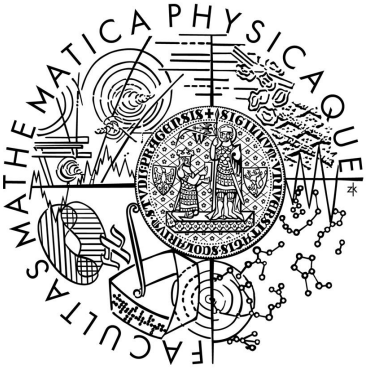
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Radoslav Zápotocký

Shlukování textových dokumentů a jejich částí

Clustering of text documents and their parts

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Michal Kopecký, Ph.D.

Studijní program: informatika
Studijní obor: softwarové systémy

Praha 2011

Velké poděkování patří hlavně vedoucímu diplomové práce RNDr. Michalovi Kopeckému, Ph.D. za nápomocnou ruku a přínosné konzultace při psaní diplomové práce.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Souhlasím se zapůjčováním práce.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 3. 8. 2011 *Radoslav Zápotocký*

Název práce: Shlukování textových dokumentů a jejich částí

Autor: Radoslav Zápotocký

Katedra (ústav): Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Michal Kopecký, Ph.D.

Abstrakt: Práce analyzuje možnosti použití vektorového modelu a shlukování aplikované na jednotlivé části dokumentu – kapitoly, odstavce a věty – z hlediska možnosti usnadnění navigace v dokumentu mezi podobnými částmi. Součásti práce je rovněž simulační aplikace (*SimDIS*), napsaná v jazyce C#, která model implementuje a nabízí nástroje pro vizualizaci vektorů a shluků.

Klíčová slova: vektorový model, shlukování, zpracování textu, C#

Title: Clustering of text documents and their parts

Author: Radoslav Zápotocký

Department: Department of Software Engineering

Supervisor: RNDr. Michal Kopecký, Ph.D.

Abstract: This thesis analyses use of vector-space model and data clustering approaches on parts of single document – on chapters, paragraphs and sentences – to allow simple navigation between similar parts. A simulation application (*SimDIS*), written in C# programming language is also part of this thesis. The application implements the described model and provides tools for visualization of vectors and clusters.

Keywords: vector-space model, clustering, text processing, C#

Table of contents

1. Introduction

The number of documents in electronic form is growing rapidly with increased number of people having access to the computer and Internet. With massive number of documents it is impossible for users to read them all to find the information they wants. Therefore, a number of techniques were invented to search for documents and new approaches are still under research. Most of these approaches are used mainly on collections of documents and take each document as one entity.

Some of problems appearing on document collections could be seen also on document level. Many documents like technical manuals or lexical dictionaries etc are often very long, not entirely well structured and with limited search or navigation ability. Those problems could be even more visible in case the documents are to be read on electronic book readers. Reading the document at whole could be time consuming and in case the document does not contain the relevant information it could be helpful to know an approximate summary of the document without any need to read it completely.

This work focuses on possibilities of applying text retrieval techniques on individual parts of a single document for addressing above mentioned issues. We will use vector-space model to look at the single document as a hierarchy of vectors, and provide tool for document content analysis.

The goal of this thesis is to create a prototype application that will allow indexing parts of the document with different levels of granularity, clustering them according to their mutual similarity by different clustering algorithms and processing them using various text retrieval approaches. The main goal is to enhance documents by additional navigation aid. The application should also visualize obtained data and make them available for further analysis in third-party applications. Never the less, it should be designed with respect to requirement on adding new extensions and features in the future.

The rest of this thesis is structured as follows:

The second chapter provides general introduction into text retrieval topic based on vector representation of documents that allows various needed computations known from linear algebra. Later it describes transformation of classical approach to indexing and processing document parts. Following third chapter takes closer look at implementation of visualization application. The fourth chapter describes an example visualizations. The following fifth chapter contains conclusion and future remarks. The user documentation and programmer documentation are located in sixth and seventh chapter.

2. Analysis

This chapter contains a short introduction into vector representation of documents and document clustering. Based on this representation it then describes the possibilities of algorithms for usage on single document and its parts.

2.1. Vector space and clustering concepts

Vector space model is a form of *ranked information retrieval*. The term *information retrieval* or *information search* in this context usually means finding the document(s) from a collection of documents, which satisfies needed information (based on definition in [1]). The result documents of the search are usually identified with respect to *search query*, a query that user formulated to express his information needs. In *ranked information retrieval* the *search algorithm* computes for each document the *rank*, the measure how much the document satisfies the *search query*, and then – using this rank – it decides which documents are to be returned as a search result and in what particular order.

There exist more document models allowing ranked query that differs from each other by document representation and/or by the used search algorithm. In *Vector space model*, each document has assigned representing *document vector*. This vector then represents the document content for the search algorithm. More detailed description of document vector could be found in section 2.1.2..

The *clustering* (or *data clustering*) is a process of grouping documents into *clusters*, where documents in one cluster should be as mutually similar as possible while documents taken from different clusters should be as different as possible from each other. In general, the clustering is not limited to documents, but can be used for various different types of data (images, music and other [2]). In this work we focus on clustering of document parts, such as chapters, paragraphs or sentences. More about it could be found in section 2.1.3..

2.1.1. Document vocabulary and terms

To create representing vector from a document, we first need to get information about its content. The common way of looking at document content, is considering it as a *bag of words*. This allows us to transform document text into *vocabulary* of *terms* and theirs frequencies – numbers of term occurrences inside individual documents. The first step is this process is a *tokenization* – a process of parsing the document text and splitting it into words (*tokens*). The *term* in this document represents a *token* in its normalized form. There are many normalization steps, which could use more or less complex computer linguistic approaches like *lemmatization*,

disambiguation, *named entity identification* etc [1]. The steps mentioned below do not require such sophisticated approaches while still provide sufficient processing for the purpose of this work. The *normalization* usually uses several steps:

- *Diacritics* – Many languages (including Czech and Slovak ones) are using diacritics in their official form. However, in many cases the text is not written with correct diacritics or is written without any diacritics at all. This is mostly common on the Internet, where many users don't use diacritics because of laziness, software limitations or habits. To solve this problems, the text could be stripped of diacritics and converted to 7-bit ASCII characters. The disadvantage of this approach is merging possibly different words into one common term.

- *Case-folding* – The words are converted to lower case. This allows to handle word with different case as the same. As a drawback, some abbreviations as "IT" could be converted to common terms.

- *Stemming* – Many different word occurrences in text could represent the same term, but differs in inflexion – prefixes or suffixes (for example *car* vs. *cars*). *Stemming* is a heuristic process that proposes an alternative to more complex *lemmatization*. It removes prefixes and suffixes from word and leaves only its base, called *stem*. For example, words *transporting*, *transported* and *transports* should be all considered as forms of common term *transport*.

- *Synonyms replacing* – In common language one think could be expressed using several different words – synonyms. For example, words *start*, *begin* and *initiate* have the same meaning for human, but they are different words for computer. A possible approach to help the computer to handle synonyms is to replace all terms with the same meaning with one term. This could be done using predefined dictionary of synonyms, called *thesaurus*.

- *Stop words filtering* – *Stop words* are usually referred as words, which occurs in language very often and usually don't have any essential meaning of their own. In English the typical words, that could be considered as stop words are *a*, *the*, *an*, *can* or *have* etc. By removing stop words, we could significantly decrease the space dimensionality – the number of different terms in index – and thus reduce process time and space required for indexes (according to *Rule of 30*[1]). During the text processing the *stop words* are usually predefined in *stop list*. The opposite approach could use *whitelist* of allowed terms in index instead of *blacklist* (stop list).

1 Rule of 30 states that the 30 most common words account for 30% of all terms in written text (Chapter 5 of [1]).

2.1.2. Document vector and indexing

In *Vector space model* the document d is represented by *n*-dimensional vector $\vec{v}_d$, where each dimension represents *weight* w_{d,t_i} of *term* t_i. In other words, the document vector is defined in form:

$$\vec{v}_d = \langle w_{d,t_1}, w_{d,t_2}, \ldots, w_{d,t_n} \rangle$$

The simplest way of computing term weights w_{d,t_i} is to define it as a number of occurrences of term t_i in the document. This is value called *term frequency* tf_{d,t_i}.

Using the *term frequency* is not optimal, because it considers all terms equally important. On the other hand, when we take documents about *insurance companies*, the term *insurance* would be probably very often in all of them, so it has almost none discriminating power. To reflect this, the *tf-idf weighting* could be used instead, which – according to [3] – produces better results in *data clustering*.

Before defining *tf-idf weight*, we need to define:

- *Document frequency* df_{t_i} – is defined as the number of documents containing the term t_i.
- *Inverse document frequency* idf_{t_i} – for term t_i is defined as:

$$idf_{t_i} = \log \frac{N}{df_{t_i}}$$, where *N* is total number of documents.

The term weight w_{d,t_i} now can be defined using *tf-idf weight* as:

$$w_{d,t_i} = \text{tf-idf}_{d,t_i} = tf_{d,t_i} \cdot idf_{t_i}$$

Looking at the documents as a vectors allows us to use standard vector operators and calculate lengths or distances. Furthermore we can easily compute *similarity* of two documents, where two documents are *similar* when their document vectors are directionally close to each other. The commonly used measure for this purpose is the *cosine similarity*, which is for two documents d_1 and d_2 in [1] defined as:

$$\text{sim}(d_1, d_2) = \cos(\varphi) = \frac{\vec{v}_{d_1} \cdot \vec{v}_{d_2}}{|\vec{v}_{d_1}| \cdot |\vec{v}_{d_2}|}$$

where φ is an angle between vectors $\vec{v}_{d_1}$ and $\vec{v}_{d_2}$.

This *cosine similarity* allows us to compare relative distribution of terms in document independently of the document vector length.

Another similarity measure could be based on *Jaccard coefficient* described in [4] as:

$$J(d_1, d_2) = \frac{|d_1 \cap d_2|}{|d_1 \cup d_2|}$$

where $|d_1 \cap d_2|$ represents the number of terms the documents have in common and $|d_1 \cup d_2|$ represents the total number of terms occurring at least in one of these

two documents. The occurrence of term in document is binary evaluated and does not take into account weights of terms.

There are many other variants of Jaccard coefficient, of which probably the most common is *Generalized Jaccard coefficient*, which is a proven metric – according to [5] – and is defined as:

$$GJC(d_1, d_2) = \frac{\sum_{i=1}^{n} min(w_{d_1,t_i}, w_{d_2,t_i})}{\sum_{i=1}^{n} max(w_{d_1,t_i}, w_{d_2,t_i})}$$

The advantage of Generalized Jaccard coefficient over its basic form is that it reflects weights of terms in the documents.

2.1.3. Document clustering

Using *clustering*, documents could be divided into *clusters*, where document pairs taken from the same *cluster* should be as similar as possible and documents in one *cluster* should be as dissimilar as possible from those in any other *cluster*.

Based on the organization of clusters, we can distinct two basic types of clustering:

- *Flat clustering* – the set of clusters is defined without any explicit organization between individual clusters.
- *Hierarchical clustering* – created clusters are organized in hierarchical structure.

Another important distinction of clustering is based on document assignment. From this point of views we can talk about:

- *Hard clustering* – where each document is assigned to exactly one cluster.
- *Soft clustering* – where document could be assigned to more clusters. In case the assignment is weighted, we are talking about so-called *Fuzzy clustering*.

The clustering of document collections is commonly used in web search, to speed up finding documents or pages matching the user's query by eliminating dissimilar cluster content from similarity evaluation. Moreover, the clustering could be used to increase diversity of the result by clustering resulting set of found documents. For example, when the user enters query *"apple"*, he or she may want to find information about the fruit, about the music recording company or about the computer manufacturer. But the search engine doesn't know which one the user wants, so by showing documents from different result clusters in the first page, it is probable that the user will find some relevant document sooner.

Another example use of clustering is creating a summary of document collection. This could be done by grouping similar documents into clusters and then replacing all documents in each cluster with surrogate piece of text, which represents them.

2.1.4. Spherical K-means clustering algorithm

K-means presented in [6] or its modification for information retrieval – *spherical k-means* – is one of the most important flat clustering algorithms, with hard assignment of documents. The algorithm starts by creating *k* random clusters, also known as *seeds*. Then it tries to optimize document assignment to clusters according to similarity of documents vectors to their cluster *centroid*, computed as mean vector of assigned documents. The value of *k* is needed as a parameter. The algorithm could be formally described as it is shown below in Algorithm 1:

```
K-means (k, { v⃗_{d_1}, ..., v⃗_{d_n} })
begin
    create k initial random seed clusters
    recompute clusters centroids
    while not clusters are stable
    begin
        for each document
        begin
            reassign document to cluster with closest centroid
        end
        recompute clusters centroids
    end
    return k clusters
end
```

Algorithm 1: K-means

Within each iteration in the while cycle, the algorithm reassigns document to the closest centroid and then recomputes the centroids. This way the centroids move around vector space to find their optimal assignments.

The *k-means* algorithm provides following advantages:

- It is one of the most used clustering algorithm, because it is relatively quick on large data sets. Its complexity is linearly proportional to the number of documents.
- Works well on numerical data.

On the other hand, it has several disadvantages:

- Result clusters have convex shapes only.
- The value of parameter *k* needs to be defined in advance.

- It provides only local optimization and could find an assignment, which is suboptimal in global scale.
- Its performance depends on initial random selection of k clusters. Different runs starts from different random assignments of vectors to clusters and thus could convert to different local maximum – different result.
- Performs poorly on high-dimensional data (such as textual documents, [7]).

2.1.5. Hierarchical agglomerative clustering algorithm

Hierarchical agglomerative clustering (HAC) represents one of the basic hierarchical clustering algorithms, presented in [8]. The algorithm starts with separate cluster for each document and in each step it creates new cluster, linking two most similar clusters (Algorithm 2) together.

Based on the way the similarity between two clusters is computed, HAC creates a whole class of algorithms. For example:

- *single link* – the similarity between two clusters is computed as the similarity of their most similar members
- *complete link* – the similarity between two clusters is computed as the similarity of their most dissimilar members
- *average link* – the similarity between two clusters is computed as an average value of similarities of all couples
- *centroid* – the similarity between two clusters is computed from their centroid vectors

```
HAC ({ v⃗_{d_1}, ..., v⃗_{d_n} })
begin
    create separate cluster for each document
    while count(clusters) > 1
    begin
        find two most similar clusters
        replace them with one new cluster, which is linked to them
    end
    return clusters
end
```

Algorithm 2: Hierarchical agglomerative clustering

The HAC algorithm provides hierarchy of clusters representing the topic hierarchy of documents and it is considered as one to provide the best clustering results. However, it is not very usable on large data sets, because it is very slow.

The algorithm usually stops when all clusters have been linked and only one – forming the root of the cluster tree – is left. The algorithm could be also expanded

with stop condition, which stops the linking of clusters when similarity of the clusters reaches predefined threshold value. Resulting cluster hierarchy would have broader root linked with all obtained top clusters. In general it is hard to tell the proper value of the threshold. Better result can be obtained by cutting through the final hierarchy.

As an opposition to HAC there is also *hierarchical divisive clustering*, which starts with only one cluster and iteratively splits largest clusters until individual documents or small enough clusters are reached.

2.1.6. Cluster labeling

Once we get the set of clusters we may need to present them to the user. For this purpose we would like to add a label to each of the clusters. The label should correspond with the content of the items contained within the cluster. This way the user will have approximate overview about the content of the cluster.

The label could be a word, group of words or a sentence. A good label should be short, should correctly describe contents of its cluster and in case of sentences, it should be in correct grammar form. There are several approaches, how to generate cluster labels.

The simplest approach to get a label for a cluster is to take one or more words with largest sum of weights from each of the documents within. In case a centroid vector for the cluster is known, we could take the words having biggest weights in it.

Another simple labeling approach is to choose a sentence, which is most similar to its centroid vector. This way we could get a grammatically correct sentence.

More complex approach is to use a modified version of *Information gain* function presented in [9] and [10]. For term t_i and cluster c_j, $P(t_i)$ is the probability that document contains term t_i, $P(c_j)$ is the probability that document is in cluster c_j and $P(\neg t_i)$ and $P(\neg c_j)$ are defined as $1-P(t_1)$ and $1-P(c_j)$. $P(t_i, c_j)$ is then the probability that document is in cluster c_j and contains term t_i, $P(\neg t_i, \neg c_j)$ is the probability that document is not in cluster c_j and does not contain term t_i, $P(t_i, \neg c_j)$ is the probability that document is not in cluster c_j but contains term t_i and $P(\neg t_i, c_j)$ is the probability that document is in cluster c_j and does not contain term t_i. Using this notation, the *Information gain* is defined as:

$$IG(t_i, c_j) = \sum_{x \in \{t_i, \neg t_i\},\, y \in \{c_j, \neg c_j\}} P(x, y) \cdot \log \frac{P(x, y)}{P(x) \cdot P(y)}$$

As explained in [4], with this definition the Information gain takes into account the presence of term in the cluster and also its absence. The high value of $IG(t_i,c_j)$ means, that the presence or absence of t_i in a document tends to be highly indicative of the document being or not-being in cluster c_j. Because we need to select words for a cluster label, we are not interested in indicative words not being in the cluster.

Therefore Information gain could be modified to account only for the presence of word within clusters:

$$IG_m(t_i, c_j) = P(t_i, c_j) \cdot \log \frac{P(t_i, c_j)}{P(t_i) \cdot P(c_j)} + P(\neg t_i, \neg c_j) \cdot \log \frac{P(\neg t_i, \neg c_j)}{P(\neg t_i) \cdot P(\neg c_j)}$$

2.2. Application of the theory to a single document

A single document could be viewed as a hierarchy of text fragments – chapters, paragraphs and sentences. Depending on a level of granularity, it could be also viewed as an ordered collection of chapters, ordered collection of paragraphs or as an ordered collection of sentences. This way it is possible to use any of algorithms originally invented for work on document collections on a collections of document fragments with only minor changes. The key differences are:

- Instead of single collection of separate documents, three parallel collections of chapters, paragraphs and sentences are used.
- Chapters, paragraphs and sentences are ordered and text fragments in them form a hierarchy. This structure is defined by the document itself. We could consider this hierarchy as a special type of hierarchical clustering where the similarity is derived from proximity.
- The vector in vector space model would represent not only whole document as in case of standard approach, but could also represent every single chapter, every single paragraph or even every single sentence, depending on currently working level of granularity.

The modified view on the document is shown in Figure 1. For the simplification, only one chapter level is considered. Possible sub-chapters are ignored and their text is taken as a content of top most chapter. Implementation of this approach in proposed experimental application is discussed later in section 3.7.1.. For each node *d* of the hierarchy we can define:

- *Predecessor* of *d* – the closest previous text part on the same level of hierarchy.
- *Successor* of *d* – the closest next text part on the same level of hierarchy.
- *Parent* of *d* – the text part one level of hierarchy higher, which contains the node *d*.
- *Children* of *d* – ordered collection of text parts one level lower, which are contained by the node *d*.

Each node has assigned its own vector of term frequencies and representing vector of weights. The node representing a document part on different levels in the

hierarchy would be referred as document, chapter, paragraph or sentence and the corresponding representing vector would be referred as *document*, *chapter*, *paragraph* or *sentence vector*.

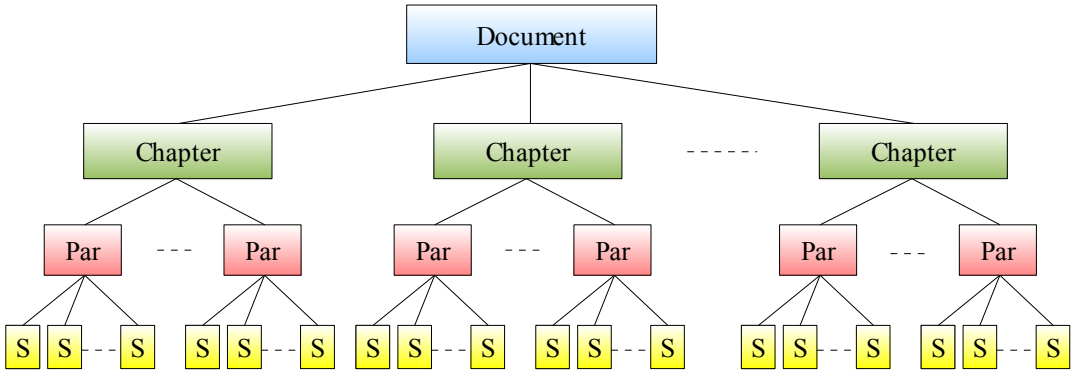


Figure 1: Overview of document hierarchy

Using clustering on document chapters, paragraphs or sentences, we will get parallel hierarchies of clusters, organized by the real similarity of their content.

2.2.1. Term, vocabulary and indexing of single document

Because we are looking at the single document as a hierarchy, we need to slightly adjust the theory used in standard approaches, described in section 2.1..

All vectors of all granularity levels must be compatible – have the same dimension, because they will be used together in computations. Therefore, we would need a global vocabulary for all parts. On the other hand, each document part – chapter, paragraph or a sentence – would need to know its term occurrences to compute its vector.

The vector of document part could be define similar as document vector $\vec{v}_d$, with a d representing document part – chapter, paragraph or sentence. The difference would be in computing term weights using *tf-idf weight*. The *idf* would be derived from the number of paragraphs within the document and number of paragraphs containing given term. This *idf* computation would be used globally for all parts on all levels of hierarchy to get compatible numbers.

Because of that each inner node d in the hierarchy consists of a concatenation of its children. It holds that the paragraph vector would be the vector sum of vectors of its sentence children etc up to the document vector would be the vector sum of vectors of its chapter children.

2.2.2. Example usage of vector model on single document

The document is at first converted to hierarchy of vectors that represents its content, and enables further processing and content analysis, such as:

- Computing similarity between chapters, paragraphs or sentences, discovering similar or the same parts that are spread across the document. This could be used for better orientation in documents, when the user would be provided with information about similar document parts, located in other chapters. For example, the user could get links to most similar chapters or link(s) to the closest chapter(s), with similarity bigger than defined threshold(s) etc.
- Analyzing consistency of document flow. For example, if consecutive paragraphs in one chapter or consecutive chapters in the document are most similar to each other or not.
- Finding parts, which represents their parents the best. For example, finding a paragraph, which represents best the content of its chapter etc.

2.2.3. Example usage of clustering on single document

The use of clustering algorithms would group together similar content of the document. This would allow to:

- Identify the topics of the document and document parts which talks about them.
- Create summary of the document, by extracting topics from the document content. The topics could be represented by clusters, where each cluster would represent one topic. Summary then would be created by taking part of document (few sentences) from each cluster. This approach is closely described later in section 2.4..
- Extract keywords of the document in similar way, how summary could be created.
- Provide parallel navigation in the document by categorizing document parts into groups similar by content.
- Analyze consistency of document parts. For example, how much computed sets of clusters correspond to their positions in the document.

2.3. Document summarization based on affinity graph

Document summarization method based on affinity graph, information richness and a diversity penalization was proposed in [11]. This was based on affinity ranking framework, which was proposed in [12] as a way to improve search performance.

In the context of one document it is possible to define:

- *Information richness* – in a document $InfoRich(s) \in [0,1]$ denote the information degree of sentence *s* – the richness of information contained within *s* with respect to the entire document.
- *Diversity* – the number of different topics within given document.

Document summarization is done in several steps, which are closely described in following sections.

2.3.1. Construction of affinity graph

At first we need a similarity matrix $M = (m_{ij})_{n \times n}$, which is created from sentences using similarity measure previously defined in 2.1.2. with regards to a single document as described in 2.2.1.. The value of similarity matrix *M* is defined as:

$$m_{ij} = \mathrm{sim}(\vec{v}_{s_i}, \vec{v}_{s_j})$$

where $\vec{v}_{s_i}$ and $\vec{v}_{s_j}$ are vectors of i-th and j-th sentence.

In further computations we would use a normalized similarity matrix $\tilde{M}$, which has the sum of each row of *M* normalized to 1.

If we consider sentences as nodes, using normalized similarity matrix $\tilde{M}$ we could create links between sentences s_i and s_j if $m_{ij} > 0$. Otherwise no link is created. This gives us an undirected weighted graph reflecting the relationship between sentences based on their similarities. This graph is called the Affinity graph,

2.3.2. Computation of information richness

The computation of information richness is based on following intuitions:

1. The more neighbors sentence has in Affinity graph, the more informative it is.
2. The more informative sentence's neighbors are, the more informative the sentence is.

Based on above intuitions the information richness could be defined in recursive manner as follows:

$$InfoRich(s_i) = c \cdot \sum_{j \neq i} InfoRich(s_j) \cdot \tilde{M}_{ij} + \frac{1-c}{n}$$

where $i, j \in 1, \ldots, n$ and *c* is a dumping factor usually set to 0.85.

2.3.3. Computation of affinity rank using diversity penalty

Using information richness and affinity graph the affinity rank is computed by implying diversity penalization. This is done be greedy algorithm where with each step a sentence with highest current affinity rank is moved away and affinity rank is recomputed by implying a diversity penalty. The algorithm goes as follows:

1. initialize two sets: $A = \emptyset$ and $B = \{s_i | i = 1 \ldots n\}$;
   for each (i in 1 ... n): set $AR(s_i) = InfoRich(s_i)$
2. sort all sentences in B descending by their current affinity rank AR
3. take sentence s_i having highest rank in B and move it from B to A
   for each (j in 1 ... n, j=i):
   $AR(s_j) = AR(s_j) - w \cdot \tilde{M}_{ij} \cdot InfoRich(s_i)$
4. if $B \neq \emptyset$ then goto 2
   else stop the algorithm and return set A

The $w \in (0,1>$ in step 3 is a dumping weight constant, which tells how strongly should the penalty be applied. The default value is 1.

Note that by this algorithm the values of Affinity rank $AR(s_j)$ could become negative. However, because we are always selecting the maximal value, it does not matter.

2.3.4. Selection of sentences for summary

After the affinity rank is computed, the sentences found in first iterations are selected into the summary text, until the length requirements are met.

This method has an advantage that once we had the affinity rank computed, we could easily generate more summaries of different length from the same document.

2.4. Document summarization using clustering

As mentioned in section 2.2.3., clustering could be also used to create document summary. The method described here is based on [13] with a difference in computing local and global similarities.

The summary of documents would be generated by selecting sentences from the text of the document. For each cluster, the sentence with maximal score is selected. The score of each sentence is computed as a weighted sum of following factors:

- Local similarity – It is computed as similarity between sentence and centroid of containing cluster. The more similar sentence to centroid, it is probable that it could best reflect the contents of the cluster.

- Global similarity – The similarity between sentence and the document, which ensures that the global context is reflected in selection.

- Sentence length – The length of summary could often be limited. This factors adds penalization for too short or too long sentences. The sentence length factor is defined as follows:

$$factor_{length} = \frac{1}{e^{|length(sentence) - length_{required}|}}$$

3. Implementation

This chapter describes implementation of simulation application *SimDIS*. It contains application design, describes used data structures and necessary interfaces.

3.1. Design and main decisions

The application should serve as an experimental visualization tool with presumed rich interaction with the user. Therefore, the application with graphical user interface (GUI) is preferable to the console application.

Expected typical use case example usage of the application would be:

1. Opening of desired document.
2. Splitting document to its parts and computation of representing vectors.
3. Analyzing mutual similarities of objects at given level of granularity.
4. Computation of clusters of objects using selected clustering algorithm.
5. In many cases the user would probably like to visualize the data in some understandable way.
6. Explore the results and/or export them for further processing outside the application.

Because the user will probably work with one document more times in different scenarios, the application should import it locally. So the user would not have to search for the document in file system over and over.

Because the user may want to work on more documents, the application should allow to store more documents at the same time. To allow better management or moving between computers, the application should save documents grouped within projects, which are closely described later in 3.4.4..

To allow easy importation of documents, the application would support documents in HTML [14] file format. It is easy to analyze it and many third-party tools could be used to convert documents in almost any other format to it. It should be also possible to add support of more file formats later. Internally the document should be stored in custom format, best suiting the application needs for fast loading and processing. Import of document in any external format then should provide necessary conversion. The document list with document meta-data would be stored separately from the documents themselves for better performance. Used data structures are described in section 3.3..

As complex technical manuals could contain thousands and more documents parts, the application should support caching of already computed data to avoid

repeating of time-consuming computations, like clusters and/or similarity matrices evaluation etc. Caches are closely described within project in section 3.4.4..

The application should be also easy to expand implemented features and to add new ones, because there exist more than one ways of text processing and visualization and it would be impossible to implement them all at once. The expandability of the application is discussed closely later in section 3.5..

3.2. Development tools

SimDIS is a GUI application written in C# language for *Microsoft .Net Framework* [15] with minimal required version 2.0. The application is targeted to run under *Microsoft Windows* [16] operating system. This platform was chosen mainly because it is most widely used.

Some of the data are stored in *SQLite* database [17] using *System.Data.SQLite* library [18]. This database is small, doesn't require user to install any additional software and data can be stored within a single file.

The application was developed under *Microsoft Visual Studio* development environment [19] and for better source code management, Subversion [20] was used.

3.3. Data structures

The application uses several data structures for storing parsed document tree, vectors, terms and clusters. Almost all of them needs to be accessible from plugins to be able to normalize terms, to run data visualizations or to create clusters. This is the reason, why they are stored in *SimDIS.PluginInterface*.

The most notable data structures provided are:

- *Document* – the root of document hierarchy, consisting of *Chapters*, *Paragraphs* and *Sentences*. All of them are derived from *ADocumentPart*, an abstract class for document part.
- *DocumentVector* – vector representation of document part.
- *DocumentTerms* – the hierarchy parallel to Document, containing a set of *Terms* for each document part.
- *Term* – containing information about normalized term, its frequency and also list of original words used in document .
- *Cluster* – the hierarchy of clusters, created by clustering algorithm. Clusters contain the set of sub clusters in case of an internal node and a set of document parts in case of the leaves.

Detailed documentation of data classes with descriptions of methods and properties could be found in Code documentation on attached CD.

3.4. Application architecture

The Figure 2 bellow illustrates the architecture of the application from the logical view. Parts of the application are described in following sections.

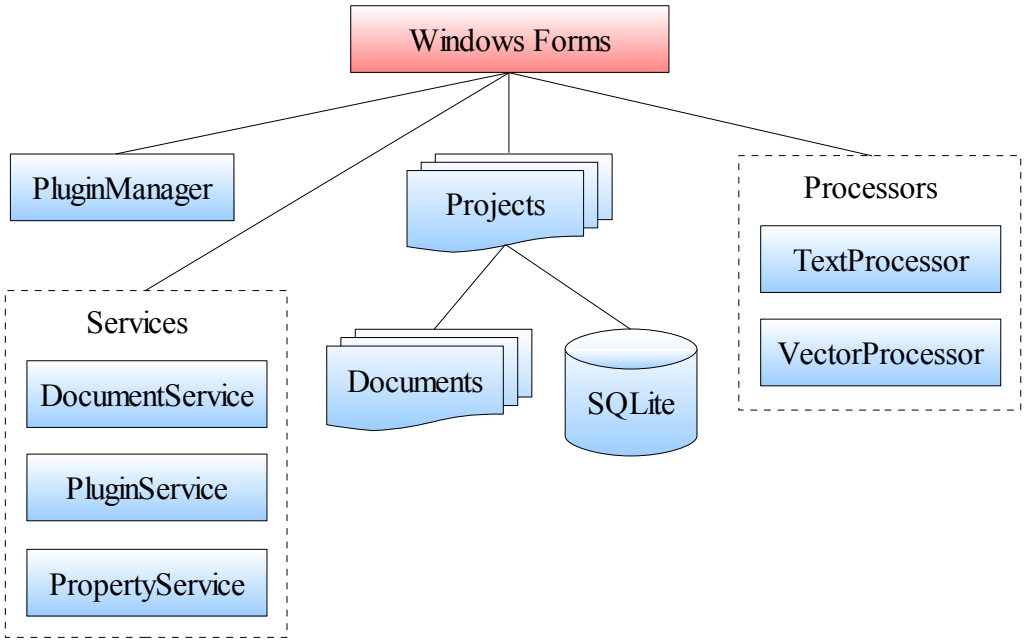


Figure 2: Overview of SimDIS core application

3.4.1. Windows forms

In the center of the application are *Windows Forms*, the GUI windows, which are responsible for interaction with the user. The most notable forms are:

- *MainWindow* – Displays list of documents stored in current project and allows document import and export. The list of documents is loaded from SQLite database and displayed using standard *DataGridView* component of .Net Framework.

- *DocumentWindow* – Represents one opened document and allows to further work with it. The document is displayed as a HTML using *WebBrowser* component. The window allows to apply modifications to the document and also shows a list of visualizers, loaded from plugins, as a tool buttons to run visualization.

There are several other forms used in programs, mostly using standard components of .Net Framework.

Visualization plugins also contains window forms to visualize data to the user. They use their own forms, which are not limited or predefined by core application. This provides unlimited potential for data visualization.

3.4.2. Plugin manager

PluginManager is a static class, which takes care of all types of plugins. It loads all available plugins at the application start-up, holds lists of them and also provides some useful methods to work with them. Plugins are more described in section 3.5..

3.4.3. Processors

In this application the processors are static classes in *SimDIS.Processors* namespace, which encapsulate text processing and creation of terms and document vectors. There are following processors:

- *TextProcessor* – is used for parsing document text and creating hierarchy of terms for document parts. The main methods is *createDocumentTerms(Document)*, which takes the document as parameter and walks through its parts and extracts words from them. From the words, the normalized terms are created, using active normalization plugins. If the normalization would result in and empty string, the word is considered as a stop word. At the end, the *DocumentTerms* object is created.
- *VectorProcessor* – is used for creating document vector hierarchy from already prepared hierarchy of document terms – the *DocumentTerms* object. The *createDocumentVectors(Document, DocumentTerms)* walks through document parts and from prepared terms it creates and assigns instances of *DocumentVector* to them.

3.4.4. Projects

The projects are used as a container for storing documents and user work. They also allow to easy transfer the work from one computer to another and to have multiple parallel projects saved in application at the same time.

Physically each project is saved as a separate sub-directory of *projects* directory located under the user-defined workspace directory. The location of workspace is stored in *Windows registry* in:

HKEY_CURRENT_USER\Software\SimDIS

The project directory contains:

- XML file project.xml with basic description of the project (for example, name of the project). It is used only to quickly identify the project. The example of the file is presented in Figure 3.

```
<?xml version="1.0" encoding="utf-8"?>
<project>
      <project-name value="The name of project" />
</project>
```

Figure 3: Example of project.xml file

- SQLite database in file storage.sq3, where for example list of documents is stored. SQL provides better and faster access to data collections or changing data than XML. The database also contains user-configurable data like properties or list of enabled plugins and their order.

- Saved documents and computation cache as separate files, using serialization of .Net Framework. The serialization has been used because it is much faster than SQLite, when storing lots of textual data and object hierarchy. The name of each cache file is derived from the document to which it belongs, cache name and ".cache" extension. For example:
  document_3_SimilarityParagraphs.cache

The list of projects is accessed through static class *SimDIS.Project.ProjectManager*, which could check the *projects* directory and get the list of all projects by calling *getProjectList()*.

Once opened, the project is represented by *Project* class in *SimDIS.Project* namespace and is partially defined in more files to increase readability of the code. The *Project* class provides functionality regarding the project and its documents, such as:

- Loading and saving the project itself – this is done automatically in its constructors, where from the parameters it knows whether is should load from directory or a new project is being created.

- Managing SQLite database – the database is accessible by the *DocumentStorage* property, which returns *DbConnection*. By accessing the property, database connection is automatically initialized. When it is accessed for the first time and the database file doesn't exist, the file is automatically created including SQL schema. This is done by *initializeDb()* and *createDbSchema()* methods.

- Saving document to project using *saveDocument()*, loading using *getDocument()* and deleting using *deleteDocument()*. Title and name of the document is saved into database and the document it selves is stored into separate file using serialization of .Net Framework.

Getting the list of authors and titles of stored documents using *getDocumentTitles()*. The list is loaded from database.

3.4.5. Services

The service classes provide various utility methods. They could be used within the core application and through the interface mapping they could be used from any plugin as well. The mapping for plugins is accessible through class:

SimDIS.PluginInterface.Common.Services

The following services are available:

- *DocumentService* – provides functionality regarding the actual document, such as computing the similarity between two vectors or cutting through cluster hierarchy according to current configuration.
- *PluginService* – wraps functionality around plugins and plugin manager itself. For example, it determines the order of normalization and filtering plugins.
- *PropertyService* – covers complex functionality regarding the configuration properties. As mentioned before, the properties are stored in the database. Inside the application each property is mapped to *ConfigurationProperty* object. The service provides methods to find, save and even parse configured values.

3.5. Plugin interfaces

To meet the requirements for a tool for testing, visualizing and analyzing impact of vector model and clustering used on document and its parts – The *SimDIS* application implements several tools for loading documents, parsing them, indexing their sub-parts, analyzing their mutual similarities, clustering them and visualizing the results. For better extensibility and easier maintenance in the future the application uses plugins for individual visualization tools, implemented clustering algorithms etc. All plugins are loaded at the start of the application from external dynamic-link libraries (DLL's). This allows adding new features without the need for source code of core application and its recompilation.
The features which may be desirable to be added or modified later are:

- More supported file formats – the file types for importing and exporting documents to/from the application. For example, it would be suitable to allow processing documents in some of e-book format as epub or another XML based format.
- Term normalization and filtering – the application should allow to add more complex filtering steps within the term normalization process discussed in section 2.1.1..
- Similarity measures – there are many various metric measures, which are possible to use as a similarity. Some of them were described in section 2.1.2..
- Clustering algorithms – there are many various algorithms, which vary in way the clusters are computed.

- Cluster labeling – as discussed in 2.1.6., there are more possible approaches to creation of cluster labels.
- Clustering and non-clustering visualizations – would allow to add different ways to look at document and its content.
- Document manipulation – based on computed data, various information could be added to the document. For example, additional navigation, links similar parts or navigation through the cluster.

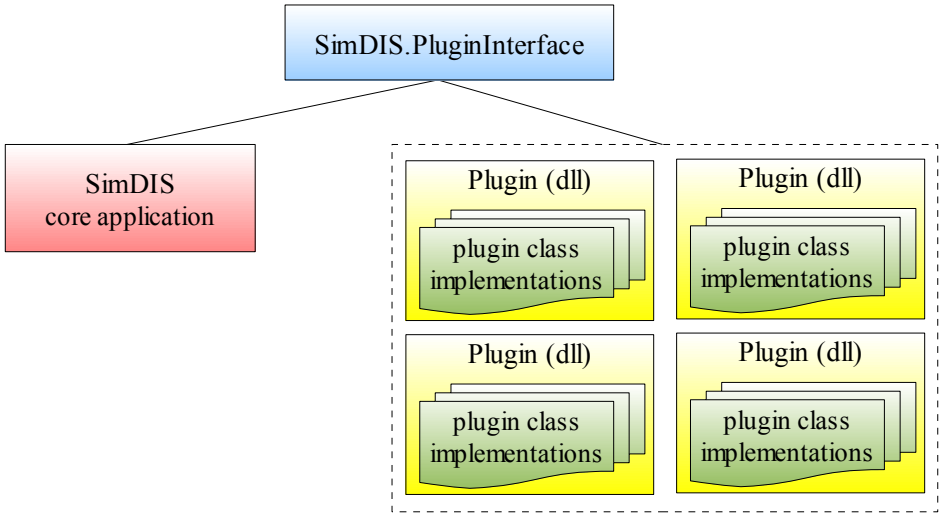


Figure 4: Basic concept of plugin implementation

Figure 4 shows the concept of plugin implementation, where the left box represents the application core part, which links the *SimDIS.PluginInterface* library depicted by box at the top of the picture, which makes accessible all necessary interfaces to plugins trough their respective and maintains data, needed by plugins to work with document. Boxes at the right side enclosed in dashed box represent the implementations of interfaces via plugins in application context. A plugin is a container which could contain several different plugin classes of following types inherited from *IPluginBase*:

- *IFilePlugin* – implements support for new file types to importing and exporting documents.
- *IWordNormalizerPlugin* – implements one step in term normalization process, like case-folding or stemming, etc. Normalization steps are closely described in section 2.1.1..
- *IWordFilterPlugin* – implements one type of term filtering. For example filtering using stop words, which was described within term normalization steps in section 2.1.1..

- *ISimilarityMeasurePlugin* – the implementation of similarity measures.
- *IAlgorithmPlugin* – the implementation of clustering algorithms.
- *ILabelingPlugin* – the implementation of cluster labeling approaches.
- *IVisualizerPlugin* – the non-cluster visualization, visualizing mainly chapter vectors, paragraph vectors, sentence vectors or document terms. To this category fits almost anything else, for example, plugin that displays similarity matrix of paragraphs or plugin that shows table of processed document terms.
- *IClusterVisualizerPlugin* – for the visualization of clusters. For example, plugin that displays clustered document parts as a browseable tree.
- *IDocumentModifierPlugin* – a plugin for inserting additional informations based on computed data directly into the document.

Each plugin is represented by one or more classes, where each one implements one or more of previous interfaces. The plugin is a dynamically loaded library (DLL), which must be located in *plugins* directory and could contain more than one plugin class.

The plugin directory is scanned at the startup of application and found plugins are loaded and registered automatically. This is done by *SimDIS.Plugins.PluginManager* using *System.Reflection.Assembly* and *System.Activator* of .Net framework. In loading process, each class from each DLL is checked, whether it implements some of the plugin interfaces mentioned above and if so, it is registered – added to appropriate list within the *PluginManager*. Later in application those lists are used to generate menu items, get supported file type or normalize words.

The *PluginManager* also provides methods to process some of the plugin functionality:

- *openDocument()* – reads document from file, using appropriate file plugin
- *saveDocument()* – saves document to an external file, using appropriate file plugin

To allow DLL to contain more than just plugin classes and for better orientation in files and classes, a naming convention is enforced:

- The name of plugin DLL must end with “Plugin” (for example, BasicFilePlugin.dll) to be loaded by application.
- The name of plugin class must end with one of: “Visualizer”, “VisualizerCluster”, “File”, “Similarity”, “Algorithm”, “Labeling”, “Filter”, “Normalizer” or “Modifier”. Otherwise the class will not be loaded.

3.6. Configuration properties

Several algorithms need some value as a predefined parameter, e.g. the K value of K-means algorithm from 2.1.4.. Because *SimDIS* application is targeted on experimental usage a possibility to easily change these parameters is required. What is more, even the core application needs to be configurable, e.g. to choose similarity measure to use. Therefore a unified configuration using properties was implemented.

The properties are stored in basic key-value pairs, where each property must have its unique key name. In application it is represented with *ConfigurationProperty* object, which – besides key and value – also contains additional information, such as descriptive name, default value and for internal use also information, whether it should be editable in configuration editor directly by the user.

Every plugin class has to implement *getPluginProperties* method, where it could return a list of *ConfigurationProperty* objects. This list defines all custom properties the plugin wants to use. This method is used by application to detect all properties.

The properties are configurable by user in Configuration window inside the application and are stored into the project database. The configuration could be different for each project.

Inside the application or within the plugins, the properties could be accessed using the PropertyService as described in 3.4.5..

3.7. Implemented plugins

A lot of important functionality is implemented as a plugin and as a part of this application, various plugins were implemented as normalizers, visualizers or clustering algorithms etc. This section describes some of them.

3.7.1. Input HTML parser

Simple HTML parser is contained within *HtmlFile* plugin. It parses the input file in several steps using regular expressions and trying to extract the contents. The basic principle is to extract the text and ignore everything else.

Although HTML standard allows to use H1 heading more times, many web documents use it only for a title and for actual chapters H2 or higher is used. Therefore the parsers takes both H1 and H2 as a beginning of new chapter. Other headings and tags DIV, TD and P are considered as beginnings of paragraphs.

Except from that, the parser also tries to extract the title of the document from the TITLE tag and the author from the META tag.

3.7.2. Term normalizers and filters

The normalization is used for grouping together words with similar meaning, but different written form. It is part of document processing, which is described in section 2.1.1..

Term normalizers are implementations of *IWordNormalizerPlugin* with *normalize(string)* being the main method. The method takes word as a parameter and applies one step of normalization on it.

The execution of normalizers is piped in order defined by the user in Configuration window. The user could also exclude specific normalizer(s) from execution.

Following normalizers were implemented:

- *ToAsciiNormalizer* – removes diacritics.
- *ToLowerNormalizer* – converts all characters to lower-case.
- *TrivialStemmerNormalizer* – represents a trivial implementation of word stemming. It tries to remove English, Slovak and Czech prefixes and suffixes to get their stems. It uses regular expressions to find first suitable prefix and suffix to remove. Because it mixes multiple languages, it cannot guarantee that the stem would be generated always correctly. On the other hand, this cannot be guaranteed even by more complex stemmers.
- *ThesaurusNormalizer* – represents a simple implementation of synonym replacing algorithm using predefined dictionary, which defines words with the same meaning. The dictionary is stored as definition for replacements in *thesaurus.txt*, where each line define a word to replace in format from>>to.

After the term is normalized, configured filters are applied. The filters are implementations of *IWordFilterPlugin* interface. The main method is the boolean *isFiltered(string)* one, which should return true if and only if the given word should be filtered and marked as a stop word. Following filtering methods were implemented:

- *StopWordsFilter* – compares word against stop list located in *stoplist.txt* file. If given word is a stop word, it results and empty string. The stop list contains stop words from English, Slovak and Czech language in lowercase 7-bit ASCII form and their stems created by the *TrivialStemmerNormalizer*.
- *WhiteListFilter* – compares word against so called white list from file *whitelist.txt*. The word is marked as a stop word if it is not presented in the list. Current file contains basic set of English words, but to be used, it is highly suggested to provide a domain specific list.

3.7.3. Similarity measures

As presented in section 2.1.2. there are more measures to compute similarity. Therefore similarity measures were moved into plugins implementing

ISimilarityMeasurePlugin interface and the user may choose in Configuration window which one to use.

Plugins *CosineSimilarity*, *JaccardCoefficientSimilarity* and *GeneralizedJaccard–CoefficientSimilarity* were implemented as a representation of Cosine similarity, Jaccard coefficient and Generalized Jaccard coefficient.

3.7.4. Clustering algorithms

Clustering algorithms are implementations of *IAlgorithmPlugin* and their main purpose it to create clustering from set of document parts (*ADocumentPart*) in method *computeCluster()*.

There are following groups of clustering algorithms implemented:

- *Flat clustering* – represented by *KMeansAlgorithm*, the implementation of K-Means clustering algorithm. In addition, *TrivialListAlgorithm* was created, which just puts each part in its own cluster.

- *Hierarchical clustering* – representation by *CentroidHACAlgorithm*, the implementation of hierarchical agglomerative clustering using centroid vectors.

- *Trivial Clustering* – In addition, *TrivialListAlgorithm* was created, which creates flat structure of clusters simply by putting each part in its own cluster. This algorithm allows applying algorithms as cluster labeling on individual chapters, paragraphs, etc.

The result of clustering algorithm is a hierarchy of *Cluster* objects. In case of flat algorithms the hierarchy consists of one level linked under fictional cluster root.

3.7.5. Cluster labeling

Cluster labelers are implementations of *ILabelingPlugin*. In section 2.1.6. three possible approaches were discussed: most common words, most similar sentence and Modified information gain. All of them were implemented in *CommonWord–Labeling*, *SentenceLabeling* and *ModifiedInformationGainLabeling*.

The user could choose which one to use in Configuration window.

3.7.6. Document visualizers

Document visualizers, the implementations of *IVisualizerPlugin*, are used for non-cluster visualization. Some of the visualizations, that were implemented:

- *WordsHtmlVisualizer* – allows to explore results of document processing, by showing tables of terms for each document part embedded in text. The output is displayed using combination of HTML and JavaScript in *WebBrowser* component.

- *DocumentTermsVisualizer* – shows the document term vocabulary table using *DataGridView*.

- *ChapterSimilarityVisualizer*, *ParagraphSimilarityVisualizer* – display a similarity matrix between chapters or paragraphs. The output is displayed in *SimilarityWindow* form using *DataGridView* component or alternatively as image, where each pixel represents one value of similarity matrix in gray scale. The pixel is the brighter the bigger the similarity is. While grid view provide exact information about similarities, the image viewer allows obtaining quick overall insight into similarity distribution.

- *ChapterConsistencyVisualizer*, *ParagraphConsistencyVisualizer* and *SentenceConsistencyVisualizer* – compute similarity of document part with its predecessor, successor and with its parent. The output is displayed in *GridWindow* form.

- *SentenceSummaryFromDiversityVisualizer* – by using affinity graph, information richness and diversity, as closely described in 2.3., this plugin creates document summarization of required length, entered by the user at the top of summary visualization window. Other parameters of this algorithm are configurable in Configuration window.

- *ParagraphSummaryFromDiversityVisualizer* – uses the same approach with affinity graph as described in 2.3., but instead of sentences, it uses paragraphs. The result would be a set of paragraphs which were chosen to be in resulting summary.

Many of above mentioned visualizers also provide ability to export data for further analysis. The formats of exported files could vary with type of visualized data. For example, the similarity matrix is possible to export as HTML file, CSV file and/or grayscale PNG file.

3.7.7. Cluster visualizers

Cluster visualizers are made as implementations of *IClusterVisualizerPlugin*. Their purpose it to provide visualization of document parts clustering. Examples of implemented cluster visualization:

- *SentenceTreeVisualizerCluster* – displays clustering of sentences in browsable tree.

- *ParagraphTreeVisualizerCluster* – displays clustering of paragraphs in browsable tree.

- *ChapterTreeVisualizerCluster* – displays clustering of chapters in browsable tree.

- *ChapterSummaryVisualizerCluster*, *ParagraphSummaryVisualizerCluster* and *SentenceSummaryVisualizerCluster* – using the method described in section 2.4. this visualizer allow to generate text summary. The summary is generated from clusters of corresponding document parts – clusters of chapters, paragraphs of sentences.

Cluster visualizers allows to set the *threshold* for cutting the cluster hierarchy when using hierarchical clustering. For example, the threshold of 0.4 will cut the cluster hierarchy, where clusters with similarities of their siblings bigger than 0.4 will be left as whole and those with smaller will be split.

3.7.8. Document modifiers

Document modifiers, the implementations of *IDocumentModifierPlugin* interface, are used to add additional information into the document. The interface defines methods for beginning and ending of document parts, which are called during the rendering of the document in Document window. The method receives document part and current context and should return generated HTML, which would be inserted directly into the document.

The examples of implemented modifiers:

- *KMostSimilarChaptersModifier* – adds table with K most similar chapters before each chapter. It also shows the position of each chapter and its similarity. The K value can be configured in Configuration.

- *NextChapterInClusterModifier* – for each chapter it adds a link to previous and next chapter within the same cluster. It also shows the label of the cluster and the similarity between chapters.

- *AllSimilarModifier* – for each chapter this modifier shows a list of all chapters with similarity equal or greater than configured threshold. The list is divided in two tables: previous chapters and next chapters. The threshold value is configurable in Configuration.

- *NextSimilarModifier* – this modifier shows a link to next and previous chapter with similarity equal or greater than the configured threshold. The threshold value could be different than the threshold mentioned in previous modifier.

4. Usage example analysis

This section provides an example of application usage and compares some of the visualization results with respect to granularity level. The example is made on book *The Underground City* from Jules Verne (available on enclosed CD in directory *TestDocuments*), which consists of 19 chapters, 1043 paragraphs and 2330 sentences. All tests were run and measured on common desktop PC with 1.66 GHz dual core processor. The sample summarization results were exported and are attached on CD in *UsageExampleResults* directory.

4.1. Basic visualization tools

After importing the document into application and generating document terms and vectors, using *DocumentTermsVisualizer* we could see, that document contained 44051 different words, which resulted in 4010 terms in document term table. As could be seen in Table 1, which shows first 12 terms ordered by their word count, the first 11 are stop words and the first non stop term is on twelve position. The total number of stop word terms was 536 and they stopped 25526 word occurrences.

Id	Term	Word count	Stop word	Inverse frequency
22	the	3010	True	0,1183334
32	of	1544	True	0,2369451
20	to	1197	True	0,2681921
125	and	900	True	0,31455
30	a	746	True	0,3626023
61	was	629	True	0,4528527
105	in	628	True	0,4243079
158	that	480	True	0,4923612
150	it	423	True	0,559308
112	his	389	True	0,6330943
93	had	377	True	0,6439521
47	harry	354	False	0,5162734

Table 1: First 12 terms in document ordered by word count

With use of ChapterConsistencyVisualizer, ParagraphConsistencyVisualizer and ChapterConsistencyVisualizer we could get similarity of document part with its parents. The average values are presented in Table 2 and this values confirms that the document contains a lot of short paragraphs with 2.23 sentences in average, because the sentences are very similar to theirs paragraphs.

	Similarity with paragraph	Similarity with chapter	Similarity with document
Chapters	x	x	0,5614398
Paragraphs	x	0,1919763	0,1199073
Sentences	0,6228683	0,1346669	0,0840155

Table 2: Similarity of document parts with theirs parents

4.2. Document summarization

Two approaches to document summary generating were implemented. First approach uses Affinity graph described in section 2.3., while the second one is based on document clustering as described in section 2.4..

The Affinity graph method is in its standard form based on sentences. For the *The Underground City* document, the computation of similarity matrix on its 2330 sentences took about 8.3 seconds and the computation of Affinity rank and sentence order required additional 3.4 seconds, which is 11.7 seconds in total. The final selection of sentences into summary text was done in almost no time. This is one of the advantages of this method, because once we have the computed data, we could quickly get several summarizations of different lengths.

If we take Affinity graph method and apply it on paragraphs, the computation time become lower compared to Affinity graph method applied on sentences due to lower number of fragments taken into account – here 1043 paragraphs instead of 2330 sentences. In this case it takes 2.47 seconds to compute Affinity graph and further 0.69 seconds on Affinity rank computation – 3.16 seconds in total.

Clustering method could be used on chapters, paragraphs or sentences, which gives us three parallel cluster structures of the document content. Therefore it may be interesting to compare them.

The application so far implements two real clustering algorithms mentioned before in section 2.1. and Table 3 shows the time in seconds spent on their computation. It is clear that with deeper granularity the computing time raises rapidly and with Centroid HAC even more.

Algorithm used	On chapters	On paragraphs	On sentences
K-means, k=10	0.78 sec.	5.88 sec.	14.28 sec.
Centroid HAC	1.17 sec.	34.22 sec.	135.6 sec.

Table 3: Time spent on computation of clusters in seconds

The Table 4 shows times spent on generating summarization of about 10 sentences long from previously computed clusters. In it we can see, that both algorithms spent approximately the same time on generating summary from clusters over different granularity. The required sentence length was set to 0 characters,

which means that the algorithm will not look at the length and will choose most similar sentences.

Algorithm	Chapters	Paragraphs	Sentences
K-means, k=10	0.45 sec.	0.53 sec.	0.51 sec.
Centroid HAC	0.7 sec.	0.8 sec.	0.78 sec.

Table 4: Time spent on generating document summaries

Using HAC for document summarization seems to be not so suitable. The main reason is its slowness in comparison to K-means and Affinity graph. The second problem is, that near to the root is has clusters which are not much mutually similar and often they contain only one item. If we choose some of this items into the summary, they would not tell much about the rest of the content.

On the other side, the K-means clustering seem to take reasonable amount of time for computation. When we compare the summaries of 100 sentence length, the K-means method generated pretty good results[2] on both sentences and paragraphs.

The Affinity graph method is a bit faster and the results on sentences[3] also looks pretty well and some of the selected parts were the same as selected by those K-means.

The Affinity graph method on paragraphs is the fastest one, but selects whole paragraphs instead of just sentences[4]. Selecting whole paragraph into the summary may provide more continuous piece of information, not just one sentence out of the context.

4.3. Additional document navigation

When we look at the similarity matrix at Figure 5, we could see that there are some parts, which are very similar – almost white – far away from the diagonal line.

By use of implemented document modifiers, we could add additional navigation into the document, which would follow us to related parts in other sections of the document.

2 Results could be found in files *Verne -03- Summary from sentences - kmeans100 len0.html* and *Verne -03- Summary from paragraphs - kmeans100 len0.html* attached on CD in *UsageExampleResults* directory.

3 Result of Affinity graph on sentences is could be found in file *Verne -02- Sentence summary using diversity - 100 sentences.html*. More detailed result in context of the whole document could be found in file *Verne -05- Sentence summary using diversity - 100 sentences info.html*.

4 The Affinity graph results on paragraph and sentences could be found in files *Verne -02- Paragraph summary using diversity - 50 paragraphs.html* and *Verne -02- Sentence summary using diversity - 100 sentences.html*.

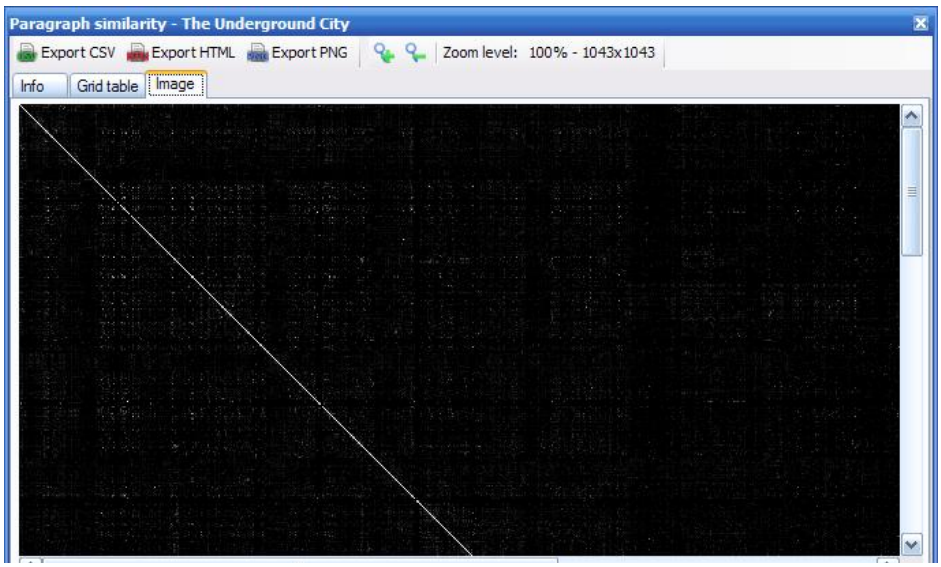


Figure 5: Part of paragraph similarity matrix of The Underground City

Figure 6 shows document with added modifiers *K most similar chapters*, *Next and previous chapter in cluster* and *Next and previous similar chapter*.

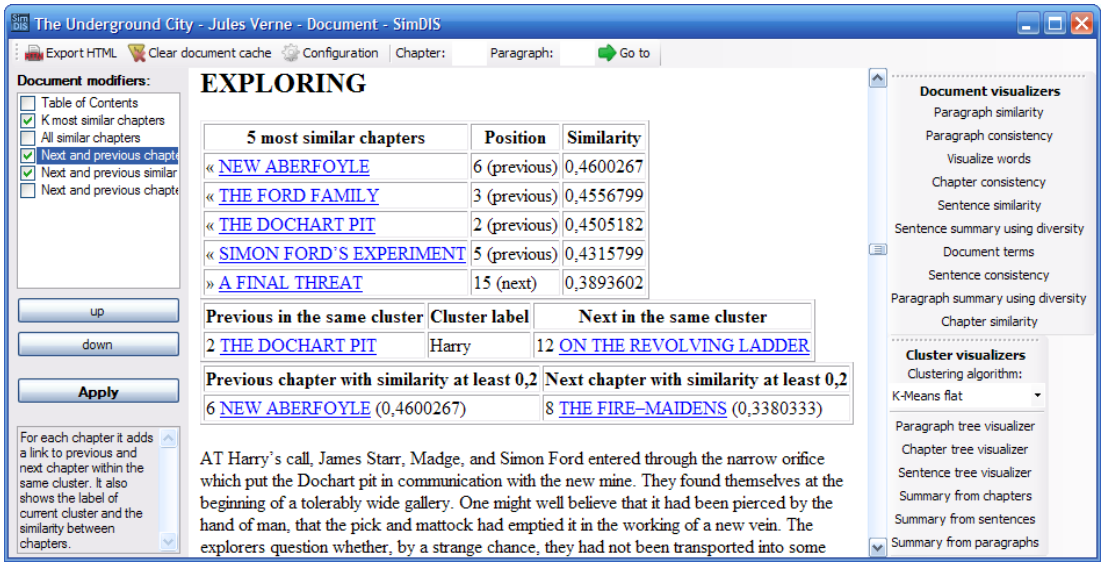


Figure 6: Document with additional navigation

By use of this links, one could quickly go through most similar chapters. Because this application is designed primarily for experimental use, the similarity and positional data are also shown.

Links generated using clusters are not the same as links based on similarity. Cluster structure depends on clustering algorithm and selected thresholds. The similarity links have advantage in possibility to find link specifically for each part. The clusters group parts together and it could happen that some part would be on the edge of the cluster and its content could be more similar to a part from different cluster.

4.4. Comparison of cluster labelers

In section 2.1.6. were proposed three possible approaches of cluster labeling:

- *Most common words* – choosing words with the highest weights in cluster centroid vector
- *Most similar sentence* – choosing a sentence which is the most similar to a cluster centroid vector
- *Modified information gain* – choosing words with the highest value of Modified information gain

Using the K-means algorithm with K=10 on sentences in document mentioned before, we get ten clusters with following labels:

Most common words	**Most similar sentence**	**Modified information gain**
Starr, James, Mr, said, Yes	"To you and to me, Mr. Starr.	Starr, Mr, James, servant, relative
new, work, Aberfoyle, vein, possible	Like these caves, New Aberfoyle was not the work of men, but the work of the Creator.	new, work, diligently, sonorous, centuries
old, engineer, good, man, overman	"Good–by, Simon," said the engineer.	lad, appetite, signal, doesn, supper
pit, Dochart, shaft, did, Yarrow	"What! In the Dochart pit?"	pit, Dochart, vain, concluded, warm
Harry, say, Nell, sea, cried	I say, do look, Harry!" cried Jack.	terrified, hoax, hidden, fierce, specimen
t, know, feet, shall, great	"Well, I don't know.	shook, earnest, jealous, pickax, obstinate
day, coal, Town, lived, life	Afterwards you will be free, if you wish it, to continue your life in the coal mine, like old Simon, and Madge, and Harry.	farm, Melrose, convey, gliding, labor
Simon, Ford, Loch, left, Sir	Come along," said Simon Ford.	Loch, Katrine, guest, English, irrevocable
opening, years, Harry, able, air	"You're right there, Jack Ryan.	endure, mixture, hope, toy, invented
little, bed, coal, sandstone, schist	As the waters were contained in no bed, and were spread over every part of the globe, they rushed where they liked, tearing from the scarcely–formed rocks material with which to compose schists, sandstones, and limestones.	forests, strata, Rob, sandstone, pressure

Table 5: Comparison of cluster labeling methods

As it could be seen on Table 5, the labels created by Most common words are not so bad at all. This is because the weights in vectors were normalized using *tf-idf* as described in section 2.1.2., where the words which are occurring in more parts across the document receive a penalty in favor of words with local occurrence.

Most similar sentence may create a label with better sense, but it could happen that the selected sentence would not cover all important words from the cluster. For example, if the parts within the cluster are dissimilar to each other.

The Modified information gain approach seems to add into labels words which are more unique to the cluster, compared to the Most common words approach. Therefore the Modified information gain is generating more unique cluster labels and seems to be a bit better.

4.5. Other documents

The application was also tested on other documents. The books attached on CD[5] in Czech language had very similar results, but some plugins could not be used – e.g. Thesaurus or White list filtering – because they currently contains only English dictionary.

When using more technical documents – e.g. this diploma thesis[6] – the results could in some cases depend on the structure of the text. For example, document containing a lot of tables, formulas or images could have a lot of scattered text, which could influence algorithms.

5 Some examples of test documents are located in *TestDocuments* directory on attached CD.

6 A version possible to use in application could be found in file *diplomova prace test.html* in *TestDocuments* directory.

5. Conclusion

5.1. Project contribution

This work took a closer look at a possibilities of non-standard applying of vector-space model and clustering techniques on individual parts within a single document. The analysis of results has shown, that a transformation from document collection to intra-document analysis needs only a minor adjustment and the theory and algorithms then can bring useful additional information about the inner structure and consistency of document.

Implemented application with built-in visualizing tools provides and easy way for testing and analyzing the content of documents in interactive form. This could help to measure the suitability of used techniques. The data could be analyzed within the application itself or could be exported for further processing.

5.2. Application results

Using list of terms, it was possible to verify the document indexation and by the frequencies of the terms, to identify possible candidates for addition to stop list.

The image presentations of similarity matrices show that real documents contain often similar areas located far away each from other. The application generates additional easy navigation between them. The same visualization could be used by authors as a hint for better document structuring by describing similar topics closer together.

Comparing K-mean and HAC algorithms, the HAC was able to detect parts which are dissimilar from the rest of the content and so it could better detect the topics of the document. However, when generating summary the K-mean was giving better results, because it was able to select more relevant sentences, which were better reflecting the overall content of document.

Generated summaries showed, that for longer texts the summaries from sentence level clusters and paragraph level clusters were very similar in quality. The application also showed, that it is possible to use Affinity graph method to use paragraphs instead of sentences. The number of paragraphs is lower than the number of sentences and therefore the computation over paragraphs would take significantly less time.

5.3. Possible future work

The possible ways of visualization are practically unlimited. Therefore, it is probable that more of them would be added in future. It could be also suitable to implement more clustering algorithms.

The implemented plugins for word stemming provides only basic approach. To improve the token normalization, some advanced lemmatizers or stemmers dependent on document language could be involved into the process.

The supported HTML format proven itself as sufficient for testing purposes. There are a lot of tools, which allows converting other types of documents to it. On the other hand, there are a lot of free electronic books available targeted for electronic book readers. So it could be handy to add native support for some of them.

6. User documentation

6.1. System requirements

This application designed to run under Microsoft Windows XP SP3, Windows Vista SP1 and Windows 7 in 32bit version. There is also native 64bit build of the application available on the enclosed CD. It could overcome the 4GB memory barrier from the 32bit operating systems and is limited only by the amount of memory available on the computer. Using this version it is possible to process larger documents as well.

To run the application, Microsoft .Net Framework 2.0 or later is required to be installed.

6.2. Installation

The installation of 32 bit version could be done via provided setup.exe wizard or simply by extracting SimDIS.zip package. The 64 bit version could be installed by extracting SimDIS.x64.zip package. All files could be found on attached CD.

No other steps are required to run the application.

6.3. Working with the application

The application is started by executing SimDIS.exe file.

6.3.1. Starting a project

If the application is launched for a first time, it prompts for a workspace directory, where it could save project data. Selected directory needs to be writable, otherwise the application would prompt for the directory again. Once the directory is selected, the welcome screen is shown to user (Figure 7). On this screen, the user can create and start a new project or to load (Figure 8) data from existing projects. The projects are represented as a directories in *projects* directory.

Figure 7: Welcome screen

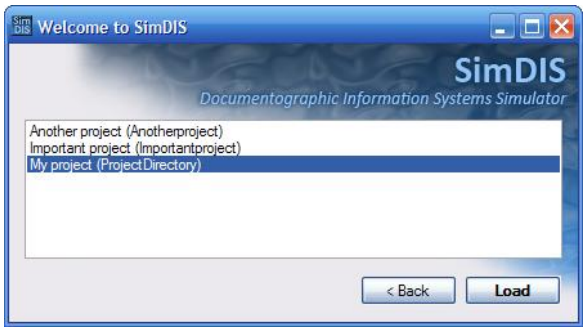


Figure 8: Project loading selection

When creating the project, the name of directory is generated automatically from the project name. In could be later renamed or copied using common tools for browsing file system in Windows.

6.3.2. Document list

After the project is loaded, the main window with list of documents is shown (Figure 9). The list contains author and title of documents and also number of chapters, paragraphs and sentences. The user can select one or more documents within the list by the mouse.

Here it is possible to manage documents, stored within the project:

- Open selected document – opens the first document that is selected in list. The document could be also opened by double clicking on it in the list.
- Import document – opens file dialog and imports one or more documents.
- Delete selected document – deletes all documents selected in list.

Under section Tools in menu, the user could find following:

- Clear project cache – deletes all cache files withing the project. Already computed values will have to be computed again when they are necessary instead of taken them from the cache.

- Configuration – opens a Configuration window, which is described later and allows to configure normalizer, plugins, properties and choosing similarity measure and cluster labeling method.

- Test normalizers – opens a window, which allows to enter the words into the text box on left side – each on single line – and then by use of selected normalizer it shows the output in the text box on the right side. For example, entering a list of stop words, applying stemmer on it and taking the result as a new list of stop words.

- About SimDIS – information about the application.

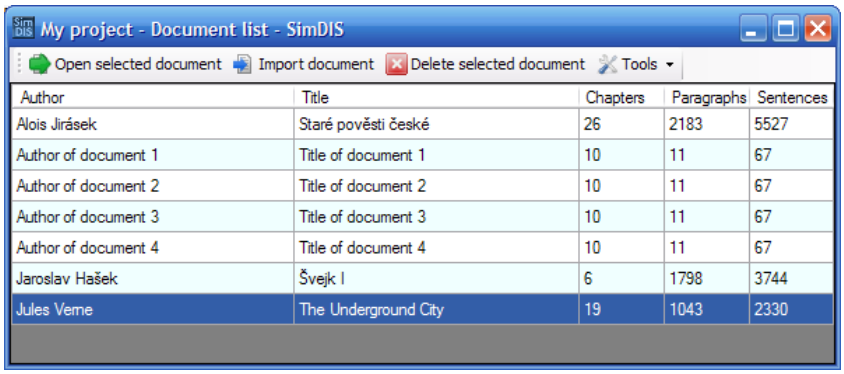


Figure 9: Document list window illustration

6.3.3. Configuration window

Configuration window allow user to configure the behavior of application and plugins. Figure 10 illustrates the configuration of text normalizers. The check box next to each of them allows to set whether the normalizer should be used or not. The order of chained execution is the same as the order in the list and could be changed by selecting wanted normalizer and by clicking on buttons up or down. The selected line would move up or down in the list. The same principle is used on text filters.

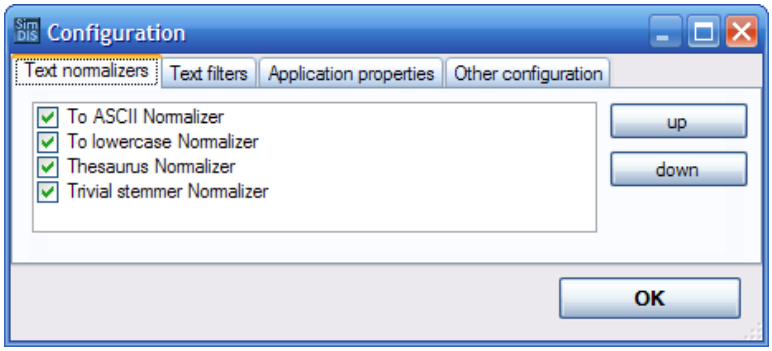


Figure 10: Configuration window - Text normalizers

The optimal order of Text normalizers is: To ASCII Normalizer, To lowercase Normalizer, Thesaurus Normalizer and Trivial stemmer Normalizer. The Thesaurus contains currently only basic set of English synonyms, so it would not have desired effect in other languages.

The order of Text filters is not very important, because they could only mark stop words. But the White List Filter contains currently only basic set of English words and therefore is should be mostly disabled and used only with a whitelist suitable for the domain of processed document. Otherwise it could mark word as a stop word incorrectly. In worst case it could mark all words as stop words.

Figure 11 show the configuration window in the Application properties tab. Here the user could configure properties for plugins and the application. The left column show a read-only description of the property and the value in right column could be edited by double clicking at it.

On the last tab – Other configuration – user can choose which similarity measure and cluster labeling are to be used.

After all desired changes in configuration are finished, the changes could be saved by clicking at OK button or by closing the window.

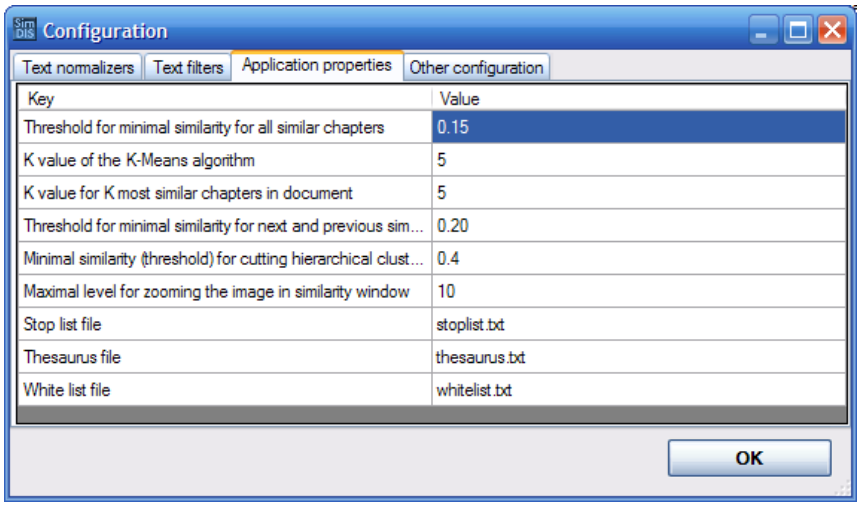


Figure 11: Configuration window - Application properties

6.3.4. Document window

By opening document from document list, the Document window is shown (Figure 12). It shows the text of the document with applied modifiers, allows to choose modifiers and to run visualization on the document.

Document window offers following actions in top menu:

- Export HTML – opens save file dialog and then exports current document with applied modifiers into selected file.

- Clear document cache – deletes all cache files, which belongs to current document.
- Configuration – opens the Configuration window described before, where user could change configuration.
- Go to – it is possible to enter the number of chapter and paragraph and by clicking on Go to button, the document text would be scrolled to this document part.

The control panel from document modification is at the left side of window, where it is possible to:

- Enable or disable the modifier by checking or unchecking appropriate check box next to desired modifier.
- Change the order by selecting one modifier and clicking on up or down button. The selected item will be moved in the list which is in the same order as they would be applied on document.
- By selecting modifier from list, its description is shown in the bottom-left.
- By clicking on Apply button, the order current order and status of modifiers is saved and the text of the document in the center part would be regenerated.

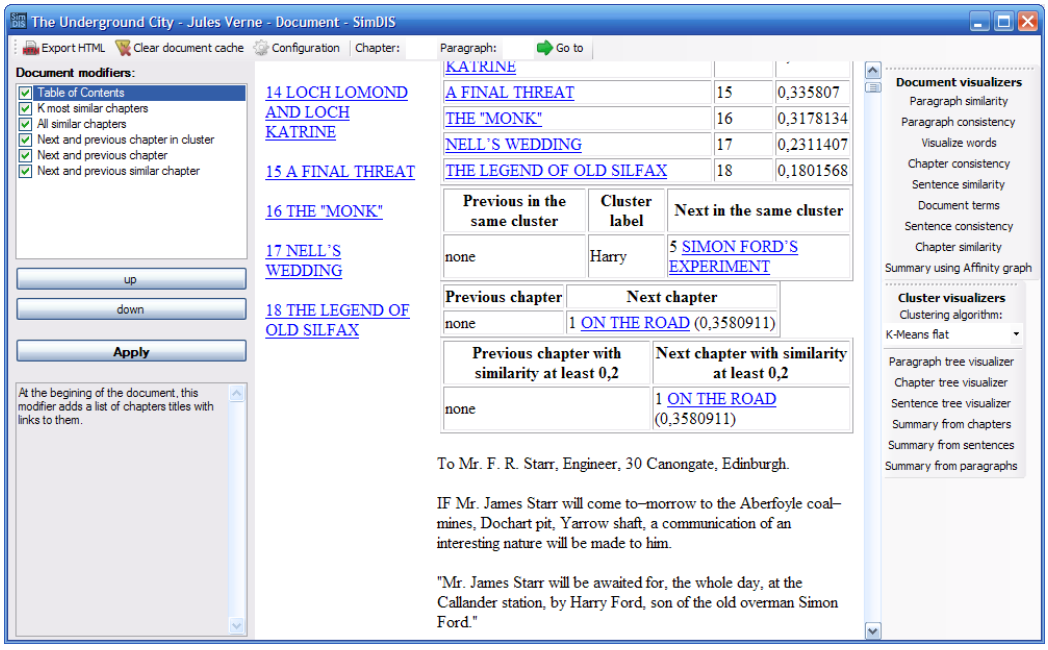


Figure 12: Document window illustration

The visualizers plugins are automatically registered and are offered to user in side menu, from which he can execute them. When visualization is executed, it usually opens its own specific window.

The visualizations of clusters and some document modifiers uses currently selected clustering algorithm.

6.3.5. Chapter, paragraph and sentence similarity

All three visualizers use the same window, which display the similarity as table, as in Figure 13, or as and image, as in Figure 14.

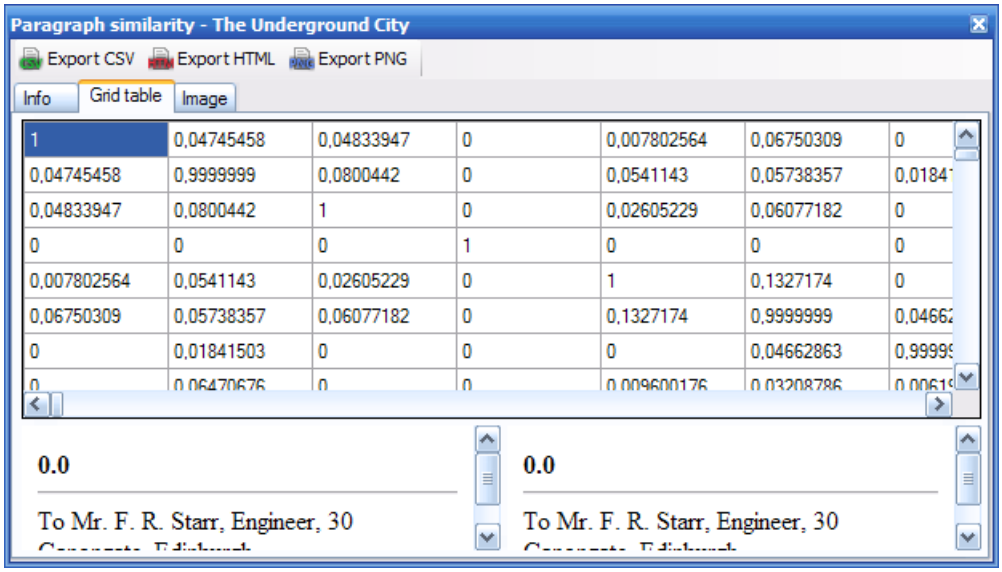


Figure 13: Similarity window showing grid view

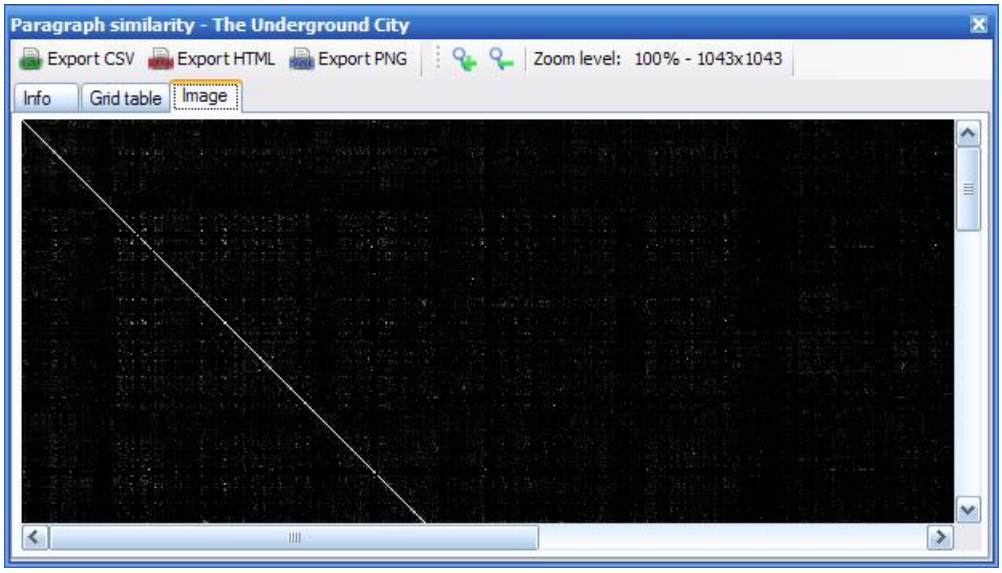


Figure 14: Similarity window showing image

The user could play with it and event export it into CSV, HTML or PNG for further processing in external programs.

6.3.6. Words and terms visualization

There are two direct visualizations for showing term vocabulary. First – Visualize words – shows whole text in a HTML window (Figure 15) and adds hidden "+" to every document part. After clicking on it, the term table and vector of corresponding document part is shown. The "+" changes to "–" and when clicked, it hides back the table.

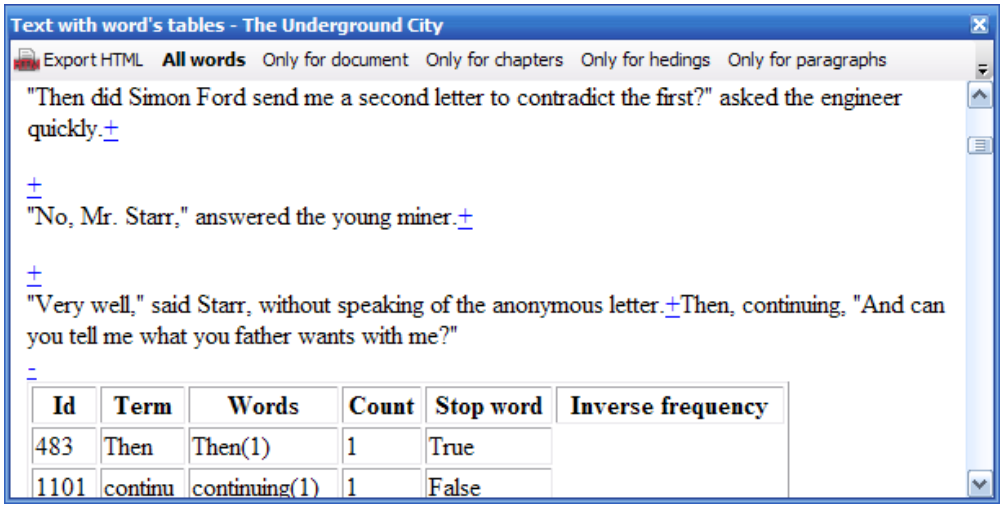


Figure 15: HTML word visualization window

By selecting in top menu, the user could affect to which parts the "+" are displayed.

The second visualization just shows global dictionary and allows to export it.

6.3.7. Working with clusters

The clustering algorithm, which is to be used for creating clusters could be selected in menu in document window as shown on Figure 16.

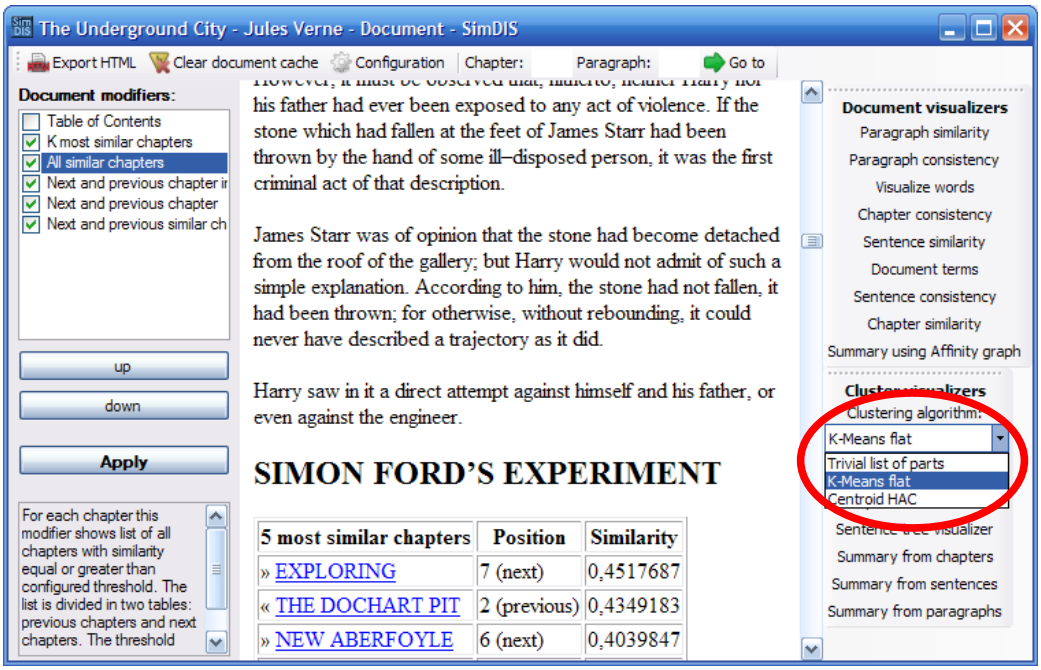


Figure 16: Selecting clustering algorithm

Then the cluster could be seen clicking on Chapter tree, Sentence tree or Summary tree visualizers. The cluster tree window would be opened (Figure 17).

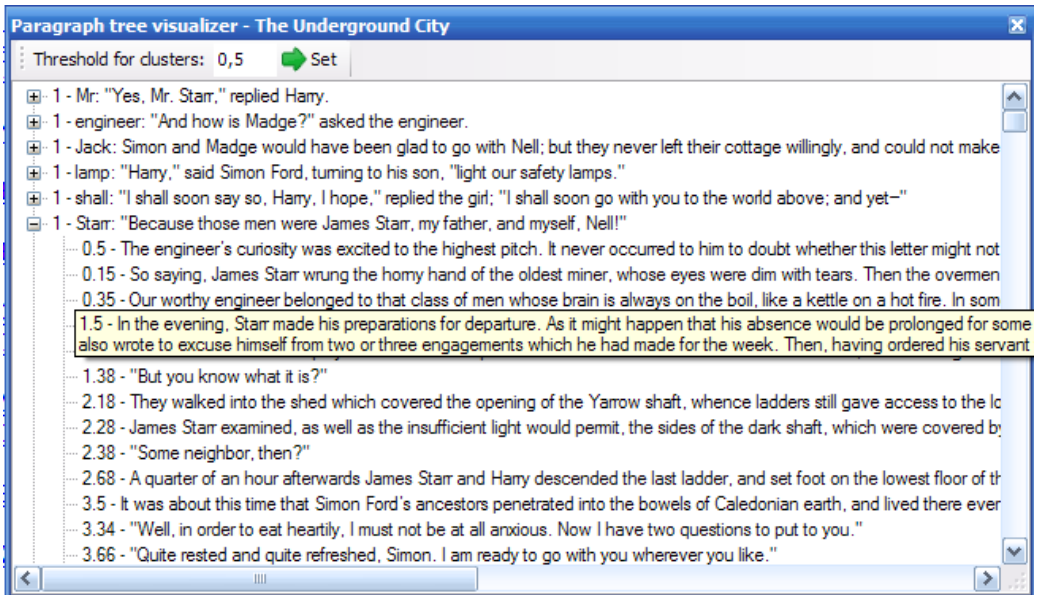


Figure 17: Cluster tree visualizer window

In this window, user could browse through the clusters. Each node represents one cluster, where first is the similarity of its sub-clusters or 1 in for leaf and then its label. The leaf cluster contains member document parts, where the first number represents their location in document.

By hovering on document part, a title with full text is shown.

The user could also specify the threshold value in top menu, which is used to cut through the hierarchical clusters.

6.3.8. Generating text summary

For automated generating of text summary, there are four possibilities accessible from the document window:

- Summary from chapters – clusters on chapter level are used
- Summary from paragraphs – clusters on paragraph level are used
- Summary from sentences – clusters on sentence level are used
- Sentence summary using diversity – using affinity graph, information richness and diversity of sentences to generate summarization
- Paragraph summary using diversity – using affinity graph summarization method on paragraphs, instead of sentences

Figure 18 shows example of summary window. After pressing *Generate* button on top, it generates the text and also displays time spent of computation. The time spent on creating clusters shows the time spent on getting the clusters and in case of loading from cache, it could be significantly lower, than real time needed to compute them.

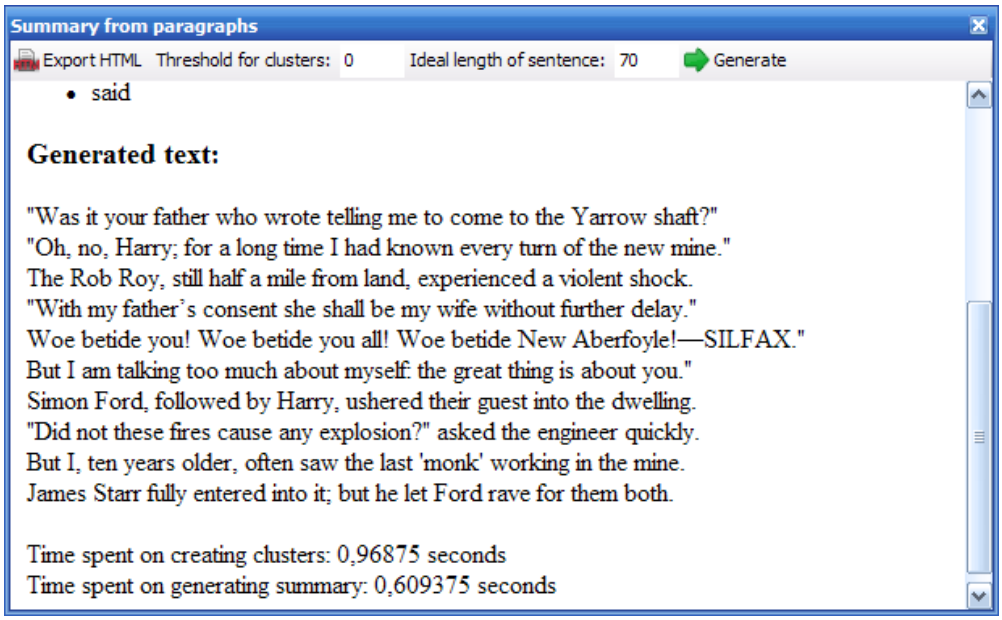


Figure 18: Summary generating window

On this window, the user could set threshold and sentence length value. The threshold is used to cut through the hierarchical clusters. The ideal sentence length is used as a parameter for text generating, which is closely described in 2.4..

7. Programmer documentation

7.1. System requirements

The application requires Microsoft .Net Framework [15] in version 2.0 or later to be installed and Microsoft Visual Studio [19] 2008 or later to open the project solution.

For compiling the application, the System.Data.SQLite library [18] and Subversion [20] are required. The Subversion is used to automatically generate assembly version and it could be omitted by removing subwcrev command from pre-build events of SimDIS project.

For creating and compiling custom plugins it is not required to have the source and compile the whole application.

7.2. Compiling the whole application

The source code of application is contained within SimDIS solution of Microsoft Visual Studio. To open it, run SimDIS.sln in root of the solution.

The solution consists of several projects, of which the main are SimDIS and SimDIS.PluginInterface. Other projects are containers for plugins.

The whole application could be build by choosing *Build* > *Build solution* from the top menu.

7.3. Custom plugin creation

The types of plugins are described in section 3.5.. This section describes creation of custom plugin without the need of the whole application using Microsoft Visual Studio 2008.

At first, the project (and solution) must be created. It is a Class library project and its name must end with Plugin. Then the reference to SimDIS.PluginInterface class library must be added as shown on Figure 19 and set the Copy Local property to false. Usually the System.Windows.Forms is also required to show visualization.

Next step it to rename Class1 to more suitable name, for example MyCustomVisualizer. It is important, that it ends correctly as described in section 3.5.. In this example I have chosen visualizer plugin, so the class must implement SimDIS.PluginInterface.IVisualizerPlugin interface.

The IVisualizerPlugin interface consists of getPluginName() method, which simply returns name of the plugin, getPluginProperties() method, which should return the list of plugin-defined properties and visualize() method, which receives information about document and runs visualization.

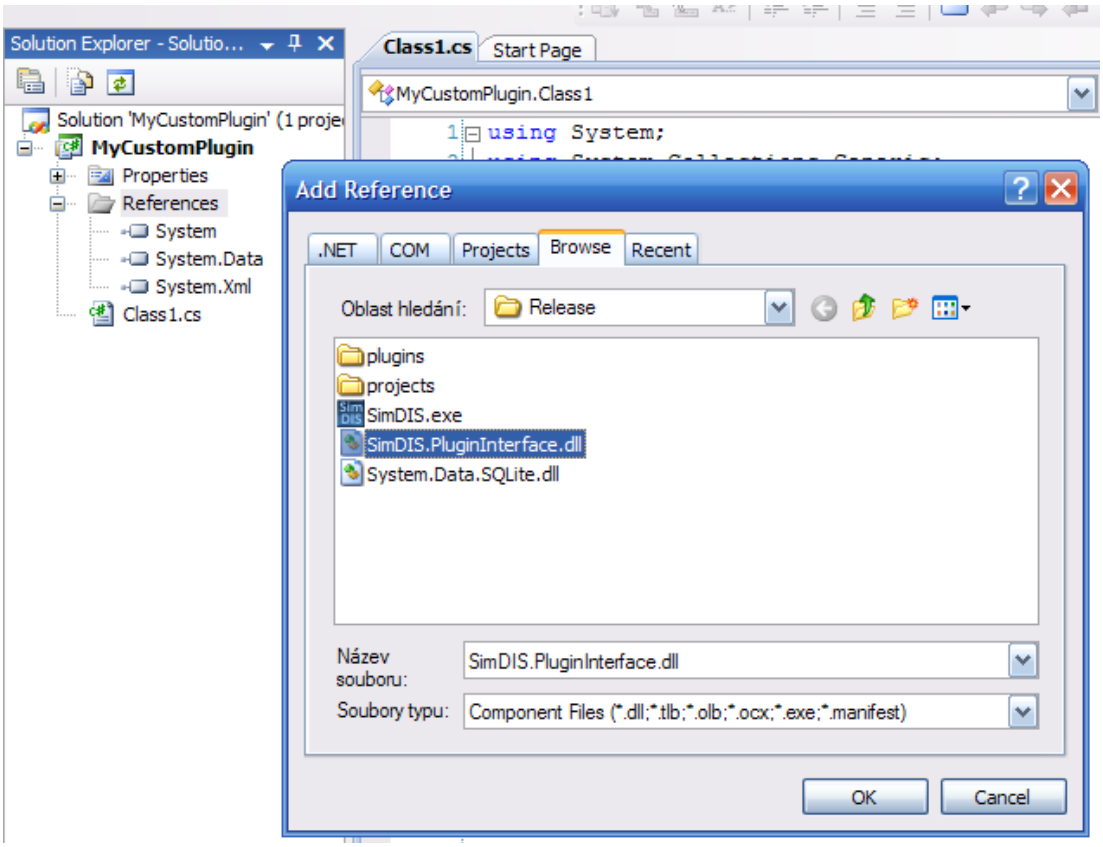


Figure 19: Adding SimDIS.PluginInterface

A simple implementation of visualization is in Figure 20, where it shows the author and the title of provided document in System.Windows.Forms.MessageBox.

```
namespace MyCustomPlugin
{
	public class MyCustomVisualizer : IVisualizerPlugin
	{
		#region IVisualizerPlugin Members

		public void visualize(IDocumentInfo documentInfo,
			IWin32Window owner)
		{
			MessageBox.Show(documentInfo.getDocument().Author
				+ ": " + documentInfo.getDocument().Title);
		}

		#endregion

		#region IPluginBase Members

		public string getPluginName()
		{
			return "My custom";
		}

		public IList<ConfigurationProperty> getPluginProperties()
		{
			return null;
		}

		#endregion
	}
}
```

Figure 20: Implementation of simple visualization plugin

When building the plugin, it is important to chose right target platform. The SimDIS application is by default compiled for x86, so the plugins should be compiled also for x86 or Any CPU.

To add and run compiled DLL of plugin, it is only needed to copy in plugins directory of SimDIS application. The application will automatically detect and register the plugin.

Bibliography

[1] Manning Ch. D., Raghavan P., Schütze H.: *An Introduction to Information Retrieval*, Cambridge University Press, Cambridge, England, 2009

[2] Gan Guojun, Chaoqun Ma, Jianhong Wu: *Data Clustering: Theory, Algorithms, and Applications*, ASA-SIAM Series on Statistics and Applied Probability, SIAM, Philadelphia, ASA, Alexandria, VA, 2007

[3] Aone Ch., Larsen B.: *Fast and Effective Text Mining Using Linear-time Document Clustering*, KDD-99 San Diego CA USA, 1999

[4] Geraci F.: *Fast clustering for web information retrieval, PhD Thesis*, Facolta di Ingegneria, University of Siena, Siena, Italy, 2008

[5] Charikar M. S.: *Similarity Estimation Techniques from Rounding Algorithms*, STOC-02, 34th Annual ACM Symposium on the Theory of Computing, Montreal, CA, 2002

[6] Lloyd S.P.: *Least squares quantization in PCM*, Technical report, Bell Laboratories, 1957, 1957

[7] Hinneburg A., Keim D.: *Optimal grid-clustering: Towards breaking the curse ofdimensionality in high-dimensional clustering.*, In Proceedings of the 25th internationalconference on very large data bases (VLDB ’99), San Francisco, 1999

[8] Lance G. N., Williams W. T.: *A general theory of classificatory sorting strategies 1. Hierarchical systems*, The Computer Journal, 1967

[9] Cover T. M., Thomas J. A.: *Elements of information theory*, John Wiley & Sons, New York, USA, 1991

[10] Pedersen J. O., Yang Y.: *A comparative study on feature selection in text categorization*, ICML-97, 14th International Conference on Machine Learning, Nashville, USA, 1997

[11] Yang J., Xiaojun W.: *Improved Affinity Graph Based Multi-Document Summarization*, Proceedings of the Human Language Technology Conference of the North American Chapter of the ACL, New York, USA, 2006

[12] Chen Z., Fan W., Ji L., Li H., Liu Y., Ma W. Y., Xi W., Zhang B.: *Improving Web Search Results Using Affinity Graph*, SIGIR’05, Salvador, Brazil, 2005

[13] Bossard A.: *Generating Update Summaries : Using an Unsupervized Clustering Algorithm to Cluster Sentences*, Laboratoire d'Informatique de Paris-Nord, 2011

[14] World Wide Web Consortium (W3C): *W3C HTML*, http://www.w3.org/html/

[15] Microsoft Corporation: *Microsoft .Net Framework*, http://www.microsoft.com/net/

[16] Microsoft Corporation: *Windows Homepage*, http://windows.microsoft.com/

[17] SQLite development team: *SQLite Home Page*, http://www.sqlite.org/

[18] Simpson R., SQLite development team: *System.Data.SQLite*, http://sqlite.phxsoftware.com/

[19] Microsoft Corporation: *Microsoft Visual Studio*, http://www.microsoft.com/visualstudio/

[20] Apache Software Foundation: *Subversion*, http://subversion.apache.org/

CD contents

Enclosed CD contains:

- dp.pdf – A PDF version of this diploma thesis.
- Binary – Compiled version of the application
- Documentation – Generated documentation of the application
- Source – Source code of the application
- TestDocuments – A sample documents compatible with SimDIS application
- UsageExampleResults – Exported results from example usage

List of tables