SPELL CHECKERS AND CORRECTORS: A UNIFIED TREATMENT

By

Hsuan Lorraine Liang

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE MASTER OF SCIENCE (COMPUTER SCIENCE) IN THE FACULTY OF ENGINEERING, BUILT ENVIRONMENT AND INFORMATION TECHNOLOGY, UNIVERSITY OF PRETORIA, PRETORIA, SOUTH AFRICA

AT

NOVEMBER 2008

To my late grandfather, Shu-Hsin.

To my parents, Chin-Nan and Hsieh-Chen.

Abstract

The aim of this dissertation is to provide a unified treatment of various spell checkers and correctors. Firstly, the spell checking and correcting problems are formally described in mathematics in order to provide a better understanding of these tasks. An approach that is similar to the way in which denotational semantics used to describe programming languages is adopted. Secondly, the various attributes of existing spell checking and correcting techniques are discussed. Extensive studies on selected spell checking/correcting algorithms and packages are then performed. Lastly, an empirical investigation of various spell checking/correcting packages is presented. It provides a comparison and suggests a classification of these packages in terms of their functionalities, implementation strategies, and performance. The investigation was conducted on packages for spell checking and correcting in English as well as in Northern Sotho and Chinese. The classification provides a unified presentation of the strengths and weaknesses of the techniques studied in the research. The findings provide a better understanding of these techniques in order to assist in improving some existing spell checking/correcting applications and future spell checking/correcting package designs and implementations.

Keywords: classification, dictionary lookup, edit distance, formal concept analysis, FSA, n-gram, performance, spell checking, spell correcting.

Degree: Magister Scientia
Supervisors: Prof B. W. Watson and Prof D. G. Kourie
Department of Computer Science

Acknowledgements

First and foremost, I would like to thank my supervisors, Bruce Watson and Derrick Kourie, for directing my research, reviewing my ideas, and providing feedback on my work. I would also like to take this opportunity to express my gratitude for their continuous patience, support, and encouragement.

I would like to thank my parents, Chin-Nan and Hsieh-Chen Liang, my brother, Albert Liang, and my grandmother, Yu-Fang Fan, for their unconditional love and support.

A special thanks to Ernest Ketcha Ngassam for many technical discussions and inspiration during this research. He devoted a great deal of his time to helping me kick start this work. Sergei Obiedkov was helpful in pointing me to related research on formal concept analysis. TshwaneDJe HLT and its CEO, David Joffe, were helpful in providing the Northern Sotho corpora as well as providing assistance for the language.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we identify the problem statement of the domain of spell checking and correcting. We also identify the intended audience of this research. An introduction to the contents and the structure of this dissertation is presented here. This study is restricted to context-independent spelling error detection and correction.

1.1 Problem Statement

Spell checkers and correctors are either stand-alone applications capable of processing a string of words or a text, or as an embedded tool which is part of a larger application such as a word processor. Various search and replace algorithms are adopted to fit in the domain of spell checking and correcting. Spelling error detection and correction are closely related to exact and approximate pattern matching respectively.

Work on spelling error detection and correction in text started in the 1960s. A number of commercial and non-commercial spell checkers and correctors, such as the ones embedded in Microsoft Word, Unix® SPELL, GNU's ISPELL, ASPELL and other variants, and AGREP have been extensively studied. However, the techniques of spell correcting in particular are still limited in their scope, speed, and accuracy.

Spell checking identifies words that are valid in some language, as well as the misspelled words in the language. Spell correcting suggests one or more alternative words as the correct spelling when a misspelled word is identified. Spell checking involves non-word error detection and spelling correction involves isolated-word error correction. Isolated-word error correction refers to spell correcting without taking into account any textual or linguistic information in which the misspelling occurs whereas context-dependent word correction would correct errors involving textual or linguistic context.

Between the early 1970s and early 1980s, research focused on the techniques for non-word error detection. A non-word can be defined as a continuous string of letters of an alphabet that does not appear in a given word list or dictionary or that is an invalid word form. Non-word spelling error detection can generally be divided into two main techniques, namely dictionary lookup techniques and n-gram analysis. In the past, it was often the case that text recognition systems, such as OCR technology, inclined to rely on n-gram analysis for error detection, whereas spell checkers mostly rely on dictionary lookup techniques. Each technique can be used individually as a basis, in combination or together with other methods, such as probabilistic techniques. Isolated-word error correction techniques are often designed based on a study of spelling error patterns. Distinctions are generally made between typographic, cognitive, and phonetic errors. Techniques such as minimum edit distance, n-gram-based techniques, neural networks, probabilistic, rule-based techniques, and similarity key techniques are discussed.

In [Kuk92], Kukich gave a survey that provided an overview of the spell checking and correcting techniques and a discussion on some existing algorithms. Chapters 5 and 6 in the book of Jurafsky and Martin [JM00] were based on Kukich's work. Hodge and Austin [HA03] compared standard spell checking algorithms to a spell checking system based on the AURA modular neural network. Their research is heavily focused on this hybrid spell checking methodology. Although descriptions on some investigated spell checking and correcting algorithms can be found in the mentioned references, little has been done to compare and represent the existing algorithms in a unified manner. Neither has any formal classification been done previously. This dissertation aims to fill these gaps and provide a unified treatment of various spell checking and correcting algorithms.

Furthermore, we explore the spell checking and correcting techniques in different language systems other than Euro-based languages, such as Northern Sotho and Chinese.

The variability of the spell checking and correcting techniques and algorithms suggests the need for a unified treatment to represent them. We do not necessarily consider all of the known spell checking and correcting algorithms. We restrict ourselves to context-independent spelling error detection and correction as context-dependent spelling detection and correction often involves syntactic analysis, such as part-of-speech tagging [RS95]. This is a vast research area in its own right, and is beyond the scope of the present research. For the purpose of this dissertation, it is assumed that all characters are of fixed width.

In this dissertation, we described the spell checking and correcting tasks mathematically. These semantic specifications are intended to provide a concise understanding of spelling error detection and correction. We will investigate the most important spell checking and correcting techniques that have been used to date. We then study various spell checking and correcting algorithms in details. Finally, we propose a classification of the algorithms and techniques found in various existing spell checkers and correctors. Experiments were conducted to establish the performance of various spell checking and correcting packages. We then propose a concept lattice to establish the correlation between these packages and their attributes.

In the subsequent section, we present an overview of the structure of this dissertation. A discussion of the intended audience is then given in Section 1.3.

1.2 Structure of this Dissertation

This dissertation consists of six chapters. Chapter 1 contains the introduction (this chapter). Chapter 2 contains the mathematical preliminaries which consists of notation and definitions used in building a mathematical model for spell checking and correcting. We provide an approach that is similar to the way in which denotational semantic is used to describe programming languages for the spelling error detection and correction tasks. In Chapter 3, we discuss the most important techniques that have been used for spell checking and correcting purposes. In Chapter 4, we extensively study the techniques and algorithms used in selected spell checking/correcting packages. The packages studied are SPELL, CORRECT, ASPELL, SPEEDCOP, the FSA package, AGREP and AURA for English, as well as DNS spell checker for Northern Sotho, and CInsunSpell for Chinese. Each algorithm is described in detail in text and some are assisted by the description in Dijkstra's Guarded Command Language. In Chapter 5, we conducted an empirical investigation on various spell checking/correcting packages studied in Chapter 4. This investigation suggests a classification in terms of characteristics, functionalities, implementation strategies, and performance. The last chapter, Chapter 6, provides conclusions of the work and indicates some directions for future research.

1.3 Intended Audience

The intended audience of this dissertation can be divided into two major groups. Each of these two groups is mentioned in the following paragraphs, along with the associated chapters of particular interest to each group.

1. *Computational linguists.* Spell checking and correcting is a part of the field of natural language processing. This classification will provide computational linguists a unified presentation of the most significant algorithms and techniques which forms a useful basis for their further research in this specific area. The discussion on the selected algorithms and the classification of the various spell checking and correcting packages given in Chapters 4 and 5, respectively, a better understanding of these spell checking and correcting techniques and algorithms.

2. *Programmers.* Programmers working on spell checkers and correctors for word editing applications, Web browsers, webmail sites, blogs, social networking sites, instant messaging clients, and search engines will benefit from the classification. A programmer can make use of the techniques, algorithms, and classification provided in Chapters 3, 4, and 5, respectively, when making a decision on the core technique(s) to be used in their applications. Chapters 3 and 4 provide a detailed background knowledge of the techniques used in various spell checking and correcting algorithms which serves as the foundation for developing better techniques or a hybrid application where more than one techniques are involved for the spell checking and correcting task. A unified representation of the strengths and the weaknesses of these algorithms are provided in Chapter 5, which could assist in improving the performance of an existing application. In addition, programmers who adopt the aspect-oriented programming concept will find the classification in particularly useful. The classification will assist the programmers in efficient decision-making on the suitable techniques to be used in the spell checking/correcting algorithms. They

can then focus on other programming and design concerns, such as aspect-aware interfaces.

Chapter 2

Mathematical Preliminaries

2.1 Introduction

In this chapter, we describe mathematically the spell checking and correcting tasks using an approach that is similar to the way in which denotational semantics is used to describe programming languages. Denotational semantics refers to semantics expressed in terms of a function—i.e. in formal mathematical terms [Mey90]. Notice that although the phrase 'denotational semantics' is conventionally used to describe the meaning of programming constructs using mathematical functions, we shall use the phrase to describe the tasks of spell checking and correcting throughout this dissertation. By formally describing the spell checking and correcting problems, we will be able to understand them better.

The overall task in this chapter is to provide a mathematical model for the following scenario:

Given some text, encoded by some encoding (such as ASCII, UNICODE, etc.), identify the words that are valid in some language, as well as the words that are invalid in the language (i.e. misspelled words) and, in that case, suggest one or more alternative words as the correct spelling.

2.2 Notation

In this section, we define the notation used in the mathematical model. We also present a number of notation and definitions of functions.

Notation 2.2.1. *Assume the following notation:*

- *U denotes a finite alphabet of symbols, i.e. the set of characters in some encoding, typically Unicode. U includes elements such as numbers, special characters (e.g. $', !, \#, \%, \ldots$), and various other characters not found in English words (e.g. à,* *ç, œ, etc.). Chinese would incorporate an entirely different set of encodings, and German would incorporate a few extra symbols in addition to those used in English.*

- U^+ *denotes the set of all strings over* U*. The language* U^+ *is the set of all strings over alphabet* U *of length greater than 0.*

- L *denotes a language over* U*, where* $L \subseteq U^+$ *and* $\neg L = U^+ \backslash L$*.*

- L^+ *denotes a set of languages (a set of valid words), each over* U*, where* $L^+ = (\cup i : 1 \leq i : L^i)$*. Elements of* L^+ *are, e.g. English, French, German, etc.*

- D *denotes a dictionary where it contains a set of valid words in the language.* $D \subseteq L \subseteq U^+$*.*

- T *denotes a text which can be viewed as a (possibly long) sequence of symbols, i.e. as some strings out of the set* U^+*.* $T \in U^+$*. It contains the alphabet of the language, numbers, special characters, and word separators (e.g. blank spaces in an English text).*

- W *denotes a set of words (valid or invalid) over the alphabet* U*, where* $W \subseteq U^+$*. A word* $w \in W$ *consists of letters of the alphabet of the language concerned, i.e. numbers*[1]*, special characters and word separators are excluded.*

- $\mathscr{P}$ *denotes a power set, i.e.* $\mathscr{P}$ *denotes a subset of a set.*

The set containment relationships between various of the above-defined entities is thus given by:

$$\forall L \in L^+ \bullet (D \subseteq L \subseteq U^+) \wedge (L \cup W \subseteq U^+)$$

This gives the relationship between the set of characters used in a language, the set of words in the language, and the set of encodings.

[1]*Numbers are not considered as a word in the language here as it is simply not sensible to deal with an uncountable infinity of real numbers, such as 0.999.*

2.3 Denotational Semantics

2.3.1 Spell checking and correcting problems

In this section, we present spell checking and spell correcting problems mathematically using an approach that is similar to the way in which denotational semantics is used to describe programming languages. In order to formally describe the process of spell checking and correcting in a string of text, we view the task generically, as a sequence of activities, each of which will be functionally described in this section. The first step is to break the input text down into a set of words and possible words in the text—i.e. the removal of blank spaces, duplicates, etc. The second step is to isolate the misspelled words in the text, and the third step is to offer a list of suggested correct spellings for the words deemed to be misspelled.

Spell checking can be divided into two main techniques, namely dictionary lookup and n-gram analysis. We define the two techniques separately. In this section, we present a pair of functions, namely $TextStripping_L$ and $Misses_L$, used in describing the spell checking problem.

We start with a formal statement of the spell checking problem that we consider. The problem is: given some text T, encoded by some encoding (such as ASCII, Unicode, etc.) identify the words that are valid in some language, as well as the words that are invalid in the language, i.e. misspelled words. For a given language, L, we need some operation on this string, that will return all the valid and invalid words in language L in the string. Call this operation $TextStripping_L$. However, it should strip out subsequences that are obviously not words or misspelled words. Thus, it should not return a sequence of blank spaces, a sequence of special characters, numbers, etc. Duplication of words must also be reduced. In other words, this operation should return a *set* of strings. Each string in the set is either a word in language L, or can reasonably be construed to be a misspelling of a word in language L. Note that it is dependent on the language under consideration.

We assume that $W \subseteq U_i^+$ for the purposes of most applications. The precise contents of W will no doubt vary from one implementation to the next. The precise way in which W is constituted, is part of what defines the semantics that is specific to a given implementation[2].

We now formally describe the generic process of spell checking in a string of text as a sequence of activities, each of which will now be functionally described. The first step is to break the text down into a set of words and possible words in the text—i.e. the removal of blank spaces, numbers, special characters, duplicates, etc. The second step is to isolate the misspelled words in the text.

To this end, the first step can be formally described by a function that we choose to call $TextStripping_L$ when it operates on a text in the language L. The function maps some text T, where $T \in U^+$, to a subset of W, i.e.

[2]Indeed, some implementations may even consider strings that include special characters as misspellings, in which case, $W \nsubseteq U_i^+$. To express the denotational semantics of such implementations it would be necessary to appropriately adapt the scheme suggested here.

Function 2.3.1. *($TextStripping_L$):*

$$TextStripping_L : U^+ \rightarrow \mathscr{P}(W)$$

Note that $TextStripping_L$ is a total function—i.e. it is defined for all elements of the domain. It is assumed that if a spell checker is presented with a text, T, that is entirely outside of its scope, then $TextStripping_L$ will return $\phi \in \mathscr{P}(W)$.

Let us take at look at the following two examples:

$$TextStripping_{\text{English}}(\text{the cat sa om the mat}) = \{\text{the, cat, sa, om, mat}\}$$

whereas

$$TextStripping_{\text{English}}(\alpha = 12.34 \text{ is an eqation}) = \{\text{is, an, eqation}\}$$

We now need an operation that further processes the result of $TextStripping_L$, removing all valid words in language L so that what remains is the set of misspelled words. A dictionary[3] D defines the set of strings that are to be regarded as valid words in the language L. Each of the misspelled words will therefore be in the set $(W - D)$. Call $Misses_L$ the operation that identifies misspelled words in terms of D. For some subset of W, $Misses_L$ will deliver the set of misspelled words relative to the set D.

Thus $Misses_L$ has the following signature:

Function 2.3.2. *($Misses_L$):*

$$Misses_L : \mathscr{P}(W) \rightarrow \mathscr{P}(W - D)$$

Let us take a look at an example. Suppose $TextStripping_{\text{English}}$ returns the set {the, cat, sa, om, mat}. Then,

$$Misses_{\text{English}}(\{\text{the, cat, sa, om, mat}\}) = \{\text{sa, om}\}$$

Note that since $Misses_L$ further processes the output of $TextStripping_L$, we can typically compose the two functions, so that $Misses_L(TextStripping_L(T))$ represents the set of misspelled words in language L that is found in text T.

Here, we consider all hyphenated words to be spelled checked as separate words. It is also assumed that case sensitivity is not taken into consideration.

The process of spell correcting often follows the process of spell checking. We thus finally propose the following as a general approach to defining the denotational semantics for the spell correcting problem. An operation is required that—for each misspelled word in the set delivered by $Misses_L$—suggests a *set* of possible alternative spellings in language L. Call the operation $Suggest_L$.

The domain of $Suggest_L$ is therefore the range of $Misses_L$, namely $\mathscr{P}(W - D)$. Each element in the range of $Suggest_L$ is a set of pairs: the first element in the pair is a

[3]Dictionary is used in this chapter as a representative of what a spell checking algorithm uses to check against, which could also be a corpus, lexicon, or word list.

misspelled word (i.e. an element of the set $(W - D)$), and the second element is the set of suggested alternative spellings for this misspelled word (i.e. a subset of D and thus an element of $\mathscr{P}(D)$). Thus, the range of $Suggest_L$ corresponds to a set of pairs, and this set being a subset of the Cartesian product: $\mathscr{P}((W - D) \times \mathscr{P}(D))$.

Because such a set of pairs is synonymous with a function, this function therefore maps elements from the set W (and more specifically, from the set $(W - D)$) to a subset of D. This is to say that an element in the range of $Suggest_L$ can be regarded as a function with the signature $(W - D) \nrightarrow \mathscr{P}(D)$.

As a result of the foregoing, $Suggest_L$ may be regarded as a function with signature:

Function 2.3.3. *($Suggest_L$):*

$$Suggest_L : \mathscr{P}(W - D) \times D \to \mathscr{P}((W - D) \nrightarrow \mathscr{P}(D))$$

As an example, a particular implementation of the $Suggest_{\text{English}}$ function, for some particular dictionary, D could behave as follows:

$$Suggest_{\text{English}}(\{\text{sa, om}\}, D) = \\ \{(\text{sa}, \{\text{sad, sag, sap, sat, saw, sax, say}\}), (\text{om}, \{\text{on, of, am}\})\}.$$

Once again, $Suggest_L$ can be composed with previously defined functions. Given a text T encoded in U for language L, suggestions based on dictionary D for words misspelled in respect of dictionary D will be delivered by $Suggest_L(Misses_L(TextStripping_L(T)), D)$.

For instance, using an English dictionary, D_{English}, the denotational semantics of a particular spell correcting algorithm might result in the following behaviour for the given input text:

$$Suggest_{\text{English}}(Misses_{\text{English}}(TextStripping_{\text{English}}(\text{the cat sa om the mat})), D) = \\ \{(\text{sa}, \{\text{sad, sag, sap, sat, saw, sax, say}\}), (\text{om}, \{\text{on, of, am}\})\}$$

Note, in reality, we may not only want the $Suggest_L$ function to simply map each misspelled word to a set of elements, but a sequence of suggested words. The order of the sequence then reflects the function's estimates of the most likely to the least likely correct spellings.

Also note that, above we have assumed that words are in their inflected forms. In practice, before executing the $Misses_L$ function, implementations generally strip off prefixes and/or suffixes from words based on some morphological analysis or affix analysis.

2.3.2 n-gram analysis

n-grams are n-letter subsequences of sequences, either of letters or words, where $n \in \{1, 2, 3, \ldots\}$. If $n = 1, 2, 3$ they are referred to as unigrams, bigrams, trigrams, respectively. In other words, n-grams are substrings of length n. We will discuss n-gram analysis in more details in Sections 3.3.2 and 3.4.4.

The action of determining n-grams may be formally described as follows:

As before, let U be a set of finite alphabet of symbols. The *ngram* operation can be seen as a function that maps some text, which is an element of U^+, to a set of n-tuples. Let U_n denote the set of n-tuples, i.e. $U_n = U \times U \cdots \times U$. The domain of *ngram* is then $\mathscr{P}(U_n)$. Thus,

Function 2.3.4. *(ngram):*

$$ngram : U^+ \rightarrow \mathscr{P}(U_n)$$

In addition, the *ngram* function can be defined as follows:

Function 2.3.5. *(ngram):*

$$ngram(u : U^+) = \{y | u = xyz \wedge |y| = n\}$$

where xyz represents a string of words, where x, y, and z each represents a substring of this particular string of words.

Let us take at look at the following example:
For $n = 3$,

$$3gram(\text{the sky is blue}) = \{\text{the, he-, e-s, -sk, sky, ky-, y-i, -is, s-b, -bl, blu, lue}\}$$

where each '-' represents a blank space.

An n-gram encountered in a text may or may not be flagged as an error. Generally, this would depend on an analysis of the surrounding text in reference to pre-compiled statistical tables (which will be discussed in further details in the following chapter). In flagging a particular n-gram as an error, there may be a risk—albeit a small one—of doing so incorrectly. Furthermore, n-gram analysis can be employed irrespective of the language involved.

2.4 Conclusion

This chapter is intended to provide the basis for spell checking and correcting problems. We provided the 'denotational semantics' of the spelling error detection and correction tasks in order to describe them formally in mathematics. The mathematical models assist us to understand the problems better by looking at their simplest forms. In the subsequent chapter, we will be discussing the most significant techniques that have been used for spell checking and correcting purposes.

Chapter 3

Spell Checking and Correcting Techniques

3.1 Introduction

This chapter describes the most significant spell checking and correcting techniques that have been used to solve in code, problems that were described mathematically in the previous chapter. This chapter forms a basis for Chapter 4 that provides algorithmic descriptions of these techniques used to detect spelling errors and suggest corrections.

Most spell checking and correcting techniques and algorithms that have been developed during the past few decades, have been based on dictionary lookup techniques (will be discussed in Section 3.3.1). In the nineties, probabilistic techniques for spell correcting were also developed and employed for this specific purposes for the first time. Most of the spell checkers and correctors follow the two-layer approach: error detection and error correction. Few combine the two layers into one. The tasks of spelling error detection and correction are well defined in the previous chapter. The denotational semantics of the input text preparation ($TextStripping_L$) was provided in Function 2.3.1. In Function 2.3.2, the denotational semantics of the spell error detection task ($Misses_L$) was given. In Function 2.3.3, the denotational semantics of the spell correcting task ($Suggest_L$) was given. These functions provide a concise and accurate statement of the *what* that needs to be done. Here an indication is given as to *how* this is done. The *how* is addressed from two perspectives: on the one hand, detailed descriptions of general techniques are discussed; on the other hand, detailed descriptions of specific example algorithms are provided.

In this dissertation, we distinguish the techniques used for spell checking and correcting purposes from the actual spell checking and correcting algorithms investigated. The techniques surveyed are a general approach to the spell checking and correcting tasks as opposed to detailed algorithmic description, which comes later in Chapter 4. There are a variety of techniques which in essence compare what is given (such as an input text) against what is known (such as a dictionary, a lexicon[1], or a corpus). Data structures

[1]A lexicon is referring to a word list without accompanying definitions which is essentially a dictionary without definitions. In this dissertation, we use the words ‘dictionary’ and ‘lexicon’ to represent a text

may differ across techniques in respect of how to efficiently store what is known—various dictionary structures and/or statistical information.

Techniques may also differ in respect of how to represent input—in raw input form versus some morphological characterisation which is briefly described in the subsequent section.

3.2 Morphological Analysis

Morphological analysis and processing techniques are used to identify a word-stem from a full word-form in natural language processing. Words are broken down into morphemes which are the smallest linguistic units that carries a meaningful interpretation. These are typically word-stems (i.e. the root or the basic form of a word) or affixes (i.e. a morpheme that is attached to a stem). Let us consider an example. The word *unreachable* has three morphemes: 'un-', '-reach-', and '-able'. The word stem in this case is '-reach-'. 'un-' is the prefix and '-able' is the suffix. Both prefixes and suffixes are affixes. Thus, 'un-' and '-able' are the affixes in this word. A morphological analyser reverses the spelling rules for adding affixes. For example, the word *apples* can be interpreted as 'apple' and 's' by the analyser.

Morphological processing involves parsing a sequence of input morphemes on the surface level or the lexical level (i.e. with respect to the word grammar). Listing all orthographic forms (i.e. spelling) is not practical for some languages, such as Finnish and Turkish, where each word-stem may easily have hundreds, thousands, or even millions of inflected forms. Words in these languages are formed by adding zero or more affixes to the base of a word, i.e. a word-stem.

For instance, the word *kirjailija* in Finnish means an 'author'. The word-stem is *kirja*, which means a book. *kirjallisuus* means literature. Thus, words are formed by continuously adding to the word-stem. According to Hankamer [Han89], Turkish is estimated to have 200 billion orthographic words (i.e. word spellings) in total. Due to the nature of these agglutinated languages, researchers designed finite-state morphological analysers and generators.

A morphological analyser determines the common word-stem of various words by stripping off the affixes of these words. A morphological generator adds affixes to a word-stem to determine the various words that can be generated from the same word-stem.

A few significant examples of morphological analysers are KIMMO [Kar83], which employed the two-level morphological model developed by Koskenniemi [Kos83], the Alvey Natural Language Toolkit [RPRB86], and the morphological analyser used in the FSA package by Daciuk [Dac98], which will be discussed in Chapter 4.

There exist two morphological levels: the strings from the surface level and the ones from the lexical level [Dac98]. The string from the surface level refers to an inflected form of a word. For instance, the word *unreachable* is an inflected form of the word *reach* as explained earlier. The string from the lexical level refers to the corresponding lexeme[2]

containing only correctly spelled words. We use these words interchangeably throughout this dissertation.

[2]A lexeme stands for the abstract entity that correlates with different dictionary entries, i.e. it is a

together with morphological annotations. Morphological annotations are used to describe the properties of a word form, such as the tense of a word. For example, the word *has* is the inflected form of *have* and has the annotations {single, present tense}.

Dictionary lookup techniques alone are thus not feasible for any morphologically complex languages, i.e. a morphological analyser is also required. Furthermore, a morphological analyser is also more handy in more complex languages. This is because a spell checker or corrector that is not equipped with a morphological analyser requires a much larger dictionary. However, this process may or may not be present in all spell checking and correcting algorithms (this claim will be obvious in Chapter 4). This is the reason why morphological analysis or affix stripping was not defined in the previous chapter.

3.3 Spell Checking Techniques

The spell checking problem was described in denotational semantics in formal mathematical terms in Function 2.3.2. It was seen that the spell checking task can be described by a function that maps a set of words as an input to a set of correct spellings. This task amounts to obtaining the intersection of a set of words and a dictionary. In addition, spell checking involves determining whether an input word has an equivalence relation with a word in the dictionary. Such an equivalence relation means that the two words can be interchanged in all context without alternating the meaning [HD80]. The equivalence relation is established by means of an exact string matching technique (i.e. there is either a match or there is not.). This present section gives an outline of how spell checking is achieved in practice by means of exact string matching techniques. The survey of spell checking techniques in Kukich's [Kuk92] remains the definitive reference to date. The work by Jurafsky and Martin [JM00], although writing almost a decade later, still draws on Kukich's work. More recent studies found in [HA03, dSP04] and the literature survey of Bodine [Bod06] also correspond to Kukich's work. This suggests that the string searching techniques included in Kukich's survey represent the most appropriate methods for the spell checking task.

Kukich [Kuk92] pointed out that techniques to detect non-word spelling errors in a text can be divided in two categories: dictionary lookup, discussed in Section 3.3.1; and n-gram analysis, discussed in Section 3.3.2. A non-word refers to a continuous string of characters and/or numbers that cannot be found in a given dictionary or that is not a valid orthographic word form. Dictionary lookup employs efficient dictionary lookup algorithms and/or pattern matching algorithms. It may also employ one of the various dictionary partitioning schemes and may rely on the various morphological processing techniques (briefly described in Section 3.2) for breaking up and/or analyzing both the input word and the stored dictionary. On the other hand, n-gram analysis makes use of frequency counts or probabilities of occurrence of n-grams in a text and in a dictionary, a lexicon, or—in most cases—a corpus.

word which is used as the main entry in the dictionary for all its inflected forms.

3.3.1 Dictionary lookup techniques

Dictionary lookup techniques are employed to compare and locate input strings in a dictionary, a lexicon, a corpus or a combination of lexicons and corpora. These are standard string searching techniques with a specific aim: to reduce dictionary search time. In order to serve the purpose of spelling error detection, exact string matching techniques are used. If a string is not present in the chosen lexicon or corpus, it is considered to be a misspelled or invalid word. At this stage, we assume that all words in the lexicon or corpus are morphologically complete, i.e. all inflected forms are included. We will be discussing more on morphological analysis, as well as dictionary partitioning and dictionary construction related issues in Section 3.5.

The dictionary lookup techniques described in Kukich's survey focus on reducing dictionary search time via efficient dictionary lookup and/or pattern-matching algorithms, via dictionary partitioning schemes and via morphological-processing routines (described in Section 3.2). The most significant dictionary lookup techniques are hashing, binary search trees, and finite-state automata. We will now describe these techniques concisely in the following sections.

Hashing

Hashing is a well-known and efficient lookup strategy, discussed in works such as [Blo70], [CLRS03], and [Knu98]. The basic idea of hashing relies on some efficient computation carried out an input string to detect where a matching entry can be found. More specifically, hashing is a technique used for searching an input string in a pre-compiled hash table via a key or a hash address associated with the word, and retrieving the word stored at that particular address. If collisions occurred during the construction of the table, a link or two must be traversed or some form of linear search must be performed. Hence, if a hash function tends to produce identical or similar hash addresses, the lookup process will be slower as a result.

In the spelling checking context, if the word stored at the hash address is the same as the input string, there is a match. However, if the input word and the retrieved word are not the same or the word stored at the hash address is null, the input word is indicated as a misspelling. The random-access nature of hash tables eliminates the large number of comparisons required for lookups. Thus, it is a faster searching technique compared to sequential search or even tree-based search, especially for searching in a large data representation. The drawback of a hash table is that without a good hash function, it would require a big hash table in order to avoid collisions. Hash tables provides $\mathcal{O}(1)$ constant lookup time on average. In the worst-case scenario, hashing requires the time complexity of $\mathcal{O}(m)$, where m is the number of words in a dictionary or corpus.

The SPELL package by Unix® is an example of where hashing was employed for fast dictionary lookup and reduced memory usage. We will take in-depth look at how SPELL made use of hashing to perform the task of spell checking in Section 4.2.

Binary search trees

Binary search trees [Knu98, CLRS03], particularly median split trees, have been used for dictionary lookup [She78] and subsequently, for spell checking. Binary search trees are especially useful for checking if a particular string, i.e. an input word, exists within a large set of strings, i.e. the chosen dictionary. A median split tree is a modification of a frequency-ordered binary search tree. The main goal of a median split tree is to make access to high-frequency words faster than to low-frequency words without sacrificing the efficiency of operations, such as word lookup, insert, and delete.

Median split trees (MSTs) [Com80] are essentially binary search trees where each node of a split tree contains a node value and a split value. A node value is a maximally frequent key value in that subtree. A split value partitions the remaining keys between the left and right subtrees in a lexical order, which is to say that it is the lexical median of a node's descendants. Hence, the tree is perfectly balanced. Median split trees are data structures for storing dictionaries, lexicons or corpora with highly skewed distributions. This is to say, they are particularly useful for efficiently finding words when the underlying frequency distribution of the occurrence of these words is highly skewed, such as often is the case in an English text. A median split tree is constructed from a set of keys, which comprise both frequency and lexical orderings. The frequency ordering is a weak linear order of the keys according to their frequencies of occurrence in the input while as the lexical ordering is a strict linear order on the keys' representation. Thus, the cost of lookup is determined by both the frequency distribution and the lexical ordering the keys. The build time is $\mathcal{O}(n \log n)$.

As pointed out by Sheil [She78], "unlike frequency ordered binary search trees, the cost of a successful search of an MST is $\log n$ bounded and very stable around minimal values." In other words, they require a maximum of $\log n$ searches to locate the looked up word within a set of n strings. It is also significantly more efficient compared to the lookup time of a linear search technique on a large data representation although it is generally slower compared to the lookup time of hashing. Median split trees also require less storage space than frequency-ordered binary search trees.

An instance of a spell checker using binary search trees can be found in [LS99]. Another very recent project [You06] also made use of binary search trees to perform fast spell checking. The worst case lookup time for a binary search tree is $\mathcal{O}(\log n)$.

Finite-state automata

Finite-state automata as presented in Aho and Corasick and Watson in [AC75, Wat95], respectively, have been used as a basis for string-matching or dictionary-lookup algorithms that locate elements of a dictionary within an input text. Using finite-state automata for string matching started in the late 60s. A finite-state automaton (FSA) is used to represent a language, which is defined to be a set of strings, each string being a sequence of symbols from some alphabet.

Consider an FSA that is required to represent any string in the language $U^*(w)$, where U is a finite alphabet of symbols, and w is some word consisting of i symbols in U (i.e. $|w| = i$). The language is thus any string of symbols from U that has w as a suffix. Such an FSA, $FSA(w) = (Q, q_0, A, T)$ can be defined as follows:

- Q is the set of all states corresponding to the prefixes of w. Thus, Q can be represented as $\{\epsilon, w[0], w[0..1], \cdots, w[0..c-2], w\}$.
- $q_0 = \epsilon$ is the start state.
- In general, A is the set of accepting states where $A \subseteq Q$. In this particular case, $A = \{w\}$.
- T is the transition function which maps a state and an input symbol to another state. Let $q \in Q$, where q is a prefix of w and let $c \in U$, where c is a character in the alphabet. Then $(q, c, qc) \in T$ if and only if qc is also a prefix of w. Otherwise, $(q, c, s) \in T$, where s is the longest suffix of qc that is also a prefix of w.

Such an FSA would only terminate in the final state if the last i symbols of the input of n symbols $(n \geq i)$ constituted the word w. The state transition function can be visually represented by the well-known state transition diagrams. If the transition function is stored as a table, then the search time is $\mathcal{O}(n)$. Please consult [Wat95] for a derivation of this algorithm.

Finite-state transducers are automata that are used to translate or transduce one string into another. Finite-state transducers are also called Mealy's automata [HMU06]. Each transition of a finite-state transducer is labelled with two symbols from U_1 and U_2, where U_1 and U_2 are alphabets of input and output, respectively. The two alphabets are often the same. For example, a/a, where a is the input and b is the output. All the definitions that hold for finite-state automata also hold for finite-state transducers.

One specific form of finite-state automata that has been widely used for spell checking and correcting purposes is a trie data structure. Tries are also known as prefix trees and can be seen as representing the transition function of a deterministic FSA even though the symbol on each edge is often implicit in the order of the branches. Details of tries are provided in [Knu98].

An example of incorporating a trie in the spell checking and correcting algorithm can be found in the Research of Chinese Text Proofreading Algorithms by Li et al [LWS00]. Tries are often used in approximate string matching algorithms. Spell checking and correcting using finite-state automata was pioneered by Oflazer [Ofl96] and Oflazer and Gzey [OG94]. The FSA package which will be discussed in Section 4.6 based its spell checking and correcting approach on the work of Oflazer and Gzey along with its own modifications.

Finite-state approaches are often used for spelling correction for agglutinating (highly inflectional) languages or languages with compound nouns (e.g. German). They are also used when a multilingual dictionary is involved in the spell correcting process or as a system for post-correction of results of OCR, where the scanned texts contain vocabularies of several languages.

Note that other standard string searching techniques such as linear (or sequential) search is not in favour for large dictionary lookup and spell checking and correcting tasks because they generally have slower lookup time compared to the aforementioned techniques.

Linear search has an expensive lookup time $\mathcal{O}(n)$ in general, where n is the number of words in a dictionary, as it must examine each element of the list in order. They thus require, on average, a large number of comparisons during searching and as a result the lookup process is slowed down.

Binary search can considerably speed up dictionary lookup time on average. For instance, if a spell checker made use of a binary search tree of English words, the tree could be balanced based on word occurrence frequency in the corpus. Words such as "a" or "the" would be placed near the root whereas words such as "asymptotically" would be near the leaves.

3.3.2 n-gram analysis

As described in Section 2.3.2, an n-gram is an n-letter subsequences of a sequence, either of letters or words, where $n \in \{1, 2, 3, \ldots\}$. If $n = 1, 2, 3$, reference is made to a unigram, bigram, or trigram, respectively. In other words, n-grams are substrings of length n. Function 2.3.4 provided the denotational semantics of the spell checking task in formal mathematical terms. The spell checking task can be seen described by a function that maps some text to a set of n-tuples where the tuples are checked for correctness, taking their context into account.

n-gram analysis is used to estimate whether each n-gram encountered in an input string representing some text, is likely to be valid in the language. A table of n-gram statistics is pre-compiled from large corpora for comparative purposes. These statistics are either in the form of frequency counts or binary values, as will be explained below.

Note that for spell checking, a probabilistic approach that relies on frequency counts is generally preferred, rather than making use of binary values. Nevertheless, in each case, a given lexicon or corpus is divided into n-grams and stored in a data structure, such as an array or a vector space in order for the sequence to be compared to other sequences in an efficient manner.

Due to the nature of non-uniform distribution of possible n-grams in a dictionary or lexicon, n-gram analysis is a reasonable technique. For example, the letters in the English alphabet can be paired in 26^2 different ways. However, not all of these pairings occur in the actual language, so that only a subset of these different pairings would be bigrams in some English text. To illustrate this, the Pride and Prejudice text which was taken from Project Gutenberg[3] [Aus98] contains 122,813 words. Table 3.1 indicates that only 432 different possible bigrams actually occurred. This is to say that only 64% of the 26^2 different bigrams actually occurred. This figure is similar to the results found in the

[3]Project Gutenberg is the oldest digital library which contains the full texts of many public domain books. The text in Project Gutenberg was chosen to be analysed in the tables below for its search result from Google. At the time of writing, Project Gutenberg has a Google search result of approximately 3,860,000; Brown Corpus is approximately 219,000; British National Corpus 330,000; and Canterbury Tales achieved a search result of approximately 1,090,000. The number of search results provides an indication of the degree of recognition of each of these sources to the general public. Project Gutenberg was the most widely referenced source in this case. The n-gram tables derived from the Pride and Prejudice text found in Project Gutenberg was set up by the author in order to demonstrate the concepts of n-gram analysis.

research of Peterson [Pet80] and Zamora et al [ZPZ81] who have done extensive work on spelling checking using n-grams.

There exist two approaches for n-gram analysis:

- *Binary-value approach.* A two-dimension bigram array containing binary values of each combination of letters or words is pre-compiled into a table in which 0 and 1 are used to present possible and impossible n-grams, respectively, as determined by the n-grams present in the chosen lexicon. Every n-gram in each of the words must have an occurrence of 1, otherwise the subsequence of text encapsulating the n-gram will be flagged as incorrect.

- *Probabilities approach.* Probabilistic values are derived from frequency counts of n-grams encountered in large corpora. The n-grams in an input string are compared to the n-gram statistics table. An index of peculiarity for each word on the basis of the n-grams it contains is then calculated. The n-gram analysis technique provides a quick and rough means for non-word error detection. However, because the analysis is probabilistically conditioned, it is possible that a word which is marked valid by n-gram analysis can in fact be a misspelled word and vice versa.

Binary n-gram analysis techniques have been widely used in OCR, which translates an image of text, such as a scanned document, into actual text characters, and in its variant, Intelligent Character Recognition (ICR) devices. ICR is the preferred option for addressing more complex recognition problems, such as recognition of hand-written or cursive scripts. Grammatical or contextual information is necessary in the recognition process. n-gram analysis is also used in conjunction with other techniques, such as dictionary lookup and morphological analysis, to achieve higher accuracy in spelling error detection.

An example of non-positional binary bigram array for English is shown in Table 3.1. As previously mentioned, the table is generated from Price and Prejudice which was found in Project Gutenberg [Aus98]. The text contains 122,813 words and 536,407 characters. Table 3.1 is to be read as a row entry, X, being the first symbol and then a column entry, Y, being the next. For example, *aa* has a binary value 0 whereas *ba* has a binary value 1. When spell-checking a given text, every n-gram in each of the words in the text must have an occurrence of at least 1. Where this is not the case, the relevant words will be flagged as incorrect.

For the probabilities approach, once the input word has been decomposed into its constituent n-grams, each n-gram is assigned a weight. This weight reflects its frequency of occurrence in some lexicon or corpus. The weights of these n-grams are used in the construction of an index of peculiarity. The peculiarity of an n-gram is determined by some pre-defined threshold value. If some n-grams' weights fall below this threshold, they seem as most peculiar. They are then flagged as potential errors and identified as likely candidates for misspellings. These erroneous flags could therefore be wrong with a certain probability. Likewise, if the word is flagged as correct, then could also be wrong with a certain probability. Thus, the probabilities approach is typified by constituent analysis.

A constituent analysis does not attempt to match any input string with a dictionary entry as would have been done by dictionary lookup techniques described earlier in this

[X,Y] = Reads Y after X

														Y													
		a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
	a	0	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
	b	1	1	0	1	1	0	0	1	1	1	0	1	1	0	1	0	0	1	1	1	1	1	0	0	1	0
	c	1	0	1	1	1	0	0	1	1	0	1	1	0	0	1	0	1	1	1	1	1	0	0	0	1	0
	d	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	0	1	1	1	0	1	0
	e	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
	f	1	0	0	0	1	1	0	0	1	0	0	1	0	1	1	0	0	1	1	1	1	0	0	0	1	0
	g	1	1	0	1	1	0	1	1	1	0	0	1	1	1	1	0	0	1	1	1	1	0	0	0	1	0
	h	1	1	0	1	1	1	0	1	1	0	0	1	1	1	1	0	0	1	1	1	1	0	1	0	1	0
	i	1	1	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
	j	1	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0
	k	1	0	0	0	1	1	1	1	1	0	0	1	0	1	1	0	0	0	1	0	0	0	1	0	1	0
	l	1	0	1	1	1	1	1	0	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	0	1	0
X	m	1	1	0	0	1	1	0	1	1	0	0	1	1	1	1	1	0	1	1	1	1	0	0	0	1	0
	n	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	o	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	p	1	1	0	0	1	1	0	1	1	0	0	1	1	0	1	1	0	1	1	1	1	0	0	0	1	0
	q	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
	r	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	0	1	0
	s	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0
	t	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	1	1	1	0	1	0	1	1
	u	1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	0	1	1	1	0	1	0	1	1	1
	v	1	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	1	0
	w	1	0	1	1	1	1	0	1	1	0	1	1	0	1	1	0	0	1	1	1	0	0	0	0	0	0
	x	1	0	1	0	1	1	0	1	1	0	0	0	0	0	0	1	1	0	0	1	1	0	0	0	1	0
	y	1	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	1	1	1	0	0	1	0	0	0
	z	1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1	1

Table 3.1: An example of a simple non-positional binary bigram array for English.

chapter. Instead, a peculiarity is associated with each input by some algorithm. This type of approach can be faster than search routines. However, they can also be less exact as the accuracy of n-gram analysis is proportional to how precisely the character composition of the input text mirrors that of the dictionary in use. Morris and Cherry [MC75] used this approach in the TYPO system. TYPO made use of peculiarity indexes based on trigrams to check whether a word was a misspelling. A peculiarity index (PI) is calculated using the following equation:

$$PI = \frac{[\log(f(xy) - 1) + \log(f(yz) - 1)]}{2} - \log(f(xyz) - 1) \tag{3.1}$$

where xy and yz both represent bigrams, xyz represents a trigram, and f is the frequency function.

For instance, to calculate the peculiarity index of a word *onh*, xy would be 'on', yz would be 'nh', and xyz would be 'onh', the entire word itself. It is likely that the frequency of 'on' would be relatively high and that the frequencies of 'nh' and 'onh' would be low. The peculiarity index would thus be relatively high, and consequently, the word *onh* is likely to be flagged as a misspelled word.

Zamora et al [ZPZ81] attempted to minimize the problem of incorrectly flagging accurate spellings by calculating the ratio of the frequency of occurrence of an n-gram in a misspelling to its overall frequency. In other words, an n-gram error probability is calculated rather than using straightforward frequency measures. The attempt successfully avoided many correctly, however, unusually spelled words.

n-gram analysis is often used in conjunction with other techniques, such as dictionary lookup, phonetic representation, and morphological analysis (described in Section 3.2) to achieve more desirable accuracy in spelling error detection. An example of non-positional[4] frequency bigram array for English is shown in Table 3.2. The table is again generated

[4]Non-positional means that the bigram may be located anywhere in a word.

from Price and Prejudice which was found in Project Gutenberg. This table is only an extract. It reads X first and then Y. For example, 'aa' is not a valid bigram as it never occurred in the text, i.e. its frequency count is 0. However, the bigram 'ba' has a frequency count of 289 and it is indicated as a valid bigram.

[X,Y] = Reads Y after X

											Y										
		a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t
	a	0	1536	1116	2417	6	463	843	6	1564	0	584	2798	1326	7944	16	771	0	4137	4818	5633
	b	289	7	0	1	4164	0	0	7	426	184	0	1005	4	0	639	0	0	322	154	106
	c	1399	0	324	3	2481	0	0	2252	523	0	428	328	0	0	3084	0	100	317	7	1279
	d	1059	3	0	206	2541	20	122	3	1913	1	1	149	85	61	915	0	1	238	403	0
	e	3099	39	1629	5404	1884	677	322	150	1063	45	85	3262	1372	6144	93	625	187	11315	3532	2591
	f	776	0	0	0	1116	572	0	0	819	0	0	135	0	2	2227	0	0	837	3	423
	g	726	88	0	4	1291	0	34	1506	540	0	0	491	9	121	547	0	0	628	222	40
	h	6301	52	0	9	14824	11	0	4	4487	0	0	43	84	5	2629	0	0	159	27	920
	i	708	298	1459	1319	1483	724	874	5	0	0	236	1749	1707	10049	2124	173	5	1463	4844	4818
	j	300	0	0	0	234	0	0	0	1	0	0	0	0	0	152	0	0	0	0	0
	k	17	0	0	0	1045	37	2	195	488	0	0	18	0	474	2	0	0	0	98	0
	l	1427	0	29	1667	3569	628	18	0	3128	0	238	3101	74	21	1323	45	0	28	220	251
X	m	1879	198	0	0	3168	58	0	1	1482	0	0	14	237	21	1299	706	0	1129	294	8
	n	550	3	1651	5585	4011	224	4789	29	851	69	359	383	12	780	3350	21	73	14	1460	3412
	o	103	340	218	538	94	3947	108	99	335	2	375	1025	2193	6021	1304	581	4	3839	1114	2683
	p	981	2	0	0	1704	1	0	102	415	0	0	886	2	0	1031	695	0	1339	161	325
	q	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	r	1340	54	681	857	7354	210	204	112	2023	1	112	460	406	606	1872	127	0	578	1877	1287
	s	1346	55	325	35	3809	129	35	2937	2161	0	98	139	130	19	1813	681	4	7	1829	3472
	t	1527	1	100	1	4395	87	0	13746	3108	0	0	697	69	74	5188	1	0	790	497	1172
	u	451	231	1025	255	390	52	862	0	357	0	1	1758	238	1421	7	454	0	2608	1668	1833
	v	358	0	0	0	4269	0	0	0	853	0	0	0	0	0	235	0	0	1	0	0
	w	2723	0	5	6	1789	3	0	2320	2505	0	13	63	0	460	995	0	0	125	85	1
	x	69	0	156	0	63	1	0	7	71	0	0	0	0	0	0	292	1	0	0	148
	y	5	63	1	171	352	7	0	1	163	0	0	17	43	16	2146	3	0	3	342	181
	z	659	0	0	0	33	0	0	0	5	0	0	4	0	0	0	0	0	0	0	0

Table 3.2: An example of a simple non-positional frequency bigram array for English.

An n-gram probability is derived from the ratio of the frequency of occurrence of an n-gram to the overall frequency of occurrences of all n-grams. For example, the bigram 'he' has a frequency count of 14,824 according to Table 3.2 and the total frequency count of all the bigrams generated is 413,594. Thus, the probability of occurrence for 'he' is approximately 0.035842. Table 3.3 illustrates how a probability distribution can be derived from the frequency table (Table 3.2), where the probabilities of all possible n-grams sums to 1 (i.e. the total frequency count of all possible n-grams, 413,594, has the probability of 1).

The use of n-gram analysis is illustrated in detail in the algorithms described in Sections 4.5, 4.8, 4.9, and 4.10. The probabilistic approaches were adopted by these algorithms.

bigram	frequency	probability
he	14824	0.035842
th	13746	0.033235
er	11315	0.027358
in	10049	0.024297
an	7944	0.019207

Table 3.3: An extract of the probabilities of bigrams generated from Table 3.2.

3.4 Spell Correcting Techniques

Spell correcting refers to the attempt to endow spell checkers with the ability to correct detected errors, i.e. to find the subset of dictionary or lexical entries that are similar to the misspelling in some way. Function $Suggest_L$ (2.3.3) provided the denotational semantics of the simple spell correcting task in formal mathematical terms. It was seen that the spell correcting task can be described by a function that maps a misspelled word to a set of possible correct spellings. Spelling correction must involve a dictionary or corpus, since the set of possible correct spellings are defined in terms of membership in the chosen dictionary or corpus. There are two types of spell correctors: interactive and automatic. The simplest spell corrector is interactive, which is to say it provides the user with a list of candidate corrections and leaves the final decision of word choice to the user. The automatic approach of spelling correction requires a significant level of machine intelligence as it is expected to correct spelling errors automatically without any user involvement. The automatic approach can be seen in some text-to-speech synthesis [Kuk90]. It is also available as an option in the Microsoft Word spell checker. However, this option only applies to a limited subset of words.

Spell correcting can be categorised by isolated-word error correction and context-dependent error correction, that is to say, attention is paid to isolated words versus textual or even linguistic context. Isolated-word error correction thus refers to spell correcting without taking into account any textual or linguistic information in which the misspelling occurs. Similarly a context-dependent corrector would correct both real-word errors[5] and non-word errors involving textual or linguistic context. In this research, we focus exclusively on isolated-word error correction which will serve as a basis to future research on context-dependent error correction.

Research on isolated-word error correction started in the late 50s and early 60s [Bla60]. The design of a particular correction technique is often influenced by the type of spelling error patterns. Distinctions of spelling error patterns are made between typographic (e.g. *taht* vs *that*), cognitive (e.g. *seprate* vs *separate*) and phonetic (e.g. *recieve* vs *receive*) errors [Kuk92]. Typographic errors occur when a wrong key is pressed, when two keys are pressed instead of one, or when keys are pressed in the wrong order. Cognitive errors occur when there is a misconception or a lack of knowledge of the correct spelling of a word. Phonetic errors are merely a special case of cognitive errors. A phonetic error refers to a misspelled word that is pronounced the same as the correct word, but whose spelling is nevertheless incorrect.

[5]A real-word error is an error that results in another valid word which occurs when a valid word is substituted for another.

Kukich [Kuk92] pointed out that roughly 80% of spelling errors tend to be single-letter errors, such as insertions, deletions, substitutions, and transpositions[6]. Furthermore, these misspellings tend to be within one letter difference in length of the intended word. Also, few errors occur in the first character of a word. To reduce lookup time, these findings were incorporated in algorithms for correcting single errors and/or dictionary partitioning according to the first characters and/or the length of a word.

Spelling error detection was described mathematically in Function 2.3.2 and various efficient techniques for this task were discussed in Section 3.3. Spelling error correction relies on some approximate string matching technique [HD80] to find a set of correctly spelled words with similar spellings to a misspelled word. This is to say, spelling error correction involves the association of a misspelled word with one word or a set of correctly spelled words in the dictionary that satisfy a similarity relation. By similarity relation, it is meant that two words are associated on the basis of some criterion of similarity. For example, the words *ond* and *and* have a similarity relation. If the first character of *ond* is substituted by an 'a', a non-word is then changed to become a valid word and the criteria of similarity is determined by *substitution*.

The spell correcting problem consists of three subproblems: spelling error detection; generation of possible correct spellings; and the ranking of the suggested corrections. These subproblems are generally treated in sequence. The first procedure of detecting spelling errors was discussed in detail in the previous section (Section 3.3). The third procedure, ranking candidate corrections, involves either some lexical-similarity measure between the misspelled words and the candidate suggestions or a probabilistic estimate of likelihood of each of the correct word in order to rank the suggestions. This procedure is sometimes omitted, especially in interactive spell correctors. The decision of selecting a word is then left to the user.

We will be looking at techniques of possible correct spelling generation and ranking the candidate corrections in this section. To achieve isolated-word error correction task, six main techniques have been explored, and they are: minimum edit distance, similarity key, rule-based techniques, n-gram-based techniques, probabilistic techniques, and neural networks. We will now give a flavour of each of these six techniques in the following subsections. These descriptions will necessarily be somewhat brief. For more complete descriptions, the various works cited in the relevant subsections should be consulted. The relative importance of the various techniques will also be discussed in the following subsections.

3.4.1 Minimum edit distance

Minimum edit distance (also called edit distance) technique is the most studied and used technique for spelling correction to date. Minimum edit distance as its name suggests is a technique where the minimum number of editing operations (insertions, deletions, substitutions, and transpositions) required to transform one string into another. The notion of using edit distance for spelling correction was first introduced by Wagner [Wag74]. Levenshtein [Lev66] introduced edit distance algorithm for carrying out correction operations on

[6]A transposition is equivalent to a double substitution. It is also called a reversal.

so-called insertions, deletions, and substitutions. Damerau edit distance [Dam64] allows the three types of editing operations mentioned and a transposition between two adjacent characters. Damerau stated in [Dam64] that these four editing operations correspond to more than 80% of all human-made misspellings.

Edit distance [WF74, OL97] entails distance measures between two strings. An insertion occurs when a letter needs to be inserted to a misspelled word, resulting in a correctly spelled word. A deletion occurs when a letter needs to be deleted from the misspelled word in order to result in a correctly spelled word. A substitution refers to the replacement (or substitution) of a letter in the misspelled word by a correct letter, thus resulting in a correctly spelled word. A transposition takes place when the positions of two adjacent letters are reversed and they need to be swopped in order to result in a correctly spelled word.

The term, *minimum edit distance* between two spellings, say w_1 and w_2, refers to the smallest number of editing operations that need to take place in order to transform w_1 to w_2. The editing operations referred to here are insertions, deletions, substitutions, and transpositions. Normally, one is concerned about the minimum edit distance between a misspelled word in a text to a word in the dictionary [Wag74]. Wagner introduced the use of an application of dynamic-programming techniques (such as the minimum edit distance technique) to spelling correction in order to reduce the runtime.

Dynamic programming was invented by Bellman [Bel57]. The essence of dynamic programming is to solve multistage decision problems. Dynamic-programming techniques make use of overlapping subproblems, optimal structure, and memoization[7] in order to reduce the runtime of algorithms [Wag95, CLRS03].

- *Overlapping subproblems* refers to the decomposition of a problem into a series of subproblems whose solutions may be reused one or more times in solving the original problem. The computation of the Fibonacci sequence is an example of where overlapping subproblems have to be solved in order to solve a problem.

- The second notion, *optimal substructure*, refers to finding an optimal solution to subproblems in order to efficiently construct an optimal solution of the overall problem. An example of this property can be exhibited by determining the shortest path between two adjacent vertices in an acyclic graph and then using this finding to determine the shortest overall path.

- The third notion, *memoization*, is a programming technique used to improve the program's efficiency by storing the results of computations for future reuse, rather than computing them again.

As pointed out by Damerau [Dam64], an early analysis by Gates [Gat37] showed that a large majority of spelling errors could be corrected by the insertion, deletion or substitution operation of a single letter, or the transposition of two letters. If a misspelling can be transformed into a dictionary word by reversing one of the error operations (i.e. insertion, deletion, substitution, and transposition), the dictionary word is said to be a

[7]Note that memoization is not the same as memorization.

plausible correction. This technique is limited to single-word errors. Thus, the number of possible comparisons is greatly reduced.

Damerau's edit distance algorithm [Dam64] detects spelling error by matching words of four to six characters in length to a list of words with high frequency of occurrence (i.e. frequently used words). If the search word is not found in the list, the word is looked up in a dictionary in which words have been sorted according to alphabetical order, word length, and occurrence of characters. If the search word cannot be found in the dictionary, the correctly spelled word is searched by the algorithm. This search is performed on both word level and character level. This is to say, under the 'one error' assumption, all words that differ in length by one character or differ in the occurrence of characters by less than or equal to two bit positions are checked against the detected words using the spelling rules. All the remaining words are thus bypassed.

If a word in the dictionary is one character longer than the detected word, then the first character in the dictionary word that is different is discarded and the rest of the characters are shifted one bit position left. If the two words match, then the word in the dictionary is reported to be the correctly spelled word by a single insertion. For example, the word *aple* is detected as a misspelled word and *apple* is a word in the dictionary and they differ by one character. The character 'p' is discarded from *apple* and the rest of the character are shifted left. *aple* is then compared with the detected word and a match is found. Therefore, *apple* is reported as the correct spelling for *aple*.

On the other hand, if the word in the dictionary is one character shorter, then the first character in the detected word that is different from the matching character in the dictionary word is considered to be incorrect. Thus, that particular character in the detected word is discarded and the rest of the characters in the misspelled word are shifted one bit position left. If there is a match for the new misspelled word and the dictionary entry, the word in the dictionary is reported to be the correctly spelled word by a single deletion. For example, the word *applle* is detected as a misspelled word and *apple* is a word in the dictionary and they differ by one character. The character 'l' is discarded from *applle* and the rest of the character are shifted left. *apple* is then compared with the word in the dictionary and a match is found. Therefore, *apple* is reported as the correct spelling for *applle*.

If the lengths of the word in the dictionary and the misspelled word are the same, but they differ by one character position, then the dictionary entry is reported as a candidate correction as they differ by a single substitution. For example, the word *aplle* is detected as a misspelled word and *apple* is a word in the dictionary and they differ by one character. The first 'l' is replaced by 'p'. The resultant word is then compared with the dictionary word and a match is found. Therefore, *apple* is reported as the correctly spelled word for *aplle*.

If the lengths of the word in the dictionary and the misspelled word are the same, but they differ in two adjacent positions, the characters are proposed to be swopped. If the two words are the same, there is a match by a single transposition. For example, the word *aplpe* is detected as a misspelled word and *apple* is a word in the dictionary and they differ by one character. The positions of the characters 'l' and 'p' are swopped. The resultant word is then compared with the dictionary word and a match is found. Therefore, *apple* is reported as the correctly spelled word for *aplpe*.

Moreover, if there are several words within one edit distance of the misspelled word, the first word appears in the dictionary according to alphabetical order will always be selected.

Minimum edit distance algorithms generally require d comparisons between the misspelled word and the chosen dictionary, where d is the number of dictionary entries. To minimize the search time, some shortcuts are devised. For instance, the so-called reverse minimum edit distance technique was used in CORRECT by Kernighan et al [KCG90]. In reverse minimum edit distance, every possible single-error permutation of the misspelled word is generated and then it is checked against a dictionary to see if any permutation can make up valid words. This forms a candidate correction set. A total of $53n + 25$ strings are checked against the dictionary if a misspelled string has length n and the size of an alphabet is 26. This number is derived from $26(n + 1)$ possible insertions, n possible deletions, $25n$ possible substitutions, and $n - 1$ possible transpositions. CORRECT will be discussed in more detail in Section 4.3.

The use of tries is another means of improving search time [JT77]. A trie is used to store each dictionary entry, character-by-character. The lookup checks for insertion, deletion, substitution, and transposition operations in a selected branch of the trie, which is a subset of the database. The SPROOF algorithm is an example of a spelling correction algorithm that made use of minimum edit distance and stored a dictionary in a trie. Further details of the SPROOF algorithm can be found in [DM81]. The underlying spell correcting function in GROPE works in a similar manner [Tay81] as in SPROOF.

Some algorithms involving minimum edit distance assign non-integer values to specific spelling transformations based on estimates of phonetic similarities or keyboard positions. An instance can be found in [Ver88]. Other more recent researches involving minimum edit distance can be found in [BM00], [GFG03], ASPELL [Atk04], JAZZY [Whi04], and the AURA system [HA02, HA03].

Garfinkel et al described an interactive spell corrector in [GFG03]. The edit distance between the misspelled word and the candidate correction was used as a model of closeness. The commonness of the candidate correction was also considered so that words that are more common are more likely to be returned as results. Lastly, a binary variable was used to indicate whether the misspelled word and the candidate correction share the same first character under the assumption that first characters are less likely to be incorrect. Each of these sub-arguments was weighted using (edit distance -1) and they were added up to obtain a score, indicated by $g(w, m)$, where w is the candidate correction and m is the misspelled word. The smaller the value of g is, the more likely it is that w is the correct candidate.

The algorithm behind JAZZY [Whi04] is similar to the ASPELL algorithm which will be discussed in Section 4.4. The FSA package by Daciuk [Dac98] also made use of the so-called cut-off edit distance, which is the minimum edit distance between an initial substring of the misspelled word and any initial substring of the correct word. This will be discussed in details in Section 4.6.

Minimum edit distance is by far the most widely used spell correcting technique and this claim will become evident as we investigate the various spell correcting algorithms in the subsequent chapter.

3.4.2 Similarity key techniques

The essence of similarity key techniques is the mapping of every word into a key. The mapping is chosen so that similarly spelled words will either have similar or identical keys. When a key is computed for a misspelled word, it will provide a pointer to all similarly spelled words in a dictionary, and these dictionary entries will be returned as candidate corrections. This is to say, all words in a dictionary having similar key values compared to the key of the current misspelled word, will be returned as possible correct words. Due to the fact that it is not necessary to compare the misspelled word with every dictionary entry, similarity key techniques are fast.

Similarity key techniques are based on transforming words into similarity keys that reflect the relations between the characters of the words, such as positional similarity, material similarity, and ordinal similarity [Fau64].

- *Positional similarity.* As indicated by its name, it refers to the extent to which the matching characters in two strings are in the same position. It generally appears in an OCR text and literary comparison and is said to be too restricted to be used on its own for spelling correction [Pet80, AFW83].

- *Material similarity.* This refers to the extent to which two strings consist of exactly the same characters but in different order. Correlation coefficients between the two strings (i.e. the misspelled word and a word that consists of the exact same characters but in different order) have been used as a measure of material similarity [Sid79]. Material similarity is seen as not precise enough for the spelling correction task as all anagrams[8] are materially similar. For example, the word *angle* is an anagram of the word *glean* and they are materially similar.

- *Ordinal similarity.* Similar to position similarity, ordinal similarity indicates the extend to which the matching characters in two strings are in the same order. The SOUNDEX system which will be discussed shortly is an example of ordinal similarity. Other examples of making use of ordinal similarity measures include the character-node trie approach in [JT77] and the similarity key technique used in the SPEEDCOP system [PZ84].

Similarity key techniques can be found in the SOUNDEX system [Dav62, OR18]. The SOUNDEX system was devised to solve the problem of phonetic errors. It took a word in English and produced a presentation consisting of digits to preserve the salient features of the phonetic pronunciation of a word. A misspelled word was also mapped into a key consisting of its first character followed by a sequence of digits. Digits were assigned according to the following pre-defined rules:

A,E,I,O,U,H,W,Y $\rightarrow$ 0;
B,F,P,V $\rightarrow$ 1;
C,G,J,K,Q,S,X,Z $\rightarrow$ 2;

[8]An anagram is a word, phrase, or sentence that formed from another by reshuffling or repositioning its characters.

D,T → 3;
L → 4;
M,N → 5; and
R → 6.

Zeros are eliminated and repetition of characters are collapsed. For instance, the word *book* will be transformed to B002 which will become B2 and the word *bush* will be transformed to B020 which will also become B2. Although *book* and *bush* are neither equivalent nor similar, they are mapped to the same key, B2. This SOUNDEX system merely serves as an illustration here. It exemplifies the notion behind a similarity key in a simplified manner, albeit far from perfect. For more details on the SOUNDEX system, please consult [JM00, Par03].

Another algorithm, Metaphone, by Philips [Phi90], which works in a similar manner to SOUNDEX, was designed to respond to the lack of accuracy of the SOUNDEX system. The Metaphone algorithm transformed words into phonetic codes based on properties in [Phi90, Phi00]. It however worked on a character-by-character scheme. Double Metaphone [Phi00] is the second generation of the Metaphone algorithm. It started by analysing single consonants according to a set of rules for grouping consonants and then groups of letters by mapping them to Metaphone codes. ASPELL is an algorithm that combines Double Metaphone together with edit distance. We will be looking more closely at ASPELL in Section 4.4.

As mentioned earlier, another example of involving similarity key techniques for spelling correction is the SPEEDCOP system by Pollock and Zamora [PZ84]. The similarity key technique in SPEEDCOP was specifically used for correcting single-error misspellings. It will be discussed in details in Section 4.5.

A variant of the similarity key approach is an algorithm that computes a fuzzy similarity measure. It is called token reconstruction [Boc91]. Token reconstruction returned a similarity measure between a misspelled word and a word in the dictionary. It was computed by taking the average of four empirically weighted indices that measure the longest matching substring from both ends of the two words. The dictionary was partitioned into buckets containing strings that have the same first character as well as the same length. The search began by checking for words with small differences in length and then checking for words that start with other characters (i.e. different first characters) in the misspelled words.

3.4.3 Rule-based techniques

Rule-based techniques involve algorithms that attempt to represent knowledge of common spelling error patterns, for transforming misspelled words into correct ones. The knowledge is presented as rules. These rules can contain general morphological information (such as rules to transform a verb into an adjective by adding '-ing' at the end of the verb), lengths of the misspelled words and more. The candidate suggestions are generated by applying all applicable rules to a misspelled word and retaining every valid word in the dictionary that results. Ranking on the suggested words is based on a predefined estimate of the probability of the occurrence of the error that the particular rule corrected. It is completely independent of any grammar or parsing formulation. It can be a mere lexical lookup routine.

The general spell correcting application designed by Yannakoudakis and Fawthrop [YF83a, YF83b] is an example of using rule-based techniques. This application made use of a dictionary, which was partitioned into many subsets according to the first characters and word lengths. This decision was based on the fact that their rules contained knowledge of the probable length of the correct word, which was determined based on the misspelled word. To generate candidate suggestions, specific dictionary partitions containing words that differ from the misspelled word by one or two errors were searched under the condition that they comply with any of the rules. When multiple candidates were found, these candidate corrections were ranked according to predefined estimates of the probabilities of occurrence of the rules.

Another specialised knowledged-based spelling correction system was designed by Means [Mea88]. The rules in this case are a set of morphological information, such as doubling a final consonant before adding the suffix '-ing'. They are checked for common inflection violation. The system then consults a set of abbreviation expansion rules to determine if the misspelling could be expanded to a word in the lexicon. If this step fails, it tries all single-error transformations (i.e. insertions, deletions, substitutions, and transpositions) of the misspelled word.

More recently, an evidence of rule-based techniques being incorporated in a (context-independent) spelling correction system can be found in the phonetic module of AURA [HA03]. This algorithm was designed to represent phonetic spelling error patterns. AURA will be described in details in Section 4.8. Rule-based techniques are also seen to be incorporated into spelling correctors that perform morphological analysis for context-dependent spelling correction [BK03].

3.4.4 n-gram-based techniques

n-gram analysis was described mathematically in Section 2.3.2 and the concept of involving n-gram analysis in spell checking was discussed in Section 3.3.2. The nature of n-gram analysis will thus not be repeated in this section. We will be looking at the use of n-grams specifically for the spell correcting task [KST92, KST94]. n-grams have been used as a basis or in combination with other aforementioned techniques discussed in this chapter in order to achieve the task of spelling correction. Spell correctors employing n-gram-based techniques follow three processes: error detection; candidate suggestion; and rank similarity.

Rank Detection

Firstly, n-grams have been widely used in correcting misspellings in an OCR text as mentioned in Section 3.3.2 in order to capture the lexical syntax of a dictionary and suggest valid corrections. An example can be illustrated by the technique proposed by Riseman and Hanson [RH74], which incorporated n-grams in their OCR correction. A dictionary was partitioned into subsets according to word lengths. Each subset had positional binary

n-gram[9] matrices and these matrices captured the structure of strings in the dictionary. Each output word from the OCR device could be checked for errors by verifying if all its n-grams have value 1.

- If at least one binary n-gram of a word has value 0, it is indicated to have a single error.
- If more than one binary n-gram of a word have value 0, the positions of the errors are then indicated by a matrix index that is shared by the n-grams with 0 value. The rows or columns in the matrices specified by the common index of the erroneous n-grams are then logically intersected in order to find possible suggestions.
- If the result of the intersection indicates that only one n-gram has value 1, a candidate correction is found.
- If more than one n-gram suggestion are found, the checked word is rejected.

The advantage of this technique is that it prevents an exhaustive dictionary search. However, it runs a risk of resulting in a non-word as a correction. This technique only handles substitution errors. Ullmann [Ull77] proposed a technique in which binary n-grams are processed in parallel.

Candidate Suggestion

The second spell correcting procedure is to suggest candidate corrections. n-grams have often been used as access keys into a dictionary for locating possible suggestions and as lexical features for computing similarity measures. This will be explained by using the technique developed by Angell et al [AFW83] as an example. The number of the non-positional binary trigrams that occurred in both a misspelled word and a dictionary word were computed. Non-positional n-grams were referred to as n-gram arrays that did not indicate the positions of the n-grams within a word.

The lexical features in this case were trigrams. The similarity measure was then computed by a function called Dice coefficient[10] which was

$$D(n_m, n_d) = 2(c/(n_m + n_d)),$$

where

- c is the number of shared/common trigrams for both the misspelled word and the word in the dictionary,
- n_m is the length of the misspelled word, and

[9]In a binary n-gram, each n-gram in a word is assigned the binary value 1 or 0, based on its occurrence in a dictionary. For example, the word *and* has bigrams ‘an’ and ‘nd’. Matching these two bigrams with bigrams of words in a dictionary, it is very likely that both bigrams exist. Thus, each bigram has a binary value 1.

[10]Dice coefficient is a correlation coefficient for discrete events.

- n_d is the length of the dictionary word.

Note that n_m and n_d can be interchanged. Furthermore, the trigrams of the misspelled word were used as access keys. These trigrams were used to retrieve words in the dictionary that have at least one trigram that is in common with the misspelled word. The similarity measure thus only needs to be computed for this subset. The drawback of this particular technique is that any words shorter than three characters cannot be accurately detected because one single error can leave no valid trigrams intact. For misspellings containing more than one error, the function for similarity measure was changed to

$$D(n_m, n_d) = 2(c/max(n_m, n_d)),$$

where $max(n_m, n_d)$ is the highest probability of the common trigrams.

Rank Similarity

The third spell correcting procedure, similarity ranking, is not executed in any of the example schemes discussed to date. However, n-gram based techniques can also be used to find and rank candidate suggestions. Both misspelled and correct words are represented as vectors of lexical features (with unigrams, bigrams, and trigrams as possible candidates for the features of the lexical space) to which conventional vector distance measures can be applied. The measures then form the basis for ranking candidate suggestions.

These techniques first position each dictionary word at some point in an n-dimensional lexical-feature space in where words are represented as n-gram vectors. The dimension of a lexical-feature space n can be very large. For example, if the lexicon consists of 10,000 words, one may use trigrams (sequences of three consecutive letters) as the feature, then $n = 10^12$ $(10^{4\times3})$. A misspelling is then projected into that specific space. The proximity of the misspelling and its nearest possible candidates in this lexical space is measured by hamming distance, dot products or cosine distances.

- The Hamming distance is a metric[11] on the vector space of the words of the same length. The hamming distance for the misspelling and a dictionary word is the number of occurrence in which the two words differ, i.e. have different characters. The smaller the hamming distance, the more similar the two vectors are.
- The dot product metric is an operation which maps two vectors into scalars and returns a real number. Each of these lexical feature vectors is associated with a corresponding coefficient value that represents the frequency of its occurrence in a particular dictionary. Two vectors are said to be similar if their dot product is small.
- The cosine distance metric between two vectors are equal to the dot product of the two vectors divided by the individual norms of the vectors. However, if the vectors are already normalised, the cosine distance simply becomes the dot product of the vectors. Two vectors are said to be similar if their cosine distance is small.

[11]A metric is a function that is symmetric, satisfies the triangle inequality and the reflexive distance is 0.

Correlation Matrix Memory techniques is an example of making use of vector distance measures based on n-gram vectors (i.e. representation of words). A dictionary of r words is represented as $n \times r$ matrix of n-dimensional lexical-feature vectors, where unigrams, bigrams, and trigrams are features of the lexical space and in where words and misspelled words are represented by sparse vectors. For correction, the n-dimensional lexical-feature vector of the misspelled words is multiplied by the $n \times r$-dimensional matrix. This yields an r-dimensional vector in which the i^{th} element represents a measure of correspondence between the i^{th} dictionary entry and the misspelled word. The element with the highest value is deemed as the most strongly correlated entry. Thus, it is indicated as the candidate correction. Correlation Matrix Memory techniques are relevant, and details can be found in [CVBW92].

The Generalized Inverse matrix is a technique used for transforming lexical-feature spaces. The goal is to minimize the interference that occurs among similar dictionary words. The technique is based on finding a so-called minimum least squares error inverse of the dictionary matrix. Details can be found in [CRW91]. The drawback of a Generalized Inverse matrix is that when the number of dictionary entries approaches the dimensionality of the feature space, the matrix saturates and the correction accuracy takes a significant decline [CFV91]. Thus, it has been concluded that simple Correlation Matrix Memory techniques are more effective and efficient in achieving the spelling correction task.

Another technique used for transforming lexical-feature spaces is Singular Value Decomposition. It is used to break down a dictionary matrix into a product of three matrices in order to identify and rank the most important factors. These factors measure the relevant similarity distance in the lexical space. In regard to spelling correction: words are represented by unigram and bigram vectors in a spelling correction matrix. This matrix is broken down into three factor matrices. A misspelled word is corrected by first summing the vectors for each of the individual-letter n-grams in the misspelled word. The sum vector is then multiplied by the singular-value matrix of weights. The location of the misspelling in the n-dimensional feature space is determined by the resultant vector. As mentioned earlier, hamming distance, dot product, or cosine distance can be used to measure the distances between the vectors for each of the suggested words and the vector for the misspelling and locate and rank the nearest correct words. Details of Singular Value Decomposition can be found in [Kuk90].

Most of the techniques that use vector distance measures based on representations of words as n-gram vectors collapse the three spelling correction procedures. They combine error detection, suggestion, and similarity ranking into one process.

n-gram-based techniques appear to be the approach of choice to construct language models for non-Latin languages, such as Bangla and Chinese [LW02].

3.4.5 Probabilistic techniques

n-gram-based techniques often act as a prelude to probabilistic techniques. Probabilistic techniques were originally used for text recognition.[12] Two types of probabilities are

[12]Text recognition is also known as optical character recognition.

relevant: transition or Markov probabilities, and confusion or error probabilities. As will be seen, transition probabilities are language dependent, whereas confusion probabilities are source dependent.

Transition or Markov probabilities determine the likelihood[13] that a certain given letter will be followed by another given letter in a given language. These probabilities can be determined by collecting n-gram frequency information from a large corpus. They are based on simplifying the assumption that the language is a Markov source—i.e. that the probability of a transition is independent of prior transitions.

Confusion or error probabilities determine the probabilities that a certain letter substitutes another given letter, given that an error has been made. n-grams are explored in all these techniques. Confusion probabilities can be determined by feeding a sample text into the OCR device and tabulating error statistics. An alternative way to estimate the probabilities is that an OCR device can generate a 26-element vector containing probability for each letter of the alphabet at the time a character is recognized [BB59].

The difference between the two probabilities is that transition probabilities are language dependent whereas confusion probabilities are source dependent. Transition or Markov probabilities are based on the assumption that the language is a Markov source and that the corpus analysed is often domain-specific. On the other hand, confusion probabilities are source dependent because different OCR devices have different techniques and features, such as font types, and each device will generate a unique confusion probability distribution. Confusion probabilities can also be based on human errors. In this case, they are estimated by extracting error statistics from samples of a typed text. For the probabilistic approach for the spelling correction task, candidate corrections are generally ranked using a Bayesian or noisy channel algorithm [KCG90, JM00].

Bledsoe and Browning [BB59] were the first to make use of the probabilities in text recognition techniques. In the first phase, they tackled individual-character recognition. This entails that a 26-element vector of probabilities is generated, for each character in an input word. This is to say if a word was made up of 5 characters, then a matrix of 5×26 was generated. A whole-word recognition phase then followed. It entails the individual letters whose overall probabilities maximize the probability of producing a valid word from the dictionary were chosen with the help of a dictionary. Bayes' rule was used to compute $P(w|m)$, the so-called posterior probability that a valid word w should be used for the OCR-determined word (i.e. the misspelled word) m. Details may be found in [BB59].

There have been several attempts to perform spelling correction based on transition and confusion probabilities [HS82]. A dynamic programming technique (mentioned in Section 3.4.1), the Viterbi algorithm [For73], is efficient and widely used to combine transition and confusion probabilities for spell correcting. In the Viterbi algorithm, a directed graph was used to capture both the structure of a dictionary and the channel characteristics of a device, such as an OCR device. Transition probabilities were deduced from the structure of a dictionary whereas confusion probabilities were deduced from the channel characteristics of a device.

[13] In this dissertation, likelihood is considered interchangeable with the word of probability.

The structure of the graph is as follows: Word boundary markers, such as blank spaces, are represented by a starting and an ending nodes; the likelihood estimate for each individual letter is represented by an intermediate node; and the transition probability between two letters is represented by labels on the edges of the graph. This graph is efficiently traversed to find the sequence of characters with the highest probability given the likelihood estimates for an OCR output and the transition probabilities of the language. There exist other modifications of the original Viterbi algorithm [ST79b]. The drawback of these techniques that make use of the Viterbi algorithm is that the string with the highest probability is not always valid.

Combining the use of probabilistic techniques with dictionary lookup techniques can achieve better error correction results. As we mentioned earlier, the string with the highest probability is not always a valid word. By looking up this particular string in the dictionary, it ensures the validity of the string suggested.

Shinghal and Toussaint [ST79a] devised a technique called predictor-corrector method which made use of a modified Viterbi algorithm to recognise the input string and then performed a binary search in the dictionary for the word with a matching value. Sinha and Prasada [SP88] devised a technique that integrated probabilities, dictionary lookup, and a variety of heuristics (i.e. rules that spelling errors tend to follow). The technique started by constructing a partial dictionary, which consisted of a list of 10,000 most frequently appeared words from the Brown Corpus and valid words obtained from performing single-character substitutions on these 10,000 words. The invalid words were corrected by using the confusion probabilities (i.e. how often a given character is substituted for another character). Then a modified Viterbi algorithm was used to estimate the correction with the highest probabilities for words that were not present in the dictionary. The dictionary was stored as a trie. A variety of rules that spelling errors tend to follow were used to rank confusion candidates.

The layered Hidden Markov Model (HMM) technique has been used in the Token Passing framework [YRT89] to guide the spelling error correction process [Ing96]. The HMMs were arranged in multiple layers, where the HMMs in each layer were responsible for different aspects of the processing of the input text. The tokenizer made use of the linguistic information extracted from the training corpus.

Kernighan et al [KCG90] and Church and Gale [CG91] devised CORRECT, which aims at correcting single-error misspellings. CORRECT made use of a reverse minimum edit distance technique (discussed in Section 3.4.1) to generate a set of single-error candidate correct spellings. A specific error within each correct word was identified. A Bayesian formula was used to rank the candidate suggestions. CORRECT will be discussed in detail in Section 4.3.

Brill and Moore proposed the use of noisy channel model or Bayes' rule for spell correcting [BM00]. This approach allows for more than one spelling error in each misspelled word. Each misspelled word and each suggested word were segmented into all possible partitions. The probability of the dictionary word being a possible correction was then computed. Dynamic programming was employed to calculate the generic edit distance between two strings. The dictionary was stored in a trie with edges indicating a vector which contained the weights of computing the distance from string to their corresponding prefixes. Details can be found in [BM00].

Probabilistic techniques often make use of language models constructed from n-grams for spell correcting. Bayes' rule on the noisy channel model is still a preferred option according to Brill and Moore [BM00]. Garfinkel et al [GFG03] made use of a probabilistic function for their interactive spell corrector. Although probabilistic techniques only started being incorporated in spell correcting in the nineties, have become widely used.

3.4.6 Neural networks

Neural networks are potential candidates for spelling correction due to their ability to do associative recall based on incomplete or noisy input. Neural nets have the ability to adapt to the specific error patterns of a certain user's domain because they can be trained on actual spelling errors. Hence, the correction accuracy for that domain is potentially improved.

The back-propagation algorithm is widely used for training neural networks [CV89]. A back-propagation network generally consists of three layers of nodes: an input layer, a hidden layer, and an output layer. The nodes within a given layer are not directly connected. However, each node in the input layer is connected to every node in the hidden layer by a weighted link and each node in the hidden layer follows the same fashion and is connected to every node in the output layer by a weighted link. The activities on input and output nodes of the network indicate input and output information respectively (i.e. when a node is turned on, a 1 is indicated; when a node is turned off, a 0 is indicated).

A back-propagation network starts by placing a pattern of activity on the input nodes, the activity is sent through the weighted links to the hidden nodes, and a hidden pattern is then computed. This latter activity is sent to the output nodes where an output pattern of activity for the system is then computed. The total activity reaching a node is the weighted sum of the activities of each of the nodes leading up to it. The algorithm provides a means of finding a set of weights, which represent relation strengths between nodes for the network that allows the network to produce either valid or similar output pattern for each input pattern. During the so-called training phase, the difference between the actual output pattern and the desired output pattern is computed for each node in the output layer. This difference is used to fine-tune each weight by an amount inversely proportional to the error. This procedure is repeated through all the examples in the training set until the weights converge. The result is a set of weights that produces a nearly correct output pattern for each input pattern in the training set and also for similar input patterns that were not in the training set.

Using a back-propagation network in a spelling correction application, a misspelled word which is represented as a binary n-gram vector (Section 3.4.4) serves as an input pattern to the network. The output pattern is the activation of some subset of r output nodes, where r is the number of words in the dictionary. For input representing some misspelled word, the aim is to ensure that the output nodes corresponding to alternative corrected word spellings are turned on. Neural networks, like probabilistic techniques, are extensively employed in OCR correctors [MBLD92].

Deffner et al [DEG90] implemented a neural network spelling corrector, which is part of a natural language interface to a database system.

Feature vectors used to represent dictionary entries and misspellings contain phonetic features (i.e. how a word is pronounced), syntactic features (e.g. noun, verbs), semantic features (i.e. features used to express the existence or non-existence of semantic properties, such as shape, colour), and letter n-grams. A similarity measure between a possible misspelling or an incomplete input string and each element of a restricted subset of the dictionary is computed by a hamming distance metric.

AURA was implemented by Hodge and Austin [HA02, HA03] in their recent research. It is a modular binary neural network architecture that made use of Correlation Matrix Memories (CMM) [Koh88] and a supervised learning rule which is somewhat similar to a hash function in order to map inputs to outputs. Two approaches were employed to resolve different types of spelling errors. AURA will be explained in more details in Section 4.8.

Although neural networks are the main technique behind AURA's spell correcting function, they have not been adopted widely by other spell correcting systems.

3.5 Other Related Issues

There are several other issues that are closely related to spell checking and correcting processes: dictionary partitioning schemes, dictionary or lexicon construction, and word boundary issues.

- Dictionary partitioning schemes were motivated by memory constraints in many early spelling checkers. Peterson [Pet80] suggested partitioning a dictionary into three levels to avoid this problem. The first level which consists of 50% of dictionary entries was stored in the cache memory. The second level which consists of another 45% of dictionary entries was stored in the main (or primary) memory. Lastly, the third level which consists of the remaining 5% of dictionary entries was stored in the secondary memory. The first level consists of the most frequently used words; the second level consists of some domain-specific terminologies; and the third level consists of less frequently used words. The memory constraints are not as much of an issue these days. However, dictionary partitioning can make a slight difference in terms of performance. Dictionary partitioning can also be done by dividing a dictionary into many sub-dictionaries according to word lengths or first letters. Spell correcting algorithms will search specific dictionary partitions for words for candidate corrections. Generally, several dictionaries and/or a few lexicons are used in a spell checker and corrector. When several lexical lists are used, they are treated in the same fashion as the dictionaries.

- Data structures for storing dictionary entries can appear in various forms. The simplest case is to store the entries in a sequential list, where words are stored in alphabetical order. This list can be indexed either by lengths or by collating sequence in order to reduce search time. The other data structures for storing dictionary entries are partial or complete hashing and tree structures, such as binary search tree or a trie, either at the word or character level. These structures are suitable for storage in the primary memory. However, either a B-tree or disk hash table is more suitable if the dictionary is to be stored in the secondary memory. Finite-state automata have also been a favourable candidate for dictionary implementation. Using finite-state automata, it allows random access to the data stored and the lexical functions, such as lexical classification (i.e. parts of speech), inflections, and conjugations, and even gender information and number marking in some languages, such as Portuguese, at a negligible cost. [KLS98] is an instance of employing minimized acyclic finite-state automata.

- Word boundaries in most spelling error detection and correction techniques are often defined by inter-word separation, such as spaces and punctuation marks. However, it is largely dependant on the language under consideration. For example, word boundaries are defined by word dividers for Amharic. In most Asian languages, such as Chinese [SGSC96], Thai, Japanese, and ancient languages, such as Sanskrit, word boundaries are not obvious. For example, in Chinese, there is no space between words to act as word boundary delimiter. Thus, dictionaries and word frequencies are heavily relied on to solve this problem. We will be looking into the techniques used in Chinese text proofreading in more details in Section 4.10.

  Furthermore, even in writing systems that make use of inter-word separation, it is not always easy to determine word boundaries when it comes to compound words, run-on words, and split words. The SPEEDCOP [ZPZ81] spell corrector discussed in Section 4.5 checked for run-on errors involving function words as a final subroutine of spell checking. The complexity of handling errors due to word boundary infractions remains one of the unsolved problems in research of spell checking and correction.

3.6 Conclusion

In recent research, it is established that most of the spell checking techniques discussed in Section 3.3 are mainly employed to perform exact string matching. Spell correcting techniques are employed to suggest an alternative or a valid word to a misspelled or an invalid word. From our brief description of the actual algorithms that employed the aforementioned spell correcting algorithms, it seems that the current trend of implementation is towards hybrid approaches, i.e. combining several of the aforementioned techniques. It would therefore seem that each technique on its own is not sufficient enough to achieve high accuracy. Furthermore, an algorithmic solution to the spell checking and correcting tasks often consists of a combination of techniques described in this chapter. More evidence of this claim will be provided in Chapter 4.

The current trend for the use of spell correcting techniques also indicates that minimum edit distance is by far the most widely studied and used spell correcting technique. Similarity key techniques have been seen in several spell checkers and correctors, such as SPEEDCOP, ASPELL, and JAZZY. A more recent evidence of involvement of rule-based techniques is found in the phonetic module inside AURA. Rule-based techniques are also incorporated into spelling correctors that perform morphological analysis for context-dependent spelling correction. n-gram-based techniques appear to be the choice to construct language models for languages, such as Bangla, Chinese [LW02], and Finnish, that possess more complicated language structures. Probabilistic techniques often make use of n-gram language models in spell correcting. They have become widely used to date. Neural networks as the main technique behind AURA's [HA02, HA03] spell correcting function have not been adopted widely by other spell correcting systems. The above claims will be supported by our investigation on the various spell checking and correcting algorithms in the following chapter.

In the subsequent chapter, the techniques described in this chapter can be seen used in the actual spell checking and correcting packages.

Chapter 4

Spell Checking and Correcting Algorithms

4.1 Introduction

This chapter provides a number of representative algorithms that incorporate the techniques described in Chapter 3 for finding spelling errors and suggesting corrections. This chapter is to show in a structured fashion the connection between the mathematical model provided in Chapter 2, the techniques described in Chapter 3, and the various algorithms discussed in this chapter.

An algorithmic solution for the spell checking and correcting tasks often requires a combination of techniques described in Chapter 3, as one single technique on its own might be too inefficient to produce the optimal results in terms of accuracy, running time and storage space. This claim will become more obvious as we progress further into this present chapter. Spell checkers and correctors these days rely heavily on approximate string matching algorithms in order to find correctly spelled words with similar spellings to an erroneous word. We are now going to describe the algorithms used by the chosen spell checkers and correctors in detail.

Throughout this dissertation, some of the algorithms are described in details and several algorithms are expressed in a variant of Dijkstra's Guarded Command Language (GCL) [Dij76]. GCL is used to standardise the presentation of various algorithms. The text description and the GCL presentation for various algorithms make it possible to compare these algorithms in a homogenous context in Chapter 5.

Taking into consideration the spell checking and correcting techniques described in Chapter 3, the algorithms discussed are:

1. Unix® SPELL, where hashing was employed;

2. CORRECT, which employed a probabilistic technique (reverse minimum edit distance);

3. ASPELL, where the Metaphone algorithm and edit distance (discussed in Section 3.4.1) were employed;

4. SPEEDCOP, where similarity keys and error reversal technique were employed;
5. The FSA package, where finite-state automata are involved in lexicon construction and/or morphological analysis and cut-off edit distances were employed;
6. AGREP, in which the bitap algorithm employed pattern matching;
7. AURA which made use of neural nets with data training performed using hamming distance, shifting n-gram approach and phonetic codes;
8. DAC Northern Sotho (DNS) spell checker, which is embedded in Microsoft Word;
9. Chinese text proofreading tool—CInsunSpell, which employed a combination of techniques such as n-gram and probabilistic approaches and edit distance.

Before we begin discussing the mentioned algorithms in detail, it is necessary to bear in mind that the spell checking and correcting procedures in these algorithms are extensions and implementations of the $Misses_L$ and $Suggest_L$ functions (Functions 2.3.2 and 2.3.3, respectively) described in Chapter 2. These functions are generic in the sense that they are given abstract implementation only, thereby specifying the semantics but not the precise implementation. Some implementations will be given in the following sections.

4.2 SPELL

The Unix® spelling checker SPELL [Rit06] was originally written by S. C. Johnson in 1979 and was subsequently re-written several times. One of McIlroy's [Ben00, McI82] versions of SPELL focused on data compression and introduced a one-byte affixability indicator with each word, which provided much better heuristics for deriving words. The advantages of McIlroy's versions over Johnson's are better word list and reduced run time. The SPELL programme requires very little memory usage as it was designed for a machine with only a 64KB address space in its main memory.

Below, the broad outline of Johnson's algorithm is first explained. This is then followed by a brief description of McIlroy's data-compression-based algorithm. Lastly, GCL code is given to explain the hashing that was used in this case.

The original SPELL programme written by Johnson works in the following manner:

It takes an input file and breaks the file content into words. These words are sorted into alphabetical order and duplicates are discarded. The sorted words are then compared to a dictionary. Words that are not found in the dictionary are flagged and reported. A Bloom filter [Blo70, Mit02] was used in SPELL. The shortfalls of this approach are its slowness and limited coverage of vocabularies.

As Bentley [Ben00] pointed out, in McIlroy's version of SPELL, a superior word list was used instead of a simple desk dictionary. This word list was built and compiled from various sources, such as the Brown Corpus and Webster's New International Dictionary of the English Language. An affix analysis was employed to reduce the size of the dictionary.

This word list contains 75,000 words. The SPELL programme performs affix analysis to the word list in order to reduce its size. The goal of affix analysis is to reduce an inflected word down to its word stem. For instance, the word *misspelled* will be stripped of its affixes (i.e. its prefix and suffix—see Section 3.3.1) according to built-in tables that contain forty prefix rules and thirty suffix rules [McI82]. Thus, the prefix of the word *misspelled* would be identified as 'mis-' and its suffix as '-ed'. After stripping off both 'mis-' and '-ed', the word stem 'spell' is inserted into a new word list called the *stop list*. The affix analysis reduces the original 75,000-word list to a 30,000-word list.

The lookup process starts by affix stripping the word being searched for, and the word itself and the stem are looked up in the stop list. If a match is found, a stop flag is attached to it. The word is also looked up in the original word list. If the word has a stop flag, it is assumed to be correct if it also appears in the original word list. Words that neither occur among, nor are derivable from (by applying affix analysis) words in the word list are output as misspellings.

A word in the word list is a sequence of alphabetical characters. Hashing schemes are then used in SPELL to represent 30,000 words in 27 bits each.

The general spell checking strategy in McIlroy's version did not change—Johnson's version already used hashing (albeit in the form of a Bloom filter, rather than per-word hash as in McIlroy's version). The scheme in SPELL uses multiple hash functions, each of which hashes the word w to a 27-bit integer, which is then used to index a bit vector B of length 2^{27} (=134,217,728). With only roughly 30,000 words to be represented in such a table, the chances of a hash collision is low—as will be discussed below.

Initially, each word w in the word list W is hashed using k independent hash functions $h_1, \cdots, h_k$—each of which is assumed to distribute over the space 2^{27} uniformly[1]. (The choice of k will be explained in the coming paragraphs.) The hash values $h_1(w), \cdots, h_k(w)$ are used to index into our bit vector B, setting the corresponding bits to 1, i.e. $B[h_1(w)] = B[h_2(w)] = \cdots = B[h_k(w)] = 1$. Note that we use the notation $X[i]$ and X_i interchangeably throughout this algorithm to indicate the element in the i^{th} position in some array X.

Here, we assume that there is a function that converts a string into an integer value. Consider an example: suppose B is 10 bit hash table (i.e. $n = 10$), the number of hash functions is 3, i.e. $k = 3$ and $h_1(w) = 2$, $h_2(w) = 4$, and $h_3(w) = 0$ for our words w to be inserted into B. In that case, we set bits B_0, B_2, B_4, as showing in the following table:

	$h_1(w)$	$h_2(w)$	$h_3(w)$					B					
Bit number:				0	1	2	3	4	5	6	7	8	9
Initially:				0	0	0	0	0	0	0	0	0	0
w inserted:	2	4	0	1	0	1	0	1	0	0	0	0	0

To look up a word, the table B is probed to see whether all hash-designated bits are set to 1. Thus, to look up the word w, $h_1(w), h_2(w), \cdots, h_k(w)$ are computed; if $B[h_1(w)] = B[h_2(w)] = \cdots = B[h_k(w)] = 1$, then the word w is indeed a correctly-spelled word. However, if any of the bits are set to 0, the word w is reported as a misspelled word.

[1]For this reason, the hash functions are also reduced modulo some prime(s) less than 2^{27}.

Although this approach reduces space dramatically, it results in a behaviour in which a misspelled string can go undetected because it happened to map into a hash address for a valid word. It can be shown that the probability that one of the k hash functions does not map a misspelled word into a valid word is given by the Poisson formula: $P = (1-1/n)^{km}$ where m is the number of entries in the table. Thus, $(1 - P)$ is the probability that one of the k hash functions does indeed map a misspelled word to a valid word. Furthermore, $(1 - P)^k$ is the probability that all the k hash functions map the misspelled word to a valid word. In the case of SPELL, the probability of an error is $2^{15}/2^{27}$, roughly the ratio of 1:4000 on average, if each entry of roughly 30,000 words is hashed into 27 bits. As explained by McIlroy [McI82], the most optimal value of k is thus determined by

$$k = \frac{n}{m} \log 2 \tag{4.1}$$

where n is the number of words in the word list W.

The hashing algorithm employed in SPELL is expressed in GCL below. Consider a hash table $B[0, n)$ (i.e. a bit array of n elements ranging from 0 to $n - 1$) for spelling checker that has n bits which are initially set 0, and a word list $W[0, r)$ that contains a set of r word. Let $w_j \in W$, $j = 0 \ldots (r - 1)$ be the j^{th} word in the word list. Assume that $n > r$. Each word w_j is hashed by k different hash functions, h_i, where $i = 1 \cdots k$. For each word w_j and each hash function h_p (where $p = 1 \cdots k$), the bit $B[h_p(w_j)]$ is set to 1. The same bit can be set to 1 more than once.

Algorithm 4.2.1 describes how the bit vector is initialised, and Algorithm 4.2.2 shows how the bit vector is checked to see if a candidate word is correctly spelled. The symbols in the algorithms have the following meanings:

- B is the bit vector in which the words are stored.
- W is the list of words to be stored,
- n is the length of the bit vector. In theory, and for the purposes of the discussion below, $n = 2^{27} = 134,217,728$. It practice, it is the first prime number less than this value.
- r is the number of words in the word list, i.e. $r = |W|$.
- $B[i]$ represents the i^{th} bit in the bit vector, for $i \in [0, n)$.
- w_j represents the j^{th} word in the word list, for $j \in [0, m)$.
- k is the number of hash functions used by the algorithms.
- h_p is the p^{th} hash function to be used. Each hash function maps a word (interpreted as an integer, rather than as a string) to an integer in the range $[0, n)$. The mapping takes place as follows: For $p \in [1, k]$, let z_p be the p^{th} prime number less than n. Then $h_p(w_j) = w_j mod z_p$.

- x is a word to be looked up in B.

Note that throughout this dissertation we use the notation $[0, x)$ to indicate the range of an array (i.e. $0, 1, \cdots, x-1$, where x is an integer variable).

The first loop of Algorithm 4.2.1 initializes the hash table, B to zeros. The next loop enters hashed information about each word list into the table. Essentially, it uses the k hash functions to compute k integers to serve as k indices into table B. At each such index of table B, the table's value is set to 1. A table V contains boolean values.

Algorithm 4.2.1(Hashing in SPELL)

```
proc $SpellHash(B, W, n)$
    $B :\, = 0^n$;
    for $w \in W \rightarrow$
        for $i \in [1, k] \rightarrow$
            $B[h_i(w)] = 1$
        rof
    rof
corp
```

To check if an affix-stripped word x is in W, we need to check whether all $B[h_p(x)]$ are set to 1 for $p = 1 \cdots k$. If all $h_p(x)$ bits are set to 1, we assume that x is in W and the algorithm returns true. If not, x is reported as a misspelling and false is returned.

Algorithm 4.2.2(Look up in SPELL)

```
func $SpellLookup(B, w) : boolean$
    for $i \in [1, k] \rightarrow$
        if $B[h_i(w)] = 0 \rightarrow$ return $false$
        $[\!]$ $B[h_i(w)] = 1 \rightarrow$ skip
        fi
    rof;
    return $true$
cnuf
```

McIlroy went on to compress the conventional hash as the storage space in earlier systems was constrained ($2^{15} \times 27$ bits into $2^{15} \times 16$-bit addressable memory in this case). A Huffman code was employed for storage compression [Huf52, Knu85].

GNU's SPELL is the open source reimplementation of Unix® SPELL. GNU's SPELL accepts as its arguments a list of files to read from. It prints each misspelled word on a line of its own.

The spell detecting algorithm found in the Unix SPELL programme [Rit06] remains unchanged from McIlroy's version. Note that SPELL only addresses the spell error detection problem.

4.3 CORRECT

CORRECT was briefly mentioned in Section 3.4.5. The CORRECT programme makes use of a combination of edit distance and the probabilistic approach. CORRECT was devised by Kernighan, Church and Gale [KCG90] and further research was carried out by Church and Gale [CG91]. There are two versions of CORRECT: with context and without context. In this dissertation, we focus on the version without taking context into consideration. The CORRECT programme combined a lookup in the word list and probabilistic techniques for correcting single-error misspellings.

CORRECT is designed to take a list of lower-case words rejected by the Unix® SPELL programme, generate candidate corrections for these misspelled words, and sort these candidates by probabilities. In CORRECT, it is assumed that a misspelled word is the result of a single error, or, more specifically, that the correct word is derivable from the misspelled word by a single spelling error transformation[2]. By a single spelling error transformation is meant a transformation that involves one of the following operations applied to the misspelled word to derive a correctly spelled word: insertion, deletion, substitution, and reversal. Kernighan et al [KCG90] defined the four operations as follows:

- *Insertion*: a misspelled word is the result of inserting an extra character in a correctly spelled word.
- *Deletion*: a misspelled word is the result of deleting a character from a correctly spelled word.

  The deletion table is a table where the entries are the misspelled words which are in turn the result of deletions from a correctly spelled word. The drawback of this table is that there is a cost in space as it has approximately a million entries which is about 10 times the number of entries in the dictionary.
- *Substitution*: one of the word's characters is replaced with another.
- *Reversal*: two adjoining characters in a word are swapped.

[2]This may look familiar: it is the reverse edit distance—see page 25.

Moreover, CORRECT assumes that such single spelling errors occur with particular probabilities. For instance, it may be more likely that 'ed' is derived from a transposition of 'de' rather than from a substitution of 'ad'. Note the definitions of insertion and deletion operations provided by Kernighan et al [KCG90] are inverse to the ones discussed in Section 3.4.1.

Note that deletions and insertions are inverse to one another: if a misspelled word is corrected by a deletion, then the misspelled word can be regarded as the result of an insertion into the correctly spelled word. Similarly, if a misspelled word is corrected by an insertion, then the misspelled word can be regarded as the result of a deletion in the correctly spelled word.

The four spelling error transformations can thus be expressed in the following functions which we choose to call *Ins*, *Del*, *Sub*, and *Rev*. Let m be a misspelled word and d be a correctly spelled word found in a dictionary. As stated in the previous section, $m_{[0,p)} = m_0, m_1, \cdots, m_{p-1}$.

The insertion operation can be seen as a predicate function whose value is determined by the following expression:

$$\begin{aligned} Ins(m,d) \quad &\triangleq \quad (|m| = |d| + 1) \wedge \exists p \in [0,|m|) \cdot \\ &\quad ((m_{[0,p)} = d_{[0,p)}) \wedge (m_p \neq d_p) \wedge (m_{[p+1,|m|)} = d_{[p,|d|)})) \end{aligned}$$

The *Ins* function can thus be implemented as follows:

Algorithm 4.3.1(Insertion)

```
func $Ins(m,d) : boolean$
    if $(|m| = |d| + 1) \rightarrow$
      $i, c := 0, 0;$
      do $((i < |d|) \wedge (m_i = d_i)) \rightarrow i := i + 1$ od;
      $\{(i = |d|) \vee (m_i \neq d_i)\}$
      if $(m_i \neq d_i) \rightarrow c := 1$
      $[\!]$ $(m_i = d_i) \rightarrow c := 0$
      fi;
      do $((i < |d|) \wedge (m_{i+1} = d_i)) \rightarrow i := i + 1$ od;
      $\{(i = |d|) \vee (m_{i+1} \neq d_i)\}$
      if $((m_{i+1} \neq d_i) \wedge (c = 1)) \rightarrow$ return $false$
      $[\!]$ $((m_{i+1} = d_i) \vee (c \neq 1) \rightarrow$ return $true$
            fi;
        ▯ $(|m| \neq |d| + 1) \rightarrow$ return $false$
        fi
    cnuf
```

The deletion, substitution, and reversal operations can be expressed as follows:

$$\begin{aligned}
Del(m,d) &\triangleq (|d| = |m|+1) \wedge \exists p \in [0,|m|) \cdot \\
&\quad (m_{[0,p)} = d_{[0,p)}) \wedge (m_p \neq d_p) \wedge (m_{[p,|m|)} = d_{[p+1,|d|)}) \\
Sub(m,d) &\triangleq (|m| = |d|) \wedge \exists p \in [0,|m|) \cdot \\
&\quad (m_{[0,p)} = d_{[0,p)}) \wedge (m_p \neq d_p) \wedge (m_{[p+1,|m|)} = d_{[p+1,|d|)}) \\
Rev(m,d) &\triangleq (|m| = |d|) \wedge \exists p \in [0,|m|-1) \cdot \\
&\quad (m_{[0,p)} = d_{[0,p)}) \wedge (m_p = d_{p+1}) \wedge (m_{p+1} = d_p) \wedge (m_{[p+2,|m|)} = d_{[p+2,|d|)})
\end{aligned}$$

The *Del*, *Sub*, and *Rev* functions can thus be similarly implemented as the *Ins* function. The notation $m_{[0,p)}$ refers to an array of characters in a misspelled word, where m_0 refers to the first character of the word m and m_{p-1} refers to the last character of m. Similarly, $d_{[0,p)}$ refers to an array of characters in a dictionary word, where d_0 refers to the first character of the word d and d_{p-1} refers to the last character of d.

In seeking candidate corrections that differ from the input (i.e. misspelled word) by a single spelling error transformation, CORRECT searches a particular word list. This list was composed from various sources, such as the word list in SPELL, the corpus collected from Associated Press (AP) newswire, two dictionaries, and one thesaurus for proposed correction.

A specific correction is identified by systematically applying all possible single-error transformations of the misspelled word and checking to see if the result can be found in the word list.

Assume a misspelled word is of length n and an English alphabet is of size 26. The number of strings that must be checked against the dictionary is $53n+25$, where $26(n+1)$ result from insertions, n from deletions, $25n$ from substitutions, and $n-1$ from transpositions.

The insertion operation requires n dictionary accesses to check if the n^{th} letter in the typographical error or the misspelled word has been inserted. The deletion operation requires many more dictionary accesses because for each deletion, 26 letters may have been deleted in $n+1$ positions. A pre-computed deletion table and hashing are used during candidate generation to minimize the number of dictionary accesses by means of only one-table lookup. This pre-computed table is also used for checking substitution and reversal. Heuristic hashing methods similar to the ones used in SPELL [McI82] are employed to store deletion tables such as those in Table 4.1. The deletion table is a table where the entries are the misspelled words which are in turn the result of deletions from a correctly-spelled word.

Key	Correction	Position
afte	r	4
aftr	e	3
afer	t	2
ater	f	1
fter	a	0

Table 4.1: An example of the deletion table for the word *after*.

The list of suggested words are ranked by probability scores using a noisy channel model and Bayes' rule [Win03]. The noisy channel model assumes that misspelled words are the result of noise being added during typing, i.e. typographic errors. The Bayesian combination states that the probability that w is a correct spelling, given that the typographical error m has occurred, is given by $P(m|w)P(w)$, where $P(w)$ is the estimated probability of the word w occurring in some text, based on the observation of some large corpus, and where $P(m|w)$ is the conditional probability that the misspelling m was produced when the word w was intended, w being either an insertion, deletion, substitution, or reversal of m.

Thus, in order to score the candidate corrections, the probability of occurrence of the suggested word is multiplied by the probability of occurrence of the specific error by which the suggested word differs from the misspelled word [KCG90, CG91].

The data collected from AP newswire was also used to estimate both sets of probabilities on a noisy channel model as the corpus contains a large number of typos. The sum of the scores for all the candidate corrections is then normalised. The probability of w is estimated by $P(w) = (freq(w) + 0.5)/(n + v/2)$, where $freq(w)$ is the number of times that w appears in the AP newswire corpus, n is the number of words in the corpus, and v is the vocabulary size—i.e. the number of words in dictionary. In this case, $n = 44,000,000$ and $v = 112,964$. $P(m|w)$ is computed from four so-called confusion matrices, where each represents spelling permutations, i.e. insertion, deletion, substitution, and reversal. Each confusion matrix represents a different type of spelling error. The elements of the matrix represent the number of times that each spelling error transformation appears in the training set. Details of the computation can be found in [KCG90, CG91].

Now let us consider the spell correcting algorithm used in the CORRECT programme. The algorithm takes the following input:

- W, the word list; and
- m, a misspelled word rejected by SPELL from Algorithm 4.2.2.

The algorithm is outlined below in GCL. It returns C, a set of pairs, $< x, y >$, where x is a correction word and y is its corresponding score—i.e. y is the probability that x is the correct spelling of m. In formal terms, $C \in U^+ \times \mathbb{R}$, where U^+ is the set of all possible non-empty strings in the alphabet, U; and $\mathbb{R}$ is the set of real numbers. In practice, the list of suggested words will be sorted in an order that begins with the words with the maximum scores.

Algorithm 4.3.2(Correct)

```
func $CorrectProb(m, W) : U^+ \times \mathbb{R}$
    $C :\, = \varnothing$;
    do $(i : [0, |W|)) \rightarrow$
        if $(Ins(m, W_i) \vee Del(m, W_i) \vee Sub(m, W_i) \vee Rev(m, W_i)) \rightarrow$
            $C :\, = C \cup \{< W_i, Scores(m, W_i) >\}$
        $[\!]$ $\neg(Ins(m, W_i) \wedge Del(m, W_i) \wedge Sub(m, W_i) \wedge Rev(m, W_i)) \rightarrow$ skip
        fi;
        $i :\, = i + 1$
    od;
    return $C$
cnuf
```

Another example of spelling correction using an error model for noisy channel spelling was developed by Brill and Moore where a new channel model based on generic string to string edits is employed to improve performance of the spelling correction task [BM00].

Note that CORRECT only addresses the spell correcting problem and it does not apply morphological analysis. The spell checking problem was taken care of by the SPELL programme.

4.4 ASPELL

ASPELL is open-source software which can either be used as a library or as a stand-alone spell checker. An example of using ASPELL for the spell checking support can be found in the PIDGIN multi-platform instant messaging client [Sof08]. ASPELL tackles spell checking using hashing (Section 3.3.1) for dictionary lookup. For spelling error correction, ASPELL [Atk04] combines Lawrence Philips' Metaphone algorithm and ISPELL's [Kue05] near miss strategy, in which a single-error transformation such as an insertion, or substitution derives a correct spelling from a misspelled word. There are two main differences between ISPELL and ASPELL:

1. ASPELL can handle Unicode Transformation Format-8 (UTF-8) [Uni03] automatically, meaning that it can spell check a language such as Japanese that uses a character set other than the Latin alphabet.

2. Secondly, ISPELL can only suggest corrections that are based on a edit distance of 1. It will not suggest more distant corrections based on English pronunciation rules.

Edit distance which was discussed in Section 3.4.1 is used in ISPELL and subsequently ASPELL to find the minimum number of 'edits' to transform the source (misspelled) word into the target (suggested) word—thereby reflecting the similarity of the misspelled word and the suggested word. In this section, insertion refers to the operation that an extra character is required to be inserted into a misspelled word to resolve the misspelling and deletion refers to the operation that a character is required to be deleted from a misspelled word in order to resolve the misspelling. The definitions of these two operations are inverse to the ones explained in Section 4.3.

As mentioned in the previous section, insertions and deletions are inverse to one another. If a misspelled word is corrected by an insertion, then the misspelled word can be regarded as the result of a deletion in the correctly spelled word and vice-versa.

ASPELL incorporates Philips' Metaphone algorithm [Phi90] to deal with common rules of the English pronunciation in order to achieve better spelling suggestions. Lait and Randell [LR93] noted that the algorithm uses 16 consonant classes and has a set of rules mapping letter combinations into the consonant classes. Vowels are only retained when they occur as the initial letter of the word.

Metaphone can be viewed as a simple partial function we choose to call *Metaphone* (4.4). Let R be the set of phonetic rules, M be a set of misspelled words, and S be their 'soundslike' equivalent[3].

Function 4.4.1. *(Metaphone):*

$$Metaphone : M \times R \nrightarrow S$$

Further information on the Metaphone algorithm can be found in [Phi90] while the phonetic rules are discussed in [LR93]. The Double Metaphone algorithm [Phi99] is the second generation of the original Metaphone algorithm which works on the same principles but with some improvement in the original's consonant classes.

In ASPELL, the edit distance takes into account transformation operations such as inserting a space or hyphen, interchanging two adjacent characters, changing a character, deleting a character, or simply adding a character to a string. Each such operation is assigned a certain cost. The total distance is the smallest total cost for transforming the misspelled word to the suggested word. The suggested word with the lowest score is deemed to be the best candidate. For example, the edit distance between two strings "test" and "tert" is 1—a single substitution is required to transform 'r' into 's'.

Thus, for a misspelled word m with edit distance d to any word w in our lexicon D, the set of candidate corrections is

$$\text{corrections} = \{w \in D | EditDistance(m, w) \leq d\}$$

[3]The 'soundslike' equivalent by definition is an approximation of how the words should sound.

ASPELL proceeds as follows:

1. Identify the misspellings by comparing a list of words against one of its built-in dictionaries. This step is described in Function 2.3.2.

2. Find all dictionary words that are edit distance two or less from a misspelled word.

3. To find a set of suggested corrections, each misspelled word is converted to its 'soundslike' equivalent using the Metaphone algorithm.

4. Find all dictionary words that have 'soundslikes' that are edit distance two or less from the 'soundslike' of a misspelled word.

The suggested corrections are ordered according to the weight average of the edit distance (i.e. each edit distance is weighted by a specific weight) of the word to the misspelled word and the soundslike equivalent of the two words. The words with the lowest scores are returned.

The strength of ASPELL lies in the fact that it uses edit distance at both the phonetic code level and the word level and this quality greatly improves the accuracy of suggestions. Unlike SPELL and CORRECT (described in Sections 4.2 and 4.3, respectively), ASPELL performs both spell error detection and spell correcting tasks.

As mentioned earlier, ASPELL can perform spell checking for many languages other than English, such as French, Spanish, German, etc. This is evident in spell checking in Skype—a free Voice over Internet Protocol (VoIP) software [Tec03]. The underlying algorithm stays the same despite the fact that the morphology for each language differs. Other major differences lie in the dictionaries and the phonetic data files involved. The dictionaries supplied by ASPELL comprises all inflected forms of words of the respective languages. Thus, no morphological analysis was applied.

ASPELL is the underlying framework used in JAZZY [Whi04], an open-source Java API used for spell checking. The only significant difference is that JAZZY incorporates the edit distance slightly differently compared to ASPELL. We will be looking into the performance of ASPELL in comparison with the other spell checkers and correctors in Chapter 5.

4.5 SPEEDCOP

SPEEDCOP (SPElling Error Detection/COrrection Project) was devised by Pollock and Zamora [PZ84, ZPZ81]. It relies on technology that uses a knowledge-based similarity key (as discussed in Section 3.4.2). It originally was aimed at automatically correcting spelling errors which are predominantly typing errors in a very large database. SPEEDCOP is designed to automatically correct spelling mistakes by finding words that are similar to the misspelled word, reducing the candidate suggestions to words that could be created by a single spelling error, and the suggestions are then sorted by the probability of the type of error and the probability of the suggested word. SPEEDCOP performs both spell checking and correcting tasks. However, it concentrates more on spell correction.

SPEEDCOP starts by generates a similarity key for each word in the dictionary and then sorts these keys in key order. It also generates a key for the misspelled word. Three dictionaries are created (the original dictionary; one for the skeleton keys; and one for the omission keys) and sorted in key order. Misspelling is corrected by locating words whose key collates most closely to the key of the misspelled words and the plausible correction is selected from these candidates by attempting to reverse the error operations encountered. This error reversal technique was first proposed by Damerau [Dam64]. Any number of strings may possess the same key. The collating proximity between two keys is a measure of similarity of the original words.

The SPEEDCOP algorithm computes two similarity keys, namely a skeleton key and an omission key. A skeleton key consists of the first character of the word followed by the remaining unique consonants in order of occurrence and then the unique vowels in order of occurrence. For instance, the skeleton key for the word "chemistry" would be *CHMSTRYEI*.

The correction results show that the most frequent cause of failure to correction with words in the dictionary is caused by the early positions of consonants in the key. This is due to the great collating distance between the skeleton key of the word and the key of the misspelling when an incorrect consonant is positioned close to the beginning of a word. It prevents the valid form of words from being retrieved from the dictionary. This is to say that one of the first few consonants in the word is incorrect and the similarity measure for the key for the word and the key for the misspelled word might be relatively great.

An omission key is built to correct this shortfall of the skeleton key. The omission key sorts the unique consonants in a reverse order of the frequency of omission and then by the unique vowels in order of occurrence. In other words, an omission key consists of unique consonants in reverse order of the frequency of omission (in English, experimentally found to be RSTNLCHDPGMFBYWVZXQKJ) of omitted letters followed by vowels in order of occurrence in the word. For example, the word "chemistry" has an omission key *YMHCTSREI*. The construction of an omission key is less intuitive compared to the construction of a skeleton key.

Both skeleton key and omission key contain the fundamental features of a word. In the process of generating the keys, any duplicate of a letter in the word—not only any adjacent duplicate—is eliminated, i.e. both keys are based on single occurrences of all the letters that appear in a word.

For example, the skeleton key of the misspelled word "chemistrie" would be *CHMSTREI* which would be placed between *CHMSTEI* ("chemist") and *CHMSTRYEI* ("chemistry"). A form of linear backwards and forwards search is used to identify the nearest keys for valid words. This continues until enough "plausible" corrections are accumulated (in a set called the retrieval set) where a plausible correction is one which requires only a single letter insertion, deletion or substitution, or the reversal of two letters (a process called error reversal).

The error reversal algorithm is applied to the retrieval set containing the dictionary words whose keys collate closest to that of the misspelled word. In SPEEDCOP, Damerau's method was adapted and improved by checking the plausibility only on the similar keys instead of the entire dictionary. If the correction is not found by the skeleton key comparison, the omission key comparison follows. For more detail on the error reversal algorithm, see [PZ84].

SPEEDCOP also incorporated a dictionary with common misspellings (apart from its main dictionary) and a function word routine in order to improve its spelling correction accuracy. If a word is misspelled, SPEEDCOP first consults a dictionary with common misspellings. If the word is not found in this dictionary, SPEEDCOP thereafter applies a 'function word'[4] routine which checks if a misspelled word is made up of a function word and a correctly spelled word. The function is designated below as *FuncWord*. It checks whether the misspelled word contains a function word and a correctly word, and, if so, returns a pair <function word, correctly spelled word>. For instance, "theman" is the misspelled word. After applying *FuncWord*, <the, man> is returned. If the misspelled word does not contain a function word and a valid word, then we assume below that *FuncWord* returns a pair $< \varnothing, \varnothing >$.

The spell checking procedure of SPEEDCOP starts by comparing words of four to six characters in length to a list of high frequency words to detect misspelling. If the word is not found in the list, the chosen dictionary is used for further searching. The algorithm takes the following input:

- m, the misspelled word;
- F, the list of high frequency words regarded as a set;
- D, a dictionary also regarded as a set;
- S, the list of skeleton keys of words in the dictionary, where S_j is used to denote the skeleton key for the j^{th} word in D;
- O, the list of omission keys of words in the dictionary, where O_j is used to denote the omission key for the j^{th} word in D; and
- d_1 and d_2, are correctly spelled words which can be either a function word or a dictionary word.

The algorithm computes a list of words that have been checked for plausibility and returns them as a set of corrections in P. It does this by computing the following:

- X, the skeleton key for m;
- Y, the omission key for m; and
- C, the set of possible corrections.

[4]A function word also called a grammatical word is one with little lexical meaning or has ambiguous meaning. It expresses grammatical relationships with other words within a sentence or mood of the speaker. Articles (e.g. *a* and *the*), pronouns, conjunctions, auxiliary verbs, interjections, particles, expletives, etc. are all function words.

The *SimilarityKeys* function is used to find the skeleton and omission keys of m, returning these as a pair <skeleton key, omission key>. The *ErrorReversal* function is used to check the plausibility of the list of corrections.

Note that in this description, the **for** construct is used as an iterator that traverses all elements of a range, say $[\ell, h)$. It is also assumed that the elements of a set, say S, are referenced by subscripts $S_0, S_1, \cdots, S_{|S|-1}$. The SPEEDCOP algorithm can be described as follows:

Algorithm 4.5.1(Speedcop)

```
func $Speedcop(m, F, D, S, O) : P$
    $P := \varnothing;$
    if $((m \in F) \vee (m \in D)) \rightarrow P := P \cup \{m\}$
    [] $((m \notin F) \wedge (m \notin D)) \rightarrow\ < X, Y > := SimilarityKeys(m)$
      for $(i : [0, |D|)) \rightarrow$
         if $(X = S_i) \rightarrow C := C \cup \{S_i\}$
         [] $((X \neq S_i) \wedge (Y = O_i)) \rightarrow C := C \cup \{O_i\}$
         [] $((X \neq S_i) \wedge (Y \neq O_i) \rightarrow$
           if $(FuncWord(m) =< d_1, d_2 >) \rightarrow C := C \cup \{< d_1, d_2 >\}$
           [] $(FuncWord(m) =< \varnothing, \varnothing >) \rightarrow$ skip
           fi;
         fi
      rof;
      for $(i : [0, |C|)) \rightarrow$
         if $((ErrorReversal(X, C_i)) \vee (ErrorReversal(Y, C_i))) \rightarrow P := P \cup \{C_i\}$
         [] $\neg((ErrorReversal(X, C_i)) \vee (ErrorReversal(Y, C_i))) \rightarrow$ skip
         fi
      rof
    fi;
    return $P$
cnuf
```

The similarity-key correction algorithm is effective in locating words most similar to the misspelled words in a large dictionary. It requires less complicated computation compared to most string distance measure techniques. The accuracy of this algorithm is largely dependent on the number of words and the type of words covered in the dictionary and the assumptions behind the key structure. No morphological analysis was applied. The drawback of SPEEDCOP is that it is restricted to single-error misspelled words whose corrections can only be found in the dictionary.

4.6 FSA

One usage of the FSA package implemented by Daciuk [Dac98] is to perform spell checking and correcting tasks with finite-state automata. The notion of a finite-state automaton (FSA) was discussed in Section 3.3.1.

The FSA package implements deterministic acyclic finite-state automata with final transitions. A deterministic acyclic FSA is an automaton that has one start state, has no ϵ-labelled transitions[5], each state has no more than one transition labelled with the same symbol, and contains no cycles. A deterministic acyclic FSA is ideal for representing dictionaries or lexicons [JM00, Wat03]. A deterministic acyclic FSA accepts a words if a transition was found for each input symbol and the last transition it traversed was final.

The FSA is constructed so that the language it accepts is exactly the words in the lexicon. If the word cannot be found in the lexicon, it is considered as a misspelled word, and a list of word suggestions is provided. To find all possible candidate corrections of the misspelled word, certain paths starting from the start state to one of the final transitions of the finite automaton must be found. More precisely, we are interested in paths spelling out correct words which are within a given edit distance of the misspelled word. This path exploration is done such that a path is abandoned if it becomes clear it will lead to an edit distance above the desired distance. Oflazer [Ofl96] used a 'cut-off' edit distance to determine if or at which point the current path should be skipped.

The cut-off distance measures the minimum edit distance (which was discussed in detail in Sections 3.4.1 and 4.4) between an initial substring of the misspelled word and the partial candidate correction. Let m denote the misspelled word and suppose its length is n. Let c denote the partial candidate string and suppose its length is r. Suppose that t is the threshold edit distance between the two strings that will be tolerated.

Then Oflazer defines the cut-off edit distance of m and c as follows:

$$cuted(m:[1,n], c:[1,r]) = \min_{l \leq i \leq u} d(m:[1,i], c[1,r])$$

where $l = max(1, r-t)$, $u = min(n, r+t)$, and i denotes the i^{th} letter in w. The initial substring of M are of length in the range from $r-t$ to $r+t$. $r-t$ must be greater than 1.

[5]With final transitions (instead of the more traditional final states), the lack of ϵ-transitions implies that Daciuk's automata cannot accept the empty string.

$r + t$ cannot be greater than m as it is not possible to append any additional characters to the end of m. For more discussion of these details, see [Ofl96].

For example, let m be *reprt* and c be *repo*. The cut-off edit distance will be:

$$\begin{aligned}
&\mathit{cuted}(\text{reprt,repo}) \\
&= \min_{2\leq i\leq 5} d(\text{reprt: } [1, i], \text{ repo}) \\
&= \min \{d(\text{re,repo}),\ d(\text{rep,repo}),\ d(\text{repr,repo}),\ d(\text{reprt,repo})\} \\
&= \min \{d(2, 1, 1, 2\} \\
&= 1
\end{aligned}$$

Since *cuted*(reprt,repo) $\leq t$, the FSA algorithm permits a transition from the node representing the partial candidate string 'repo' to a next possible node in the finite state diagram. This transition is permitted because there is a possibility that by the time a final state is reached, the edit distance between the misspelled word and the candidate correction will be less or equal to t. Had it been the case that *cuted*(reprt,repo) $> t$, no further exploration along the path of the candidate string would be worth while.

In general, the algorithm checks if the cut-off edit distance of m and c is within the threshold t before extending c. If the cut-off edit distance is greater than t, the last transition is abandoned, we return to the previous state and we move on to the next possible path. This action of backtracking is recursively applied when the search cannot be carried on from a specific state. On the other hand, if a final state is reached while the cut-off edit distance stays within the threshold t, c is considered a valid candidate correction for the misspelled word m.

Daciuk's algorithm uses depth-first search in the FSA with the cut-off edit distance. Daciuk details several efficiency modifications to this algorithm—such as tabulating the cut-off edit distance computation to reuse computed values.

In the FSA package, morphological analysis is applied if the lexicon used does not already have all inflected forms. The morphology of a language is then fully captured in a single finite-state transducer or a finite-state acceptor. Initially, the FSA is used to test the word for acceptance. If the word is rejected (a misspelling is identified), the following two steps are applied:

1. Morphological analysis of the misspelled word is used to recognise the lexeme (see Section 3.2) as well as its morphological annotations.

2. The lexeme and the morphological annotations are used for morphological generation—the inverse of morphological analysis—generating possible candidate correction of the misspelled word. However, if the lexeme is not present in the lexicon or inflected forms of the given annotations cannot be found, no candidate correction is generated.

The FSA package is a significant example of using the finite-state approach to tackle spell checking and spell correcting tasks. Languages that are agglutinating or highly inflectional, such as Finnish, Turkish and Zulu, and languages with compound nouns, such as German, are well suited to the finite-state approaches (more specifically deterministic finite-state automata) along with edit distance technique. In [MS02, MS04], Mihov and Schulz extensively discussed the details of such approach.

4.7 AGREP

AGREP (*approximate* GREP) is a bit-mapping technique for pattern matching which was reinvented [Dom64, NR02, WM92a] in 1992. The software package developed by Udi Manber and Sun Wu is described in [WM92b]. AGREP allows for approximate string matching [Smy03] where the number of mismatched characters can be specified by the users. It is record oriented and these records are defined by specifying a pattern that marks the beginning of a new record, which can be overlapped and nested inside other records. Multiple pattern searching is achieved by making use of the bitwise AND and OR operators. The text and pattern can be DNA sequences, lines of source code, etc. In the case of spelling checking/correcting, the text and the pattern are specifically referred to as sequences of characters.

The main underlying algorithm, bitap (bit-parallel approximate pattern matching), in AGREP supports many extensions other than exact and approximate string matching, such as approximate regular expression pattern matching, simultaneous matching of multiple patterns, etc. AGREP uses several different algorithms to optimize its performance. It uses a variant of Horspool algorithm [Hor80] for simple exact queries when the length of the strings do not exceed 400 characters. For longer strings that exceed this particular length restriction, an extension of the Horspool algorithm is used, however, pairs of characters as opposed to single characters are used to build the shift table in this case. For other types of patterns, AGREP is based upon Baeza-Yates and Gonnet's [BYG92] numeric scheme for exact string matching (also known as the Shift-And/Shift-Or algorithms). Approximate string matching are handled by PEX [NR02], a partition scheme for simple patterns with errors and row-wise bit-parallelism[6] (BPR) [WM92a].

In this section, we will focus on the Shift-And algorithm (which AGREP is based on) that is used for spell checking as spell correcting is also based upon this algorithm. The Shift-And algorithm can be described as follows:

Assume that there exists a text T of size n (i.e. $T = t_1t_2\cdots t_n$) and a pattern P of size m (i.e. $p = p_1p_2\cdots p_m$). Assume also that $m < w$, where w denotes the size of a computer word, so P can fit in a computer register. Let $U = \{u_1, u_2, \cdots, u_{|U|}\}$ which denotes a set of finite alphabet of symbols (as defined in Section 2.2) that appear in T and P.

The algorithm builds a pre-computed table B, which stores a bit mask of size m for each character. The mask in $B[u_i]$ has the j^{th} bit set if $P_j = u_i$. To optimize this table, only the arrays for the characters that appear in P are constructed. The bitwise operations work in such a way that if a character does not belong to the alphabet, its mask will be 0 for all bits and if a character belongs to the alphabet, its mask will be 1 for all the bits for the position in the pattern. The bitwise operations make this algorithm extremely fast as only characters where the bit is set are compared.

A bit mask D is used to represent the set of all the prefixes of P that match a suffix of T and $D = d_m\cdots d_1$, i.e. $d_1 = 1$ represents a match for p_1, $d_2 = 1$ represents a match

[6]Bit-parallelism packs many values in a single computer word and update them all in a single bit operation.

j/i	c	a	t
c	1	0	0
a	0	1	0
t	0	0	1
s	0	0	0

Table 4.2: Example of a bit mask table.

for p_1p_2, and $d_m = 1$ represents a match for $p1_p2 \cdots p_m$, etc.
Initially, we set $D_i = 0^m$ (note that 0^m denotes m zeros). If the characters from T are examined from left to right after reading the i^{th} character of T, D_i denotes the value of D after the i^{th} character of T has been processed (i.e. D_i contains information about all matches of prefixes of P that end at i). The algorithm ensures that $D_i[j] = 1$ if and only if $p_1p_2 \cdots p_j = t_{i-j+1}t_{i-j+2} \cdots t_i$. This is to say the j^{th} position in D_i is said to be active if and only if the first j characters of P match exactly the j characters in T starting at $i - j + 1$ and ending in i. A match is reported whenever $d_m = 1$. When we read t_{i+1}, we need to compute the new set D_{i+1}. The algorithm also ensures that $D_{i+1}[j + 1] = 1$ if and only if $D_i[j] = 1$ *and* $t_{i+1} = p_{j+1}$.

For each t_{i+1}, the transition from D_i to D_{i+1} can be described as follows:

$$D_{i+1} : ((D_1 << 1)|0^{m-1}1)\&B[t_{i+1}] \tag{4.2}$$

where $<< 1$ indicates moving the bits to the left and enters a 1 from the right (i.e. a right shift). $|$ denotes the bitwise OR and & denotes the bitwise AND.

It can be shown that (4.2) results in $D_i\&(10^{0-1}) \neq 0^m$ if and only if there is a match in position $j - m + 1$ (Algorithm 4.7.1).

An example of a table B is given in Table 4.2. If $T = cats$ and $p = cat$, $B = \{c, a, t\}$ in this case. The bit masks for the characters that appear in p are $B[c] = 100$, $B[a] = 010$, and $B[t] = 001$.

When reading the c of T, $D_1 = 100$ AND $B[c] = 100$. Thus, the new $D_1 = 100$. When reading the a, $D_2 = 110$ (after a right shift of the values in D_1 and fills the first position with a 1). D_2 AND $B[a]$ produces the new $D_2 = 010$. When reading the t, $D_3 = 101$ (after a right shift of the values in D_2 and fills the first position with a 1). D_3 AND $B[t]$ produces the new $D_3 = 001$. The last bit of D_3 is a 1. Hence, in step 3, the first occurrence of P is found, i.e. $i = 3$. Thus, the starting position where the complete match is found is at position 1 ($i - m + 1 = 3 - 3 + 1$). Notice that s is not present in P. Thus, $B[s] = 000$.

Now let us consider spell checking in AGREP. The algorithm takes the following input:

- U, the alphabet;
- P, a pattern of size m, where $P = p_1p_2 \cdots p_m$; and
- T, a text of size n, where $T_j = t_1t_2 \cdots t_n$.

The algorithm computes the following:

- B, a table which stores a bit mask of size m for each character; and
- D, a bit mask of size m that is used to contain information of a set of all the prefixes of P that match a suffix of T.

The algorithm then outputs the position in T where an occurrence of P occurred. The algorithm can thus be expressed as follows:

Algorithm 4.7.1(Exact matching in AGREP)

```
func $BitapExactMatching(P, T) : int$
    $D, match := 0^m, -1;$
    for $(c \in U) \rightarrow B[c] := 0^m$ rof;
    for $(i : [1, m]) \rightarrow B[p_i] := B[p_i] | 0^{m-i}10^{i-1}$ rof;
    for $(j : [1, n]) \rightarrow$
        $D := ((D << 1) | 0^{m-1}1) \& B[t_j];$
        if $(D \& 10^{m-1} \neq 0^m) \rightarrow match := j - m + 1$
        $[\!]$ $(D \& 10^{m-1} = 0^m) \rightarrow$ skip
        fi
    rof;
    return $match$
cnuf
```

Assume that Algorithm 4.7.1 can be done in constant time, the time complexity of spell checking based on the Shift-And algorithm is (n), where n is the size of a text word in T.

Bitap then extended the spell checking task described in 4.7.1 to perform the spell correcting task. The way that bitap naturally maps onto bitwise operations distinguishes it from other string searching algorithms. If the input file contains an exact match of a particular pattern and if errors are allowed, then there will be several matches and the number of exact and approximate matches depends on the number of errors allowed in a word. For example, if the pattern is the word *book* and if the file contains this pattern, a match for *book* will be reported, but also for *_book*, *boo*, *ook*, and *book_* if one error is allowed in each word. There is one exact matching, two deletions, and two insertion operations which together constitute a total of five matches.

The spelling correction by AGREP can be described as determining whether an input text T contains a substring which is approximately equal to a given string, under some measure of closeness. Thus, the algorithm is to find all substrings in T that are of distance

k (i.e. at most k errors (insertion, deletion, or substitution errors)), which is measured by edit distance, to P. In other words, the aim is to find all substrings in T that contain a proximity of the pattern P with edit distance at most k.

The spell correcting task can thus be described as follows:

Assume a text T of size n and a pattern P of size m. Let the alphabet be U where $U = \{u_1u_2 \cdots u_{|U|}\}$. A bit mask D is used to represent the matches between T and P. The algorithm builds a pre-computed table B, which stores a bit mask of size m for each character. For each type of error (i.e. insertion, deletion, or substitution), there exist at most k number of errors allowed in a word. For example, if we allow matching with one insertion error (i.e. $k = 1$), there exist at least two matches: an exact match (i.e., a match with 0 error) and a match with one insertion error allowed.

Initially, we set $D_i = 0^m$. D_i once again denotes the value of D after the i^{th} character of T has process. D_i is computed for exact matches. For approximate matches, k additional arrays which are denoted as $D_i^1 \cdots D_i^k$ are introduced here. The array D_i^d, where $d \leq k$, stores all possible matches with at most d mismatches (either insertions, deletions, or substitutions). Thus, D_i^1 is a bit array that contains information about all possible matches up to T_i with at most one insertion, deletion, or substitution. This algorithm ensures that $D_i^1[j] = 1$ (i.e. the j^{th} position in D_i^1 is set to be active) if an only if the first j characters of P match exactly the $j+1$ characters in T starting at $i-j+2$ and ending in i. A match is reported whenever $d_m = 1$. For an exact match, it has to be the case that $D_i[m] = 1$. For a match with at most one insertion, one deletion, or one substitution, $D_i^1[m] = 1$. The algorithm ensures that $D_{i+1}^d[j+1] = 1$ if and only if $D_i^d[j] = 1$ and $t_{i+1} = p_{j+1}$ or any one of the following conditions: D_i^{d-1}, $D_i^{d-1} = 1$, and $D_{i+1}^{d-1} = 1$.

The transition from D_i to D_{i+1} for the exact matches is as discussed earlier. In order to obtain a match of the first j characters up to t_{i+1} with $\leq d$ errors (i.e. the transition from D_i^d to D_{i+1}^d), the following cases need to be considered:

- Exact matching: there is a match of $p_1p_2 \cdots p_j = t_{i-j+1}t_{i-j+2} \cdots t_i$ with $\leq d$ and $p_j = t_{i+1}$. This is handled in the same manner as in 4.7.1.
- Inserting t_{i+1}: there is a match of $p_1p_2 \cdots p_j = t_{i-j+1}t_{i-j+2} \cdots t_i$ with $\leq d-1$ errors. This requires a right shift of D_i^d, one & with $B[t_i]$, and one — with a right shift of D_i^{d-1}.
- Deleting p_i: there is a match of $p_1p_2 \cdots p_{j-1} = t_{i-j+1}t_{i-j+2} \cdots t_{i+1}$ with $\leq d-1$ errors. This requires a right shift of D_i^d, one & with $B[t_i]$, and one — with a right shift of D_{i+1}^{d-1}.
- Substituting t_{i+1}: there is a match of $p_1p_2 \cdots p_{j-1} = t_{i-j+1}t_{i-j+2} \cdots t_i$ with $\leq d-1$ errors. This requires a right shift of D_i^d, one & with $B[t_i]$, and one — with D_i^{d-1}.

For each t_{i+1}, the transition from D_i^d to D_{i+1}^d can be described as follows:

$$D_{i+1} : ((D_i << 1)|0^{m-1}1)\&B[t_j] \tag{4.3}$$

$$D_{i+1}^d : ((D_i^d << 1)\&B[t_j])|D_i^{d-1}|(D_i^{d-1}|D_{i+1}^{d-1}) << 1) \tag{4.4}$$

Now let us consider spell correcting in AGREP. The algorithm takes the following input:

- Σ, the alphabet;
- k, the number of errors allowed;
- P, a pattern of size m, where $P = p_1 p_2 \cdots p_m$; and
- T, a text of size n, where $T_j = t_1 t_2 \cdots t_n$.

The algorithm computes the following:

- B, a table which stores a bit mask of size m for each character;
- D, a bit mask of size m that is used to contain information of a set of all the prefixes of P that match a suffix of T; and
- D^d, a bit mask of size $m + k$ that is used to contain information of a set of all matches in T that contain approximations of P with edit distance at most k.

The algorithm then outputs the position in T where an occurrence of P with at most k distances. The algorithm can thus be expressed as follow:

Algorithm 4.7.2(Approximate matching in AGREP)

```
func $BitapApproxMatching(P, T, k) : int$
    $D, D^d, oldD, newD, match : = 0^m, 0^m, 0^m, 0^m, -1;$
    for $(c \in U) \rightarrow B[c] : = 0^m$ rof;
    for $(i : [1, m]) \rightarrow B[p_i] : = B[p_i]|0^{m-i}10^{i-1}$ rof;
    for $(j : [0, k]) \rightarrow D^d : = 0^{m-1}1^i$ rof;
    for $(r : [1, n]) \rightarrow$
        $oldD : = D;$
        $newD : = ((oldD << 1)|0^{m-1})\&B[t_j];$
        $D : = newD;$
        for $(j : [1, k]) \rightarrow$
            $newD : = ((D^d << 1)\&B[t_j])|oldD|((oldD|newD) << 1);$
            $oldD : = D^d;$
            $D^d : = newD;$
            **rof**;
            **if** $(newD\&10^{m-1} \neq 0^m) \rightarrow match :\, = j$
            $[\!]$ $(newD\&10^{m-1} = 0^m) \rightarrow$ **skip**
            **fi**
        **rof**;
        **return** *match*
    **cnuf**
```

There exist $k+1$ bit masks. Thus, the time complexity of spell correcting is $\mathcal{O}((k+1)n)$.

If the number of errors is small compared to the size of the P, then the running time can be improved by the partition approach. Information on the partition approach is available in [WM92a].

The complexity of the search pattern affects AGREP's efficiency. Another shortcoming of AGREP is that there exist many restrictions to the length of the pattern strings. No morphological analysis was applied in AGREP. Refer to [Smy03] for details on the algorithms used in AGREP.

AGREP is highly flexible as it supports not only simple string matching but many other extensions, such as sets of characters, wild cards, unknown number of errors, a combination of patterns with and without errors, non-uniform costs, a set of patterns, long pattern, regular expressions, and very large alphabet. The AGREP programme will form part of the classification in Chapter 5.

4.8 AURA

The AURA (Advanced Uncertain Reasoning Architecture) modular neural system [HA02, HA03] was developed to act as a pre-processor for an information retrieval system. It is a spell checker and corrector. The basis of the AURA system is the AURA modular neural network [Aus96].

AURA is a hybrid spell checking and correcting (refer to Section 3.4.6) architecture as will be explained in more detail below. It makes use of phonetic matching and Correlation Matrix Memories (CMMs) [Koh88, Aus96] to correct single-letter spelling errors (i.e. insertions, deletions, substitutions, and reversals) and phonetic spelling errors. CMM was briefly described in Section 3.4.4. CMM is a single-layer binary associative neural network or memory which can be thought of as a matrix of binary weights and is used for storing and matching a large amount of patterns efficiently. Numerical inputs are converted into binary ones with the most achievable uniformity. The fundamental function of CMM is the learning of binary vectors that represent items of information and retrieval of these items when the appropriate inputs are provided. CMM is a binary matrix memory structure that stores a mapping μ between a binary input vector of length m and a binary output vector of length n. Thus,

$$\mu : \{0,1\}^m \rightarrow \{0,1\}^n$$

[HA03]. AURA makes use of CMMs to map inputs to outputs through a supervised learning rule that is somewhat similar to a hash function.

CMM allows very fast retrieval. The retrieval is independent of the total amount of information learned and stored in the memory for given CMM size. CMMs in AURA output all expected matches (no false positive or false negative matches) during single iteration partial matching [TA97]. The orthogonal vectors used for uniquely identifying each output word also contributes to the fast retrieval as there are incorrect matches caused by bit interference [TA97]. New inputs are incorporated into the matrix and do not require additional memory which makes the storage in CMM efficient.

AURA focuses on isolated-word error correction (Section 3.4). It constitutes a set of methods aimed at high performance pattern matching and low computational cost. AURA is typically used in very large data sets (more than one million elements). A scoring scheme is used for word retrieval from each spelling approach and an overall score for each matched word is calculated.

The phonetic spelling errors are overcome by a coding approach similar to that of Soundex and Phonix together with transformation rules derived from Phonix and Double Metaphone used in ASPELL [Phi99]. Phonix was specifically designed for name matching and includes Dutch phonetic rules. Soundex makes use of seven codes and Phonix makes use of nine codes to preserve the letter pronunciation similarities. More details on Soundex and Phonix can be found in [Kuk92] and [Gad90] respectively.

Two different approaches are employed to correct typing errors. Insertion and deletion errors are handled by an n-gram approach which matches small character subsets of the misspelled word (refer to Section 3.4.4). Statistical correlations between two adjacent characters are incorporated in this approach. Substitution and reversal errors are handled by a Hamming distance approach (refer to Sections 3.4.4 and 3.4.6).

AURA makes use of a lexical token converter [HA03] to map characters onto binary vectors. It is assumed that both the input word and words in the lexicon comprise sequences of characters from twenty-six characters ($a \cdots z$) and four punctuation characters ($-, ', \&, /$) which constitutes a total of thirty characters in a finite alphabet. (Only four punctuation characters are used to conserve memory.) Words are translated to binary bit vectors by mapping characters onto specific bit positions. The lexicon is thus represented by a binary matrix (the CMM) that stores all bit vectors, i.e. (30×30-bit chunks) $\times$ (number of words).

The lexical token converter entails the following:

1. A binary bit vector of size 960 is divided into a series of 30-bit chunks where each chunk represents a character in the finite alphabet for the inputs. Words comprised of up to thirty characters may be represented. Two additional chunks are reserved for the shifting n-gram which will be described later.

2. Each word is divided into characters and each character is represented by the appropriate bit set in the chunk in order of occurrence. For instance, all occurrences

of the letter 'a' in a word are represented by bits set in the first chunk. The position of each set bit in this chunk corresponds to where each 'a' occurred in the word.

3. Similarly, each 'b' in a word is represented by a bit set in the second chunk, etc. Chunk ordering thus corresponds to the normal lexicographic ordering for the English alphabet, followed by the ordering $(-,', \&, /)$. Thus, every occurrence of '/' in a word results in the corresponding bits being set in the 30^{th} chunk.

4. The result is that the spelling of a word is represented by a binary bit vector which is produced by concatenating all the chunks. This binary bit vector forms the input to the CMM which represents the lexicon. All the bits in any unused chunks are set to zero.

4.8.1 Spell checking in AURA

The spell checking task in AURA is tackled in the following manner: A set of words are submitted for spell checking. The list of words in the lexicon is held in an array. If the input word is present in the lexicon, an option is presented allowing one to accept the match result or to carrying on searching for similar words. Similar words have the same word stem but different suffixes, such as {*look*, *looks*, *looked*}. If the input word cannot be found in the lexicon, it is assumed to be a misspelling and AURA returns a set of suggestions from the lexicon with no ordering. Spell checking is performed using Hamming distance and a length match (i.e. making use of the length of the input word as the threshold).

The length of the input word is set as the threshold. Hamming distance is used to find all lexicon words beginning with the input words. A binary input vector with all bits set to 1 is used to count the number of matching characters in each lexicon word. All the characters of the input word and lexicon words are left aligned. Each character of the input word is then compared with the character of the lexicon words in the same position. Only words that contain the exact matching of the input word are output to a bit vector. A length match entails searching the word length array for lexicon words that have the same word length as the input word. The matching lexicon words are output to a bit vector. The final step is to combine the two output results using a logical AND. The resultant bit vector is passed to the lexical token converter to retrieve the exact matching word.

If the spell checking procedure does not return any exact matching words, it is assumed that the input word is a misspelling. We then produce a list of alternative suggestions by combining the best match found with binary Hamming distance and the best match found with shifting n-gram. Two output vectors are combined by a bitwise OR and indexes to the lexical token converter to retrieve word suggestions.

4.8.2 Spell correcting in AURA

The AURA system uses two CMMs independently for spelling suggestions for misspellings caused by typing errors: one for the words with binary Hamming distance matching and shifting n-gram and one for the phonetic codes [HA02]. A scoring system is used in order to rank the word suggestions. Only the best ten possible suggestions are presented to the user in order to achieve high recall and accuracy. The scores for the binary Hamming distance and shifting n-gram approaches are separated. The score from the phonetic module is added to both the n-gram and Hamming distance scores in order to produce two word scores. The highest score amongst the two is considered the overall word score.

For single-letter spelling errors, CMM for binary Hamming distance and shifting n-gram is used. The inputs are formed from the word spellings and the outputs are the matching words from the lexicon. Both the inputs and the outputs are translated to their respective binary bit vectors by the data-to-binary lexical token converter explained earlier.

Each word in the lexicon is represented by a unique orthogonal binary vector which forms the output from the CMM. One bit is set according to the position of the word in the alphabetical list of all words. The output from the CMM is used to identify when the word has been retrieved as a match by looking up the bit set and use the position to index the words in the alphabetical list of all words.

To train the network, the binary patterns representing the word spellings or phonetic codes form the inputs to the CMM and the CMM outputs the binary patterns for the words that match the inputs. An binary input vector is associated to an orthogonal output vector which serves as an identifier to uniquely identify each spelling in the lexical token converter. When both an input row and an output column are both set to one, the CMM is then set to one. After storing all associations between the input and output vectors, the CMM weight w_{kj} is as given by the following equation [HA03]:

$$w_{kj} = \sum^{\forall i} (I_j^i \& O_k^i) \tag{4.5}$$

where I and O denote the binary input and output vectors respectively, & represents the bitwise AND, i indicates the position in the vector, j indicates the row number, and k indicates the column number. w_{kj} is an integer value, i.e. the sum of a number of 0 and 1 computations with 0 being the result of a bitwise AND operation that yields false and 1 being the result when the bitwise AND yields true.

As mentioned earlier, two approaches are used for recalling from the network and they are shifting n-grams and binary Hamming distance.

Hamming distance approach

For the binary Hamming distance approach, the CMM is used to retrieve the lexicon words that have matches with the input word by having the same character in the same position in the word. Thus, the CMM is used to count the aligned letters between the lexicon words and input word. The spelling pattern is applied to the network for recall

CMM	output activation	binary output vector
we	0	0
the	2	0
tea	1	0
none	0	0
tree	4	1
three	2	0
treasure	3	0

Table 4.3: Recalling from the CMM using binary hamming distance.

only. An output activation vector is produced by adding the columns:

$$O_j = \sum^{\forall i}(I_i \& w_{ji}) \tag{4.6}$$

where I, O and & are as indicated in the previous equation. i and j indicate the row and column in the vector respectively, and w_{ji} is an integer value.

The number of bits set in the input vector (i.e. the length of the input word) is used as the threshold in order to retrieve all lexicon words that contain the input word at the beginning. This is to say the input word forms a word stem to the retrieved lexicon words. A bit vector with all bits set to 1 and with the threshold sets to the length of the input word in order to count the number of matching characters in each lexicon word.

The output vector of the CMM is used to produce a binary output vector by setting the threshold. The output vector from the CMM represents the matching words between the lexicon words trained into the CMM and the input word presented to the CMM for recall. More precisely, it represents an identifier for the input word to uniquely identify the word in the lexical token converter. The threshold is then set to the highest activation value in the output vector to retrieve the best matches. This is to say the threshold is set according to the number of bits set in the input word vector. For instance, if the input word consists of two characters with one bit sets in each chunk, then the threshold is set to 2 in order to identify any exact matches from the lexicon words that match both characters in the corresponding positions (i.e. all columns that sum to 2). All the values (i.e. positions) in the binary output vector where the corresponding values of the output activation vector is greater than or equal to a predefined threshold are set to 1. The rest bits are set to 0. In case where only a partial match of the input word is requested, the input word is sent to the CMM and the threshold is set to M, where $M <$ the length of the input word. Thus, the threshold of the output vector is set at the highest activation value in order to retrieve all best matching lexicon words. The binary output vector is then passed to the lexical token converter which separates the bit vector into separate orthogonal vectors that represent the matches that have the maximum number of characters matching the input word. the word associated with each separate orthogonal binary vector is then retrieved.

If the input word is *tree* which has a word length of 4, an example of the output vector is illustrated in Table 4.3.

As an example of spell checking, the input word *tree* is compared to the lexicon words trained into the CMM. The input word consists of four letters. Hence, the threshold is set to 4. During the Hamming distance match, only those lexicon words with the first four letters matching the four letters of the input word with each matching characters at the same position are retrieved. In this case, only the word $\{tree\}$ is retrieved to the bit vector. During a length match, any lexicon words containing exactly four letters are retrieved. Thus, the words $\{none, tree\}$ are retrieved to the bit vector. The exact match is then identified by combining the two outputs with a logical AND. Thus, the bit set in the output bit vector indexes into the lexical token converter to retrieve the word *tree* as the exact match.

For spelling correction, if $M = 3$, the word *treasure* is retrieved as its first 3 letters match the first 3 letters of *tree*. The bit vector is passed on to the lexical token converter to retrieve the actual word.

Note that the "?" convention from Unix is adopted to represent unknown characters in a word. All bits in the chunks are set to represent a universal OR. This is useful when the user is unsure of the correct spelling of a word.

Shifting n-gram approach

For the shifting n-gram approach, the lexicon words are trained into the CMM to start with. The CMM is then used to count the number of the n-grams in the input word present in each lexicon word. Three n-gram approaches [Ull77] were used: unigrams (for input words with less than four characters), bigrams (for input words having between four and six characters), and trigrams (for input words with more than six characters).

Firstly, the length of the input word is identified. Then the type of n-gram approach is selected. The first n characters of the input word is input and left-aligned to the CMM. The threshold is set at n to indicate the number of character in the n-grams of the lexicon words that match the n-gram of the input word. We are now set to find lexicon words matching all the characters of the n-gram, i.e. all words in the output vector with an output activation of the n specified for their first n characters.

A bit vector is then produced according to the output activation. The first n-gram of the input word is now shifted one place to the right and input to the CMM. This procedure is repeated until the first character of the n-gram of the input word is shifted to the position of the last character of the longest word in the lexicon. All the output bit vectors are then combined by a logical OR operation. The resultant output vector thus indicates any word that has matched any of the n-gram positions. All the rest of the n-grams are input into the CMM following the same procedure as described. All the resultant output bit vectors are then summed to produce an integer vector which represents a count of the number of n-grams matched for each word.

The maximum value of the output activation vector is used as the threshold and the bits in a output bit vector are set according to the corresponding output activation values. The output vector that correspond to the output activation vector is passed on to the lexical token converter to retrieve the matching words. The lexical token converter separates the bit vector into separate orthogonal bit vectors before retrieving the word associated with each individual orthogonal vector. In other words, there is one matching word in the lexical token converter for each bit set for orthogonal output vectors. All three

CMM	output activation	bigram matches (bit vector)
we	0	0
the	1	0
tea	1	0
none	0	0
tree	2	1
three	1	0
tr	0	0

Table 4.4: Recalling from the CMM using shifting n-gram.

approaches follow the same procedure. The difference between the three approaches is the size of the comparison window.

Let us take a look at an example: if the input word is *tree* which has a word length of 4. Thus, the bigram approach is selected. A simple example of the output vector is as follows:

The first bigram of *tree* is "tr". "tr" is thus left-aligned to the CMM. It is compared with the first bigram of each word in the CMM. Thus, an output vector of [011021] is produced with the threshold set at 2. The fifth word in the CMM, *tree* has an output activation of 2 which indicates that it is a complete match to the first bigram of the input word. In this round of matching, an output bit vector of [000010] is produced. Now "tr" of the input word is shifted right one character and input to the CMM. An output vector of [000000] is produced. The resultant output vector for the first bigram is [000010]. When we matched all bigrams {tr, re, ee} from the input word, there are three resultant output bit vectors ([000010], [000011], and [000011]) representing the words that have matched each bigram respectively. The output of all three vectors are then summed and a vector [000032] is produced. The threshold is set at 3 which is the maximum value of the integer vector in order to find the best matching word. The output vector [000010] corresponding to the highest activation value is then passed on to the lexical token converter to retrieve the matching word which is "tree" in this case. Notice that here 4-bit chunks are used for simplicity.

For more details on the Hamming distance and n-gram approaches used for spell checking and correcting in AURA, please consult [HA02, HA03].

4.8.3 Phonetic checking and correcting

Apart from these two approaches, a phonetic coding approach is adopted to check and correct phonetic spelling errors. As mentioned earlier in this section, each word is transformed to a 4-character code by combining codes similar to that of Soundex [Kuk92] and that of Phonix [Gad90] with phonetic transformation rules derived from Phonix as well as Double Metaphone [Phi99] which is used in ASPELL. Fourteen phonetic codes indexed from 0 to D were used to preserve the letter pronunciation similarities. Each code is represented by a single character and each word is translated into a 4-character phonetic code. Only similar sounding words map to the same code. The code generated for each letter is given as follows:

letter	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	-	’	~	
code	0	1	-	2	0	3	4	0	0	4	5	6	7	8	0	9	-	A	B	C	0	D	0	-	0	B	0	0	0	0

Note that the letters c, q and x do not have phonetic codes because according to the transformation rules, they are always mapped to other letters. For example, c can be mapped to s or k. For details on the phonetic transformation rules applied to this method and the algorithm used for phonetic code generation, please refer to [HA03].

For phonetic spell checking, the phonetic codes form the inputs of the CMM and the outputs are the matches from the lexicon words. Each word is translated to a four-character code. A binary bit vector of length 62 is divided into 23 characters (entire English alphabet excluding the letters c, q and x) and three 13-bit chunks. Each of these three chunks represents a phonetic code according to the table illustrated above. The position of the bit set in each chunk is the hexadecimal value of the code. A binary bit vector is then produced by concatenating these three chunks. This binary bit vector represents the phonetic code of the input word for the CMM. The output of the CMM is formed from the unique orthogonal binary bit vector representation of each word in the lexicon which is the same as with the shifting n-gram and binary Hamming Distance approaches.

The recall from the phonetic method is similar to the binary Hamming distance recall. The input word is converted into a four-character code which is input into the CMM. A output vector representing the superimposed outputs of the matching words is recalled from the CMM. For exact matches, a threshold is set to the maximum output activation value in order to retrieve all lexicon words that best match the input word phonetically. For partial matching, the vectors of several word matches are superimposed in a single output vector according to a predefined threshold. Since all lexicon words are held in an array, the position of any bits set in the output vector corresponds to the position of that specific word in the array. The matched words are retrieved from the word array using the bits set in the output vector with a predefined threshold as its positional indices into the array.

The AURA system performs spell checking by starting with the Hamming distance and a length match approach as described earlier. If no exact matches are returned from this process, it is assumed the input word is spelled incorrectly. The input word is then checked against the lexicon and a list of suggested spellings is produced. The input vector for the binary Hamming distance approach is input into the first CMM. Then, the input vector for the shifting n-gram approach is input into the same CMM. Lastly, the input vector for the phonetic spelling approach is input into the second CMM. An individual output vector is generated for each approach with an activation vector for each lexicon word. The locations of the best matching words are identified from the positions of the bits set in the output vectors with their predefined thresholds. Three different scores are produced—one for the best match of each of the approaches. The three scores are calculated as follows:

$$\begin{aligned} n - gramScore &= 2 \times (t - (|w| - (L_i)) \\ hammingDistScore &= 2 \times (t - (|w| - (L_i)) - ((2 \times |n|) - 1) \\ phoneticScore &= 2 \times (t - (|w| - (L_i))/(2 \times |p|) \times (|w| - (|n| - 1)) \end{aligned}$$

where t denotes the threshold set for each approach, w denotes the input word, L_i denotes a lexicon word, n denotes the n in n-gram used, and p denotes the phonetic code of a word.

Note that the AURA modular network is not language-specific due to its bit vector approach. Thus, it can process any languages with a Latin alphabet. Only the phonetic codes and transformation rules will need to be adjusted according to the language. No morphological analysis was applied in AURA.

4.9 Spell Checkers/Correctors for the South African Languages

It is estimated that there are around 2,000 to 3,000 languages spoken in Africa with eleven of them being official languages in South Africa. In addition to English and Afrikaans, 9 other official languages come from the family groups of Sotho (Northern Sotho/Sesotho sa Leboa, Southern Sotho/Sesotho and Setswana), Nguni (Zulu/isiZulu, Xhosa/isiXhosa, Swati/SiSwati and Ndebele/isiNdebele), Venda/Tshivenda, and Tsonga/Xitsonga.

As noted by de Schryver and Prinsloo [dSP04], many techniques have been used in the past to spell check the South African languages. The first spell checkers for Xhosa, Zulu, Northern Sotho, and Setswana were developed by Prinsloo as components in WordPerfect 9 of the WordPerfect Office suite 2000 [PdS01]. Spell checking and correcting tasks are in fact much more complex when performing on the languages in the Nguni group than on the languages in the Sotho group.

One challenge of spell checking the South African languages heavily relates to the conjunctive nature of the languages of the Nguni group—phrases are made up of parts of words joined together.

Another challenge is that it is often the case that a text is written in various African languages, that is to say, a text contains more than one language. The pre-spell-checking process is to categorize and recognize each language involved. Umqageli is a text categorization tool developed by Maniacky [Man03] which recognizes and distinguishes various African languages such as Northern Sotho, Southern Sotho, Setswana, Venda, Xhosa, Swati, Zulu, Ndebele, in a document. It is based on calculating and comparing profiles of n-gram frequencies [CT94]. For the Nguni languages, it is often found that spell checking and correcting is achieved by dictionary lookup and n-gram analysis [dSP04].

Finite-state morphology is a technique used to detect and correct spelling errors, especially for the conjunctive languages. The Xerox finite-state tools [BK03] can be used for decomposing morphologically complex languages with a high level of conjunctiveness (i.e.

agglutination which was explained in Section 3.5). An example of using finite-state morphology to treat African languages can be found in [PB02]. A spell checker for Afrikaans was developed by van Huyssteen and van Zaanen [vHvZ03] based on morphological analysis which was briefly described in Section 3.5. Another technique that has been widely used to recognize more valid words is n-gram analysis. This technique has been described in detail in Sections 2.3.2 and 3.3.2. n-gram analysis is often used in conjunction with dictionary lookup techniques to achieve a more satisfying result.

In Chapter 5, we will be looking into the DAC Northern Sotho (DNS) spell checker in a form of a MS Word installer for MS Windows. The DNS spell checker was retracted from the website of the South African Department of Arts and Culture for development of future versions. Unfortunately, no literature has been located that publishes the details of the underlying techniques that were used for spell checking and correcting.

4.10 CInsunSpell — Chinese Spell Checking Algorithms

So far we have mostly discussed spell checking and correcting in English or Latin languages. It is rather interesting to find out how languages other than Latin-based ones perform the spell checking and correcting tasks, especially for a language such as Chinese, in which the word formation is not derived from 'spelling' in the conventional sense of western languages.

There exist two different Chinese writing systems, namely simplified and traditional. Nevertheless, the spell checking and correcting (or rather text proofreading) techniques do not differ [Cha94, LWS00, ZHZP00, LW02]. In this section, we will be discussing CInsunSpell, a spell checking and correcting algorithm and implementation for Chinese.

There are significant differences between Chinese and English [LWS00]:

1. Chinese does not have evident word boundaries, such as a white space character, as in English. Chinese characters are linked together to construct a sentence without any boundary characters in between each character. According to Li et al [LWS00], each Chinese word can consist up to four characters[7]. Thus, before any analysis can be performed on a Chinese text, it must be segmented into words.

2. The Chinese character set contains more than 6,700 characters. This language character set is hard on parameter calculation of some models such as Markov Models. As a result, likelihood path search and many other efficient techniques for Latin-alphabet based languages are not suitable for Chinese.

3. Chinese's input methods are different compared to the English letter-by-letter input. Chinese can only be keyed into a computer using a special code, such as encoding

[7]Some researchers claim that there exist words which consist up to seven characters because of different ways of segmenting words.

> the keystrokes, WuBi or Pinyin-input method. Each character is represented by a unique code [Uni03]. This implies that spell checking within each character is meaningless—rather, spell checking is on a word-level within a text. The smallest language unit in Chinese is a character and each word consists at least one character. One sentence consists of several words according to the grammar.

The Chinese input methods include input gased on encoding, pronunciation, or structure of the characters. In general, the Pinyin method is based on the Pinyin method of Romanisation (i.e. the Roman letters to represent sounds in Mandarin). The sound representations of the Roman alphabet for Mandarin differ from that for other languages. For example, 'g' corresponds closely to the sound of 'k' in English. Different countries adopt different Pinyin systems: for instance, the People's Republic of China adopted Hanyu Pinyin as opposed to Tongyong Pinyin which is used in Taiwan. The Pinyin method allows a user to input Chinese characters by entering the 'sound' or phonetic combination of a Chinese character. A list of possible characters with the same sound (i.e. have the same phonetic combination) is provided. The user then chooses the desired character from the list of sound-equivalent characters. For example, the Chinese equivalent of the word 'woman' would be entered as 'nu' as its phonetic combination in the Hanyu Pinyin system. For any further information on the Pinyin systems, please refer to [Tsa04].

There also exist insertion, deletion, substitution, and reversal spelling error types in Chinese. The error types are single-character insertion, single-character deletion, single-character substitution, single-character reversal, and string-substitution (i.e. 2- or 3-character substitution) errors. Note that, due to the dependency between characters in a word, it is not only possible to perform spell correcting on isolated-word errors, though improved results are obtained by spell correcting context-sensitively [ZHZP00]. Most errors occur in a Chinese text when characters have similar pronunciation (or homophones[8]), similar shape, similar meaning, or similar input keystroke sequence in the input method used.

CInsunSpell which was developed in 2002 is a more recent Chinese-specific spell checking and correcting system [LW02]. It achieves spell checking and spell correcting in two different processes. For spell checking, character trigrams (discussed in detail in Sections 2.3.2 and 3.3.2) within a fixed-size window are used to locate the misspelled words in a specific area. This process reduces memory consumption as the Chinese character set is relatively large. The use of trigrams for spell checking in English was proposed by Mays, Damerau and Mercer [MDM91].

Edit distance techniques [Lev66] (discussed in detail in Section 3.4.1) are adopted by CInsunSpell for spell correcting in order to obtain the information of word construction in a set (this set is called a confusion set).

A confusion set is constructed for situations in which a character in a word string is transformed into another character which is either confusable (e.g. homophones which have the same or similar pronunciation yet differ in spellings or meanings) or irrelevant, thus causing a misspelling. For instance, when a deletion error occurs, the original character in the input string is changed into a null character which results in a misspelling.

[8]Homophones have different meanings but the same pronunciation. An example of homophones in English, would be the words *see* and *sea*.

The confusion set is thus constructed to find a valid character to correct the misspelled character. The confusion set in CInsunSpell is constructed by collecting the 3,755 most frequently-used characters from the Chinese character set.

Weights are then automatically and dynamically distributed among these characters in the confusion sets. Bayesian models are involved in the weight distribution process. Different characters in a confusion set affect their adjacent characters differently in a text. Thus, different weights are assigned automatically according to the effects to these characters. A list of possible characters are then output as the suggestion results.

Due to the nature of the language, the accuracy of a Chinese word heavily depends on its relation with its adjacent characters. Errors occur when this relationship is disturbed. Thus, an approach is derived from tackling word-sense disambiguation[9] [NZ97] in order to find the position of this disruption in the relationship. In tackling word-sense disambiguation, statistical approaches, such as n-gram analysis, are used. The approaches start by defining a window of n words around each word to be disambiguated in the content and then statistically analyzing the n surrounding words. Bayesian models and decision trees are used for training and correcting. Another approach that is used recently is kernel-based methods (e.g. as support vector machines).

As mentioned previously, spell checking in CInsunSpell is performed based on character n-grams (more specifically trigrams). It starts by setting one fixed-sized sliding window W to locate an area where there may contain an error. There exist approximately k characters on either side of the centre character m in a window. n-gram analysis is applied to determine whether m is a misspelling. It is assumed that the fixed-size window is of size $2k+1$ where W can be represented as $W = \{c_1, c_2, \cdots, c_k, m, c_{k+1}, c_{k+2}, \cdots, c_{2k}\}$. The probability of the first k characters is calculated by the following equation:

$$P(c_1, c_2, \cdots, c_k) = \frac{N(c_1, \cdots, c_k)}{N(t_1, \cdots, t_k)}$$

where $N(c_1, \cdots, c_k)$ is the number of times the string $\{c_1, \cdots, c_k\}$ appears in the corpus and $N(t_1, \cdots, t_k)$ is the total number of any k-character string appears in the training corpus.

An example would be the following: a corpus of 100 characters would have 98 different groups of 3 successive characters. Maybe $c_1, \cdots, c_k =$ 'the', and say this string appears 5 times in the corpus. Then $\frac{N(c_1, \cdots, c_k)}{N(t_1, \cdots, t_k)} = 5/98$.

In practice, this probability is often estimated by using n-gram. Similarly, the probability of the last k characters is

$$P(c_{k+1}, c_{k+2}, \cdots, c_{2k}) = \frac{N(c_{k+1}, c_{k+2}, \cdots, c_{2k})}{N(t_{k+1}, t_{k+2}, \cdots, t_{2k})}$$

An assumption is made that the first k characters of W are correct and m is the character that requires checking.

CInsunSpell algorithm performs the spell checking task in the following steps:

[9]Word-sense disambiguation is to determine the correct/intended meaning or sense of a word in context.

- *Step 1.* CInsunSpell uses a fixed-size sliding window, $W =< c_1, c_2, c_3, c_4, c_5 >$, from the beginning of the text T, $k = 2$[10], and the centre word in this case is c_3. Trigrams are generated from the list of characters within the window. For example, the trigrams in this case would be $< c_1, c_2, c_3 >$, $< c_2, c_3, c_4 >$, and $< c_3, c_4, c_5 >$.

- *Step 2.* If $-\log(P(c_1, \cdots, c_k))$ and $-\log(P(c_{k+1}, \cdots, c_{2k}))$ individually are greater than the pre-defined thresholds for the front (i.e. $< c_1, \cdots, c_k >$) and the back (i.e. $< c_{k+1}, \cdots, c_{2k} >$) strings respectively, then c_3 is reported as a misspelled character. From the experiments conducted by Li and Wang, the threshold for the front string is best set at 19.8720 and the threshold for the back string is best set at 20.5650 [LW02].

- *Step 3.* If the negative log values are less than or equal to the pre-defined thresholds, bigrams instead of trigrams will be generated from the list of characters in the window. This is because the higher the rank of n-gram, the more sparse the data and the less accurate the probability. Thus, in order to increase the performance of the algorithm, a smoothing procedure may be invoked to handle sparse data. The smoothing procedure takes the window W of trigrams and reduces it to a window of bigrams, namely $WB =< c_2, c_3, c_4 >$ (c_1 and c_2 are not considered as they are assumed to be correct). The relationship of characters in each bigram is examined. In other words, the negative log values of $< c_2, c_3 >$ and $< c_3, c_4 >$ are compared against the pre-defined thresholds in the same manner as described in the previous paragraph. Furthermore, in order to reduce the sparse data in the bigram situation, the window WB is extended by one more character, i.e. $< c_2, c_3, c_4, c_5 >$. Recall the assumption made earlier that the first k characters of the window are correct. Thus, if $k = 2$, the first two characters which form the first bigram, $< c_2, c_3 >$, are assumed to be correct. $< c_3, c_4 >$ is the suspected bigram (i.e. containing a possible misspelling). The relation between c_4 and c_5 is evaluated in order to determine if c_4 is a misspelling. The relationship between two characters is once again evaluated by their probability. Thus, three pre-defined thresholds (i.e. probabilities for the front string, the back string, and the extension) are required for the evaluation.

- *Step 4.* After examining $W =< c_2, c_3, c_4, c_5 >$, the detection process moves on to the next fixed-size window, i.e. $< c_3, c_4, c_5, c_6 >$. The process repeats itself from *Step 2.* c_2 and c_3 are now assumed to be correct in this case.

One must take into account that the smoothing method (i.e. the use of bigrams instead of trigrams) is also used to overcome the situation where one of the first two characters is an unintended character, such as the Chinese equivalent of the symbol ”, and this usually happens at the beginning of a text. However, there exists an exception that the earlier assumption that the first k characters of the window are correct cannot be applied here. Another way of dealing with smoothing the sparse data is to construct a small thesaurus lexicon containing only one-character words (usually parts of speech). When sparse data

[10]According to [LW02], their experimental results have shown that when $k = 2$ and when the language model is represented by trigrams, the result is optimal.

occurs, the algorithm can look up a character in the lexicon for a replacement character before rechecking. Details on the smoothing method can be found in [LW02].

The next step is to correct the misspellings detected by the spell checking process described earlier. Firstly, the Bayesian formula [MDM91, Win03] is employed to build the language model for spelling correction in order to find the candidate suggestions to correct the spelling errors reported in the previous spell checking process. To find the closest string d for the misspelled word m, where w is a correctly-spelled word, Li and Wang [LW02] made use of the following equations:

$$d = \overset{argmax}{w} P(w|m)$$

$P(w|m)$ is computed from the Bayes' rule:

$$P(w|m) = \frac{P(w)P(m|w)}{P(m)}$$

where

- $\overset{argmax}{w}$ refers to the value of w for which $P(w|m)$ has the largest value;
- $P(w)$ is the probability of the occurrence of w; and
- $P(m|w)$ is the probability of a correctly-spelled word w transformed to the misspelled word m as a result of one of the characters in w is wrongly spelled.

The value of $P(w)$ is estimated using trigrams, i.e.:

$$P(w) = \prod_{i=1}^{n} P(w_i|w_{i-1}, w_{i-2})$$

where n is the length of w and w_i is an individual character in w.

The value of $P(m|w)$ is estimated from the prior probability of the individual characters:

$$P(m|w) = \prod_{i=1}^{n} P(m_i|w_i)$$

where m_i is an individual character in m. Thus, $P(m_i|w_i)$ is the probability of w_i transformed to m_i. In other words, it reflects the likelihood that w_i may be wrongly spelled to m_i.

All the characters in the confusion set are possible candidates for correcting the misspelled characters detected. $P(m_i|w_i)$ as mentioned above is the probability of a character in the input string w_i being transformed to a character in the misspelled word m_i and it reflects the relation between the characters. This probability is the weight of the trigram model $P(w_i|w_{i-1}, w_{i-2})$, which is used to determine the probability of occurrence of a correctly-spelled word.

Minimum edit distance as discussed in Section 3.4.1 has been adopted to perform automatic weight distribution in order to distinguish the relation between each character pair

(m_i, w_i). As explained earlier, there exist 1-character words and multi-character words (i.e. 2-character words, 3-character words, and 4-character words) in Chinese. According to Li and Wang [LW02], one of the characteristics of Chinese words is that the more characters that a word is made up of, the more closely related these characters are. Thus, for example, the characters in a 4-character word are much more closely related compared to the characters in a 2-character word. The weight distribution of characters in the confusion set thus relies on the length of the words (i.e. the number of characters each word contains) that are constructed by the current character in the confusion set with its surrounding ones in some text.

The procedure of automatic weight distribution is as follows:

- *Step 1.* Minimum edit distance is used in CInsunSpell to determine the possibilities of some characters in the confusion set to become valid suggestion candidates. For instance, there exists a text $T =< w_0, w_1, \ldots, w_n >$ (here the word order in the text is important and we view T as a sequence) and w_3 is a misspelled character. w_{3i} is a character which is proposed to be a candidate suggestion for w_3. Substitute w_{3i} into each possible four-character words (in order to obtain the highest degree of association between characters) using minimum edit distance and we get $< w_0, w_1, w_2, w_{3i} >$, $< w_1, w_2, w_{3i}, w_4 >$, $< w_2, w_{3i}, w_4, w_5 >$, and $< w_{3i}, w_4, w_5, w_6 >$.

- *Step 2.* Each of these words is matched against words in the dictionary. If matches are found, it indicates that the misspelled words that contain w_3 are one edit distance away from the correctly-spelled words found in the dictionary. w_{3i} is categorised into the four-character word class in the confusion set. If any of the words cannot find matches, the words are shorten to contain one character less and matching is processed. For example, if a match was not found for any of the four-character sequences in *Step 1*, then these sequences are then shorten to three-character words: $< w_0, w_1, w_2 >$, $< w_1, w_2, w_{3i} >$, $< w_2, w_{3i}, w_4 >$, $< w_{3i}, w_4, w_5 >$, and $< w_4, w_5, w_6 >$. Words, such as $< w_0, w_1, w_2 >$ and $< w_4, w_5, w_6 >$, will not be processed as they do not contained w_{3i} the possible candidate suggestion for the misspelled word w_3.

- *Step 3.* *Step 2* is repeated until all words that can be find matches in the dictionary have been tested and until each word contains only one character.

- *Step 4.* The candidates of w_3 in the confusion set are categorised into four different classes, namely Class_4, Class_3, Class_2, and Class_1 and they represent 4-character word class, 3-character word class, 2-character word class, and 1-character word class respectively. Candidates in Class4 are assigned the maximum weights (due to the highest degree of association between characters) and those in Class_1 are assigned the minimum (due to the lowest degree of association between characters). Weights for candidates in the same class are equal. In case where the current character can form more than one multi-character word with its neighbouring characters, it is categorised into the class with the maximum word length.

The weight of each class, calculated in terms of a so-called $BasicWeight$, is determined as follows:

$$WeightClass_1 = BasicWeight$$

$$WeightClass_2 = 5 * BasicWeight$$

$$WeightClass_3 = 10 * BasicWeight$$

$$WeightClass_4 = 20 * BasicWeight$$

where

$$BasicWeight = \frac{1}{(5 * n + 10 * m + 20 * r + (3755 - n - m - r))}$$

In this formula, n is the number of characters in the 2-character words, m is the number of characters for 3-character words, and r is the number of characters for 4-character words. Refer to [LW02] for an explanation of why these various constants have been selected for the calculation of the weight of each class.

CInsunSpell thus achieves the spell correcting task by combining the language model construction processes (applying Bayes' rule, minimum edit distance technique, and the weight distribution calculation). The best suggestion candidates are output from Class$_4$ which is followed by suggestions from Class$_3$, Class$_2$, and Class$_1$, respectively. The suggestions are output in descending order according to the possibilities of these candidates.

For candidate characters within words with the same length (i.e. in the same class), the sound and shape similarities of the characters are then taken into account to determine the accuracy of the candidate suggestions. Further details of this process can be found in [LW02].

It is observed that although the language structures of English and Chinese are rather different, the spell checking and correcting techniques do not differ significantly. The techniques mentioned in Chapter 3 such as n-grams (Section 3.3.2), minimum edit distance (Section 3.4.1), and probabilistic techniques (Section 3.4.5) are also employed for Chinese spell checking and correcting in CInsunSpell. Morphological analysis can be applied to Oriental languages. However, CInsunSpell adopted the n-gram analysis approach instead as it is less complicated and more time efficient compared to morphological analysis.

4.11 Conclusion

Spell checking in the mentioned spell checkers and correctors mostly rely on dictionary lookup techniques such as the ones discussed in Section 3.3. The spell correcting techniques discussed in Section 3.4 are widely used in both spell checkers and correctors. Each of these techniques may be used on its own or in conjunction with the other techniques. In more recent development, an algorithmic solution for the spell checking and correcting tasks often comprises of a combination of techniques. This is because one single technique can be inefficient on its own. Combining the techniques aims at achieving the optimal results in terms of accuracy, running time and/or storage space.

As we can see from this chapter, in the earlier stage of spell checking and correcting research, dictionary lookup techniques have been the main focus and minimum edit distance techniques have been widely used in the spell checking and correcting algorithms for languages based on Latin alphabet. n-gram-based techniques are popular amongst the more complex language structures such as Zulu where the language has a conjunctive nature and Chinese where the definition of a word boundary is not obvious. Probabilistic techniques only later join in as a solution to the spell checking and correcting problems. n-gram-based techniques are inseparable from probabilistic approaches. Probabilistic techniques are used in conjunction with dictionary-lookup techniques to achieve higher accuracy. Finite-state automata were generally used in conjunction of edit distance technique for spelling correction in a highly inflectional or agglutinating language or a language with compound nouns.

Some of the various algorithms described in this chapter will be used in the classification in Chapter 5. We will be comparing these algorithms in terms of characteristics, functionalities and empirically-determined performance in the following chapter.

Chapter 5

Empirical Investigation and Classification

5.1 Introduction

This chapter provides the comparison and classification of various spell checking and correcting algorithms described in Chapter 4. These algorithms were matched up with each other in terms of their performance, functionalities, and implementation strategies.

For comparison in performance, Unix® SPELL (Section 4.2), ASPELL (Section 4.4), AGREP (Section 4.7), FSA PACKAGE (Section 4.6), MS Word 2003, DAC Northern Sotho (or Sesotho sa Leboa) spell checker (Section 4.9), and CInsunSpell (Section 4.10) were use. Experiments on SPELL, ASPELL, FSA PACKAGE, and AGREP were conducted on a Linux platform (Mandrakelinux 10.1) and the rest on Windows (XP Professional 2002). The particular hardware platform used to perform the experiments described in this chapter was an Intel Pentium D at 2.8GHz, with 512MB of RAM and 40GB of hard drive. For more detail on the processor, refer to [Cor06, Gov01]. Of course, the algorithms' performance may vary from one hardware platform to another and, to this extent, the results below should be regarded as indicative rather than conclusive. This machine was equipped with both the Windows and Linux operating systems. Performance was evaluated with respect to spelling error detection, suggestion accuracy (for packages that provide suggestions), and efficiency.

Four English-based spell checking and correcting packages tested were SPELL, ASPELL, FSA PACKAGE, and AGREP—all open source products. The seventh version of SPELL can be found in the Unix repository [Rit06]. ASPELL is downloadable from GNU's website [Atk04]. The current version (version 0.44) of FSA PACKAGE is available for download from `http://www.eti.pg.gda.pl/katedry/kiw/pracownicy/Jan.Daciuk/personal/fsa.html#FSApack`. AGREP is accessible via the ftp portal of University of Arizona [WM91, WM92b]. The performance of these packages was evaluated with main focus on spelling error detection and suggestion accuracy. The packages that run on the Linux platform were also evaluated in terms of time efficiency.

MS Word 2003 within MS Office suite was also selected as part of the evaluation, because of its wide usage and ready availability, even though information about the techniques applied are not publicly available. However, the comparison of MS Word 2003 against the three aforementioned packages has its focus on accuracy only. This is due to the fact that MS Word 2003 is running on a different platform compared to the other four selected systems. The functionalities and implementation strategies of all the algorithms described in the previous chapter will be closely compared in Section 5.5.

To provide a contrasting perspective, two non-English based spell checking and correcting packages were also investigated. The first was DAC Northern Sotho (DNS) Spell Checker that can be installed into MS Word[1]; and CInsunSpell for Chinese, an experimental Chinese spell checker/corrector developed at the Harbin Institute of Technology [LW02]. CInsunSpell was the only spell checker involved in the experiments that performs spell checking/correcting tasks on a non-Latin-based language. Both of these spell checkers run on Windows only.

CORRECT (Section 4.3), unfortunately, was not part of the testing as it is out of trace according Church [CG91]. It was part of some Bell Laboratories' products at one stage. SPEEDCOP (Section 4.5) and AURA will also not be part of the testing as SPEEDCOP has been patented and used as part of some commercial products, whereas AURA has been licensed to Cybula Ltd. as the underlying method for various commercial software and hardware applications.

In designing our experiments, we chose to rely on controlled data instead of open data for these experiments. The data set used for each experiment is described in detail in each section.

5.2 Spell Checking and Correcting for English

5.2.1 Data Sets

Initially, only one set of test data was involved in the experiments for spell checking and correcting packages for English. The data set was a list of 15,000 most frequently-used words, extracted from the British National Corpus (BNC) [BNC05]. This list is comprised of 111,333 characters. BNC consists of over 100 million words which is approximately equivalent to the full text content of 1,000 books. The word list extraction was performed using the Oxford WordSmith Tools 4.0 [Pre06]. This data set consists of 15,000 correctly spelled words including atomic lexical morphemes (e.g. book), compound words (e.g. newspaper), words with derivational morphemes (e.g. stabilise), and morphologically complex words (e.g. unsatisfied).

Additionally, a total of 1,000 unique misspellings were extracted from the Birkbeck Spelling Error Corpus [Mit85] (This corpus consists of 36 files and has a total size of approximately 1.8GB.) Thus, the initial test set comprised 16,000 words in total.

[1]Note that Northern Sotho is one of the more prominent indigenous South African Languages. The spell checker was commissioned in 2003 by the South African Department of Arts and Culture. The original version used in this study has been withdrawn from the public domain, while an update is currently in production.

Out of the 1,000 spelling errors, 742 of them were identified as single-letter errors, 201 were two-letter errors, 44 were three-letter errors, 9 were four-letter errors, and the remaining 4 were in the category of five-letter errors. The single-letter errors were further subdivided according to the spelling error types [Kuk92]. Thus, 230 of them were identified as insertion errors, 188 deletion, 290 substitution, and 34 transposition errors. The data set that consists of the 15,000 correctly spelled words from the BNC and the 1,000 misspellings mentioned here will be referred to as $\mathbf{DS}_1$ throughout this chapter.

To further verify the test results of both spell checking and correcting, a sanity check for consistency was required. Hence, a second list of 1,000 spelling errors was extracted from Birkbeck Spelling Error Corpus. Each of these misspellings was different from the ones in the previous data set. Amongst these 1,000 spelling errors, 734 misspellings were single-letter errors, 198 were two-letter errors, 45 were three-letter errors, 16 were four-letter errors, and 7 were five-letter errors. The 734 single-letter errors were comprised of 237 insertion, 183 deletion, 288 substitution, and 26 transposition errors. The data set that consists of the 15,000 correctly spelled words from the BNC and the second list of spelling errors mentioned in this paragraph will be referred to as $\mathbf{DS}_2$ throughout this chapter.

The clear correspondence between the characteristics of the first and second sample of spelling errors, suggests that they could both be used with a reasonable degree of confidence that they each reflect the sorts of spelling errors that are likely to occur in practice, under the assumption that the Birkbeck Spelling Error Corpus itself accurately reflects the extent and range of such errors.

In order to measure the performance of the morphological intelligence in SPELL and FSA PACKAGE (as they are the only two packages that are equipped with morphological analysers), two extra data sets were used. The first data set consists of 15,000 correctly spelled words extracted from the BNC as well as 500 misspelled words, where misspellings occurred only in the stems of the words (e.g. reach). The second data set consists of 15,000 correctly spelled words extracted from the BNC as well as 500 misspelled words with misspellings occurring only in morphologically complex words (e.g. unreachable). These two data sets will be referred to as $\mathbf{DS}_3$ and $\mathbf{DS}_4$ respectively throughout this chapter.

5.2.2 Performance

In this part of the dissertation, we discuss the experiments conducted on various spell checking and correcting packages for English using the data sets discussed in Section 5.2.1. The performance is measured in terms of spelling error detection, spelling error correction, and time efficiency. We relied on controlled data instead of open data for these experiments. SPELL, ASPELL, FSA PACKAGE, and MS Word 2003 were all equipped with their standard supplied dictionaries. These dictionaries are used for a comparison against the other systems. However, AGREP itself does not contain a built-in or supplied dictionary. Thus, the standard dictionary in SPELL which consists of 29,197 words is adopted to be used for the comparison purposes. Various experiments were conducted to compare the

	SPELL	ASPELL	FSA PACKAGE	AGREP	MS Word
Total Errors in Data	1000	1000	1000	1000	1000
Total Actual Errors Detected	1000	992	1000	1000	999
Incorrectly Detected	0	0	0	0	0
Recall Rate (%)	100	99.2	100	100	99.9
Precision Rate (%)	100	100	100	100	100

Table 5.1: Spell checking results for English using the first data set ($\mathbf{DS}_1$).

performance of these packages with the primary focus on their accuracy. After experimenting with $\mathbf{DS}_1$, we evaluated the performance in the same manner using $\mathbf{DS}_2$. $\mathbf{DS}_2$ was used as a sanity check to verify the validity of the experimental results obtained from the first series of experiments.

Spelling Error Detection

As mentioned in the earlier chapters of this dissertation, the essential task of spelling error detection is to compare what is given against what is known, and there are a variety of techniques for doing so. Data structures may differ across techniques in respect of how to efficiently store what is known—various dictionary structures and/or statistical information. They may also differ in respect of how to represent input—in raw input form versus some morphological characterization.

In this subsection, we look at the performance with respect to the spell checking abilities of the following packages: SPELL, ASPELL, FSA PACKAGE, AGREP, and MS Word 2003. SPELL and ASPELL rely on hashing (refer to Sections 4.2 and 4.4, respectively). FSA PACKAGE made use of finite-state automata and cut-off edit distance (refer to Section 4.6). AGREP opted for a string-matching algorithm, called the bitap algorithm (refer to Section 4.7). While this algorithm can be used for approximate string matching, it clearly has to be used in exact matching mode for spelling error detection. As mentioned previously, it is not known which technique is used by MS Word.

Each of the these packages is executed with the initial set of test data to determine the number of misspelled words with the misspellings ranging from one to five error combinations. For spelling error detection, the number of times each type of error (i.e. single-letter, two-letter, three-letter, four-letter, or five-letter spelling errors) occurs is recorded. The performance of these packages was measured by the percentage recall rate, i.e. the number of genuine misspellings detected (as opposed to correctly spelled words incorrectly identified as misspellings) as a percentage of the number of misspellings in test data.

The experimental results based on $\mathbf{DS}_1$ are summarised in Table 5.1.

Note that the recall rate is an indication of the rate of accurate error detection in percentage terms, i.e. the total number of misspellings detected over the total actual number of errors in test data as a percentage. Thus,

$$\text{error detection recall rate} = \frac{\text{number of misspellings detected}}{\text{actual number of errors in test data}} \times 100$$

	SPELL	ASPELL	FSA PACKAGE	AGREP	MS Word
Insertion Errors	230	227	230	230	230
Deletion Errors	188	188	188	188	187
Substitution Errors	290	288	290	290	290
Transposition Errors	34	34	34	34	34
Total	742	737	742	742	741

Table 5.2: Single-letter errors detected using ($\mathbf{DS}_1$).

	SPELL	ASPELL	FSA PACKAGE	AGREP	MS Word
Total Errors in Data	1000	1000	1000	1000	1000
Total Actual Errors Detected	1000	996	1000	1000	998
Incorrectly Detected	0	0	0	0	0
Recall Rate (%)	100	99.6	100	100	99.8
Precision Rate (%)	100	100	100	100	100

Table 5.3: Spell checking results for English using the second data set ($\mathbf{DS}_2$).

The precision rate was determined by ratio (expressed as a percentage) of the number of misspellings and the number of errors detected, whether these be actual errors detected or incorrectly detected errors. Thus,

$$\text{precision} = \frac{\text{number of misspellings detected}}{\text{number of actual errors detected} + \text{number of incorrectly reported errors}} \times 100$$

The single-letter errors reported in Table 5.1 have been further identified according to the spelling error types in Table 5.2.

SPELL, FSA PACKAGE , and AGREP achieved the highest (100%) accuracy. MS Word performed second best, missing out on a word that contained a single-letter error. ASPELL performed the poorest. It failed to detect five of the single-word misspellings and three of the two-letter misspellings. Note that in all cases, 100% precision was achieved, i.e. it was never the case that a correctly spelled word was incorrectly identified as a misspelling.

As a sanity check for consistency, the above experiment was repeated with $\mathbf{DS}_2$. $\mathbf{DS}_2$ is of the same size as $\mathbf{DS}_1$. The details regarding the content of $\mathbf{DS}_2$ can be found in Section 5.2.1. The results of the experiments are summarised in Table 5.3.

The findings here correspond to the previous experiment: SPELL, FSA PACKAGE , and AGREP achieved 100% recall, MS Word failed to recall two misspelled words, which include one single-letter error and one two-letter error, and ASPELL failed to recall four misspelled words which include two single-letter error, one two-letter error, and one three-letter error (thus performing slightly better than before). Of course, each package performed at 100% precision again, because no new correct words were added to $\mathbf{DS}_2$.

Spell checking heavily depends on the dictionary supplied or used by the spell checkers and correctors. The results of the above experiments might have been different if a different dictionary had been installed for each of the selected packages. One of the main differences between ASPELL and the other English spell checking packages is that ASPELL

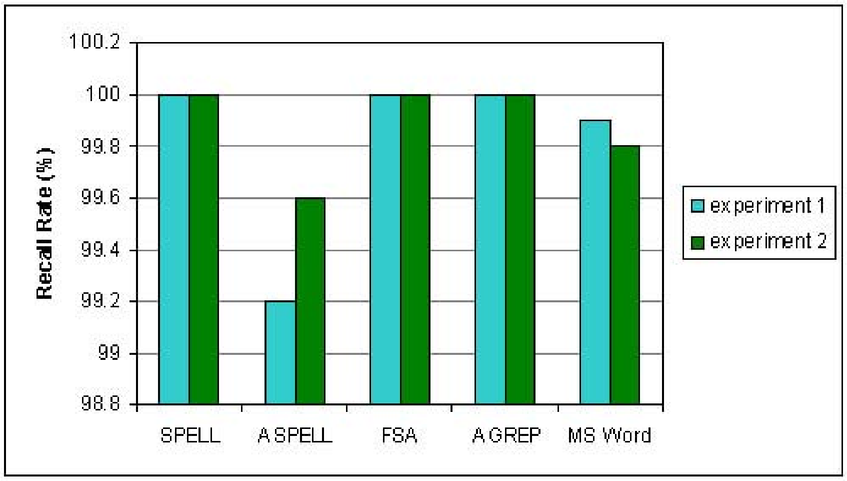

Figure 5.1: Spell checking recall rates for spell checking/correcting packages for English.

partially depends on a large rule-based system, the Metaphone algorithm, in order to handle phonetic misspellings (refer to Section 4.4).

It is rather interesting to see that although MS Word is a commercial product, its spelling error detection is not as effective as the other open-source packages, except ASPELL. The contents of the dictionaries used by the investigated packages are important factors that may affect the accuracy of spelling error detection. The dictionaries used by SPELL and FSA PACKAGE contain more words and superior affix analysis schemes compared to the dictionary used by ASPELL. They may also contain more words and superior affix analysis schemes compared to the dictionary used by MS Word. Thus, we can conclude that the dictionaries used by the two investigated packages are deficient compared to the ones supplied by SPELL and FSA PACKAGE .

Furthermore, it is of interest to determine how morphological analysis influenced the performance of those spell checking and correcting packages, which are equipped with a morphological analyser or affix stripping scheme. The packages that incorporate some form of morphological analysis are SPELL, FSA PACKAGE , and MS Word (per assumption). However, morphological analysis only occurs in FSA PACKAGE when it performs the spell correcting task. Two extra data sets $\mathbf{DS}_3$ and $\mathbf{DS}_4$ (described in Section 5.2.1) were required for the measurement of the morphological intelligence of these packages.

The first experiment was conducted using $\mathbf{DS}_3$. Both SPELL and MS Word achieved 100% recall rate (i.e. all errors were detected) with this experiment. The second experiment was conducted using $\mathbf{DS}_4$. Again, both SPELL and MS Word achieved 100% recall rate with this experiment. The results of these two experiments indicate that the morphological intelligence of these two packages is at a similar level.

Spelling Correction

In this subsection, the spell correcting accuracy measurements were carried out on all previous packages — SPELL excepted since it does not have spell correcting abilities. For

	ASPELL	FSA PACKAGE	AGREP	MS Word
Found in the First 5 Suggestions	844	908	741	886
Found in the First 10 Suggestions	903	908	820	904
Total Correction Suggestions	950	908	835	904
Total Errors Detected	992	1000	1000	994
Suggestion Accuracy (%)	95.77	90.80	83.50	90.95

Table 5.4: Spell correction results for English using $\mathbf{DS}_1$.

spell correcting in English, ASPELL, FSA PACKAGE , AGREP, and MS Word 2003 were tested. Again, two sets of test data, $\mathbf{DS}_1$ and $\mathbf{DS}_2$ (as described in Section 5.2.1), were used to perform spelling suggestion. The experiment results based on $\mathbf{DS}_1$ described in Section 5.2.1 are given in Table 5.4.

The first two rows show the number of times that each spell checking/correcting package had the correct spelling in the first 5 and first 10 suggestions, respectively. First 5 matches suggest that the correctly spelled word for a particular misspelling can be found in the first five candidate suggestions provided by the package. Similarly, first 10 matches suggest that the correctly spelled word for a particular misspelling can be found in the first ten candidate suggestions (including the ones in the first 5 matches) provided by the package. Due to the fact that FSA package is an automatic corrector (i.e. provided only one suggestion per misspelling), the results for the first two rows remained unchanged.

The suggestion accuracy is an indication of the ratio of the total number of correct suggestions to the total number of errors detected in percentage terms. Thus,

$$\text{error correction recall rate} = \frac{\text{number of correct suggestions}}{\text{number of misspells detected}} \times 100$$

By this measure, ASPELL offered the best correction performance while, in contrast to the spell checking results, AGREP performed the worst amongst the English packages. The performance of AGREP is directly linked to the dictionary or source file that the misspellings are matched against. If the words contained in the chosen dictionary or source file are not words that are in the same category/level as the misspelled words, the detection rate could be affected. Recall that ASPELL uses the Metaphone algorithm in conjunction with the reverse edit distance technique for spell correcting. This appears to commend the use of phonetic-based algorithms for spell correcting. Of course, the correction rate as a measure should be treated with a degree of caution since it will reflect well on a package that detects very few errors but offers correct suggestions for all of them. To some extent, this plays a role in ASPELL although the raw number of total correct suggestions provided by ASPELL (950) exceeds that of all rival packages.

Out of the total number of correct suggestions in each case, 88.84% of the correct suggestions were found in the first 5 candidate suggestions using ASPELL, 100% were found by FSA PACKAGE , 88.74% were found using AGREP, and MS Word was able to provide 98.01% of the correct suggestions in the first 5 candidate suggestions. These figures yield an average of 93.90% of the correct suggestions that can be found in the first 5 suggestions provided by the four packages.
An average of 98.31% of the correct suggestions can be found in the first 10 candidate

	ASPELL	FSA PACKAGE	AGREP	MS Word
Found in the First 5 Suggestions	755	892	609	755
Found in the First 10 Suggestions	769	892	98.86	770
Total Correct Suggestions	799	892	704	770
Total Errors Detected	992	1000	1000	994
Suggestion Accuracy (%)	95.12	89.20	83.81	91.67

Table 5.5: Spell correction results for English using $\mathbf{DS}_2$.

suggestions (including the ones found in the first 5 matches). This is an indication that the English packages are efficient spelling correctors.

As a sanity check for consistency, the above experiment was repeated with $\mathbf{DS}_2$. $\mathbf{DS}_2$ is of the same size as $\mathbf{DS}_1$. The details regarding the content of $\mathbf{DS}_2$ can be found in Section 5.2.1. The results of the experiments are summarised in Table 5.5.

By this measure, ASPELL again offered the best correction performance and AGREP again performed the worst amongst the spell checking/correcting packages for English. This experiment showed a slight improvement (increased by 0.31%) in AGREP's performance, MS Word 2003 also has a slight improvement in its performance (by 0.72%), and ASPELL and FSA PACKAGE performed slightly worse (decreased by 0.65% and 0.60% respectively). However, due to the fact that these figures are very close to the ones obtained from involving $\mathbf{DS}_1$, the sanity check was successful.

As mentioned previously, ASPELL partially depends on the Metaphone algorithm [Phi00]. In theory, it should therefore cope well with phonetic errors. The experimental results offer a hint of evidence in this direction. It turns out that there are approximately 10% fewer phonetic spelling errors in $\mathbf{DS}_2$ than in $\mathbf{DS}_1$, possibly accounting for ASPELL's slight performance dip in the second case.

Out of the total number of correct suggestions in each case, 94.49% of the correct suggestions were found in the first 5 candidate suggestions using ASPELL, 100% were found using FSA PACKAGE , 86.49% were found using AGREP, and MS Word was able to provide 98.05% of the correct suggestions in the first 5 candidate suggestions. These figures yield an average of 94.76% of the correct suggestions that can be found in the first 5 suggestions provided by the four packages.

An average of 98.78% of the correct suggestions can be found in the first 10 suggestions (including the ones found in the first 5 matches). Once again, this is an indication that the English packages are efficient spelling correctors.

Figure 5.2 shows how the error correction performance of the various spell checking/correcting packages match up with each other. Note that the rates for these English-based packages were obtained from the experiments conducted using both sets of test data.

It is of interest to determine how morphological analysis influenced the performance of the spell checking/correcting packages, which are equipped with a morphological analyser or affix stripping scheme. The spell correcting packages that incorporate some form of morphological analysis are FSA PACKAGE and MS Word. Two extra data sets, $\mathbf{DS}_3$ and $\mathbf{DS}_4$ (described in Section 5.2.1), were required for the measurement of the morphological intelligence of these two packages.

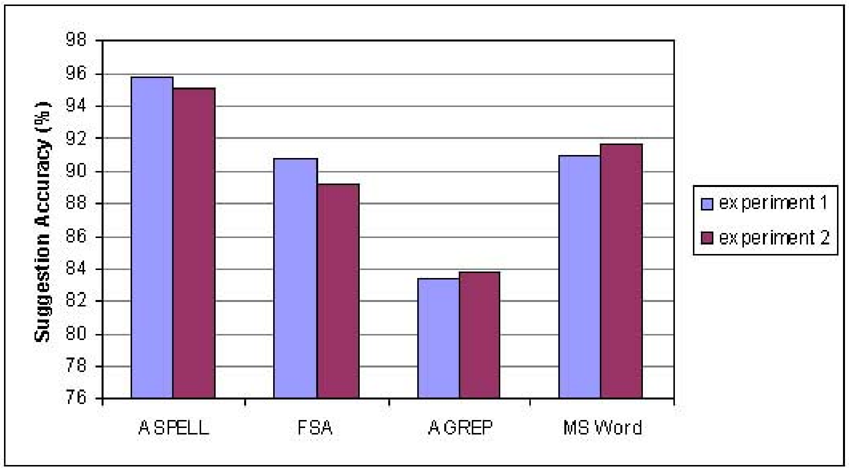

Figure 5.2: Suggestion accuracy for various spell checking/correcting packages for English.

DS$_3$ consists of 15,000 correctly spelled words and 500 misspelled words and misspellings occurred only in the word-stems. FSA PACKAGE achieved 86.60% correction accuracy. No correct suggestions were returned for 67 of the 500 errors. MS Word achieved a much higher correction accuracy (98.40%). Only 8 out of 500 errors were not provided with a correct suggestion and these misspellings are mainly phonetic errors. 472 of the errors were provided with the correct suggestions as the first option on each list of suggestions. The experimental results suggest that MS Word has a more sophisticated spelling corrector compared to FSA PACKAGE .

DS$_4$ consists of 15,000 correctly spelled words and 500 misspelled words, where misspellings occurred only in morphologically complex words. FSA PACKAGE achieved 85.20% correction accuracy. No correct suggestions were returned for 74 of the 500 errors. MS Word achieved a much higher correction accuracy (99%). Only 5 out of 500 errors were not provided with a correct suggestion. 485 of the errors were provided with the correct suggestions as the first option on each list of suggestions.

Comparing the results based on **DS**$_3$ to the ones based on **DS**$_4$, the performance of MS Word has improved by 0.60%. This is in contrast to the performance of FSA PACKAGE (decreased by 1.40%). This suggests that the morphological intelligence of MS Word is higher than that of FSA PACKAGE since FSA PACKAGE performed worse when a morphologically more complicated data set was involved.

Efficiency

SPELL, ASPELL, FSA PACKAGE , and AGREP were evaluated in terms of time performance for the error detection task. Notice that MS Word 2003 was excluded in these experiments as it runs on Windows whereas the rest of the packages run on the Linux platform. In order to achieve accuracy and fairness of the experimental results, MS Word 2003 was not taken into consideration here.

DS$_1$ described in Section 5.2.1 was used. In each case, the average and standard deviation time, measured over 50 runs, has been determined. The results are presented

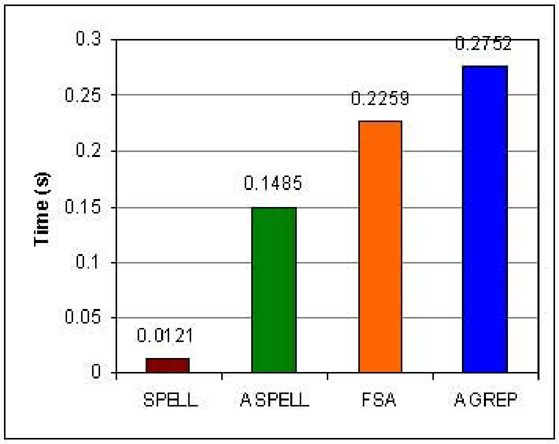

Figure 5.3: Mean of time performance of the spell checking/correcting packages for English.

in Figure 5.3.

The time performance measured for FSA PACKAGE excluded the time needed to build the dictionary into a finite automata (using fsa_build).

It is obvious that SPELL achieved the best performance amongst all the packages. On average, AGREP was approximately 23 times slower than SPELL, FSA PACKAGE was almost 19 times slower than SPELL, and ASPELL was approximately 12 times slower than SPELL. Clearly, hashing on its own, as used in SPELL and ASPELL, is far more effective in terms of efficiency than the combined algorithm used in the other packages. However, this added efficiency is traded off against the need by the other algorithms to provide for some sort of approximate matching (i.e. spelling suggestions) so that correction can later occur. Simple hashing does not support correction functionality. Hashing contributes to a constant time of $\mathcal{O}(1)$ and this is independent of the size of data. It is the fastest algorithm amongst all as we have already seen from our experimental results in this section.
ASPELL made use of hashing for dictionary lookup. As mentioned earlier, hashing contributes to a constant time of $\mathcal{O}(1)$. Theoretically, ASPELL should have a very similar time performance compared to SPELL. However, ASPELL was clearly slower than SPELL. It is conjectured that the reason of ASPELL being slower than SPELL has to do with the way in which Metaphone algorithm has been integrated into the overall ASPELL package, which provides more overheads to the entire package. Recall that the edit distance algorithms require a time complexity of $\mathcal{O}(mn)$, where m and n are the lengths of the source and target words, respectively, and the Metaphone algorithm has a time complexity of $\mathcal{O}(t)$, where t is the size of the table where all the phonetic rules are stored. Thus, the total time complexity for spelling correction using ASPELL is $\mathcal{O}(mn) + \mathcal{O}(t)$.

The time complexity of an automaton acceptor depends on the length of the string (i.e. the length of the word) being tested for acceptance. The automata acceptor in FSA PACKAGE has a time complexity of $\mathcal{O}(m)$, where m is the length of the word being searched.

The bitap algorithm in AGREP has a predictable time complexity of $\mathcal{O}(mn)$, where m is the length of the word being searched and n the size of the alphabet. The time complexity is a result of the bit array data structures of bitap and it is not affected by either the structure of the queried words or by the provided pattern.

AGREP was unexpectedly slow, in that the bitwise operation in the bitap algorithm which should theoretically enhance efficiency. It is conjectured that this relatively poor performance has to do with the way in which bitap algorithm has been integrated into the overall AGREP package, namely to provide for both spell checking *and* correcting. It would be interesting to investigate, as a matter of future research, whether a specialised spell checking package based on the bitap algorithm could be developed that rivals SPELL in efficiency. It would also be of interest to investigate the effect of the number of misspellings on time performance by fixing the number of misspellings in the documents.

5.3 Spell Checking and Correcting for Northern Sotho

5.3.1 Data Set

For evaluation of DNS Spell Checker (as an MS Word installer) for Northern Sotho, the TshwaneDJe Sesotho sa Leboa corpus [HLT06] was obtained from TshwaneDJe HLT, a company specialised in human language technology. It was used to make comparison against the supplied dictionary which is 628KB. The TshwaneDJe Sesotho sa Leboa corpus contains approximately 5.3 million words and is about 30MB in size. It derives from sources such as newspaper articles, magazine articles, short stories, and official documentations.

A list of 908 misspelled words was provided by the owner of the corpus. These misspelled words were identified from 65,000 words, which were manually scanned and extracted from the text collected. Amongst the 908 misspelled words, there were 804 single-letter errors, 78 two-letter errors, and 26 three-letter errors. The 804 single-letter errors were comprised of 570 insertion, 52 deletion, 156 substitution, and 26 transposition errors.

If this error rate (908 out of 65,000 words) is directly extrapolated, then one would have to conclude that almost 1.40% of the words in the TshwaneDJe Sesotho sa Leboa corpus contains spelling mistakes. This is considered to be a rather high error rate. One possible reason that could contribute to this figure is that some existing editing tools do not provide or support many characters required in these African languages. Hence, properly typing an article in an African language, such as Tshivenda, becomes a rather difficult job. It often requires a software driver, such as 'South African Keyboard' [Pro06] in order to find or insert the special characters required for a specific language. However, a full exploration of the reasons for the high error rate is beyond the scope of this dissertation.

5.3.2 Performance

Spelling Error Detection

DNS Spell Checker detected 882 (including 765 actual misspellings and 117 correctly spelled words that were incorrectly detected) out of 908 errors identified in the data, which represents a recall rate of 97.14%. DNS spell checker is clearly less mature than all the English spell checking/correcting packages and requires further fine tuning in terms of the content of the dictionary and morphological analysis scheme. Northern Sotho is much more complicated than English in regard to the morphological structuring of words. Thus, it is necessary to adopt a more sophisticated morphological analysis scheme. It failed to detect 26 misspellings. In addition, it reported 117 correct spellings as misspelled words, which translates to a rather low precision rate of 86.73%, compared to the previously discussed systems which all yielded 100% precision. This suggests that the dictionary simply does not have enough words in it. However, it was noticed that the majority of those words that were incorrectly detected as misspellings, were proper nouns—names of people or places.

Spelling Correction

For spelling correction, 81.18% correct suggestions were found in the first 5 matches provided by DNS Spell Checker and 85.49% were found in the first 10 matches. It was noticed that in more than 80% of the cases where spelling errors that were detected by DNS Spell Checker, there were fewer than five alternative spelling suggestions. 67 misspellings were not provided with any correct suggestions. Thus, the overall suggestion accuracy is 91.24%, which is fairly good, exceeding that of FSA PACKAGE , AGREP, and MS Word. However, the prior caution about using the suggestion accuracy uncritically as a measure, should be recalled. The low recall rate of DNS Spell Checker relative to that of the spell checking/correcting packages for English, tends to discount somewhat the value of its high suggestion accuracy. Unfortunately, details on the correction strategy used in DNS are not available as it is merely an MS Word installer. Nevertheless, it will be interesting to check the performance of the next version—currently under development—against the above results.

5.4 Spell Checking and Correcting for Chinese

So far we have only considered spell checking and correcting packages for the Latin-based alphabet (including African Reference alphabet, which is largely based on the Latin alphabet). It is of interest to find out how a package designed for non-Latin alphabet differs in performance.

According to Li et al [LWS00], each Chinese word can consist up to four characters. Furthermore, the notion of a blank space between words does not exist: characters simply appear, one after another. Thus, before any analysis can be performed on a Chinese text, it must be segmented into words. The smallest language unit in Chinese is a character and each word consists at least one character. One sentence consists of several words according to the grammar. Detailed background knowledge was provided in Section 4.10.

5.4.1 Data Set

For evaluation of the Chinese spell checking/correcting package, CInsunSpell, the training data was obtained from The Lancaster Corpus of Mandarin Chinese from the Oxford Text Archive [MX05]. It was used as the training set because it contains 15 different text categories, which makes diverse sources. The size of the training set was approximately 30MB. The data set was obtained from People's Daily 1993, 1994 [Dai94], and 2000 [Xia05]. The test data contains approximately 59,000 Chinese characters. A total number of 595 single-word[2] errors were identified, amongst which 82 were character-insertion errors, 119 were character-deletion errors, 316 were character-substitution errors, 18 were character-transposition errors, and 60 were string-substitution errors.

This finding suggests that approximately 1% of the characters in the corpus contain spelling mistakes. The character-insertion, deletion, and transposition errors often occur due to carelessness while typing. String-substitution errors occur either when the typist is not paying attention or if the typist lacks knowledge of some word spellings/formations. Character-substitution errors, however, mostly occur when the typist lacks knowledge of the correct spellings/formations of the relevant word. Approximately 80% of the 316 character-substitution errors were also phonetic errors. Thus, the cause of these errors is very likely due to the Chinese character input methods used for the corpus (as discussed in Section 4.10), especially the Pinyin input method.

5.4.2 Performance

Spelling Error Detection

CInsunSpell, was measured against a slightly different criteria than used for the Latin-based spell checking and correcting packages (in this case, the packages for English and Northern Sotho). Spelling error detection was performed at the character level within each word and also at the string (i.e. sub-word) level (refer to Section 4.10). CInsunSpell was executed to determine the number of real misspelled words, their error types (i.e. single-character insertion, single-character deletion, single-character substitution, and string-substitution errors), and the number of warnings reported amongst which incorrect warnings were identified. The experimental results are presented in Table 5.6.

In all but the last row, the final column in Table 5.6 indicates the percentage of column one ("Initial Error Identified") relative to column two ("Errors Detected"). Thus, the recall rate per error type varies from 55.55% in the case of transpositions, to 86.55% in the case of deletions. The overall recall rate of 80.84% is less than that of DNS Spell

[2]A single word can be anything from one to four character combinations.

	Initial Error Identified	Error Detected	%
Character Insertion	82	69	84.14
Character Deletion	119	103	86.55
Character Substitution	316	257	81.33
Character Transposition	18	10	55.55
String Substitution	60	43	71.67
Total Error Identified/Detected	595	481	80.84
Warnings	N.A.	666	72.22

Table 5.6: Spell checking results for CInsunSpell.

Checker (97.14%) and is far less than that of the spell checking and correcting packages for English.

The final row of Table 5.6, marked as "Warnings", refers to the 666 words that were flagged as errors. As is evident from the previous row, 481 of these words were genuine errors, while the remainder were "false positives". This results in a precision rate of 72.22%, which is the figure in the final column of the final row. This precision rate is less than that of DNS Spell Checker (86.73%) and is far less than that of the spell checking and correcting packages for English.

It is interesting to compare these results with those found by the authors of CInsun-Spell as reported in [LW02]. Two sets of test data were used. In the first, recall and precision rates[3] of 55% and 84%, respectively, were found. This data set appears to have an intersection with the data set used above (drawn *inter alia* from the People's Daily of 1993). In this case, their recall is lower than in the experiment described above, while their precision is somewhat higher.

In the second set of test data, their figure dropped to 33.33% and 37.44% for recall and precision, respectively, both considerably worse than our results above. In fact, their recall results are also somewhat worse than another system that they used for comparative purposes (the so-called HeiMa checker) while their precision results are better. The authors explained that high precision is of particular importance since constant false warnings of possible errors can become tiresome. However, it is not immediately evident why there is such a high variation between their results for the two different test sets.

The fact that CInsunSpell performs worse than the spell checking and correcting packages for English and DNS spell checker in spell checking is partially attributable to the complexity of the language structure concerned. It is also attributable to Chinese's input methods used for the corpus (as discussed in Section 4.10), especially the Pinyin input method.

Spelling Correction

For spelling correction, CInsunSpell correctly suggested 334 words and thus obtained a correction rate of 69.44%. This compares very closely with its reported correction rate in [LW02], which provides a figure of 68.94%. Indeed, this figure is far higher than the HeiMa corrector's recall (48.62%) that the authors used for comparison.

[3]Precision is, somewhat unconventionally, termed "accuracy" in their paper.

	SPELL	CORRECT	ASPELL	FSA PACKAGE	AGREP
Implemented Year	1979	1990	1998	1998	1988-1991
Spell Checking Technique(s)	hashing	N.A.	hashing	finite automata hashing	bitap exact matching
Spell Correcting Technique(s)	N.A.	probability reverse edit distance	Metaphone edit distance	finite automata cut-off edit distance	bitap approximate matching
Corrector Nature	N.A.	interactive	interactive	automatic	interactive
Morphological Analysis	exists in checking	none	none	exists in correcting	none
Flexibility	characters	characters	characters	characters and regular expression	characters and regular expression
Language(s)	English only	English only	English and others	English and others	English and others
Encoding(s)	ASCII only	ASCII only	ASCII/ Unicode	ASCII/ Unicode	ASCII/ Unicode
Dictionary Involved	yes	yes	yes	yes	optional
Time Complexity	$\mathscr{O}(1)$	N.A.	$> \mathscr{O}(1)$	$\mathscr{O}(m)$	$\mathscr{O}(mn)$

Table 5.7: Classification of spell checking/correcting packages Part I.

In our context, 84.13% of correct suggestions were found in the first 5 matches, and 95.81% were found in the first 10. Comparative figures are not available from [LW02].

The same as spell checking in Chinese, the fact that CInsunSpell performs worse than the spell checking and correcting packages for English and DNS Spell Checker in spell correcting is partially attributable to the complexity of the language structure concerned. It is also attributable to Chinese's input methods used for creating the corpus.

5.5 Classification

In the earlier part of this chapter, the empirical performance of several spell checking and correcting packages have been examined. In this section, we present a classification of all of the ten spell checking and correcting algorithms that nine of which (except MS Word) have been discussed in Chapters 3, 4, and 5. Their functionalities, characteristics, and implementation strategies are presented across two tables, as seen in Tables 5.7 and 5.8. The ten rows of each table correspond to attributes of the packages, and each column in each table corresponds to some package.

SPELL is the earliest developed spell checker and it only performs spelling error detection task whereas CORRECT only performs spell error correction task and all the others perform both spelling error detection and correction. 50% of the algorithms made use of some forms of edit distance technique for spelling correction regardless of the language being checked. This finding supports the statement that edit distance is the most widely used technique for spelling correction, which was mentioned in Section 3.4.1. All the packages (except SPELL) are interactive spelling correctors (i.e. require the user's input in choosing the correct spelling from a list of candidate suggestions), except FSA PACKAGE

	SPEEDCOP	AURA	MS Word	DNS	CInsunSpell
Implemented Year	1984	2002	2003	2003	2002
Spell Checking Technique(s)	similarity key	neural nets	N.A.	N.A.	trigram/ probability
Spell Correcting Technique(s)	error reversal	hamming distance n-gram phonetic code	N.A.	N.A.	edit distance weight distribution
Corrector Nature	automatic	interactive	interactive/ automatic	interactive	interactive
Morphological Analysis	none	none	assume existence	N.A.	none
Flexibility	characters	characters	characters	characters	characters
Language(s)	English only	English and others	English and others	Northern Sotho only	Chinese only
Encoding(s)	ASCII only	ASCII only	ASCII/ Unicode	ASCII only	Unicode only
Dictionary Involved	yes	yes	yes	yes	yes
Time Complexity	N.A.	N.A.	N.A.	N.A.	N.A.

Table 5.8: Classification of spell checking/correcting packages Part II.

and SPEEDCOP, which can only be automatic and MS Word, which can also be automatic when the configuration is set up accordingly. The techniques used for the various packages were described in Chapter 4. FSA PACKAGE and AGREP are the only two spell checking and correcting packages that support matching for both characters and regular expressions whereas all the others can only process matching of characters. The packages that can handle encoding other than ASCII (i.e. Unicode) are ASPELL, FSA PACKAGE , AGREP, MS Word, and CInsunSpell. All the packages supply their own dictionaries for spell checking/correcting purposes, except AGREP, which does not come with a built-in dictionary. It performs matches against any text supplied or specified by the user.

5.5.1 Concept lattice

In this section, a concept lattice (or a Galois lattice) [GW99] was constructed to characterise and analyse the spell checking and correcting algorithms investigated in Chapter 4 as well as in this chapter. Concept lattices offer a classification method that is used in formal context analysis (FCA) to better understand and analyse data such as that provided earlier in Section 5.5.

A concept lattice is a very practical representation of a hierarchical structured graph, which represents an ordering of so-called "concepts" and their attributes (or properties). In such a concept lattice, common attributes of a group of objects are identified and may be depicted in a line diagram. The lattice depicted in the line diagram below was derived from Tables 5.7 and 5.8.

Objects in the lattice correspond to the various spell checking and correcting algorithms. They are shown in clear rectangles, and linked by dashed lines to certain nodes. Attributes are depicted in greyed rectangles, also linked to certain nodes by dashed lines.

The attributes correspond to characteristics and functionalities depicted in the tables. Here, we focus mainly on the characteristics, functionalities, and implementation strategies of the various packages—i.e. this lattice does not reflect properties of packages related to efficiency.

Each node in the line diagram is called a concept. Each concept corresponds to a class of algorithms and also to a set of attributes that are held in common by the algorithms in the class. The algorithms associated with a concept are those that can be reached by a downward path from the concept.

The concept's attributes are those that can be reached by an upward path. The line diagram has a so-called bottom concept, also depicted at the bottom of the diagram. This concept's class of algorithms is empty (the empty set), and its class of attributes is the entire set of possible attributes in this context. It is construed to mean that no package possesses all the attributes under discussion.

Similarly, the line diagram has a top concept. Its class of objects includes all packages under consideration, and its class of attributes corresponds to the singleton set that contains only the attribute "characters". In other words, all the packages perform matching on characters.

Each concept between the top and bottom concepts is characterised by its distinctive set of attributes and set of objects. Thus, the attribute set {hashing, morphological analysis, English, ASCII, supplied dictionary, spell checking, characters} is associated with the concept that is identified with the object set {SPELL}. The properties associated with the "AGREP" object set include some, but not all of the aforementioned properties, and include a number of others as well—for instance "spell correcting", "regular expressions", and "Unicode".

FCA technology allows for the formal derivation of so-called implications that exist between attributes in a concept lattice. The tool Concept Explorer [Yev03] can be used to derive these implications. In the present example, fifteen implications between attributes were identified. Such implication rules form the basis of FCA knowledge mining strategies, and sometimes provide interesting nuggets of information about the objects. This is supported by the following examples:

- *Rule 1* shows that all ten objects performs matching on characters. This is the top concept as mentioned earlier.

- *Rule 2* shows that if an object performs morphological analysis and performs matching on characters, then it also performs spell checking in English, can handle the ASCII encoding and has a supplied dictionary. This implication is supported by three objects, namely "SPELL", "FSA" and "MS Word". In other words, these three objects share the common properties: "morphological analysis", "characters", "spell checking", "English", "ASCII", and "supplied dictionary". These properties form a subset of each attribute set associated with each of these objects.

- *Rule 5* shows that if an object performs matching on characters and in English, then it can also handle the ASCII encoding. This implication is supported by eight objects (all the packages except "FSA" and "AGREP").

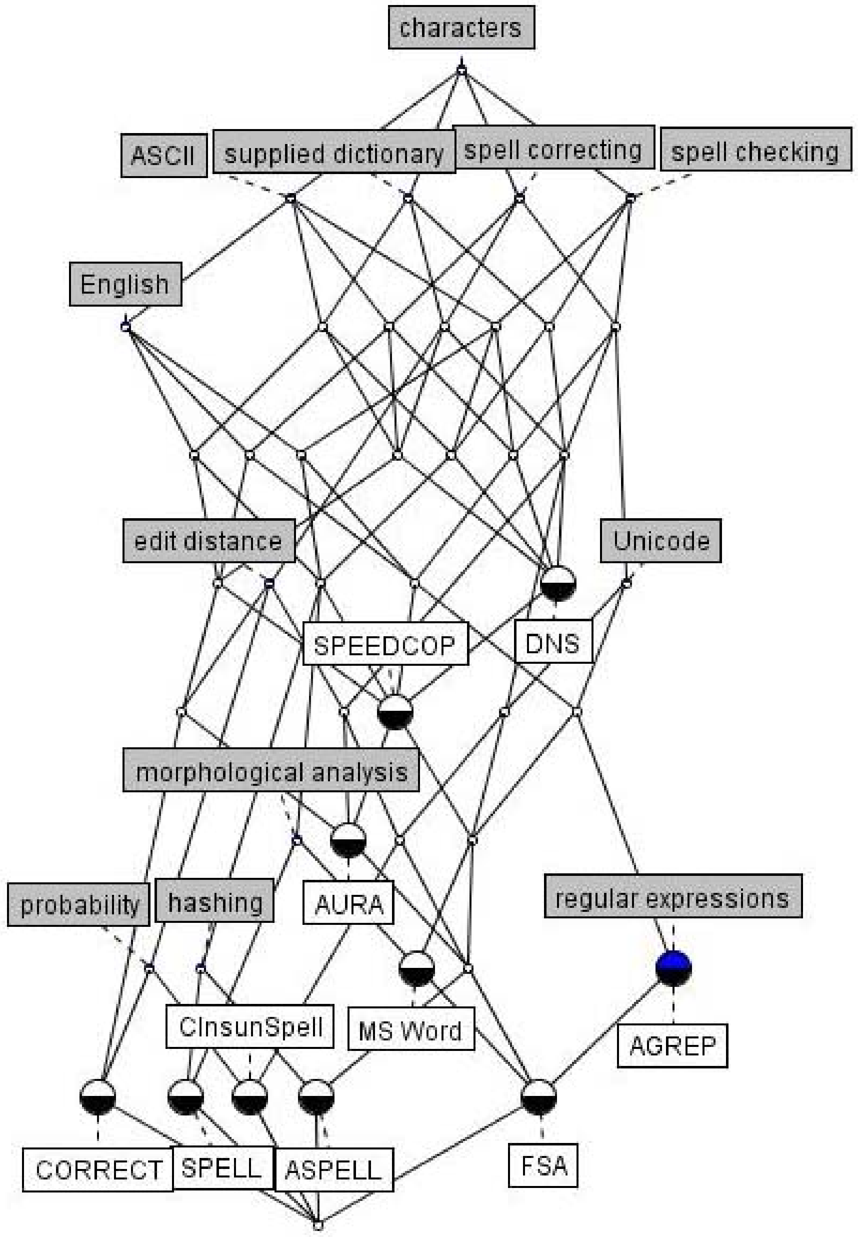

Figure 5.4: Concept lattice for various spell checking and correcting packages.

- *Rule 8* shows that if an objects performs both spell checking and correcting in English, performs matching on both characters and on regular expressions, can handle both the ASCII and Unicode encodings, and has a supplied dictionary, then it employs edit distance and performs morphological analysis. This implication is supported by one object, namely "FSA".

Such a lattice-based classification of algorithms is therefore considerably richer than a mere table layout. It provides complete information about which attributes are held in common by which objects. While the above implications are clearly not meant to be interpreted as universally true, they nevertheless provide an interesting starting point to investigate whether they are indeed universally true, or whether algorithms can be found that negate them. Indeed, if no existing algorithm can be found, this might signal a challenge to explore the possibilities of actually developing one.

5.6 Conclusions

In this chapter, we classified the spell checking and correcting algorithms described in Chapter 4 in the light of performance, characteristics, functionalities, and implementation strategies. The performance of various spell checking and correcting packages was evaluated in detail in terms of spelling error detection and correction accuracy and efficiency for the spell checking task. Classification was first presented in a tabular format in Section 5.5. The data was further represented in a concept lattice in Section 5.5.1 for a considerably more in-depth classification.

According to the experimental results in Section 5.2.2, SPELL has the best time performance although it was the first implemented spell checker. It was designed under the circumstances where hardware memory and space were extremely limited. The time performance has not been improved by any of the more recent spell checking and correcting packages. It was noticed that the number of misspellings contained in a document may vary the time performance. It was also noticed that the size of the dictionary (despite the way it is structured) may also influence the time performance. Thus, it would be interesting to investigate, as a matter of future research, how and to what extent these elements affect the efficiency.

The size of the dictionary used by each package may well influence its spell checking and correcting performance. However, it was noticed that the content of the dictionary may have an even more significant effect on the performance. This was seen in the experimental results of DNS Spell Checker (Section 5.3).

Half of the algorithms involved in the classification employed some forms of edit distance technique for spelling correction despite the fact that the time performance is considerably worse compared to hashing and despite the languages being spell checked. Edit distance has proven to be effective in Latin-based languages (e.g. English) as well as non-Latin-based languages (e.g. Chinese).

Four of the algorithms employed n-gram and probability techniques. Phonetic information has been incorporated to tackle the problems with phonetic errors such as in ASPELL. ASPELL appears to achieve better spelling correction accuracy. However, the time performance seems to suffer because of the phonetic sub-algorithm within ASPELL. FSA PACKAGE and AGREP are the only packages that handle data types other than characters, as they handle regular expressions. Spell checking and correcting packages developed from the early 90's, such as ASPELL, FSA PACKAGE , AGREP, MS Word, and CInsunSpell, all support the Unicode encoding.

From the description and classification of the aforementioned algorithms, it was seen that the current trend for the implementation of spell correcting algorithms is towards hybrid approaches, i.e. combining several techniques. It would therefore seem that each spell correcting technique described in Chapter 4 on its own is not regarded as sufficient to achieve the high accuracy requirements.

It was also seen that only 30% of the spell checking and correcting packages achieve automatic spelling correction, namely SPEEDCOP, FSA PACKAGE , and MS Word. The rest still only possess an interactive spelling corrector which is to say that they require user input for selection from a list of candidate suggestion to complete the spell correcting task. It was also observed that some words do not have many similarly spelled words and thus in the suggestion phase, it was always possible to suggest the correct spellings of the words albeit many errors occur in each misspelling. On the other hand, if a word has many similarly spelled words, it becomes rather difficult to find the correct spelling even with just one error occurring in the word.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The spell checking and correcting problems have been dealt with as two separate procedures—spelling error detection and correction—albeit that a small number of spell correctors combine the two processes into one. In this dissertation, we have approached the spell checking and correcting problems in the four major steps:

- *Step 1.* Provide the denotation semantics of the spell checking and correcting problems—i.e. the spelling error detection and correction tasks were described in terms of mathematical functions. The denotational semantics provides the basic foundation for the tasks in their simplest forms for better understanding of the problems. The spell checking and correcting tasks were viewed generically as a sequence of activities. For spell checking, two steps were identified to achieve the task:
  - $TextStripping_L$: The text is broken down into a set of words and possible words in the text.
  - $Misses_L$: The misspelled words in the text are identified and isolated.

  The process of spell correcting then follows the process of spell checking. A general approach is identified as $Suggest_L$. For each misspelled word in the set delivered by $Misses_L$, a set of possible alternative spellings is suggested.

- *Step 2.* Identify the techniques used to achieve the spell checking and correcting tasks. The spell checking techniques that have been used to this date are divided into two main categories, namely dictionary lookup and n-gram analysis. Dictionary lookup techniques are used to compare and locate an input string in a dictionary. Dictionary lookup techniques are further divided into hashing, binary search trees, and finite automata. These are standard string searching techniques aimed at reducing search time. The use of n-gram analysis can be described by a function that maps some text to a set of n-tuples. A probabilistic approach which relies on n-gram frequency counts is used for spell checking.

The design of the type of spell correcting techniques is heavily influenced by the type of spelling error patterns. Spelling techniques that have been explored in this dissertation focus on isolated-word error correction. Techniques that have been used to date are minimum edit distance, similarity key techniques, rule-based techniques, n-gram based techniques, probabilistic techniques, and neural networks. It was seen that each of these spell correcting techniques can be used either on its own or together with other techniques to achieve the spell correcting task. The current trend of implementation off spell correcting algorithms tends to adopt the latter as it seems that each technique on its own is not sufficient to achieve high accuracy.

Other than the techniques employed for spell checking and correcting, there exist other issues, such as dictionary construction, morphological analysis, and word boundary issues. Dictionaries used or supplied may include all inflected forms of words. However, in reality, this might not be the case as it is expensive to keep all inflected forms in a data structure. It is more sensible to keep only the word stems in the dictionary file. Each input word has to undergo morphological processing to be stripped off any affixes. For some non-Latin languages, there are issues concerning word boundaries.

- *Step 3.* Investigate various spell checking and correcting algorithms, namely SPELL, CORRECT, ASPELL, FSA PACKAGE, AGREP, SPEEDCOP, AURA, DNS Spell Checker, and CInsunSpell. It was seen that spelling error detection in the various selected spell checking and correcting packages heavily depend on dictionaries (size and details) used or supplied and that spell checking and correcting algorithms designed for English tend to prefer the use of dictionary lookup techniques. It was also seen that edit distance is indeed the most widely used and studied spell correcting technique no matter the language. Probabilistic techniques have become popular in past two decades. In addition, n-gram-based and probabilistic techniques have become the natural choice for dealing with languages that possess more complex language structures by nature, such as Chinese and Bangla. It was also seen that according to the current trend, an algorithmic solution for the spell checking and correcting tasks often involves a combination of techniques described in Chapter 3 in order to produce optimal results in terms of accuracy.

  From our investigation of the various algorithms, it was seen that edit distance is still by far the most widely studied and used spell correcting technique to date. Similarity key techniques have not been seen used since SPEEDCOP. More recent evidence of involvement of rule-based techniques is found in the phonetic module inside ASPELL and AURA. Rule-based techniques are also incorporated into spelling correctors that perform morphological analysis for context-dependent spelling correction. n-gram-based techniques appear to be the choice to construct language models for languages with more complex structures and for performing context-dependent spelling correction. Probabilistic techniques often make use of n-gram language models in spell correcting. They have become a well-accepted and used method despite the fact that their adoption for solving spell correcting problems is relatively recent. Neural networks have not been widely adopted by isolated-word spell correcting systems such those which have been studied here. During the investigation, it was noticed that although spell checking and correcting are applied on different languages, similar spelling error patterns were identified. Spell checking and correcting in languages other than English employed the same techniques identified in Chapters 3 and 4.

- *Step 4.* Empirically investigate the performance the various spell checking and correcting algorithms and classify them. The investigation in Chapter 4 serves as a prelude to the classification. Various spell checking and correcting algorithms were classified later in the same chapter in terms of their performance, characteristics, functionalities, and implementation strategies. The performance of various spell checking and correcting packages was evaluated. Experiments were conducted to investigate the recall accuracy for spelling error detection and the suggestion accuracy for spelling correction. The morphological intelligence of FSA PACKAGE and MS Word was also investigated. Further experiments were conducted to evaluate the efficiency for four specific spell checking/correcting packages, namely SPELL, FSA PACKAGE, ASPELL, and AGREP. Classification of the various spell checking and correcting packages was then constructed in order to provide a unified presentation. Furthermore, a concept lattice was constructed to illustrate how each package relates to the others. It provided a more detailed understanding of the relation between each package as well as the relations between these packages and their attributes.

  According to the experimental results, it was seen that packages that employed hashing for spell checking, such as SPELL, achieved the best time performance. ASPELL was found to be significantly slower compared to SPELL and AGREP. This may be caused by the large set of phonetic transformation rules which is included. From the classification, it was seen that the current trend for the implementation of spell checking and correcting packages is towards hybrid approaches. It was also seen that edit distance has indeed been the most widely used spell correcting technique amongst all the selected packages and this finding corresponds with our investigation in Chapter 4.

My personal perspective based on the research reported in this dissertation is that the design of a good spell checker and corrector has to achieve a balance between a well structured dictionary or word list, well designed and implemented matching algorithms, and a good understanding of human behaviour.

6.2 Future Work

Spell checking and correcting have become a part of everyday life for today's generation. Whether it be working with text editing tools, such as MS Word, or typing text messages on one's cellular/mobile phone, spell checking and correcting are an inevitable part of the process.

From the classification presented in Chapter 5, it was seen that the current trend for the implementation of spell checking and correcting packages is towards hybrid approaches. It was also noticed that most of the investigated spelling correcting packages are interactive rather than automatic. To date, automatic spelling correction with high correction accuracy still remains a challenge. There is a considerable degree of difficulty in suggesting the correctly spelled words without taking the context into consideration. This indicates that there is a need to combine isolated-word error correction techniques with context-dependent error correction techniques in order to achieve higher suggestion accuracy.

The empirical investigation in Chapter 5 opens the door to further research regarding spell checking and correcting languages other than Euro-based languages, such as Chinese and Arabic. Languages with a large alphabet and a more complex structure incline towards employing probabilistic techniques. Spell correcting such languages tends to be context dependent, as the straight-forward isolated-word error correction methods do not achieve high correction rate. Other issues such as word/text segmentation also provide a great challenge.

Given the apparent success of ASPELL in spelling correction, spell checking and correcting packages that are able to handle phonetic errors appear to achieve better suggestion accuracy. However, the time performance seems to suffer considerably because of the phonetic sub-algorithms within these packages. It would be worth experimenting and investigating ways to improve the time performance.

As mentioned in Chapter 5, AGREP was unexpectedly slow, in that the bitwise operation in the bitap algorithm which is used should theoretically enhance efficiency. It would be interesting to investigate whether a specialised spell checking package based on the bitap algorithm could be developed to challenge SPELL in efficiency.

Google's spell checking/correcting algorithm automatically checks to see if a query word is used in the most common version of a word's spelling. It does not check against a supplied dictionary, but rather checking against the frequency of occurrences of all words being searched on the Internet using this particular search engine. Spelling suggestions were also found according to the frequency of occurrences of all words being searched. Unfortunately, the matching/searching algorithm behind Google's spell checking/correcting algorithm is not available in the public domain. It would be of interest to compare the performance of Google's spell checking/correcting algorithm against the packages involved in the empirical investigation in this dissertation.

It was noticed that the number of misspellings contained in a document affects the time performance. It was also noticed that the size of the dictionary (despite the way it is structured) may also influence the time performance. Thus, it would be interesting to investigate how and to what extent these elements affect the time performance by fixing the number of misspellings in each document as well as providing the same dictionary to each package involved. Furthermore, in order to improve the performance, we propose an investigation on the use of parallelism (e.g. two automata perform simultaneous searching) in finite-state spell checking and correcting packages.

Most spell checking and correcting packages (interactive spell correcting packages in particular) are designed for their first-language speakers, i.e. it is assumed that the users already know how to spell in the designated languages. Few spell checking and correcting packages cater for people who have little knowledge of the languages for which they need to spell check or correct. This group of users include people who are learning a foreign language or children who are still in the process of learning a language. Spelling corrections provided by most packages are given in words in the same language. It is often difficult for a user who is categorised in this group to decipher the meaning of each suggested correction and make the right choice. Thus, it is proposed to adopt the concept of aspect-oriented programming (AOP). AOP allows each system to separately express its aspects of concern, and then automatically combine these separate descriptions into a final executable form [KLM$^+$97]. This entails keeping the spell checking and correcting techniques and functionalities as the core algorithms which remain unchanged throughout the entire package, and introducing the concept of aspect-aware interfaces. This proposed concept can be illustrated as follows:

The most efficient and/or effective spell checking and correcting techniques (e.g. hashing, edit distance, and phonetic algorithm) indicated in the classification in Chapter 5 may be used as the core algorithms of the package. For each target user group (e.g. first-language speakers, language learners, and children), a different interface which caters for each group of users is separately implemented. For first-language speakers, suggested corrections are returned as a list of words; for language learners, suggested corrections together with a brief explanation of the each suggestion are returned; and for children, suggested corrections along with a picture representing the meaning of each suggested correction are returned. Please consult [KLM$^+$97, KM05] for more details on incorporating the AOP concept.

Lastly, it is possible that more than one language appears in the same text, such as a English translation of a French article. When both the original text and the translated output appear in the same document, it is essential that the spell checking/correcting package can identify the languages and spell check and correct accordingly. In the case of African languages, a document is often written in various languages. It is necessary to pre-process the text with language identification. This function would be beneficial to be included as part of the design of a spell checking/correcting package. Spell checkers and correctors developed for the African languages are still at the infant stage. There is great potential for improving their scope and accuracy.

Bibliography

[AC75] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[AFW83] R. Angell, G. Freund, and P. Willett. Automatic spelling correction using a trigram similarity measure. *Information Processing and Management*, 19(4):255–262, 1983.

[Atk04] K. Atkinson. Aspell, 2004. http://www.gnu.org/software/aspell/ (Last accessed: March 2008).

[Aus96] J. Austin. Distributed associative memories for high speed symbolic reasoning. *Fuzzy Sets and System*, 82(2):223–233, 1996.

[Aus98] J. Austin. Pride and Prejudice, 1998. Project Gutenberg. http://www.gutenberg.org/etext/1342 (Last accessed: March 2008).

[BB59] W. W. Bledsoe and I. Browning. Pattern recognition and reading by machine. In *Proceedings of the Eastern Joint Computer Conference*, volume 16, pages 225–232, 1959.

[Bel57] R. E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Ben00] J. Bentley. *Programming Pearls, Second Edition*. Addison-Wesley, Massachusetts, 2000. Private Communication. September 2006.

[BK03] K. R. Beesley and L. Karttunen. *Finite State Morphology*. CSLI Publications, Stanford, 2003.

[Bla60] C. R. Blair. A program for correcting spelling errors. *Information and Control*, 3(1):60–67, 1960.

[Blo70] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BM00] E. Brill and R. C. Moore. An improved error model for noisy channel spelling correction. In *ACL '00: Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, pages 286–293, 2000.

[BNC05] BNC. British National Corpus, 2005. http://www.natcorp.ox.ac.uk/ (Last accessed: March 2008).

[Boc91] A. K. Bocast. Method and apparatus for reconstructing a token from a token fragment, U.S. patent number 5,008,818, 1991. Design Service Group, Inc. McLean, V.A.

[Bod06] J. Bodine. Improving Bayesian spelling correction. http://www.cs.cornell.edu/courses/cs474/2006fa/projects/jacqueline%20bodine.doc, 2006.

[BYG92] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[CFV91] V. Cherkassky, K. Fassett, and N. Vassilas. Linear algebra approach to neural associative memories and noise performance of neural classifiers. *IEEE Transactions on Computers*, 40(12):1429–1435, 1991.

[CG91] K. W. Church and W. A. Gale. Probability scoring for spelling correction. *Statistics and Computing*, 1(2):93–103, 1991. Private Communication with K. W. Church. September 2006.

[Cha94] C.-H. Chang. A pilot study on automatic Chinese spelling error correction. *Journal of Chinese Language and Computing*, 4(2):143–149, 1994.

[CLRS03] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* MIT Press, Massachusetts, 2003.

[Com80] D. Comer. A note on median split trees. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):129–133, 1980.

[Cor06] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual, 2006. http://www.intel.com/design/processor/manuals/248966.pdf (Last accessed: March 2008).

[CRW91] V. Cherkassky, M. Rao, and H. Wechsler. Fault-tolerant database using distributed associative memories. *Information Sciences*, 53(1):135–158, 1991.

[CT94] W. B. Cavnar and J. M. Trenkle. n-gram-based text categorization. In *Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, 1994.

[CV89] V. Cherkassky and N. Vassilas. Back-propagation networks for spelling correction. *Neural Networks*, 3(7):166–173, 1989.

[CVBW92] V. Cherkassky, N. Vassilas, G. L. Brodt, and H. Wechsler. Conventional and associative memory approaches to automatic spelling correction. *Engineering Applications of Artificial Intelligence*, 5(3):223–237, 1992.

[Dac98] J. Daciuk. *Incremental Construction of Finite-State Automata and Transducers, and their Use in the Natural Language Processing.* PhD thesis, Technical University of Gdańsk, 1998.

[Dai94] People's Daily. People's Daily corpus, 1993–1994. http://www.people.com.cn/ (Last accessed: March 2008).

[Dam64] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

[Dav62] L. Davidson. Retrieval of misspelled names in an airline passenger record system. *Communications of the ACM*, 5(3):169–171, 1962.

[DEG90] R. Deffner, K. Eder, and H. Geiger. Word recognition as a first step towards natural language processing with artificial neural networks. In *Konnektionismus in Artificial Intelligence und Kognitionsforschung. Proceedings 6. sterreichische Artificial Intelligence-Tagung (KONNAI)*, pages 221–225, London, 1990. Springer-Verlag.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall Series in Automatic Computation, Englewood Cliffs: Prentice-Hall, New Jersey, 1976.

[DM81] M. R. Dunlavey and L. A. Miller. On spelling correction and beyond. *Communications of the ACM*, 24(9):608–609, 1981.

[Dom64] B. Domolki. An algorithm for syntactical analysis. *Computational Linguistics*, 8:29–46, 1964.

[dSP04] G.-M. de Schryver and D. J. Prinsloo. Spellcheckers for the South African languages, part 1: The status quo and options for improvement. *South African Journal of African Languages*, 24(1):57–82, 2004.

[Fau64] R. D. Faulk. An inductive approach to language translation. *Communications of the ACM*, 7(11):647–653, 1964.

[For73] G. D. Forney. The Viterbi algorithm. In *Proceedings of the IEEE*, volume 61(3), pages 268–278, 1973.

[Gad90] T. N. Gadd. PHOENIX: the algorithm. *Program: Automated Library and Information Systems*, 24(4):363–369, 1990.

[Gat37] A. I. Gates. *Spelling Difficulties in 3867 Words.* Bureau of Publications, Teachers College, Columbia University, New York, 1937.

[GFG03] R. Garfinkel, E. Fernandez, and R. Gopal. Design of an interactive spell checker: Optimizing the list of offered words. *Decision Support Systems*, 35(3):385–397, 2003.

[Gov01] S. Govindarajan. Inside the Pentium 4, 2001. http://www.pcquest.com/content/technology/101021101.asp (Last accessed: March 2008).

[GW99] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, Berlin-Heidelberg, 1999.

[HA02] V. J. Hodge and J. Austin. A comparison of a novel neural spell checker and standard spell checking algorithms. *Pattern Recognition*, 35(11):2571–2580, 2002.

[HA03] V. J. Hodge and J. Austin. A comparison of standard spell checking algorithms and a novel binary neural approach. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1073–1081, 2003.

[Han89] J. Hankamer. *Morphological Parsing and the Lexicon*, pages 392–408. MIT Press, Massachusetts, 1989.

[HD80] P. A. V. Hall and G. R. Dowling. Approximate string matching. *ACM Computing Survey (CSUR)*, 12(4):381–402, 1980.

[HLT06] TshwaneDJe HLT. Sesotho sa Leboa corpora, 2006. Private Communication. September 2006.

[HMU06] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation, Third Edition*. Addison-Wesley, 2006.

[Hor80] R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10(6):501–506, 1980.

[HS82] J. J. Hull and S. N. Srihari. Experiments in text recognition with binary n-gram and Viterbi algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(5):520–530, 1982.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the IRE*, volume 40, pages 1098–1101, 1952.

[Ing96] P. Ingels. Connected text recognition using layered HMMs and token parsing. In *Proceedings of the Second Conference of New Methods in Language Processing*, pages 121–132, 1996.

[JM00] D. Jurafsky and J. H. Martin. *Speech and Language Processing: An introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice-Hall, 2000.

[JT77] F. E. Muth Jr. and A. L Tharp. Correcting human error in alphanumeric terminal input. *Information Processing and Management*, 13(6):329–377, 1977.

[Kar83] L. Karttunen. KIMMO — a general morphological processor. In *Texas Linguistic Forum. Department of Linguistics, The University of Texas*, volume 22, pages 165–186, 1983.

[KCG90] M. D. Kernighan, K. W. Church, and W. A. Gale. A spelling correction program based on a noisy channel model. In *Proceedings of the 13th International Conference on Computational Linguistics*, volume 2, pages 205–210, 1990.

[KLM$^+$97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, J.-M. Loingtier C. V. Lopes, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[KLS98] T. Kowaltowski, C. L. Lucchesi, and J. Stolfi. Finite automata and efficient lexicon implementation. Technical Report IC-98-02, Institute of Computing, University of Campinas, 1998.

[KM05] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th International Conference on Software Engineering*, pages 49–58, 2005.

[Knu85] D. E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6:163–180, 1985.

[Knu98] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching, Second Edition.* Addison-Wesley Professional, 1998.

[Koh88] T. Kohonen. *Correlation Matrix Memories*, pages 171–180. MIT Press, Cambridge, M.A., 1988.

[Kos83] K. Koskenniemi. Two-level model for morphological analysis. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 683–685, 1983.

[KST92] J. Y. Kim and J. Shawe-Taylor. An approximate string-matching algorithm. *Theoretical Computer Science*, 92(1):107–117, 1992.

[KST94] J. Y. Kim and J. Shawe-Taylor. Fast string matching using an n-gram algorithm. *Software—Practice and Experience*, 94(1):79–88, 1994.

[Kue05] G. Kuenning. ISPELL, 2005. http://www.cs.hmc.edu/ geoff/ispell.html (Last accessed: March 2008).

[Kuk90] K. Kukich. A comparison of some novel and traditional lexical distance metrics for spelling correction. In *Proceedings of INNC-90-Paris*, pages 309–313, 1990.

[Kuk92] K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys (CSUR)*, 24(4):377–439, 1992.

[Lev66] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Technical report, Soviet Physics Doklady, 1966.

[LR93] A. J. Lait and B. Randell. An assessment of name matching algorithms. Technical report, Department of Computer Science, University of Newcastle upon Tyne, 1993.

[LS99] G. S. Lehal and K. Singh. A study of data structures for implementation of Punjabi dictionary. In *Cognitive Systems Reviews and Previews, ICCS '99*, pages 489–497, 1999. J. R. Isaac and K. Batra (editors).

[LW02] J. Li and X. Wang. Combine trigram and automatic weight distribution in Chinese spelling error correction. *Journal of Computer Science and Technology*, 17(6):915–923, 2002.

[LWS00] J. Li, X. Wang, and Y. Sun. The research of Chinese text proofreading algorithm. *High Technology Letters*, 6(1):1–7, 2000.

[Man03] J. Maniacky. Umqageli (automatic identification of Bantu languages). http://www.bantu-languages.com/en/tools/identification.php (Last accessed: March 2008), 2003.

[MBLD92] O. Matan, C. J. C. Burges, Y. LeCun, and J. S. Denker. Multi-digit recognition using a space displacement neural network. In J. M. Moody, S. J. Hanson, and R. O. Lippman, editors, *Neural Information Processing Systems*, volume 4, pages 488–495. Morgan Kaufmann Publishers, San Mateo, C.A., 1992.

[MC75] R. Morris and L. L. Cherry. Computer detection of typographical errors. *IEEE Transactions on Professional Communication*, PC-18(1):54–64, 1975.

[McI82] M. D. McIlroy. Development of a spelling list. *IEEE Transactions on Communication*, 30(1):91–99, 1982. Private Communication. July 2006.

[MDM91] E. Mays, F. J. Damerau, and R. L. Mercer. Context based spelling correction. *Information Processing and Management*, 27(5):517–522, 1991.

[Mea88] L. G. Means. Cn yur cmputr raed ths. In *Proceedings of the Second Conference on Applied Natural Language Processing*, pages 93–100. Association of Computational Linguistics, 1988.

[Mey90] B. Meyer. *Introduction to The Theory of Programming Languages*. Prentice-Hall International Series in Computer Science, 1990.

[Mit85] R. Mitton. Birkbeck spelling error corpus, 1985. http://ota.ahds.ac.uk/ (Last accessed: March 2008).

[Mit02] M. Mitzenmacher. Compressed Bloom filters. In *IEEE/ACM Transactions on Networking (TON)*, volume 10(5), pages 604–612, 2002.

[MS02] S. Mihov and K. U. Schulz. Fast string correction with Levenshtein-automata. *International Journal of Document Analysis and Recognition*, 5(1):67–85, 2002.

[MS04] S. Mihov and K. U. Schulz. Fast approximate search in large dictionaries. *Computational Linguistics*, 30(4):451–477, 2004.

[MX05] A. M. McEnery and R. Xiao. The Lancaster corpus of Mandarin Chinese, 2005. http://ota.ahds.ac.uk/ (Last accessed: March 2008).

[NR02] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002.

[NZ97] H. T. Ng and J. M. Zelle. Corpus-based approaches to semantic interpretation in natural language processing. *AI Magazine*, 18(4):45–64, 1997.

[Ofl96] K. Oflazer. Error-tolerant finite state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22(1):73–89, 1996.

[OG94] K. Oflazer and C. Gzey. Spelling correction in agglutinative languages. In *Proceedings of the Fourth Conference on Applied Natural Language Processing*, pages 194–195, 1994.

[OL97] B. J. Oommen and R. K. S. Loke. Pattern recognition of strings with substitutions, insertions, deletions, and generalized transpositions. *Pattern Recognition*, 30(5):789–800, 1997.

[OR18] M. K. Odell and R. C. Russell. U.S. patent numbers 1,261,167 (1918) and 1,435,663 (1922), 1918. U.S. Patent Office, Washington, D.C.

[Par03] R. Parsons. Soundex — the true story, 2003. http://west-penwith.org.uk/misc/soundex.htm (Last accessed: March 2008).

[PB02] L. Pretorius and S. E. Bosch. Finite-state computational morphology — treatment of the Zulu noun. *South African Computer Journal*, 28:30–38, 2002.

[PdS01] D. J. Prinsloo and G.-M. de Schryver. Corpus applications for the African languages, with special reference to research, teaching, learning and software. *Southern African Linguistics and Applied Language Studies*, 19(1–2):111–131(21), 2001.

[Pet80] J. L. Peterson. Computer programs for detecting and correcting spelling errors. *Communications of the ACM*, 23(12):676–687, 1980.

[Phi90] L. Philips. Hanging on the Metaphone. *Computer Language Magazine*, 7(12):38—44, 1990.

[Phi99] L. Philips. Double Metaphone, 1999. http://aspell.net/metaphone/dmetaph.cpp (Last accessed: March 2008).

[Phi00] L. Philips. The Double-Metaphone search algorithm. *C/C++ User's Journal*, 18(6):38–43, 2000.

[Pre06] Oxford University Press. Oxford WordSmith Tools 4.0, 2006. http://www.lexically.net/wordsmith/version4/index.htm (Last accessed: March 2008).

[Pro06] Opensource Software Translation Project. South African keyboard, 2006. http://translate.org.za/content/view/24/41/lang,en/ (Last accessed: October 2007).

[PZ84] J. J. Pollock and A. Zamora. Automatic spelling correction in scientific and scholarly text. *Communications of the ACM*, 27(4):358–368, 1984.

[RH74] E. M. Riseman and A. R. Hanson. A contextual postprocessing system for error detection using binary n-grams. *IEEE Transactions on Computers*, 23(5):480–493, 1974.

[Rit06] D. Ritchie. Unix spell checker (SPELL), 2006. Private Communication. September 2006.

[RPRB86] G. J. Russell, S. G. Pulman, G. D. Ritchie, and A. W. Black. A dictionary and morphological analyser for English. In *Proceedings of the Eleventh Conference on Computational Linguistics*, pages 277–279, 1986.

[RS95] E. Roche and Y. Schabes. Deterministic part-of-speech tagging with finite-state transducers. *Computational Linguistics*, 21(2):227–253, 1995.

[SGSC96] R. Sproat, W. Gale, C. Shih, and N. Chang. A stochastic finite-state word-segmentation algorithm for Chinese. *Computational Linguistics*, 22(3):377–404, 1996.

[She78] B. A. Sheil. Median split trees. A fast look-up technique for frequently occurring keys. *Communications of the ACM*, 21(11):947–958, 1978.

[Sid79] A. A. Sidorov. Analysis of word similarity on spelling correction systems. *Programming and Computer Software*, 5(4):274–277, 1979.

[Smy03] W. Smyth. *Computing Patterns in Strings*. Addison-Wesley, Canada, 2003.

[Sof08] Edgewall Software. PIDGIN, 2008. http://developer.pidgin.im/ (Last accessed: April 2008).

[SP88] R. M. K. Sinha and B. Prasada. Visual text recognition through contextual processing. *Pattern Recognition*, 21(5):463–479, 1988.

[ST79a] R. Shinghal and G. T. Toussaint. A bottom-up and top-down approach to using context in text recognition. *International Journal of Man-Machine Studies*, 11:201–212, 1979.

[ST79b] R. Shinghal and G. T. Toussaint. Experiments in text recognition with the modified Viterbi algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(2):184–192, 1979.

[TA97] M. Turner and J. Austin. Matching performance of binary correlation matrix memories. *Transactions of the Society for Computer Simulation International*, 14(4):1637–1648, 1997.

[Tay81] W. D. Taylor. GROPE — a spelling error correction tool, 1981. AT&T Bell Labs Tech. Mem.

[Tec03] Skype Technologies. Skype, 2003. http://skype.com/intl/en/useskype/ (Last accessed: April 2008).

[Tsa04] C.-H. Tsai. Similarities between Tongyong Pinyin and Hanyu Pinyin: Comparisons at the syllable and word levels, 2004. http://research.chtsai.org/papers/pinyin-comparison.html (Last accessed: March 2008).

[Ull77] J. R. Ullmann. A binary *n*-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *The Computer Journal*, 20(2):141–147, 1977.

[Uni03] Unicode. The Unicode® standard: A technical introduction, 2003. http://www.unicode.org/standard/principles.html (Last accessed: March 2008).

[Ver88] J. Veronis. Computerized correction of phonographic errors. *Computers and the Humanities*, 22(1):43–56, 1988.

[vHvZ03] G. B. van Huyssteen and M. M. van Zaanen. A spellchecker for Afrikaans, based on morphological analysis. In *TAMA 2003 South Africa: Conference Proceedings*, pages 189–194, 2003. Pretoria: $(SF)^2$ Press.

[Wag74] R. A. Wagner. Order-n correction for regular languages. *Communication of the ACM*, 17(5):265–268, 1974.

[Wag95] D. B. Wagner. Dynamic programming. *The Mathematica Journal*, 5(4):42–51, 1995.

[Wat95] B. W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Eindhoven University of Technology, 1995.

[Wat03] B. W. Watson. A new algorithm for the construction of minimal acyclic DFAs. *Science of Computer Programming*, 48(2-3):81–97, 2003.

[WF74] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[Whi04] T. White. Can't beat Jazzy, 2004. http://www-128.ibm.com/developerworks/java/library/j-jazzy/ (Last accessed: March 2008).

[Win03] R. L. Winkler. *Introduction to Bayesian Inference and Decision, Second Edition*. Probabilistic Publishing, 2003.

[WM91] S. Wu and U. Manber. AGREP, 1991. ftp://ftp.cs.arizona.edu/agrep/agrep-2.04.tar.Z (Last accessed: March 2008).

[WM92a] S. Wu and U. Manber. Fast text searching: Allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

[WM92b] S. Wu and U. Manber. AGREP — a fast approximate pattern-matching tool. In *Proceedings (USENIX) Winter 1992 Technical Conference*, pages 153–162, 1992.

[Xia05] R. Xiao. People's Daily corpus 2000, 2005. http://bowland-files.lancs.ac.uk/corplang/pdc2000/ (Last accessed: March 2007).

[Yev03] S. Yevtushenko. Concept Explorer, 2003. http://sourceforge.net/projects/conexp (Last accessed: March 2008).

[YF83a] E. J. Yannakoudakis and D. Fawthrop. An intelligent spelling error corrector. *Information Processing and Management*, 19(2):101–108, 1983.

[YF83b] E. J. Yannakoudakis and D. Fawthrop. The rules of spelling errors. *Information Processing and Management*, 19(2):87–99, 1983.

[You06] N. Young. Super fast spell checking in *C♯*, 2006. http://www.codeproject.com/useritems/SuperFastSpellCheck.asp (Last access: March 2008).

[YRT89] S. J. Young, N. H. Russel, and J. H. S. Thornton. Token passing: A simple conceptual model for connected speech recognition systems. Technical Report CUED/F-INFENG/TR38, Cambridge University Engineering Department, 1989.

[ZHZP00] L. Zhang, C. Huang, M. Zhou, and H. Pan. Automatic detecting/correcting errors in Chinese text by an approximate word-matching algorithm. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, pages 248–254, 2000.

[ZPZ81] E. M. Zamora, J. J. Pollock, and A. Zamora. The use of trigram analysis for spelling error detection. *Information Processing and Management*, 17(6):305–316, 1981.