FACTA UNIVERSITATIS
Series: **Automatic Control and Robotics** Vol. 15, N^{o} 3, 2016, pp. 171 - 185
DOI: 10.22190/FUACR1603171I

COMPARISON OF DATA MINING ALGORITHMS, INVERTED INDEX SEARCH AND SUFFIX TREE CLUSTERING SEARCH

UDC 681.327.12:(159.95+159.2)

Miloš Ilić[1], Dejan Rančić[2], Petar Spalević[1]

[1]University of Priština, Faculty of Technical Science, Republic of Serbia
[2]University of Niš, Faculty of Electronic Engineering, Department of Computer Science, Niš, Republic of Serbia

Abstract. *New documents are created every day and the number of digital documents in the world is exponentially growing. Search engines do a great job by making these documents easily available to the world population. Data mining works with large amount of data sets and offers data to the end user; it consists of many different techniques and algorithms. These techniques allow faster and better search for large amounts of data. Clustering is one of the techniques used in a data mining process; it is based on data grouping according to the features, or any property they have in common, thus, a search process is faster, and a user gets better search results. On the other hand, an inverted index is a structure that provides fast search too, but this structure does not create clusters or groups of similar data. Instead, it processes all data in a document and measures appearance of specific terms in a document. The goal of this paper is to compare these two algorithms. The authors created applications that use these two algorithms and tested them on the same corpus of documents. For both algorithms, the authors are presenting improvements that provide faster search and better search results.*

Key words: *application; clustering; data mining; inverted index; Lucene; suffix tree*

1. INTRODUCTION

Information and data around us carry a large amount of knowledge. People use different techniques to process information in everyday life. To be able to discover and extract knowledge from data is a task that many researchers and practitioners are endeavouring to accomplish. There is a lot of hidden knowledge waiting to be discovered. The process of knowledge discovering is a challenge created by today's abundance of

Received May 09, 2016
Corresponding author: Miloš Ilić
Faculty of Technical Science, Kneza Miloša 7, 38220 Kosovska Mitrovica, Republic of Serbia
E-mail: milos.ilic@pr.ac.rs

data. Developments in the field of data have led to today's sophisticated and powerful database systems. Huge databases are rich with data, but on the other hand, they are poor with information hidden in the stored data. Knowledge discovery in databases is a process of identifying some valid, novel, useful and understandable patterns from large datasets [1]. That is an automatic, exploratory analysis and modelling of large data repositories. Data mining is core in knowledge discovery, which involves inferring algorithms that explore data, developing a model and discovering previously unknown patterns. Data mining for knowledge discovery is a whole process of applying computer-based methodology, including new techniques. The model is used for understanding data, analysis and prediction phenomena.

The need to understand large, complex, information-rich data sets is common to all fields of business, science and engineering. Data mining is an iterative process within which the progress is defined by discovery, either through automatic or manual methods. Data mining is a search for new, valuable, and nontrivial information in large volumes of data [2]. It is a cooperative effort of humans and computers. The best results are achieved by balancing the human experts' knowledge of describing problems and goals with the search capabilities of computers. Data mining activities can be put into one of the two categories: the first category is predictive data mining, which produces a model of a system described by the given data sets; the second category is descriptive data mining, which produces some new nontrivial information, based on the available data set. On a predictive end of the spectrum, the goal of data mining is to produce a model, expressed as an executable code, which can be used to perform classification, prediction, estimation, or other similar tasks. On a descriptive end of the spectrum, the goal is to gain understanding of an analyser system by uncovering patterns and relationships in large data sets.

In business, main knowledge discovery and data mining application areas include marketing, finance, fraud detection, manufacturing, telecommunications, and Internet agents [3]. Inverted index is a structure that can be used in a data mining process. It is a sorted list of words, with the list of corresponding documents attached to each word. Another technique that is used in data mining is clustering, which is based on data grouping according to the character, or to any property they have in common. In so doing, knowledge search within data is much faster. The authors described and implemented both algorithms, in order to test the execution time, i.e. the time needed for search and search results.

The assessment of data mining algorithms is a specific job which can be performed based on multiple criteria. Namely, beside the memory and CPU occupancy and execution time, many other criteria can be observed, some of them being entropy, f-measure and recall, and they can be used especially when two or more algorithms need to be compared [4]. In this paper, we compared different data mining algorithms, choosing thus the best criteria for our task.

2. INVERTED INDEX STRUCTURE

In order to search with in large amounts of data processed by data mining techniques, one can use information retrieval. It is a finding material of an unstructured nature that satisfies the information need from within large collections [5]. In this form, finding materials are usually text documents, typically stored on computers. For this task, the

simplest and the best method is the inverted index search, since it reduces the number of documents that need to be processed, by identifying the ones that contain a search term. Inverted index is a data structure that maps each word in a dataset to a list of IDs of the documents, in which the word appears to retrieve them efficiently. Inverted index for a document collection consists of a set of the so-called inverted list, known as postings lists. Each inverted list corresponds to a word which stores all the IDs of documents where this word appears in ascending order [6].

The second component of an inverted index is a dictionary. It serves as a lookup data structure on top of the postings lists. For every query term in an incoming search query, a search engine first needs to locate the postings list of the term, before it can start processing the query. It is a job of a dictionary to provide the mapping from terms to the location of their postings lists in the index. A dictionary is usually stored in an operative memory, because an access to data in a memory is much faster than an access to a data disk. It is a practice to keep in memory as much data as possible, especially the frequently accessed ones [7]. The technique of keeping frequently used disk data in the main memory is named caching. Dictionary terms are usually arranged alphabetically. Inverted index algorithm contains two phases - the first one is index construction. In this phase, a text collection is processed sequentially, one token at a time, and postings list is built in an incremental fashion for each term in the collection. The output of the first phase is postings lists, for example, an inverted index structure. Information stored in the index is used in the second phase, a query processing. This phase processes the search query and output search result in form of the documents containing the query term.

Inverted index structure can be used for text and web documents. When used for web documents, a web crawling must be applied. With simple cases, in a postings list, one only needs to have a document ID, since the existence of a posting itself indicates the presence of a term in a document. A complex case includes positions of every occurrence of a term in a document, properties of that term, or even the results of an additional linguistic processing. In cases like this, based on term proximity, phrase queries and document scoring are provided [8]. The next example shows how to create and use an inverted index for some corpus of text documents. Suppose we have the following documents:

d1: Clustering is one of the data mining techniques,

d2: The inverted index technique can be used for searching within documents,

d3: Inverted index and clustering are the data mining techniques

What is known from the start is that all words, strings and characters in a document represent descriptors and set of terms, which is very important for postings lists and the dictionary set of terms. Many documents in corpus may have the same descriptors and set of terms.

It is essential to extract a unique set of terms from all documents - repetition of terms should not exist in the extracted set of terms, which is used to create an inverted index structure. A set of terms for our example is presented below.

t1: clustering; **t2:** data mining; **t3:** techniques; **t4:** inverted index; **t5:** searching; **t6:** documents.

As stated above, inverted index structure has two components: dictionary and posting lists, and our example shows a basic inverted index structure (see Fig. 1). Some terms are presented in one document, some in two, and some of them are presented in all documents

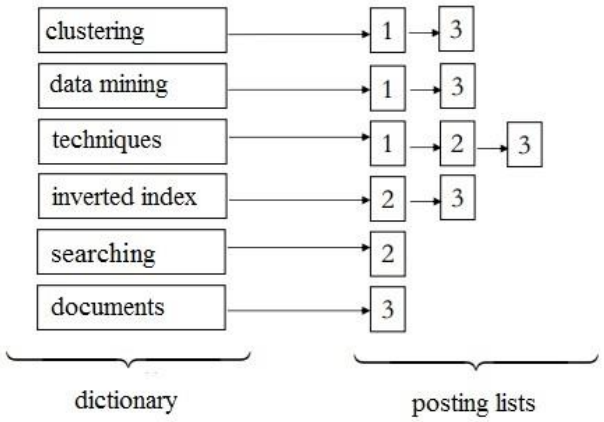

Fig. 1 A basic inverted index structure for the previous example

This form of an inverted index is good and provides fast query processing. Thus, searching for query in a big set of documents is reduced to smaller document set that contains a unique query term. If a search query is not one keyword, but a string of words or phrases, terms in query can be concatenated with logical operators. Therefore, a union term in the dictionary must be found, and a union of document IDs from corresponding posting lists will be returned. In a big set of documents, every possible speed up is preferable. To accelerate the processing of queries, one more field is usually added to dictionary to remember the number of documents in which the term appears. This method is named an extended inverted index. Another method is to create posting lists with skip pointers. If there are two regular posting lists and we walk through them simultaneously, they are linear in the total number of postings entries. If the list length is m and n, the intersection takes O(m + n) operations. A postings list intersection can be processed in sub-linear time if the index is not changing too fast. One way to do this is to use a skip list by augmenting postings lists with skip pointers (at indexing time). An example for posting lists with skip pointers is shown below (see Fig. 2). Skip pointers are effective shortcuts that allow avoiding processing parts of the postings list which will not figure in search results.

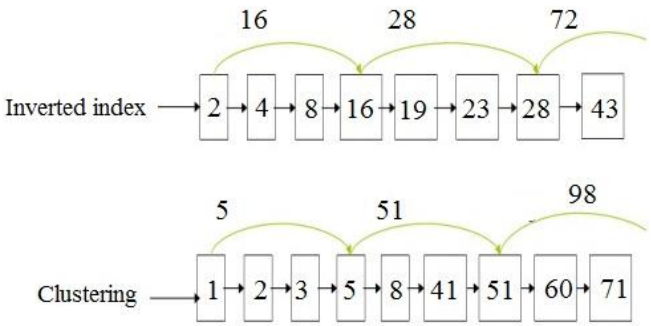

Fig. 2 Postings lists with skip pointers

In the above-presented example, the terms **t1**: clustering and **t4**: inverted index are in many more documents than in the previous example. In both cases, the next step in a process of an inverted index creation is memory representation. Postings lists are usually stored on the hard disk and the process of disk reading or writing is slow, as it takes time for the disk head to move to the part of the disk where the data are located. In order to present the dictionary of index terms, the best data structure must be chosen, hence, the most commonly used structures are the following: arranged arrays, binary trees, B-trees and hash tables. Each has some characteristics in the use of inverted index representation; yet, the best data structure for representation of an inverted index is a B-tree. B-tree is a balanced tree, which means that the number of levels on each path from the root to the leaf is the same. Search time is considerably less, compared to the search time in binary tree with the same number of terms. Such a representation provides fast search, small memory consumption and little communication between the massive memory and main memory. B-tree for the inverted index in Fig. 1 is presented on Fig. 3.

Another question about the inverted index with skip lists is how to place skip pointers. There is a tradeoff- more skips lead to shorter skip spans, and we are more likely to skip. But it also means lots of comparisons of the skip pointers, and lots of space for storing them. Fewer skips means fewer pointer comparisons causing long skip spans, which means that there will be fewer opportunities to skip. A simple heuristic method for placing skips, which seems to work well in practice, is to use the $\sqrt{R}$ evenly-spaced skip pointers for a postings list of the length P. Building effective skip pointers is easy if the index is relatively static. If not, and if a postings list keeps changing due to some updates, it is harder to create skip pointers. Skip pointers are helpful only with AND queries, not with OR queries.

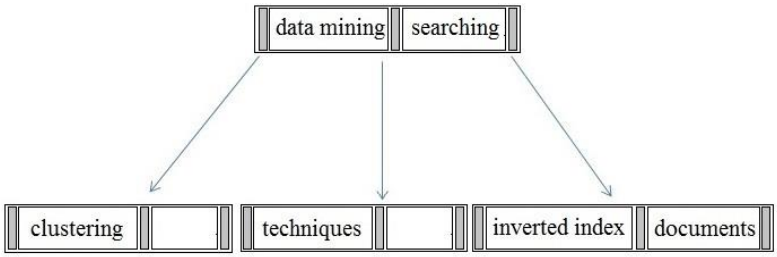

Fig. 3 Inverted index presented by B-tree

In our example, we utilize a B-tree with three edges and two keys per node. In picture above (see Fig. 3), grey rectangles stand for edges, while white rectangles represent keys. We choose the nodes with three edges and two keys, since there is a small number of terms. The number of keys and edges in a node determine the processing speed for a B-tree. The fastest search is in B-trees with five edges and four keys. With the B-tree representation of the inverted index, the process of creation is finished. At this point, searching for specific term in the documents can start. In case of large document collections, the resulting number of matching documents may far exceed the number which a human user could possibly sift through. To prevent this, a search engine rank-orders the documents that match a query. For each matching document, a search engine computes the score with respect to the query at hand. Scoring can be done in several ways, parametric index being

one of them. With this method, a query processing consists of postings intersections, except that we may merge postings from standard inverted and parametric indexes. There is one parametric index for each field, for example, the date of creation, and it allows selecting only the documents that match the date specified in the query- that way the number of documents can be significantly reduced.

Given these scores, the final step before presenting results to a user is to pick out the K highest-scoring documents. While one could sort a complete set of scores, a better approach would be to use heap to retrieve only the top K documents in order. For representation of a search result, we propose using a tree structure. This could be a very good practice when there are a lot of documents that contain a searching term. The best tree for representation is a B-tree. Keys in the nodes represent file paths. The tree must be balanced, and criterion for creation is a score for a search term. This kind of representation can provide a fast traverse and extraction of the branch with the best documents.

2.1. The application created for inverted index algorithm

In this part of paper, we are describing our application that uses an inverted index structure to search some query in a set of text documents. We have created, an application in Java environment (NetBeans IDE 8.0), and used Lucene, an open source library. One of the key factors behind Lucene's popularity and success is its simplicity [9]. Lucene is a high - performance scalable library. It is a mature, free, open-source project, implemented in Java. Indexing with Lucene looks like a deceptively simple and monolithic operation. Complex set of operations inside simple API can be broken down into three major and functionally distinct groups: conversion to text, analysis and index writing.We have used functions and methods from two versions of Lucene library (4.0.0 and 2.3.0), because some of the functions in one version are not presented in another. These functions process the documents in a way that is most applicable to our needs. A class diagram of application is presented below (see Fig. 4).

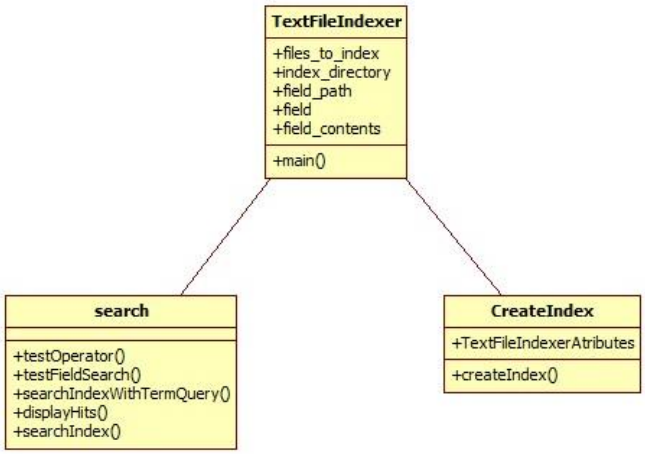

Fig. 4 A class diagram for inverted index application

There are three classes with appropriate attributes and methods. The existence of only three classes proves Lucene's simplicity. The class <CreateIndex> uses attributes defined in the class <TextFileIndexer>, and creates an inverted index structure. In this class, we used the library methods for an easier inverted index creation. The one created in the end of this process is stored in the file whose path is defined in an <index_directory> attribute. We have used a created inverted index in a class <search>. At this point, we have to start searching for some query term and present results in the form of path and score [10]. Methods of calculating the score sort an output in a descending order, as well. Thus, at the top of the document list, there is a document with the highest score. If we do not change the documents, the inverted index will not be calculated again in next executions.

3. SUFFIX TREE CLUSTERING ALGORITHM

Clustering is a widely adopted data mining model that partitions data points into a set of groups, each of which is called a cluster. A data point has a shorter distance to points within the cluster than to those outside the cluster. The fundamental assumption we make when using clustering in information retrieval is a cluster hypothesis, reading: "documents in the same cluster behave similarly with respect to relevance to information needs".

The hypothesis states that if there is a document from a cluster that is relevant to a search request, then it is likely that other documents from the same cluster are also relevant. This is because clustering puts together documents that share many terms.

Suffix tree clustering is one of the clustering algorithms. A suffix tree clustering algorithm represents a hierarchical group of algorithms [11]. A hierarchical clustering outputs the hierarchy, a structure that is more informative than the unstructured set of clusters returned by a flat clustering. This is an incremental clustering algorithm which is based on the identifying phrases that are common for the groups of documents. A phrase is an ordered sequence of one or more words. This algorithm does not treat documents as a collection of words, but as a string of words. Thus, it operates by using the proximity information between words.

A suffix tree clustering algorithm consists of four steps, and the first step is document cleaning. In this step, a string of text representing the content of each document is transformed by using a light stemming algorithm. Stemming algorithm is a process of a linguistic normalization, in which variant forms of a word are reduced to a common form. It is stripping the HTML tags, separator and common stop words, as well as extracting word stems (deleting word prefixes and suffixes and reducing plural to singular). An original string representing the document is saved, and pointers to the beginning of each word in the string transformed to the position are taken in the original string [12]. A simple form of some stemming algorithms is their advantage, and for iterative algorithms, the rule that applies is that iteration takes care of both deletion and replacement; on the other hand, a disadvantage is that some of iterative algorithms are very heavy, leading to over- stemming.

In the second step, a suffix tree clustering algorithm uses the plain text created in the first step. This step denotes a construction of a generalized suffix tree structure. The first phase in this step is a construction of suffix tree structure. The suffix tree is a data structure which contains all the suffixes of a given string. The string may be a string of

characters or a string of words. A suffix tree is a rooted, directed tree in which every internal node has at least two children nodes. Every edge in the tree is labelled with a non-empty sub-string of S. The label of a node is defined to be the concatenation of the edge-labels on the path from the root to that node [13]. There are no two edges out of the same node with edge-labels that begin with the same word. This feature makes the tree compact or generalized, and this is the second phase in second step of suffix tree clustering algorithm. The best algorithm for a compact suffix tree clustering is Ukkonen's algorithm. His method also builds the tree by the most compact and technical representation, as previously described. An example of a generalized suffix tree is presented in Fig. 5. This is an example for the following strings: "STC is a clustering algorithm", "K-means is a clustering algorithm, too" and "STC and K-means are clustering algorithms."

Circles in the tree represent nodes, numbers in the squares represent document groups - the first number denotes a document, while the second stands for a word position in a string of words. Each node of a suffix tree represents a group of documents and a phrase common to all of them. At this point, we have to identify based clusters and calculate the score for each cluster. Each node in a generalized suffix tree represents a base cluster.

The score $S(B)$ for each base cluster is a function of the number of the documents it contains, and the words that make up its phrase. The score $S(B)$ for the base cluster B and a phrase P is defined as in (1).

$$S(B) = |B| f(|P|) \tag{1}$$

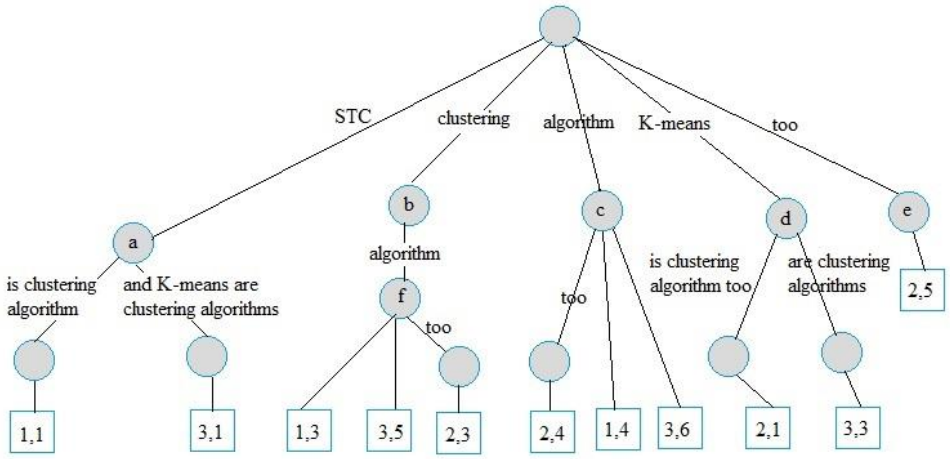

Fig. 5 An example of a generalized suffix tree for the given strings

In above presented equation, $|B|$indicates the number of documents in a base cluster and $|P|$is the number of words. There are six base clusters in our example, and each of them will be included in the next step of the suffix tree clustering, named after combining base clusters into a cluster. Why is the next step so important? There are base clusters in the present form, which is good, but the documents may be sharing more than one phrase. To avoid document overlapping and a nearly identical cluster, this step is assigned to

merge base clusters with the high overlap in a document set. Binary similarity is the best measure for this problem. It calculates whether a base cluster should be merged or not. If conditions in formulas (2) and (3) are met, binary similarity have value one, otherwise it shall be defined as zero.

$$|B_m \cap B_n| / |B_m| > 0.5 \tag{2}$$

$$|B_m \cap B_n| / |B_n| > 0.5 \tag{3}$$

In these two formulas, B_m and B_n are two base clusters, and $|B_m|$ and $|B_n|$ are their sizes, respectively. An intersection of their sizes $|B_m \cap B_n|$ represents the number of documents that are common for both clusters. At this point, a cluster graph can be created. The node in the graph denotes basic clusters and the edges connect two nodes in the graph with representing clusters, with one similarity. A cluster is defined as a connected graph, made up of main clusters. Each cluster consists of a union of documents of all its base clusters. The final clusters are sorted according to the number of hits and mutual overlap of their basic clusters. If a new document is added in the suffix tree, the algorithm must update the base clusters and recalculate similarities between these and other base clusters. Each node that is restored or created after adding a new document in the suffix tree is denoted.

In the end, an algorithm checks to see if the changes in the graph of base clusters may cause changes in final clusters. Another advantage of the STC algorithm is that it does not require from a user to specify the number of clusters. On the other hand, this algorithm requires determining specifications, and based on that specification, it will define the threshold of similarity between the base clusters.

3.1. The Application Created for Suffix Tree Clustering Algorithm

Suffix tree clustering algorithm is a good algorithm for text and web page documents. The execution time for both sets of documents is approximately the same. Readers may find more about execution results for web and text documents in [14]. The application created for comparison and presented in this paper is designed in Visual Studio environment.

The authors have created a form application with the appropriate classes that provides suffix tree implementation and tests for different set of text documents. The form provides an interaction with a user and displays results. From the form, a user can load specific directory containing all documents for clustering, or they can load a single file and cluster that document. When the suffix tree clustering is finished, a user can start searching for a search query. The application uses the created structure and searches for documents containing a searched query; the execution result is displayed in the form of a file path and the number of hits for a specific query in a document. The application class diagram is presented in Fig. 6. - there are four classes for a suffix tree and one representing the user form. All classes have appropriate attributes and methods. The class diagram below shows how classes are connected in the application.

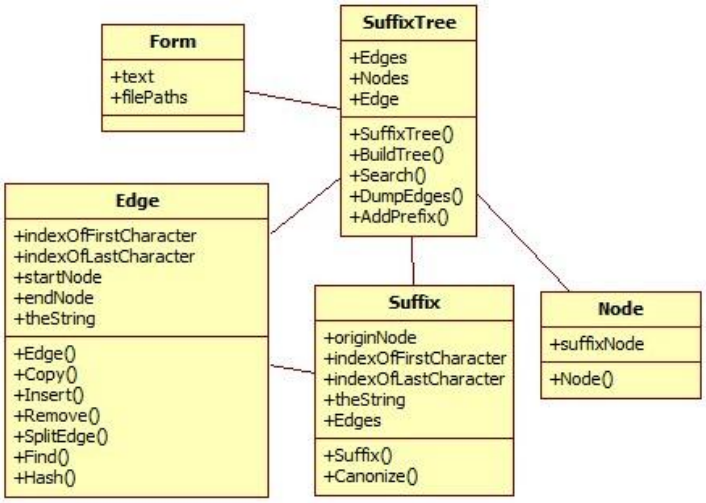

Fig. 6 A class diagram for a Suffix tree clustering application

The class <SuffixTree> has the method applied in a search process. The class <Node> implements a node in the suffix tree, and class <Edge> provides attributes and structures that keep information of edges. As stated above, the edge keeps strings representing suffixes, and concatenation of edge labels from the root node to the leaf node build a string of words. A class representing a form does not have any specific methods or attributes. This class implements the listeners for two buttons- the first button loads any individual document or group of documents and the second one starts a search for some query. The form class calls methods from the <SuffixTree> class, as presented in Fig. 6. The method <BuildTree()> creates a suffix tree structure and clusters for specific or selected documents. A created suffix tree clustering structure, ready for future search, is the result of this method. The Method <Search()> in the <SuffixTree> class takes the string as an argument and starts search for that string in a created structure.

If the search is successful, this method returns all documents in which a searched string was found. Moreover, this method returns the number of hits for every specific document containing the searched query. In so doing, the number of documents that needs to be processed and read is reduced. A user gets information which documents have the searched terms, knowing thus which documents are the best for reading. If the number of returned file paths is large, we can create a B-tree structure, based on a file path and the number of hits in documents. From this structure, we can extract the best branch and present it to a user.

4. THE COMPARATIVE ANALYSIS OF BOTH APPLICATIONS

The application and data structures presented above are different in many ways. The most striking difference is that the first application is implemented in java environment, while the second one is implemented in C#.Net environment. That may cause alteration in execution time, memory and CPU consumption. The second obvious difference is that the structures used in applications do not have similarities- each structure processes data in its

specific way. As stated before, the first application uses inverted index structure and B-tree for the memory representation of a created inverted index, the second one uses the suffix tree structure and creates clusters of documents. Both structures have specific memory representation and search processes.

In data mining, we use techniques with the purpose of processing large amount of data and returning the most relevant documents. In this process, the basic criterion for the quality evaluation of data mining algorithms is the number of returned true/ false documents. Returned true documents are the documents that really contain search terms. False documents are the documents returned as the ones containing a search term, but the term is not true. Like other algorithms, data mining algorithms can be compared based on complexity, execution speed and memory space occupation.

An important consideration for every algorithm and application execution is the configuration of a computer on which the application is executed. Depending on the CPU speed, amount of free main memory and the operational system, execution can be faster or slower. Both our applications were executed on the same computer. Table 1 shows the characteristics of the computer components on which applications were executed.

Table 1 Characteristics of computer components

Component name	Characteristic
Processor	AMD Athlon™ 64 X2 Dual-Core Processor TK-55 1.80 GHZ
Installed memory (RAM)	3GB
System type	64-bit Operating System
Windows edition	Windows 7 Professional

Operation system and basic system processes take about 10-20 % of CPU usage, and about 1.06 GB of memory. While the application is executed, all other programs or applications are turned off, only an appropriate development program is launched. In case Visual Studio is launched, CPU usage increases for about 5%, and memory occupancy is 1.25GB.When the NetBeans is the only launched program, CPU usage is about 14-29%, and memory occupancy is approximately 1.50GB. These are starting parameters, and execution tests will only increase these numbers. For both applications, we will be testing execution time in two rounds. In the first round, we have to measure execution time needed for creation of an implemented structure for the same set of documents, while in the second round, we measure execution time for the search process for some query term or query string. At the same time, we measure CPU usage and memory occupancy in both rounds. We shall start our tests with small set of documents, and in the following rounds we shall increase the size and number of documents in the directory. Execution is repeated ten times for each set of documents, in each round and for each application. Average execution time, CPU usage and memory occupancy are calculated and presented in figures below. We have decided to test applications on ten sets of documents. Each set is of different size-the first is 100kb and contains two documents. Size ranges from 100kb to 125mb, and the number of documents in the sets is different, as well, ranging from 2 to 23. Documents are of random size, and have random plain text and because of the plain text, they contain a large number of words.

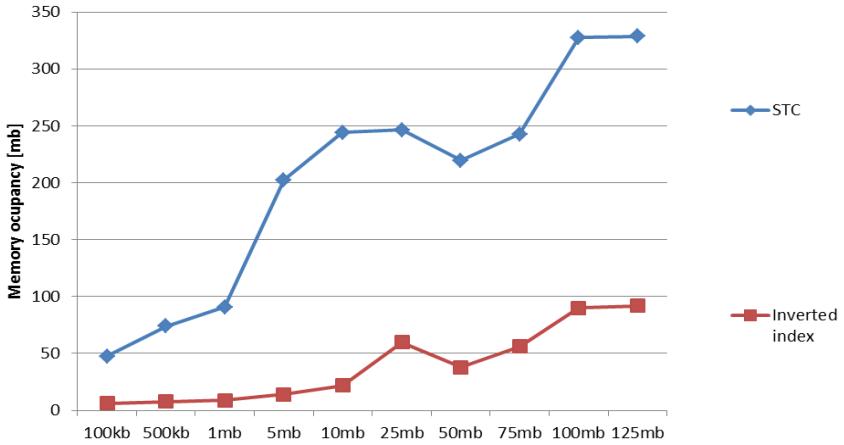

Fig. 7 Comparative view of memory consumption for STC and inverted index creation

Fig. 7 shows that inverted index structure takes up less memory than the suffix tree clustering structure. A Java virtual machine, a core of java environment, reserves approximately 0.70GB of the main memory for java processes when the execution starts. As presented above, the space inverted index uses just a small part of it. In both cases, when the size of a document set increases, the used memory increases, as well. CPU usage for both structures is presented in Fig. 8.

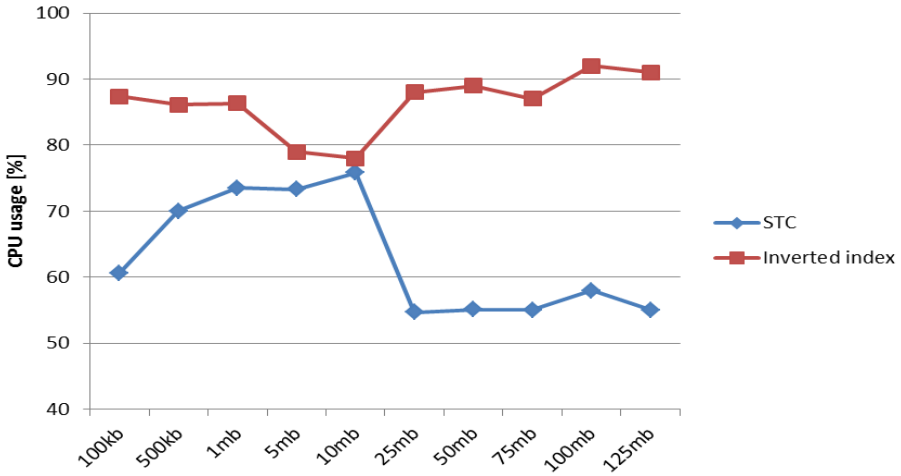

Fig. 8 Comparative view of CPU usage for STC and inverted index creation

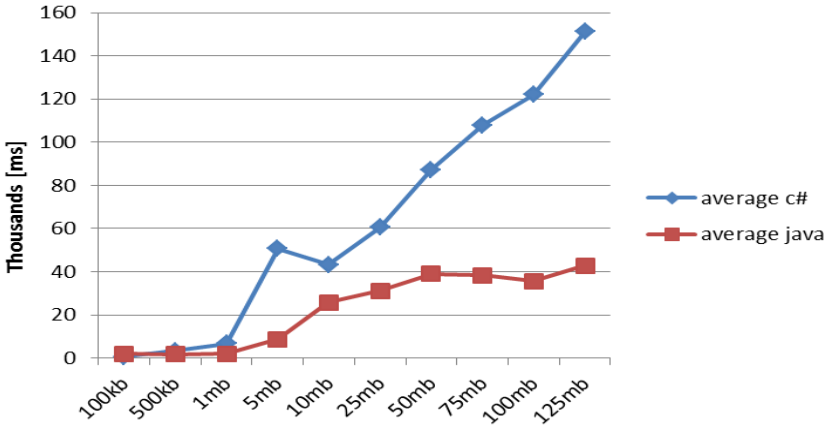

Fig. 9 Comparative view of execution time for STC and inverted index creation

The processes in the application implementing inverted index structure uses more CPU time. This application often uses approximately all available CPU time; otherwise, the STC structure uses less CPU time for the same document sets. Execution time for both applications is presented in Fig. 9. A bigger document demands higher execution time in both cases. The first execution time for an inverted index is higher than the other in the same document set.

This is because an inverted index is not calculated again unless documents are not changed, and application just checks for changes in a document set. As previously mentioned, in the second round the same parameters are measured, but this time for the sake of a searching process. The comparison of average memory consumption, CPU usage and execution time is presented in Fig. 10, 11 and 12, respectively.

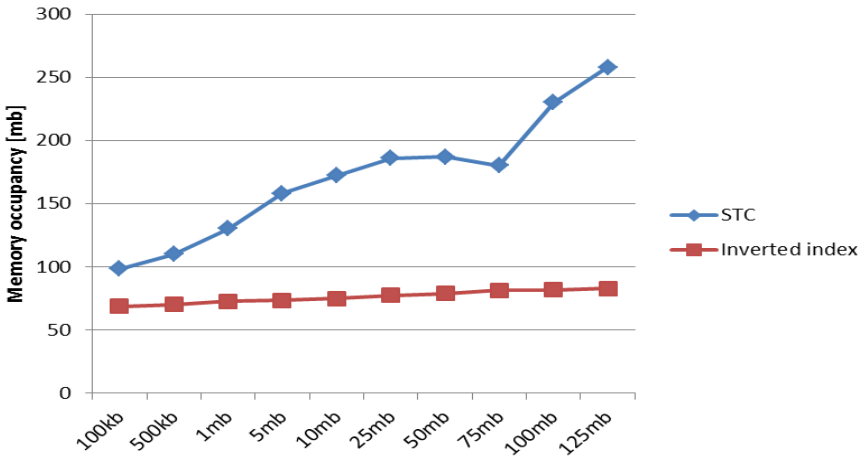

Fig. 10 Comparative view of memory consumption in a searching process

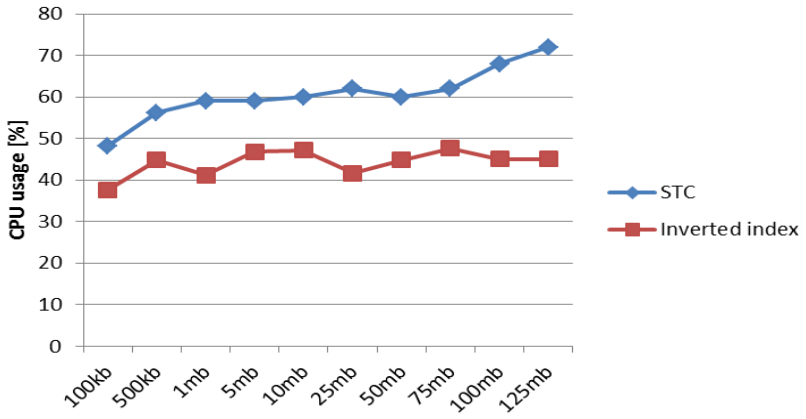

Fig. 11 Comparative view of CPU usage in a searching process

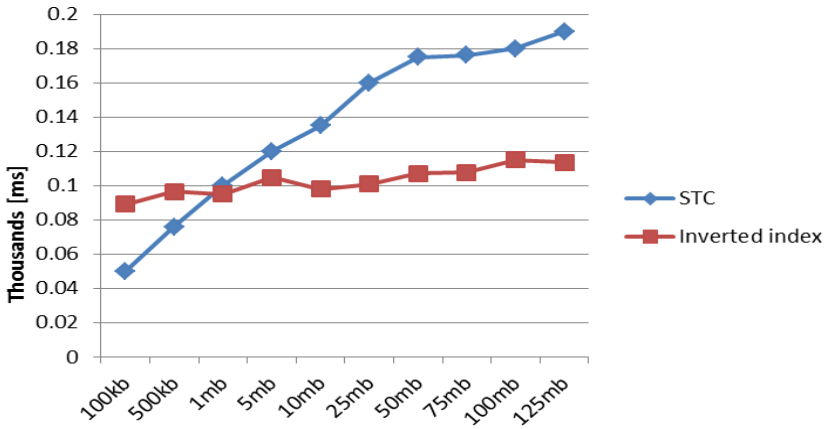

Fig. 12 Comparative view of execution time for a searching process

Experimental results presented above describe advantages and disadvantages of both data mining structures. In one part of implementation, one structure has better performance; in another part, another structure is better. Depending on the size of a document, in the same parts, the execution results of both applications are approximately the same. When the size of a document set increases or decreases, comparative execution results are different. An obvious fact would be that both inverted index and suffix tree clustering are good techniques for data processing.

5. CONCLUSION

The key idea for this paper is to describe these two techniques and compare performances for both techniques on the same document set and under similar conditions. These two applications may seem very different at first sight, and yet, these applications

do the same work, in their own way. The above-described performances are key performances when we want to compare some algorithms. Execution results have demonstrated that the inverted index and suffix tree clustering techniques could be used in the process of searching for large amount of data. Both applications reduce number of documents and provide faster search. A user can utilize these applications to search for some specific documents in a large set, based on a search query, or a key word.

We did four hundred executions in our research and provided sufficient parameters to assess the quality- what is certain is that both applications are of good quality. Our plan for the future is to combine some big database with these techniques, using them in the best possible way.

REFERENCES

[1] O. Maimon, L. Rokach, *Data Mining and Knowledge Discovery Handbook*, Springer, New York, USA, 2010, pp. 1-208.

[2] M. Kantardzi, *Data mining concepts models methods and algorithms*, John Wiley & Sons, Inc., Hoboken, New Jersey, 2011, pp. 5-25.

[3] L. Xu, C. Jiang, J. Wang, J. Yuan, Y. Ren, Information Security in Big Data: Privacy and Data Mining, IEEE Access, Vol. 2, 2014, pp. 1149-1176.

[4] M. Rafi, M. Maujood, M. Fazal, S. Muhammad, A comparison of two suffix tree-based document clustering algorithms, Information and Emerging Technologies (ICIET), Karachi, 2010, pp.1-5.

[5] C. Manning, P. Raghavan, H. Schütze, *An introduction to information retrieval*, Cambridge University Press, Cambridge, England, 2009, 67-149.

[6] H. Wu, G. Li, L. Zhou, Genix: Generalized Inverted Index for Keyword Search, Tsinghua Science and Technology, Vol. 18, Num. 1, 2013,pp.77-87

[7] A. Jain, A. Bajpai, M. Rohila, Efficient Clustering Technique for Information Retrieval in Data Mining, International Journal of Emerging Technology and Advanced Engineering, Vol. 2, Issue 6, 2012, pp.12-20.

[8] R. Konow, G. Navarro, C. Clarke, A. Ortiz, Faster and smaller inverted indices with treaps, Proceedings of the 36th International ACM SIGIR conference Research and Development in Information Retrieval, 2013, pp. 193-202.

[9] M. McCandless, E. Hatcher, O. Gospodnetic, *Lucene in Action*, Manning Publication Co., 180 Broad Suite 1323, Stamford, USA, 2010, pp. 233-345.

[10] M. Ilic, P. Spalevic, M. Veinovic, Inverted Index Search in Data Mining, Proceedings of the 22nd Telecommunications forum - TELFOR, Belgrade, pp. 943-946.

[11] M. Shindler, Clustering for Information Analysis and Retrieval: Algorithms and Applications, PhD. Dissertation, Department of Computer Sciences, University of California, Los Angeles, 2011, pp.1-158.

[12] D. Sharma, Stemming Algorithms: A Comparative Study and their Analysis, International Journal of applied Information systems, Foundation of Computer Science FCS, New York, USA, Vol. 4, Num. 3, 2012, pp.7-12

[13] M. Galaen, Document klynging (documents clustering), Master of Science in Informatics, Norwegian University of Science and Technology, 2008, pp. 19-42.

[14] M. Ilic, P. Spalevic, M. Veinovic, Suffix Tree Clustering – Data mining algorithm, Proceedings of the Twenty-Third International Electrotechnical and Computer Science Conference ERK, Vol. B, Portoroz, Slovenia, 2014, pp. 15-18.