Poznan University of Technology

Institute of Computing Science
Laboratory of Computing Systems

Masterthesis

The construction of application-specific and index supported string similarity predicates

---

Fundamentals and Design of Similarity Queries

Student:
André Reckhemke, No.: 3008577
12th September 2005

Attendants:
Professor Dr. habil. Tadeusz Morzy
Dr. Eike Schallehn, University of Magdeburg

Poznan University of Technology
Institute of Computing Science
ul. Piotrowo 3a
60-965 Poznan

Reckhemke, André:
The construction of application-specific and index supported string similarity predicates
Thesis, Poznan University of Technology, 2005.

Acknowledgment

The present masterthesis is the last part of the polish post-graduate study in computer science - degree as 'magister inzynier'. The program is a cooperation of the University of Poznan and the University of Applied Science in Wolfsburg.

The subject of the masterthesis was defined jointly in close collaboration with the University of Magdeburg - Institute of Technical and Business Information Systems.

Foreword

This project would never have been completed successfully without the help of several people. First of all I want to say thank Prof. Dr. Morzy and Dr. Schallehn for their support in research questions. Both have helped me to define the subject of the masterthesis and to structure it. I also want to thank the University of Applied Science in Wolfsburg -Prof. Dr. Harbusch- for the organization of the program. A special gratitude is for my parents without whom I would never have had the possibility of conducting my studies. At the same place I would like to thank Claudia Schulze and Bernhard Schenkel for their support.

The present masterthesis gave me the chance to get an insight in the area of index supported string similarity predicates. This subject combines linguistic questions with up-to-date techniques of the computer science. It makes it possible to improve my theoretical knowledge of tree structures and index structures in reference to computer based linguistic research areas.

Abstract

In times of increasing computer readable data as well as a further worldwide expansion of the internet it is more and more important to find context relevant information in large data stocks.

This masterthesis presents a complete overview about proven methods of the fields of information technology as well as computational linguistics, which can be combined to achieve better results in search engines or systems for information retrieval. It clarifies, that the homogenization of plain text reduces the volume of index structures and concurrently increases the quality of hitlists. Furthermore it shows the careful and context dependent dealing with abbreviation, acronyms and synonyms.

The core of this work is a general approach to support stepwise application-specific and index supported string similarity predicates. It uses the hybrid ternary search trie as one of the fastest index structure for strings. Tries guarantee best results for exact matching as well as inexact matching in preprocessed data and can be used for external data storage. Especially ternary hybrid search tries are easy adaptable for common strings and provide best average results for inexact matching without any limitations. Furthermore, this work presents a general structure to achieve an improved application-specific hitlist in search engines.

The masterthesis ends with a Java prototype that homogenised strings, saves data in tries as well as alternate index structures and support approximate string matching based on the levenshtein distance. The application splits the data in tokens, expands abbreviations, discards stop words, and applies some homogenization on the remaining strings. For the evaluation the prototype loads XML common text data from the digital bibliography & library project. All string operations in the ternary hybrid search tries have delivered good results for a practical application and motivate for further research. Tests with acronyms, abbreviations and SOUNDEX have delivered an improved and application-specific hitlist.

Contents

1 Preface **1**

1.1 Motivation and Objective 1

1.2 Introduction 2

1.3 Definitions 3

2 Fundamentals **4**

2.1 Computational Linguistics 5

2.1.1 Corpora 5

2.1.2 Morphology 7

2.2 Homogenization of Strings 7

2.2.1 Tokenising 7

2.2.2 Abbreviations 9

2.2.3 Acronyms 10

2.2.4 Synonyms 11

2.2.5 Stop Words 11

2.2.6 Numbers 11

2.2.7 Character Transformation 12

2.3 String Similarity 12

2.3.1 String Distance Measurements 13

2.3.2 Lemmatization 16

2.3.3 Stemming 16

2.3.4 Phonetic Transformation 20

2.4 Index Structures for Strings 22

2.4.1 Tries 22

2.4.2 Suffix Solutions 24

2.5 Algorithms for String Matching 25
2.5.1 Dynamic Programming Algorithms 25
2.5.2 Algorithms Based on Automata 26
2.5.3 Bit Parallelism 27
2.5.4 Filtering Algorithms 28

3 Previous and Related Work 29
3.1 Trie Developments 29
3.1.1 String B-Tree 29
3.1.2 Ternary Search Trie 30
3.2 Secondary Storage with Trees 31
3.3 Special Algorithm for Data Clustering 33

4 String Similarity Predicates 35
4.1 Introduction 35
4.2 Concept of String Similarity Predicates 37
4.3 Prototyping 38
4.3.1 Preparation 38
4.3.2 Java Implementation 39
4.4 Experimental Results 40
4.4.1 Hybrid TST Evaluation 40
4.4.2 Evaluation String Similarity and Word Lists 45

5 Conclusion and Further Work 47

A Word Lists 49

B Java Implementation 58

C Evaluation Hybrid TST 61

D PHONIX Rules 63

E Bibliography 66

List of Abbreviations

abbr.	abbreviation
acr.	acronym
adj.	adjective
adv.	adverb
avg.	average
approx.	approximation
cf.	confer
conj.	conjunction
desc.	description
det.	determiner
e.g.	example given
ex.	example
etc.	et cetera
frq.	frequency
i.e.	id est
inf.	infinitive
no.	number
prepro.	preprocessing
prep.	preposition
pron.	pronoun

List of Acronyms

ACM	Association for Computing
BNC	British National Corpus
CL	Computational Linguistics
DBLP	Digital Bibliography & Library Project
DP	Dynamic Programming
IR	Information Retrieval
IT	Information Technology
LCS	Longest Common Substring
LRS	Longest Repeated Substring
MIDWP	Modified version of Invariant Distance from Word Position
NLP	Natural Language Processing
PHONIX	PHONetic Index
PATRICIA	Practical Algorithm To Retrieve Information Coded in Alphanumeric
PoS	Part of Speech
TST	Ternary Search Trie
XML	Extended Markup Language
WWW	World Wide Web

List of Tables

1.1 Notations and definitions 3

2.1 Definitions in computational linguistics 7
2.2 Current problems with abbreviations 9
2.3 Current problems with acronyms 10
2.4 Current problems with synonyms 11
2.5 Frequently used words of the BNC 11
2.6 Calculation matrixes of the levenshtein distance 14
2.7 Examples of levenshtein distances 14
2.8 Examples of enhanced levenshtein distance 15
2.9 Examples hamming distance 15
2.10 Examples damerau distance 15
2.11 Stemming: table lookup method 17
2.12 Stemming: n-gram method 18
2.13 Stemming: successor variety 18
2.14 SOUNDEX groups 20
2.15 SOUNDEX examples 20
2.16 PHONIX groups 21
2.17 PHONIX examples 21
2.18 Ukkonen's CutOff mechanism 26
2.19 Dynamic programming: time and space complexity 26
2.20 Algorithms based on automata: time and space complexity 27
2.21 Bit parallelism: time and space complexity 28
2.22 Filtering algorithms: time and space complexity 28

4.1 Hybrid TST: measurements for insertation 41

4.2 Hybrid TST: measurements for exact matching 42
4.3 Hybrid TST: approximate string matching (n=1,000,000) 43
4.4 Hybrid TST: approximate string matching (n=2,000,000) 44
4.5 Hybrid TST: approximate string matching (n=4,000,000) 45
4.6 Word groups of the DBLP 46

A.1 DBLP: abbreviation list 49
A.2 DBLP: continuation abbreviation list 49
A.3 BNC: lemmatised abbreviation list 50
A.4 DBLP: acronym list with possible substitutions 51
A.5 DBLP: continuation acronym list 52
A.6 BNC: lemmatised acronym list 53
A.7 DBLP: general frequency list 55
A.8 BNC: general frequency list 57

D.1 PHONIX substitutions rules 65

List of Figures

2.1 Areas of string similarity 13
2.2 Standard trie 23
2.3 Compressed trie 23
2.4 PATRICIA trie - standard implementation 24
2.5 PATRICIA trie - digital implementation 24
2.6 Suffix trie 25
2.7 Suffix tree 25
2.8 Suffix array 25
2.9 Graph: Ukkonen's CutOff mechanism 26

3.1 String B-tree 30
3.2 Structure hybrid TST 31
3.3 Insertation process with the quasi-parallel method 32
3.4 Data Clustering with Lujan-Mora and Palomar - precision 34
3.5 Data Clustering with Lujan-Mora and Palomar - error rate 34

4.1 Index structures of the general approach 38
4.2 Structure of the implementation 39
4.3 Java implementation - prototype 40
4.4 Hybrid TST: graph for insertation 41
4.5 Hybrid TST: graph for exact matching 42
4.6 Hybrid TST: graph for approx. string matching (n=1,000,000) . . 43
4.7 Hybrid TST: graph for approx. string matching (n=2,000,000) . . 44
4.8 Hybrid TST: graph for approx. string matching (n=4,000,000) . . 45

B.1 Java implementation: start 58

B.2 Java implementation: $edist_{Lev}$ request 58
B.3 Java implementation: acronym request (ex. 1) 59
B.4 Java implementation: acronym request (ex. 2) 59
B.5 Java implementation: abbreviation request 60
B.6 Java implementation: SOUNDEX request 60

C.1 Hybrid TST evaluation of approx. string matching for m=2 . . . 61
C.2 Hybrid TST evaluation of approx. string matching for m=4 . . . 61
C.3 Hybrid TST evaluation of approx. string matching for m=6 . . . 61
C.4 Hybrid TST evaluation of approx. string matching for m=10 . . . 61
C.5 Hybrid TST evaluation of approx. string matching for m=14 . . . 62
C.6 Hybrid TST evaluation of approx. string matching for m=20 . . . 62

Chapter 1

Preface

1.1 Motivation and Objective

In times of a worldwide globalisation and an increasing meaning of education, the knowledge of useful information gets more and more important. At the beginning of the twentyfirst century, the internet stands on eye-level with traditional mass-media (e.g., books, TV, radio, newspaper) as a carrier of information and news.

Parallel to the expansion of the internet, the advance developments in biotechnology (e.g., genetic engineering) produce similar volumes of data. Both of these have in common that their flood of data in the last ten years grew stronger than the required computer capabilities [BiR95]. Another similarity is, that both internet and biotechnology applications have to analyse data mostly in text form. From this point of view it is not surprising that past and present developments can be applied in both fields. One of the most relevant intersection is the usage of approximate string matching in large text data.

In the contrast to the biotechnology the internet has to face the challenge not only to concentrate on request times but also to find more context relevant information. Associated with this aim the further steps in this field have to consider that documents can include mistakes in orthography or words being abbreviated. Other areas of information are substituted with their acronyms or less important and can be ignored. The concurrent analysis of texts in different languages is against this background a rather visionary objective [Mit05]. All these tasks are united in the fields of computational linguistics.

This masterthesis shows that both sciences can be very well combined and describes all relevant parts of each single field. A prototype, programmed in Java, completes the current masterthesis.

1.2 Introduction

The masterthesis bridges the gaps between *computational linguistics* CL and *information technology* IT to bring a solution for better information retrieval of text-based data. The work is divided in five chapters. The second chapter begins with an introduction of different aspects of both fields and thus lays the basis for the further understanding. The third chapter presents the recent developments including an interesting approach of the University of Alicante. The fourth chapter presents the results of the masterthesis together with a realised prototype in Java. The last passage includes a resume and shows the areas for further research. The appendix includes different kinds of frequency lists for abbreviations, acronyms and enumerations to identify stop words. Furthermore it includes a rare and complete composition of the PHONIX substitution rules.

The first subsection of the following chapter introduces corpora and morphology. Both are important areas of the CL and are useful prerequisites to homogenise strings. The further subsections are describing methods for approximate string matching - seen rather from the information technical point of view. The last two are focused on index structures for strings and on methods for approximate string matching. The latter analyse the drawbacks and advantages for the areas of dynamic programming, bit-parallelism, filter schemes and algorithms based on automata.

The third chapter presents an overview of previous on this topic. It shows the results of two former studies and describes an interesting improvement for tries. The studies are about the secondary storage with trees and the reducing of inconsistency in integrating data from different sources. The latter study introduces an interesting approach to find similarity data, as seen from a more logical point of view. The trie improvement called *hybrid ternary search trie* (hybrid TST) is a mix of an array and a trie.

The objective of this masterthesis is the construction of application-specific and index supported string similarity predicates. Therefore it combines approaches from the CL with techniques from the IT. The fourth chapter describes a solution to reduce the volume of the origin data with a concurrent improvement of the application-specific hitlist. The realised Java prototype homogenises data (i.e. expanding acronyms, ignoring stop words) and implements one of the fastest index structure for strings - hybrid TST [Sed03]. This enhanced trie structure allows fast approximate string matching based on the CutOff mechanism and expands existing knowledge in this area.

The last chapter summarises results of this masterthesis and presents an overview of fruitful research avenues and open problems.

1.3 Definitions

In the context of approximate string matching and k-distance functions some definitions are common. Table 1.1 shows the notations used in this masterthesis. If not other mentioned, all declarations given in this context like *O()* describe the worst-case scenario. Statements whise differ from this are seperatly accentuated.

Notation	Definition
T	prossed text(string)
s	number of strings
n	length of the T
Σ	alphabet
a	size of Σ
P	pattern / string
m	length of P
occ	number of hits
w	length of a computer word
k	edit distance

Table 1.1: Notations and definitions

Chapter 2

Fundamentals

This chapter lays the basics to understand the construction of application-specific and index supported string similarity predicates. It introduces in special fields of the CL and describes corpora and morphology. The further subsections describe the homogenization of text, present methods to identify similar words and explain fast index structures for strings. The last part of this chapter explains the most common schemes for approximate string matching. The aim of this section is to present a comprehensive overview of the most common methods, which are known in this context. Besides that, this chapter should help to bridge the gap between CL and IT with the joint aim to achieve best results in exact matching and in imprecise data requests.

Although the following pages present a complete overview, some aspects are emphasised. Some parts of CL (e.g., speech recognition) were left out, because they are more applied on spoken texts and not -as we used- on written text. The application of linguistic methods causes the preprocessing of text. For this reason the last two sections of this chapter are more focussed on algorithms, which uses this advantage. All here mentioned approaches and methods are developed for the usage in English texts. The reason is that the most developments in CL and IT are made-up in Anglo-Saxon countries. Most of them can be adapted for other languages, but these cases are not further explained and can be looked up with the help of the attached bibliography.

2.1 Computational Linguistics

Computational Linguistics is an interdisciplinary field dealing with the logical modelling of natural language from a computational perspective. Actually fallen into oblivion, since the beginning of the 1990ies linguistics passes a renaissance [Ham02]. The reasons for this are the flood of unstructured information in the *world wide web* WWW and the steady economic globalisation with its merging of languages. The combination of morphological methods with the possibilities of the current IT (e.g., databases, artificial intelligence) makes this science more and more important for many of today's computer supported information systems. Typical fields of application are search engines and programs for *information retrieval* (IR). The visionary of CL is to identify identical information for a specific context, independent of its origin. The major parts of CL are shown below[Mit05]:

- computer aided corpus linguistics (e.g., BNC),
- design of parsers for natural languages
- design of taggers (e.g., POS-taggers),
- definition of specialised logics (e.g., resource logics for NLP),
- research in the relation between formal and natural languages in general, and
- machine translation (e.g., by a translating computer).

Basically all listed parts are necessary to achieve best results in the CL. However, the title of the masterthesis is 'The construction of application-specific and index supported string similarity predicates'. The task is to identify and to combine the most relevant parts of both fields of knowledge. For this reason we are concentrated on the first two items. The tagging process supports in individual cases the homogenisation of strings (e.g., detection of acronyms), but can be neglected in this work (development level).

2.1.1 Corpora

The word 'corpus', derived from the latin word meaning 'body', may be used to refer to any text in written or spoken form. However, in modern linguistics this term is used to refer to large collections of texts which represent a sample of a particular variety or use of language(s) that are presented in machine readable form. Other definitions, broader or stricter, exist [EnW96]. Corpora do support the following work [Ham02]:

- creating of concordances,
- construction of word statistics,
- creation of frequency lists, and
- reference for tagging processes.

Computer-readable corpora can consist of raw text only, i.e. plain text with no additional information. Besides that, many corpora add some kind of linguistic information, here called mark-up or annotation. The most relevant enrichment is the *part of speech* (PoS) and is determined during the tagging processes. The PoS is assigned to each single word (no ignoring of same words) in the corpus. It describes the specific grammatical category of the word within the analysed sentences (e.g., noun, verb, adjective). Formerly the tagging process was done manually, current corpora are tagged automatically - but based on the information of the former work. The tagging work consumes some time and is processed in several steps (e.g., disambiguating) [Zie00].

There are many different kinds of corpora. They can contain written or spoken (transcribed) language, modern or old texts, texts from one language or several languages. The texts can be whole books, newspapers, journals, speeches etc., or consist of extracts of varying length. The kind of texts included and the combination of different texts vary between different corpora and corpus types. 'General corpora' consist of general texts, texts that do not belong to a single text type, subject field, or register.

An example for a general corpus is the well mixed *British National Corpus* (BNC). The BNC is a very large (over 100 million words) corpus of modern English, both spoken and written. The project was carried out and is managed by an industrial/academic consortium lead by Oxford University Press, of which the other members are major dictionary publishers Addison-Wesley Longman and Larousse Kingfisher Chambers; academic research centres at Oxford University Computing Services, Lancaster University's Centre for Computer Research on the English Language, and the British Library's Research and Innovation Centre. Work on building the corpus began in 1991, and was completed in 1994. The project was funded by the commercial partners, the Science and Engineering Council (now EPSRC) and the DTI under the *Joint Framework for Information Technology* (JFIT) programme. Additional support was provided by the British Library and the British Academy [1].

[1] quoted from http://www.natcorp.ox.ac.uk/

2.1.2 Morphology

Morphology is a subdiscipline of linguistics that studies word structure. While words are generally accepted as being the smallest units of syntax, it is clear that in most (if not all) languages, words can be related to other words by rules. A morpheme describes the smallest significant unit of a word, table 2.1 shows a delimitation to other definitions in this context [Bau03].

Desc.	Definition
phoneme	the basic unit of sound that can be used to distinguish words or morphemes
syllable	smallest rhythmical unit(s) of a word
word	basic unit of a language

Table 2.1: Definitions in computational linguistics

The classical morphological analysis reduces words and forms of words to their roots. Therefore it identifies each single grammatical information (i.e. case, number, tense, finite), inventories them, classifies the units and applies rules to form syntactically correct words [LNP04]. These words are also called lemma, are not further reducible and are for example used to create dictionaries. A similar process is stemming, but the resulting stems have no lexicographical background (cf. subsection 2.3.3 on page 16).

A simple example represents the word *unbelievable*, which is reduced to *belief*. The prefix *un* is classified as a derived adjective and has no significant meaning, the same to the suffix *able*.

2.2 Homogenization of Strings

The aim behind the homogenization of text is the translation of different notations in one uniform format - independent of language, spelling mistakes and other linguistic characteristics. This area is one of the main parts in linguistic studies and not solved yet. The next subsections will show the basics, which are needed to homogenised single words for similarity predicates in computer science.

2.2.1 Tokenising

Tokenising is the splitting of text in single word units (tokens) along obvious word boundaries (i.e. spaces). This step is necessary, because all linguistic analyses (e.g., tagging, text search) start at the word level. The objective is to

identify words and number sequences following by orthographic specifications. The following passage describes the tokenising process with its main steps and problems.

Erasing Whitespaces and Special Characters

In this step the text will be split in single tokens. For this purpose one token is defined as a sequence of characters between blanks, tabulators or line breaks. During the next operation each token will be relieved from enclosed special characters as round brackets, square brackets, hyphens or apostrophes. In linguistic science -especially in the development of text corpora- this distinction is fundamentally far-reaching (e.g., the treatment of end punctuation marks and syllabifications) [Zie00].

Splitting of Words

Dependent of the language and on the context some parts of sentences are put together of other words (e.g., sub-menu). For this purpose each single token will be parsed to special characters like slashes or hyphens. The following exceptions are given:

- if one of the words consists of just only one character, the combined word will not be split. This context mostly appears at measurement units (e.g., x-ray) and
- written numbers with slashes will not divided in several parts. This concerns description of years, fractions or references.

Uniting of Numbers

In some cases big numbers are be written in tokens with three characters, divided with a whitespace (e.g., 20 000) - it is necessary to unite these tokens. Very difficult to handle are constellations of sequences of numbers with different meanings (e.g., in 1995 1,000). This special case is impossible to solve and results mostly in misinterpretation, but the error rate is negligibly small.

Disambiguating of Dots

This process is the most complex step of all operations described above and requires well prepared tokens. The major task is to distinguish end punctuation marks, abbreviations and the dot of ordinate numbers from each other. This will be supported by lists of abbreviations and lists of suffixes, normally those

extracts can be delivered from text corpora. A major problem of abbreviations is their unsteadiness, that means, abbreviations are constantly changing like many aspects of the language. For this reason the recognition of abbreviations will be optimised with the help of lists of suffixes, which include sequences of characters, which occur exclusively at the end of abbreviations [Ham02]. If one token does not occur in the list of abbreviation, the end of the token is checked with the list of suffixes. If a hit results , the token will be replaced with its long not abbreviated name.

2.2.2 Abbreviations

Abbreviation is strictly a shortening, but more particularly, an abbreviation is a letter or group of letters, taken from a word or words, and employed to represent them for the sake of brevity. In modern English there are several conventions for abbreviations and the choice may be confusing. The only rule universally accepted is that abbreviations should be consistent in the document, and to this end publishers express their preferences in a style guide.

In general the most common abbreviations are be used in the same way. Reports, using frequency lists of the BNC have shown that all abbreviations included in the first thousand entries are -more or less- standardised. A closer look to the extraction of the BNC shows an unintentional but a new piece of knowledge. Words are be shorted by the same letters, but often with a different usage of dots (e.g., ie. and i.e.). Sometimes words are abbreviated with a dot at the end, sometimes after each single letter. Furthermore is to be noticed, that shortened plurals are not standardised. No matter where these mistakes come from, the challenge is to homogenise all abbreviations of the same word (lemmatization, cf. appendix A on page 50).

Abbr.	Variants	Substitution
no	no, no.	number
Mr	M.R., Mr, Mr.	mister
per cent	per cent, per cent.	per centum
.net		Microsoft XML Web Services platform
asn.1		Abstract Syntax Notation One
x.500		CCITT Directory Services Protocol
802.11b		Wireless LAN Equipment Standard update

Table 2.2: Current problems with abbreviations

Technical lists show another phenomenon. So far, all abbreviations consist of letters. In the opposite to the conventional abbreviations, technical ones also include numbers. These cases have to be taken into account during the uniting

of numbers and the disambiguating of dots. Table 2.2 shows an extract of the mentioned problems, a complete frequency list with more problem definitions is given in appendix A on page 49.

2.2.3 Acronyms

Acronyms and initialises are abbreviations formed from the initial letter or letters of words, such as NATO (north atlantic treaty organization) or ACM (Association for Computing Machinery). They are pronounced in a way that is distinct from the full pronunciation of what the letters stand for. Of the two words, acronym is the much more frequently used and known, and many speakers and writers refer to all abbreviations formed from initial letters as acronyms. However, many others differentiate between acronyms and initialises. An acronym is a pronounceable word formed from the initial letter or letters of the constituent words. An initialise is an abbreviation pronounced as the names of the individual letters, and is formed only from the initial letter of constituent words. This distinction is supported by many dictionary definitions, but not by all.

In practice the difference between acronyms and abbreviations is smaller than it should be. The stipulation to write acronyms exclusively in capital letters disappears more and more [Mit05]. This fact produces similar problems as mentioned in the subsection above. Furthermore acronyms are ambiguous and not easy to distinguish from personal initials. This problem makes it difficult to replace acronyms and initialises with their correct substitution in mixed text. Table 2.3 shows an extract of the mentioned problems, a complete frequency list with more problem definitions is given in appendix A on page 51.

Acronym	Variants	Substitution
NHS	NHS	National Health Service
LA	L.A., LA, La	Los Angeles
RAF	R.A.F., RAF	Royal Air Force
DNA		Deoxyribo Nucleic Acid
		Digital Network Architecture
		Defense Nuclear Agency
HP	H.P., HP, H-P	Hans Peter
		Hewlett-Packard

Table 2.3: Current problems with acronyms

2.2.4 Synonyms

Synonyms are words with the same meaning. They are used to make text more readable. This sounds advisable and should be easy to applicable. But the simple substitution of words with their synonyms can change the pronouncing of sentences and results sometimes in misunderstanding. There are partly differences with the definition of similar words. These difficulties allow no simple replacement, because it would provide a false result list in IR. Unfortunately no corpora do provide lists, concerning synonyms. In this case such lists have to be created entirely manually (cf. table 2.4).

Synonym	Substitution
head	head, bonce, conk, crown
husband	hubby, cuckold
child	bairn, infant, kid, mopped, sprog

Table 2.4: Current problems with synonyms

2.2.5 Stop Words

Stop words are defined as less meaningful words, respectively words without any significant contribution to a special context (sentence). These words are mostly expletives like articles, adverbs, prepositions and conjunctions. For this reason, systems for IR do ignore stop words during their indexing processes and in most cases they will also be deleted. The BNC determines in the first 100 hits up to 90 per cent expletives, without any context to a special subject field. Table 2.5 below shows words, which are always displayed on top of the most frequency list (cf. appendix A on page 56).

No.	Stop Word	Word Class
1.	the	det.
2.	of	prep.
3.	and	conj.
4.	a	det.
5.	in	prep.

No.	Stop Word	Word Class
6.	to	inf.
7.	it	pron.
8.	is	verb
9.	to	prep.
10.	was	verb

Table 2.5: Frequently used words of the BNC (top ten)

2.2.6 Numbers

Numbers are, similar to acronyms and abbreviations, difficult to homogenise [Ham02]. If numbers are tokenised without any other characters, or does the

token describe a number in letters (e.g., twelve instead of 12), the further homogenization is possible without any problems. Difficulties occur when the token includes dots or is arranged with letters. Latter leaves the possibility, that the string represents a roman illustration.

2.2.7 Character Transformation

The last part in the homogenization process is the character transformation. This step shifts all characters to lower case and substitute characters with accents. The latter replaces special characters with their simpler variants (e.g., é → e). The transforming of all characters to lower case is one of the most important steps to homogenise words and should be applied at the end of the homogenization process. The reason for this is that the identification of acronyms mostly differs between upper-case and lower-case (cf. subsection 2.2.3). Especially for the German language is the shifting very useful, because all nouns begin with an upper-case. Furthermore it considers words at the beginning of sentences. These words are mostly begin with an upper-case, just for optical reasons. From the technical point of take case-sensitive comparisons take more performance than non case-sensitive and are easier to handle.

2.3 String Similarity

This part explains the fundamental methods of defining the similarity between strings and is separated in the four subsections string distance measurements, lemmatization, stemming and phonetic transformation. Each subsection describes by means of selected examples its most important schemes. These decades-old ideas are still up-to-date and represent the basis of today's combined or enhanced methods [ZoD96].

All methods in the area of string similarity have in common that they support the search for strings in a set of words. Some of them have their main field in data clustering based on linguistic processes (e.g., stemming), the other ones are based on mathematical or logical algorithms (e.g., edit-distance). For this reason the schemes of string similarity can be distinguished in the area of IR and in the area of *concrete string measurements* (CSM). IR has its origin in the linguistics and has been completed until the triumphal march of the WWW. IR has the objective to identify whole documents, which describe the same context (text level). The methods of CSM are used to match similar written words or similar pronounced expressions in documents (word level). These methods are mostly applied in search engines (e.g., Google) or support checks at orthography.

A further distinction can be made by looking at the result set of each method. In

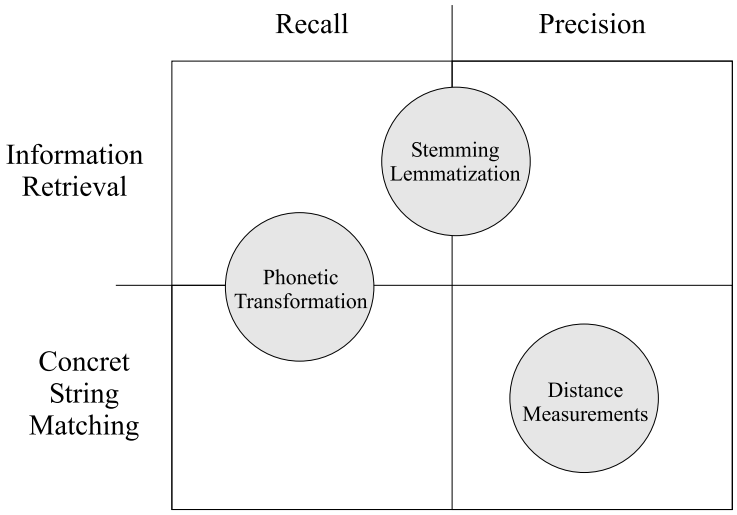

Figure 2.1: Areas of string similarity

this context the two expressions *Recall* and *Precision* are known. Precision is a measure of the usefulness of a hitlist; recall is a measurement of the completeness of the hitlist [Mit05].

It is difficult to process a complete and fix categorization for each method (e.g., n-gram, affix removal, SOUNDEX) in specific areas. Dependent of their field of application the result set can be slightly different. Therefore figure 2.1 shows not each single method, but the orientation of the four string similarity groups.

2.3.1 String Distance Measurements

The string distance measurement describes the similarity between two strings on the basis of the combination of edit functions and/or transposing characters. The edit distance (short, edist) or k-distance (short, k-dist) is defined as the minimised number of operations to translate from one string into another string. The following paragraphs are describing the most popular representatives.

Levenshtein Distance

The Russian Vladimir I. Levenshtein introduced in 1965 a general algorithm to calculate $edist_{Lev}$ with the edit functions insert, delete and replace [Lev65]. The method can be applied on arbitrary pairs of strings with different lengths and has no restrictions like word length or byte length. It uses a two dimensional matrix, which has to be fully filled out to receive a valid result. For this reason the time complexity of the algorithm *O(nm)* can be high for long strings. In practice the levenshtein distance is used for proofing orthography and identifying duplicates.

Definition 2.1 *Be* $A = \{A_1, A_2, ..., A_i\}$ *a String with length* i *and* $B = \{B_1, B_2, ..., B_j\}$ *a String with length* j.

$$edist_{Lev} = \begin{cases} D_{i,j} = 0 & \text{for } i = 0 \text{ and } j = 0 \\ D_{0,j} = j & \text{for } i = 0 \text{ and } j > 0 \\ D_{i,0} = i & \text{for } i > 0 \text{ and } j = 0 \\ D_{i,j} = \begin{matrix} min\{D_{i,j-1} + 1, \\ D_{i-1,j} + 1, \\ D_{i-1,j-1} + \delta(A[i], B[j])\} \end{matrix} & \text{for } i > 0 \text{ and } j > 0 \\ \delta(A[i], B[j]) = \begin{cases} 0 & \text{for } A[i] = B[i] \\ 1 & \text{otherwise} \end{cases} \end{cases}$$

Example 2.1 *To illustrate the definition 2.1 let us assume we have the destination string STRINGS and two source strings STRANDS and SPHINX. Table 2.6 shows the complete calculation matrix of both examples with the respective result cell on the right below. Table 2.7 shows the levenshtein distance for two more examples emphasing that the* $edist_{Lev}$ *must not be correlated to other string similarity methods like soundex, metaphone or stemming.*

-	j	S	T	R	I	N	G	S
i	0	1	2	3	4	5	6	7
S	1	0	1	2	3	4	5	6
T	2	1	0	1	2	3	4	5
R	3	2	1	0	1	2	3	4
A	4	3	2	1	1	2	3	4
N	5	4	3	2	2	1	2	3
D	6	5	4	3	3	2	2	3
S	7	6	5	4	4	3	3	2

-	j	S	T	R	I	N	G	S
i	0	1	2	3	4	5	6	7
S	1	0	1	2	3	4	5	6
P	2	1	1	2	3	4	5	6
H	3	2	2	2	3	4	5	6
I	4	3	3	3	2	3	4	5
N	5	4	4	4	3	2	3	4
X	6	5	5	5	4	3	3	4

Table 2.6: Calculation matrixes of the levenshtein distance

STRANDS → STRINGS	$\Longrightarrow$	$edist_{Lev} = 2$
SPHINX → STRINGS	$\Longrightarrow$	$edist_{Lev} = 4$
STINGS → STRINGS	$\Longrightarrow$	$edist_{Lev} = 1$
STRONGROOM → STRINGS	$\Longrightarrow$	$edist_{Lev} = 5$

Table 2.7: Examples of levenshtein distances

The original levenstein algorithm uses a weight of 1 for each edit-function. In some derivatives the replacement process has a weight of 2. The reason therefore is that this edit function is a combination of insertation and deletion. In adaption of definition 2.1 table 2.8 shows the new results.

STRANDS → STRINGS	$\Longrightarrow$	$edist_{Lev} = 4$
SPHINX → STRINGS	$\Longrightarrow$	$edist_{Lev} = 7$
STINGS → STRINGS	$\Longrightarrow$	$edist_{Lev} = 1$
STRONGROOM → STRINGS	$\Longrightarrow$	$edist_{Lev} = 7$

Table 2.8: Examples of enhanced levenshtein distance

Hamming Distance

This measurement is named after the mathematician Richard Wesley Hamming (one of the founders of ACM) and describes the smallest number of substitutions to lead over one string in another string which has the same length. Mainly used in coding and information theory the hamming distance $edist_{Ham}$ shows the number of places in which the two strings differ (cf. table 2.9)[Ham80].

STRINXS → STRINGS	$\Longrightarrow$	$edist_{Ham} = 1$
STRANDS → STRINGS	$\Longrightarrow$	$edist_{Ham} = 2$
11001101 → 11001111	$\Longrightarrow$	$edist_{Ham} = 1$
11110000 → 00001111	$\Longrightarrow$	$edist_{Ham} = 8$

Table 2.9: Examples hamming distance

Damerau Distance

The damerau distance is based on the levenshtein distance and enhanced this method of proceeding by transposing characters. Normally the cost of transposing two adjacent characters has a weight of one, but there are a lot of variations in circulations (cf. table 2.10).

STRNIGS → STRINGS	$\Longrightarrow$	$edist_{Dam} = 1$
STGINRS → STRINGS	$\Longrightarrow$	$edist_{Dam} = 2$
HTRNIGS → STRINGS	$\Longrightarrow$	$edist_{Dam} = 2$
SPHINX → STRINGS	$\Longrightarrow$	$edist_{Dam} = 4$

Table 2.10: Examples damerau distance

Episode-Distance

The Episode-Distance has the characteristic that it only allows insertation with the weight of 1. Therefore its application is not practical for tasks in the CL. In the literature the search problem in many cases is called 'episode-matching',

since it models the case where a sequence of events is sought, where all of them must occur within a short period[2].

2.3.2 Lemmatization

The lemmatization process is one part of the morphological analysis in the CL and sizes a word down to its basic form. This basic form (lemma) has to be a regular expression in the specific language and is not further reducible (e.g., an entry in the data dictionary). The process assigns any word formation like inflections or word concatenations to its lemma and orientates itself for this purpose mostly on prefixes and suffixes. Dependent on the language many inflections are not formed by a specific set of rules (e.g., mice → mouse). For those cases it is necessary to prepare a dictionary with special lemmata - this method is called lemma selection. Basically the lemmatization process is deeply language dependent and should be carefully adapted for each language.

2.3.3 Stemming

Similar to the lemmatization, stemming helps to reduce a word to its roots. In contrast to the lemmatization process the root word called stem must not be a correct morphological part of the respective language. This characteristic supports an easier implementation of computer based methods. There are different stemming methods and algorithms to be found in the literature. Most of them are based on linguistical knowledge and are -more or less- dependent on a specific language [LNP04].

Stemming is used in the computer linguistic (e.g., working with corpora) and is an essential part of systems for information retrieval. It is primarily helpful if the user is not exclusively search to single words but rather for whole documents. The reason for that is that stemming leads back all variants of one morphological expression to one single stem. This coherence leads to better results, because in most cases one document contains similar words with the same stem. Other advantages of stemming are the reduction of similar words (e.g., index structure) and less space for the vocabulary.

As mentioned above, there are various stemming algorithms known in the literature, some with a couple of interpretations (e.g., affix removal). Common to all is the preprocessing of texts (e.g., rejection of stop-words), stemming of all variants of one word (conflation), and the ignoring of double stems [LNP04]. The following paragraphs are describing the most popular methods and show their advantages and disadvantages.

[2]quoted from [Nav01]

Table Lookup

This method is based on simple rules and a comprehensive data dictionary. Each word of a text passage is parsed word by word and is compared with the dictionary. In case of a successful table lookup the word is replaced with its stem, respectively the index structure is updated. To increase the chance for a successful table lookup several support steps can be applied to the word to be stemmed (e.g., formation/removing of the plural). Is no stem found, it is supposed that the word to be replaced is a proper name. Table lookup is the simplest of all known methods and requires a complete and well updated data dictionary. Table 2.11 shows an example for the stem run.

Stem	Words
run	dash, run, running, runable, runnings, rush, ran, sprint, sprinter

Table 2.11: Stemming: table lookup method

n-gram

Here, the word to be stemmed is split in several overlapped n-grams (substrings). 'n' is defined as the fixed number of characters included in one substring. This method based on the calculation of a similarity measurement of two words. The value is compared with a predefined threshold (mostly 0.6) [Ham02]. Is the threshold less than the calculated value, both words are defined as morphological identical - otherwise both words are describing different morphological processes (cf. example 2.2). The using of n-grams to define similar words in a text can be very time consuming. Therefore it is popular to combine this method with the above mentioned table lookup process.

Definition 2.2 *Similarity Measurement S:*

$$
\begin{aligned}
n_c &= \text{Number of common n-grams} \\
n_1 &= \text{Number of n-grams in the 1. word} \\
n_2 &= \text{Number of n-grams in the 2. word}
\end{aligned}
$$

$$
S = \frac{2 * n_c}{(n_1 + n_2)}
$$

Example 2.2 *Given are two words A and B with a fixed length of two characters for each n-gram.*

Example 1	Example 2
A_1 = st tr ri in ng	A_2 = st tr ri in ng
B_1 = st ti in ng	B_2 = st tr ra an nd
$S_1 = \frac{2*3}{(5+4)} = \frac{6}{9} = 0,67$	$S_2 = \frac{2*2}{(5+5)} = \frac{4}{10} = 0,4$

Table 2.12: Stemming: n-gram method

Successor Variety

The successor variety method splits words in its segments and substitutes these substrings with their stems. The difficulty is to identify the single segments in each word. Therefore a couple of ideas are mentioned in the literature (e.g., peak and plateau). All of them have in common that they define a word with the help of the number of possible successor characters. Based on the assumption, that at the end of each single segment the number of successor characters decreases, the word is split at each single transition (cf. example 2.3). The definition of possible successors comes from a corpora or any other dictionary.

Example 2.3 *To illustrate the successor variety method we define a small dictionary of fifteen words and determine for 'enjoyable' the single word segments with the help of the 'peak and plateau' algorithm.*

Dictionary:
able, ably, enigma, enigmatic, enjoin, enjoy, enjoyable, enjoyment, enlarge, enlighten, erase, ergonomic, etch, joy, joystick

Step	Prefix	Successor-Variety	Characters
1.	e	3	n, r, t
2.	en	3	i, j, l
3.	enj	1	o
4.	enjo	1	y
5.	**enjoy**	**2**	**a, m**
6.	enjoya	1	b
7.	enjoyab	1	l
8.	enjoyabl	1	e
9.	enjoyable	1	blank

Table 2.13: Stemming: successor variety

Affix Removal

The affix removal methods identify common prefixes and suffixes in words and delete these parts in a special order. There are several known approaches in the literature, most of them have in common a preprocessing step, a repeatable main part and a postprocessing part. The preprocessing step identifies special letters and substitutes these characters with a homogenised variant (e.g., in German: a→ae, ß→ss). Basically, the main part identifies and deletes the longest possible prefix and/or suffix of the to be stemmed word. Therefore all methods apply special rules to the word and delete all affixes in a repeatable process. The main part ends, if the process can not find any more erasable word parts.The exact implementation depends on the language and differs from method to method. The postprocessing step checks the result and sometimes it proceeded a re-substitution of characters. The problems of all affix removal methods are overstemming and understemming. Both defined as follow [Mit05]:

- if two words belong to the same conceptual group, and are converted to the same stem, then the conflation is correct; if however they are converted to different stems, this is counted as an understemming error, and

- if the two words belong to different conceptual groups, and remain distinct after stemming, then the stemmer has behaved correctly. If however they are converted to the same stem, this is counted as an overstemming error.

Another problem is the identifying of proper names. These words must be excluded. Therefore all methods have a list with suffixes, which should identify proper names (e.g., -er). In this context, nouns like painter or reader are not stemmed.

In general, affix removal methods are easy to implement and achieve good results, depending on the effort of implementation. But also affix removal is not perfect. The best known methods in this area are Lovins, Salton, Dawson, Porter and Paice. Martin Porter developed his method in 1980 at the University of Cambridge [Por80]. The core of the work is the counting of word patterns, which presents a sequence of vowels and consonants (VC). In dependency of the counted patterns and the current suffix, the porter algorithm applies a special rule to the word. Mostly it deletes the suffix. This process will be repeated and repeated until no more VC pattern are counted. The stem 'includ' presents the words include, included, includes and including. A counterexample is the stem 'design'. It presents the words designated and designed.

2.3.4 Phonetic Transformation

Besides the two other areas of string similarity measurements, phonetic transformation represents the third and the last column of methods to identify similar text information. The aim of the phonetic transformation is to group words with the same spelling and the same pronunciation. Unfortunately these methods are also heavily language dependent and have to be adapted to each specific language. All of them have in common that they are working better with proper names than with other words. The reason for that is that proper names have in different languages mostly a similar or sometimes an identical pronunciation. Another common characteristic of all below mentioned phonetic methods is, their conversion of words into a phonetic code - consisting of letters and numbers. The following paragraphes are describing the most popular methods.

SOUNDEX

SOUNDEX is the best-known phonetic matching method and was developed by Odell and Russell (patented 1918). They were the first people who transformed words in a phonetic code. Based on the sound of each letter, they separated the whole alphabet in seven groups (cf. table 2.14). The algorithm consists of four steps. In the first step all letters -beside the leading one- have to be transformed in their phonetic code. After that, all adjacent repetitions and all characters based on a vowel are deleted. In the last step the SOUNDEX code is represented by the first four characters, the remaining substring will be rejected. If the phonetic string in step three is too short, it will be completed with '0'. The SOUNDEX scheme is an easy to implement algorithm, which leads from bad to good results - depending on the text to be transformed. The greatest disadvantage is that the algorithm just considers the first part of a string and does neglect specific pronunciations. Basically only identical SOUNDEX codes presents similar words. However, in some variants also similar SOUNDEX codes present similar words. As illustrated in table 2.15 some results are plausible (No. 3 and No. 4), some not (No. 5 and No. 6).

Letter	Substitution
A E I O U H W Y	0
B F P V	1
C G J K Q S X Z	2
D T	3
L	4
M N	5
R	6

Table 2.14: SOUNDEX groups

No.	Word	Soundex Code
1.	SPHINX	S152
2.	STING	S352
3.	STAND	S353
4.	STRAND	S353
5.	STRONGROOM	S365
6.	STRING	S365
7.	SING	S520

Table 2.15: SOUNDEX examples

PHONIX

The PHONIX method is an enhancement of the SOUNDEX algorithm with a slightly different set of codes and more than 100 different rules of substitutions for substrings (cf. appendix D on page 63). Based on the neglecting of pronunciations in the SOUNDEX algorithm T.N. Gadd developed PHONIX (PHONetic Index) in 1988 [Gad88]. The improvements are: nine character groups (cf. table 2.17), the leading letter will also be transformed in a phonetic code, the code consists of eight characters and there are rules for a preprocessing. Similar is the substitution of letters with numbers, ignoring of vowels and keeping the first character. The algorithms consists of five steps:

1. apply substitution rules,
2. ignore A E I O U H W Y,
3. delete all adjacent characters (beside the first one),
4. ignore alphabetical characters, and
5. maximum key length consists of eight letters.

Letter	Substitution
A E I O U H W Y	0
B P	1
C G J K Q	2
D T	3
L	4
M N	5
R	6
F V	7
S X Z	8

Table 2.16: PHONIX groups

No.	Word	PHONIX Code
1.	SPHINX	S87528
2.	STING	S8352
3.	STAND	S8353
4.	STRAND	S83653
5.	STRONGROOM	S8365265
6.	STRING	S83652
7.	SING	S852
8.	SINK	S852
9.	STRINGS	S836528

Table 2.17: PHONIX examples

Metaphone

Metaphone is based on PHONIX and was developed by Lawrence Philipps in 1990 [Phi90]. The first version was seen as too strict in some details and has been improved therefore. The improved version called 'double metaphone'. This development level provides two results values, with the background to highlight the pronouncing of other languages.

2.4 Index Structures for Strings

Index structures are efficient algorithms on extracted and reorganised data with the objective to support a faster access to its source. They can work with all types of data, but they are mostly used in combination with numbers [Gus97]. The advantage of numbers is the fix size (e.g., 32Bit) and their internal presentation. This results to less overhead for further calculations (e.g., Hashing Index). The drawbacks of the traditional index structures are the lower efficiency in combination with strings (variable length) and the fact that they are not (or unfavourable) useable for approximate request [GoT98]. The following subsection shows index structures especially developed for the usage with strings. All of them have in common that they can be used for external data storage and any data operation as well as different kinds of search requests (e.g., substrings, wildcards, approximate schemes) can be applied to these special index structures.

2.4.1 Tries

All mentioned tries are described with the focus on their essential features. Tries (reTRIEvial) are regular tree structures and can be designed with n children (n-ary). In most cases the practical implementation yield its place its illustrated description. The background is, that the effective implementation follows very sophisticated and extensive algorithms. For example, the PATRICIA trie uses specially added pointer connections (called Skip-Bits) for its construction. Other tries are implemented as 2-way digital tries and transform characters for a faster processing to a binary code. All technical finesses have been described in many papers before and can be looked up with the help of the bibliography. The objective of this passage is to show the basic statements with their drawbacks and advantages.

Standard Trie

The standard trie was developed by Brandais [Bra59] and Fredkin [Fre60] and files all strings in their alphabetical order. Each node (without the root) is labelled with a character and the paths from the external nodes to the root yield one string. The main drawbacks are: nodes with one successor take too much space, the trie is not well balanced and according to its implementation, the internal nodes are different to the leaves. Its advantage is the easy way of implementation. The construction time is $O(n)$, same for the space. The complexity for exact matching and prefix matching is $O(m)$. The performance of substring searches and mismatch requests depends on the number of children and depends slightly on the used method (e.g., CutOff), but the average search

time is *O(m + occ)*. Figure 2.2 shows an example with the words: hand, hint, stand, strand and string.

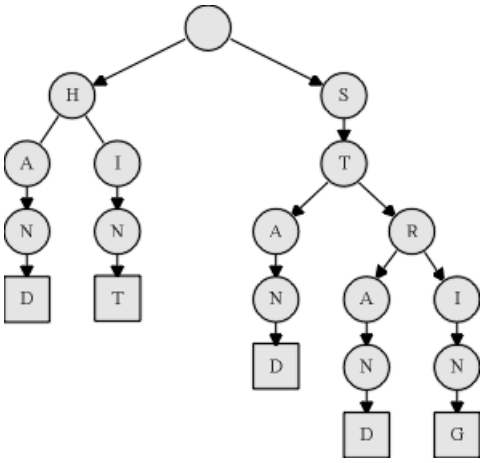

Figure 2.2: Standard trie

Compressed Trie

The compressed trie is the compact version of the standard trie. It shrinks nodes and edges with just one child and labels the result node with all summarised characters. This solution reduces solely the space problem of the standard trie (the remaining problems are still open). The amount of space used decreases by up to 1/3 compared to standard tries [Gus97]. The compressed trie has the same time and space complexity as the standard trie. Figure 2.3 refers to the above illustrated example.

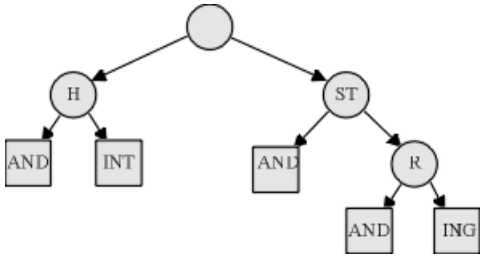

Figure 2.3: Compressed trie

PATRICIA Trie

The PATRICIA trie (*Practical Algorithm To Retrieve Information Coded in Alphanumeric*) was published in 1968 by D. Morrison [Mor68]. It is an compressed trie with another way to label nodes and edges (always two successor nodes). The latter are only labelled with the first character (branching character) and all leaves of the PATRICIA trie are pointers to the strings (cf. figure 2.4); figure 2.5 presents the binary implementation. The PATRICIA trie reduces the problem

of different nodes, but is also not well balanced. The implementation process is very extensive because of the further developments by McCreight [McC76] and Ukkonen [Ukk95]. Both have improved the space and time complexity. To date the construction time and space complexity is *O(n)*. The PATRICIA trie provides the best average results of exact matching and prefix matching. The k-distance requests and substring matching are less efficient, because all strings are not fully saved in the trie (only the reference).

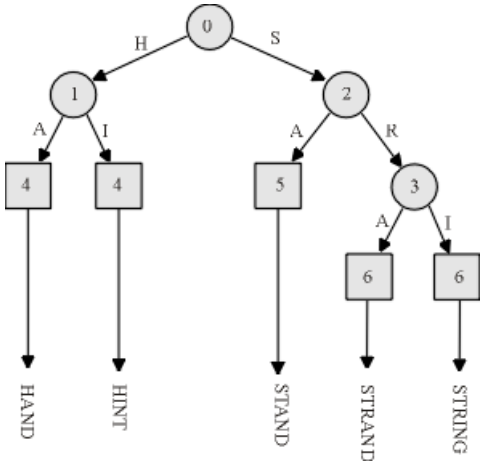

Figure 2.4: PATRICIA trie - standard implementation

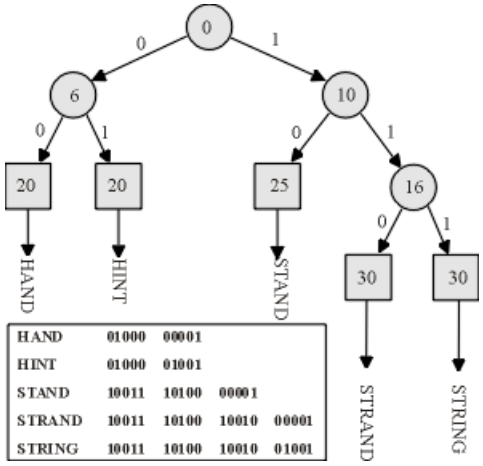

Figure 2.5: PATRICIA trie - digital implementation

2.4.2 Suffix Solutions

The subsection suffix solution combines suffix tries, suffix trees and suffix arrays. All have in common that they save all suffixes of a text (string) and that their performance analysis depends on the way of implementation as well as of the size of the used alphabet. Suffix solutions are mainly used in the bioinformatics for problems concerning the *longest common substring* (LCS) and *longest repeated substring* (LRS). The suffix trie is the first development level and not further used (cf. figure 2.6). Its problem is the quadratic complexity for space and time.

Weiner [Wei73] solved these problems with his introduction in suffix trees. This development level based on PATRICIA trie and was improved by E. McCreight [McC76] and E. Ukkonen [Ukk95](cf. figure 2.7). The suffix tree has a complexity of *O(n)* for construction time and space. It achieves best results for texts with a small alphabet. In this case the complexity for the LCR, LRS, exact matching and prefix matching is always around *O(m)* (average). Substring requests are solved in *O(m+occ)* and inexact matches have a complexity of *O(mk + occ)*.

The third development level called suffix arrays [MaM93] and achieves best results for larger texts and/or larger alphabets (cf. figure 2.8). This solution can derived from a suffix tree and sorts all suffixes in alphabetical order. The array saves the pointers to each single suffix and reduces the space up to approxi-

mately 65%. Because of the sorted suffixes, suffix arrays are less dependent on the alphabet and are used for substring matches *O(m + log n + occ)*. The main drawback is the higher construction time *O(n log n)* [Sun04].

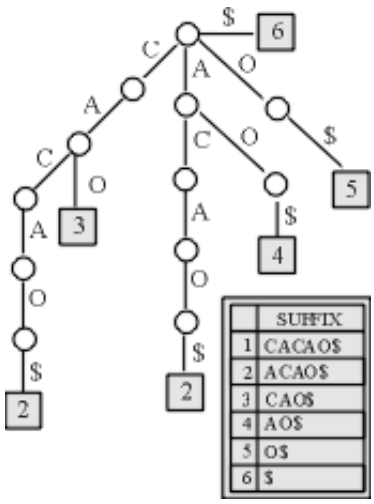

Figure 2.6: Suffix trie

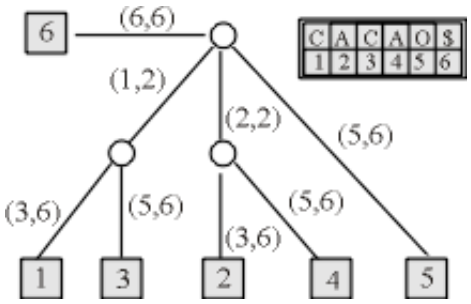

Figure 2.7: Suffix tree

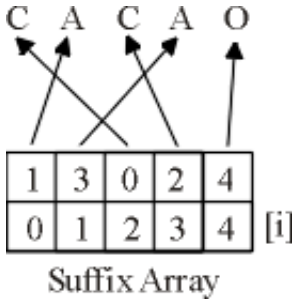

Figure 2.8: Suffix array

2.5 Algorithms for String Matching

This passage continues the previous section and presents an overview of algorithms for string matching. The objective is to introduce the most relevant methods for exact matching and k-distance requests. All have in common that they are used in the bioinformatics or in text editors (e.g, vgrep) for pattern matching.

2.5.1 Dynamic Programming Algorithms

Dynamic programming (DP) algorithms have been the first approaches to solve the k-mismatch problem. They use a m×n matrix and were introduced by Sellers [Sel80] with a time complexity of *O(nm)*. There are many results, which improve this algorithm and use the advantages of the geometrical properties (i.e.

values in neighbour cells differ at most by one) of the DP-matrix. Based on these enhancements and a further contribution of Ukkonen (CutOff mechanism - example 2.4) [Ukk85], Landau and Vishkin [LaV88] [LaV89] developed an algorithm that introduced the diagonal transition approach. They used the fact, that the diagonals in DP matrixes are steadily increasing and achieved a time complexity of *O(nk)*. In 1989 Galil and Park [GaP90] presented an algorithm, that used the same knowledge, but is more practical *O(kn)*. The latest and best algorithm was developed by Chang and Lampe and used the method of 'Column Partition', it has a time complexity of $O(kn/\sqrt{a})$ [ChL92]. Table 2.19 shows the time and space complexity of all here mentioned algorithms.

Example 2.4 *To illustrate Ukkonen's CutOff mechanism let us assume we have a standard trie, search string 'sand' and edist is limited to one. This algorithm calculates for each prefix the minimum number of edit operations.*

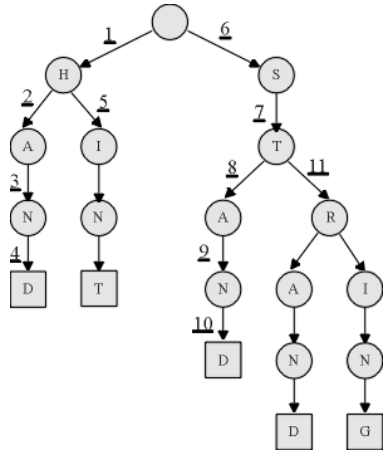

Figure 2.9: Graph: Ukkonen's CutOff mechanism

Searchpath	Prefix	edist	Action
1	H	1	accepted
2	HA	1	accepted
3	HAN	1	accepted
4	HAND	1	hit
5	HI	2	cutoff
6	S	0	accepted
7	ST	1	accepted
8	STA	1	accepted
9	STAN	1	accepted
10	STAND	1	hit
11	STR	2	cutoff

Table 2.18: Ukkonen's CutOff mechanism

Algorithm	Search Time		Preprocessing	Extra
	Worst Case	Avg. Case		
Sellers	*O(mn)*	*O(mn)*	-	*O(mn)*
Ukkonen (CutOff)	*O(mn)*	*O(kn)*	-	*O(m)*
Galil/Park	*O(kn)*	*O(kn)*	$O(m^2)$	$O(m^2)$
Chang/Lampe	*O(mn)*	$O(kn/\sqrt{a})$	*O(ma)*	*O(ma)*

Table 2.19: Dynamic programming: time and space complexity [MiM02]

2.5.2 Algorithms Based on Automata

This area is also rather old. It is interesting because it gives the best worst-case time algorithm *O(n)*, which matches the lower bound of the problem). However, there is a time and space exponential dependence on m and k that limits its

practicability[3]. Ukkonen achieves this time-complexity with his proposed development in 1985 [Ukk85]. The main drawback was the construction time and space complexity of $O(min(3^m, m(2ma)^k))$ for the necessary deterministic finite automaton. This solution is only possible for small patterns and lower error levels.

One of the main objectives of all kinds of algorithms -based on the simulation of automata- is the reduction of space complexity. The most important solutions were developed by Kurtz [Kur96] and Navarro [Nav97B] and Wu, Manber and Myers [WMM96]. The latter are based on the four russian technique [ADKF75] and use Ukkonen's CutOff mechanism. This algorithm is developed for large patterns as well as large alphabets and is the fastest string matching algorithm [Sta02] (cf. table 2.20).

Algorithm	Search Time		Prepro.	Extra
	Worst Case	Avg. Case		
Kurtz/Navarro	$O(n+n(min(t,n)))$	$O(n+mt(1-e^{(n/t)}))$	-	$O(min(a,n)$ $min(m,a))$
Wu/Manber/ Myers	$O(mn/log\ n)$	$O(kn/log\ n)$	$O(ma)$	$O(n+(ma/log\ n))$

Table 2.20: Algorithms based on automata: time and space complexity [MiM02]

2.5.3 Bit Parallelism

Algorithms based on *bit parallelism* (BP) use the special properties of a computer word (e.g., 32bit). These techniques are especially constructed for patterns which fit in one word. Further developments lift this restriction with the help of partitions and can be applied to longer patterns [Sta02]. The costs for this enrichment is the lower performance.

BP was introduced by Bates in 1989 [Bae89] and is separated in the two approaches: parallelize automaton and parallelize matrix. Each of them is based on one of the former mentioned algorithms (automata, DP-matrix). They are taking advantage of the intrinsic parallelism of the bit operations inside a computer word. By using this fact cleverly, the number of operations that an algorithm performs can be cut down by a factor of the computer word, the acceleration is very significant in practice and will further improve with technological process[3].

Table 2.21 (cf. [MiM02]) shows the time and space complexity for the most important approaches. The researcher are Wu and Manber [WuM92], Baeza [BaN96] [BaN99], and Myers [Myr98] [Myr99].

[3]quoted from [Nav01]

	Search Time		Preprocessing	Extra
	Worst Case	Avg. Case		
Wu/Manber	$O(kn\lceil m/w \rceil)$	$O(kn\lceil m/w \rceil)$	$O(ma + kn\lceil m/w \rceil)$	$O(ma)$
Baeza	$O(n)$	$O(n)$	$O(a + m\ min(m,a)$	$O(a)$
Myers	$O(mn/w)$	$O(kn/w)$	$O(ma)$	$O(a)$

Table 2.21: Bit parallelism: time and space complexity [MiM02]

2.5.4 Filtering Algorithms

Filter algorithms (FA) focussed on text passages, which are unimportant for the specific request and do not provide any hits. Under certain conditions FA can ignore large passages and therefore achieve a good performance. But all algorithms have in common that they use existing methods to analyse the remaining text and are most effective with small error levels (k-distance).

FA are based on the subpattern technique. The searched pattern is split in several subpatterns, which are compared with a part of the text. Is the subpattern not included in this passage, this part can be ignored. This method allows that not the whole pattern has to be checked. In case of using DP-matrixes this method saves a lot of time. However, if the k-distance becomes too large, too many comparisons have to be done and the performance will decrease.

This area of string matching methods is the youngest of all four approaches and still very active. The most important solutions are from Wu and Manber[WuM92], Tarhio and Ukkonen[TaU93], Baeza[BaP96], and Navarro[Nav97A] (cf. table 2.22 [MiM02]).

Algorithm	Search Time		Preprocessing	Extra
	Worst Case	Avg. Case		
Tarhio/Ukkonen	$O(mn/k)$	$O(a/(a-2k))$	$O((k+a)m)$	$O(ma)$
Navarro	$O(mn)$	$O(n)$	$O(a+m)$	$O(a)$
Wu/Manber	$O(mn/w)$	$O(mn/w)$	$O(a+m)$	-
Baeza	-	$O(n,k \leq m/log\ n)$	$O(m)$	$O(m^2)$

Table 2.22: Filtering algorithms: time and space complexity [MiM02]

Chapter 3

Previous and Related Work

This chapter presents two studies and an improvement for trie structures. The studies are about data clustering and secondary storage of trees. The trie improvement called hybrid TST. As these further developments and studies were essentially based on the formerly explained methods, all here mentioned approaches are presented in a rather compact way.

Former evaluation of tries and approximate string matching algorithms are not explicit presented in this masterthesis: on the one hand the test parameter of all studies differs (e.g., CPU, RAM, implementation language), on the other hand so many different special solutions (of common methods) are available, which not allows a simple comparison of measurements. Detailed evaluations are done by Navarro [Nav01], Michailidis and Margaritis [MiM02] as well as by Stamme [Sta02].

3.1 Trie Developments

3.1.1 String B-Tree

The string B-tree is a mix of the well known B-tree and the PATRICIA trie and was developed by P. Ferragina and R. Grossi [Far97]. The objective of the string B-tree development was to combine both approaches for the usage as external index structure especially for strings: on the one hand the PATRICIA trie with its excellent characteristics in string matching and space performance (cf. section 2.4.1 on page 23), on the other hand the B-tree as successful external index structure. The uniting of both methods reduces at the same time the drawback of the unbalanced trie structure.

The efficiency of the B-tree depends on the size of the internal nodes and their degree. The more elements one node includes, the more branching arms exist.

This results in lower levels and influences the number of external data requests. In case of using numbers this method is almost perfect, because the comparison of numbers within one node is very fast. In case of using strings this proceeding is very time consuming. Firstly, strings have different lengths and it is therefore impossible to load one block for each node. Secondly, the substitution of the original string with its references is also very time consuming, because this results in one external data request for each comparison.

The space efficiency of the PATRICIA trie solves the problem. In the String B-tree each internal node is represented by on single PATRICIA trie [Sch02] (cf. figure 3.1). A detailed analysis is presented in the study [Sch02].

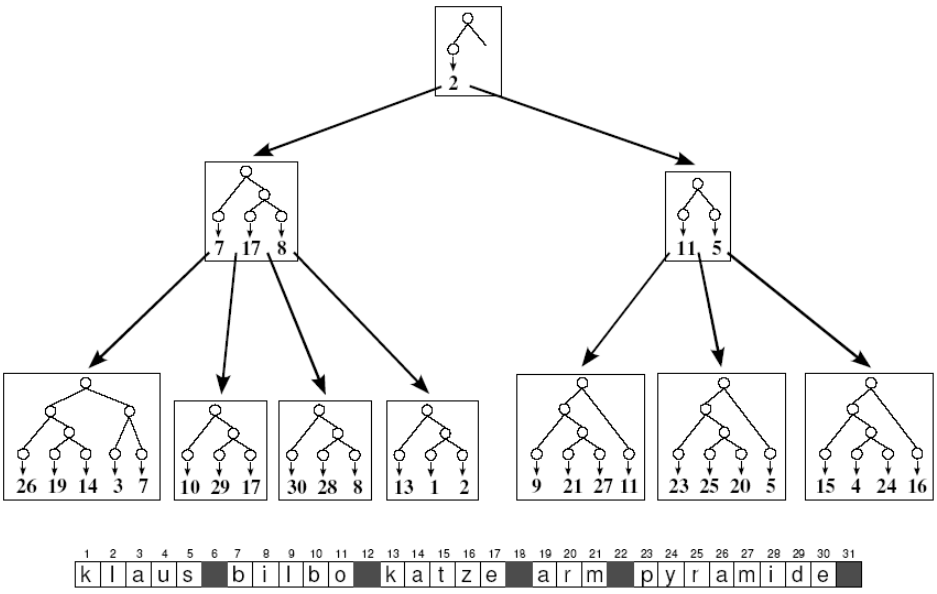

Figure 3.1: String B-tree (taken from [Sch02])

3.1.2 Ternary Search Trie

Definition 3.1 *The ternary search trie is used to save strings with an unlimited number of characters and is part of the trie family. Each node saves exactly one character s_i and has at most three children. Strings where the i-character is lower than s_i are saved in the subtrie of the left child; strings where the i-character is greater than s_i are saved in the right subtrie. Strings where the character i is equal to s_i are saved in the third (middle) subtrie.*

The *ternary search trie* (TST) -developed by Bentley and Sedgewick- is the latest development level in the trie history and based on n-ary tries and digital search tries (e.g., PATRICIA trie - cf. figure 2.5 on page 24). It works with the digital transformation of each character and reduces the drawbacks of n-ary tries (many children have no successor nodes). The TST includes the advantages of a digital

search trie (e.g., being space-saving) and offers fast access to all nodes. These characteristics make the TST a trie, that is well suitable in saving common strings and supports all string operations in a fast way [Sed03]. Furthermore, the TST is known as a well balanced trie, whose depth depends on the number of stored strings s and on the maximum length n of the strings. The TST has a maximal depth of $\lceil log(s) \rceil + n$ (evaluation based on Bentley and Sedgewick [BeS97]). A very theoretical analysis of the TST is presented in the study [Cle97].

Derived from the TST, the hybrid TST enhanced this trie with a fix array as the header (cf. figure 3.2). In its basic version, each cell presents one character and leads to an 256-array considering all ASCII characters. The more strings are saved in the hybrid TST, the more start character should one array cell include. The header (root) implementation improves the time for selected string operations (e.g., exact matching, insertation) as well as the balance of the trie and presents the hybrid TST as one of the fastest trie in these fields [Sed03].

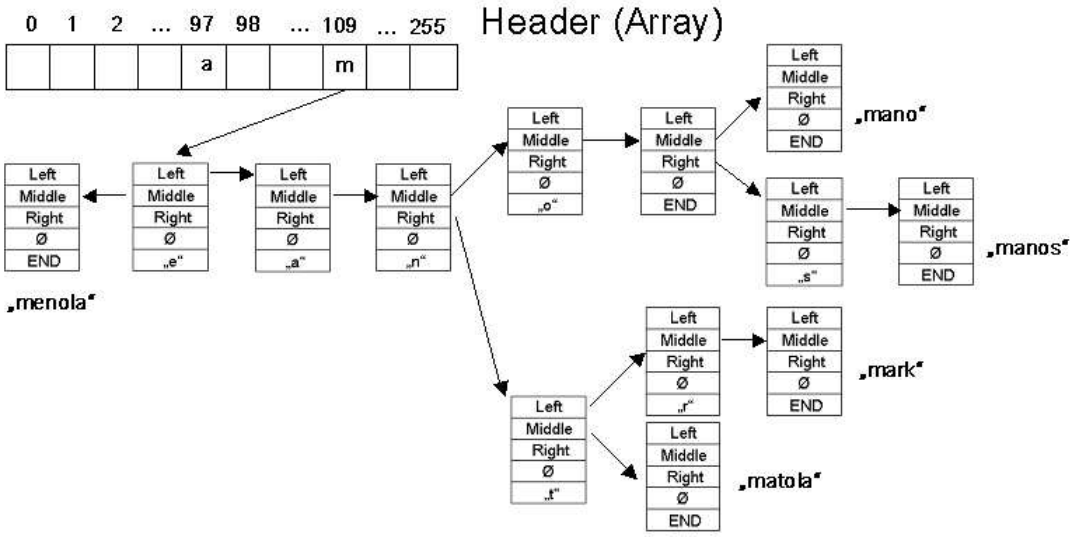

Figure 3.2: Structure hybrid TST

3.2 Secondary Storage with Trees

The string B-tree was developed as an index structure especially for the external usage and is a mix of the PATRICIA trie and the B^+-tree (see above). T.Ferragina and R.Grossi substantiate their solution with the general structure and the lower balance of trees (and tries), which do not support an efficient and independent implementation.

Muhs describes in his thesis an efficient and independent solution to use suffix trees for secondary storage [Muh03]. In his study he compares different string indexes, explains external storage systems, describes approaches for the external

creation of tree structures and implements an individual suggestion. In the first step he evaluates suffix trees, suffix arrays and string B-trees. Only the suffix tree supports good performance in exact and approximate matching. The string B-tree has with its included PATRICIA trie some drawbacks in approximate string matching. Furthermore, Muhs states, that for secondary storage not the structure determines the performance of trees, but their algorithm. All these facts lead to the decision to proceed the next steps with the suffix tree and to enhance this index structure for external usage.

The thesis includes a general description of the construction rules of suffix trees and describes the application of different string operations like exact and approximate matching. In this context he explains that the paging characteristic of trees depends on the compression and on the number of nodes. Based on Gil and Itai [GiI99], he describes the basic packing methods for trees (and their administration), which are necessary to reduce the I/O activity. This leads to an adaption of the traditional tree structure and substantiates the new dynamic tree structure - optimised for external usage. Furthermore, Muhs introduces three construction rules for the new tree structure, which differ from traditional methods: All of them have in common that they are based on the parallel insertation of strings to reduce the I/O activity.

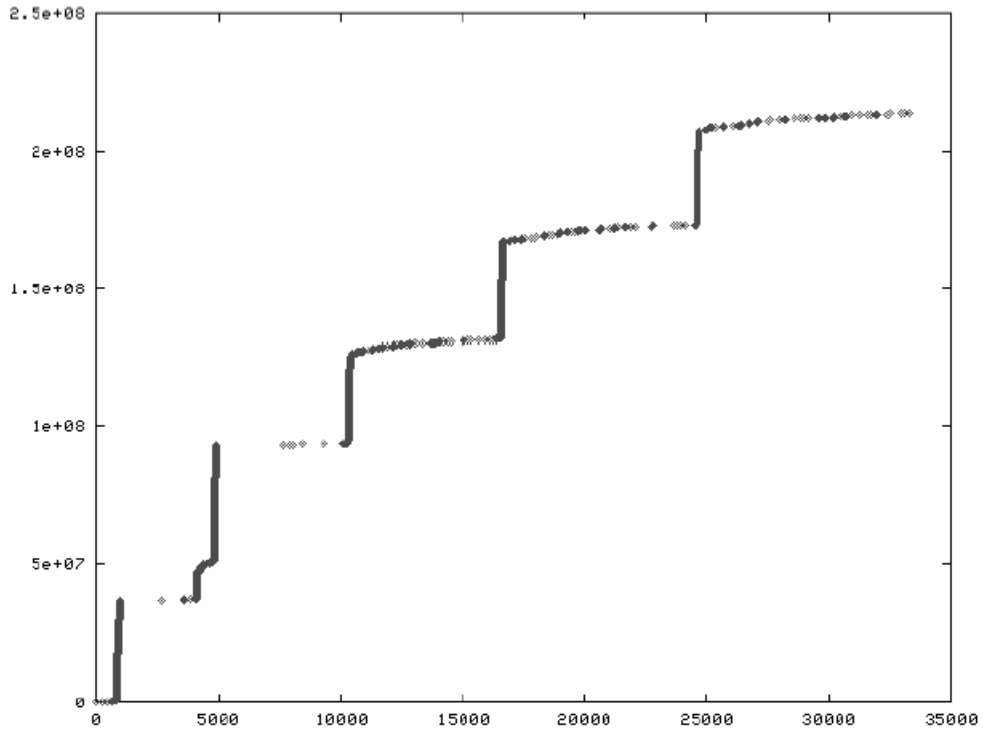

Figure 3.3: Insertation process with the quasi-parallel method (taken from [Muh03])

His thesis closes with an evaluation of one of the three algorithms, and proceeds some tests with exact and approximate string matching. Figure 3.3 shows the linear insertation process of the quasi-parallel method. The time is assigned

to the Y-axis, the X-axis represents the inserted bytes. The illustration shows phases with a high activity (concurrently insertations) and phases with a low activity (I/O processes). The more bytes are inserted in the tree, the more the I/O activity increases (longer 'stairs'). The evaluation for approximate string matching based on the dynamic programming approach and results request times around 0.15s for k=2,3. Muhs describes its complete C/Phython implementation as successful and useable for realtime applications. A more detailed analysis is presented in the original study.

3.3 Special Algorithm for Data Clustering

Information fusion is the process of integration and interpretation of data from different sources in order to derive information of a new quality. Lujan-Mora and Palomar developed an approach that clusters similar sentences (short, strings) based on an individual threshold, and substitutes each value of one cluster with its unique form [LuP01]. The algorithm homogenises strings and solves problems with word order, and different translations (i.e., misspelling). Problems with numbers, synonyms and different languages are still open and not handled in this study.

The algorithm is defined as follows:

1. read all strings of the database: (a) read a string, (b) expand abbreviations and acronyms, (c) remove accents, (d) shift string to lower case, (e) remove stop words, and (f) store the string (if it has been stored previously, its frequency of appearance is increased by one unit),
2. sort all strings in descending order, by their frequency of appearance,
3. cluster all similar strings,
4. verify the generated clusters for errors, and
5. update the original data with its unique form (centroid).

The core of the study is the clustering process. The clustering algorithm chooses the strings from greatest to smallest frequency of appearance, since it assumes that the most frequent strings have a greater probability of being correct, and thus, they are taken as being representative for the rest. The approach compares in succession each string with the current centroid of each existing cluster, based on the predefined threshold. The centroid is defined as the unique form of all assigned strings to one cluster, and has to be recalculated each time when a new string joins the cluster. The comparison of strings is based on a predefined

threshold and is proceeded with special methods (basis for the calculation of string similarity; e.g., combined levenshtein distance MIDWP), which handle strings of two or more words. Each string is broken up into its words - the idea is to pair the words so that the sum of the, for example, levenshtein distance is minimised. The stronger the definition of the threshold is, the more clusters are created. If a comparison of one string with all centroids is not successful, a new cluster will be created. The major drawback of this approach is the bad runtime, which allows no usage for realtime applications.

Lujan-Mora and Palomar have evaluated this clustering approach for all four algorithms, which compare strings with more than two words. Figure 3.4 presents the precision (cf. 2.3 on page 12); figure 3.5 shows the error rate. In general, both figures have identical shapes for all algorithms. The following dependency is presented: the higher the tolerance of the threshold, the more increases the error rate. Lujan-Mora and Palomar recommend a threshold between 0.1 and 0.3, but the exact value depends on the data set. A more detailed analysis is presented in the original study.

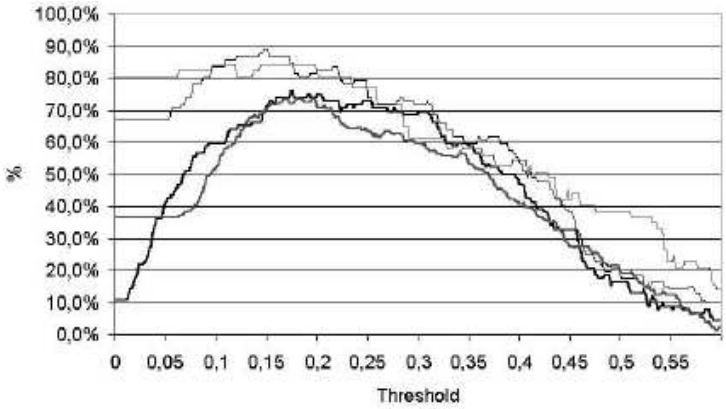

Figure 3.4: Data Clustering with Lujan-Mora and Palomar - precision (taken from [LuP01])

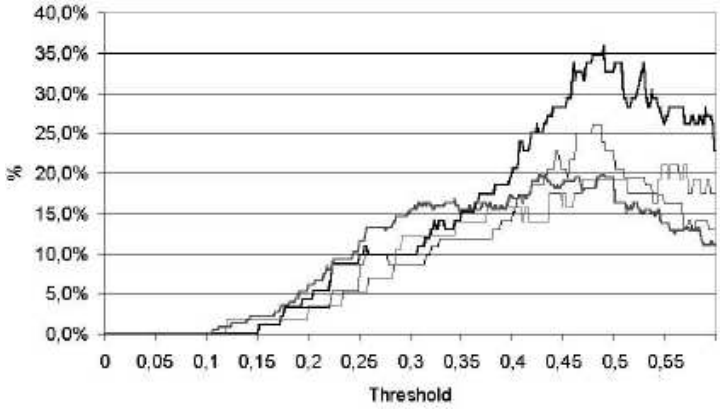

Figure 3.5: Data Clustering with Lujan-Mora and Palomar - error rate (taken from [LuP01])

Chapter 4

String Similarity Predicates

This chapter presents the core of this masterthesis and improves existing knowledge for approximate string matching supported by techniques of the IT and CL. It combines one of the fastest index structure for common strings with methods to make homogenised strings comparable. This section explains how to achieve better results in approximate string matching and introduces a general approach for application-specific string similarity predicates. Furthermore it implements the approach as a Java prototype and evaluates the results.

All here mentioned methods were described in the former chapters and are not repeated here. For this reason some text passages of this chapter do include references to the relevant sections.

4.1 Introduction

This section summarises the last two chapters and focussed on important methods for the next steps. The major parts to achieve the objective of the masterthesis are the homogenization of strings, the usage of word lists and the support by index structures. For perfect pretreatment of text -to make strings and their context comparable- the process should be the following:

1. homogenization of each single token,
2. expanding of acronyms and abbreviations,
3. replacement of each single word with its lemma,
4. substitutions of synonyms, and
5. deleting of stop words.

The achievement of the defined goal heavily depends on the effort of implementation and on the maintenance of each single word list. In best cases the preprocessing leads to a quality of 95%-99% [Zie00]. The remaining percents are mostly based on mistakes in the disambiguating of dots and in a false substitution of synonyms. The working with lemmas is only useful if large tables exist (table lookup method), which include all types of words. Only this method extensively solves problems with irregular verbs (cf. 2.3.2 on page 16). The main drawback is not the required space on external storage systems, but the maintenance of each single assignment (languages are active). Other methods (algorithms) like phonetic transformations or stemming are easier to handle (no maintenance), but do not deliver the same quality. For this reason, a mix of the latter mentioned methods improves the search for application-specific information. In the contrast to traditional index structures, tries and trees are developed for the storage of strings. They are efficient in time and space (the length of the string has no influence) and can be used for external storage systems. Besides that, these index structures support different kinds of string searching operations. For example, some of them efficiently support approximate string matching and other ones are better for pattern matching. Suffix solutions (cf. 2.4.2 on page 24) save all suffixes of a string and are ideal to search (approximate) patterns of variable length in large preprocessed texts. The most tries only store the string once and are perform therefore better on time and space (average). Some tries store the complete string in its structure, PATRICIA tries store only parts of the strings and refer in their leaves to the whole string (cf. 2.4.1 on page 23). PATRICIA tries are most efficient in the space complexity, but not the best choice for fast approximate string matching. One of the best tries, that combines all features for string matching is the hybrid TST (cf. 3.1.2 on page 30). Its root can be individually customised and all nodes have the same size. However, less documentation of this tree exists and has not yet been evaluated for approximate string matching.

The approximate string matching is supported by several approaches (cf. 2.4 on page 22). Some of them are not applicable to preprocessed texts and are optimised to special problems. For example: algorithms based on bit parallelism are most effective for patterns, which fit in one computer word; filter algorithms are specialised for plain text and are focused on the identification of irrelevant text passages. All of them have in common that the process to compare the similarity of two strings is reduced to a 2-dimensional matrix (DP-Matrix or diagonal transition approach). Ukkonen's CutOff mechanism supports approximate string matching in tries. This algorithm can be applied to prefixes of strings and determines for each level the minimum number of edit operations.

4.2 Concept of String Similarity Predicates

The objective of this masterthesis is to enlarge the knowledge of index supported string similarity predicates. Therefore string transformation rules from the CL will be combined with the hybrid TST. The approximate string matching will be conducted with Ukkonen's CutOff mechanism (cf. 2.5.1 on page 25).

Building on temporary summary, this masterthesis provides the following own contributions:

- evaluation of the hybrid TST for different string matching operations,
- index based approach for application-specific string matching, and
- analysis and evaluation of word lists.

The general approach is supported by three kinds of word lists and applies several transforming rules to one string. The stop word list can be adapted from the frequency list of the BNC and has to be enriched with useless words, which often occur in the specific area. In general, the most frequent words of the BNC are expletives. Abbreviations can also be taken from the BNC, but must be completed with abbreviations of the specific area. If the abbreviation list is not enriched with words of the specific area, the application-specific search for information would result in a low quality of the hitlist. The same is valid for acronyms (cf. 2.2.3 on page 10). Especially the last two word lists have in common that they can be adapted from lists of general corpora, but must be enriched with words of the specific area. This can be done either by using special corpora, or by preprocessing of the specific text and the creation of local abbreviation and acronym lists. In a next step synonyms have to be substituted, and this process should be handled very carefully.

For improving application specific string matching, several transforming rules will be applied to one string and connected by a simple OR operation (better recall). Therefore it is necessary to create different index structures for each transformed string (e.g. SOUNDEX, stemming). For saving strings of variable length the hybrid TST is the best choice; numbers or strings with identical length can be stored in the B^+-tree (or hash index). Further information, which are not included in the trie structure are saved in an so called infoobject (e.g., original data). This infoobject is created once and can be accessed by the references of the specific index structure (cf. figure 4.1).

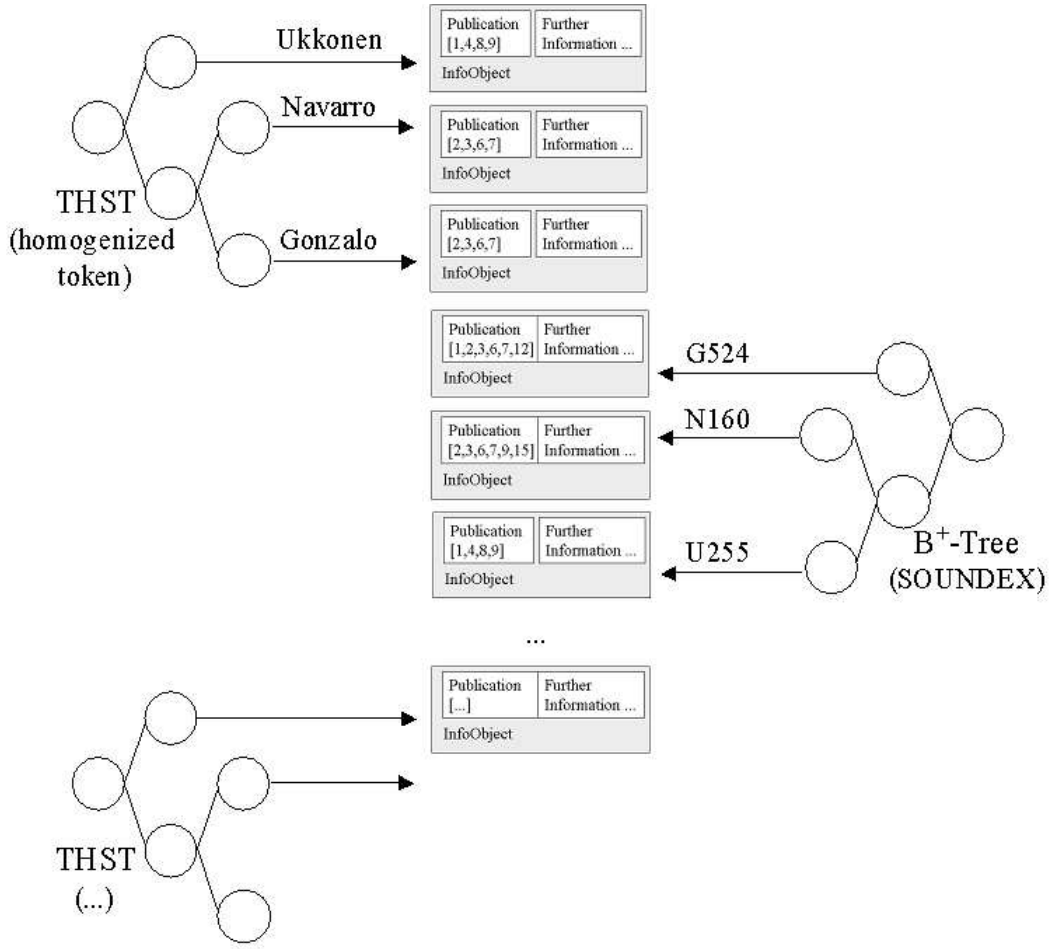

Figure 4.1: Index structures of the general approach

4.3 Prototyping

4.3.1 Preparation

The preparation includes two aspects: the selection of text data and the creation of word lists. Because of the affinity of the most mentioned methods to the English language, the data of the *digital bibliography & library project* (DBLP) was selected. The DBLP provides bibliographic information on major computer science journals and proceedings. The 250 MB data volume is structured as XML and allows evaluation of word lists with authors (names) and titles. One dataset (author and title) is defined as a publication.

Based on the data of the DBLP, the creation of word lists is the other part of the preparation. The stop word list is taken from the general frequency list of the BNC and mainly includes the first 100 entries. In this case the general frequency list of the DBLP predominantly consists of verbs and allows only the selection of few words (e.g., based). The abbreviation and acronym lists base on common entries of the BNC and of much more special entries in the DBLP. Appendix

A includes the analysed word lists of both data sources. All DBLP word lists were created manually by a preprocessing of the plain XML file. Only this step guarantees an improvement of the application-specific hitlist.

4.3.2 Java Implementation

For evaluation reasons, the Java prototype implements the hybrid TST and the SOUNDEX code. The former supports exact and approximate matching. The latter exemplary shows the possibility to combine several matching rules and is stored in a B^+-Tree. Figure 4.2 shows the general structure of the implementation. Author and titles are saved in different and independent index structures, each singles publication is saved with an unique index in a hash. Each leave (hybrid TST, B^+-Tree) refers to an infoobject, which includes the relevant publications. The hitlists combine all sub-hitlist by a simple set operation (OR).

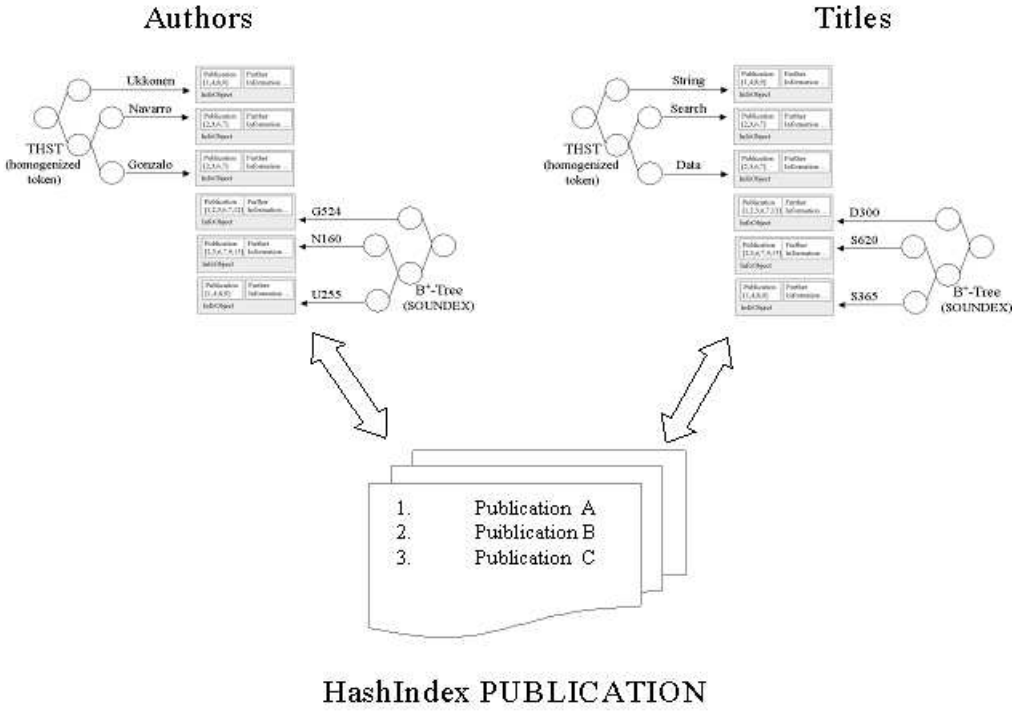

Figure 4.2: Structure of the implementation

Figure 4.3 presents the developed Java prototype. In the first step the different word lists are loaded, the second step the DBLP data (i.e. tokenising, homogenisation, expanding of acronyms). The application provides approximate string matching as well as the usage of the SOUNDEX and can be easy adapted for further improvements (e.g., stemming). This figure shows a request based on SOUNDEX (hitlist: Nafarieh) and levenshtein (hitlist: e.g. Naval). The prototype supports requests with an unlimited number of parameters (Author:

Navaro, Title: measur, fuzy) and calculates the string similarity to all index values for each of the parameters. The appendix B on page 58 presents some more examples (e.g, acronyms, abbreviations).

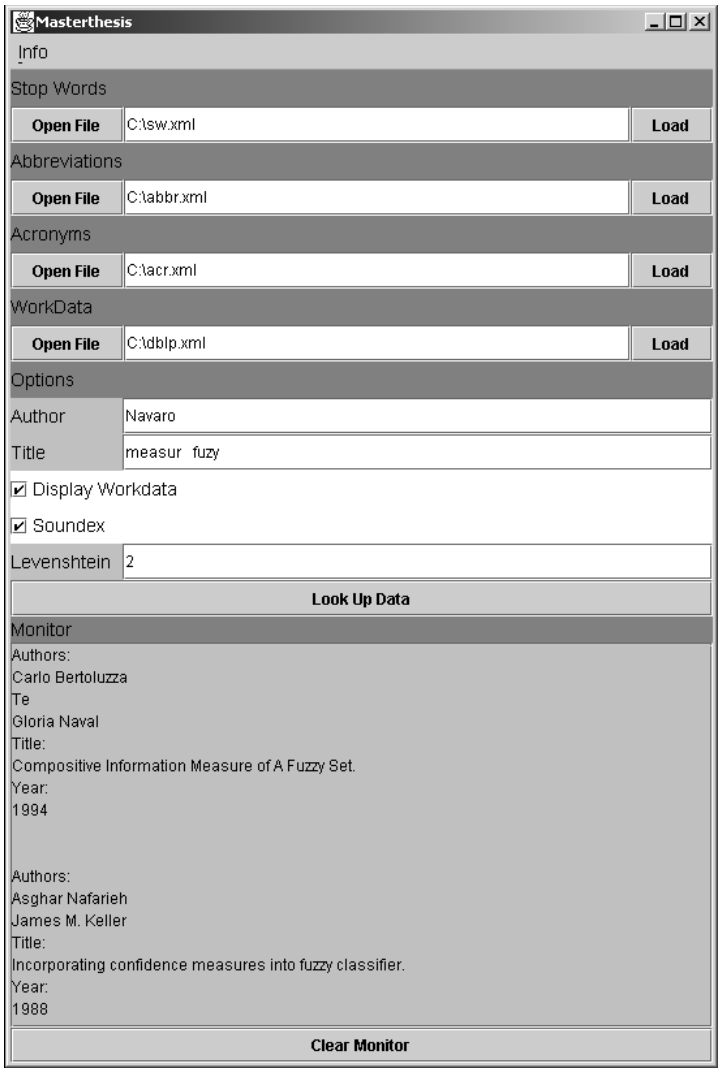

Figure 4.3: Java implementation - prototype

4.4 Experimental Results

4.4.1 Hybrid TST Evaluation

The following tables and graphs show the different evaluations of the hybrid TST. The test environment is: IBM/PC Windows 2000 (SP4), Intel CPU 1.6GHz, RAM 1GB. The data consists of: S(#) {1,000,000; 2,000,000; 4,000,000}, n{1...20}, a {26}. The sizes of m are various and depends on the evaluated single string operation.

Insertation

Figure 4.4 (cf. table 4.1) shows the linear insertation process of the hybrid TST for 4,000,000 unique strings and fulfils the *O(n)* expectations of tries. The hybrid TST insertation process was optimised in two steps: in the first step the data was loaded from the file in a dynamic Java object (ArrayList), in the second (monitored) step each string was inserted from the memory in the hybrid TST. The break by around 1,668,000 insertations caused by the internal memory and object administration of Java. Many tests with different numbers of strings and different orders of strings lead always to a small break. The insertation process without the preprocessed loading in the memory takes for all 4,000,000 strings around 40,000 ms and shows more than five breaks.

10^6	Time	10^6	Time	10^6	Time	10^6	Time
1	421 ms	11	4,727 ms	21	14,762 ms	31	19,128 ms
2	842 ms	12	5,158 ms	22	15,182 ms	32	19,559 ms
3	1,282 ms	13	5,608 ms	23	15,623 ms	33	19,999 ms
4	1,703 ms	14	5,999 ms	24	16,074 ms	34	20,460 ms
5	2,174 ms	15	6,440 ms	25	16,484 ms	35	20,850 ms
6	2,574 ms	16	6,870 ms	26	16,965 ms	36	21,311 ms
7	2,985 ms	17	13,019 ms	27	17,365 ms	37	21,752 ms
8	3,445 ms	18	13,450 ms	28	17,786 ms	38	22,192 ms
9	3,836 ms	19	13,880 ms	29	18,257 ms	39	22,653 ms
10	4,307 ms	20	14,301 ms	30	18,657 ms	40	23,084 ms

Table 4.1: Hybrid TST: measurements for insertation

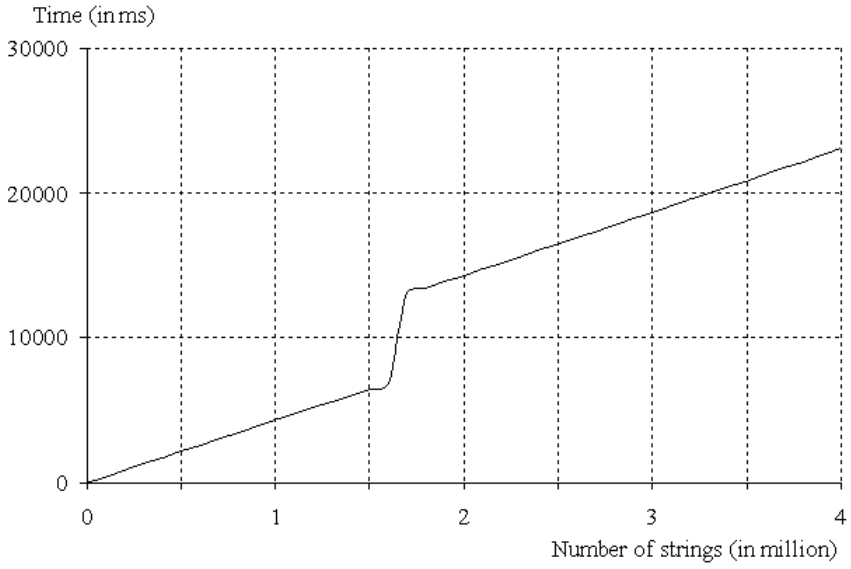

Figure 4.4: Hybrid TST: graph for insertation

Exact matching

Figure 4.5 (cf. table 4.2) presents a comparison of different exact matching requests for all three hybrid TST volumes. The time is assigned to the Y-axis, the X-axis represents different lengths of strings. Each measurement includes 10,000 exact matching requests. The graph shows that the time for the exact matching process depends a little bit on the volume of the trie (average); the zigzag course caused by the Java measurement and can be neglected. The figure shows that the time for one exact matching request is minimal and takes around 0.015 ms in memory.

m	#1,000,000	#2,000,000	#4,000,000
2	121 ms	150 ms	160 ms
4	140 ms	140 ms	150 ms
6	131 ms	160 ms	150 ms
8	150 ms	160 ms	141 ms
10	130 ms	130 ms	170 ms
12	150 ms	171 ms	150 ms
14	151 ms	160 ms	160 ms
16	120 ms	140 ms	171 ms
18	150 ms	160 ms	140 ms
20	130 ms	130 ms	170 ms

Table 4.2: Hybrid TST: measurements for exact matching (for 10,000 repetitions)

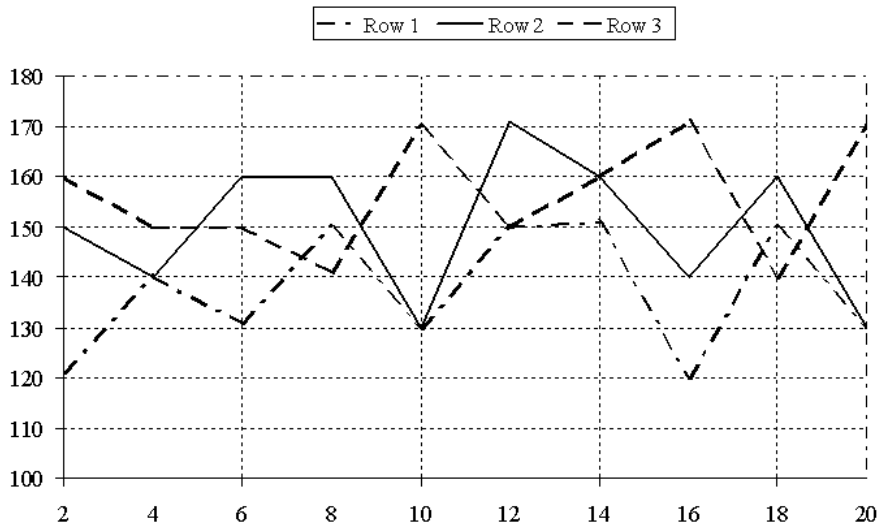

Figure 4.5: Hybrid TST: graph for exact matching (for 10,000 repetitions) - Row1 (n=1,000,000), Row2 (n=2,000,000), Row3 (n=4,000,000)

Approximate String Matching - 1,000,000 Strings

Figure 4.6 (cf. table 4.3) presents a comparison of approximate matching for six different pattern lengths in a hybrid TST of 1,000,000 strings. The time is assigned to the Y-axis, the X-axis represents different k distances. It shows that the request time of each graph (logarithmical scale) increases with the pattern length and the k distance. Furthermore, it shows the exponential dependency of k and m. Another presentation style is included in appendix C on page 61.

k	m=2	m=4	m=6	m=10	m=14	m=20
1	10 ms	16 ms	18 ms	20 ms	22 ms	24 ms
2	126 ms	176 ms	218 ms	274 ms	314 ms	360 ms
3	743 ms	1,043 ms	1,359 ms	1,840 ms	2,149 ms	2,557 ms
4	1,846 ms	2,519 ms	3,563 ms	5,271 ms	6,521 ms	8,125 ms
5	2,557 ms	3,647 ms	5,129 ms	7,799 ms	10,070 ms	13,265 ms

Table 4.3: Hybrid TST: approximate string matching (n=1,000,000)

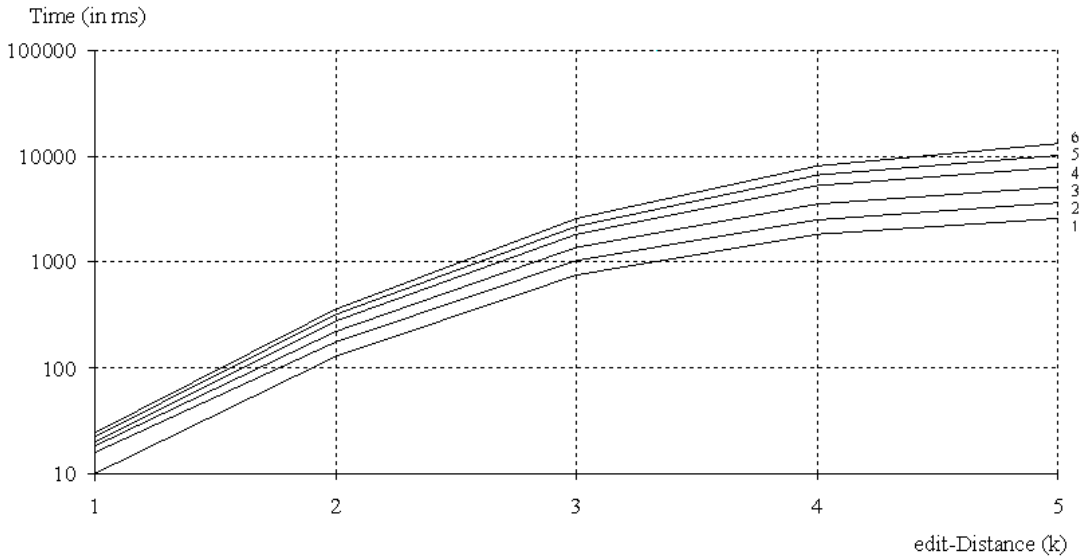

Figure 4.6: Hybrid TST: graph for approx. string matching (n=1,000,000) - 1. m=2, 2. m=4, 3. m=6, 4. m=10, 5. m=14, 6. m=20 - logarithmical scale

Approximate String Matching - 2,000,000 Strings

Figure 4.7 (cf. table 4.4) presents the same dependencies like the evaluation before. The graph shows for k $\leq$ 2 comparable values, the other measurements do considerably increase. Another presentation style is included in appendix C on page 61.

k	m=2	m=4	m=6	m=10	m=14	m=20
1	10 ms	12 ms	14 ms	18 ms	22 ms	30 ms
2	130 ms	208 ms	262 ms	328 ms	374 ms	438 ms
3	981 ms	1,454 ms	1,918 ms	2,499 ms	2,974 ms	3,581 ms
4	2,942 ms	4,222 ms	5,954 ms	8,526 ms	10,801 ms	13,705 ms
5	4,696 ms	6,495 ms	9,173 ms	13,850 ms	17,915 ms	23,554 ms

Table 4.4: Hybrid TST: approximate string matching (n=2,000,000)

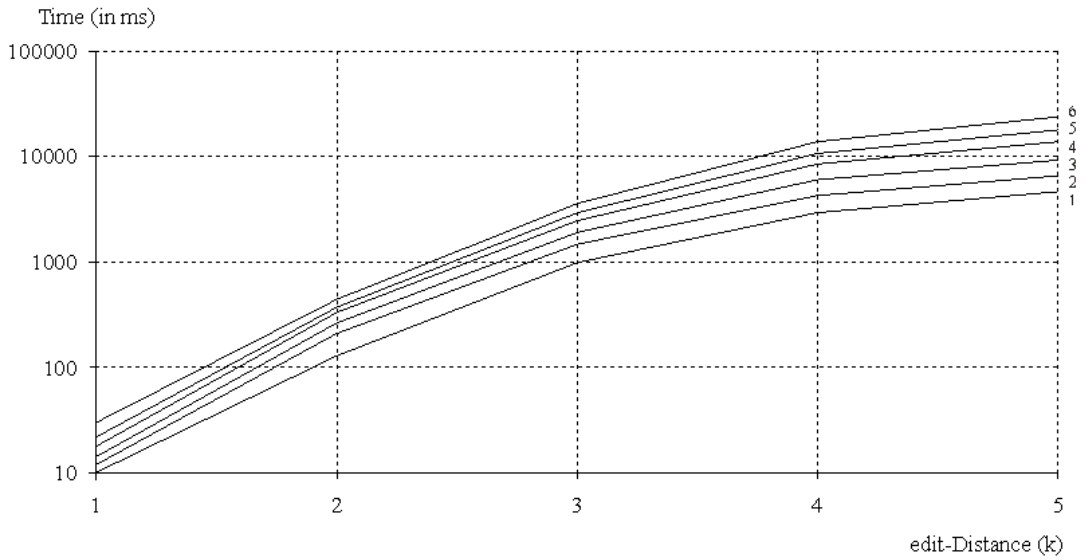

Figure 4.7: Hybrid TST: graph for approx. string matching (n=2,000,000) - 1. m=2, 2. m=4, 3. m=6, 4. m=10, 5. m=14, 6. m=20 - logarithmical scale

Approximate String Matching - 4,000,000 Strings

Figure 4.8 (cf. table 4.5) presents the same dependencies like the evaluations before. The graph shows for k $\leq$ 2 comparable values, the other measurements do considerably increase. Another presentation style is included in appendix C on page 61.

k	m=2	m=4	m=6	m=10	m=14	m=20
1	12 ms	14 ms	18 ms	26 ms	28 ms	36 ms
2	174 ms	302 ms	350 ms	438 ms	502 ms	590 ms
3	1,698 ms	2,669 ms	3,228 ms	4,220 ms	4,987 ms	5,960 ms
4	6,247 ms	8,987 ms	11,735 ms	16,676 ms	20,783 ms	25,885 ms
5	10,124 ms	14,144 ms	19,341 ms	28,635 ms	36,829 ms	48,027 ms

Table 4.5: Hybrid TST: approximate string matching (n=4,000,000)

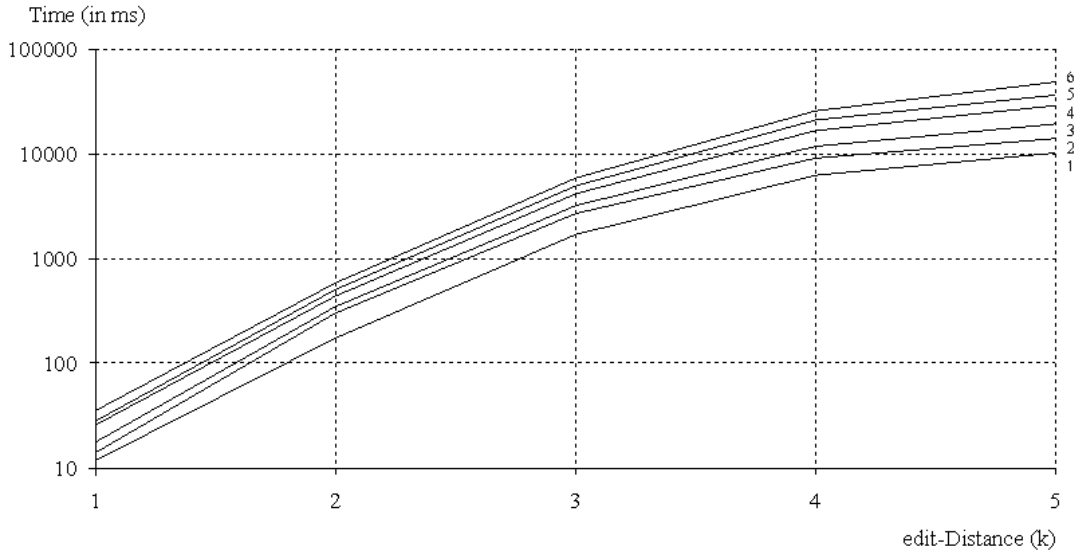

Figure 4.8: Hybrid TST: graph for approx. string matching (n=4,000,000) - 1. m=2, 2. m=4, 3. m=6, 4. m=10, 5. m=14, 6. m=20 - logarithmical scale

4.4.2 Evaluation String Similarity and Word Lists

This evaluation analyses the DBLP data in context to abbreviations, acronyms and stop words. Table 4.6 splits the DBLP data in several word groups. Furthermore, the table distinguishes between authors (names) and titles. In general, the effectiveness of word list depends on the to be analysed data. In case of the titles, the most important step is the expansion of acronyms (e.g., DBMS $\rightarrow$ DataBase Management System). This definitely increases the number of hits and reduces

the number of entries in the hybrid TST (≈21,810). The reason is that almost each expanded word is already in the hybrid TST, some for abbreviations. The using of stop words is very useful, if not the complete title is saved (cf. section 3.3 on page 33). The current implementation of the masterthesis shrinks this advantage. But, the using of a stop word list improves the runtime for complete title requests and reduces slightly the number of entries in the hybrid TST. The application of word lists to names (authors) is less useful. The expansion of initials (acronyms) is almost impossible (without the acceptance of mistakes) and the number of stop words as well as abbreviations can be neglected.

The combination of approximate string matching and SOUNDEX has increased the total number of hits. The quality of this hitlist depends on the operation mode: an OR operation shows the join of both hitlists (better recall), an AND operation presents the intersection and improves the quality (better precision). An evaluation, just based on quantitative statements (cf. 3.3 on page 33) is a task for the further work.

Word Type	Set	Author	Title	Σ
abbreviations	all	0	1,386	1,386
	grouped	0	535	-
acronyms	all	501,092	120,319	621,411
	grouped	100	21,810	-
stop words	all	1,432	1,178,459	1,179,891
	grouped	68	112	-
other	all	2,627,880	3,546,642	6,174,522
	grouped	59,313	32,428	-
Σ	all	3,130,404	4,846,806	7,977,210
	grouped	159,481	54,885	-

Table 4.6: Word groups of the DBLP

Chapter 5

Conclusion and Further Work

The conclusion and the further work of the masterthesis can be separated in two parts: on the one hand the quantitative evaluation of the hybrid TST, on the other hand the qualitative evaluation of the hitlist.

The first part is about the profit of the hybrid TST and its usage for approximate string matching. It is a complex trie structure that allows all kinds of string matching methods. The insertation of 4,000,000 strings shows a linear time complexity, the exact matching depends slightly on the data volume of the hybrid TST. Based on the CutOff mechanism the results for approximate string matching are better than with PATRICIA tries (cf. [ShM95]), but worse for k $\geq$ 3 (in context to realtime application). With its scalable root array it can be better balanced than other tries and presents a good trie structure for all kinds of strings and matching problems. The current implementation has a one-letter root array, but for professional applications it should be extended to three or four letters (depending on the total numbers of strings). That should improve the times for all string operations, but with the slight drawback of a larger and fixed array. Open work tasks are easy to find. Interesting aspects are the dependencies between the size of the alphabet, the number of total tokens and the size of the root array. What are the best adjustments for the hybrid TST? What are the best parameters for an usage as external data storage and how does a C/C++ (assembler) implementation improve the performance?

The second part evaluates the hitlist. The masterthesis shows that the quality of the hitlist sinks or rises with the maintenance of the used word lists, the precision of the applied methods to homogenise strings and the number of used methods to make strings similar. The stop word list can be easily adapted from frequency lists of corpora. The abbreviation list and the acronym list have to be - application-specific - created manually, same for synonyms. In this work many technical acronyms consist of numbers and letters, besides that, they have been separated by a dot (e.g., X.500). This fact is also important for the tokenising

process. The homogenization of strings is a very sophisticated process and should be handled with care. In this masterthesis the tokenising process does not include all possible steps, but achieves also good results. Further interesting points are the transforming methods to make strings comparable. This work uses the levenshtein distance and the SOUNDEX code. The hitlist combines both sub-hitlists by a simple OR operation. An improvement of the precision of the hitlist can be achieved with further transforming methods and a combination of all sub-hitlists by a simple AND operation (intersection). This approach should be analysed in further work with the help of the stemming process (e.g., affix removal). Another open task is the definition of a comprehensive and objective threshold to determine the quality of a hitlist. However, this is a rather visionary (approximate) objective, due to the dynamic of languages.

This masterthesis shows that it is possible to combine methods from IT with methods from CL. Furthermore, the hybrid TST achieves good results for all evaluated string operations and fulfils the requirements for the usage in realtime applications. All mentioned activities focused on English written text, but in most cases it can be easily adapted to French, German or Spanish texts. This implies that languages which assign different meanings to the same word (e.g., Thai), can not be analysed. In this case, the meaning of the word depends on the pronunciation and emphasis. The combination of all languages in one system has to be the primary objective, even if it will never be achieved.

Appendix A

Word Lists

Abbreviation List of the DBLP

No.	Abbreviation	Frequency	Substitution
1.	802.11	19	IEEE Wireless LAN protocol
2.	u.s	14	United States
3.	.net	9	Microsoft XML Web Services platform
4.	asn.1	5	Abstract Syntax Notation One
5.	x.500	5	CCITT Directory Services Protocol
6.	x.25	4	CCITT Packet Switching Protocol
7.	x.400	3	CCITT Message Handling Protocol
8.	802.11b	3	Wireless LAN Equipment Standard update
9.	x.509	2	Standard for Public-Key Infrastructure
10.	eds.	2	edition

Table A.1: DBLP: abbreviation list (frq. per million words)

No.	Abbr.	Frq.	No.	Abbr.	Frq.	No.	Abbr.	Frq.
11.	z39.50	2	21.	802.5	2	31.	802.3	1
12.	i.t	2	22.	u.k	2	32.	c.e	1
13.	802.6	2	23.	802.14	2	33.	802.12	1
14.	h.264	2	24.	ieee802.11	2	34.	e.c	1
15.	ed.	2	25.	d.c	2	35.	ph.d.'s	1
16.	w.r.t	2	26.	al.'s	2	36.	e.v	1
17.	i.i.d	2	27.	802.4	1	37.	inc.	1
18.	ph.d	2	28.	comp.spec.	1	38.	x.21	1
19.	h.263+	2	29.	a.d	1	39.	dot.com	1
20.	802.11a	2	30.	802.11e	1	40.	etc.	1

Table A.2: DBLP: continuation abbreviation list (frq. per million words)

Abbreviation List of the BNC

No.	Abbr.	PoS.	Variants	Frq.	Substitution
1.	no	Det.	no.	1342	number
2.	Mr	NoC.	M.R., Mr, Mr.	674	mister
3.	no	NoC.	no.	662	number
4.	per cent	NoC.	per cent, per cent.	384	per centum
5.	Mrs	NoC	Mrs, Mrs.	221	mistress
6.	hon	Adj.	hon., hon	106	honor
7.	no	NoC.	no, no., nos	100	number
8.	fig	NoC.	fig., fig, figs	78	figuratively
9.	etc	Adv.	etc, etc.	74	et cetera
10.	eg	Adv.	e.g., eg, eg., e.g	71	exempli gratia
11.	ie	Adv.	i.e., ie, ie., i.e	65	id est
12.	ltd	Adj.	ltd, ltd.	64	limeted
13.	pp	NoC.	pp., pp	59	per person
14.	Inc	Adj.	Inc, Inc.	59	incorporated
15.	Corp	NoC.	Corp, Corp.	53	cooperation
16.	Jan	NoP.	Jan. Jan	45	january
17.	ref	NoC.	ref, ref., refs	44	reference
18.	Dec	NoP.	Dec. Dec	44	december
19.	et al	Adv.	et al, et al.	42	et alii
20.	Oct	NoP.	Oct., Oct	37	october
21.	pc	NoC.	pc, P.C., pcs, pc., p.c	37	personal computer
22.	Nov	NoP.	Nov., Nov	34	november
23.	Sept	NoP.	Sept., Sept	32	september
24.	qv	Uncl.	q.v., q.v	30	quod vide
25.	Feb	NoP.	Feb., Feb	30	february
26.	Aug	NoP.	Aug., Aug	29	august
27.	cm	NoC	cm, cm.	22	centimeter
28.	no	Adv.	no.	17	number

Table A.3: BNC: lemmatised abbreviation list (frq. per million words)

Acronym List of the DBLP

No.	Acronym	Substitution	Frequency
1.	XML	290	Extensible Markup Language
2.	ATM	287	Asynchronous Transfer Mode*
			Automated Teller Machine
			Adobe Type Manager
			Anti Tank Missile
3.	VLSI	218	Very Large Scale Integration
4.	QoS	170	Quality of Service
5.	DNA	163	Deoxyribo Nucleic Acid*
			Digital Network Architecture
			Defense Nuclear Agency
6.	UML	138	Unified Modelling Language*
			Unified Method Language
7.	IP	138	Internet Protocol*
			Information Provider
			Intermediate Point
			Industrial Products
8.	CMOS	119	Complementary Metal Oxide Semiconductor
9.	FPGA	113	Field Programmable Gate-Array
10.	IT	105	Information Technology*
			Industrial Technology
			Information Theory
			Intelligent Terminal
			Italy

Table A.4: DBLP: acronym list (frq. per million words)

No.	Acr.	Frq.	No.	Acr.	Frq.	No.	Acr.	Frq.
11.	AI	98	22.	CDMA	57	33.	TREC	45
12.	NP	96	23.	RNA	56	34.	ISDN	44
13.	I/O	92	24.	SAT	54	35.	SIMD	44
14.	MPEG	92	25.	DBMS	53	36.	LAN	43
15.	CORBA	91	26.	MPI	53	37.	MRI	43
16.	TCP	90	27.	BIST	52	38.	DSP	42
17.	WWW	80	28.	GIS	52	39.	P2P	41
18.	CAD	80	29.	WDM	52	40.	CSP	38
19.	IBM	65	30.	MR	50	41.	LOTOS	36
20.	PC	59	31.	UNIX	48	42.	RSA	36
21.	IEEE	58	32.	SQL	48	43.	APL	35

continuation on the next page...

No.	Acr.	Frq.	No.	Acr.	Frq.	No.	Acr.	Frq.
44.	IS	35	79.	FFT	22	114.	PCA	17
45.	VHDL	34	80.	HMM	22	115.	ATPG	16
46.	SDL	34	81.	DCT	22	116.	B2B	16
47.	CT	34	82.	QSAR	21	117.	AC	16
48.	MAC	33	83.	DFT	21	118.	RTL	16
49.	CASE	32	84.	RBF	21	119.	BSP	16
50.	MIMD	32	85.	DSM	20	120.	FIR	16
51.	PVM	31	86.	PRAM	20	121.	TSP	16
52.	ML	30	87.	HOL	20	122.	DASD	16
53.	GA	30	88.	ABR	20	123.	MPLS	15
54.	ACM	30	89.	OpenMP	20	124.	PAC	15
55.	ISBN	30	90.	TCP/IP	19	125.	VDM	15
56.	IC	30	91.	HTML	19	126.	CLP	15
57.	EM	30	92.	ERP	19	127.	CM	15
58.	SoC	28	93.	ALGOL	18	128.	RF	15
59.	COTS	28	94.	CD	18	129.	ER	15
60.	OO	27	95.	RDF	18	130.	PCS	15
61.	OSI	26	96.	SVM	18	131.	DES	15
62.	OLAP	26	97.	PROLOG	18	132.	RT	15
63.	IV	26	98.	VRML	18	133.	MIS	15
64.	HCI	26	99.	LR	18	134.	MEMS	15
65.	CSCW	26	100.	MAX	18	135.	CCS	14
66.	IR	25	101.	ICA	18	136.	NLP	14
67.	RISC	25	102.	DB	18	137.	SMP	14
68.	NC	25	103.	JPEG	18	138.	LSI	14
69.	NMR	24	104.	US	17	139.	U.S	14
70.	CBR	24	105.	LISP	17	140.	MAP	14
71.	CPU	24	106.	DC	17	141.	OS	14
72.	FORTRAN	24	107.	ISO	17	142.	HPF	14
73.	ICT	24	108.	UK	17	143.	GUI	14
74.	TV	23	109.	OCR	17	144.	MT	14
75.	VBR	23	110.	CIM	17	145.	FDDI	14
76.	VLIW	23	111.	MOS	17	146.	SAR	14
77.	ILP	22	112.	ASIC	17	147.	CS	13
78.	VR	22	113.	OR	17	148.	ITS	13

Table A.5: DBLP: continuation acronym list (frq. per million words)

Acronym List of the BNC

No.	Acr.	Variants	Frq.	Substitution
1.	UK	U.K., UK	177	United Kingdom
2.	US	U.S., US	162	United States
3.	EC	EC	67	Conseil d'Europe
4.	TV	T.V., TV	66	Television
5.	MP	M.P., MP, MPs	58	Member of Parliament
6.	USA	U.S.A., USA	52	United States of America
7.	UN	UN	44	United Nations
8.	BBC	B.B.C., BBC	43	British Broadcasting Corporation
9.	IBM	IBM	43	International Business Machines
10	DNA	DAN, DNAs	34	Deoxyribonucleic Acid
11	NHS	NHS	25	National Health Service
12	LA	L.A., LA, La	24	Los Angeles
13	RAF	R.A.F., RAF	20	Royal Air Force
14	AIDS	AIDS	19	Acquired Immune Deficiency Syndrome
15	IRA	IRA	18	Irish Republican Army
16	DC	D.C., DC	17	District of Columbia
17	USSR	U.S.S.R, USSR	17	Union of Soviet Socialist Republics
18	HIV	HIV	16	Human Immunodeficiency Virus
19	FA	FA	15	First Aid
20	NATO	NATO	15	North Atlantic Treaty Organisation
21	ICI	ICI	14	Incoming Call Identification
22	IT	IT	14	Information Technology
23	NT	NT	14	New Technology
24	BR	B.R., BR, Br.	13	Best Regards
25	CD	CD, CDs	13	Compact Disc
26	EEC	E.E.C., EEC	13	European Economic Commission
27	BC	B.C., BC	11	Before Christ
28	HP	HP	11	Hewlett-Packard
29	IMF	IMF	11	International Monetary Fund

Table A.6: BNC: lemmatised acronym list (frq. per million words)

Frequency List of the DBLP

No.	Word	Frq.
1.	of	25567
2.	for	20348
3.	and	16797
4.	the	16013
5.	in	13391
6.	on	7773
7.	to	5892
8.	an	5360
9.	based	5298
10.	with	5052
11.	using	4284
12.	systems	4190
13.	system	3518
14.	data	3006
15.	analysis	2618
16.	design	2361
17.	model	2339
18.	networks	2223
19.	time	2126
20.	algorithm	2112
21.	approach	2050
22.	parallel	2032
23.	by	2016
24.	from	1915
25.	software	1757
26.	learning	1667
27.	algorithms	1590
28.	distributed	1581
29.	information	1547
30.	network	1484
31.	logic	1454
32.	control	1426
33.	performance	1366
34.	web	1348
35.	object	1347
36.	management	1284
37.	efficient	1270
38.	new	1245

No.	Word	Frq.
39.	knowledge	1203
40.	models	1180
41.	computer	1171
42.	database	1150
43.	image	1142
44.	language	1128
45.	method	1126
46.	dynamic	1115
47.	real	1082
48.	multi	1075
49.	problem	1051
50.	modeling	1034
51.	simulation	1025
52.	oriented	1008
53.	applicatio	999
54.	evaluation	986
55.	computing	974
56.	high	924
57.	programmin	922
58.	study	898
59.	problems	888
60.	neural	865
61.	applicatio	862
62.	linear	858
63.	mobile	848
64.	architectu	847
65.	framework	841
66.	programs	832
67.	support	822
68.	adaptive	815
69.	case	801
70.	graphs	796
71.	fuzzy	788
72.	processing	788
73.	as	769
74.	multiple	765
75.	optimal	762
76.	environmen	760

No.	Word	Frq.
77.	scheduling	757
78.	theory	757
79.	memory	748
80.	complexity	748
81.	process	748
82.	generation	720
83.	retrieval	720
84.	two	718
85.	databases	716
86.	detection	707
87.	search	696
88.	virtual	683
89.	large	680
90.	digital	677
91.	automatic	673
92.	methods	671
93.	der	671
94.	structure	668
95.	developmen	667
96.	level	666
97.	agent	637
98.	techniques	637
99.	user	632
100.	fast	621
101.	languages	608
102.	graph	608
103.	non	602
104.	space	602
105.	video	592
106.	test	588
107.	program	588
108.	von	587
109.	towards	582
110.	functions	580
111.	fault	577
112.	reasoning	576
113.	multimedia	574
114.	service	573

continuation on the next page...

No.	Word	Frq.	No.	Word	Frq.	No.	Word	Frq.
115.	informatio	573	152.	temporal	483	189.	low	388
116.	recognition	570	153.	structures	477	190.	integrated	384
117.	its	568	154.	protocol	476	191.	research	379
118.	power	565	155.	visual	464	192.	integratio	378
119.	some	564	156.	set	464	193.	distribute	376
120.	images	561	157.	extended	461	194.	pattern	374
121.	decision	540	158.	is	461	195.	text	370
122.	testing	538	159.	between	454	196.	wireless	370
123.	tool	538	160.	scheme	453	197.	properties	369
124.	trees	537	161.	modelling	448	198.	processes	368
125.	through	536	162.	interface	446	199.	circuits	367
126.	access	536	163.	free	443	200.	hybrid	366
127.	finite	535	164.	state	439	201.	relational	364
128.	genetic	534	165.	matching	435	202.	type	362
129.	self	533	166.	comparison	435	203.	electronic	362
130.	order	531	167.	human	433	204.	simple	362
131.	internet	528	168.	agents	432	205.	into	360
132.	routing	527	169.	security	431	206.	robot	360
133.	und	520	170.	at	427	207.	ming	360
134.	applicatio	518	171.	motion	423	208.	technique	357
135.	use	517	172.	interactiv	420	209.	sequences	355
136.	sets	516	173.	functional	419	210.	solving	354
137.	technology	516	174.	flow	416	211.	general	353
138.	tree	515	175.	automata	410	212.	clustering	353
139.	optimization	513	176.	semantic	410	213.	computatio	352
140.	objects	505	177.	local	409	214.	error	351
141.	query	498	178.	synthesis	409	215.	concurrent	351
142.	planning	498	179.	selection	408	216.	code	347
143.	engineering	498	180.	mining	408	217.	class	342
144.	formal	494	181.	random	408	218.	abstract	341
145.	services	493	182.	quality	404	219.	function	340
146.	estimation	490	183.	environmen	402	220.	machines	339
147.	constraint	490	184.	specificat	400	221.	complex	338
148.	semantics	490	185.	verificati	392	222.	feature	338
149.	machine	489	186.	context	390	223.	issues	337
150.	over	488	187.	codes	389	224.	rules	337
151.	line	487	188.	via	389	225.	domain	334

Table A.7: DBLP: general frequency list (frq. per million words)

Frequency List of the BNC

No.	Word	PoS.	Frq.
1.	the	Det.	61847
2.	of	Prep	29391
3.	and	Conj.	26817
4.	a	Det.	21626
5.	in	Prep.	18214
6.	to	Inf.	16284
7.	it	Pron.	10875
8.	is	Verb	9982
9.	to	Prep.	9343
10.	was	Verb	9236
11.	I	Pron.	8875
12.	for	Prep.	8412
13.	that	Conj.	7308
14.	you	Pron.	6954
15.	he	Pron.	6810
16.	be*	Verb	6644
17.	with	Prep.	6575
18.	on	Prep.	6475
19.	by	Prep.	5096
20.	at	Prep.	4790
21.	have*	Verb	4735
22.	are	Verb	4707
23.	not	Neg.	4626
24.	this	DetP.	4623
25.	's	Gen.	4599
26.	but	Conj.	4577
27.	had	Verb	4452
28.	they	Pron.	4332
29.	his	Det.	4285
30.	from	Prep.	4134
31.	she	Pron.	3801
32.	that	DetP.	3792
33.	which	DetP.	3719
34.	or	Conj.	3707
35.	we	Pron.	3578
36.	's	Verb	3490
37.	an	Det.	3430
38.	n't	Neg.	3328

No.	Word	PoS.	Frq.
39.	were	Verb	3227
40.	as	Conj.	3006
41.	do	Verb	2802
42.	been	Verb	2686
43.	their	Det.	2608
44.	has	Verb.	2593
45.	would	VMod.	2551
46.	there	Ex.	2532
47.	what	DetP.	2493
48.	will	VMod.	2470
49.	all	DetP.	2436
50.	if	Conj.	2369
51.	can	VMod.	2354
52.	her*	Det.	2183
53.	said	Verb	2087
54.	who	Pron.	2055
55.	one	Num.	1962
56.	so	Adv.	1893
57.	up	Adv.	1795
58.	as	Prep.	1774
59.	them	Pron.	1733
60.	some	DetP.	1712
61.	when	Conj.	1712
62.	could	VMod.	1683
63.	him	Pron.	1649
64.	into	Prep.	1634
65.	its	Det.	1632
66.	then	Adv.	1595
67.	two	Num.	1561
68.	out	Adv.	1542
69.	time	NoC.	1542
70.	my	Det.	1525
71.	about	Prep.	1524
72.	did	Verb	1434
73.	your	Det.	1383
74.	now	Adv.	1382
75.	me	Pron.	1364
76.	no	Det.	1343

continuation on the next page...

No.	Word	PoS.	Frq.	No.	Word	PoS.	Frq.
77.	other	Adj.	1336	110.	those	DetP.	888
78.	only	Adv.	1298	111.	go	Verb	881
79.	just	Adv.	1277	112.	being	Verb	862
80.	more	Adv.	1275	113.	because	Conj.	852
81.	these	DetP.	1254	114.	down	Adv.	845
82.	also	Adv.	1248	115.	're	Verb	835
83.	people	NoC.	1241	116.	yeah	Int.	834
84.	know	Verb	1233	117.	three	Num.	797
85.	any	DetP.	1220	118.	good	Adj.	795
86.	first	Ord.	1193	119.	back	Adv.	793
87.	see	Verb	1186	120.	make	Verb	791
88.	very	Adv.	1165	121.	such	DetP.	763
89.	new	Adj.	1145	122.	on	Adv.	756
90.	may	VMod.	1135	123.	there	Adv.	746
91.	well	Adv.	1119	124.	through	Prep.	743
92.	should	VMod.	1112	125.	year	NoC.	737
93.	her*	Pron.	1085	126.	over	Prep.	735
94.	like	Prep.	1064	127.	'll	VMod.	726
95.	than	Conj.	1033	128.	must	VMod.	723
96.	how	Adv.	1016	129.	still	Adv.	718
97.	get	Verb	995	130.	even	Adv.	716
98.	way	NoC.	958	131.	take	Verb	715
99.	one	Pron.	953	132.	too	Adv	701
100.	our	Det.	950	133.	more	DetP	699
101.	made	Verb	943	134.	here	Adv	699
102.	got	Verb	932	135.	own	DetP	695
103.	after	Prep.	927	136.	come	Verb	695
104.	think	Verb	916	137.	last	Ord	691
105.	between	Prep.	903	138.	does	Verb	687
106.	many	DetP.	902	139.	oh	Int	684
107.	years	NoC.	902	140.	say	Verb	679
108.	er	Uncl.	896	141.	no	Int	662
109.	've	Verb	891	141.	going	Verb	658

Table A.8: BNC: general frequency list (frq. per million words)

Appendix B

Java Implementation

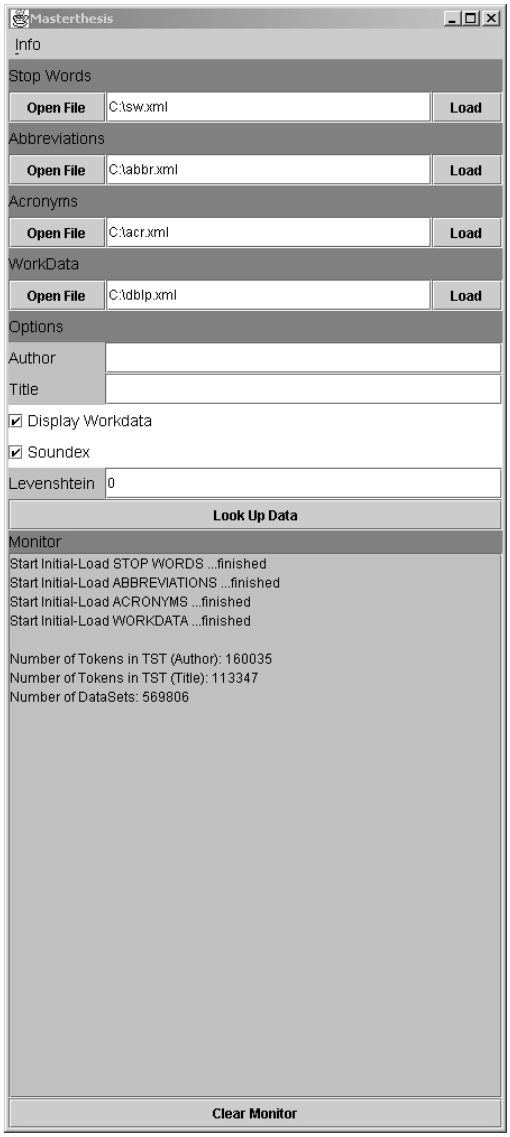

Figure B.1: Java implementation: start

Figure B.2: Java implementation: $edist_{Lev}$ request

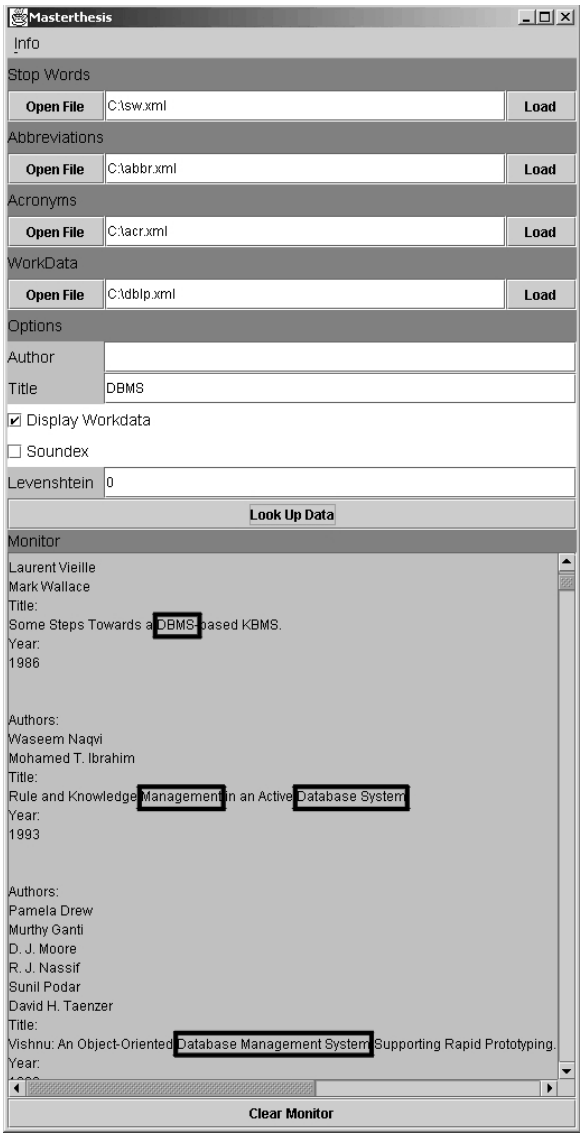

Figure B.3: Java implementation: acronym request (ex. 1)

Figure B.4: Java implementation: acronym request (ex. 2)

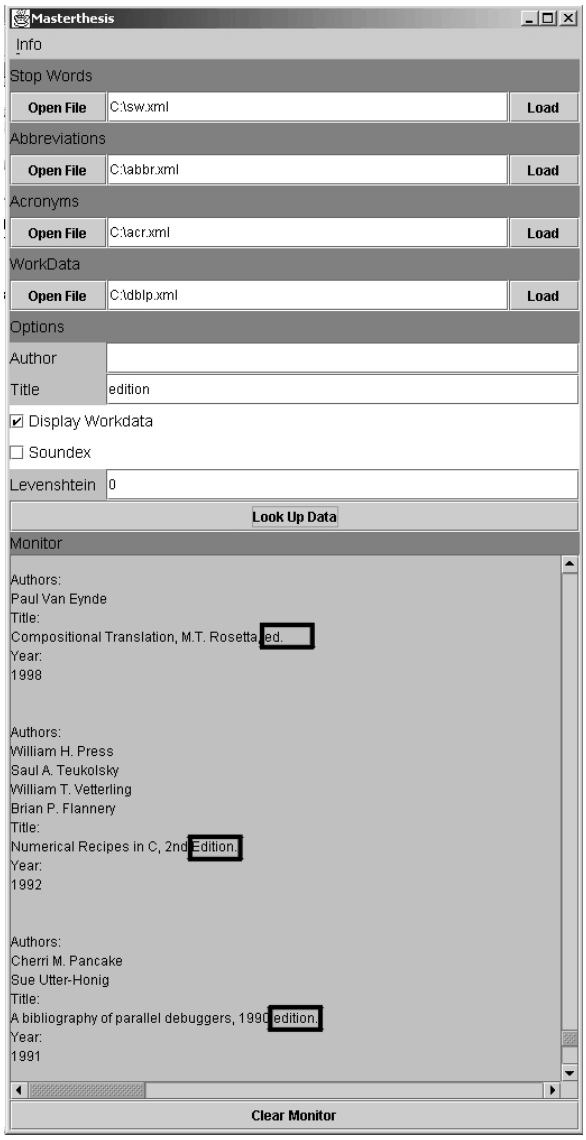

Figure B.5: Java implementation: abbreviation request

Figure B.6: Java implementation: SOUNDEX request

Appendix C

Evaluation Hybrid TST

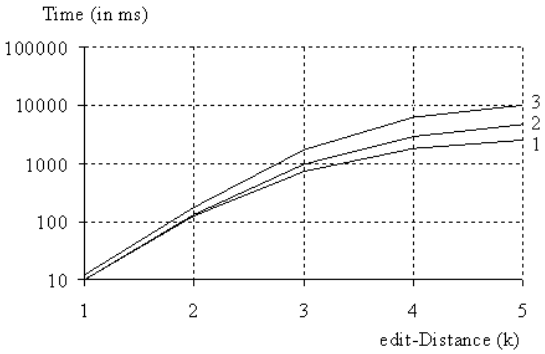

Figure C.1: Hybrid TST: m=2,n={1. 1.000.000, 2. 2.000.000, 3. 4.000.000}

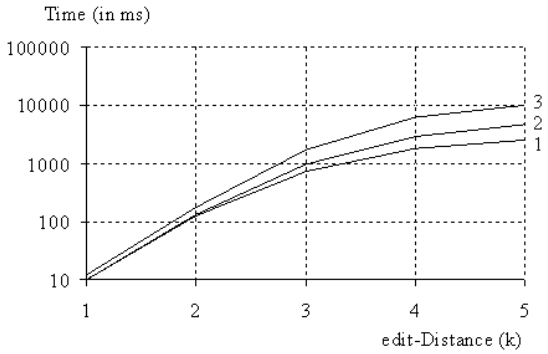

Figure C.2: Hybrid TST: m=4,n={1. 1.000.000, 2. 2.000.000, 3. 4.000.000}

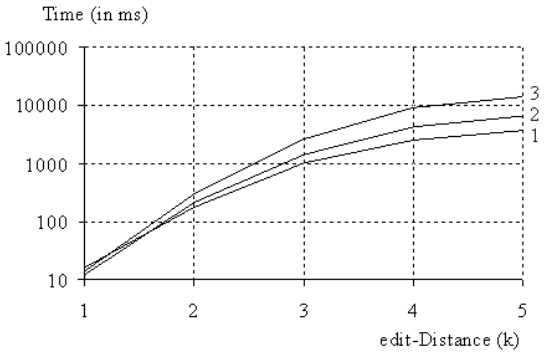

Figure C.3: Hybrid TST: m=6,n={1. 1.000.000, 2. 2.000.000, 3. 4.000.000}

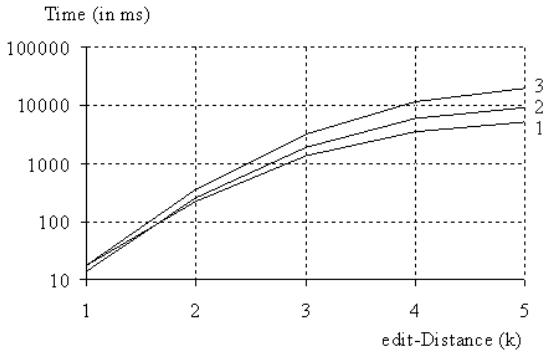

Figure C.4: Hybrid TST: m=10,n={1. 1.000.000, 2. 2.000.000, 3. 4.000.000}

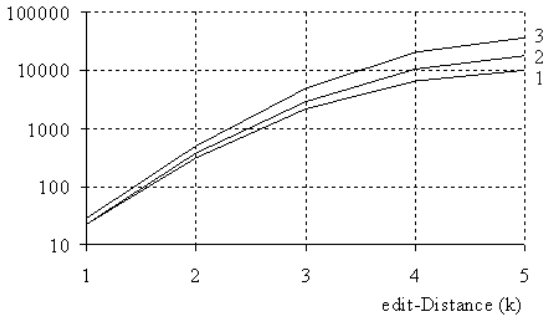

Figure C.5: Hybrid TST: m=14,n={1. 1.000.000, 2. 2.000.000, 3. 4.000.000}

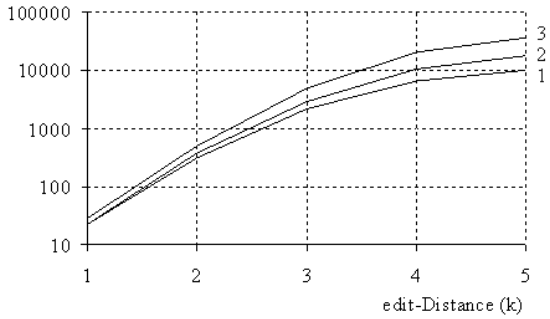

Figure C.6: Hybrid TST: m=20,n={1. 1.000.000, 2. 2.000.000, 3. 4.000.000}

Appendix D

PHONIX Rules

No.	Sub.	Start	Middle	End
1.	g	dg	dg	dg
2.	ko	co	co	co
3.	ka	ca	ca	ca
4.	ku	cu	cu	cu
5.	si	cy	cy	cy
6.	si	ci	ci	ci
7.	se	ce	ce	ce
8.	kl	cl if cl$v$		
9.	k	ck	ck	ck
10.	k			gc
11.	k			jc
12.	kr	chr if chr$v$		
13.	kr	cr if cr$v$		
14.	r	wr		
15.	nk	nc	nc	nc
16.	kt	ct	ct	ct
17.	f	ph	ph	ph
18.	ar	aa	aa	aa
19.	sh	sch	sch	sch
20.	tl	btl	btl	btl
21.	t	ght	ght	ght
22.	arf	augh	augh	augh
23.	ld		lj if $v$lj$v$	
24.	low	lough	lough	lough
25.	kw	q		
26.	n	kn		
27.	n			gn

continuation on the next page...

No.	Sub.	Start	Middle	End
28.	n	ghn	ghn	ghn
29.	n			gne
30.	ne	ghne	ghne	ghne
31.	ns			gnes
32.	n	gn		
33.	n		gn if gn$c$	gn if gn$c$
34.	s	ps		
35.	t	pt		
36.	c	cz		
37.	z		wz if $v$wz	
38.	ch		cz	
39.	lsh	lz	lz	lz
40.	rsh	rz	rz	rz
41.	s		z if zv	
42.	ts	zz	zz	zz
43.	ts		z if cz	
44.	rew	hroug	hroug	hroug
45.	of	ough	ough	ough
46.	kw		q if vqv	
47.	y		j if vjv	
48.	y	yj if yj$v$		
49.	g	gh		
50.	e			gh if $v$gh
51.	s	cy		
52.	nks	nx	nx	nx
53.	f	pf		
54.	t			dt
55.	til			tl
56.	dil			dl
57.	ith	yth	yth	yth
58.	ch	tj if tj$v$		
59.	ch	tsj if tsj$v$		
60.	t	ts if ts$v$		
61.	ch	tch	tch	tch
62.	vskie		wsk if $v$wsk	wsk if $v$wsk
63.	n	mn if $v$mn		
64.	n	pn if $v$pn		
65.	sl		stl if $v$stl	stl if $v$stl
66.	ent			tnt
67.	oh			eaux

continuation on the next page...

No.	Sub.	Start	Middle	End
68.	ecs	exci	exci	exci
69.	ecs	x	x	x
70.	nd			ned
71.	dr	jr	jr	jr
72.	ea			ee
73.	s	zs	zs	zs
74.	ah		r if vrc	r if vrc
75.	ah		hr if $v$hr$c$	hr if $v$hr$c$
76.	ah			hr if $v$hr$c$
77.	ar			re
78.	ah			r if vr
79.	le	lle	lle	lle
80.	ile			le if $c$le
81.	iles			les if $c$les
82.	null			e
83.	s			es
84.	as			ss if $v$ss
85.	m			mb if $v$mb
86.	mps	mpts	mpts	mpts
87.	ms	mps	mps	mps
88.	mt	mpt	mpt	mpt

Table D.1: PHONIX substitutions rules

Appendix E

Bibliography

Bibliography

[ADKF75] Arlazarov V.L., Dinic E.A., Kronrod M.A., Faradzev I.A.: On economic construction of the transitive closure of a directed graph, *English Translation in Soviet Math. Dokl., Vol. 11, pp. 1209-1210, 1975*

[Bae89] Baeza-Yates R.A.: Efficient text searching, Ph. D. Thesis, *Dept. of Computer Science, University of Waterloo, 1989*

[BaN96] Baeza-Yates R.A., Navarro G.: A fast heuristic for approximate string matching, *Proc. of the 3.rd South American Workshop on String Processing, (Carleton Univ. Press), pp. 47-63, 1996*

[BaN99] Baeza-Yates R.A., Navarro G.: A faster algorithm for approximate string matching, *Algorithmica, Vol. 23, No. 2 pp. 127-158, 1999*

[Bau03] Bauer L.: Introducing Linguistic Morphology, *Edinburgh University Press, 2003*

[BaP96] Baeza-Yates R.A., Perleberg C.H.: Fast and practical approximate string matching, *Information Processing Letters, Vol. 59, No. 1, pp. 21-27, 1996*

[BeS97] Bentley J., Sedgewick R.: Fast algorithms for sorting and searching strings, *In Eighth Annual ACM-SIAM Symposium on Discrete Algorithms - SIAM Press, 1997*

[BiR95] Binstock A., Rex J.: Practical algorithms for programmers, *Addison-Wesley, 1995*

[Bra59] Brandais R.: File searching using variable length keys, *Proceedings of Western Joint Computer Conference, Vol. 15, pp. 295-298, 1959*

[ChL92] Chang W.I., Lampe, J.: Theoretical and empirical comparisons of approximate string matching algorithms, *Proc. of the 3rd. Annual Symposium on Combinatorical Pattern Matching, No. 664 (Springer-Verlag Berlin) pp. 175-184, 1992*

[Cle97] Clement J.: The analysis of hybrid trie structures, *Algorithms project, INRIA Rocquencourt, 1997*

[ElN00] Elmasri R., Navathe S. B.: Fundamentals of database systems (3rd Edition), *Addison-Wesley, 2000*

[EnW96] McEnery T., Wilson A.: Corpus linguistics, *Edinburgh University Press, 1996*

[Far97] Farah M.: Optimal suffix tree Construction with large alphabets, *FOCS, 1997*

[FeG99] Ferragina P. Grossi R.: The string B-tree: A new data structure for string search in external memory and its application, *ACM, Vol. 46, No, 2, pp. 236-280, 1999*

[Fre60] Fredkin e.: Trie memory, *Communications of ACM, Vol. 3, pp. 490-499, 1960*

[Gad88] Gadd T. N.: Fisching fore werds: Phonetic retrieval of written text in information systems, *Program-Automated Library and Information System, 22, pp. 222-237, 1988*

[Gad90] Gadd T.N.: PHONIX: The algorithm, *Program-Automated Library and Information Systems, 24, pp. 363-366, 1990*

[GaP90] Galil Z., Park K.: An improved algorithm for approximate string matching, *SIAM Journal on Computing, Vol. 19, No. 6, pp. 989-999, 1990*

[GiI99] Gil J., Itai A.: How to pack trees, *Journal of Algorithms, Vol. 32, No. 2, pp. 108-132, 1999*

[GoT98] Goodrich M. T., Tamassia R.: Data structures and algorithms in Java, *John Wiley and Sons, 1998*

[Gus97] Gusfield D.: Algorithms on strings, trees, and sequences, *Cambridge University Press, 1997*

[Ham02] Hammond M.: Programming For Linguists, *Blackwell Publishers, 2002*

[Ham80] Hamming R. W.: Coding and information theory, *Prentice-Hall, 1980*

[Kur96] Kurtz S.:Approximate string searching under weighted edit distance, *Proc. of the 3rd. South American Workshop on String Processing Carleton University Press, 1996, pp. 156-170*

[LaV88] Landau G., Vishkin U.:Fast string matching with k differences, *J. Comput. Syst. Sci, Vol. 37, No. 1, pp. 63-78, 1988*

[LaV89] Landau G., Vishkin U.: Fast parallel and seriell approximate string matching, *J. of Algorithms, Vol. 10, No. 2, pp. 157-169, 1989*

[Lev65] Levenshtein V.: Binary codes capable of correcting spurios insertations and deletions of ones, *Probl. Inf. Transmission 1, pp. 8-17, 1965*

[LNP04] Linke A., Nussbaumer M., Portmann P., Willi U.: (German) Studienbuch Linguistik (Reihe Germanistische Linguistik 121), 5. erweiterte Aufl., *Tübingen, Niemeyer, 2004*

[LuP01] Lujan-Mora S., Palomar M.: Reducing Inconsistency in Integrating Data From Different Sources, *IDEAS '01 - Proceedings of the International Database Engineering & Applications Symposium, pp. 209-218, 2001*

[McC76] McCreight E.: A space-economical suffix tree construction, *Algorithm, J. of ACM, Vol. 23, No. 2, 1976*

[MaM93] Manber U., Myers G.: Suffix Arrays: A new method for online string searches, *SIAM Journal on Computing, Vol. 22, No. 5, 935-948, 1993*

[MiM02] Michailidis P.D., Margaritis K.G.: Online approximate string searching algorithms: Survey and experimental results, *Intern. J. Computer. Math., 2002, Vol. 79, No.8, pp867-888, 2002*

[Mit05] Mitkov R.: The Oxford Handbook of Computational Linguistics, *Oxford University Press, 2005*

[Muh03] Muhs H.: (German) Unscharfe Suche von Zeichenketten in großen Datenmengen, *thesis, University of Magdeburg, 2001*

[Myr98] Myers E.: A fast bit-vector algorithm for approximate string matching based on dynamic progamming, *In Proc. of the 9th Annual Symposium on Combinatorial Pattern Matching, No. 1448 (Springer Verlag Berlin), pp. 1-13, 1998*

[Myr99] Myers E.: A fast bit-vector algorithm for approximate string matching based on dynamic progamming, *ACM, Vol. 46, pp. 395-415, No. 3, 1999*

[Mor68] Morrison D.: PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric, *ACM, Vol. 15, No. 4, pp. 514-534, 1968*

[Nav97A] Navarro G.:Multiple approximate string matching by counting, *Proc. of the 4rd. South American Workshop on String Processing Carleton University Press, 1997, pp. 125-139*

[Nav97B] Navarro G.:A partial deterministic automaton for approximate string matching, *Proc. of the 4rd. South American Workshop on String Processing Carleton University Press, 1997, pp. 112-124*

[Nav01] Navarro G.: A guided tour to approximate string matching, *ACM, Vol. 33, No. 1, pp. 31-88, 2001*

[Phi90] Philips L.: Hanging on the metaphone, *Computer Language, Vol. 7, No. 12, 1990*

[Por80] Porter M.F.: An algorithm for suffix stripping, *Program 14(3), pp. 130-137, 1980*

[Sch02] Scholz, M.: (German) Experimentelle Untersuchung von String-Trees bezüglich ihrer Anwendbarkeit in Genom Datenbanken und im Information Retrivial, *FU Berlin, 2002*

[Sel80] Sellers P.: The theory and computation of evolutionary distances: pattern recognition, *J. Algor., Vol. 1, pp. 359-373, 1980*

[ShM95] Shang H., Merrett T.H.: Tries for approximate string matching, *IEEE Transaction on Knowledge and Data Engeneering, Vol. 8, No. 4, pp. 540-547, 1995*

[Sed03] Sedgewick R.: Algorithms in Java - Parts 1-4 (3rd Edition), *Addison-Wesley, 2003*

[Sta02] Stamme K.: (German) Einsatz von Bitparallelen Algorithmen und Filtern zur approimierten Zeichenkettensuche, *Uni. Hannover - Inst. für Informatik - Diplomarbeit, 2002*

[Sun04] Sung W.-S.: Combinatorial methods in bioinformatics, *2004/2005 Semester 1, 2004*

[TaU93] Tarhio J., Ukkonen E.: Approximate Bayer-Moore string matching, *SIAM Journal on Computing, Vol. 22, No. 2, pp. 243-260, 1993*

[Ukk85] Ukkonen E.: Finding approximate patterns in strings, *J. Algor., Vol. 4, No. 1-3, pp. 132-137, 1985*

[Ukk95] Ukkonen E.: Constructing suffix Trees online in linear time, *Algorithmica, Vol. 14, No. 3, pp. 249-260, 1995*

[Wei73] Weiner P.: Linear pattern matching algorithms, *Switching and Automata Theory, 1-11, 1973*

[WuM92] Wu S., Manber U.: Fast text searching allowing errors, *Communications of ACM, Vol. 35, No. 10, pp. 83-91, 1992*

[WMM96] Wu S., Manber U., Myers E.: A subquadratic algorithm for approximate limited expression matching, *Algorithmica, Vol. 15, No. 1, pp. 50-67, 1996*

[Zie00] Zierl M.: (German) Entwicklung und Implentierung eines Datenbanksystems zur Speicherung und Verarbeitung von Textkorpora (2. Auflage), *Friedrich-Alexander Universität Erlangen-Nürnberg Inst. für Computerlinguistik, 2000*

[ZoD96] Zobel J., Dart P.: Phonetic string matching - lessons from information retrieval, *ACM, pp. 166-172, 1996*

Statuary Declaration

I hereby explain that the presented masterthesis was made by myself and the given bibliography has been exclusively used.

Wolfenbüttel, September 2005

André Reckhemke