Bachelor Thesis in

Textual Similarity

Abdulla Ali

Kongens Lyngby

IMM-BSc-2011-19

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-BSc: ISSN 2011-19

Abstract

This thesis deals with one of Information Retrieval's big interest: Textual Similarity. Textual Similarity is a process where two texts are compared to find the Similarity between them.

This thesis goes into depth with subjects as Similarity and its definition, Language and Words, then algorithms and other tools used to find Textual Similarity.

Two algorithms, Optimal String Alignment Distance and Cosine Similarity, are then implemented with and without Stemmer and Stop Word removal algorithms. Many tests with the structures of the sentences being different from each other have been run. These have then been compared to the algorithm options to decide which one works best. These have also been compared with the writer's expectations and human testers.

The result was that while Optimal String Alignment Distance had its advantages in some fields, Cosine Similarity had its own in others. It was also concluded that both run better with a stemmer and stop word removal added to them than without.

Abstrakt

Denne opgave handler om en af Information Retrieval's store interesse: Tekstsammenligning. Tekstsammenligning er en process hvor to tekster sammenlignings for at finde deres lighed.

Denne opgave will gå i dybden med emner som lighed og dennes definition, sprog og ord,så algoritmer og andre værktøjer for at finde tekstsammenligning.

To algoritmer, Optimal String Alignment Distance og Cosine Similarity, er så blevet implementeret med og uden Stemmer og Stop Word removal algoritmer. Der er lavet mange test hvor strukturen af sætninger har været forskillig fra hinanden. Disse har så været sammenlignet med algoritme valgmulighederne for at beslutte hvilken er best. De har også været sammelignet med undertegnet's egne forventninger og de menneskelige testere.

Resultatet var at selvom String Alignment Distance havde sine fordele i nogle områder så havde Cosine Similarity sine i andre. Det blev også konkluderet at begge kører bedre med en stemmer og stop word removal end uden.

Preface

This thesis was prepared at Informatics Mathematical Modeling, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the BSc. Degree in engineering.

This thesis deals with different aspects of textual similarity and the processes that play into it. It touches subjects related to similarity such as language with mainly focus on the sentence structure, algorithms and other tools that can be used to study and measure textual similarity. For this purpose an application has been made and tests of many different kinds have been made to try to see the capabilities of the chosen tools.

This thesis consists of a summery in Danish and English, 10 chapters on the subject of textual similarity, a bibliography and an appendix at the end of the thesis.

The source code, application, javadoc, raw data, test-sheets and other relevant content can be found on the attached CD.

Lyngby, June 2011
Abdulla Ali

Acknowledgements

I thank God for everything and nothing. Without God I would be lost. Without God I would not be here. Without God…

I thank my grandparents and know that even after their deaths they live strongly in my heart.

I thank my parents for teaching me about the things good parents do. It is only because of them that I turned out to be a person with a strong personality, not giving up not matter what and always following my heart in whatever I do.

I thank Robin Sharp for "forcing" me by believing that I could finish this thesis less than 3 months while I thought of it as impossible. Again it has been proven to me that nothing is impossible if you try your best. I thank him for his inputs since they made me view the subject from different perspectives.

Thank you all!

Contents

CHAPTER 1

Introduction

Similarity is a broad and abstract topic. Every time Similarity is mentioned, the question pops up: “What kind of Similarity?” The topic is quite big in the Information Retrieval field and lately it has become quite the hype again. People are making search engines, plagiarism programs, optimizing search algorithms, finding out how to specify the searches better and faster, and where to focus whether it be images, sound or strings.

This paper takes part in one kind of Similarity too; Similarity between two texts aka Textual Similarity.

Starting with a look into Similarity in general where different kinds of similarities are presented and Similarity itself is defined. Hereafter the focus moves to Language: The building structure from the small morphemes to long blocks of texts. How does words interact with each other and how complex they are is also taken up here. Before ending the topic a tool called Wordnet is presented, one that seems to be a great online lexical reference system which can be used to find much information about a word, e.g. synonyms.

Then leaving the linguistic and moving to the mathematical, or rather the computer science domain. Algorithms used for Textual Similarity are presented, each with their advantages and disadvantages. Other tools for Textual Similarity are mentioned and a few ways to calculate the Similarity Score.

Out from the analysis a decision is made to decide what and what not to use. Some questions are answered and there have been more emphasize on some of the tools. Design and structure of the made tool are shown and how the classes for the tool work.

The next few topics after these are tests of every kind. Structural test to check the inside of the program, function test to check the outside and usability test to see how handy the tool is in other people’s opinion. The tests do not stop here since the real tests, the tests for Similarity, starts. First the tests methods are described, then the expectations, later the results and to end this paper, a discussion of the results and similarity.

CHAPTER 2

Analysis

2.1 Similarity in General

Similarity: Comparison of commonality between different objects

Similarity is a complex concept which has been widely discussed in the linguistic, philosophical and information theory communities.

Similarity has been a subject of great interest in human history since a long time ago. Even before computers were made, humans have been interested in finding similarity in everything. Each and every field of study provides their own definition of what similarity is. In Psychology similarity "**…**refers to the psychological nearness or proximity of two mental representations."[1] while in music it's "…a certain similarity between two or more musical fragments"[2] and in geometry "Two geometrical objects are called similar if they both have the same shape."[3] Definitions for similarity are different for every field but what keeps going in each of them is the use of one big field to prove the similarity: Math.

Math is used to calculate similarity where it dominates the field. After the start of two new fields in last century, Information Theory and Computer Science, the topic of similarity has not become smaller at all. Instead by using the computer it has been easier to find out how similar two or more things are to each other.

This chance caused by these fields where mathematics can be applied and the easiness to make the calculations fast, have made humans invent algorithms to make new ways to calculate similarity easier, faster and as correct as possible.

The intuitive concepts of similarity should be about the same for mainly everyone.

1) Object A's and B's similarity is related to their commonality. The more they have in common, the bigger their similarity is.

2) Object A's and B's similarity is related to their differences. The more they have in uncommon, the lesser their similarity is.
3) Object A's and B's similarity has maximum when both are identical to each other, no matter how much commonality they share.

So similarity is the ratio between the amount of information in the commonality and the amount of information in the description of A and B. If commonality of A and B is known, their similarity would tell how much more information is needed to determine what A and B are. [4]

Similarity can be separated into different categories to specify what kind of similarity is needed and for what e.g. image similarity or sound similarity. By specifying the object which similarity has to be found on, the algorithms get filtered down to the important and related ones.

One of the specified categories is textual similarity; the idea of taking two or more strings and comparing them with each other to find out how similar they are. This category is so interesting and filled with opportunism that many university professors and students have studied far into this category and written many a paper on the textual similarity. Reason for this is that humans are different; they got different ideas and different thresholds when it comes to how similar something is based on a scale.

- The idea of which algorithm is more precise than the rest.
- The different opinions about when the objects are similar and when they are talking about the same topic.
- The question is if similarity can be found just by using some algorithms or if there is a need to include grammar and protocols of language too.

Textual similarity which is a subcategory of Information Retrieval is quite close to its brother the search engine since both takes a query and find forth the similar texts for the query. Where it is expected from search engines to find the respective query's document of relevance and rank them, it is expected from the text similarity to find out how similar the query is to the documents. Both overlap a lot but are still a bit different.

2.2 Language – Word and Sentence Structure

Language: A way to communicate with others.

Language is the communication of thoughts and feelings through a system of arbitrary signals, such as voice sounds, gestures or written symbols.

Language is how humans communicate and convey their feelings or thoughts. Language can be separated into pieces for further analysis of what it is that makes, the language, language; sounds, words and syntax.

Though language is not just pieces, it will suffice since there is no need to go too deep into the structure of language in this paper.

These pieces can be put into a hierarchical structure which gives a basic structure of how language could be set up in linguistic units.

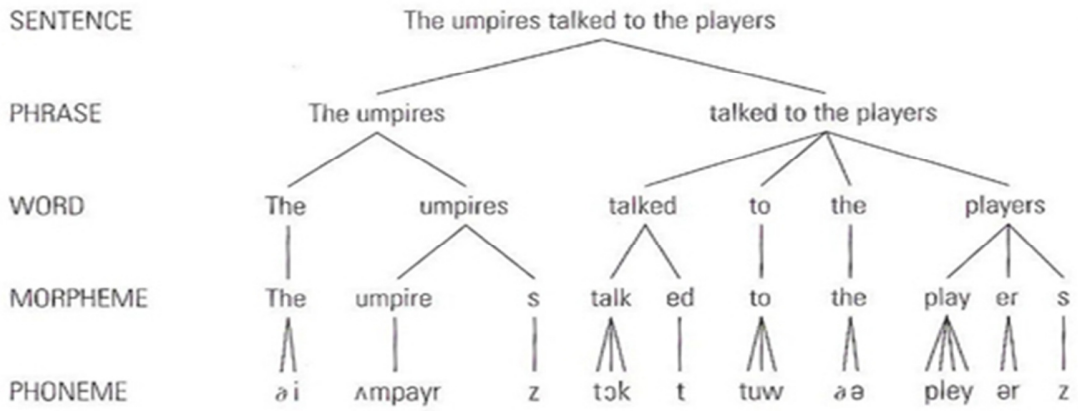


Figure 1: The Hierarchy of Linguistic Units

First, there are the thoughts or the feelings that needs to be conveyed. The thoughts and feelings can be split into sentences. These are coherent sequences of words used to show what the person is trying to say. The sentences can be cut into phrases which are groups of words that form constituents and functions as single units in the sentences' syntax. Words come from splitting the phrases and are made of morphemes. Morphemes are small units that carry meaning. There are two kinds of morphemes; free and bound. The free morphemes are units that can stand alone and give meaning, as in being ideas, actions or objects as in "play" or "umpire" in Figure 1. The bound morphemes are units that bind themselves to the free morphemes to add information which is crucial for the interpretation. Free bounds in Figure 1 could be "s" or "ed". At the bottom of this structure are the phonemes. The phonemes, the smallest unit, are the sound the person has to make to distinguish the words in language. They are mostly represented by letters in the English alphabet but can also be represented by other symbols, e.g. the phoneme for "the" is looking strange.

So each level of these makes the language; phonemes put together makes morphemes, which put together makes words. These words assembled with each other after a certain structure makes phrase which put together makes up sentences used to express a person's thoughts or feelings.

Since this paper does not focus on the spoken language but on the written language, there is no need to go deeper into phonemes. The morphemes will be continued on later, under the Stemmer section. Word will be analyzed now where after the focus will be moved to syntax where analyses about sentences and phrases are included.

Words make up the language. Words are used to name events, objects and actions in the world of language. Words are said to be "refers" in the sense they refer to something. Like the word "paper" refers to this paper right now, while the word "laughing" refers to an action done by a person. Knowing the word means knowing how to use it and in which situations. It makes more "sense" saying: "I was jumping of the horse" than saying "I was jumping off the bites." Words can also refer to things that do not exist like "utopia" where the meaning is

understandable but it does not exist. These words have no problem in being used. More about words will be discussed under Wordnet section.

Sequences of words in phrases and sentences are governed by rules called syntax. Syntax provides an important function because they specify the relationship between the words in a sentence. Meaning it tells which words are related to which words. Having only words will not make sense (dog, cat, bit) since they will not tell what has happened with whom and where. Inserting the words by the rules of syntax (the cat bit the dog) will make it appear how the situation, in the sentence described, is. Syntax is not meaning of the sentence since sentence without meaning can easily be made by going by the rules of syntax. The poem "Jabberwocky" by Lewis Carrol is a perfect example of this where sentences like "He left it dead, and with its head/He went galumphing back." are sentences that are perfectly language (English) but gibberish. Linguists say that syntax and the organization of sentences can be shown easily by the phrase structure rules. One such rule says that a sentence (S) consists of a noun phrase (NP) and a verb phrase (VP):

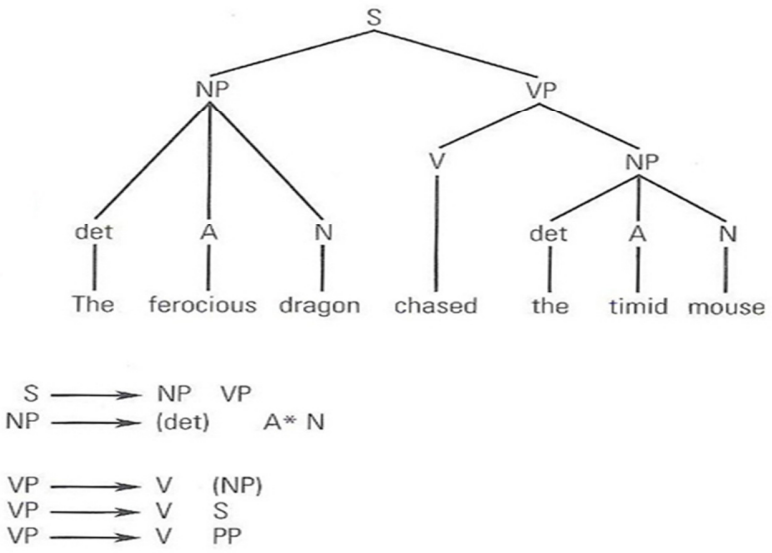


Figure 2: Phrase Structure with rules

This kind of rules splits the sentence into two parts; the performer (NP) and the information about the performer (VP). The verb phrase (P) can be rewritten so VP -> V NP, where the V indicates the action described by the sentences and NP specifies the target of that action.

Another structure that is also important is the D-structure. This structure reflects the speaker's intention in writing a sentence by including if there should be emphasis on some words, if it is a question, or a comment etc.

This is how the phrase structure becomes a guide for understanding sentences and why speakers reject most sentences that they do not find into this structure.

Phrase structure is a great thing but how to figure it out. Where do the words in the sentence get placed? It is far from all the sentences that follow S -> NP V NP structure, many are much variable of these and thus their structure is far more difficult to figure out. To figure it out the sentences need to get parsed. To parse means to resolve into its elements. In this case finding out where each word's syntactic role is. The best way would be to wait until the sentence ends and then figure out the structure of the sentence. This takes time and understanding the text will be slow but the errors made by the reader or hearer would be less since all information was given before processing it. Humans do not work this way. They start their interpretation of the sentence as soon as they start reading or hearing the words. This approach is faster and more efficient but can sometimes lead to errors. Thus the reader or hearer has to reanalyze the sentence when the error causing sentences occur. Still it is the favorite method of humans compared to the first method. Examples of those reanalyzing sentences can be:

- The secretary applauded for his efforts was soon promoted.
- The old man the ships.

Notice that these sentences need to be reread to find out what they really say. First reading stops, when the reader finds out that something is wrong and then reread them to make sense in her mind.

Examples like these are called garden-path sentences. They are called this because the reader is led to one interpretation (*led down one path*) only to find out that it was the wrong path. The interpretation gets rejected to form a new.

Parsing is guided by two big players too: Syntax and Semantics.

Syntax: The reader is "looking" for active sentences than passive sentences and thus thinks that every sentence is about the performer performing something. One thing that helps understanding this kind of thinking is by looking at the bound morphemes since they indicates what kind of word it is that is being read; "kicked" is a verb called "kick" because of "ed" morpheme.

Semantics: The reader assumes and is thus guided by semantic factors. These factors make the understanding of active sentences easier than passive sentences. If looked on the secretary sentence from before. The reader assumes that the "secretary" is a women thus the "his" must refer to someone else thus semantic factors has already played in the parsing process for the reader.

Another thing important to the sentence that is being read is the context. Sentences alone can be confusing if the context is not known. Knowing the context makes it much easier to imagine the picture in the mind.

- Put the apple on the towel into the box.

Just reading this sentence is quite hard to understand. Though if someone gives these hints: "There is a table with a box. Two apples of which one is on the table and one is on the towel." will make the sentences much clearer and understandable.

2.2.1 Wordnet

As said before words are used to name events, objects and actions in the world of language. The thing forgotten here is that one word can be referring to more than one event, object or actions. One fine example could be “house” which can be a noun “a house” or a verb “to house”. Looking into the noun “house” it will soon become obvious that “house” as noun can refer to many nouns since the noun can be used in different meanings; “She waited till the whole house were asleep before sneaking out.” or “There we saw a house”. Both sentences use the word “house” as a noun but in different relations. Thus words can be referred to many things and the reader should have experienced these before knowing about how they work. Of course knowing the word as one of its meaning helps quite a lot understand the other meanings of the word.

This way the words can be nouns, verbs, adverbs and adjectives but it does not stop here. There is one thing that should not be forgotten when it comes to words, they can be idioms or slangs too. Here the word gets a different meaning than the one it had originally. A “trick” can for example mean “theft” when slangs come into the picture.

On the other hand, looking on the meanings of the words then one word can easily be replaced with another that means the same, those are called synonyms. Each meaning of the word has its own set of synonyms. If “trick” is taken as the “prank” or “joke” meaning then the synonyms can be “gag”, ”jest”, ”put-on”, ”shenanigan” or “tomfoolery” while “trick” as “deceit” can as example take the synonyms “cheat”, “bluff”, “deception”, “plot”, “illusion”, “swindle”, “trap” and so on.

Before moving on to other of words capabilities, it is best to present a tool that has made to deal with these things and also many more of a words abilities and relations. The tool is called Wordnet and is an online lexical reference system developed by George A. Miller.[5] Many others have contributed to this system since its beginning and still being worked on. The system takes English (can be developed to other languages too) nouns, verbs, adverbs and adjectives and organize them into synonym sets, each representing an underlying lexical concept. The sets get linked via different relations. The goal is to produce a combination of a thesaurus and a dictionary which can be used and support text analysis in information retrieval, artificial intelligence and many more other text processing fields.

Wordnet separates words into four categories; nouns, verbs, adverbs and adjectives. The reason is that these four groups follow their own rule for grammar, meaning they got their own morphemes for each group, as introduced before. Each of the synsets is composed of synonyms of words, meaning different meaning of a word in different groups. Every synset have a short defining sentence attached to explain the synsets meaning. Under each of the four categories which the Wordnet groups into synsets, the synsets connects to other synsets. This is done by semantic relations where each word synsets connections relay on the type of the word that is being worked with. They made of:

- Nouns
    - Hypernym
    - Hyponym

    - Coordinate term
    - Holonym
    - Meronym
- Verbs
    - Hypernym
    - Troponym
    - Entailment
    - Coordinate terms
- Adjectives
    - Related nouns
    - Similar to
    - Particle of verb
- Adverbs
    - Root adjectives

Hypernym and Hyponym: Y is a hypernym of X if every X is a Y or Y is a hyponym of X if every Y is a X. This is also called superordination/subordination, superset/subset or in computer science language a is-a relation. This means if taken as an example that if the word is "leaf" which is a hyponym of "tree". The "tree" itself is the hypernym of "leaf" but the hyponym of "plant". Thus a hierarchical structure is made where the hyponym inherits all the features of the more generic concept. It also adds at minimum one feature that differentiates it from its hypernym and from other hyponyms of that hypernym.

Coordinate term: Y is a coordinate term of X if X and Y share a hypernym. This is quite easy to understand. If two hyponyms share the same hypernym they are coordinate terms. "Crow" is a coordinate term of a "dove" and "dove is a coordinate term of a "crow", why? Because both have the same hypernym called "bird". Another example could be "to scream" and "to whisper" since both uses the hypernym "To talk".

Holonym and Meronym: Y is a holonym of X if X is part of Y or Y is a meronym of X if Y is a part of X. This refers to things being part of others. Thinking in examples; "wheel" is a meronym of "car" and "car" is holonym of "wheel". This means that holonym and meronym are each other's opposite. This is also called whole-part or Has-a relation. While holonym defines the relationship between a word denoting the whole and a word denoting its part, meronym defines the relationship between a word denoting the part and a word denoting it's whole.

Troponym: The verb Y is a troponym of verb X if the activity Y is doing X in some manner. Troponym tells what kind of *manner* is done in, an example shows it clearly: "To limp is a troponym of to walk".

Entailment: The verb Y is entailed by X if by doing X you must be doing Y. Said in another way: "Something is happening to make something happening" The entailment can be changed out with the word "thus". "He was killed" thus "He is dead" or "He is snoring" thus "He is sleeping". Going back to the term entailment the only thing that gets changed is "He is snoring" entails "He is sleeping" meaning he is sleeping which is causing the snoring.

Two other important functions for Wordnet are antonyms and morphology relations:

Antonym: As known is the opposite of the word in question. Though it does not mean that the opposite of the word is always true. This means that the antonym of a given word X is sometimes not-X, but not always. Looking at examples; to “rise” doesn’t mean that something or someone just had a “fall” and is “rising” from that. The relation is still there both words are antonyms.

Morphology: Working with full words is quite hard for a program. To make it easier, it works with its root aka the word’s stem. After all having the word “trees” or “tree” is the same thing and having the word “talked” or “talk” has the same meaning. More about this under Stemmer section.

As shown the language of humans are quite complicated and filled with connections. It is complex and a huge thread network where many words are connected to each other on some way or another; either as synonym or as antonyms, in syntax or in semantics or in structures or parsing. No matter what, the language is an enormous field where many people have found various algorithms to analyze it. The next section will give some insight into.

2.3 Algorithms used for Textual Similarity

As mentioned before, many have discussed which algorithms to use and when to use them, when it comes to textual similarity. After filtering from the bulk of algorithms to specify similarity, then to information retrieval and last to textual similarity, there is still many left. Still they can be put into each their boxes where three of those areas for text similarity stick out more than the rest;

- Vector space model
- Edit distance
- Ontology based

The problem does not only consist in how much they stick out but also how good they are for a given task. Some are good enough to run a search on similarity on shorts strings, while the long ones give a poor result, others the reverse. Some run fast on short strings while others the reverse. Some are best to be as accurate as possible while not caring about their slowness, while others run fast and keep being accurate as a second.

2.3.1 Vector Space Model

Also called term vector model is a math model for representing text documents as vectors of identifiers, like terms or tokens. Of course the term depends on what is being compared but are normally single words, keywords, phrases or sentences. A document collection consisting of Y documents, indexed by Z terms can be shown in a Y x Z matrix M. Thus the queries and the documents are representing as vectors and every dimension corresponds to a separate term hence every element in the matrix M is a weight for the term Z in the document Y. If the term appears in the document, the value in the matrix for the specific element changes, otherwise not. Using this together with the assumption of document similarities theory, the similarity between two documents can be calculated by comparing the difference between the angels of each document vector and the query vector. This calculation ends up given a result ranging from 0 to 1, both numbers included. If the document and

the query vector are orthogonal the result is 0 and there will be no match aka the query term does not exist in the document. If the result is 1 it means that both the vectors are equal to each other. To compute these values many ways have been found, some more known than others.

2.3.1.1 Cosine Similarity

The standard way of quantifying the similarity between two documents x_1 and x_2 is to computer the cosine similarity of their vector representations and measure the cosine of the angle between the vectors. [6]

$$Cosine\ Similarity(x_1 + x_2) = \frac{V(x_1) \cdot V(x_2)}{|V(x_1)||V(x_2)|}$$

While the denominator is the product of the vectors', $V(x_1)\ and\ V(x_2)$, Euclidean lengths, the numerator shows the dot product which is the inner product of the vectors. The effect of the denominator is to length-normalize the vectors $V(x_1)\ and\ V(x_2)$ to unit vectors: $v(x_1) = \frac{V(x_1)}{|V(x_1)|} and\ v(x_2) = \frac{V(x_2)}{|V(x_2)|}$.

The result of the angle will show the result. If the angle is 0 between the document vectors then the cosine function is 1 and both documents are the same. If the angel is any other value then the cosine function will be less than 1. Does the angle reach -1 then the documents are completely different. Thus this way by calculating the cosine angle between the vectors of x_1 and x_2 decides if the vectors are pointing in the same direction or not.

2.3.1.2 Term Frequency-Inverse Document Frequency

The term frequency–inverse document frequency or TF-IDF weight is a way to give words that appear in a text a weight. This way some words becomes heavier than other words and affect the similarity score to be more precise in another words it makes some words more important that other words in the text that are being compared.[7]

To understand TF-IDF it would be best to split up and take one piece at a time; TF and IDF.

TF or term frequency, as it says the frequency a term appears in a given document. Usually the term is normalized so a bias towards longer documents, who would have a higher term frequency for the specific terms, to get a measure of the important term t within the document d. There are many kinds of term frequency variants like sublinear TF scaling or maximum TF normalization but this paper will work with the basic and most known one. The term frequency can then be defined as:

$$\mathrm{TF}_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}}$$

$n_{i,j}$ is frequency of the term *d* in the document *d*. The dominator is the sum of the frequency of all the terms appearing in the document d meaning the size of the document *d*, aka $|d|$.

There is a big problem with only having TF alone since all terms are considered to be equally important when it comes to their relevance on a query. Some terms will have no or very little discriminating power in determining

relevance. As an example if the documents are on birds, the documents will have the term bird many times in the documents. This will emphasize documents which happen to use the word “bird” more frequently, without looking on the weight of more important terms. If the important terms are “yellow” “bird” then “yellow” will be not get any heavy weight if it occurs rarely but it will still be a good term to differentiate the relevant documents from the non-relevant. That is why there is needed a mechanism for attenuating the effect of terms that occur too often in the collection of documents to be meaning for determination of relevance.

That is the reason for the inverse document frequency factor to be added to the TF equation. It will diminish the weight of terms that happens to occurs too frequently in the documents and increases the weight of terms that only occurs rarely.

$$IDF_i = log \frac{|D|}{1 + |\{j : t_i \in d_i\}|}$$

|D| is the number of documents in the document set.

$|\{j : t_i \in d_i\}|$ is the number of documents where the term t_i appears. Though if the term t_i is not found in the document, it will equal in a zero division thus it is common to add 1.

So the TF-IDF equation will be:

TF-IDF = $\mathrm{TF}_{i,j} * IDF_i = \frac{n_{i,j}}{\sum_k n_{i,j}} * log \frac{|D|}{1+|\{j:t_i \in d_i\}|}$

TF-IDF will filter out the frequent terms when it gets a high weight term. This happens when a term has a high frequency in a document but a low frequency in all the documents put together. The TF-IDF value will be greater than 0 if IDF is greater than 1.

2.3.1.3 Jaccard Similarity Coefficient, Jaccard Distance, Tanimoto Coefficient.

All three are the same but extended a little from the Jaccard Similarity Coefficient.[8] The Jaccard Similarity Coefficient calculates the similarity between sets and is defined as:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Simple calculation: The size of intersection of A and B divided by the size of the union of A and B.

Jaccard Distance which instead of similarity measures dissimilarity between can be found by subtracting Jaccard Similarity Coefficient from 1:

$$JD(A,B) = 1 - J(A,B) \; or \; JD(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

Tanimoto Coefficient is an “extended” Jaccard Similarity Coefficient and Cosine Similarity put together. Meaning by having Cosine Similarity yield Jaccard Similarity Coefficient, Tanimoto Coefficient can be represented:

$$T(A,B) = \frac{A \cdot B}{\left||A|\right|^2 + \left||B|\right|^2 - A \cdot B}$$

This is in case of binary attributes which are a special case of discrete attributes that only have 2 values, normally 0 or 1. Tanimoto Coefficient runs from -1/3 to 1 unlike the Cosine Similarity that runs from -1 to 1.

2.3.1.4 Euclidean Distance or L2 Distance

Euclidean distance aka L2 distance aka Euclidean norm is another similarity measure in the vector space model. Euclidean distance is so common that the talking about Distance is nearly always referred to this distance. This similarity measure differentiates from the other vector space model similarity measures by not judged from the angle like the rest but instead of the direct Euclidean distance between the vector inputs. To simplify it if there are 2 points then Euclidean distance calculates the distance between those points instead of the direction of the vectors like it is done in Cosine Similarity. Euclidean distance examines the root of square differences between the coordinated of the pairs in the vectors x and y:

$$|x \to y| = \sqrt{\sum_{i=1}^{M} (x_i - y_i)^2}$$

2.3.1.5 Advantages of the Vector Space Model

- Simplicity since it is based on a linear algebra model.
- Ability to incorporate term weights; any kind of term weight can be added.
- Can measure similarity between almost everything; query and document, document and document, query and query, sentence and sentences and so on.
- Partial matching is allowed.
- Ranking of documents compared to their relevance is allowed.

2.3.1.6 Disadvantages of the Vector Space Model

- Assumed independence relationship among the terms.
- Long documents make similarity measuring hard; Vectors with small scalar products and a high dimensionality.
- Semantic sensitivity; the terms uses a query to find similarity in the document or the document should not contain spelling mistakes. Same with different vocabularies. This will result in bad results since it will result in poor inner product.
- Weighting is not formal but automatic.
- The order of the terms in the document becomes lost and plays no role in vector space representations.

2.3.2 Edit Distance

This group of algorithms works in another way than vector space model. It takes two strings of characters and then calculates the edit distance between them. The edit distance will be the numbers of actions the algorithm has to take to transform both of the sentences into each other. The actions are many and vary from substitution to elimination to transposon to deletion and so on. Each definitions and number of actions depends on the algorithm being chosen to calculate the edit distance.

Edit Distance be solved by using dynamic programming. Take the problems and split it into subproblems. Solve each subproblem just once, remember the results for later, and put then the results together to get an optimal solution for the whole problem. Following this recipe and computing the results in an optimal solution in a bottom-up approach for solving problems of increasing size makes this a perfect example of dynamic programming. Another way of doing it could be by taking all the actions under Edit Distance and going through them all but that would take too much time.

The idea is to use as minimum actions as possible to calculate the transformation for both strings to be the same. Each action equals the cost 1, so Edit Distance does not calculate the similarity but the distance in cost measures meaning metric.

Each algorithm has their own advantage and disadvantage and will therefore be mentioned in their description.

2.3.2.1 Hamming Distance

The Hamming Distance takes two strings of equal length and calculates the number of positions at the places where the characters are different.[9] It calculates the least number of substitutions needed to make one string into another. If looked at some examples it becomes clear that Hamming only use substitution and nothing else:

100000 and 101010 = 2

“Jumbo” and “Dudmo” = 3

Hamming is mostly used in error-correcting codes in the fields like telecommunication, cryptography and coding theory. Since it is all a matter of finding out where the differences in different data are. If taken telecommunication then it is only a matter if the number is 0 or 1 and thus easily calculated how different the two data packages of equal lengths are.

The advantage: It is quite easy to implement and work with since it only subtracts any differences while adding 1 to each total cost for each subtraction.

The disadvantage: The length of both strings should be the same otherwise Hamming cannot be used as algorithm on the two strings.

2.3.2.2 Jaro Winkler Distance

The Jaro Winkler Distance is an extended version of the Jaro Distance and is given by:

$$d_{jw} = d_j + (lp(1 - d_j))$$

d_j is the Jaro Distance for two strings s1 and s2, given by:

$$d_j = \frac{1}{3} + \frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m - t}{m}$$

Where m is the number of matching characters and t is the number of transpositions.[10]

Every character in one of the strings gets compared to all of its matching characters in the other string. In Jaro Distance only the number of transpositions is counted compared to before in Hamming where only the number of substation was counted.

l is the prefix length for both strings to a maximum of 4 characters.

p is the prefix weight, a constant scaling factor for how much the score can be adjusted upwards. *p*'s standard value is set to 0.1 but can be set higher. Though to not get a distance larger than one, *p* should not be set over 0.25.

So the Jaro Winkler Distance takes the matching characters and the transposition to find the Jaro distance and uses the Winkler modification to score higher if the initial start is the same for both strings. The score ranges from 0 to 1 where 0 is no similarity and 1 is exact the same strings.

The advantage: Using this makes it easier to find duplicates in strings since the only thing the Jaro Winkler Distance does is transpose the letters in a string thus used in duplicate detection.

The disadvantage: It works best with short strings such as person names. It only transposes making it only find duplication and not much else.

2.3.2.3 Levenshtein Distance

Just as Distance is referred to Euclidean Distance, Edit Distance is normally referred to Levensthein Distance. [11] This Distance algorithm is just like the Hamming Distance but still much more than that. Levenshtein Distance allows not only substitution but also insertion and elimination of single characters. The two next examples show how Levenshtein Distance works and the cost is calculated.

Comparing "stopping" and "topping":

1. **s**topping -> topping (elimination of "s")

The cost was 1.

Comparing "hump" and "jumping"

1. hump -> jump (substitution of "j" for "h")
2. jump -> jumpi (inserting of "i" at the end)
3. jumpi -> jumpin (inserting of "n" at the end)

4. jumpin -> jumping (inserting of “g” at the end)

The cost was 4.

So the cost is just like the other edit distances, always one for the actions allowed in the specific distance algorithm.

Levenshtein is perfect to run for finding the similarity on small strings, to small string and a big string where the editing difference is a small number to be expected.

If the cost is 0 then the strings are exact the same if the cost is equal to the length of the longs string then the strings are totally opposite.

The advantage: Can do what Hamming Distance can do and elimination and insertion. Levenshtein Distance is not restricted by the strings needing to have the same length. The cost can at most be the length of the longest string.

The disadvantage: Running Levenshtein on two long strings results in a long time and a big cost that is proportional to the product of the two string lengths.

There are two variants of Levenshtein that are important to look at: The Optimal String Alignment Algorithm and the Damerau-Levenshtein.

Both the algorithms can do the same thing as Levenshtein except they can accomplish transposition too. The difference between Optimal String Alignment and Damerau-Levenshtein is that Optimal String Alignment Algorithm only completes transposition under the condition that no substring is edited more than once whereas Damerau-Levenshtein is not restricted by such a thing. That is also why Optimal String Alignment is sometimes called the Restricted Edit Distance.

2.3.3 Fuzzy Logic

Fuzzy logic deals with the truth value. Fuzzy logic, a set that does not have sharp boundaries, takes the Boolean 0 and 1 value and breaks them into pieces so instead of only two values being able; the truth value can range from 0 to 1. So while probabilistic logic and fuzzy logic both can range from 0 to 1, fuzzy logic plays with the degrees of how true something is hence the fuzzy part. As guessed fuzzy logic is not quite objective as probabilistic logic since the part of decided how true something is quite subjective. There is no universal truth for the truth values except 0 and 1 where 0 is 0% true (or not true) and 1 is 100% true.

“Today is a sunny day.”

Trying to set the truth values according to fuzzy logic could be done in this way:

If there are no clouds in the sky then today is a 100% a sunny day. If there are a few clouds in the sky then it may be an 80% sunny day. If there are many clouds in the sky then maybe it is a 50% sunny day. If it is raining a little and the sky is filled with clouds then it is a 20% sunny day. Now if it is snowing and the sky is filled with dark clouds then it is probably 0% sunny day aka not a sunny day at all.

Notice how the words “maybe” and “probably” sneak in the sentences above. It is because the fuzzy in deciding how much of a sunny day it is aka the truth values are being decided by the subject instead of math where one thing is proven to be that fact. This fuzzy area where the truth values can change is what fuzzy logic is about.

Notice also that the explanations for how sunny a day it is, is following a specific control rule which is quite a heuristic rule and common to use in fuzzy logic to decide how true a value is:

If <condition> **Then** <action>

Fuzzy logic for truth value can be evaluated for a truth T as:

$$T(A \wedge B) = \min(T(A), T(B))$$

$$T(A \vee B) = \max(T(A), T(B))$$

$$T(\neg A) = 1 - T(A)$$

Fuzzy logic is being used by many these days all over the world though statisticians and some control engineers still used their Bayesian logic and 0 and 1 logic (2 valued).

The advantage: Nothing becomes universal true except 0% and 100%. Gets close to the human common sense; the human way of thinking and judgments are the lures of fuzzy logic.

The disadvantage: The truth value is subject and not objective, making the decider decide how true something is.

2.4 Stemming

Looking back to the language section there were something called morphemes. Morphemes were bound and free and made the words in the language. Focusing more on the free morphemes it can be seen that they are the ones that make the morphological variants of the words while the free morphemes are the base form of word. By removing the bound morphemes all the morphological variants of the words will be gone.

Stemming is such a technique to return the words to their stems, the base form of the word. By stemming words in information retrieval the difference forms of a certain word ends up with one base form. The stemmed word does not need to be identical to the morphological base form of the word. This will remove the morphological variants of the word that is acting as a hindrance to find the relevant results for the terms in the queries and documents. It matters not if the word is a dictionary stem form or not since all the variants of the word will be the same after being stemmed. For this reason algorithms called stemmers have been developed so that it becomes easier to process, more precise and also saving storage space for the program.[12]

There are many stemmers out at the moment and more will probably come. Though there are five that are more famous than others: Porters, Paice/Husk, Lovins, Dawsons and Krovets stemmer algorithms.

Going into depth with the advantages and disadvantages for each stemmers and their algorithm can become a paper for itself and is outside the range of this paper. Instead it will be said that each Stemmer has their own way of stemming the suffix from the words: Some focus on morphology (Krovets), some on iterative stemming

(Paice/Husk), some on other stemmers (Dawsons is based on Lovins) and others on the going through some rules and stems by focusing on the English language (Porters).

No matter which stemmer it is, they end up removing the suffix such as "–ed" or "–s" depending on the word just as a stemmer should.

2.5 Stopword

Words like "a", "the", "and" and "of" are words that appears many times in documents. For humans they have relevance in understanding what the sentence is saying but when it comes to text processing these words only hinders by making the process slow, take space and less precise since they affect the weighting of words in the weight is taken into account in the similarity algorithm.

Words like these have little information value and are called stop words. By filtering those words out before running the processing part on the data the runtime will go down, there will used less space and the similarity will be more precise.

There is no precise list for stop words since it is controlled by human input and not automated about what kind of stop words are relevant to have in the stopword list. This stopword list will then later use the words in the list to filter the words from the data.

2.6 Similarity Score

Similarity score is the measure to show how similar two set of data are to each other. The set of data can be about as in this case about two different texts. To find the similarity is to find the comparison between the two texts and grade it after a score system.

For Vector Space Model there is no really need to invent a scoring system since it is a result via the cosine function gives a range from 0 to 1, where 0 means that the vectors are orthogonal and thus totally opposite meaning both texts pieces are entirely different while 1 means that the vectors are pointing at the exact same direction and thus the same meaning both text pieces are absolutely similar aka they are the same. The numbers in between 0 and 1, shows how similar both texts are depending on the two vectors angel to each other. Taking these similarities and multiplying with 100 gives the percentage of how similar both texts are.

For Edit Distance it is a bit more different. Since the Edit Distance does not find the similarity between the texts (or the vectors since there are no vectors) but finds the metric, the distance between both texts, there are different ways to calculate similarity.

One of them could be using the distance for the string. The maximum character length of the longest texts is the maximum distance that can be found. This means that while the distance is that maximum length then the texts are absolutely different (not similar at all) while distance on 0 means that both texts are equal. Using these points to scale the method to calculate the similarity can be found by:

$$\text{Similarity}(A, B) = \frac{Distance(A, B)}{MaximumLength(A, B)}$$

Multiplying this result with 100 will give the similarity between both texts in percentage.

Another way of finding similarity could be:

$$\text{Similarity}(A, B) = \frac{1}{1 + Distance(A, B)}$$

Distance(A,B) is the distance between the texts via insertion and deletion actions to transform the texts into each other.

A measure in difference of trigrams in two texts:

$$\text{Similarity}(A, B) = \frac{1}{1 + |tri(A)| + |tri(B)| - 2\ x\ |tri(A)\ \cap\ tri(B)|}$$

tri(A) is the set of trigrams in A and tri(B) is the set of trigrams in B. Trigrams are word cut into pieces of 3 characters(tokens), if the word is “similarity” then the set of trigrams would be tri(similarity)= {sim, imi, mil, ila, lar, ari, rit, ity}.[4]

2.7 Timing

The structure of the similarity algorithms, the stemming algorithms, going through the stop word list and the structure of the program makes the program run in times that depends on what is being chosen. Of course this does not mean that the different things cannot be timed. Timing is important to find out how the algorithms work when compared to each other, to do so there are many ways to and functions to time.

Java itself allows timing too, from Java 5 a new package for timing where introduced. The package java.lang.management has methods to report CPU and user time per thread instead of only report the time via the wall clock. These times are not affected by other systems activity, making them ideal for benchmarking.

2.7.1 Wall Clock Timing:

The time is measured in the real world elapsed time the user has to wait before the task has been completed. There are two ways to measure wall clock timing: By nanoseconds and by milliseconds.

Wall clock timing is affected by the other activity on the system. On some systems the application should be put in front for not being too affected by becoming a lower priority because it is not placed in the front. Thus the timing can give poor results unless a larger number of tests are being run where the average is of those are used and the system is unloaded every time before running.

2.7.2 Single-Threaded Task Timing via CPU, System and user Time.

The managementFactory class in the java.lang.management package has static methods to return different kinds of “MXBean” objects to report timing. One of such timing MXBean is the ThreadMXBean which can report:

User time: The time spent on running the applications code.

System time: The time spent on running the OS code as the agent of the application.

These two methods in ThreadMXBean are only run in nanoseconds.

Both methods can be used to find the CPU time which is the complete CPU time spent by the application. It is found by adding User time and System time.

CHAPTER 3

Decision based on the Analysis.

The decision for choosing different algorithm and functions is quite affected by the writer's skills and knowledge in programming.

The language to implement the program will be Java since it is the only language learned, and learned while this project has been running. This does not mean that it is bad to use Java. Java is a good environment to work in since it is object-oriented and is easy to work with. The algorithms will be affected since there are languages that may work better or worse than Java when implementing these. Still implementing them, as long as both are done in the same language, they can be compared. Of course the way to implementing can affect the timing for algorithms since this is dependent on the programmer's skills.

The algorithms to be compared will be one from Space Vector Model and one from Edit Distance since having both algorithms from the same group is not as interesting as comparing two different techniques from different ends of the spectrum.

From the Vector Space Model department the most interesting algorithm to test out seems to be the classic Cosine Similarity.

The reason for not picking Euclidean Distance or Jaccard Similarity Coefficient is that the first is quite basic and literally uninteresting while the other is limited in the way that it works best on small sets or strings. The algorithms need to be worked on big sets and small sets alike.

TF-IDF did not get picked either since it does not do a lot being alone. If used with Cosine Similarity it would work quite well and give better similarity results. As said before, the decisions are affected by the programming skills of the programmer so it is decided not to use this. A lack of time is also one of the reasons to affect the rejection of trying to implement TF-IDF with Cosine Similarity.

3.1.1 Cosine Similarity (Cosine Similarity continued…)

Shown in Algorithms used for Textual Similarity section, Cosine Similarity between two texts can be computed as the cosine of the angle between the vectors.

$$Cosine\ Similarity(x_1 + x_2) = \frac{V(x_1) \cdot V(x_2)}{|V(x_1)||V(x_2)|}$$

This means to find the Cosine Similarity, one has to

- Find the dot product of the vectors $x_1\ and\ x_2$
- Determine the magnitude of vector x_1
- Determine the magnitude of vector x_2
- Multiply the magnitude of vector x_1 with magnitude of vector x_2
- Divide the found dot product of the vectors $x_1\ and\ x_2$ by the products of the magnitudes of vector x_1 and vector x_2

Before going further it is best to mention that a vector is a geometric structure with both length aka magnitude and direction. Mathematical operations like addition, multiplication, division can be done if and only if the vectors being performed on have an equal number of dimensions.

The problem with making strings into vectors is not as big as thought. A normal coordinate system is 3 dimensional with the dimensions x, y and z. The same thing can happen to with the string representing each token in it as a dimension. To make it easier to understand the tokens in this explanation with will be characters and lowercased instead of words. So ‘a’, ‘b’, ‘c’, ‘d’, ‘e’ etc are all their own dimension. That means there are 26 dimensions in total (going by English alphabet).

There are two ways of storing the letters in the vectors; one is by occurrence and the other by frequency. Storing only the occurrence of a letter in the vector it will result in a binary occurrence vector but storing the frequency in which a letter occurs in the vector, it will result in a frequency of an occurrence vector. The different can be shown with the word “banana”:

The dimensions are

a,b,c,d,e,f,g,h,l,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z

The binary occurrence vector is

1,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0

The frequency of occurrence vector is

3,1,0,0,0,0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0

The vector gets reduced quite a lot by just taking the union of the dimension of both strings the dimensions to be stored in. It means all the 0s will not be stored in the vector and the dimensions to calculate will be reduced making it easier on for the calculations.

A union as an example between the strings “banana” and “bananapie” would be result in the vector:

(a, b, e, i, n, p).

It is best to use the frequency occurrence vectors since they are more accurate when it comes to measuring similarity.

If the words are from before then the frequency occurrence would be:

banana = (3,1,0,0,2,0)

bananapie = (3,1,1,1,2,1)

The cosine function for these two strings can now be calculated:

$$The\ dot\ product = (3 * 3) + (1 * 1) + (0 * 1) + (0 * 1) + (2 * 2) + (0 * 1) = 24$$

$$The\ magnitude\ of\ banana = \sqrt{3^2 + 1^2 + 0^2 + 0^2 + 2^2 + 0^2} = \sqrt{24}$$

$$The\ magnitude\ of\ bananapie = \sqrt{3^2 + 1^2 + 1^2 + 1^2 + 2^2 + 1^2} = \sqrt{27}$$

$$The\ product\ of\ magnitudes = \sqrt{24} * \sqrt{27} = \sqrt{648}$$

$$Division\ of\ the\ dot\ product\ with\ the\ products\ of\ the\ magnitudes = \frac{24}{\sqrt{648}} = 0.94280$$

According to the Cosine Similarity “banana” and “bananapie” are 94.28% similar.

This example has been shown with characters as tokens but can easily be done with words, phrases or sentences as tokens instead. The recipe is the same. Words will be used as the tokens in this paper though.

To implement Cosine Similarity is just like doing it the mathematical way as shown. Convert the strings to vectors and take the union of those vectors to create a shared dimensional space. Apply the function to the frequency of occurrence vectors and the similarity is found.

From Edit Distance area the most interesting algorithm seems to be Levenshtein, but since Levenshtein seems to be missing transposition another variant of this will be chosen; not Damerau-Levenshtein but Optimal String Alignment aka Restricted Edit Distance.

The reason for not picking the Hamming, Jaro Winkler or the original Levenshtein Distance is the same for all three: They are limited in their actions.

Hamming Distance works best in error correcting code field and only on strings that are on equal length. This which makes it invalid in this case since the texts strings are allowed to be on unequal lengths. Jaro Winkler is quite good but it halts in the length of the string. It works best on very short strings as names and small DNA strings. Levenshtein, as told before does not include transposition as Jaro Winkler does. For it to be able to transpose and get more interesting to test out, the movement to Damerau-Levenshtein and Optimal String Alignment is needed.

3.1.2 Optimal String Alignment Algorithm (Levenshtein continued…)

Damerau-Levenshtein algorithm (DL) has the same abilities as Levenshtein but uses transposition too. The reason for the algorithm not being DL but the little different algorithm Optimal String Alignment(OSA) algorithm is that OSA does not have the extra alphabet array like DL needs which seems to be input specific and not automatic. An implementation of DL is placed in the Appendix A.

OSA works basically like Levenshtein in that sense that it uses dynamic programming too which can easily be shows in a result matrix between the two strings: “I want to program” and “I can program” where the distance is at the bottom right of the matrix: 6.

		I		w	a	n	t		t	o		p	r	o	g	r	a	m
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
I	1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
c	2	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	3	2	2	2	2	3	4	5	6	7	8	9	10	11	12	13	14	15
n	4	3	3	3	3	2	3	4	5	6	7	8	9	10	11	12	13	14
	5	4	3	4	4	3	3	3	4	5	6	7	8	9	10	11	12	13
p	6	5	4	4	5	4	4	4	4	5	6	6	7	8	9	10	11	12
r	7	6	5	5	5	5	5	5	5	5	6	7	6	7	8	9	10	11
a	8	7	6	6	6	6	6	6	6	5	6	7	7	6	7	8	9	10
g	9	8	7	7	7	7	7	7	7	6	6	7	8	7	6	7	8	9
r	10	9	8	8	8	8	8	8	8	7	7	7	7	8	7	6	7	8
a	11	10	9	9	8	9	9	9	9	8	8	8	8	8	8	7	6	7
m	12	11	10	10	9	9	10	10	10	9	9	9	9	9	9	8	7	6

The only difference between the Levenshtein distance algorithm and a few extra commands that makes the “transposition”:

```
if(i > 1 and j > 1 and str1[i] = str2[j-1] and str1[i-1] = str2[j]) then
                d[i, j] := minimum(
                                    d[i, j],
                                    d[i-2, j-2] + cost    // transposition
                                  )
```

This transposition seems legal enough to call the OSA algorithm DL algorithm but there is a different. The difference is in how they transpose. DL algorithm can handle edits where the word has been misspelled twice while the OSA algorithm cannot. It will only accept one transposition and then move on to the next substring.

Shown by example with two string: “ot” and “two”. The DL distance will result in 2 edits while OSA distance would result in 3 edits:

DL(ot, two) = ot -> to -> two = 2

OSA(ot, two) = ot -> t -> tw -> two = 3

Thus the OSA is called for doing fake transposition and not being the same as DL.

Though bringing reality into this, it is unexpected and rare to see people to have two spelling mistakes in one substring. Not to mention that in language semantics words like “ot” is quite different from “two” which would have a closer relation to “lot” than “two” in the writer’s biased opinion. This is another reason why the real DL algorithm has not been used. [13]

The implementation of this algorithm is quite easy. Both strings get computed into a matrix which is as big as string one x string two. This will be used to hold the distance values and calculate them into the matrix by seed filling the matrix. The distance between both strings will be the last value computed as soon in the above table. The idea as shown is to get the strings transformed into each other by minimum number of actions (edits).

If one string contains nothing then the distance will be the length of the other string. A pseudo code can be found in Appendix B.

Wordnet is a great concept to find out if one of the texts is using synonyms for the words in the other texts, meaning they are still saying the same thing with the difference of using different words for the same things. The thing that will be affected is quite obvious; time. Looking up in a lexical dictionary and thereafter switching the words out and then seeing if the texts are similar and keep doing this process for each word will affect the time heavily. This function will not be implemented since the programmers skills are not yet good enough to be able to implement such an advanced method to connect with the program and the algorithm. If it is easy enough then the programmer could not see the solution thus it is decided to not use it.

Instead of implementing TD-IDF and Wordnet into the application, it will contain Stemmer and Stop Word functions.

The Stemmer will be Porter Stemming algorithm since it seems to be much easier to implement than the other stemmers. Porter’s Stemmer seems to be quite widely used too by the information retrieval field and seems to be the one that works best. It is as said before made to work for English language and is implemented in steps to stem a given word. Depending on the word it may be stemmed 0-5 times depending on how many steps it fits into. While every word goes into all 5 steps they will only get stemmed in the steps if they fits into the rule for that step if not then the word is return not stemmed.

Stop Words as mention before in its section is a great way to get rid of words that affect the similarity measure for similarity between 2 documents. These words will also be in under the English language and will be implemented by comparing if the word is equal to the stop word or not before deciding on the removal of it.

As it already has been hinted more than once, the language of the texts that are going to be compared will be English. English is easier to use since it only has 26 characters compared to Danish 29 and the Japanese 48+kanjis which is quite many since it is based on Chinese words and there keeps appearing new ones every day. Another reason for limiting to English only is that the stop word list is in English that in itself is not a big problem since it is a list of words put into a list and can be changed to another language. The Porter Stemmer is probably the biggest reason for limiting to English only. Porter Stemmer is build up so it only handles the words from the English language. Dealing with words from other language would mean to rebuild the Porters Stemmer and that is outside of this paper's range.

To let the algorithms handle the texts better, all texts get through a lowercase process so case sensitiveness disappears. The symbols ";:(){\"-,.!?#" all gets eliminated since they will only hinder the similarity and has nothing to do with the texts being the same. "&" is replaced with "and" because some people prefer this symbols instead of the word itself. Last the "' " gets replaced with " ' " since the programmer's system cannot handle the first. All these symbol changes in the text are to make the algorithms have a better chance to find out how similar the texts are. Those symbols will only hinder the algorithms instead of helping out with measuring the similarity while they do not say anything about similarity thus this action will happen as a preprocessing process before the algorithms are run on both texts.

A special measure to find the similarity score in Cosine Similarity is not needed since Cosine Similarity, as shown before, calculates a value between 0 and 1 which can easily be converted to percentage by multiplying with zero. Optimal String Alignment is another case though since only the distance is found. Out of the measures shown in the Similarity Score section only one will be needed. Since

$$\text{Similarity}(\text{A}, \text{B}) = \frac{1}{1 + |tri(A)| + |tri(B)| - 2\ x\ |tri(A)\ \cap\ \ tri(B)|}$$

deals with trigrams and this is not how Optimal String Alignment algorithm will work, it can be filtered away from the possible options.

$$\text{Similarity}(\text{A}, \text{B}) = \frac{1}{1 + \ Distance(A, B)}$$

Would be a nice idea to implement but after testing out with the strings "Mary got blue eyes" and "Mary got big blue eyes" which resulted in a similarity on 0.2000 (20%) [14] it has been decided to instead use the simple method to calculate similarity between two strings in Optimal String Algorithm:

$$\text{Similarity}(\text{A}, \text{B}) = \frac{Distance(A, B)}{MaximumLength(A, B)}$$

A bit of fuzzy logic will also be implemented into the Similarity Score in the way it is shown to the user. The options will not just be similar or not similar but instead will contain four options. If the score is between 0%-30% then the texts are different meaning not similar, 31%-50% means that they are a little similar, 51%-85% would mean that they are mostly similar to each other and last 86%-100% means they are completely similar to each other that they can close to be or are the same pieces of texts. [15]

For the timing part the wall clock time will be used and there are 2 reasons for this:

Wall clock time seems to be the only one to be able to calculate both in nanoseconds and in milliseconds, which is quite important when measuring time for longer text pieces.

According to Java documentation notes

“The returned value is of nanoseconds precision but not necessarily nanoseconds accuracy.”

This means that the timing will not be exact because of the overhead in JVM and OS, and is hardware dependent. So moving to milliseconds when measuring is better the faster it is done. [16]

CHAPTER 4

Design and Structure of the Application

From the start it was decided to follow the Model-View-Control (MVC) architecture to approach the structure design of the application. It is done to make it easier to implement the different classes, have a better overview of the application and make it easier for future developer to understand how the application works. Following the MVC structure means that the application will be separated into 3 units:

Model: This unit of the application as it hints is about dealing with the storing, manipulating or maintaining the data or the behavior of the whole application. This is where the application's "heavy" work is happening.

View: Again this unit of the application speaks for itself too. It deals with all the data which displays the data from the Model unit to the viewer. Typically it consist of a user interface elements e.g. GUI.

Control: This unit of the application connects the Model unit with the View unit. The user's response via the interface gets accepted by this Control unit and sent to the Model which is then instructed to work and then send the result back to the View unit to display it to the user.

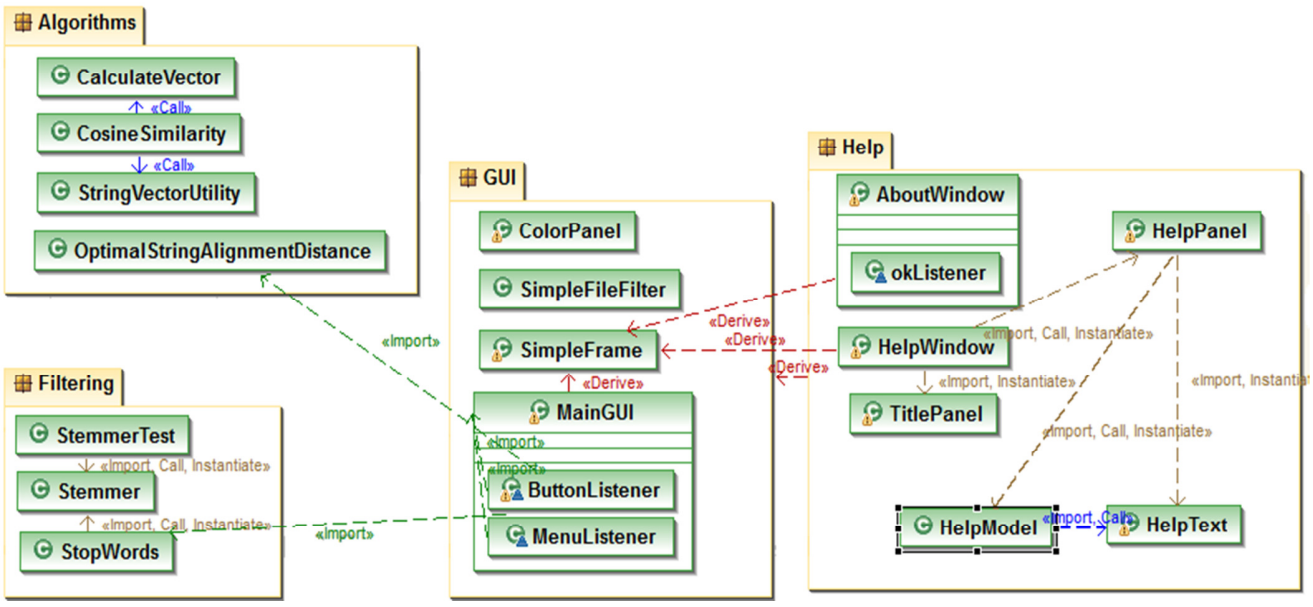
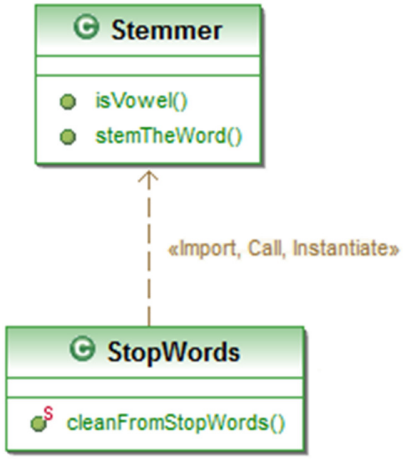


Figure 3: Package overview with all the classes.

The above figure shows nicely how the whole application is build up by showing all the classes only in their packages. The Model unit here is the “Filtering” and “Algorithms” packages with the classes which do all the heavy work. The “GUI” package which says it by its name is the View unit in the application. It is hard to see where the Control Unit is and some may think it is the last package left “Help” but that is a wrong assumption. The Control unit is implemented into the “GUI” package in under “MainGUI” class as internal listener classes. The “Help” package is a separate part of the application and has its own MVC architecture just as the application itself.

This section will deal with all the classes implemented into the program where each part will have its diagram shown and explained. The full view of all the classes with its methods and dependencies can be found in the Appendix C where both the classes, without the methods but with the package name, connections are shown, and the classes, with their methods shown.

4.1 Filtering Package



This package consists of two classes; Stemmer and StopWords.

The texts get sent from MainGUI class into the StopWords class to get filtered. It is here they get stemmed and all the stop words are removed from the text before sent back to the MainGUI class again.

4.1.1 StopWords Class

When the text arrives at StopWords class, they get cut into words and thne compared with the stop word list before they get sent into the Stemmer class. Reason for this is that there should be no reason for stemming words that are not needed anyway. The time it would take to stem and then checked if they are stop words gets saved, not to mention a stop word stemmed will not look alike the stop word that should be removed. Thus it end up going back to the text again when it should really be removed. The main function cleanFromStopWords() is the only function in StopWords class and it is in there the whole process happens.

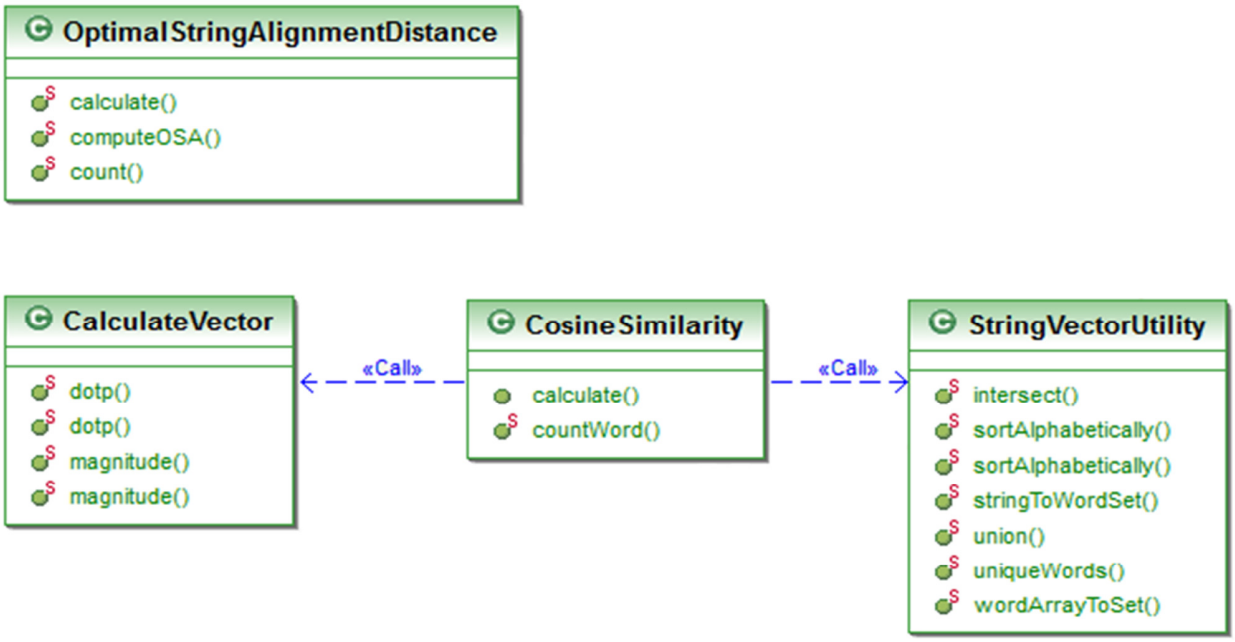
The Stop word list can be implemented into the program manually but it is found more useful to let the user decide what kind of stop words are needed to be removed and what are not. That is the reason why the stop word list is a txt file that is called into the StopWords class and then put into a HashSet. This way the words from the Hashset are compared with the texts and depending on the user the stop words varies. Still a stop word list named stopword.txt is enclosed with the application. The words used in that stopword list can be seen in Appendix D and consist of 493 stop words. [17]

4.1.2 Stemmer Class

The Stemmer class which is called in the StopWords class is called just right after the word from the text has been checked for being a Stop word or not. If it is a stop word nothing happens except it being filtered from the text. If it is not a stop word then it is put into the Stemmer class to get stemmed before it can be appended with the other words and then returned to the MainGUI class.

The Stemmer class uses the function stemTheWord() as a public function which is what method the StopWords class uses to send the word to the Stemmer class to get stemmed. The Stemmer class consists of many private classes to stem the word sent from the StopWords class. There are 5 steps as written in Porter's algorithm to stem a word. [18] Each step has its own way of stemming the words according to the rules of Porter's algorithm. If the word is either stemmed or not before it is returned from that step only to be sent to the next step. The rules are manually implemented and cannot be changed like the stopword list since they are following the rules that Porter made to make the algorithm fit the English language.

4.2 Algorithm Package



This package contains all the classes needed to calculate the similarity between two texts, both for Cosine Similarity (CS) and for Optimal String Alignment (OSA) distance. While the OSA only has one class called OptimalStringAlignmentDistance, the CS algorithm has 3 classes called CalculateVector class, StringVectorUtility class and Cosine Similarity class.

4.2.1 OptimalStringAlignmentDistance Class

The OptimalStringAlignmentDistance class has three public and two private methods in it. The private methods are two overriding methods which find the minimum of two or three integers which are then used in the compute method. The Minimum function which takes three integer parameters is used in the algorithm part that looks the same as Levenshtein while the extra part that makes this OSA uses the method which takes two integer parameters. For more details about the algorithm, see Levenshtein and OSA under Decision based on the Analysis and Algorithms used for Textual Similarity.

The computeOSA() method is where the whole algorithm for OSA is written. This function takes two texts as string parameters and calculates the edit distance for the strings. It returns the distance as an integer return value.

The count() method is a simple method that takes the two texts as strings and returns the maximum length of those strings in integer.

By getting the distance from the computerOSA() and count() methods, the calculate() method does as it names says: It returns similarity of both texts from the OptimalStringAlignmentDistance class to the MainGUI class.

As mentioned before, CS uses three classes to calculate the similarity between two texts. The reason for splitting it up into two classes is to make a better overview of what happens where and how they all get connected.

4.2.2 StringVectorUtility class

StringVectorUtility class is the class that takes the strings of both texts and converts them to vectors, intersects, unions them and finds the unique words in the strings. Basically it is a utility class that does the preparation before the math calculations are going to start. After doing the preparations where the union function is the most important one in the whole class, it sends the result to CosineSimilarity class.

The union() method takes the two texts and strings and puts them into a TreeSet for strings. Though it cannot just put them into the TreeSet it sends the strings to another method in the class, called stringToWordSet(). This method cuts the string into words and then places them into a HashSet before returning the set to the TreeSet which merges them all into a merged vector called mergedVector. Before returning to the result to CosineSimilarity class, the mergecVector is put through another method to find the uniqueWords in the merged vector.

4.2.3 CalculateVector Class

The CalculateVector class is as it names implies: The class where calculation to get ready for CS takes place. It consists of two overriding dotp() methods that both calculates the dot products of two vectors. It also consist of two overriding magnitude() methods that calculates the magnitude of a vector.

Both methods send, depending on which of the two overriding method are used, their results back to the CosineSimilarity class. dotp() returns its result as integer while magnitude() sends its result as a double. For the detailed information about the math and the algorithm, see Cosine Similarity under Decision based on the Analysis and Algorithms used for Textual Similarity.

4.2.4 CosineSimilarity Class

The main calculations of the CS happen in the CosineSimilarity class. It contains three methods in its body: Two public and one private. The public method countWord() which counts the words and returns the result to the private method createFrequencyOfOccurrenceVector(). This method finds the frequency of the vector in the string and returns the results to the calculate() method in the class.

calculate() starts with taking two texts as string as its parameters. Then it gets the strings union via the StringVectorUtility class, only to move to its own class' private method createFrequencyOfOccurrenceVector() to find the frequency occurrence of the vector for each string by using the union found before as one parameter and the other being the given string.

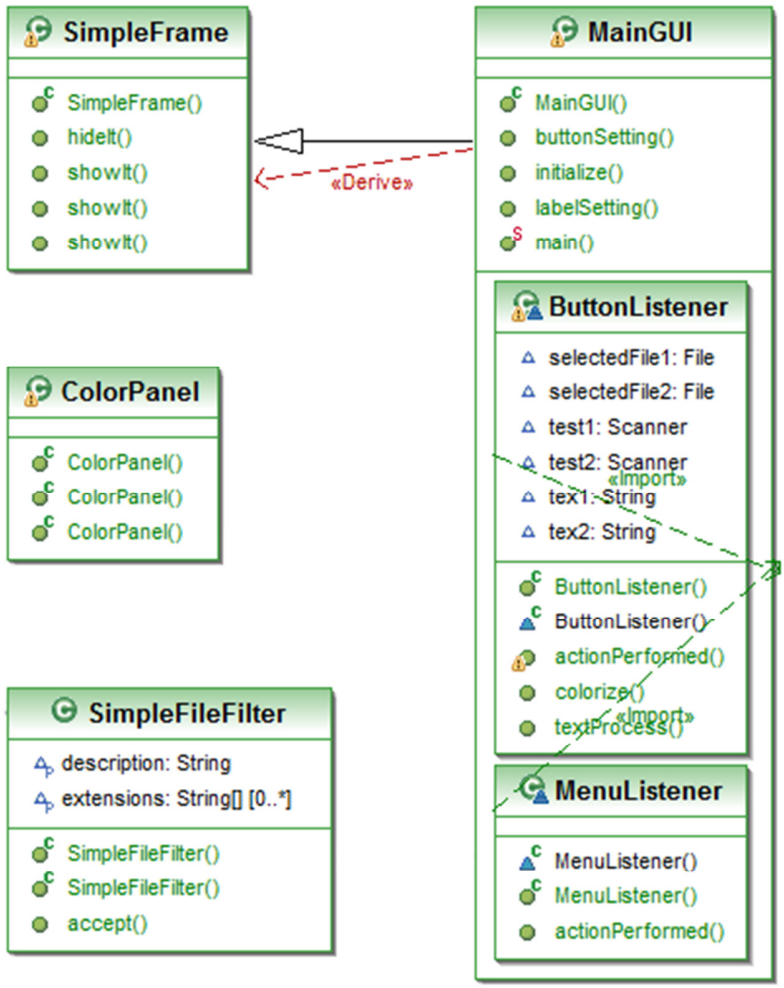
The returned vectors are now used as parameters to find the dot products for both vectors via the CalculateVector class. The magnitude for each vector is calculated using the same class as with dot product.

At the end it returns the result to MainGUI by putting the dot product and the magnitudes into the Cosine Similarity equation.

4.3 GUI Package

This package is a bit special compared to the other packages. It consists of six classes whereof two are used to extend one, two are internal classes in one and one is used as function in one.

SimpleFrame class and ColorPanel class are the classes that extend in the other classes, mainly MainGUI.



4.3.1 SimpleFrame Class

SimpleFrame is a frame to be drawn on with specified settings. The SimpleFrame() method in it sets the settings e.g. size and location of the appearing frame. The hideIt() method will hide the frame if set to true while

the three showIt() overriding methods each sets a specified setting for the frame when showing it e.g. the title of the frame or visibility.

4.3.2 ColorPanel Class

The ColorPanel works in the same way as the SimpleFrame except it sets the colour of the frame and changes size if needed to.

4.3.3 SimpleFileFilter Class

SimpleFileFilter class is a class that checks if the file, the application is going to open, has the specified file format. It takes the extension and name of the file and compares those to see if the file is that specified file format. It return true if it is or false. This way the files in the directory become filtered to only show the file format that has been specified while the rest are hiding. It also sends the description given to it as the file of type the OpenDialog will accept.

4.3.4 MainGUI Class

The heart of the GUI package and the largest class in the whole application. It is the main GUI class which shows how the application should look. It has all the buttons, menu, settings and is the View unit of the MVC system. This class has its own functions and to make it special, it has the Control unit as its internal classes: ButtonListener and MenuListener. It is also here the main function for the whole program delves.

It was decided after many tries that the best way to show the intended design, the GridBagLayout should be used as the layoutmanager for the application. It works perfect with insets and anchor methods.

The label and buttons are made in this class with their sizes, names and placement in the GUI. To make the code less redundant methods as buttonSetting and labelSetting have been made to set their specific settings and placement. This is done to make the code give a better overview. It is also in this class that the menus have been made for the GUI. Menus that will be used by the user to many things e.g. close the program, restart or use the help function in the program.

One of the important things that have been added later is the “new” option in the menu. That option resets all the stats and the user can start to upload the files anew. The function for this resetting is also in the MainGUI and it called initialize(). It sets everything back to zero. More about this under Usabilty Test section.

Before moving on it is worth mentioning that it is here that the menuListeners and buttonListeners are added to each menu and button.

4.3.5 MenuListener Class

This is an internal listener class in the MainGUI class and it handles the menu actions. Pressing on one of the menu options will lead to an action and this is where it shows what happens. Since there is a simple GUI menu in

this application it does simple things. “New” resets the application, “Exit” exits the application, “About” shows another frame called about and “Help” shows the help functions frame.

4.3.6 ButtonListener Class

This is also another internal listener class in the MainGUI class. It handles every button in the GUI and attaches them to their actions. As mentioned before both the MenuListener and ButtonListener are Control Unit of this application and it is their job to control what is going to happen when a menu or a button has been pressed.

When the “Text 1: “ and “Text 2: “ buttons have been pressed, the text files are being read. It is also here that the file format is specified, in this case to txt files only. The text is being read and the case for each letter is being lowered so no capital letters exist in the text string before it is put through the other options in the application.

The next four buttons are one for each procedure of each algorithm. The difference between the first two and the last two is that the stemmer and stopword has been added to the last ones as their actions before the algorithms are to run on the text files.

The procedure goes like this:

1. Start the time.
2. Preprocess the text files.
3. Clean the text files from stop words and stem the rest of the words. This is only run if the option is supposed to run these.
4. Run the texts through the algorithms.
5. Round the similarity to proper percentage value.
6. Stop time.
7. Show the result on the GUI.
8. Colorize the result depending on how it is rated on the similar scale.
9. Make the time to string.
10. Show the time on the GUI.

The only thing to mention here is the colorize() call which is also a function inside ButtonListener class to make it easy for the user to see how similar the two texts are. The function for preprocessing the texts before running them through the algorithms aka textprocess() also exist under this class. Which numbers these colors are based on and what symbols are being processed can be read under the section Decision based on the Analysis.

4.4 Help Package

The Help package consists of classes related to the Help and About functions in the menu under Help.

The AboutWindow class is a simple frame with information and a button to shut down the frame. Together with HelpWindow class it extends the SimpleFrame class.

The HelpWindow is the GUI for this help function and uses a BorderLayoutmanager to keep the different panels into their place. The class adds the panels made in other classes into it and sets the frame settings to show the user.

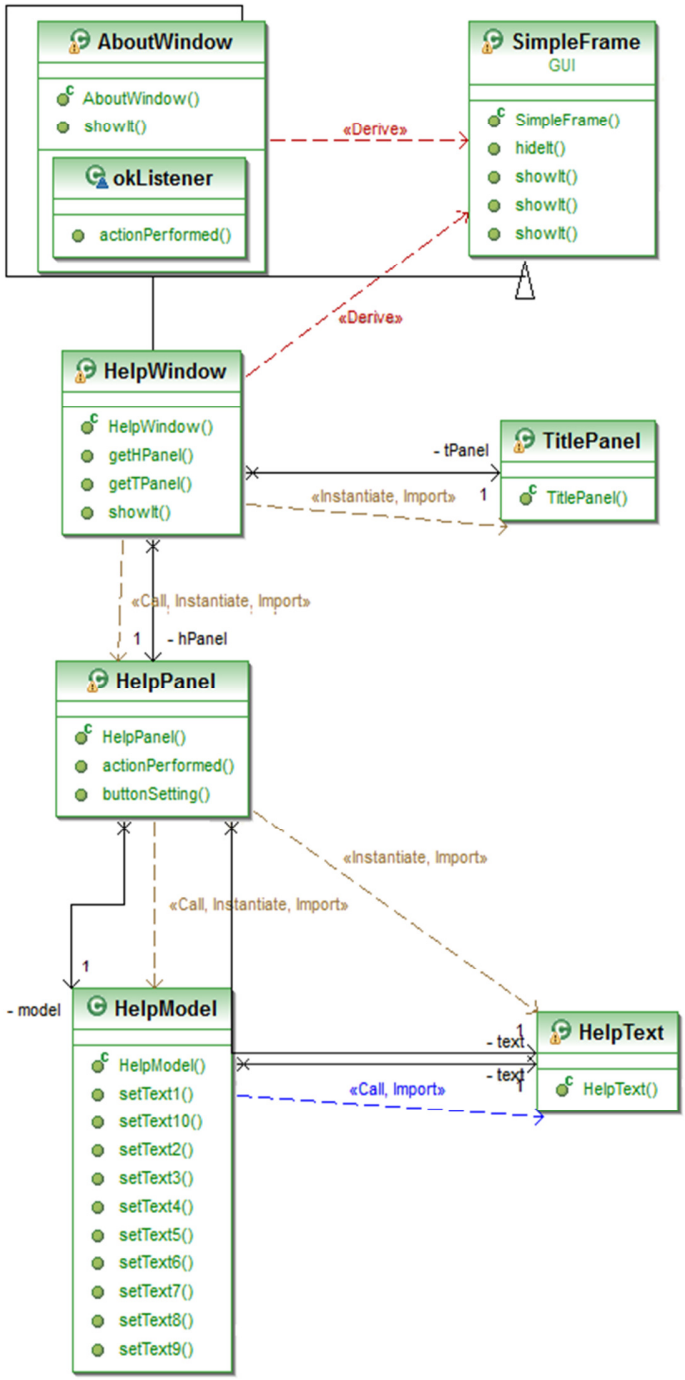
The HelpText class is a simple class that shows the initial text on the help function. This is the frame that is changed every time the text is changed.

The HelpModel contains ten methods where each method has a specific setting and the text to show. These texts are called and shown on via the HelpText class.

TitlePanel class is a very simple class that contains the title of the Help function that is shown above the help text to the user. This is not changed through the whole process the user uses Help in the application.

HelpPanel is the class that uses the FlowLayoutmanager to put the buttons for each of the help options to the user. The action listeners to the buttons are also in this class and the same are the actions related to those. Each of the buttons are connected to their action which pressed shows the specific text for the button by getting the text from the HelpModel and showing it through the HelpText.

Again to remove redundancy and made a better overview, a method to set the buttons settings has been made and is called buttonSetting().

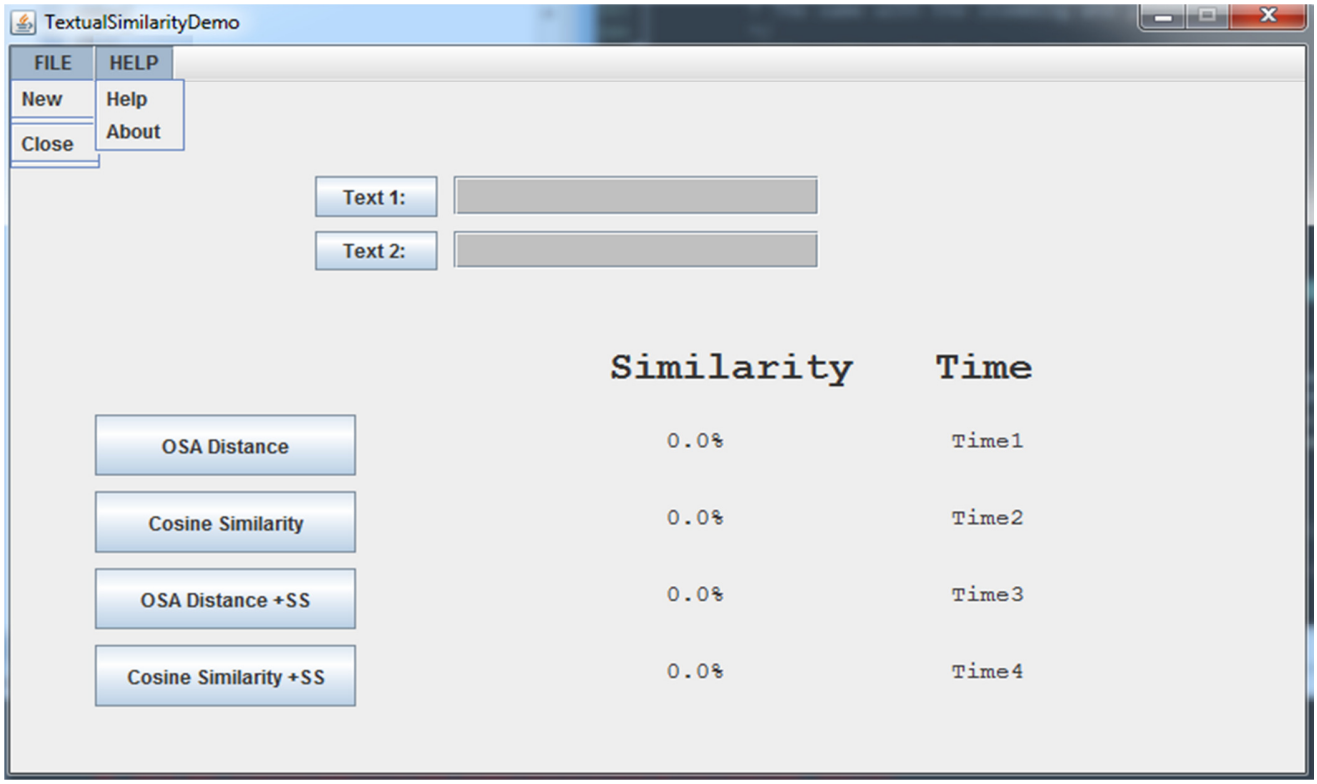


The idea to the design of the application has always from the start been to make it simple to the user to use without adding too many smart functions to confuse the user. To say it simple: The focus has been on usability.

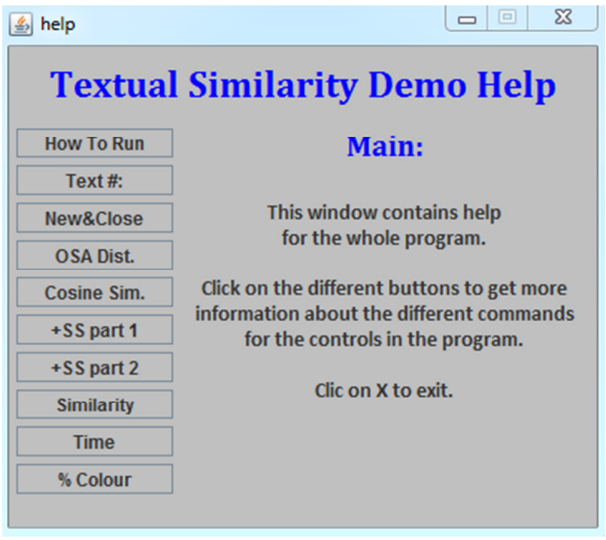
As said before the language for the application is Java which is platform independent meaning it is able to run on every computer that has Java installed. This is something nearly every computer home has installed on their machine since Java is being used to many of the application that are being run on people's computers even without them knowing.

While the program should be able to run on every computer that has Java installed, it is still recommend running it on computers made after 2005 to get good results and no lagging. This program was written and run on a machine with the stats:

- Intel Core i3 CPU 2.67GHz
- 4GB memory (only 3.8GB usable)
- Windows 7 – 64bit with service pack 1
- Java SE 6 Update 25 Installed



The above window shows the design of the application of how it looks with the FILE and HELP menu options open to show what kind of functions can be chosen from the menubar. As it can be seen the design is simple and easy to use if the user still have any problems with understanding the different buttons and what to do, then the user can always go into the HELP menu and choose Help to get some help understanding how the different buttons works in the program. The picture below shows the Help window:



CHAPTER 5

Application Tests

Every program is filled with logical, syntax and semantic errors, which are not proviso while the application is developed. Even when the errors found, designing and coding the application, are fixed under the development period, there are still many errors that are not known until much later, also called bugs. Bugs are mainly consisting of semantic and logical mistakes because all syntax errors are founds by the compiler under the program is being coded.

An example could be if the stemmer in the application is working correct or not. By checking out the stemmer's abilities, it becomes clear if the stemmer is functioning as it should or there are mistakes in the coding. This method ensures if the stemmer is working correct or not, what should be changed if it is not working correct, if it is necessary to change and so on.

Thus many kinds of test have to be run on the application to ensure that it works as it is intended to do so and to minimize the bugs that can be found later after its release. The prototype has been tested under 3 different kinds of tests:

- Structural Test (White Box)
- Functional Test (Black Box)
- Usability Test

5.1 Structural Test (White Box)

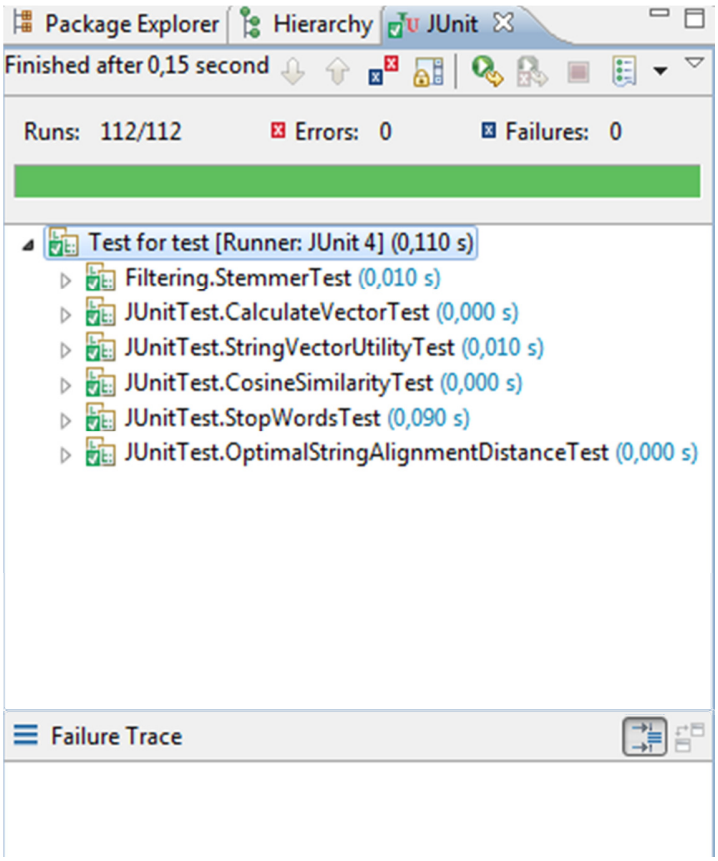
Structural test (also called White Box Testing, Clear Box Testing, Translucent Box Testing, Glass Box Testing, or Translucent Box testing) is usually about running test on all the methods in the application. These are used to test if all the methods are running as they should. Since all the tests are designed to test the inner structure of the application, these tests have to be changed if the application's method changes.

The approach to test all the structural tests is done via JUnit testing. JUnit tests the method directly so to test a method is working, the output for that test should be equal to the expected output. Using this approach to test all the branches for a given method in a given class, is the same as testing the structure aka structural test.

All of the JUnit tests are placed in the package JUnitTest except one test. This makes it easier to run the tests on the classes by having them in one package for all the tests. The problem with why StemmerTest was not moved was that it was not working if it was moved to JUnitTest package. The problem could not be found so it was placed in the Filtering package together with the Stemmer class. To make it easier and get a better overview the JUnit Suite called AllTests was made. By running this test suite, the tester is able to run all the tests (also StemmerTest) in one go from the JUnit package and check for errors.

Many of the tests have setUp and tearDown methods in them. The setUp method is used to set up the testing class' method up before running the test and then go back to initial after the test is done by using the tearDown method.

After doing the total of 112 structural tests for all classes in the application many errors were found and fixed. The figure shows the overall view of all the JUnit classes in the JUnit Suite. The tests for each class in detail can be seen in Appendix E.



Two serious errors were found and fixed in the Stemmer class and StopWord class.

In the Stemmer class the word as a string thrown into the class via StopWord was not going through all the five steps but instead only through the first one only. This was fixed easily but was the kind of error that actually made the similarity percentage go up by many points. Not noticing it would have disastrous since it is related to similarity which is the main part of this project.

In the stopWord class it would first stem the word and then remove it if it was a stop word from the stopword list. Problem with this was that most of the words that should be removed since they were in the stopWord list were not removed since they had been stemmed and were not alike the words in the list. By making a little change this was fixed but again it was serious enough to affect the similarity and the time of the application.

Many errors big and small were fixed but still some errors may still exist since structural tests are tests made by the programmer who is a human. Humans can only do those tests that they can think of themselves which means that all possible tests for the application is impossible to do thus some errors will be unaccounted for.

Looking away from the unseen possibility of other structural test that could be made for this application, not all of the functions of this application can be tested by structural test. Those are tests that test the functional part of the application and can only be tested by functional testing.

5.2 Functional Test (Black Box)

In contrast to the structural test, the functional test (Black Box testing) takes basis in the outer structure of the application. The designer of the functional test knows nothing of the structure of the application and designs the test to check the application itself based on its functions. These tests are founded on the available user inputs and decided on if the output gives the expected result. The test will not test all the branches in the application structure but will help to show if there is missing some functions or something is going wrong in the application.

This test is based on the Design and Structure of the Application section and the main concept with the application in this thesis: Similarity.

To be able to do that scenario called use-cases are made. Use-cases are scenarios which are founded on what the application is capable of, in this case the above mentioned section, and check out if the user is able to do those things in a given situation. It is this way that the functional test can be made to test out the functionality of the application.

A “use-case” is a test that consists of the following categories:

Goal:	The purpose with this functionality.
Requirement:	What conditions to fulfill before the use-case can start.
Operation accomplished:	What happens if the functionality works.
Operation failed:	What happens if the functionality fails.
Summary:	A short summery of a success sequence of events.
Restrictions:	When the actions cannot be legal.
Actor:	Who will do the test.
Priority:	How important the functionality to the application.
Used:	How often this functionality is used.

An example of a use-case can look like this, the rest can be found in Appendix F:

use-case 2: Loading 2 files into the program

Goal:	To load 2 files into the program.
Requirement:	Txt files to compare should be available on the computer.
Operation accomplished:	If the file name is written on the text field just beside the buttons to load the file into the program.
Operation failed:	If the file name is not written on the text field just beside the buttons to load the file into the program.

Summary:	The program shows an Open Dialog box, hereafter the txt files are shown, the user chooses a file and It is uploaded into the program while the title is shown in the text field.
Restrictions:	None.
Actor:	User.
Priority:	High.
Used:	Often.

JUnit is not used in this form for testing but instead the interaction that goes between the user and the graphical user interface. Reason for this is that JUnit cannot be used to test out all the functional test, like it could with structural test, since some of the tests are done with the application has reached a specific point only done by a user.

The results of the functional test based on the use-cases can be seen on the table in Appendix G and shows clearly that the application passes the test. Though the tests which passed are not as interesting as the ones that did not pass, there is still one that is quite interesting to mention while it still passed. The specific result is the one with * in the table. Reason for this * is that the application has been made so that when the user

- presses on the algorithm buttons without loading any files or
- presses on “New” and then the algorithms buttons without reloading any files

a pop up window will not meet the user but instead the similarity on these will be 0% for OSA (with or without SS) and 100% for CS (with or without SS). This is no error but a result of handling an error.

It can be concluded that the functional test worked as it should and the application does what it is designed to do. There have been some mistakes which were fixed after the first functional test. An example is the mentioned * from before.

It should be said that “resetting tests” use-cases were added after the usability test.

Even if the structural and the functional tests have shown many errors, which were corrected and fixed, it is not enough to end the testing of the application. The last test is a test to show what the application can be missing and should be added. Something the other two tests aren’t able to do.

5.3 Usability Test

According to the International Organization for Standardization (ISO) usability is defined as:

> **The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. [ISO 9241, del 11.]**

Meaning usability is a nice interaction between effectiveness, efficiency and satisfaction for the user.

Many programmers look down on usability and think it is a waste of time or that they already thought of it thus they do not need to run usability test on their application. Still usability testing is very important part of the design and programming.

Doing the test in the design process will save the programmer quite some time, which can be used on some other more needed part of the process. The result of the test will for the programmer be inputs about functions and finesses wished to be added to the program. It is best to use this plan if the design specifications are abstract.

Doing the test in the programming process will show the programmer if the made product live up to the specific user group's expectations. The result of the test will for the programmer be inputs about what works, should be removed, corrected, added, deleted and eventually extended with.

From the start it was decided to make the usability at the end of the process, doing the test in the programming process, since the design structure for the application was quite specific. The user group shall belong to a broad group, instead of just a specific, since this application is made for everyone who wants to compare two texts. The user will not get any information about the program except that it is an application to compare two texts and the files should be in txt format.

After testing the program in various ways the user is expected to fill out the scheme with some questions, shown in Appendix H.

After getting the inputs from the testers on the scheme, which asks about the usability, it can be seen that the things the testers liked are

- Simple interface and not too complicated
- Help function

It is always hard for the programmer to know if the interface is as simple and manageable as wished. Doing this kind of test gives a peace of mind when the testers agree on that just like in this case. The wish was to make it easy for the user to use the program without giving up because of complexity and this result shows that the users agree with the programmers' solution that the application is simple and not too complicated. The rest of the things the testers found good or interesting can be found in Appendix I.

While it is good to be proud of the good things in the application it is more interesting to look at the bad things and the wishes for addition into the application.

The bad things and wishing to be add can be found in Appendix J and K.

The bad things in the application:

- Can only run txt files.
- Too simple – You cannot reset the stats.
- While looking for files the type of files can be changed from "txt only" to "all files" if wanted.
- Too Simple – You do not know if the similarity value is enough to say how similar the texts are.

- Help function should tell about every function.

The suggestions wished to be added into the application:

- Colours for similarity after some standards.
- “New” option in the menubar.
- SS to be separate.
- SS to be added via radiobuttons/Checkbuttons.

Looking at both list it can be seen that “Too Simple – You don’t know if the similarity value is enough to say how similar the texts are.” and “Colours for similarity after some standards” are talking about the same thing and since both got over five votes, it will be added into the applications functionality. The same can be said about “Too simple – You can’t reset the stats” and ““New” option in the menubar since both things are referring to the same problem.

Help function which seemed to be fine as it was, was not fine according to the testers and since the value for changing/adding a function was set to 5 or over people wanting the same thing, it will be done.

The first negative thing “Can only run txt files” will not be changed since it was from the start decided that the application would only run txt format files. It was thought that txt file formats were the only ones that could be run as file format. This doesn’t seem to be since someone found the error “While looking for files the type of files can be changed from “txt only” to “all files” if wanted.” Another problem attached to it is that the extension of the files can be changed easily from *.pdf to *.txt and then run on the application.

A solution could be to check the magic signature of the files, sadly since the application runs txt files only, the application cannot be using a specific magic signature since txt files have none of the kind.

About the problem the testers found, this has been fixed in the same way as “New” where a pop up window will not meet the user but instead the similarity on these will be 0% for OSA (with or without SS) and 100% for CS (with or without SS). This can show the user that the file has not gone through the similarity process since the application only takes txt files. If there had been more time this could have been fixed in a better manner.

“SS to be separate” and “SS to be added via radiobuttons/checkbuttons” are some functions that would be added or changed if the users, who wanted this change, had been over four people. Since it was only four and three people who wanted these so these additions or changes, the changes have not been applied to the application.

CHAPTER 6

Similarity Tests

Under Similarity section it was shown that similarity is different from one working field to another, from person to person, and from algorithm to algorithm. The similarity needs to be measured. Not only for two texts but also for the different algorithm options given to the user in the application. Of course what a piece of software decides cannot only be enough when trying to find similarity for two texts. This is why the user group from the usability testing will be tested in similarity too so the results of the application can be compared to the human idea of similarity. At the end the time for finding similarity in the application for the different algorithm options will be compared too.

The tests given to the testers are 3 tests with 30 exercises in total and consist of:

1. 4 exercises with one line text pieces.
2. 13 exercises and 6-7 lines of text for each text piece.
3. 13 exercises but many more lines in the text piece; 20-24.

The first test is to give them an idea how the test works and the rest is the main part of the test to check how similar they find two pieces of texts.

The manner the test is taken happens by showing two pieces of texts; first one is the original while the second one is the edited text piece. The tester can see that from the labels as shown in the example:

Question 3:
Text 1(original)
Mary got blue eyes.

Text 2(edited)
Mary got big blue eyes.

This helps the tester recognize what to compare to and not get too tired of reading the same thing over and over and lose the motivation after a few exercises. Before each of the 3 test the user is presented to a quote.

What the quote says is actually quite unimportant but is used to refresh, flush or reset the mind of the tester before moving to the next test. The quotes are by the test designer's taste intended to motivate the tester to finish the test by using positive quotes.

The exercises for the main part included these changes:

1. Two texts that look alike 100% - The same piece of text.
2. Two texts that are totally different 0% - Different pieces of text.
3. Two texts that are 50% alike, one uses part of phrases/sentences from the other.
4. Two texts that are 25% alike, one uses part of phrases/sentences from the other.
5. Two texts that are 75% alike, one uses part of phrases/sentences from the other.
6. Two texts that are totally alike but changes in the sentence structure.
7. Two texts that are alike but with spelling mistakes (14 mistakes in a total of 64 words)
8. Two texts that are alike but with editing mistakes (5 "words" and 7 "letter placement in words" changes in total of 64 words)
9. Two texts that are about the same things but in different words and sentences.
10. Two texts that are about different things but in the same words and sentences.
11. Two texts that say the same things but in present tense and in past tense.
12. Two texts that say the same things but in plural and in singular.
13. Two texts that are 50% alike, first half is alike while the rest is totally different.

Only 1, 2, 5 and 6 are used in test 1 where test 2 and 3 uses all of those but in different order so the tester cannot read the pattern of the test structure.

These tests will also be run on the application to compare the human input with the application's way of finding similarity between the texts.

Of course the test will be timed to see the performance and complexity of the algorithms and these will also be compared to each other and the changes depending on the size of the text pieces.

Another way of testing the time will be taking a text pieces and add more and more words to see how the time develops depending on the text size as an extended version of above time test. The text being used will be from test 3 and will just be added more and more lines so the text in those pieces may repeat in the same text.

The structure of the tests, the quotes and the information about the text pieces can be found in Appendix L while the tests given to the testers can be found on the CD since 19 pages of text fills too much in the report as appendix while other important things should be included instead.

After reading the text the testers are supposed to answer an answer sheet where they have to answer two things for the question: "How similar are the texts?"

The answersheet looks like this where the total number of rows depends on how many exercises the test includes:

Question	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
1											
…											
13											

Question	TOTALLY DIFFERENT	LITTLE SIMILAR	MOSTLY SIMILAR	TOTALLY SIMILAR
1				
…				
13				

As shown the tester places his/her opinion into the field that expresses his/her decision for the similarity between the two texts best. While the first answersheet works in percentages, the other forces the tester to make some borders for similarity.

The guidance given to the tester for Similarity test can be found in the Appendix M.

Each test were made in pdf so no text editing could be done. The answersheets came out in 3 different formats so the testers did not have any problems not having the right tool to open the format. The format was in doc, docx and odt. These were then filled and sent back to be used as result for human decision for similarity between two texts in different cases.

Last it is important to mention that all the tests for the application can vary depending on the computer being used and these will be done on a computer:

Medion P6630
Intel(R) Core(TM) i3CPU M390 @ 2.67GHz 2.66GHz
4.00 GB (3.8GB usable) memory with 600GB HDD
64-bit System - Windows 7

CHAPTER 7

Expectations

Normally before running tests and getting their results, many expectations are made. These expectations tell what the maker of these tests expects as results from running the tests on the application and in this case how the specific algorithms will behave.

There are five factors that play into the complexity of the program:

1. Compiling the whole program from the first time.
2. The stemmer.
3. The Stop Word list.
4. The Optimal String Alignment Distance Algorithm.
5. The Cosine Similarity.

These factors interact with each other in this way that when the application is running the first time, 1. will be included into the algorithm method no matter what. 4. and 5. can be run alone if the application has been run at least once where 1. was included in the run. 2. and 3. cannot be run alone since they are both an addition to 4. and 5. in their respective separate algorithm option. 2. and 3. will always be run together and not separate.

Factor 1: Starting with the complexity for the application, it is easy to see that compiling the whole program and running it the first time will result in a time constant K, no matter how big the text strings are. This constant will affect the timing until the strings becomes big enough so the constant will not have any effect on the algorithm being chosen.

Factor 2: Looking into the Stemmer class, it is noticed, that it follows some steps where there are only “if or elseif or else” statements. A few for-loop also appear but not more than only one for-loop each time. Based on these facts, the Stemmer class runs in time O(n) and not O(n log n).

Factor 3: The StopWord class runs in O(n) too because even if has two while-loops they are not nested but separate. There are some if statements but they should not be able to beat the runtime on O(n).

Factor 4: Moving to the first algorithm Optimal String Alignment has 4 for-loops which can be read from the code. While two of them are separate and gives a runtime on O(n), the other two are nested, a for-loop with a for-loop inside it. This changes the runtime from linear to quadratic, since it is a nested loop. The running time for this algorithm will therefore be on O(n^2) and not higher than this because the nested loop is not deeper than 2 for-loops.

Factor 5: The last factor and algorithm, Cosine Similarity, has 3 classes. Looking into those it can be seen that all of the functions run for-loop and nearly always separate. There is one function that have a nested for-loop that could cause the runtime to become O(n^2) instead of O(n) but that function isn't used by the Cosine Similarity. This is why the run time for the Cosine Similarity will only stay at O(n).

After going through these factors it is quite easy to calculate what the runtime for each algorithm option will be. At the start, with small text strings, the constant K will play a big role every time the application has been started and this will affect the time for each and every of the options available in the application. The longer the strings become the less the constant can affect the runtime for the options being chosen. With larger strings the constant will have no mentionable effect on runtime.

Moving away from the start up and looking on the options alone, meaning running the application first time and then run the option again to remove the first drive with constant K, the runtime result should be:

OSA Distance: Since it is only the Optimal String Alignment running it will be O(n^2) as mentioned above.

Cosine Similarity: Since it is only Cosine Similarity running it will be O(n) as mentioned above.

OSA Distance +SS: Adding stemmer and stop word to the algorithm will give runtime: O(n^2) +O(n)+ O(n)=

O(n^2) + O($2n$)= <u>O(n^2)</u>

Cosine Similarity +SS: O(n)+ O(n)+ O(n)= O($3n$)= <u>O(n).</u>

Thus it is expected that the longer the strings becomes the longer it takes Optimal String Alignment algorithm to take to calculate the similarity than Cosine Similarity.

Looking on the similarity for the exercises and how each algorithm option will behave it is expected that for each exercise a similar or different result will appear.

For exercises "Two texts that look alike 100%" and "Two texts that are totally different 0%" all four should give the same result and be correct at that. Though there is a little hunch saying that the latter exercise will not be exactly at 0%. Reason for that is that there could be some words that that appear in both text pieces and are used as tokens in the options where Cosine Similarity (CS) is used. Optimal String Alignment Distance (OSA) may also be a little higher than zero but not as much as CS will be since OSA works with the characters in the string instead of tokens. This will not be seen in test 1 and test 2, only in test 3 since both test 1 and test 2 use words for the 0% similarity while it has been harder to do so with test 3.

For the exercises "Two texts that are 50% alike, one uses part of phrases/sentences from the other", "Two texts that are 25% alike, one uses part of phrases/sentences from the other" and "Two texts that are 75% alike, one

uses part of phrases/sentences from the other" it is expected that all 4 options will at least be on 50%, 25% and 75% and again as said above CS will probably be a little bit more than those percentages.

With "Two texts that are totally alike but change in the sentence structure" exercise, the options with CS included will give a similarity on 100% while OSA options will be a similarity on 0% or a little more. This hangs together with CS working with tokens so even changing the structures should not pose a problem since the words are not changed while OSA works with the order of the characters and finds the difference between those and not the tokens.

For all the mentioned exercises so far the algorithm options with +SS will be faster than the other two since they will remove the stop words and stem them to make it easier for the string to get through since the space they take will be less.

CS will do bad with the exercises "Two texts that are alike but with spelling mistakes" and "Two texts that are alike but with editing mistakes" since it works with tokens and has no function to correct those while these will be inside the OSA scope which should be able to reach near 90%-100%. The Stop word in SS function will not be able to help CS or OSA so the time for these will raise where it before would have fallen. Reason is that the spelling mistakes and editing mistakes will not be able to remove these words and probably not be able to stem them either since they do not have the form they need to do so anymore.

The exercises "Two texts that are about the same things but in different words and sentences" and "Two texts that are about different things but in the same words and sentences" will react differently for the options. The first exercise will not give a good result in CS and the same with OSA while the latter exercise will give good results in all options. The options with SS will behave in the same way since it is not their job to figure out that different words are being used or that the topic is different but the sentence the same. SS will remove and stem the words as usual though.

The SS algorithm options will shine in the exercises "Two texts that say the same things but in present tense and in past tense" and "Two texts that says the same thing but in plural and in singular" since it is here that the stemmer from SS will do its work. This will result in the first two options giving bad results while the SS options should give a similarity on near 100% if not exact 100%.

The last exercise "Two texts that are 50% alike, first half is alike while the rest is totally different" will probably result in at least 50% since the first half is alike while the other is not, which means that for OSA options the result is expected be 50% while for CS it may go up to 65%.

It is expected that the Similarity results from the application will divert from the Similarity results from the human testers. The reason is that while the application will be following the algorithms made for them, they are not good enough to be able to "think out of the box" while the humans on the other side think in the same way as the algorithm but more than just only that. Sometimes they will use one of the first algorithms while in other cases they will use the second. It can happen that they may mix both of the algorithms in their mind. This is still not the limit for the human mind which is capable of thinking of more than just these two algorithms put together. Understanding how the human brain works is out of range for this paper thus there no need to go deeper than this.

Though it is also expected that the similarity for the short phrases aka test 1 and test 2 will give better results from humans while the longer test 3 may be the same or lower result than the algorithm options. Reason for this is that motivation to go through long texts and still remember it from sentence to sentence, word to word and characters to character when reading the next text piece and finding similarity for both, becomes quite low.

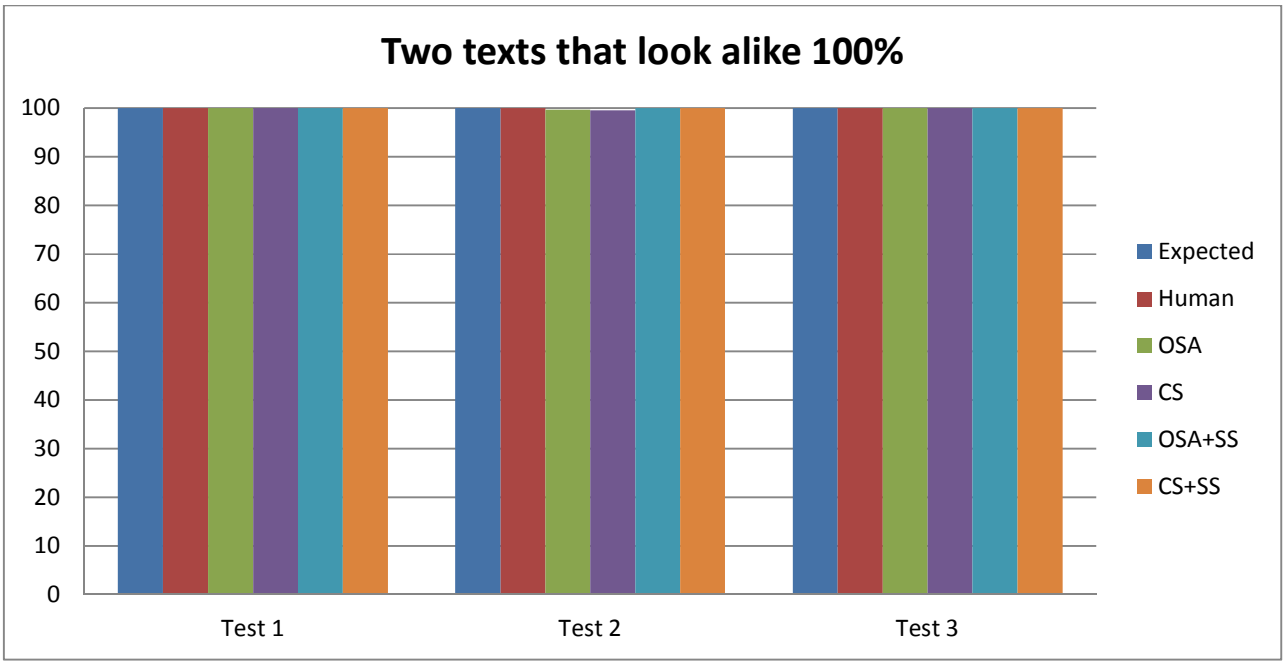
With all that said, the humans testers are subjective meaning they decide based different from person to person thus their result will vary. This spreading of results can be avoided by having many testers go through the test and take the average of those test to represent the human input for similarity.

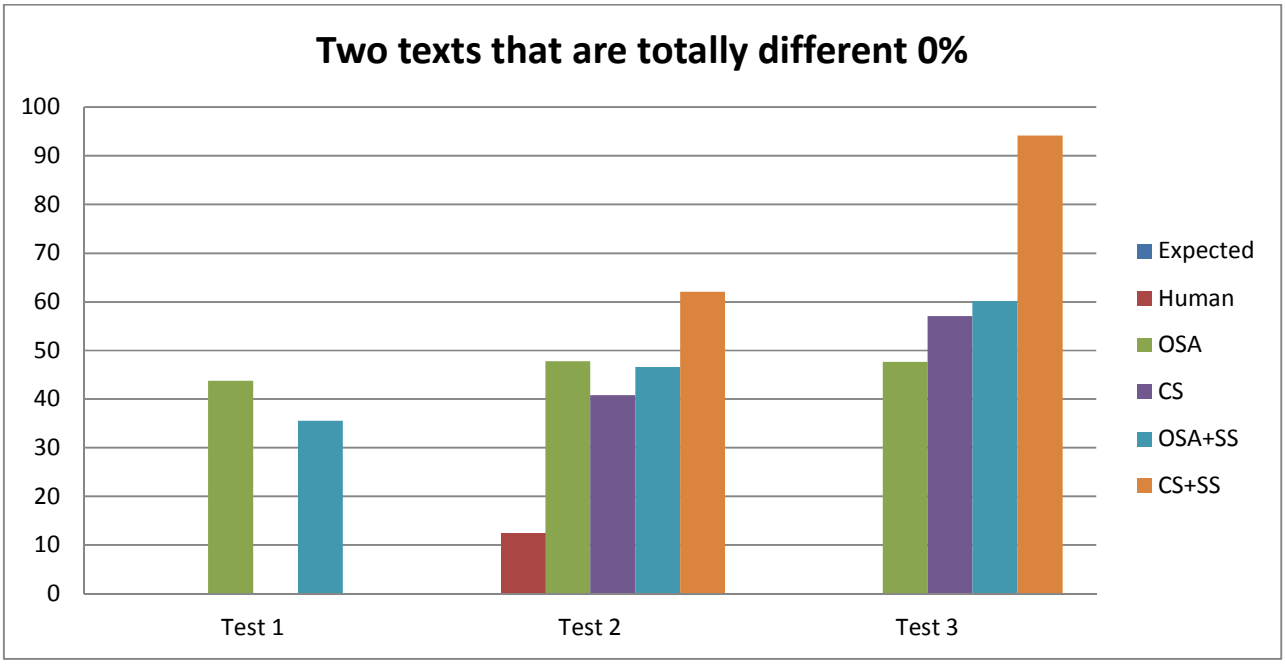
CHAPTER 8

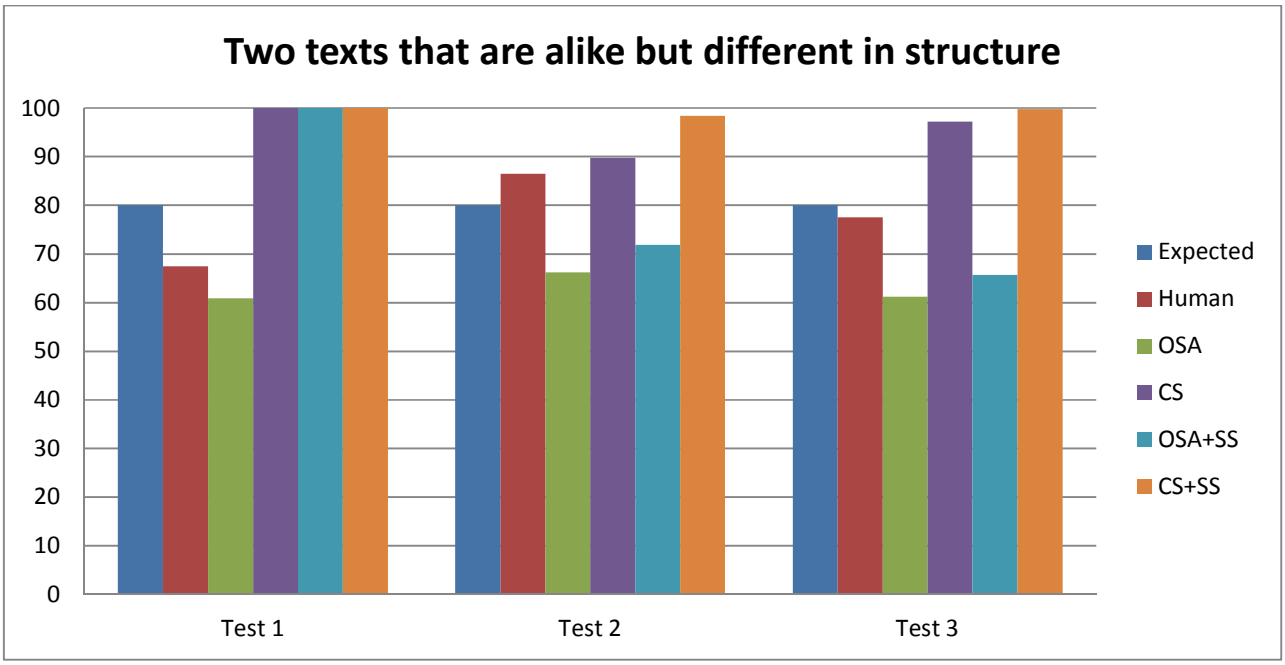
Results of the tests

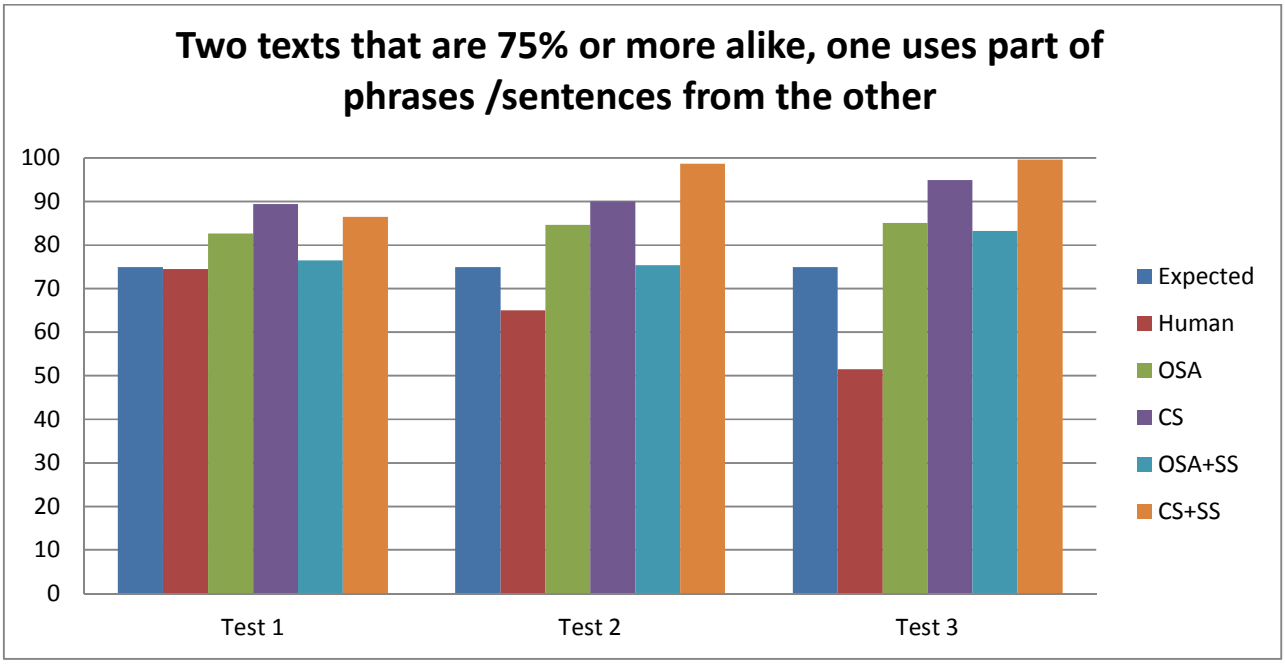
8.1 Similarity Results of Test 1,2 and 3 based on the exercises:

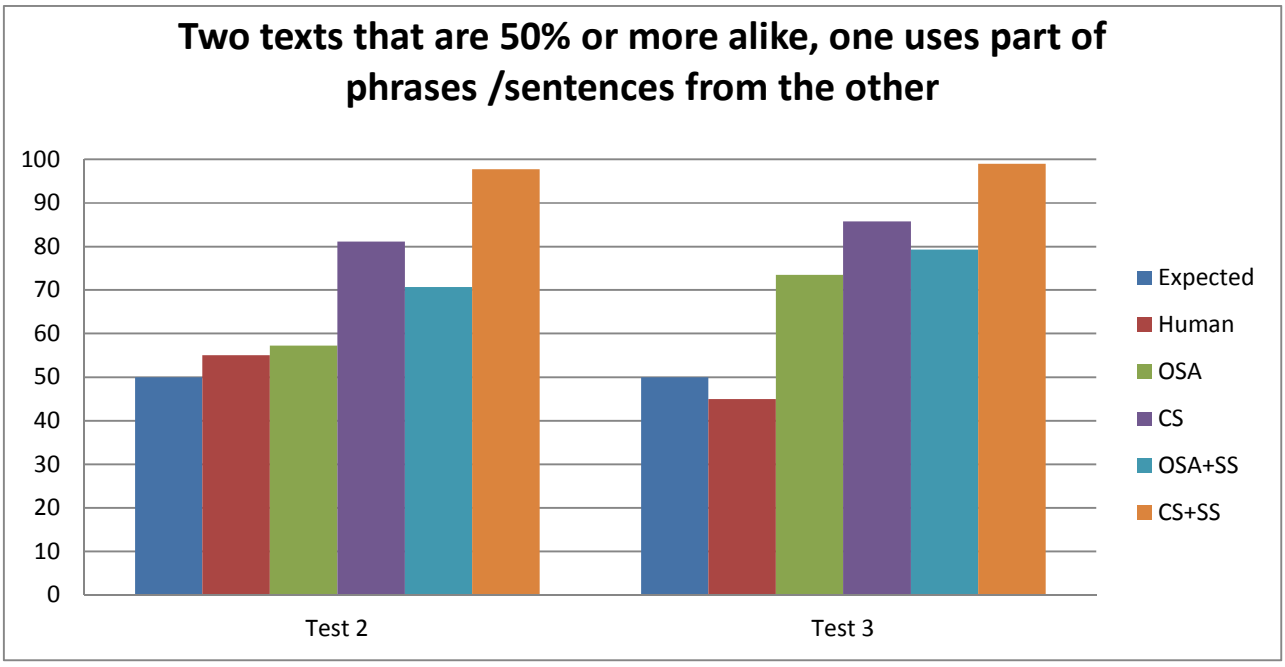
These graphs are based on the sorting done from the raw DATA which is on the CD. The sorting can be seen in Appendix N.

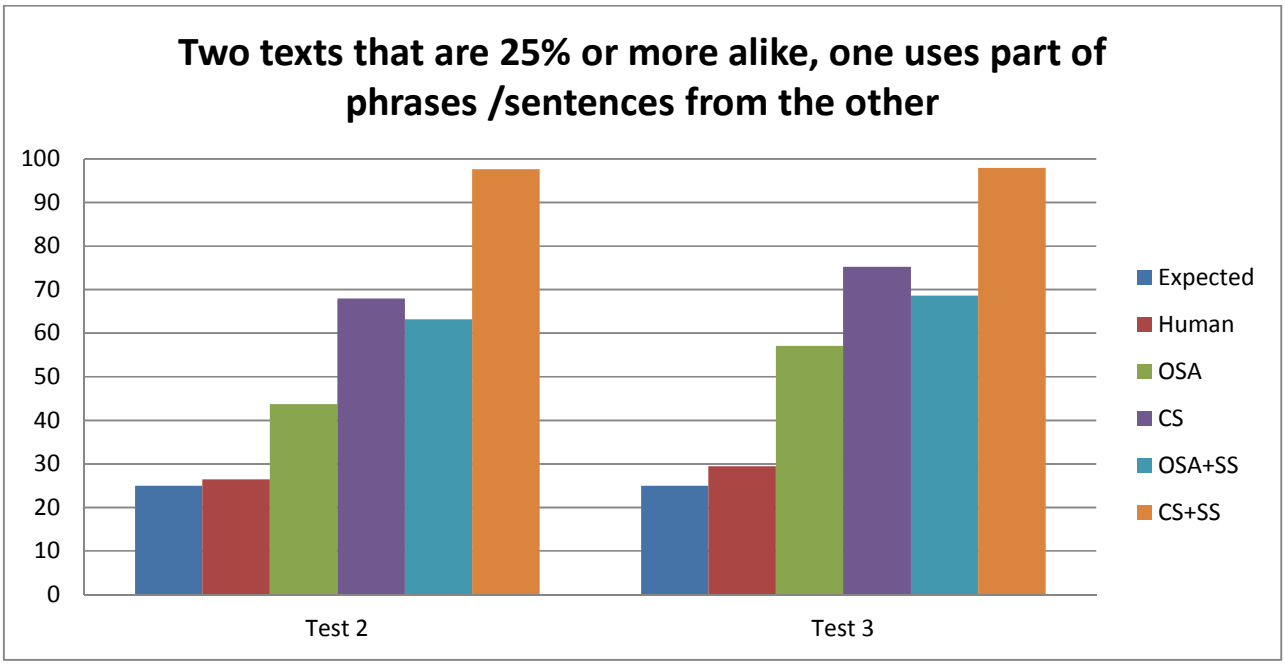


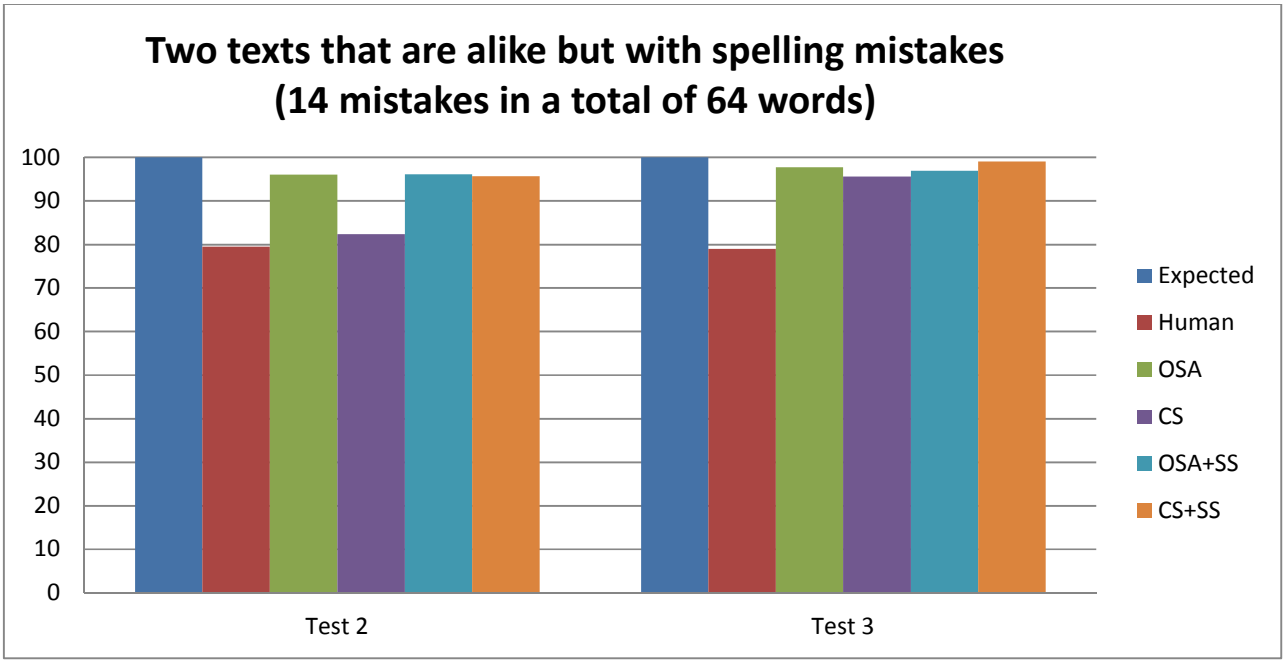

Two texts that are totally different 0%
100
90
80
70
60
50
40
30
20
10
0
Test 1
Test 2
Test 3
Expected
Human
OSA
CS
OSA+SS
CS+SS

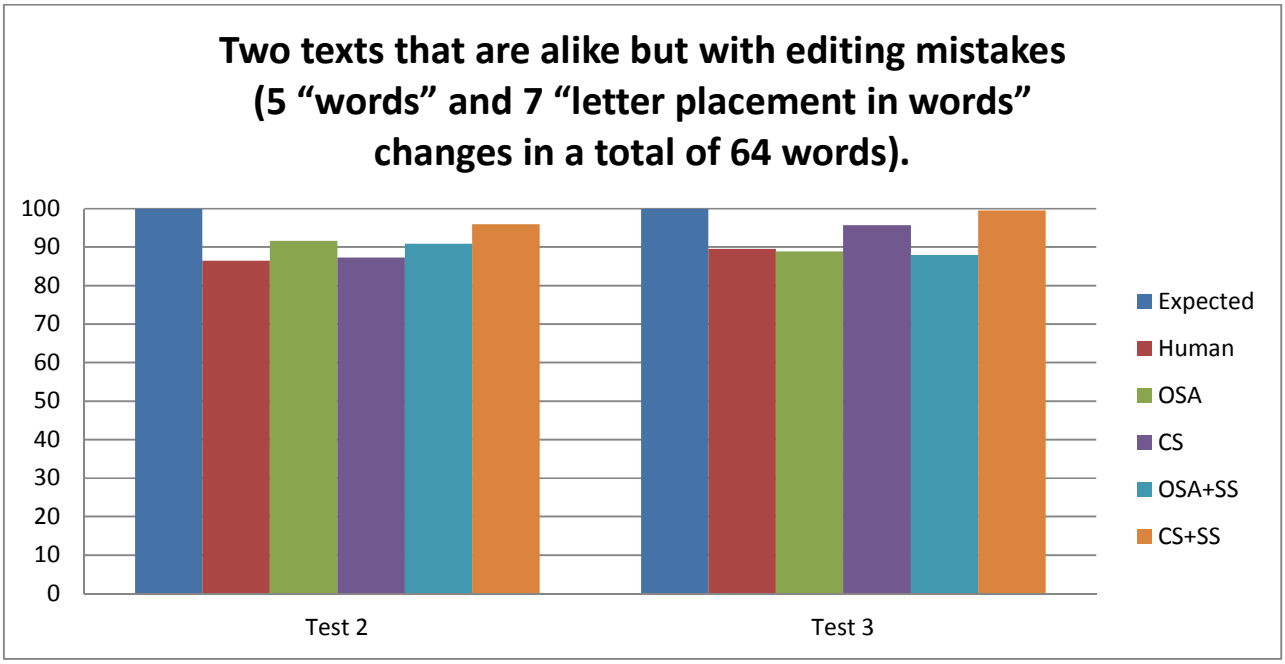

Two texts that are alike but different in structure
100
90
80
70
60
50
40
30
20
10
0
Test 1
Test 2
Test 3
Expected
Human
OSA
CS
OSA+SS
CS+SS

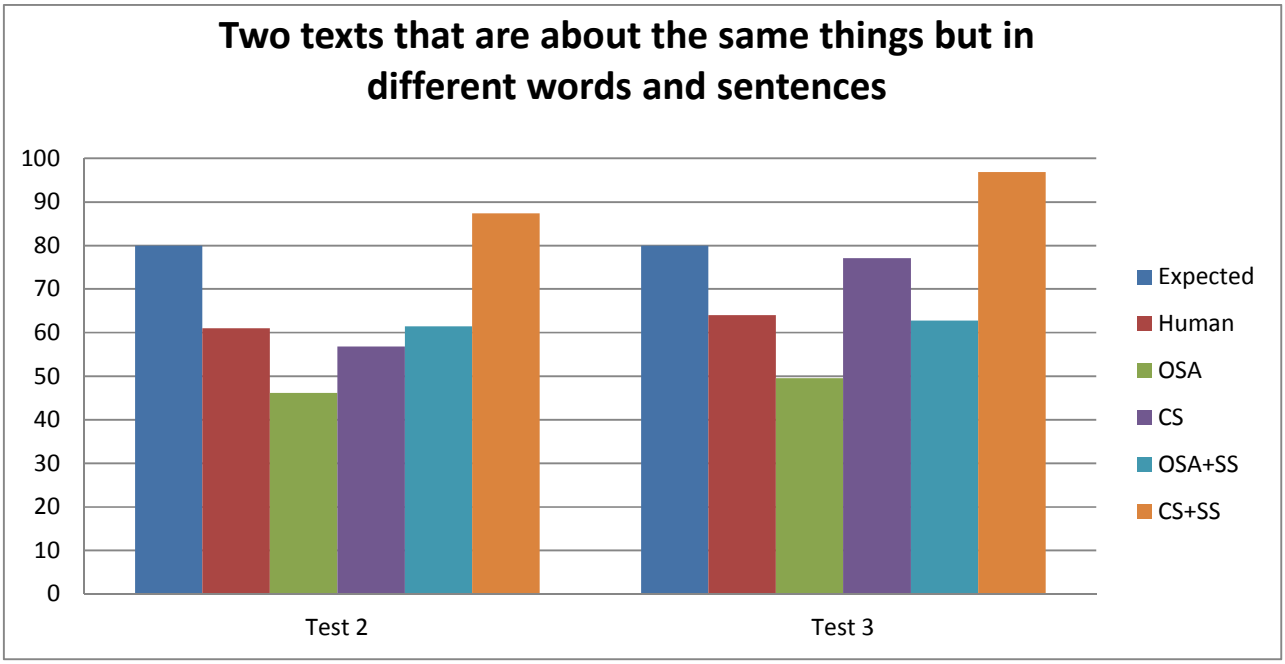

Two texts that are 75% or more alike, one uses part of phrases /sentences from the other
100
90
80
70
60
50
40
30
20
10
0
Test 1
Test 2
Test 3
Expected
Human
OSA
CS
OSA+SS
CS+SS

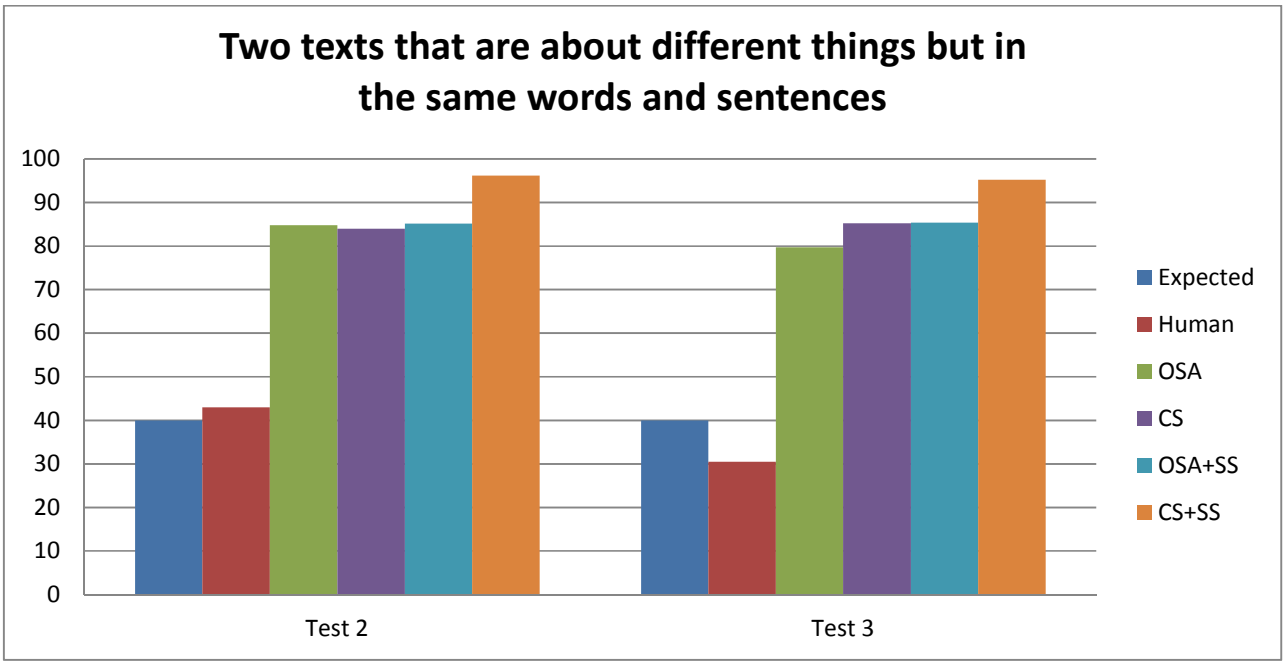

Two texts that are 50% or more alike, one uses part of phrases /sentences from the other
100
90
80
70
60
50
40
30
20
10
0
Test 2
Test 3
Expected
Human
OSA
CS
OSA+SS
CS+SS

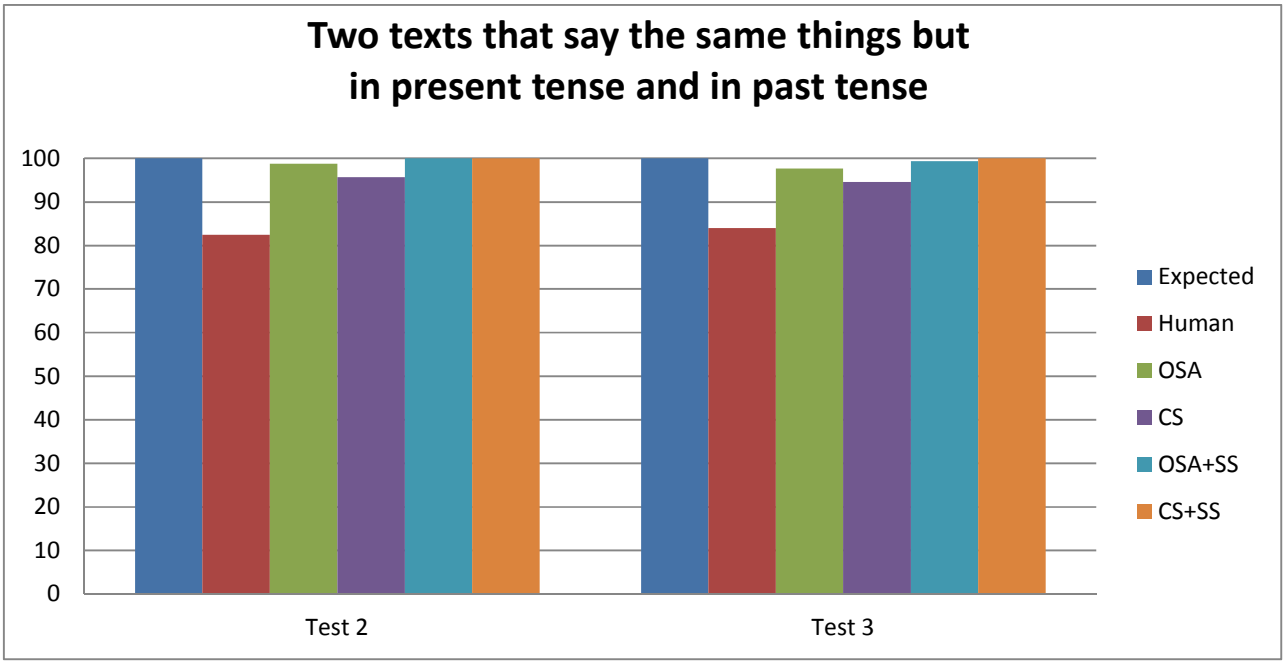

Two texts that are 25% or more alike, one uses part of phrases /sentences from the other
100
90
80
70
60
50
40
30
20
10
0
Test 2
Test 3
Expected
Human
OSA
CS
OSA+SS
CS+SS

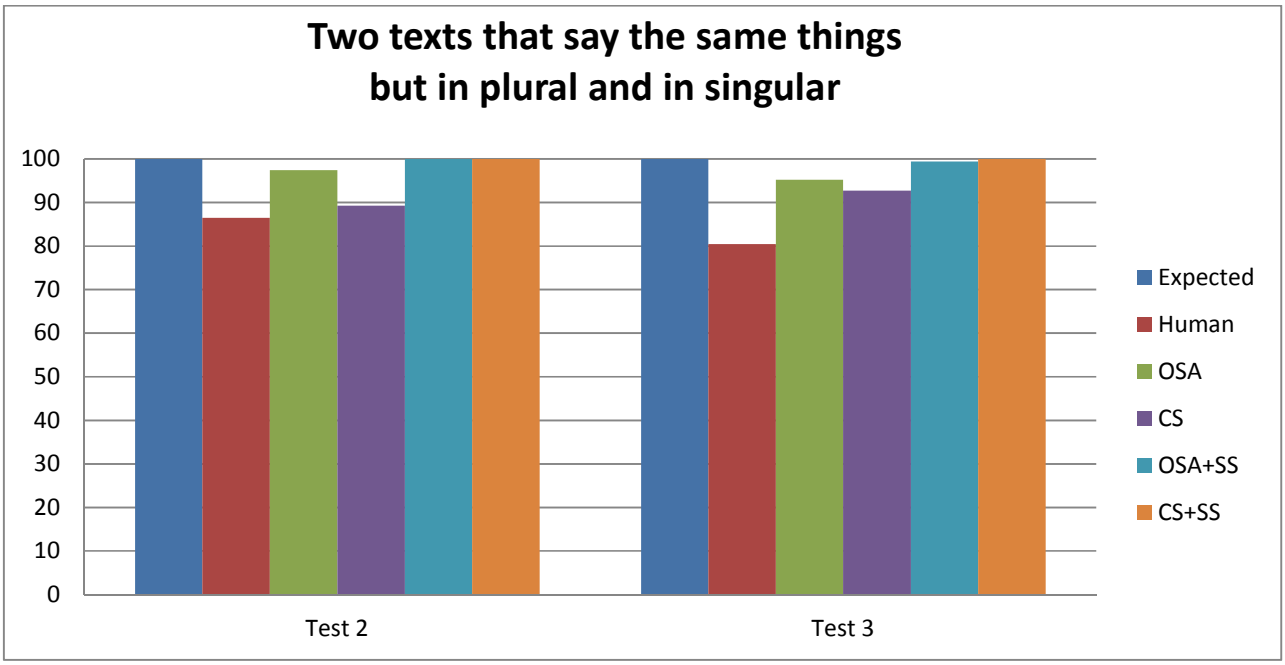

Two texts that are alike but with spelling mistakes (14 mistakes in a total of 64 words)
100
90
80
70
60
50
40
30
20
10
0
Test 2
Test 3
Expected
Human
OSA
CS
OSA+SS
CS+SS

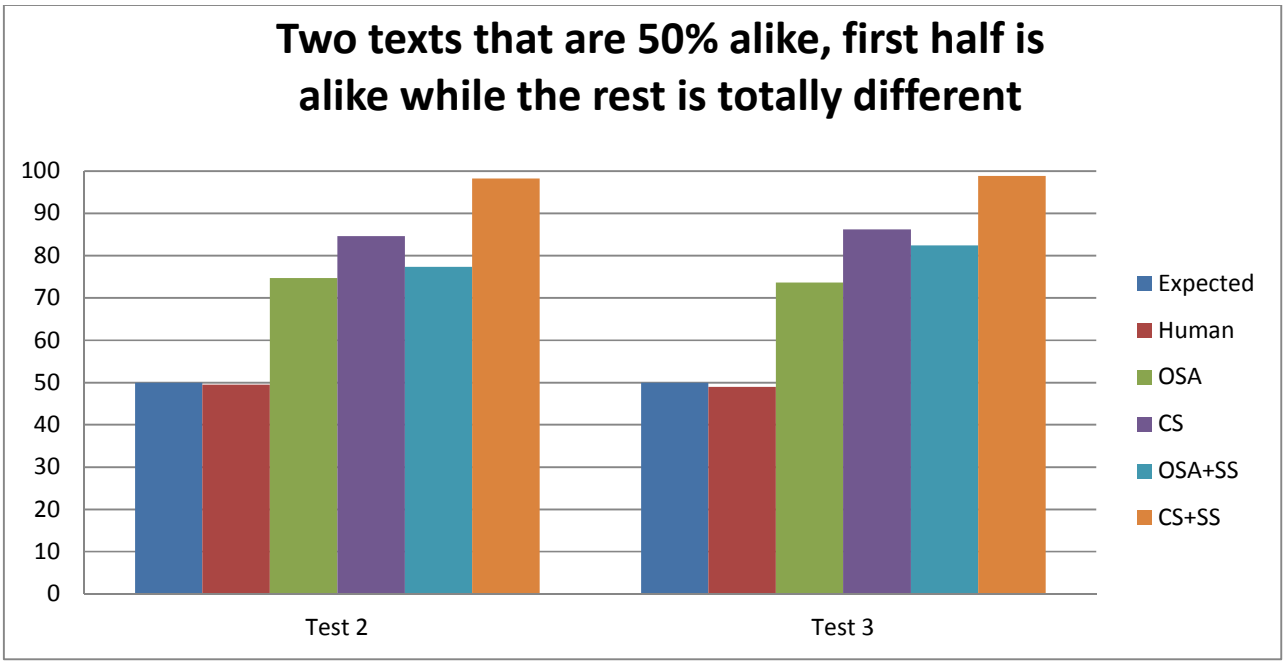

Two texts that are alike but with editing mistakes (5 “words” and 7 “letter placement in words” changes in a total of 64 words).
100
90
80
70
60
50
40
30
20
10
0
Test 2
Test 3
Expected
Human
OSA
CS
OSA+SS
CS+SS


Two texts that are about the same things but in different words and sentences
100
90
80
70
60
50
40
30
20
10
0
Test 2
Test 3
Expected
Human
OSA
CS
OSA+SS
CS+SS


Two texts that are about different things but in the same words and sentences
100
90
80
70
60
50
40
30
20
10
0
Test 2
Test 3
Expected
Human
OSA
CS
OSA+SS
CS+SS


Two texts that say the same things but in present tense and in past tense
100
90
80
70
60
50
40
30
20
10
0
Test 2
Test 3
Expected
Human
OSA
CS
OSA+SS
CS+SS


Two texts that say the same things but in plural and in singular
100
90
80
70
60
50
40
30
20
10
0
Test 2
Test 3
Expected
Human
OSA
CS
OSA+SS
CS+SS


Two texts that are 50% alike, first half is alike while the rest is totally different
100
90
80
70
60
50
40
30
20
10
0
Test 2
Test 3
Expected
Human
OSA
CS
OSA+SS
CS+SS

8.2 Performance/Running Time Tests:

These graphs are based on the sorting done from the raw DATA which is on the CD. The sorting can be seen in Appendix N.

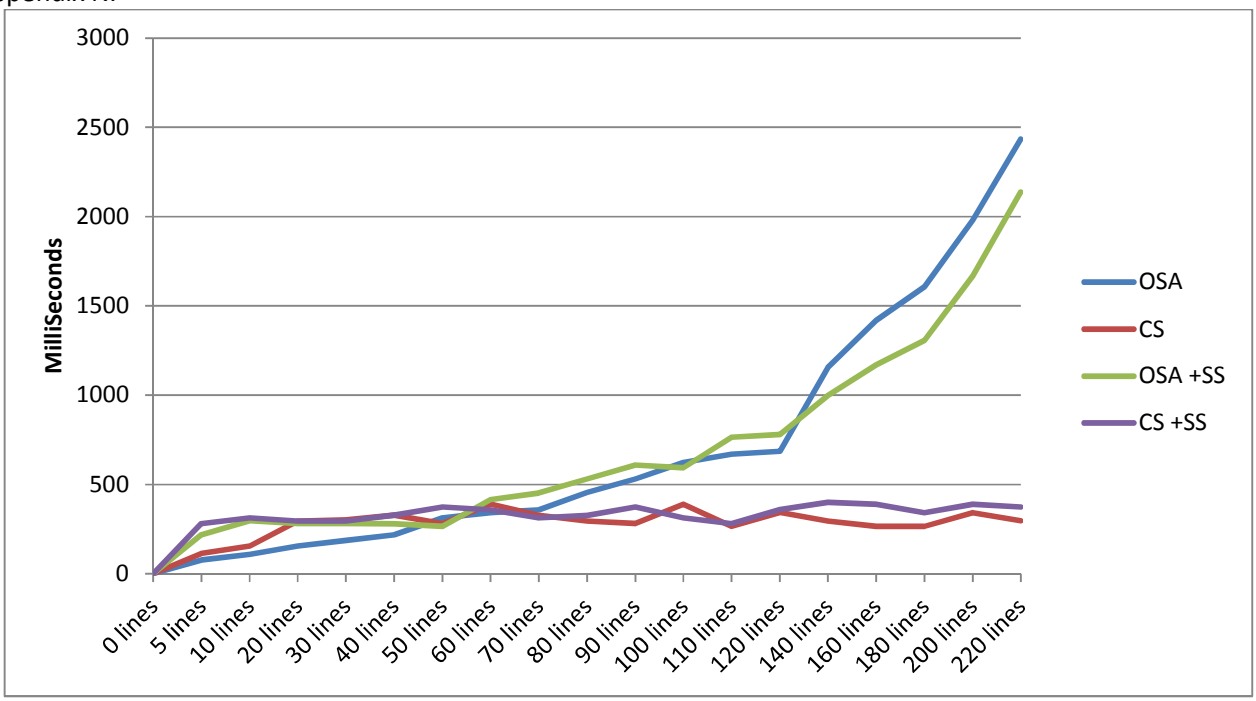


Figure 4: With restarting the computer before for each test to clean the cache.

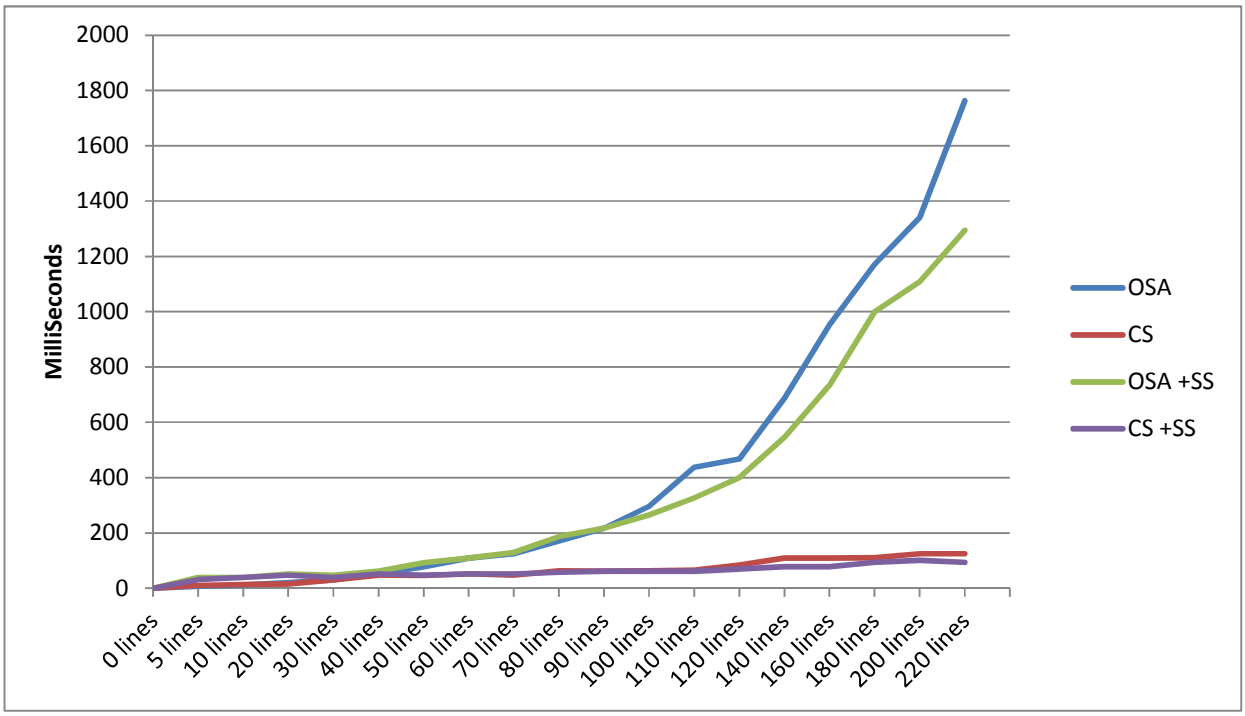


Figure 5: Without restarting the computer before for each test but instead restarting the application before each exercise and letting it run a few times.

CHAPTER 9

Discussion based on the Results

The journey has been long, starting from a little tour into similarity, going into language with its rules and grammars and then moving on to algorithms used for textual similarity. Each of the topics showing a new different world compared to the previous but still being connected. Widening out just to get knowledge and then an idea of what tool to use to make a good application to measure similarity between two texts. Limiting the application so it does not become too big while including tools that it can use quite effectively.
After finding this out and the algorithms to use, the structure and design for the application is made and just right after tested to make it better and bug free.
Making tests of similarity with the application, while expecting how it will behave and getting some results could be the end of all of it but the questions will still linger in the air: “Did the application work fine?” “Were it remarkable?” “Where are its limits?” “What about the performance on time and similarity?”

These questions and more will be discussed here starting with looking at the results from the previous chapter, starting with the exercise results for the similarity in all 3 tests.

It can be seen that in the first exercise: Two texts that look alike 100%.” gives and understandable results for all parts since the text is the same for both texts. The more interesting part is the next exercise “Two texts that are totally different 0%.” where all of the options were supposed to be 0%. Still knowing how the algorithms work it makes perfect sense. Optimal String Alignment Distance algorithm (OSA) edit the distance by changing characters so if there are sentences that look alike the other text, then according to OSA the “similarity” is found and it adds to its total. Cosine Similarity (CS) works with tokens. The first test will give 0% since there are no words that look alike in the sentence while the longer the texts the more tokens it will find that exist on both texts. Stemming and removal of Stop words seem to make it worse, making it seem that the texts do have something in common. The ones who come closest to the expectations are the human testers even with test 3 being long which should have killed the motivation.

Moving on the tricky exercise “Two texts that are totally alike but change in the sentence structure”, OSA behaved as said under “Expectations of the tests” section. Even when adding stemmer and stop word it still behaves badly while CS works fine with or without SS because of its token system. The human testers managed to handle it very well too but because of the text not being the exact same, the similarity in the eyes were not on 100%.

The next 3 exercises “Two texts that are 75% alike, one uses part of phrases/sentences from the other.”, “Two texts that are 50% alike, one uses part of phrases/sentences from the other.” and “Two texts that are 25% alike, one uses part of phrases/sentences from the other.” seem to be no problems for the 4 algorithm options while the human testers cannot seem to find the borderline specially not with the long text pieces in test 3. SS is a big minus in these exercises though. After stemming and removing the stop words the algorithm finds a higher similarity that it was expected. Especially in CS+SS which reaches near 100% because all the different words seem to be filtered away leaving only the same big words in both texts for it to reach that high on the scale. So instead of a little better as expected, CS+SS goes too far on the scale.

It does not come as a surprise to see that OSA and OSA+SS do well in “Two texts that are alike but with spelling mistakes” and also in “Two texts that are alike but with editing mistakes” since it should be their domain. The surprising ones are CS and CS+SS who should have gone a little down. Though it seems like the spelling and editing mistakes were not enough to shake the weight in CS, another reason could be that the spelling and editing mistakes are the same, making it go into the same tokens so no big effect happened with CS. To do that the mistakes should be different for each of the words so they would become their own tokens making many more different dimensions for CS to cover thus lowering the similarity.

The exercises “Two texts that are about the same things but in different words and sentences” and “Two texts that are about different things but in the same words and sentences” will react differently for each of the options. The first exercise gave a good result which is in order with the writer’s expectation and the human testers. OSA shows a not so good result alone but with SS it gives out a good result on the scale. The latter exercise is no surprise either. All of the options show a high similarity mainly because the same words are being used about a different topic. The writer and the human testers can see that the texts are talking about different topics and are not related but the application cannot.

It is easy to see that the options SS as told in “Expectations of the tests” shows what it is capable off. The stemmer is at work here, both in “Two texts that say the same things but in present tense and in past tense” and “Two texts that says the same thing but in plural and in singular”. While it was expected that OSA and CS alone would give a not so good result they still managed to get over 90%. With SS it is on 100% flat since the texts which were modified were stemmed. Thinking a bit about it makes sense that OSA get a high score and better than CS since it is an edit distance and it edits the difference easily to find the similarity.

Last exercise for similarity “Two texts that are 50% alike, first half is alike while the rest is totally different” results in the writer and the human testers being on 50% in nearly unison for both tests while all the algorithm options finds a similarity much higher than the expected 50% for OSA or 65% for CS. The answer for why this happens is the same as given in the exercise “Two texts that are totally different 0%.”

Moving to the exercises for the running time for the 4 algorithm options it can be decide fast that even with the stemmer and the stop word function, SS, OSA and CS are behaving like themselves making the 4 algorithms become 2 patterns, one for OSA and one for CS. The interesting thing is that constant K from “Expectation of the tests” where a guess on the runtime were made seems to appear quite clear in the results. Comparing both graphs for runtime it can be seen that while OSA is slowly going for a quadratic function as it is supposed to, CS’s linear function is covered by constant K, which is the time it takes to compile the code and run the application for the first time.

On the graph over the runtime without the constant K it is much easier to see the linear function of CS and CS+SS and the quadratic start of OSA. Sadly the limit of the application and its handling long text got to known here. The application can handle up to texts pieces with 220 lines where one line is 10 words long, so 2200 words in total. After that it won't give an input.

Another thing to notice which is not that clear on the graphs is that the time SS uses extra, compared to the OSA and CS alone, is in the long run won easily since the space for storage of text pieces and thus the time it takes the SS options to run their OSA or CS algorithm becomes smaller than without SS. This can clearly be seen on OSA with big text pieces while it is a bit harder to see it on CS.

9.1 Overall comparison of the 4 algorithm options

As shown the human testers manage to get a better result than the algorithms as long as the borderline to similarity is not blurry but a clear cut for them. The reason is clear, they do not have to wander under the same rules that each algorithm option is bound by. They can switch from one way of thinking to another if the situation asks for it, the algorithm options cannot.

Picking one of the 4 algorithm options would be a bit hard since it depends on which situation they are needed in. As shown before OSA is best to fix spelling mistakes, editing mistakes and also when it comes to singular/plural and past/present tense sentences. It is able to find the similarity easily and would be good to implement into an application if there is a need for that. On the other hand CS would not be as good here but when it comes to restructured sentences and phrases CS will handle this much better. OSA would get a bad result since it goes from one character to the next, from right to left while CS does not care as long as the tokens are the same. This leads to a big problem for CS which only notices the tokens and not in the context they are being used in. Another problem for CS is that the words that appear many time seems to have a bigger weight no matter how common they may be ("a", "the", "I") so this will only make CS measure a higher similarity score because of words like those when there is no real similarity.

This moves the topic to the SS, the stemmer and stop words removal, where both OSA and CS do well with SS. Removing all the small not so important words and stemming the words makes it much easier to find out how similar the two texts are. It works quite fine with the SS function added to it. CS does wonder too but mostly they are a bit too high on the scale than they need to be. SS in CS case remove all the words that are unpleasant for CS since they are used differently from text to text but since they are in the list they get removed and what is left are only the many words that goes again and again. In both OSA's and CS's case they both get worse the more words are added to the text pieces. While OSA goes on the quadratic path, CS starts to get bad results so in both cases the smaller the text pieces they run on. the better the results they will give.

9.2 Extensions

To make CS better so the result when dealing with big text pieces does not happen, it would be a wise idea to add TF-IDF to CS before running it on the text pieces. This way some words becomes heavier than other words and will affect the similarity score to be more precise. This way the CS which has a nice runtime compared to OSA, could be efficient and fast enough to find similarity.

Another idea could be to connect both OSA and CS with WordNet. It will make the runtime big but the similarity would be much more precise since it would reserve the idea of synonyms in play while the existing options are not capable of doing at this point.

CHAPTER 10

Conclusion

Similarity has and will always be a fascinating concept for humans. It will always be weighted and discussed. This paper too has discussed what Similarity could be and how to define it. Making it more specific, the Textual Similarity, language has been introduced and explained how complex it is.

Algorithms have been introduced and other tools to compare similarity of two texts. From all of these, 2 algorithms and some tools have been implemented into an application. The job of this application has been to find Similarity between different kinds of text structure so the algorithms could be compared and find out which was the best and fastest.

Tests have also been made to make the application better and without errors to cause the Similarity precision any faults.

The results have been that the first algorithm Optimal String Alignment Distance is a good algorithm to use with small text pieces and same structured text pieces. Cosine Similarity can handle big texts but not too big since it is a token system. It will result in errors where common words as “as”, “I” and “the” will get a too big an influence on its Similarity score.

It has been decide that while both algorithms are good at their own domains adding a stemmer and stop word removal to them is only a big plus. Both algorithms become faster and more effective.

This paper also hints that maybe more tools as WordNet or TF-IDF should be added to help the algorithms to become more precise even if the runtime will get affected by those tools.

Still it was a satisfactory result testing out the limitations of the two big algorithms used in the Information Retrieval field, getting them to understand better and conclude that both have their advantages and disadvantages and thus are still needed in today’s world.

Bibliography

(1: http://en.wikipedia.org/wiki/Similarity_%28psychology%29)

(2: http://en.wikipedia.org/wiki/Musical_similarity)

(3: http://en.wikipedia.org/wiki/Similarity_%28geometry%29)

(4: An Information-Theoretic Definition of Similarity by Dekang Lin from University of Manitoba)

(5: http://en.wikipedia.org/wiki/WordNet)

(6: http://en.wikipedia.org/wiki/Cosine_similarity)

(7: http://en.wikipedia.org/wiki/Tf%E2%80%93idf)

(8: http://en.wikipedia.org/wiki/Jaccard_index)

(9: http://en.wikipedia.org/wiki/Hamming_distance)

(10: http://en.wikipedia.org/wiki/Jaro%E2%80%93Winkler_distance)

(11: http://en.wikipedia.org/wiki/Levenshtein_distance)

(12: http://www.comp.lancs.ac.uk/computing/research/stemming/general/index.htm)

(13: http://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance)

(14: The strings are tested using Dr. E. Garcia's Levenshtein tool on page http://www.miislita.com/searchito/levenshtein-edit-distance.html)

(15: Network Security slides by Prof. Huy Kang Kim in course IMS584, fall 2010.)

(16: Java tip: How to get CPU, system, and user time for benchmarking by Dr. David Robert Nadeau, march 20, 2008 http://nadeausoftware.com/articles/2008/03/java_tip_how_get_cpu_and_user_time_benchmarking)

(17: This list is trimmed according to the programmer and taken from: http://armandbrahaj.blog.al/2009/04/14/list-of-english-stop-words/)

(18: New models in probabilistic information retrieval by C.J. van Rijsbergen, S.E. Robertson and M.F. Porter, 1980)

Figure 1: Page 303 from Cognition by Daniel Reisberg, 4th Edition

Figure 2: Page 317 from Cognition by Daniel Reisberg, 4th Edition

Appendix A

Implentation of the Damerau-Levenshtein algorithm after wikipeadia code in actionscript:

```
            public static int Damerau-Levenshtein( String a, String b, int alphabetLength){
                final int INFINITY = a.length() + b.length();
                int[][] H = new int[a.length()+2][b.length()+2];
                H[0][0] = INFINITY;
                for(int i = 0; i<=a.length(); i++) {
                  H[i+1][1] = i;
                  H[i+1][0] = INFINITY;
                }
                for(int j = 0; j<=b.length(); j++) {
                  H[1][j+1] = j;
                  H[0][j+1] = INFINITY;
                }
                int[] DA = new int[alphabetLength];
                Arrays.fill(DA, 0);
                for(int i = 1; i<=a.length(); i++) {
                  int DB = 0;
                  for(int j = 1; j<=b.length(); j++) {
                    int i1 = DA[b.charAt(j-1)];
                    int j1 = DB;
                    int d = ((a.charAt(i-1)==b.charAt(j-1))?0:1);
                    if(d==0) DB = j;
                    H[i+1][j+1] =
                      min(H[i][j]+d,
                          H[i+1][j] + 1,
                          H[i][j+1]+1,
                          H[i1][j1] + (i-i1-1) + 1 + (j-j1-1));
                  }
                  DA[a.charAt(i-1)] = i;
                }
                return H[a.length()+1][b.length()+1];
              }

              private static int min(int ... nums) {
                int min = Integer.MAX_VALUE;
                for (int num : nums) {
                  min = Math.min(min, num);
                }
                return min;
              }
              */
//public static void main(String[] args0){
//           String a = "I decided it was best to ask the forum if I was doing it right";
//           String b = "I thought I should ask the forum if I was doing it right";
//           System.out.println(Damerau-Levenshtein(a, b));
//}
```

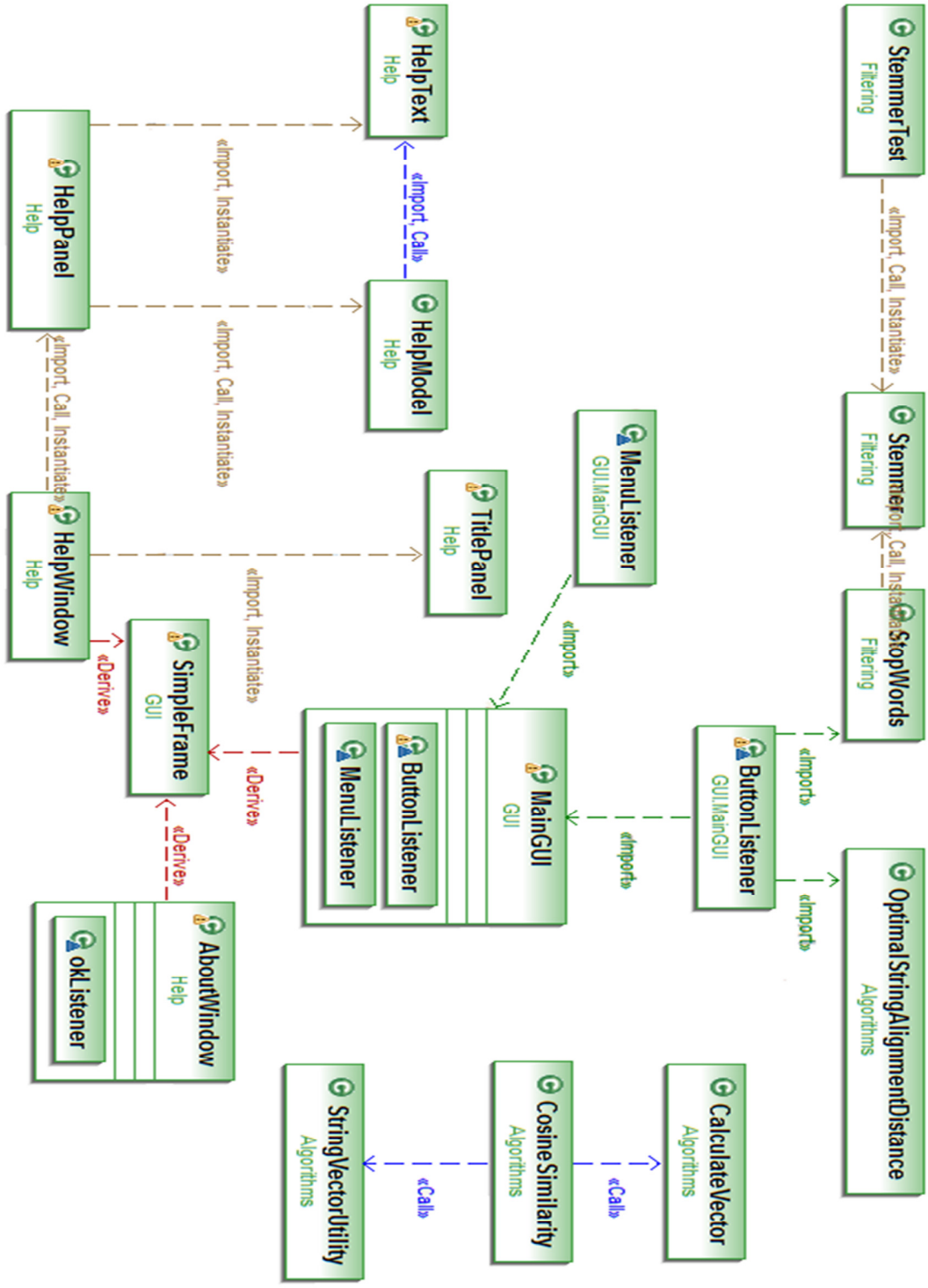
Appendix B

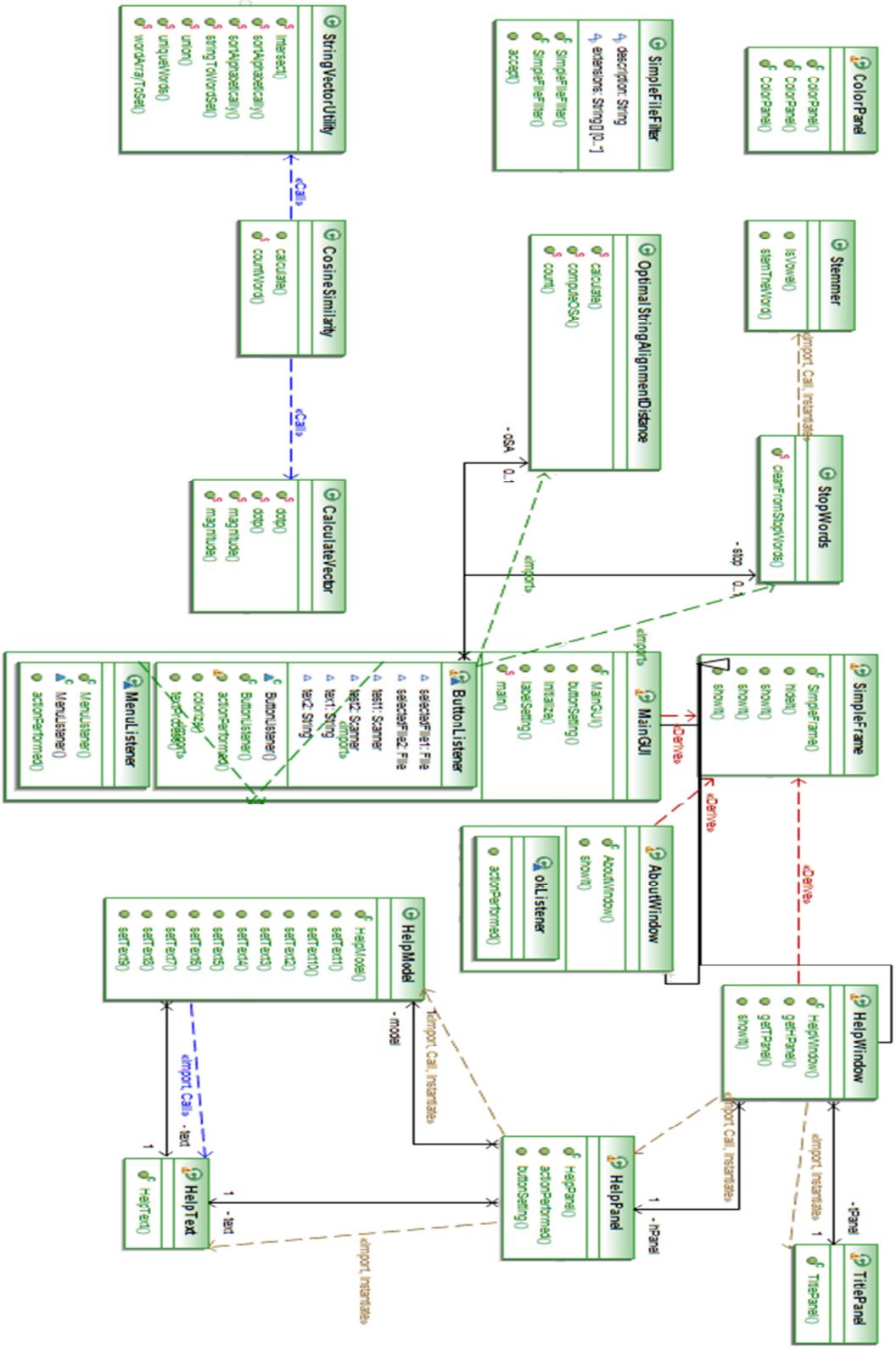
Pseudo code for Optimal String Algorithm taken from wikipeadia:

```
int OptimalStringAlignmentDistance(char str1[1..lenStr1], char str2[1..lenStr2])
   // d is a table with lenStr1+1 rows and lenStr2+1 columns
   declare int d[0..lenStr1, 0..lenStr2]
   // i and j are used to iterate over str1 and str2
   declare int i, j, cost
   //for loop is inclusive, need table 1 row/column larger than string length.
   for i from 0 to lenStr1
       d[i, 0] := i
   for j from 1 to lenStr2
       d[0, j] := j
   //Pseudo-code assumes string indices start at 1, not 0.
   //If implemented, make sure to start comparing at 1st letter of strings.
   for i from 1 to lenStr1
       for j from 1 to lenStr2
           if str1[i] = str2[j] then cost := 0
                                else cost := 1
           d[i, j] := minimum(
                                d[i-1, j  ] + 1,     // deletion
                                d[i  , j-1] + 1,     // insertion
                                d[i-1, j-1] + cost   // substitution
                              )
           if(i > 1 and j > 1 and str1[i] = str2[j-1] and str1[i-1] = str2[j]) then
               d[i, j] := minimum(
                                d[i, j],
                                d[i-2, j-2] + cost   // transposition
                               )


   return d[lenStr1, lenStr2]
```

Appendix C


StemmerTest
Filtering
«Import, Call, Instantiate»
Stemmer
Filtering
StopWords
Filtering
«Import»
«Import»
ButtonListener
GUI.MainGUI
«Import»
OptimalStringAlignmentDistance
Algorithms
«Import»
MainGUI
GUI
MenuListener
GUI.MainGUI
«Import»
ButtonListener
MenuListener
«Derive»
SimpleFrame
GUI
«Derive»
AboutWindow
Help
okListener
«Import, Instantiate»
HelpWindow
Help
«Import, Call, Instantiate»
TitlePanel
Help
«Import, Instantiate»
HelpPanel
Help
«Import, Call, Instantiate»
HelpModel
Help
«Import, Call»
HelpText
Help
«Import, Instantiate»
«Call»
CalculateVector
Algorithms
«Call»
CosineSimilarity
Algorithms
«Call»
StringVectorUtility
Algorithms


ColorPanel
SimpleFileFilter
StringVectorUtility
Stemmer
OptimalStringAlignmentDistance
CosineSimilarity
StopWords
CalculateVector
SimpleFrame
MainGUI
ButtonListener
MenuListener
AboutWindow
okListener
HelpWindow
HelpPanel
HelpModel
HelpText
TitlePanel
«import, Call, Instantiate»
«Call»
«import»
«Derive»
- stop
- gSA
- model
- text
- hPanel
- tPanel

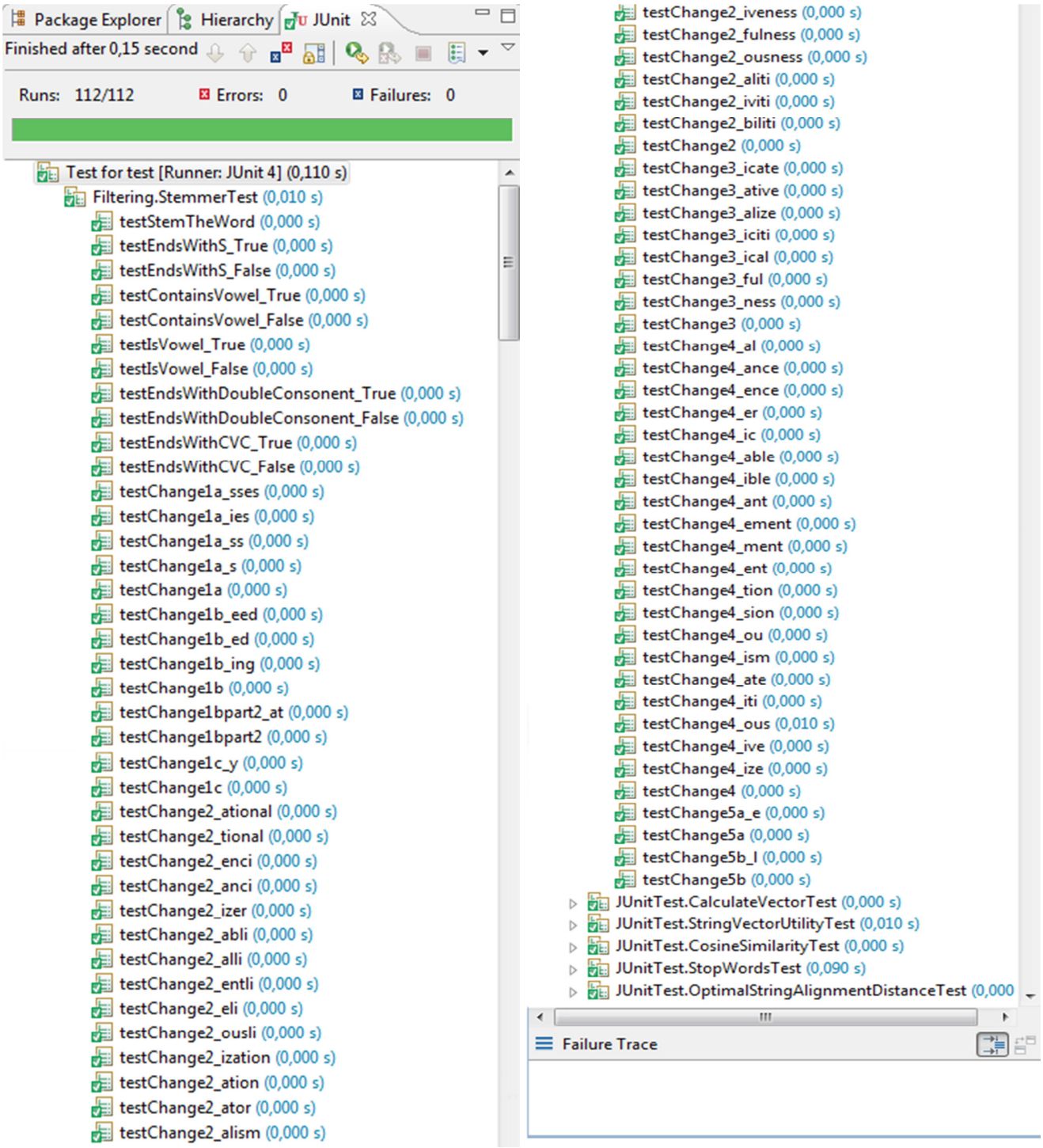
Appendix D

StopWord List:

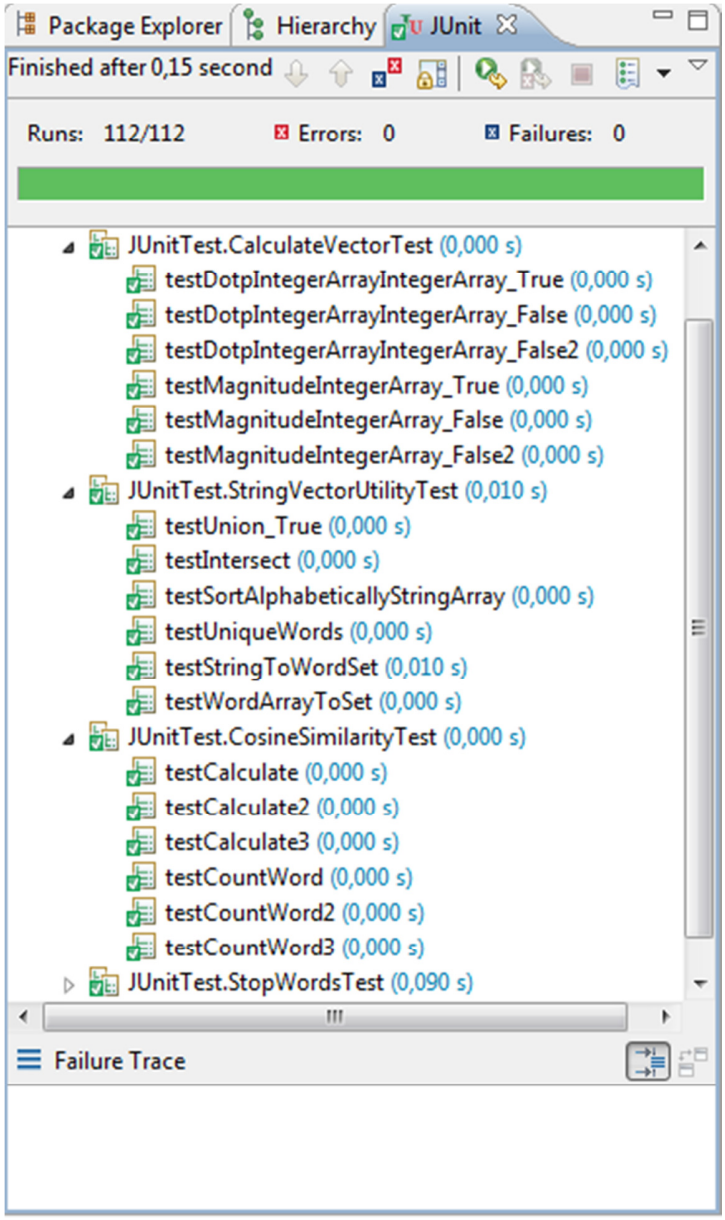
a able about above according accordingly across actually after afterwards again against ain't all allow allows almost alone along already also although always am among amongst an and another any anybody anyhow anyone anything anyway anyways anywhere apart appear appreciate appropriate are aren't around as aside ask asking associated at away awfully be became because become becomes becoming been before beforehand behind being believe below beside besides best better between beyond both brief but by c'mon came can can't cannot cant cause causes certain certainly changes clearly co com come comes concerning consider considering contain containing contains corresponding could couldn't currently definitely despite did didn't do does doesn't doing don't done down downwards during each edu eg either else elsewhere enough entirely especially etc even ever every everybody everyone everything everywhere exactly example except far few followed following follows for from further furthermore get gets getting given gives go goes going gone got gotten had hadn't happens hardly has hasn't have haven't having he he's hello hence her here here's hereafter hereby herein hereupon hers herself hi him himself his hither hopefully how howbeit however i'd i'll i'm i've if ignored immediate in inasmuch inc indeed indicate indicated indicates inner insofar instead into inward is isn't it it'd it'll it's its itself just keep keeps kept know knows known last lately later least less lest let let's like liked likely little look looking looks ltd mainly many may maybe me mean meanwhile merely might more moreover mostly much must my myself namely near nearly necessary need needs neither never nevertheless new next no nobody non none noone nor normally not nothing now nowhere obviously of off often oh ok okay on once ones only onto or other others otherwise ought our ours ourselves out outside over overall own particular particularly per perhaps placed please plus presumably probably provides que quite rather really reasonably regarding regardless regards relatively respectively right said same saw say saying says secondly see seeing seem seemed seeming seems seen self selves sensible sent serious seriously several shall she should shouldn't since so some somebody somehow someone something sometime sometimes somewhat somewhere soon sorry still sub such sup sure take taken tell tends th than thank thanks thanx that that's thats the their theirs them themselves then thence there there's thereafter thereby therefore therein theres thereupon these they they'd they'll they're they've think third this thorough thoroughly those though through throughout thru thus to together too took toward towards tried tries truly try trying twice under unfortunately unless unlikely until unto up upon us use used uses using usually various very want wants was wasn't way we we'd we'll we're we've welcome well went were weren't what what's whatever when whence whenever where where's whereafter whereas whereby wherein whereupon wherever whether which while whither who who's whoever whole whom whose why will with within without won't wonder would wouldn't yes yet you you'd you'll you're you've your yours yourself yourselves

Appendix E

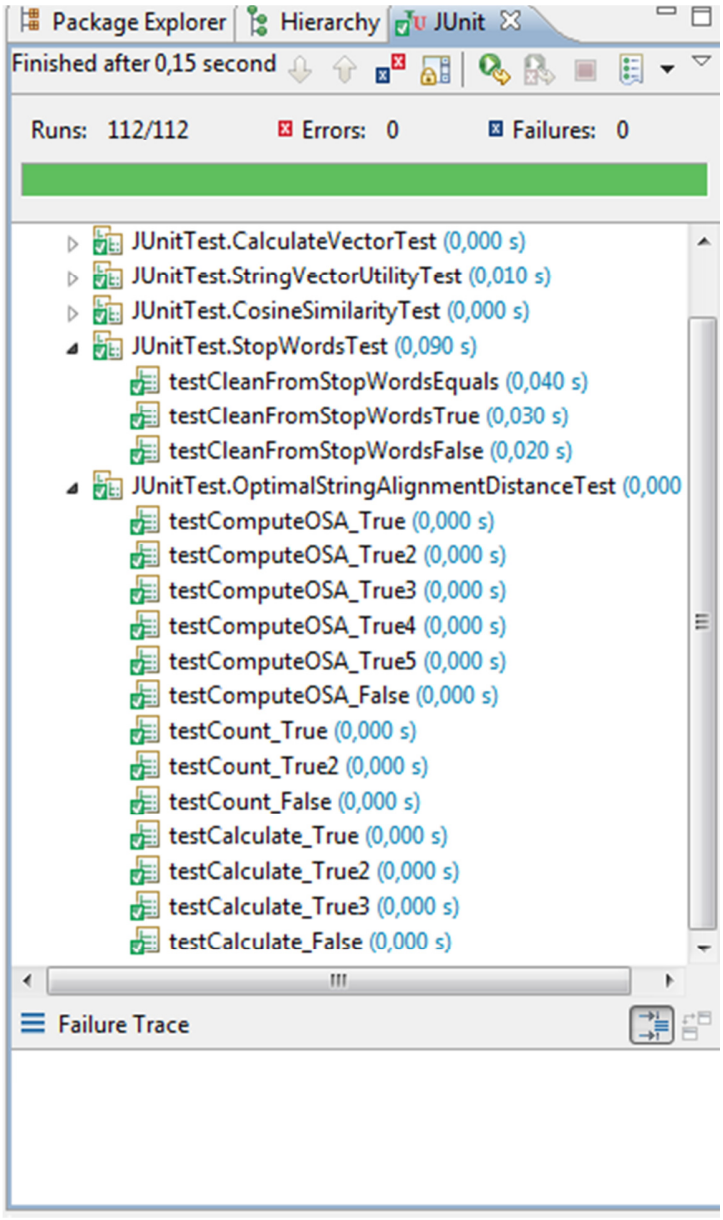
The white box tests made for the Stemmer Class.



The white box test for Test Cosince Similarity and StringVector and Calculate vector classes:



The white box test for the Optimal String Alignment distance class and the StopWord class:



Appendix F

Use-cases used to do the functional test by running through these scenarios.

use-case 1: Loading 2 files into the program

Goal:	To load 2 files into the program.
Requirement:	Txt files to compare should be available on the computer.
Operation accomplished:	If the file name is written on the text field just beside the buttons to load the file into the program.
Operation failed:	If the file name is not written on the text field just beside the buttons to load the file into the program.
Summary:	The program shows an Open Dialog box, hereafter the txt files are shown, the user chooses a file and it's uploaded into the program while the title is shown in the text field.
Restrictions:	None.
Actor:	User.
Priority:	High.
Used:	Often.

use-case 2: Finding the similarity between 2 txt files

Goal:	Find the similarity for 2 txt files
Requirement:	Txt files uploaded into the program. Press on one of the similarity algorithm buttons.
Operation accomplished:	The label under "Similarity" for that button changes.
Operation failed:	The labels under "Similarity" don't change.
Summary:	By pressing one of the algorithm buttons, the program runs the two texts and finds out how similar they are. It shows it on the screen.
Restrictions:	None.
Actor:	User.
Priority:	High.
Used:	Often.

use-case 3: Finding different similarities between 2 txt files

Goal:	Find different similarities for 2 txt files
Requirement:	Txt files uploaded into the program. Press the similarity algorithm buttons, more than just one button.
Operation accomplished:	The labels under "Similarity" changes.
Operation failed:	At l-east label under "Similarity" doesn't change.
Summary:	By pressing the algorithm buttons, the program runs the two texts and finds out how similar they are. It shows it on the screen under each of the algorithms run on.

Restrictions:	Wait until one algorithm has finished before pressing on the next button.
Actor:	User.
Priority:	High.
Used:	Often.

use-case 4: Finding the time to find similarity between 2 txt files

Goal:	Find the time it takes to find the similarity for 2 txt files
Requirement:	Txt files uploaded into the program. Press the chosen similarity algorithm button.
Operation accomplished:	The label under “Time” changes.
Operation failed:	The label under “Time” doesn’t change.
Summary:	By pressing the algorithm buttons, the program runs the two texts and finds out how similar they are while running the time the process takes. It shows the time on the screen under the “Time” label.
Restrictions:	None.
Actor:	User.
Priority:	High.
Used:	Often.

use-case 5: Finding different times for the similarities between 2 txt files

Goal:	Find all the times for the different similarities for 2 txt files
Requirement:	Txt files uploaded into the program. Press the similarity algorithm buttons, more than just one button.
Operation accomplished:	The labels under “Time” changes.
Operation failed:	At least one label under “Time” doesn’t change.
Summary:	By pressing the algorithm buttons, the program runs the two texts and finds out how similar they are while running the time the process takes. It shows the time on the screen under each of the algorithms run on them.
Restrictions:	Wait until one algorithm has finished before pressing on the next button.
Actor:	User.
Priority:	High.
Used:	Often.

use-case 6: Resetting tests

Goal:	Reset the system.
Requirement:	Txt files uploaded into the program. Similarity and Time found for some or all Similarity algorithms. Press on “New” under “File” menu.

Operation accomplished:	The text fields are empty while the labels “Similarity” and “Time” are back to “0%”.
Operation failed:	One of the text fields isn’t empty or one labels “Similarity” and “Time” isn’t back to “0%”.
Summary:	By pressing on the “New”option under “File”, the program cleans it’s data, reset the labels and the text fields so a new comparison can be be done.
Restrictions:	None.
Actor:	User.
Priority:	High.
Used:	Often.

use-case 7: Help function to guide the user

Goal:	Let the user be guided by the applications help function.
Requirement:	None.
Operation accomplished:	The user who’s confused or have questions about how to use the program, finds the “Help” option under “HELP” menu and uses the function or find the answer to use the program.
Operation failed:	The user can’t find the answer to the question in the Help function.
Summary:	By pressing on “Help” in the menu “HELP” the user finds a Help window which the user can navigate through to get explanations of on how to use the program.
Restrictions:	None.
Actor:	User.
Priority:	High.
Used:	Often.

use-case 8: Closing the Application

Goal:	Close the program.
Requirement:	Press on the option “Close” under menu “File”.
Operation accomplished:	The program exits.
Operation failed:	The program doesn’t exit.
Summary:	By pressing on menu “File” and then option “Close” the user can quite the program while resetting everything.
Restrictions:	None.
Actor:	User.
Priority:	High.
Used:	Often.

Appendix G

Result of functional testing done using the use-cases.

Nr.	*Input*	*Use-Case*	*Result*
1	Legal	*Loading 2 files into the program*	Success
1	Illegal	*Loading 2 files into the program*	Success*
2	Legal	*Finding the similarity between 2 txt files*	Success
2	Illegal	*Finding the similarity between 2 txt files*	Success
3	Legal	*Finding different similarities between 2 txt files*	Success
3	Illegal	*Finding different similarities between 2 txt files*	Success
4	Legal	*Finding the time to find similarity between 2 txt files*	Success
4	Illegal	*Finding the time to find similarity between 2 txt files*	Success
5	Legal	*Finding different times for the similarities between 2 txt files*	Success
5	Illegal	*Finding different times for the similarities between 2 txt files*	Success
6	Legal	*Resetting tests*	Success*
6	Illegal	*Resetting tests*	Success*
7	Legal	*Help function to guide the user*	Success
7	Illegal	*Help function to guide the user*	Success
8	Legal	*Close the Application*	Success
8	Illegal	*Close the Application*	Success

Appendix H

The usability scheme form given to the tester:

Questions to be answered after using the program:

Age:

Gender:

Computer experience:

3 good things in the program:

3 bad things in the program:

Things to add to the program:

Appendix H

Overview over the testers for usability test and similarity test:

Nr.	Age	Gender	Occupation	Computer-experience
1	17	Girl	Student	Experienced
2	14	Girl	Student	Average
3	21	Guy	Student	Experienced
4	26	Girl	Worker	Average
5	25	Guy	Student	New beginner
6	26	Girl	Student	Experienced
7	25	Girl	Workless	Average
8	25	Girl	Worker	Experienced
9	34	Girl	Housewife	Average
10	22	Girl	Student	New beginner
11	20	Guy	Worker	Average
12	18	Girl	Student	Experienced
13	18	Girl	Student	Experienced
14	17	Guy	Student	Experienced
15	24	Girl	Student	Average
16	25	Girl	Programmer	Experienced
17	24	Guy	Programmer	Experienced
18	28	Girl	Teacher	Experienced
19	22	Guy	Programmer	Experienced
20	26	Guy	Worker	New beginner
21	25	Guy	Worker	Experienced
22	25	Guy	Worker	Experienced
23	25	Guy	Workless	Experienced

Appendix I

Positive feedback from the usability test:

Nr	Good things in the application
1	Simple interface and not too complicated
2	Help function
3	More than one option to find similarity
4	Txt are the only ones shown - Filtered

Appendix J

Negative feedback from the usability test:

Nr	Bad things in the application
1	Can only run txt files
2	Too simple – You can't reset the stats
3	While looking for files the type of files can be changes from "txt only" to "all files" if wanted.
4	Too Simple – You don't know if the similarity value is enough to say how similar the texts are.
5	Help function should tell about every function.

Appendix K

Suggestions to improve the application:

Nr	Suggestions for a better application
1	Colours for Similarity after some standards
2	"New" option in the menubar
3	SS to be separate
4	SS to be added via radiobuttons/checkbuttons

Appendix L

The structure of the test used for the Similarity test with quotes.

Phrases on 1 lines.

Before we start a quote by John Stone: The art of medicine consists in amusing the patient while nature cures the disease.

1. Two texts that look alike 100%.
2. Two texts that are totally different 0%.
3. Two texts that are 75% or more alike, one uses part of phrases/sentences from the other.
4. Two texts that are alike but different in structure.

The text pieces used are self-made for this part of the test.

Short Phrases on 6-7 lines.

Quote by Walt Whitman in Song of Myself: I believe in the flesh and the appetites, seeing, hearing, feeling are miracles. And each part of me is miracle.

1. Two texts that look alike 100%.
2. Two texts that are totally different 0% .
3. Two texts that are 50% alike, one uses part of phrases/sentences from the other.
4. Two texts that are 25% alike, one uses part of phrases/sentences from the other.
5. Two texts that are 75% alike, one uses part of phrases/sentences from the other.
6. Two texts that are totally alike but changes in the sentence structure.
7. Two texts that are alike but with spelling mistakes (14 mistakes in a total of 64 words).
8. Two texts that are alike but with editing mistakes (5 "words" and 7 "letter placement in words" changes in a total of 64 words).
9. Two texts that are about the same things but in different words and sentences.
10. Two texts that are about different things but in the same words and sentences.
11. Two texts that say the same things but in present tense and in past tense.
12. Two texts that say the same things but in plural and in singular.
13. Two texts that are 50% alike, first half is alike while the rest is totally different.

Texts used chapter 9(page 117), 15(page 209) and 17(page 231) from Behavior & Medicine by Danny Wedding.

Longer Phrases on 20-24 lines.

Quote by Leo Tolstoy: One can live magnificently in this world, if one knows how to work and how to love, to work for the person one loves, and to love one's work.

1. Two texts that are alike but with spelling mistakes (33 mistakes in a total of 197 words).
2. Two texts that are alike but with editing mistakes (18 "words" and 18 letter placement changes in a total 197 words).
3. Two texts that are 50% alike, one uses part of phrases/sentences from the other.
4. Two texts that are 25% alike, one uses part of phrases/sentences from the other.
5. Two texts that are 75% alike, one uses part of phrases/sentences from the other.
6. Two texts that are totally alike but changes in the sentence structure.
7. Two texts that say the same things but in present tense and in past tense.
8. Two texts that are about the same things but in different words and sentences.
9. Two texts that are about different things but in the same words and sentences.
10. Two texts that look alike 100%.
11. Two texts that are totally different 0%.
12. Two texts that say the same things but in plural and in singular.
13. Two texts that are 50% alike, first half is alike while the rest is totally different.

Texts used chapter 5(page 100), 6(page 114) and 12(page 274) from Introduction to the Human Body by G.J. Tortora.

Quote by Sir William Osler: Let us emancipate the student, and give him time and opportunity for the cultivation of his mind, so that in his pupilage he shall not be a puppet in the hands of others, but rather a self-relying and reflecting being.

Appendix M

The only guidance given to the testers for Similarity test:

There are 3 tests and all 3 should be taken. They are cut into 3 pieces to make it easier for you to know how far you reached. I wanted to make a web based test but I used the whole day yesterday failing to find out how. Thus we are doing it this way.

The tests are the same, meaning you do the same thing as you do in one of the tests.

Each test is cut into questions. For each questions you put your X on the answer sheet that comes with the tests. So 3 tests with 3 answer sheets.

With each question there's 2 pieces of tests. One is original and the other is edited. They are labeled to make it easier to know which is which.

Here's how you do the test in steps:

1. Read the first text (the original)
2. Read the second text (the edited)
3. Decide how similar the texts are in your opinion.
4. Put a X in the answer sheet for the specific test under the box (from 0% to 100%) you find matching with your decision about the similarity for the two texts.
5. Do the same for question in how similar they are (*totally different* to *totally similar*)
6. Now move to the next question in the tests and repeat from step 1.

Let me give you an example:

Question 3:
Text 1(original)
I want Glass Mask

Text 2(edited)
I want more Glass Mask

I find these two texts to be similar...about 75%. I go to the answer sheets and find Question number 3's place. Round up the 75% to "80%" since in my opinion it's closer...and then put an X into that box. Then I move a little down on the answer sheet and put my X on the "Mostly Similar" out of Question 3. After that I go to the next question until I'm done. I remember to save in the middle of the tests a few times and then at the end.

The Original text files can be found on the CD in the folder '*Test 1 exercises text'*, '*Test 1 exercises text*' and '*Test 1 exercises text'.*

Appendix N

The raw DATA which is on the CD in a nice sorted way according to each tests. These stats show similarity and time in nanoseconds for each exercise under each test.

Test 1:

Runtime in Nanoseconds for each algorithm in each exercise.

	OSA	CS	OSA+SS	CS+SS
Two texts that look alike 100%.	691762	1342719	31338708	37143427
Two texts that are totally different 0%.	741421	1582160	26946769	35343383
Two texts that are 75% or more alike, one uses part of phrases/sentences from the other.	508909	1057853	31942700	32990160
Two texts that are alike but different in structure.	576278	1356583	27934273	33126548

Similarity in percentage for each algorithm and human in each exercise.

	Expected	Human	OSA	CS	OSA+SS	CS+SS
Two texts that look alike 100%.	100	100	100	100	100	100
Two texts that are totally different 0%.	0	0	43,75	0	35,48	0
Two texts that are 75% or more alike, one uses part of phrases /sentences from the other.	75	74,5	82,61	89,44	76,47	86,47
Two texts that are alike but different in structure.	80	67,5	60,87	100	100	100

Test 2:

Runtime in Nanoseconds for each algorithm in each exercise.

	OSA	CS	OSA+SS	CS+SS
Two texts that look alike 100%.	2104566	3973153	35314135	34546672
Two texts that are totally different 0%.	2634258	4860916	32145688	33054848
Two texts that are 50% or more alike, one uses part of phrases/sentences from the other.	272139	324573	28876096	34948182
Two texts that are 25% or more alike, one uses part of phrases/sentences from the other.	2971865	6660136	29290198	34999875
Two texts that are 75% or more alike, one uses part of phrases/sentences from the other.	2629254	5202690	31000175	32584272
Two texts that are totally alike but changes in the sentence structure.	2978410	5913319	32616223	34763127
Two texts that are alike but with spelling	2982644	4506688	30182527	34538015

mistakes (14 mistakes in a total of 64 words).				
Two texts that are alike but with editing mistakes (5 "words" and 7 "letter placement in words" changes in a total of 64 words).	2800174	6157767	31880571	33958182
Two texts that are about the same things but in different words and sentences.	2298962	4460493	32933630	37705725
Two texts that are about different things but in the same words and sentences.	2905652	6199342	29573526	35498779
Two texts that say the same things but in present tense and in past tense.	2732423	4649506	33490844	31809739
Two texts that say the same things but in plural and in singular.	3134318	5496026	33340327	33364194
Two texts that are 50% alike, first half is alike while the rest is totally different.	2964166	5737394	36098157	37971356

Similarity in percentage for each algorithm and human in each exercise.

	Expected	Human	OSA	CS	OSA+SS	CS+SS
Two texts that look alike 100%.	100	100	99,69	99,59	100	100
Two texts that are totally different 0%.	0	12,5	47,79	40,83	46,59	62
Two texts that are 50% or more alike, one uses part of phrases/sentences from the other.	50	55	57,25	81,12	70,65	97,69
Two texts that are 25% or more alike, one uses part of phrases/sentences from the other.	25	26,5	43,72	67,94	63,15	95,62
Two texts that are 75% or more alike, one uses part of phrases/sentences from the other.	75	65	84,63	89,96	75,4	98,63
Two texts that are totally alike but changes in the sentence structure.	80	86,5	66,18	89,74	71,88	98,44
Two texts that are alike but with spelling mistakes (14 mistakes in a total of 64 words).	100	79,5	96,08	82,35	96,11	95,67
Two texts that are alike but with editing mistakes (5 "words" and 7 "letter placement in words" changes in a total of 64 words).	100	76	91,67	87,26	90,91	95,82
Two texts that are about the same things but in different words and sentences.	80	61	46,18	56,79	61,45	87,39
Two texts that are about different things but in the same words and sentences.	40	43	84,8	83,95	85,23	96,21
Two texts that say the same things but in present tense and in past tense.	100	82,5	98,76	95,68	100	100
Two texts that say the same things but in plural and in singular.	100	86,5	97,44	89,29	100	100

Two texts that are 50% alike, first half is alike while the rest is totally different.	50	49,5	74,69	84,55	77,39	98,2

Test 3:

Runtime in Nanoseconds for each algorithm in each exercise.

	OSA	CS	OSA+SS	CS+SS
Two texts that look alike 100%.	29713318	22310685	48105957	37895866
Two texts that are totally different 0%.	33302999	28082275	50458017	45338535
Two texts that are 50% or more alike, one uses part of phrases/sentences from the other.	31398100	27198875	48398875	43552290
Two texts that are 25% or more alike, one uses part of phrases/sentences from the other.	24255256	27907326	48293397	41893133
Two texts that are 75% or more alike, one uses part of phrases/sentences from the other.	34070458	27449614	53236222	38625633
Two texts that are totally alike but changes in the sentence structure.	24733756	18698043	46773209	35364021
Two texts that are alike but with spelling mistakes (14 mistakes in a total of 64 words).	26943201	20979134	48008179	32552727
Two texts that are alike but with editing mistakes (5 "words" and 7 "letter placement in words" changes in a total of 64 words).	25456900	17410241	42133036	39767119
Two texts that are about the same things but in different words and sentences.	26057047	21265473	42308440	37564216
Two texts that are about different things but in the same words and sentences.	26398504	19669052	46029445	33976021
Two texts that say the same things but in present tense and in past tense.	28210503	23059763	54071221	35734510
Two texts that say the same things but in plural and in singular.	28576169	22711805	47215178	37052050
Two texts that are 50% alike, first half is alike while the rest is totally different.	29914649	23730389	53993034	42536081

Similarity in percentage for each algorithm and human in each exercise.

	Expected	Human	OSA	CS	OSA+SS	CS+SS
Two texts that look alike 100%.	100	100	100	100	100	100
Two texts that are totally different 0%.	0	0	47,63	57,04	60,14	94,17
Two texts that are 50% or more alike, one uses part of phrases/sentences from the other.	50	45	73,49	85,79	79,26	99,02

Two texts that are 25% or more alike, one uses part of phrases/sentences from the other.	25	29,5	57,1	75,27	68,6	97,94
Two texts that are 75% or more alike, one uses part of phrases/sentences from the other.	75	51,5	85,03	94,92	83,27	99,65
Two texts that are totally alike but changes in the sentence structure.	80	77,5	61,18	97,17	65,67	99,74
Two texts that are alike but with spelling mistakes (14 mistakes in a total of 64 words).	100	79	97,71	95,6	96,96	99,06
Two texts that are alike but with editing mistakes (5 "words" and 7 "letter placement in words" changes in a total of 64 words).	100	73,5	88,91	95,67	88,02	99,49
Two texts that are about the same things but in different words and sentences.	80	64	49,59	77,12	62,77	96,84
Two texts that are about different things but in the same words and sentences.	40	30,5	79,7	85,24	85,43	95,23
Two texts that say the same things but in present tense and in past tense.	100	84	97,64	94,6	99,35	99,99
Two texts that say the same things but in plural and in singular.	100	80,5	95,22	92,71	99,46	99.99
Two texts that are 50% alike, first half is alike while the rest is totally different.	50	49	73,58	86,22	82,44	98,87

Appendix O

The raw DATA which is on the CD in a nice sorted way according to the two modes.

The restarting and running the algorithm time. This were based on average 10 runs since restarting the computer and then run it takes too much time doing it on the same computer to get best output.

	OSA	**CS**	**OSA+SS**	**CS+SS**
0 lines	0	0	0	0
5 lines	78	114	218	281
10 lines	109	156	297	312
20 lines	156	296	281	296
30 lines	187	303	281	296
40 lines	218	328	280	328
50 lines	312	281	265	374
60 lines	343	390	416	358
70 lines	358	327	452	312
80 lines	456	296	530	327
90 lines	530	281	609	374
100 lines	624	390	593	312
110 lines	671	265	764	281
120 lines	686	344	780	359
140 lines	1155	296	999	400
160 lines	1419	265	1170	390
180 lines	1607	265	1307	343
200 lines	1981	343	1669	390
220 lines	2434	297	2138	374

The let the application run and then running the algorithm time. This were based on running the program a few time and then take the average of 25 runs without restarting the computer but only the application for each exercise. The test was done on the same computer.

	OSA	**CS**	**OSA+SS**	**CS+SS**
0 lines	0	0	0	0
5 lines	7	10	40	32
10 lines	12	14	40	40
20 lines	20	16	52	47
30 lines	31	31	47	40
40 lines	52	47	62	52
50 lines	78	46	93	47
60 lines	109	52	109	52
70 lines	125	47	130	52
80 lines	171	63	187	58
90 lines	218	62	218	62
100 lines	296	63	265	62
110 lines	437	65	327	62
120 lines	468	85	400	70
140 lines	687	109	546	78
160 lines	952	109	733	78
180 lines	1170	110	999	94
200 lines	1341	125	1108	102
220 lines	1763	125	1294	94

Appendix P

The manual to the application: Textual Similarity Demo

Press on Text1: to add the file you want to compare with another.

Press on Text2: to add the file you want to compare with the first file.

Remember both should be txt files since that's the only file format the program supports.

Now you can press on OSA Distance, Cosine Similarity, OSA Distance +SS and Cosine Similarity to apply the type of option on your files to find similarity.

Press on New in File to reset.

Press on Close in File to close the application.

Press on About in HELP to find out about the maker of the application.

Press on Help in HELP to find out more about the program in details.