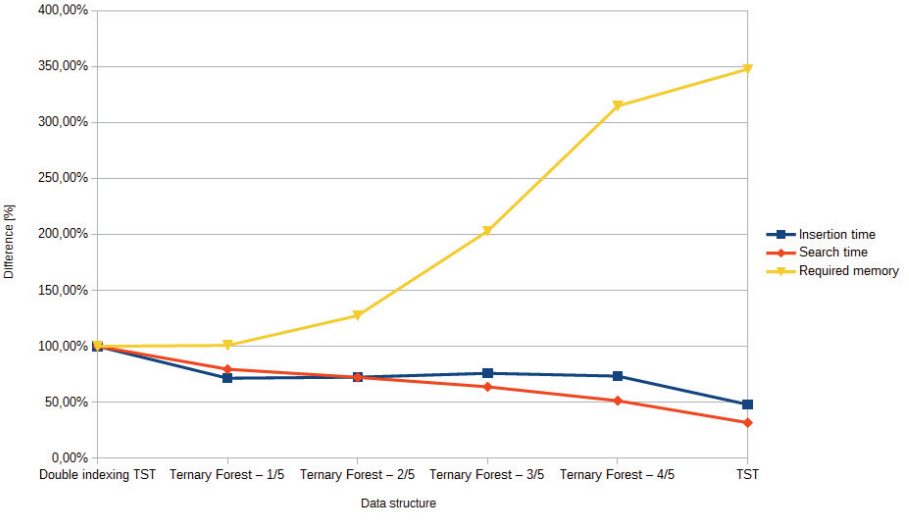

Figure 8. Relative comparison of double indexing ternary search tree (left), ternary forest and ternary search tree (right)

The results have shown that ternary forest using 1 word indexing in word tree has greatly improved performance. Insertion time speedup is ~30% and search time is ~20% faster than double indexing. The memory requirements show only negligible increase, less than 1%.

5 Conclusion

This paper described two improvements on ternary tree for efficient n-gram indexing. First, ternary tree compression showed how to save up to 43% of computer memory by removing unused references, without major slowdown.

Second improvement was more focused on n-gram indexing as such. By using ternary forest instead of common two-level indexing search time has decreased ~20% and insertion time ~30% with negligible increase of memory requirements.

Acknowledgement: This work was partially supported by the Grant of SGS No. SP2014/110, VŠB - Technical University of Ostrava, Czech Republic, and was supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070) and by the Bio-Inspired Methods: research, development and knowledge transfer project, reg. no. CZ.1.07/2.3.00/20.0073 funded by Operational Programme Education for Competitiveness, co-financed by ESF and state budget of the Czech Republic.

6 References

1. Baeza-Yates, Ricardo, and Berthier Ribeiro-Neto. *Modern information retrieval*. Vol. 463. New York: ACM press, 1999.
2. Ceylan, Hakan, and Rada Mihalcea. "An Efficient Indexer for Large N-Gram Corpora." *ACL (System Demonstrations)*. 2011.
3. Flor, Michael. "Systems and Methods for Optimizing Very Large N-Gram Collections for Speed and Memory." U.S. Patent Application 13/168,338, 2011.
4. Huston, Samuel, Alistair Moffat, and W. Bruce Croft. "Efficient indexing of repeated n-grams." *Proceedings of the fourth ACM international conference on Web search and data mining*. ACM, 2011.
5. Kim, Min-Soo, et al. "n-gram/2l: A space and time efficient two-level n-gram inverted index structure." *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005.
6. Kratky, M., et al. "Index-based n-gram extraction from large document collections." *Digital Information Management (ICDIM), 2011 Sixth International Conference on*. IEEE, 2011.
7. MOFFAT, ALISTAIR AUTOR, and Timothy C. Bell. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.
8. Pomikálek, Jan, and Pavel Rychlý. "Detecting Co-Derivative Documents in Large Text Collections." *LREC*. 2008.
9. Robenek, Daniel, Jan Platoš, and Václav Snášel. "Efficient in-memory data structures for n-grams indexing."
10. Scholer, Falk, et al. "Compression of inverted indexes for fast query evaluation."*Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2002.