Formalizations Of Error Models With Applications To Spelling Error Correction

By

Jing Xu

A thesis submitted in partial fulfillment of

the requirements for the degree of

Master of Applied Science (in Computer Science)

Saint Mary's University

Halifax, Nova Scotia

Submitted April 20, 2004

Your file Votre référence
ISBN: 0-612-90329-X
Our file Notre référence
ISBN: 0-612-90329-X

Dedicated to my husband for his sincere support

Formalizations Of Error Models With Applications To Spelling Error Correction

By Jing Xu

Date of Submission: April 20, 2004

Abstract:

For many information processing applications, there are several different existing error models and error correction algorithms. This research focuses on a general methodology for defining error models describing different types of errors in information processing. It includes formal definitions of channels and the error models and a general algorithm for applying an error model to correct errors. This general methodology represents all existing error models and corrects errors in a consistent way.

This research also discusses the computation of error models with application to spelling error correction. Different error models for various spelling error correction problems have been investigated. The improved Brill and Moore error model has been implemented to describe the approach of computing a spelling error model for specific users. Based on the general methodology devised in this research, four error models based on the improved Brill and Moore error model have also been described and tested.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

It is a pleasure to thank many people who made this thesis possible.

First and foremost, I would like to thank my thesis supervisor, Dr. Stavros Konstantinidis (Saint Mary's University) for supervising me, providing resources and subjects, and offering direction and penetrating criticism. He gives me lots of helpful comments and critical review of this thesis.

Gratitude and sincere thanks are due to my thesis external examiner Dr. Todd Wareham (Memorial University of Newfoundland) for his valuable comments and suggestions. I also want to thank Dr. Sageev Oore (Saint Mary's University) and Dr. Cezar Campeanu (University of Prince Edward Island) my thesis advisor, for their support and critical review on this thesis.

I am grateful to all my friends and professors from Math and Computing Science, Saint Mary's University, for being the surrogate family during the time I stayed there. Special thanks are due to Dr. Pawan Lingras for giving me the opportunity to be involved in this program. I also would like to thank Rose Daurie and Owen M. Smith for their care and technical support. I cannot end without thanking my parents and my husband, for their constant encouragement and love I have relied on throughout the time of writing this thesis. It is to them that I dedicate this work.

Chapter 1

Introduction

1.1 The Statement of the Problem

In a real world communication system, errors may occur anywhere and anytime. They can happen in computer to computer communications, human to computer communications (typing errors), or human to human communication (speech errors). A set of data D generated from a sender may be transformed into D' (D with errors) at the receiver side after passing through a noisy channel. In order to reduce/eliminate errors in a system, it is essential for us to have a thorough understanding of them.

Error modeling is used to assist in describing and analyzing various errors in an information processing system. A *channel* is a finite description of the (possibly infinitely many) error situations permitted in a communication system. An *error model* is the set of possible channels that one can use in modeling the errors of a communication system.

Over the years different error models and error correction algorithms have been developed for the spelling error correction problem. However, the characteristics of these models/algorithms determine their limitations in applying only to specific situations. For instance, references [3], [5], [31] only consider the isolated word error correction problem where spelling errors resulting in non-words will be corrected, and [21] only considers the real-word error correction problem where the spelling errors resulting in actual words will be corrected. To date, there has yet been no extensive work conducted in developing a general methodology for error models and a corresponding general error correction algorithm that can be used to describe all existing error models and correct errors in the same way.

1.2 Objectives and Scope

The focus of this research is the development of a general methodology for error modeling and error correction and its application to spelling error correction in computer typesetting. This methodology includes formal definitions of what a channel and an error model are and the algorithm that can correct errors described by the channel of a particular error model. The main application of this methodology in this research is to compute the channel corresponding to a specific typesetter.

The scope of the work in this thesis covers the following areas:

1. Investigate existing spelling error models and spelling error correction methods.

2. Introduce definitions of an error model and a general error correction algorithm

for all information processing applications.

3. Compute the channel of an error model for a specific typesetter from sample data. An existing spelling error model — the Brill and Moore Error Model — has been implemented with improvements according to our general methodology.

4. Apply the error model to correct spelling errors of a specific typesetter.

5. Compare experimental results. Four error models, based on the improved Brill and Moore error model, have been described and tested.

1.3 Structure of Thesis

This paper is organized into 8 chapters. The second chapter gives basic notions that will be used in this thesis and background information about error patterns, string distances, finite state automata and tries. Chapter 3 reviews techniques and issues related to automatic spelling error detection and correction in three areas. Chapter 4 introduces a general methodology for error correction in information processing applications. It covers the formal definitions of channel and error model, the correction algorithm and examples. Chapter 5 discusses an approach to generating string pairs from given sample data, introduces an algorithm for string pairs generation and gives some experimental results. These string pairs are necessary for computing the channel that describes errors in the sample data of the typesetter. Chapter 6 describes the implementation of the improved Brill and Moore error model. Chapter 7 consists of several testing cases. Four modified models derived from the improved Brill and Moore error model are described and tested. Chapter 7 also conducts comparisons on different error models. Finally, Chapter 8 gives conclusions and discusses the future works of this research.

Chapter 2

Basic Notions and Background

2.1 Basic Notions

An *alphabet* is a finite nonempty set of symbols. It is often denoted by Σ. For example, $\Sigma_A = \{0, 1\}$ is an alphabet of two symbols, 0 and 1, and $\Sigma_B = \{a, b, c\}$ is an alphabet of three symbols, a, b and c. Sometimes the space and comma symbols are in an alphabet while other times they are meta symbols used for descriptions. A *word* or *string* is a finite sequence $a_1 \ldots a_n$ such that each a_i is in Σ. For example, *01110* and *111* are strings over the alphabet Σ_A, *aaabccc* and *bbb* are strings over the alphabet Σ_B. The *empty string* is the string with no symbols, usually denoted by λ. The *empty string* has length zero. Vertical bars around a string indicate the *length* of a string. For example $|00100| = 5$, $|aab| = 3$, and $|\lambda| = 0$. A *language* is a set of strings over the alphabet Σ. The set may be empty, finite or infinite. The set of all possible strings over the alphabet Σ is denoted by Σ^*.

λ-NFA([10], [29])

A nondeterministic finite automaton with λ-transitions (λ- *NFA*) is a quintuple $A = (\Sigma_A, Q_A, S_A, F_A, T_A)$ such that Σ_A is an alphabet, Q_A is a finite nonempty set of states, S_A is the start state, F_A is the set of final states, and T_A is the set of transitions. Each transition in T_A is of the form $q_1 x q_2$, where q_1 and q_2 are states and x is either λ or a symbol that belongs to the alphabet. In this case, x is the *label* of the transition. A *computation* of A is an expression of the form $q_0 x q_1, \ldots, q_{n-1} x_n q_n$ such that each $q_{i-1} x q_i$ is a transition in T_A. A *computation* is accepting if q_0 is the start state and q_n is a final state. In this case, the string $x_1 \ldots x_n$ is called the accepted word. We denote by $L(A)$ the language accepted by A. An example of a λ-NFA is as below:

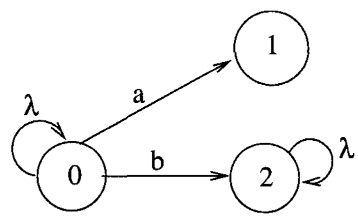


Figure 2.1: Some transitions of the λ-NFA are: 0a1, 2λ2

If the *label* of every transition in T_A is not λ then A is called a *nondeterministic finite automaton (NFA).* If, moreover, for every transitions of the form $q_0 x q_1$ and $q_0 x q_2$ we have that $q_1 = q_2$ then A is called a *deterministic finite automaton (DFA).* A finite automaton can be interpreted as a language recognizer or transducer.

Weighted Finite Automaton

We recall from [23], [26] the definitions of *weighted finite automaton* and *composition* of *weighted finite automata.*

Some applications such as text, speech recognition and image processing, require more general devices to account for the variability of the input data and to rank various output hypotheses. A *weighted (finite) automaton* is a finite automaton in which each transition is labelled with some weight and possibly initial and final weights in addition to the usual transition label. In this research, we use *weighted automata* as a simple efficient representation for all the inputs, outputs and transition information in text recognition. More formally a *weighted (finite) automaton* (WFA) W is a quintuple $W = (\Sigma_W, Q_W, S_W, F_W, T_W, K_W)$ such that Σ_W is a set of transition labels, Q_W is a finite set of states, S_W is the start state, F_W is a set of final states, and T_W is the finite set of transitions, and K_W is the weight function that assigns a real number weight to each transition in T_W. Weights introduced on transitions also define an underlying edge-weighted directed graph for which classical algorithms (shortest paths, maximal flow, etc.) apply. We can view any non-weighted automaton as a weighted automaton in which all transitions have weight 1. An example of a WFA is as below:

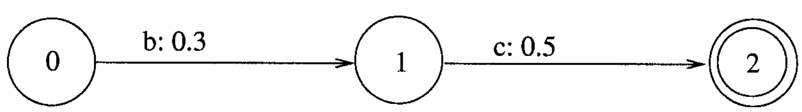


Figure 2.2: The weight of the transition $1c2$ is 0.5

A *weighted (finite-state) transducer (WFST)* is a weighted finite automaton W whose transitions are labelled with both an input and an output label such that

$\Sigma_W = \Sigma^* \times \Gamma^*$ for given finite alphabets Σ and Γ. It is a mapping from pairs of strings over two alphabets to weights. For a given pair $l = (s, w) \in \Sigma^* \times \Gamma^*$ we define $l(\text{in})=s$ and $l(\text{out})=w$. Note that the input and output label of a transducer could be the empty string λ. An empty input label indicates that no input string needs to be consumed when traversing the transition, while an empty output label indicates that no string is output when traversing the transition. Empty labels are needed because input and output strings do not always have the same length. An λ- *NFA* A can be considered as a *WFST* when each transition $q_1 x q_2$ of A is replaced with $q_1(x/x)q_2$ with weight equal to 1. The example of WFST can be viewed in the next section.

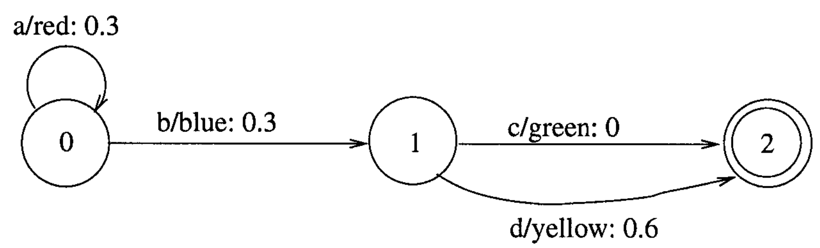
Composition ([23], [10])

Composition is a key operation on *FST*. The composition operator is denoted by $\circ$ and its defination is similar to the intersection operation for recognizers. In the classical case, a *WFST* for the composition of two given *WFST* A and B is constructed by considering the cross product of states of A and B.

A single composition algorithm is used to combine in advance information sources such as language models and dictionaries. Informally, the *composition* of two *WFST* A and B is a generalization of *NFA* intersection. Each state in the composition corresponds to a state pair in which one state is in A and another state is in B. If a transition in A is $q_0(x/y)q_1$ and a transition in B is $s_0(y/z)s_1$, then the transition $(q_0, s_0)x/z(q_1, s_1)$ is in $A \circ B$. The weight of this transition is the sum of the weights of the corresponding transitions in A and B. If the start states of A and B are q_0 and s_0, the start state in $A \circ B$ is (q_0, s_0). If the set of final states of A is $F_A = \{f_{a_1}, \ldots, f_{a_n}\}$ and the set of final sates of B is $F_B = \{f_{b_1}, \ldots, f_{b_m}\}$, the final states in $A \circ B$ have to be in the set $\{f_{a_i}, f_{b_j}\}$, where $i = 1 \ldots n$ and $j = 1 \ldots m$. The composition operation thus formalizes the notion of coordinated search in two graphs, where the coordination corresponds to a suitable agreement between paths labels. The example below shows the detail of computing the composition for two *WFST*.

Example of Composition on $WFST$

Given $WFST$ A as shown below, $(S_A = 0)$



the transitions in A are:

0 (a/red:0.3) 0

0 (b/blue:0.3) 1

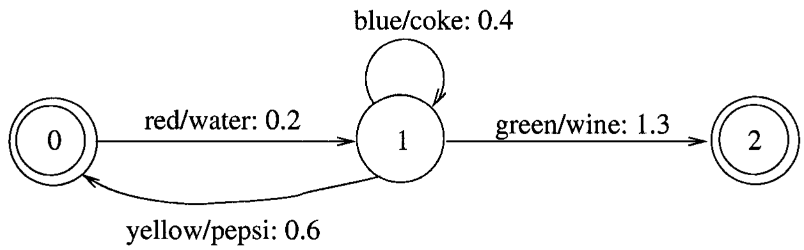
1 (c/green:0) 2

1 (d/yellow:0.6) 2

Given $WFST$ B as shown below, $(S_B = 0)$

the transitions in B are:

0 (red/water:0.2) 1



1 (blue/coke:0.4) 1

1 (yellow/pepsi:0.6) 0

1 (green/wine:1.3) 2

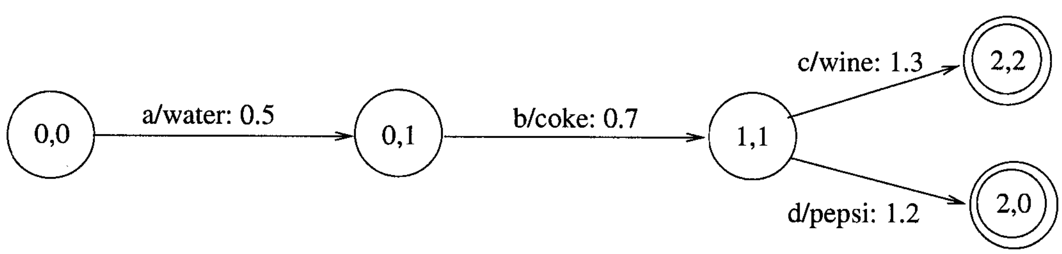
Then $A \circ B$ includes the following transitions — in fact, there are exactly the transitions of $A \circ B$ that are reachable from the start state (0,0) and can reach a final state of $A \circ B$.

(0,0) (a/water:0.5) (0,1)

(0,1) (b/coke:0.7) (1,1)

(1,1) (c/wine:1.3) (2,2)

(1,1) (d/pepsi:1.2) (2,0)



2.2 Background Information

2.2.1 Errors and string difference

Given an alphabet Σ and the set R of real numbers, we define the set E of *edit operations*. An *edit operation* is a pair (x, y) or x/y, where $x, y \in \Sigma \cup \lambda$, such that not both x, y are empty. If $x \neq y$, we call (x, y) an *error*. There exist three common errors:

(1) insertion error:λ/x;

(2) deletion error: x/λ;

(3) substitution error: x/y with $x \neq y$ and $x, y \in \Sigma$.

Given $\Sigma = \{a, b\}$, the possible edit operations are:

$a/a, b/b, a/b, b/a, a/\lambda, b/\lambda, \lambda/a, \lambda/b$

A cost function $f : E \to R$ assigns costs to the edit operations in E. Usually the cost values are assigned depend on applications. For example:

$f(a/a) = 0, f(a/b) = 2, f(a/\lambda) = 1, f(b/a) = 3, f(b/\lambda) = 1, f(b/b) = 0$

Definition 1: An *e-string* (edit or error string) is a string in E^*. The empty e-string over E is (λ/λ). If $h = (x_1/y_1)\ldots(x_n/y_n)$ is an e-string then we say that h transforms the word $x_1 \ldots x_n$ to $y_1 \ldots y_n$. Moreover we define the input and output parts of h such that $inp(h) = x_1 \ldots x_n$ and $out(h) = y_1 \ldots y_n$. Given an e-string $h = e_1 e_2 \ldots e_n$, then the cost of h is $f(h) = \sum_{i=1}^{n} f(e_i)$.

For example, if $h = (a/a)(b/a)(b/b)(b/\lambda)(a/\lambda)$

then $inp(h) = abbba$, and $out(h) = aab$ and under the cost function described above,

$$\begin{aligned} f(h) &= f(a/a) + f(b/a) + f(b/b) + f(b/\lambda) + f(a/\lambda) \\ &= 0 + 3 + 0 + 1 + 1 \\ &= 5 \end{aligned}$$

Definition 2: Suppose a cost function f is given, we define the *f-difference* $D_f(u, v)$ between two strings $u, v \in \Sigma^*$ to be the **minimum** cost of an e-string h that transforms u to v.

For example, Given $E = \{x/x, x/y, x/\lambda, \lambda/x : x, y \in \Sigma, x \neq y\}$

Cost function $f(x/x) = 0, f(x/y) = f(x/\lambda) = f(\lambda/x) = 1$

Then, the *f-difference* between string $s_1 = aabbb$ and $s_2 = aaba$ is

$D(s_1, s_2) = f((a/a)(a/a)(b/b)(b/a)(b/\lambda)) = 2.$

The proceeding concepts formalize the notion of error found in the literature on spelling error correction. Damerau (1964) [6] found that 80% of all misspelled words (non-word errors) in a sample of human keypunched text were caused by single-error misspelling, a single one of the following *edit operations*:

insertion — insert a character into the source string, such as *the* $\rightarrow$ *ther*;

deletion — delete a character from the source string, such as *the* $\rightarrow$ *th*;

substitution — substitute or replace one character with a different character at the same position in the sequence, such as *the* $\rightarrow$ *thw*;

transposition — reversal of two adjacent letters, such as *the* $\rightarrow$ *teh*;

Kukich (1992) [17] divided human typing errors into two categories: typographic errors and cognitive errors. In typographic errors (*spell* $\rightarrow$ *speel*) we assume that the writer knows the correct spelling. The errors usually occur as the result of mistyping. In cognitive errors (*separate* $\rightarrow$ *separite*), the errors are usually caused by typists misspellings of words. Phonetic error (*naturally* $\rightarrow$ *nacherly*) is a special class of cognitive errors in which the writer knows a phonetically correct spelling but lacks the knowledge on the sequence of letters for the intended word.

From Grudin's study (1983) [8], we know that most common errors result from the striking of a key immediately adjacent, either horizontally or vertically, to the intended key. The correct character could be replaced by a character immediately adjacent in the same row such as $right \rightarrow rihgt$. It is called a row error. Substitutions of a neighbouring letter could happen within the same column when the key for the substituted letter is in the same column as the key for the correct letter and is adjacent to the correct key, such as $father \rightarrow ragher$. This is called a column error.

Besides the row and column errors, transposition errors, doubling errors and alternation errors have also played a major role in determining the structure of the model.

A transposition error is the reversal of two adjacent letters, which is one of the most common and most interesting categories of errors, such as

$because \rightarrow becuase$ $\qquad\qquad$ $which \rightarrow whihc$

Transposition errors also involve adjacent keys (e and r, o and p), as in

$supremely \rightarrow supermely$

We also can see another interesting example where the four keystrokes on the right hand (n, space, o, n) have all been displaced with respect to the five left-hand keystrokes.

went down$\rightarrow$*wne todnw*

A doubling error occurs when a word contains a double letter, the wrong letter is sometimes doubled, such as

$look \rightarrow lokk$ $\qquad$ $school \rightarrow scholl$

Alternation reversal errors are akin to the doubling error, but with an alternating sequence. Such as:

$these \rightarrow thses$ $\qquad$ $there \rightarrow threr$

2.2.2 Levenshtein distance

Levenshtein distance (LD) [19], [16] is a measure of similarity between two strings s and w, that are referred to as the source string s and the target string w. The distance is the minimum number of single-symbol deletions, insertions, or substitutions required to transform s into w. The greater the Levenshtein distance, the more different the strings are.

For example,

If $s =$ "string" and $w =$ "string", then $\mathrm{LD}(s, w) = 0$, because no transformations are needed. The strings are already identical.

If $s =$ "string" and $w =$ "strang", then $\mathrm{LD}(s, w) = 1$, because one substitution (change $'i'$ to $'a'$) is sufficient to transform s into w.

Levenshtein distance is named after the Russian scientist Vladimir Levenshtein, who introduced it in 1965. It is also called *edit distance*. This distance has been used in such areas as spell checking, speech recognition, DNA analysis, plagiarism detection, etc. ([16], [9])

2.2.3 Dynamic programming

The classic algorithm [16], [20] for calculating the edit distance between two strings uses dynamic programming.

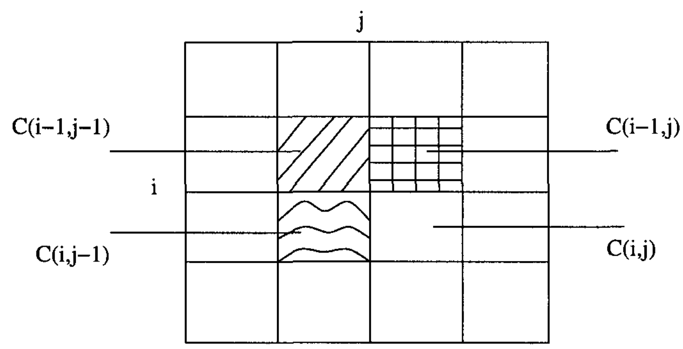


Figure 2.3: Table of Dynamic Programming

Suppose we are given two strings A and B where $|A| = n$, $|B| = m$, $A(i)$ is the ith character in A, and $B(j)$ is the jth character in B. In Figure 2.3, assume that $C(i, j)$ is the minimum cost of changing $A(1) \ldots A(i)$ to $B(1) \ldots B(j)$. There are four possibilities corresponding to three different edit operations:

delete: if $A(i)$ is deleted in the minimum change from A to B, we have $C(i, j) = C(i-1, j) + 1$;

insert: if the minimum change from A to B is the insertion of a character to match $B(j)$, then we have $C(i, j) = C(i, j-1) + 1$;

replace: if $A(i)$ is replacing $B(j)$, then $C(i, j) = C(i-1, j-1)+1$, if $A(i) \neq B(j)$;

match: if $A(i)$ is equal to $B(j)$, then we have $C(i,j) = C(i-1, j-1)$.

Now, we get to the formula for calculating $C(i,j)$.

The base cases are: C(0,0)=0 and

for(i=1 to n)

C(i,0) = i

and

for(j=1 to m)

C(0,j) = j

The general case is:

$$C(i,j) = min \begin{cases} C(i-1,j)+1 & \text{deletion,} \\ C(i,j-1)+1 & \text{insertion,} \\ C(i-1,j-1)+G(i,j) & \text{substitution.} \end{cases}$$

where $G(i,j) = 0$ if $A(i) = B(j)$,

$G(i,j) = 1$ if $A(i) \neq B(j)$;

Note that, each entry only depends on the entries immediately above it and to its left as illustrated in Figure 2.3.

In the dynamic programming algorithm, we maintain a matrix $C[1 \ldots n, 1 \ldots m]$ in which each entry $C[i,j]$ stores the minimum number of edit operations $A(i)/\lambda$, $\lambda/B(j)$,or $A(i)/B(j)$ required to transform the string composed of the first i symbols of A into the string composed of the first j symbols of B. Thus we need to know the values of $C[i-1,j], C[i,j-1]$, and $C[i-1,j-1]$. The last change can be determined according to which of the possibilities leads to the minimum value of $C[i,j]$.

The dynamic programming table for computing the edit distance between two strings A of length n and B of length m can be filled in time $\Theta(nm)$.

2.2.4 Trie

A digital tree (usually called a *trie* from re**trie**val [35], [36]) is a finite automaton with a tree structure useful for storing strings over an alphabet. The idea is that all strings sharing a common stem or *prefix* hang off a common node. When strings are words over $a \dots z$, a node has at most 26 children - one for each letter. More formally, each node of the trie contains the following fields: Character; Valid bit; An array of 26 pointers, one for each letter. The valid bit indicates if the node is a terminal or not. If it is, the value is 1, otherwise it is 0.

For example, given strings an, ant, all, boy, the corresponding trie is given in Figure 2.4.

In this study a trie is implemented as a linked-list in which each node has at most 26 child elements. The data structure of a trie for a dictionary is illustrated in Figure 2.5:

A binary search trie (**bst**) ([37]) is called a ternary tree where a search on letters is conducted like in a standard binary search tree over the alphabet set. Ternary search trees combine attributes of binary search trees and digital search tries. Like tries, they proceed character by character. Like binary search trees, they are space efficient, though each node has three children, rather than two. A search compares

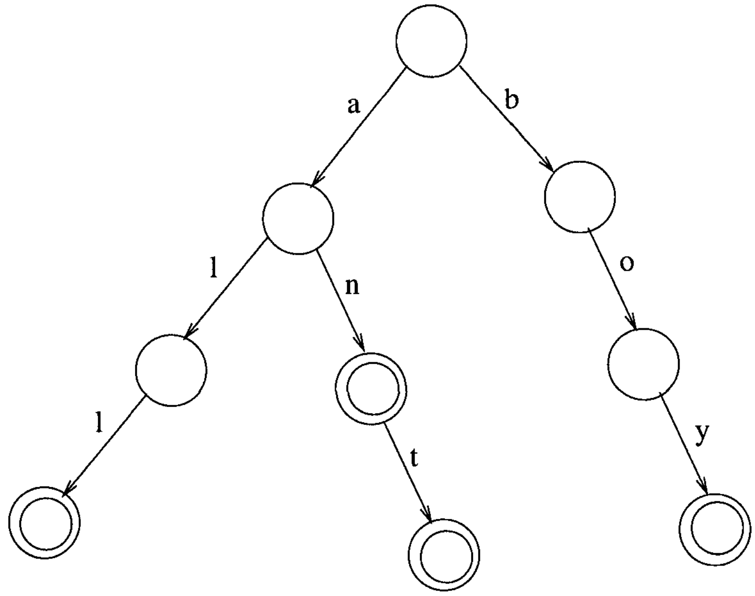
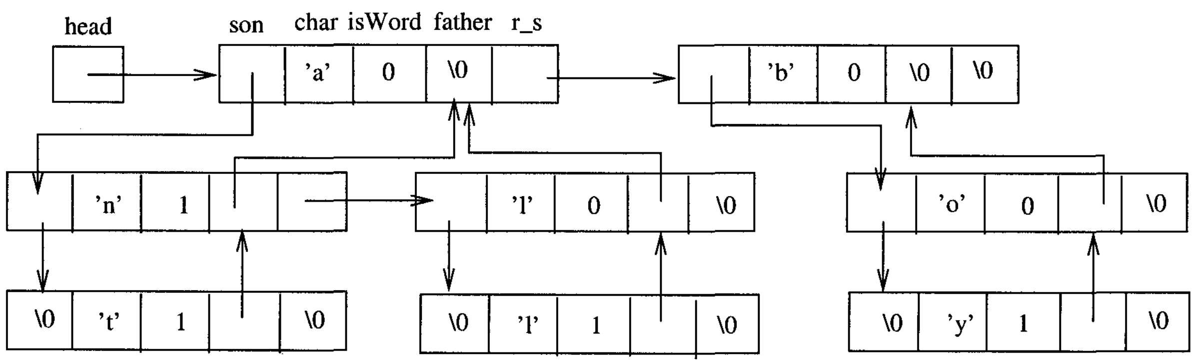


Figure 2.4: Regular Trie to Store Dictionary



Basic Data Structure defined for regular search trie:

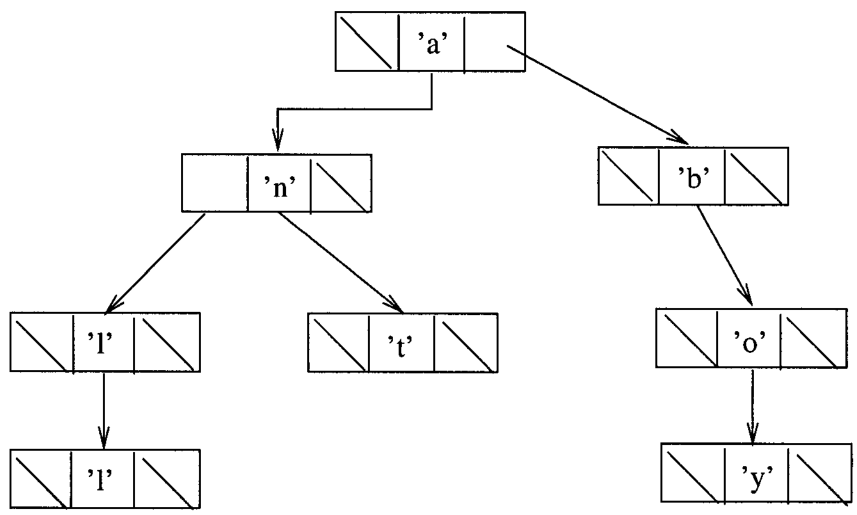
```
typedef struct TrieNode * Trie;

struct TrieNode{
        char  ch;
        Trie son;
        Trie r_s;
        Trie father;
        bool isWord;
        vector<double> distance;
};
```

Figure 2.5: Data Structure for Trie

the current character in the search string with the character at the node. If the search character is less, the search goes to the left child; if the search character is greater, the search goes to the right child. When the search character is equal, though, the search goes to the middle child, and proceeds to the next character in the search string. The process of searching in a ternary search trie with n strings for a string of length k requires at most $O(logn + k)$ comparisons.

The Figure 2.6 represent an example that store strings *ant, all, boy* by using ternary search trie.



Basic data Structure defined for ternary search trie:

```
typedef Struct   searchTrieNode * sTrie;
struct searchTrieNode{
        char ch;
        sTrie left;
        sTrie middle;
        sTrie right;
}
```

Figure 2.6: Data Structure of BST

2.2.5 Finite state machine

The AT&T Finite State Machine (*FSM*) Library [38] will be used as the finite state machine tool in this research.

The FSM library created by Mehryar Mohri and Michael D. Riley is a set of general-purpose software tools available for the Unix environment, for building, combining, optimizing, and searching in weighted (finite) automata *(WFA)* and weighted (finite) transducers *(WFST)*.

FSM includes about 30 stand-alone commands to construct, combine, determinize, minimize, search, and compose *WFA* and *WFST*. These commands manipulate *WFA* and *WFST* by reading from and writing to files or pipelines. The following example shows the commands that create *WFA*s and *WFST*s.

(1) The command that creates a *WFA* is

$$Fsmcompile - a.syms < b.stxt > b.fsa$$

where *a.syms* is a symbol file that stores all the symbols used in *b.stxt*. The file format is described as below:

a 1
b 2

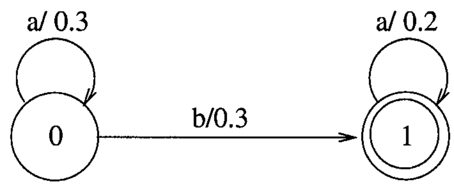
The text file *b.stxt* contains a textual representation of the *WFA*. The file format is described as below:

ST	DT	LB	CT
0	0	a	0.5
0	1	b	0.1
1	1	a	0.2
1			

ST: start state DT: destination state

LB: lable CT: cost

The file $b.fsa$ contains the *WFA* created by reading from the text file $b.stxt$. The graphic representation of $b.fsa$ is as below:



(2) The command that creates a *WFST* is

$$Fsmcompile - x.syms - x.syms - t < y.stxt > y.fst$$

where $x.syms$ is a symbol file that stores all the symbols used in $y.stxt$. The file format is described as below:

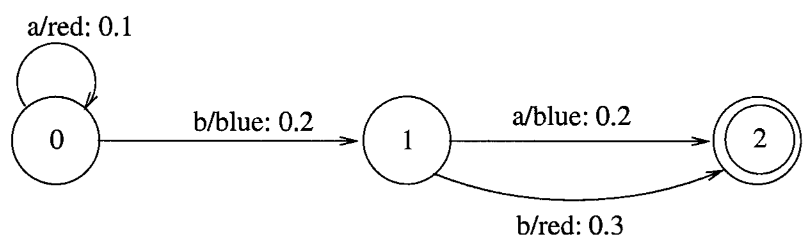
a 1
b 2
red 3
blue 6

The text file $y.stxt$ contains a textual representation of the *WFST*. The file format is described as below:

ST	DT	IL	OL	CT
0	0	a	red	0.1
0	1	b	blue	0.2
1	2	a	blue	0.2
1	2	b	red	0.3
1				

ST: start state DT: destination state
IL: input lable OL: output lable
CT: cost

The file $y.fst$ contains the $WFST$ created by reading from the text file $y.stxt$. The graphic representation of $y.fst$ is as below:



(3) The command for composition

Suppose we are given two WFSTs $c1.fst$ and $c2.fst$ as showing in Section 2.1 (Example of Composition on $WFST$). The command for composition between $c1.fst$ and $c2.fst$ is:

$$Fsmcompose\ c1.fst\ c2.fst > c.fst$$

$c.fst$ is the ocmposition of $c1.fst$ and $c2.fst$.

Chapter 3

Literature Survey

3.1 Introduction

The study of typing comprises a fascinating mixture of elements from motor skills and typewriter mechanics to anatomy and cognitive control structures.

The detection and correction of spelling errors is an integral part of modern word-processors. Most existing spelling error correction techniques focus on isolated words, without taking any information that might be gleaned from the textual context in which the string appears. Such isolated-word correction techniques are unable to detect real-word errors such as typographic, phonetic, cognitive, and grammatical errors. For descriptive purposes, Kukich (1992) [17] breaks the field down into three increasingly broader problems:

(1) **non-word error detection**: detecting spelling errors that result in non-word (such as *the* $\rightarrow$ *teh*).

(2) **isolated-word error correction**: correcting spelling errors that result in non-words such as correcting *teh* to *the*, but looking only at the word in isolation.

(3) **Context-dependent error detection and correction**: using the context to help detect and correct errors even if they accidentally result in actual words of English (real-word errors). Some of these errors result from typos ($there \rightarrow three, from \rightarrow form$); some result because the writers substituting the wrong spelling of a homophone or near-homophone ($dessert \rightarrow desert, piece \rightarrow peace$).

The working history for the first problem started in the early 1970s and continued into the early 1980s. During that period of time, a number of efficient pattern-matching and string comparison techniques were explored for deciding whether an input string appears in a predefined word list or dictionary ([7], [12], [18], [32], [33]). Work on the second problem began as early as in the 1960s and has continued into the present. Various general and special purpose correction techniques have been devised ([6], [27], [34], [5], [3]). Work on the third problem spanned from the early 1980s to the present ([21]).

In this chapter, we are going to describe several spelling error models, error detection and correction methods corresponding to each of these three problems. However, the existing spelling correction techniques are limited in terms of their scopes and special cases.

3.2 Techniques for Correcting Words in Text

Research has focused progressively on the three problems mentioned in Section 3.1 for correcting words in text. In response to the first problem (non-word error detection), *n-gram* analysis and dictionary lookup methods have been developed for detecting spelling errors that result in non-words. With respect to the second problem (isolated-word error correction), some error models have been developed. For the third problem (Context-dependent error detection and correction) , statistical-language models have been developd.

3.2.1 Nonword error detection

N-grams analysis and dictionary lookup are the two main techniques for the nonword error detection problem. Dictionary lookup technique is a straightforward task. N-grams refers to n consecutive letters in a word or string. N-gram error analysis techniques work by examining each N-gram in an input string and looking it up in a precompiled table of N-gram statistics to ascertain either its existence or its frequency. If a non-existent or rare N-gram is found the word is flagged as a misspelling, otherwise not. N-grams statistics initially played a central role in text recognition techniques while dictionary-based methods dominated spelling correction techniques.

N-gram techniques usually require either a dictionary or a large corpus of text in order to precompile an N-gram table. The simplest N-gram table is called a binary bigram array and is a two-dimensional array of size 26×26 whose elements represent all possible two-letter combinations of the alphabet. The value of each element in the array is set to either 0 or 1 depending on whether that bigram occurs in at least one word in a predefined lexicon or dictionary.

Errors made by optical character recognition (OCR) devices typically confuse characters with similar features, such as O and D, S and 5, t and f, or m and n. The N-gram analysis technique has proven useful for detecting such errors because they tend to result in improbable N-gram. For example, Morris and Cherry (1975) [25] used digram frequencies to convert an unknown text word to the dictionary word that it most closely resembles. Digram frequency tables are used to make the most probable substitution for this. The new word is then looked up in the dictionary and the result will be repeated until a valid word is created. This method only applies to substitution errors.

3.2.2 Isolated-word error correction research

Isolated-word error correction techniques have been developed for the problem of correcting words in text. Some of these correction methods include allowing the user to write over an error, allowing for keyboard correction, and providing n best matches for the user to select from. We can group isolated-word error correction techniques into the following main classes:

(1) minimum edit distance techniques (see [6] for instance);

(2) similarity key techniques (see [27] for instance);

(3) rule-based techniques (see [34] for instance);

(4) probabilistic techniques;

In this research, we focus on the probabilistic techniques.

3.2.3 Probabilistic techniques for isolated-word correction

Probabilistic Models [13]

The issues in finding spelling errors in text can be explored using the Bayes Rule and the noisy channel model. The Bayes rule and its application to the noisy channel model used in data communications provide the probabilistic framework for many problem-solving issues such as detection and correction of spelling errors, speech recognition, etc. ([5], [3], [21], [11])

Figure 3.1 shows how the noisy channel model works.

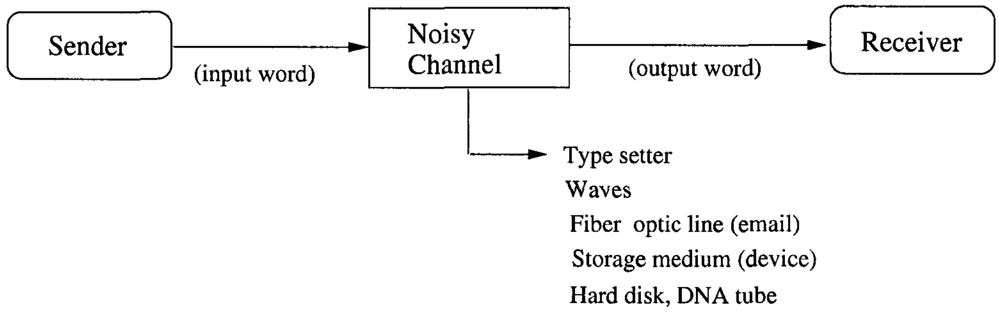


Figure 3.1: The Noisy Channel Model

The problem of spelling correction for typing or for Optical Character Recognition (OCR), can be modeled as the problem of mapping one string of symbols to another. Given an incorrect sequence of letters in a misspelled word, we need to figure out the correct sequence of letters in the correctly spelled word. The noisy channel introduces noise which makes it hard to recognize the true word. We want to build a model of the channel and figure out how to modify the misspelled word and hence recover the true word.

We use Baysian classification for the noisy channel model. In Baysian classification, we are given some observation and we want to determine which set of classes it belongs to. For spelling error detection, the observation might be the string of letters that constitutes a possible-misspelled word and we want to classify this observation to a particular word. For example, the word "*separate*", no matter how this word is misspelled, we would like to recognize it as "*seperate*".

Given an input word "*acress*", we want to find the words corresponding to this string. Bayesian classification considers all possible words and chooses the word which is most probable given the observation we have ("*acress*") out of the possible words. That is we want to find out of all words in the dictionary, the single word such that $P(word|observation)$ is the highest. The equation for picking the best word given is:

$$W_{max} = argmax_{w \in V} P(w|s) \tag{3.1}$$

Where,

w : our estimate of the correct w

s : the observation string

V : vocabulary

The function $argmax_x f(x)$ returns the x where $f(x)$ is maximized.

We can use Bayes's rule to rephrase $P(w|s)$ in terms of three other probabilities.

$$P(w|s) = \frac{P(s|w)P(w)}{P(s)} \tag{3.2}$$

Thus we can get the following equation by substituting the above into Equation

3.1:

$$W_{max} = argmax_{w \in V} \frac{P(s|w)P(w)}{P(s)} \tag{3.3}$$

In this equation, the source model $P(w)$ is the probability of occurrence of the word itself, which can be estimated by the frequency of the word and $P(s|w)$ the noisy channel model is the probability that the speller transform the word w into the word s. We will see how to compute $P(s|w)$ later. The probability $P(s)$ of the observed string is harder to estimate; however, we can ignore it as we are maximizing over all words and $P(s)$ doesn't change for each word. Therefore we can replace Equation 3.3 by

$$W_{max} = argmax_{w \in V} P(s|w)P(w) \tag{3.4}$$

From Equation 3.4, we can see that the most probable word, given some observation s, can be computed by taking the product of the source probability $P(w)$ and the noisy channel probability $P(s|w)$ for each word w, and choosing the word with the highest product.

The noisy channel model assumes that the natural language text is generated as follows: first a person chooses an input word w, according to the probability distribution $P(w)$ (the source); then the person attempts to output the word w, but the noisy channel induces the person to output string s instead, according to the distribution $P(s|w)$ (the channel). For the same observed string s, the probability of different input strings are different. In computer typesetting, for example, under typical circumstances such as people's knowledge, typing skill, keyboard layout, etc., we would expect the following order of probabilities:

$$P(one|one) > P(oen|one) > P(two|one).$$

Church and Gale's method

In 1991, Church and Gale [5] described a program named *correct* which corrects single-error misspellings by using a noisy channel algorithm based on the equation 3.4. In their study, Church and Gale assume that the correct word differs from the misspelling just by a single insertion, deletion, substitution or transposition. Their program corrects the words rejected by the program named *spell* [22] by generating a list of potential correct words ranked according to Equation 3.4 and choosing the highest-ranked one. A database of genuine errors extracted from a 44 million-word corpus of AP newswire stories is used as the training set in the program.

Computing the likelihood term $P(s|w)$ (error model) is difficult as the probability of a word being mistyped depends on several external factors, such as the different typists; and how familiar they are with the keyboard, whether one of their hands happens to be more tired than the other, etc. Luckily, it can be estimated pretty well since the most important factors in predicting an insertion, deletion, substitution or transposition are all simple local factors, such as the identity of the correct letters itself, the surrounding context, and the way that the letter was misspelled. For instance, the letters m and n are often substituted for each other. This is partly because of the fact that these two letters are pronounced similarly and they are next to each other on the keyboard, and partly because of the fact that they occur in similar contexts.

The channel probabilities $P(s|w)$ can be computed from four 26×26 confusion matrices, each of which represents the number of times one letter is incorrectly used in place of another: **(1) sub[x,y]**, the number of times that correct letter $'y'$ is typed as incorrect letter $'x'$ (y/x). For example, the cell [o,e] in a substitution confusion matrix will give the count of times that e is substituted by o. **(2) ins[x,y]**, the number of times that correct letter $'x'$ is typed as $'xy'$ (x/xy). For example the cell [t,s] in an insertion confusion matrix gives the count of times that (t/ts) appears. **(3) del[x,y]**, the number of times that the letters $'xy'$ are typed as $'x'$ (xy/x). **(4) trans[x,y]**, the number of times that $'xy'$ is typed as $'yx'$ (xy/yx). The probability of inserting or deleting a character is conditioned on the letter appearing immediately to the left of that character.

Church and Gale estimated $P(s|w)$ using the previous four matrices as follows:

$$P(s|w) \approx \begin{cases} \mathbf{del}[w_{p-1}, w_p]/\mathbf{count}[w_{p-1}, w_p], & \text{if deletion} \\ \mathbf{ins}[w_{p-1}, s_p]/\mathbf{count}[w_{p-1}], & \text{if insertion} \\ \mathbf{sub}[s_p, w_p]/\mathbf{count}[w_p], & \text{if substitution} \\ \mathbf{trans}[w_p, w_{p+1}]/\mathbf{count}[w_p, w_{p+1}], & \text{if transposition} \end{cases}$$

where w_p is the pth character of the word w, s_p is the pth character of the typed word and p is where the edit operation occurs. Church and Gale's method only considers a single edit operation between s and w, p is unique.

count[x,y] and **count[x]** represent the number of times that $'xy'$ and $'x'$ appear in the training set.

In their paper, Church and Gale considered as the candidate source words only those words that are a single basic edit away from s, using the edit set as described before. The Church and Gale mode is essentially a weighted Leveshtein technique. In their proposed error model, they assigned different probabilities to each unique edit, which makes the model a weighted Levenshtein technique.

Brill and Moore's Method

In 2000, Eric Brill and Robert C. Moore [3] presented a new channel model for spelling correction. The new channel model they described is based on generic string-to-string edits. It solves the problem of automatically training a system to correct generic single word spelling errors.

The Church & Gale error model mentioned above is based on the single edit operation between two strings, which is the minimum number of single edit operation insertions, substitutions, deletions and transpositions. The Brill and Moore error model is a much more generic error model that allows all edit operations of the form x/y, where $x, y \in \Sigma^*$ for some alphabet, Σ is an alphabet. It conditions the position where the edit operation occurs in the string by the location of the substring x in the start, middle, or end of the source word.

In the misspelling correction process, they first trained the error model to get a set of probabilities $P(x/y)$ and then they applied the error model to non-real word spelling errors. Compared with Church and Gale's weighted Levenshtein distance technique, a 52% reduction in spelling correction error rate was achieved by using the improved error model. With a language model, their error model gave a 74%

reduction in error. One exciting future of this research is to obtain error models that adapt to an individual or subpopulation. More details for Brill and Moore's model will be discussed in section 6.1.

Touranova and Moore's Pronunciation Modeling

In 2002, Kristina Toutanova and Robert C. Moore [31] presented a method that incorporates word pronunciation information in a noisy channel model of spelling error correction problem. Spelling errors are generally grouped into two classes [25]: typographic and cognitive. Typographic errors are mostly errors related to the keyboard. Cognitive errors are those misspellings whose pronunciation is same as the correct word. Cognitive errors occur when the writer does not know how to spell a word.

In [31], the authors took an approach to model phonetic errors explicitly by building a separate error model for cognitive errors. Two different error models were built by using the Brill and Moore learning algorithm. One was a letter-based model (LTR) which is exactly the Brill and Moore model. The other was a phone-sequence-to-phone-sequence error model (PH). In PH, the misspelled/correct word pairs were converted into pairs of pronunciations of the misspelled and the correct words, which were then run gainst the Brill and Moore algorithm. Finally these two error models were combined as a log linear model.

In their paper, Toutanova and Moore presented a method that uses word pronunciation information to improve spelling correction accuracy. Compared to the letters-only model, the combined model reduces the error rate over 23% for *1-Best* correction, and even higher for *2-Best, 3-Best* and *4-Best.* Here the *n-best* list will contain the n most probable correct words for a misspelling.

3.2.4 Context-dependent word correction techniques

Reviewing the methods we described so far for isolated-word error correction problem, there always remains a residual class of errors that is beyond the capacity of those techniques to handle. This is the class of real-word errors in which one correctly spelled word is substituted for another correctly spelled one. In this section, we will describe a statistical method for the context-dependent word correction problem.

Statistically based error detection and correction

Statistical language models (SLM) are essentially tables of conditional probability estimates for some or all words in a language that specify a word's likelihood to occur within the context of other words. In the statistical language-modeling approach, contextual information can be used to help set it expectations for possible word choices. Thus, low-probability word sequences can be used to detect real-word errors, and high-probability word sequences to rank correction candidates.

Mayes and Damerau's Method

Mays and Damerau [21] discussed how to detect and correct real-word spelling errors by using word trigrams. They employed the noisy channel model to correct spelling errors. This model is similar to the model used in speech recognition as we discussed before (equation 3.4).

Here, $P(w)$ is the probability that the complete sequence of words $w = w_1, \dots, w_n$, will be produced by the text generator. The probability of w is defined as:

$P(w) = P(w_1) \times P(w_2|w_1) \times \dots \times P(w_i|w_{i-1}w_{i-2}) \times \dots \times P(w_n|w_{n-1}w_{n-2})$.

The quantity $P(s|w)$ is the probability that the speller and typist transform the sequence of words w into another sequence of words $s = s_1, \dots, s_m$.

In this error model, each output word s_i is considered to occur in its correct location without depending on adjacent words. That is as $n = m$, the following equation is obtained:

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

For each $P(s_i|w_i)$, if a speller could produce s_i when w_i is intended, then s_i is in the confusion set $\mathcal{C}$ which might include all simple misspellings of the word w_i. Here, the confusion set is determined by applying exactly one of four basic edit operations described before. The error model $P(s_i|w_i)$ can be computed as:

$$P(s_i|w_i) = \begin{cases} \alpha, & if s_i = w_i, \\ \frac{1-\alpha}{|\mathcal{C}|-1}, & otherwise. \end{cases}$$

where,

- $|\mathcal{C}|$ is the number of words in confusion set $\mathcal{C}$.

- The constant α represents the prior probability of a typed input word, it can be determined by experimentation.

- $1-\alpha$ represents the remaining probability, which is equally divided among the other words in the confusion set.

If α is set too high the result will have the tendency to retain typed input words even if they are incorrect. If α is set too low the result will tend to change typed input words even if they are correct. In Mays and Dameraus's study, they tested a range of values for α and found that the optimum value is between 0.99 and 0.999.

Chapter 4

General Methodology

In Chapter 3, we described many situations in text recognition where decisions have to be made based on incomplete or uncertain information and discussed several error models. In this chapter, we will introduce a general methodology for defining error models that allow us to describe not only human spelling errors in the texts but also various errors in the real world such as speech errors, DNA computing errors[1], etc..

In text recognition, uncertainty and incompleteness arise from a number of sources, such as contextual effects, homophones or typist variabilities. Finite-state stochastic modeling is a flexible general method that handles such situations. This approach consists of employing a probabilistic type of a WFST for the uncertainty or incompleteness of information. Research in this field is motivated by the fact that deterministic automata *(DFA)* are not suitable for modeling even the simplest forms of behaviour, such as the acquisition of a conditioned reflex [2]. Thus, the finite-state

[1]These can be random substitution, insertion and deletion nucleotide errors in the DNA strands that participate in DNA computations.

stochastic model is a particularly suitable approach to our general error model in text recognition.

An abstract model for these situations of uncertainty is that there are two sequences of random variables: $y(1), y(2), y(3), \ldots, y(t)$ and $x(1), x(2), x(3), \ldots, x(t)$. The *x's* represent the sequence that we wish to know, but are not able to observe directly. The *y's* represent the sequence which are related to the *x's* and which we can observe or we have already deduced by other means. The stochastic modeling consists of formulating a probabilistic model that receives a sequence of *y's* and produces a sequence of *x's* based on the sequence of *y's*. When a sequence of *y's* is observed, certain techniques are used to find the sequence of *x's* which best fits the observed sequence of *y's*. That is, the sequence of *x's* according to the model is the sequence which is the most likely to produce the observed sequence of *y's*.

4.1 The Classic Stochastic Automata

The classic stochastic system is considered as working on a discrete time-scale. It uses states, input signals and output signals, in the same way as deterministic automata *(DFA)*. Thus, in every step, exactly one signal is received, exactly one signal is emitted, and exactly one state occurs. In the stochastic system, for a given situation, the external and internal reactions of the system are not uniquely determined, but for every imaginable reaction, there is only a certain probability that would output y and enter the state z for input x at the same time. This does not imply any loss of generality if we assume that this probability only depends on certain situations and not on the number of step t in which this situation occurs, or on the past history of the situation as it happens in Markov Chains.

Definition of Classic Stochastic Automata

A stochastic automaton $\zeta = [\Sigma_X, \Sigma_Y, Z, H]$ is defined as follows:

(1) Σ_X, Σ_Y, Z are arbitrary non-empty sets and

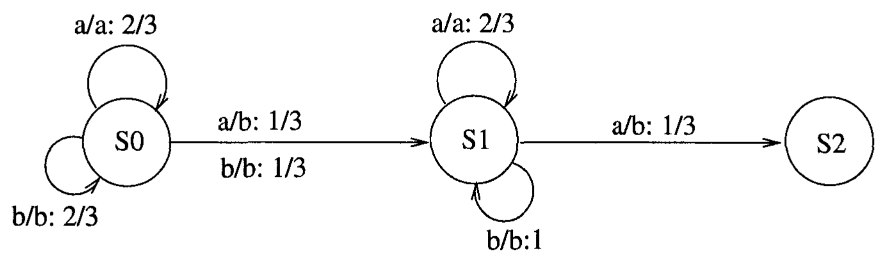
(2) H is a function defined on $Z \times \Sigma_X$, such that each $H[z, x]$ is a discrete probability measure over $\Sigma_Y \times Z$, that is, $\sum_{y \in \Sigma_y} \sum_{z_1 \in Z} H[z, x_i](y_i, z_1) = 1$

The elements $x_i \in \Sigma_X$ are called input letters of ζ, and Σ_X is the input alphabet of ζ. The elements $y_i \in \Sigma_Y$ are the output letters of ζ, and Σ_Y is the output alphabet of ζ. The elements $z \in Z$ are called the states of ζ. The stochastic automaton ζ operates on a discrete time scale in a countable infinite number of steps $t = 1, 2, \ldots$. In each step t, ζ receives exactly one input signal, generates exactly one output signal and reaches exactly one state. The function $H[z, x_i](y_i, z_1)$ over ζ describes the probability that in current state z the signal y_i would be generated and the next state will be z_1, if the input signal is x_i. The value of $H[z, x](y, z')$ is the probability that the stochastic automaton ζ output the string y if the input string is x, the start state is z and the final state is z'. It is assigned as follows:

$$
H[z,x](y,z') = \begin{cases} 1, & if x = y = \lambda; \\ \prod_{i=1}^{n} H[z_i, x_i](y_i, z_j), where \begin{cases} j = i+1 & x_i \neq y_i, \\ j = i & otherwise, \end{cases} & \\ \qquad if x = x_1 \ldots x_n, y = y_1 \ldots y_n, n \geq 1, z_i \in Z; & \\ 0, & otherwise. \end{cases}
$$

Example

Figure 4.1 shows an example of classic stochastic automaton.



H[S0,a](a,S0) + H[S0,a](b,S1) = 2/3 +1/3 = 1
H[S0,b](b,S0) + H[S0,b](a,S1) = 2/3 +1/3 = 1
H[S1,a](a,S1) + H[S1,a](b,S2) = 2/3 +1/3 = 1
H[S1,b](b,S1) = 1

Figure 4.1: Example of Classic Stochastic Automaton

If we send the input string *aaba* to this stochastic automaton, then the probability of the possible output string *abbb* is:

$$
\begin{aligned}
H[S_0, aaba](abbb, S_2) &= H[S_0, a](a, S_0) \times H[S_0, a](b, S_1) \times H[S_1, b](b, S_1) \times H[S_1, a](b, S_2) \\
&= \frac{2}{3} \times \frac{1}{3} \times 1 \times \frac{1}{3} \\
&= \frac{2}{27}.
\end{aligned}
$$

The above example and the definition of classic stochastic automata illustrate that when we view the classic stochastic automata as channels they only allow substitution errors. However, in Chapter 2, we have seen that there are three common errors in text recognition: *substitution*, *insertion* and *deletion*. Therefore, the classic stochastic automaton is not adequate to describe all of them. We will introduce a new type of stochastic automata in the next section.

4.2 Definition of Channel

From Section 4.1, we know that the classic stochastic automaton only can describe substitution errors. In this section, we introduce a new type of stochastic automaton which we call a *channel* that will be able to describe all types of errors. A *channel* describes error behaviours in different situations. For example, a *channel* could describe edit operations in spelling error correction; speech errors in speech recognition DNA string errors in DNA computing, etc.

A *channel* is a particular type of *weighted finite transducer(WFST)* that allows us to describe formally the combination of errors that are permitted in some information processing applications. As we described in Chapter 2, a $WFST$ C consists of

- An input alphabet Σ_X, an output alphabet Σ_Y;
- A set $S = \{S_1, S_2, \ldots, S_n\}$ of states $(n \geq 1)$;
- Labelled transitions $(S_i, x_i/y_i, S_j)$ with $S_i, S_j \in S$ ad $x_i, y_i \in \Sigma^*$ and;

- A function w that maps any transition $(S_i, x_i/y_i, S_j)$ to a number $w(S_i, x_i/y_i, S_j)$, called the weight of the transition.

A *channel* is a $WFST$ with the following restrictions:

- Allowoble ransitions are of the form $(S_i, x_i/y_i, S_j)$ where $S_i, S_j \in S$, $x_i \in \Sigma_X \cup \lambda$, and $y_i \in \Sigma_Y \cup \lambda$ (note that this includes $x_i/y_i = \lambda/\lambda$)

- All weights of transitions are positive numbers

- For every state S_i and for every input symbol $x_i \in \Sigma_X$, $H[S_i, x_i]$ is a discrete probability measure on $(E \cup \{\lambda/\lambda\}) \times S$ such that

  1. $H[S_i, x_i](x_i/y_i, S_j) = w(S_i, x_i/y_i, S_j)$ if $(S_i, x_i/y_i, S_j)$ is a transition, or $H[S_i, x_i](x_i/y_i, S_j) = 0$ otherwise;

  2. $H[S_i, x_i](\lambda/y_i, S_j) = w(S_i, \lambda/y_i, S_j)$ if $(S_i, \lambda/y_i, S_j)$ is a transition, or $H[S_i, x_i](\lambda/y_i, S_j) = 0$ otherwise;

  3. $H[S_i, x_i](x_i'/y_i, S_j) = 0$ for all $x_i' \neq x_i$; and

  4. $$\sum_{y_i \in \Sigma_Y \cup \lambda, S_j \in S} (H[S_i, x_i](x_i'/y_i, S_j) + H[S_i, x_i](\lambda/y_i, S_j)) = 1 \qquad (4.1)$$

Note that for every pairs of states S_i, S_j and output y_i, the quantity $H[S_i, x_i](\lambda/y_i, S_j)$ is independent of x_i, that is, $H[S_i, x_i](\lambda/y_i, S_j) = H[S_i, x_i'](\lambda/y_i, S_j)$ for all $x_i, x_i' \in \Sigma_X$. This means that the probability of moving from state S_i to state S_j and output y_i without consuming the input is independent of the input. Thus, $H[S_i, \lambda](\lambda/y_i, S_j) = H[S_i, x_i'](\lambda/y_i, S_j)$ for all $x_i' \in \Sigma_X$.

A channel with the above definition is also a particular type of *wfse-system* [14], which is a $WFST$ in which each labelled transition either is one edit operation or λ/λ.

Let C be a stochastic transducer and E be the set of basic edit operations x_i/y_i. A *C-event* ζ is an expression of the form $e_1S_1 \ldots e_nS_n$, where $n \geq 1$, $e_i = x_i/y_i$, each $e_i \in E \cup \lambda/\lambda$, $S_i \in S$ and $inp(e_1 \ldots e_n) \neq \lambda$. Intuitively, $e_1S_1 \ldots e_nS_n$ represents the event that the channel C will perform the edit operation e_1 and move to state S_1, then perform e_2 and move to S_2, etc. For each state $S_i \in S$ and each *C-event*, the number $H_{S_0}(\zeta)$ is the probability of the event ζ from start state S_0, is defined as:

$$H_{S_0}(\zeta) = H[S_0, x_1](x_1/y_1, S_1) \times H[S_1, x_2](x_2/y_2, S_2) \ldots \times H[S_{n-1}, x_n](x_n/y_n, S_n).$$

The example below illustrates this theory.

Example

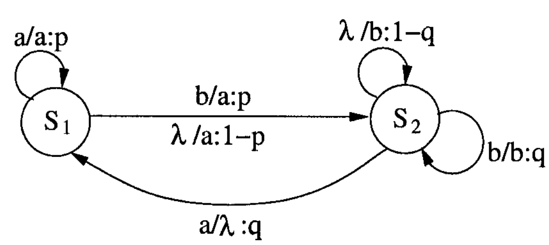


Figure 4.2: Example of Channel

In Figure 4.2, we have

$$\begin{aligned} H[S_1, a](a/a, S_1) &= p, \\ H[S_1, a](\lambda/a, S_2) &= 1 - p, \end{aligned}$$

$$H[S_1,a](x_i/y_i,S_j) = 0, \textit{ in all other cases}$$

Therefore,

$$\sum_{y_i \in \Sigma_Y \cup \lambda, S_j \in S} (H[S_1,a](a/y_i,S_j) + H[S_1,a](\lambda/y_i,S_j)) = 1.$$

Also,

$$\begin{aligned} H[S_1,b](b/a,S_2) &= p, \\ H[S_1,b](\lambda/a,S_2) &= 1-p, \\ H[S_1,b](x_i/y_i,S_j) &= 0, \textit{ in all other cases} \end{aligned}$$

Therefore,

$$\sum_{y_i \in \Sigma_Y \cup \lambda, j \in S} (H[S_1,b](b/y_i,S_j) + H[S_1,b](\lambda/y_i,S_j)) = 1.$$

Also,

$$\begin{aligned} H[S_2,a](a/\lambda,S_1) &= q, \\ H[S_2,a](\lambda/b,S_2) &= 1-q, \\ H[S_2,a](x_i/y_i,Sj) &= 0, \textit{ in all other cases} \end{aligned}$$

Therefore,

$$\sum_{y_i \in \Sigma_Y \cup \lambda, j \in S} (H[S_2,a](a/y_i,S_j) + H[S_2,a](\lambda/y_i,S_j)) = 1.$$

Also,

$$\begin{aligned} H[S_2,b](b/b,S_2) &= q, \\ H[S_2,b](\lambda/b,S_2) &= 1-q, \\ H[S_2,b](x/y,S_j) &= 0, \textit{ in all other cases} \end{aligned}$$

Therefore,

$$\sum_{y_i \in \Sigma_Y \cup \lambda, j \in S} (H[S_2, b](b/y_i, S_j) + H[S_2, b](\lambda/y_i, S_j)) = 1.$$

Thus, some *C-events* can be generated:

$H_{S_1}((a/a)S_1(b/a)S_2(\lambda/b)S_2(b/b)S_2) = p \times p \times (1-q) \times q > 0,$

$H_{S_1}((a/a)S_1(a/b)S_2) = H[S_1, a](a/a, S_1) \times H[S_1, a](a/b, S_2) = p \times 0 = 0.$

Let Z_b be the set of channel event ζ such that $inp(\zeta) = b$. Then

$$\begin{aligned}
\sum_{X \in Z_b} H_{S_1}(X) &= H_{S_1}((b/a), S_2) \\
&\quad + H_{S_1}((\lambda/a)S_2(b/b)S_2) \\
&\quad + H_{S_1}((\lambda/a)S_2(\lambda/b)S_2(b/b)S_2) \\
&\quad + H_{S_1}((\lambda/a)S_2(\lambda/b)S_2(\lambda/b)S_2(b/b)S_2) + \ldots \\
&= p + (1-p) \times \sum_{r=0}^{\infty}(1-q)^r \times q \\
&= p + (1-p) \times \frac{1}{1-(1-q)} \times q \\
&= p + (1-p) \times \frac{1}{q} \times q \\
&= p + 1 - p \\
&= 1
\end{aligned}$$

The set of possible outputs of the channel when the input is b is $a \cup ab^*b$

Definition An *error model* is a set of channels. Intuitively, an error model is the set of possible channels that appear to model the errors in a particular information processing application.

From the definition of a channel that each transition $S_1(x_i/y_i)S_2$ has input label x_i and output label y_i. However, for some channels, transitions $S_1(x/y)S_2$ with $|x| \geq 2$ or $|y| \geq 2$ are needed, e.g., the channels of Church and Gale and Mays and Damerau error models. Therefore, we need to convert each transition $S_1(x/y)S_2$ to sequence of transitions with single-symbol labels.

Convert the transition

If given a transition $S_1(x/y)S_2$ with $|x| = n$ and $|y| = m$, where n or m is greater than 1, and $H[S_1, x](x/y, S_2) = P$, we conver $S_1(x/y)S_2$ to a sequence of transitions with single labels as below:

$$S_1(x/y)S_2 = \begin{cases} S_1(x_1/y_1)S_2(x_2/y_2)S_3\ldots(x_n/y_m)S_{n+1} & \text{n=m}, \\ S_1(x_1/y_1)S_2\ldots(x_m/y_m)S_{m+1}(x_{m+1}/\lambda)S_{m+2}\ldots(x_n/\lambda)S_{n+1} & n > m, \\ S_1(x_1/y_1)S_2\ldots(x_n/y_n)S_{n+1}(\lambda/y_{n+1})S_{n+2}\ldots(\lambda/y_m)S_{m+1} & n < m, \end{cases} \tag{4.2}$$

In each of these three cases, only the probability of the first transition $S_1(x_1/y_1)S_2)$ is P and all others are all equal to 1. Therefore,

$$H[S_1, x](x/y, S_2) = \prod_{i=1}^{mx(m,n)} H[S_i, x_i](y_i, S_{i+1}) = P$$

Assign the probabilities

In rule 4.1, we know

$$\sum_{y_i \in \Sigma_Y \cup \lambda, S_j \in S} (H[S_i, x_i](x_i'/y_i, S_j) + H[S_i, x_i](\lambda/y_i, S_j)) = 1$$

Therefore, we have

$$\sum_{y_i \in \Sigma_Y \cup \lambda, S_j \in S} H[S_i, x_i](x_i/y_i, S_j) = P_1,$$

And

$$\sum_{y_i \in \Sigma_Y \cup \lambda, S_j \in S} H[S_i, x_i](\lambda/y_i, S_j) = P_2,$$

where $0 \leq P_1 \leq 1$, $0 \leq P_2 \leq 1$, $P_1 + P_2 = 1$.

4.3 Error Correction with a Given Channel

Channel Error Correction Problem

Problem Definition Given a set of words D called the *dictionary*, a channel C and a channel output y, find a channel event ζ of C with the highest probability such that $inp(\zeta) \in D$ and $out(\zeta) = y$.

Although we focus on spelling errors in this research, the above problem definition also applies to other information processing applications.

Let $w = inp(\zeta) \in D$. For every $w' \in D$ and for every channel event ζ' with $inp(\zeta') = w', out(\zeta') = y$, we have

$$H[S_0, w](\zeta) \geq H[S_0, w'](\zeta')$$

This formula says that the probability of transforming w to y with the channel event ζ is greater than or equal to, any other transformations of $w' \in D$ into y with some channel event ζ'.

The *wfse-system* corresponding to a given channel

Given a channel C, a *wfse-system* B can be defined as:

Given states S_i, S_{i+1} of C and label x_i/y_i of C, we have the transition $S_i(x_i/y_i)S_{i+1}$ in B provided the probability

$H[S_i, x_i](x_i/y_i, S_{i+1}) > 0$

The *cost* of each transition in B is:

$C_B(S_i(x_i/y_i)S_{i+1}) = -\log(H[S_i, x_i](x_i/y_i, S_{i+1}))$.

The *cost* of a path $q_0(x_1/y_1)q_1 \dots (x_n/y_n)q_n$ in B is the quantity:

$\sum_{i=1}^{n} C_B(q_{i-1}(x_i/y_i)q_i)$.

Given a *wfse-system* B, we can define the *wfse-system* B^{-1} to be exactly the same as B only with the following change:

If $S_i(x_i/y_i)S_{i+1}$ is in B then $S_i(y_i/x_i)S_{i+1}$ is in B^{-1}.

The string to regular-language correction problem addressed in [14] is related to the channel error correction problem.

In the string to regular-language correction problem, we are given a string s, an *NFA* A and a *wfse-system* B. The language $L(A)$ is supposed to contain all the "syntactically correct words". We want to compute an e-string h that describes the edit operations permitted by B that would transform s to a syntactically correct word with the minimum cost. If we construct the $(|s| + 1)$-state automaton A_s to accept string s, then we can use the λ-*NFA* $A_s \circ B \circ A$ to solve this problem.

Channel Correction Algorithm

In the channel error correction problem, an NFA A_D can be created to store the dictionary D, the channel output y will be stored in a deterministic automaton A_y. As we know, if $S_i(x_i/y_i)S_{i+1}$ is in *wfse-system* B, then $S_i(y_i/x_i)S_{i+1}$ is in B^{-1}.

The correction of the channel output y over the dictionary D can be determined by finding the channel event ζ with the minimum cost in the weighted directed graph $A_y \circ B^{-1} \circ A_D$ such that $inp(\zeta) = y$ and $out(\zeta) \in D$. This also can be considered as finding a *path* p in the weighted automaton $A_y \circ B^{-1} \circ A_D$ such that p has the smallest cost with $inp(p) = y$ and $out(p) \in D$.

To see this, first note that $A_y \circ B^{-1}$ can be considered as the λNFA accepting all words w such that w is an output of B^{-1} when y is used as input, that is, $w \in B^{-1}(y)$. Then $(A_y \circ B^{-1}) \circ A_D$ represents the set of all words w as above that belong to D as well. Hence, a minimum cost path $q_0(y_1/x_1)q_1 \ldots (y_n/x_n)q_n$ in $A_y \circ B^{-1} \circ A_D$ defines the word $w = x_1 \ldots x_n$ in D that would result from $y = y_1 \ldots y_n$ via the *wfse-ssystem* B^{-1}. Equivalently, $q_0(x_1/y_1)q_1 \ldots (x_n/y_n)q_n$ is a path of B of minimum cost $\sum_{i=1}^{n} w_B(q_{i-1}(x_i/y_i)q_i)$ such that $x_1 \ldots x_n \in D$ and $y_1 \ldots y_n = y$. Equivalently again, $\zeta = (x_1/y_1)q_1 \ldots (x_n/y_n)q_n$ is a channel event of C with the highest probability $H_{q_0}(\zeta)$ such that $x_1 \ldots x_n \in D$ and $y_1 \ldots y_n = y$, which solves the channel error correction problem.

AT&T Tool

From Chapter 2, we know that the AT&T FSM library provides tools to describe and manipulate finite state automata. Therefore, we can use these tools to describe A_y, A_D and B^{-1} and compute $A_y \circ B^{-1} \circ A_D$.

The FSM command used to create A_y (if $A_y = oen$) is:

$$Fsmcompile - idic.syms < oen.stxt > oen.fsa$$

The FSM command used to create A_D is:

$$Fsmcompile - idic.syms < dic.stxt > dic.fsa$$

The FSM command used to create B^{-1} is:

$$Fsmcompile - idic.syms < model.stxt > model.fsa$$

The FSM command used to create $A_y \circ B^{-1}$ is:

$$Fsmcompose\ oen.fsa\ model.fst > basic.fst$$

The FSM command used to create $basic.fst \circ A_D$ is:

$$Fsmcompose\ dic.fsa > oen.fst$$

where, *dic.syms* stores all the symbols used in dictionary;

dic.stxt contains a textual representation of dictionary;

model.stxt contains a textual representation of B^{-1};

oen.stxt contains a textual representation of A_y(oen).

Again, the above methodology is applicable to any channel and applications other than spelling error correction. Moreover, for the spelling error correction problem, this methodology can find the correction of any type of spelling errors as permitted by the given channel.

In the next section, two examples of spelling error correction methods from Chapter 3 are described by using the general methodology.

4.4 Examples

In Chapter 3, we have seen two methods of probabilistic technique for spelling error correction problem:

— Church and Gale's method

— Mays and Damerau's method.

The Church and Gale's method is for isolated-word error correction; the Mays and Damerau's method is for context-dependent (real) word error correction. Both of these two methods use the probabilistic technique to solve problems, but they use different error models and different error correction algorithms. However, the general methodology is able to describe these two methods in the same way.

4.4.1 Church and Gale's method

Church and Gale [5] presented a probabilistic technique for the isolated-word error correction problem.

Recall that the error model of [5] uses the following probabilities:

$P(xy|x) = \frac{del[x,y]}{N(xy)}$ $\qquad$ $P(x|xy) = \frac{add[x,y]}{N(x)}$

$P(x|y) = \frac{sub[y,x]}{N(x)}$ $\qquad$ $P(xy|yx) = \frac{rev[x,y]}{N(xy)}$

This method also can be described by using our general methodology as below (Figure 4.3):

In Figure 4.3, y could be any letter, that is, $y \in \Sigma$. The probabilities $P2, P3, P4, P5$ are computed as above. Therefore, we have:

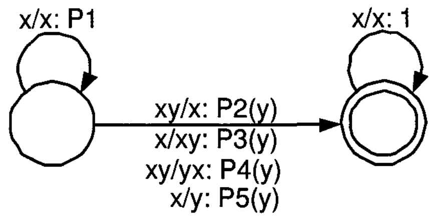


Figure 4.3: Church and Gale's Error Model

$$\sum_{y\in\Sigma} P2(y) = \sum_{y\in\Sigma} \frac{del[x,y]}{N(xy)} \qquad \sum_{y\in\Sigma} P3(y) = \sum_{y\in\Sigma} \frac{add[x,y]}{N(x)}$$
$$\sum_{y\in\Sigma} P4(y) = \sum_{y\in\Sigma} \frac{sub[y,x]}{N(x)} \qquad \sum_{y\in\Sigma} P5(y) = \sum_{y\in\Sigma} \frac{rev[x,y]}{N(xy)}$$

In [5], the probabilities for the non-error pair $(x|x)$ are not assigned. Thus, according to our theory of channels, $P(x|x)$ is assigned as:

$$P1 = 1 - \sum_{y\in\Sigma}(P2(y) + P3(y) + P4(y) + P5(y))$$

Once a channel of this error model has been computed, we can use the channel correction algorithm introduced in Section 4.3 to correct spelling errors. Again, as mentional in Section 4.3, if this channel is given, we can correct not only non-word spelling errors but also the real-word errors by using the channel correction algorithm.

4.4.2 Mays and Damerau's method

Mays and Damerau [21] presented a statistical technique capable of detecting and correcting real-word errors when they occurred in sentences. Recall that the method of [21] use the probabilities:

$$P(s_i|w_i) = \begin{cases} \alpha & \text{if } s_i = w_i, \\ \frac{1-\alpha}{|\mathrm{C}|-1} & \text{otherwise.} \end{cases}$$

For a word w, we denote by $C(w)$ the *confusion set* of w. This is the set of possible mispellings of the word w.

Our general methodology also can be used to describe Mays and Damerau's method. As we illustrated in our general methodology, the sum of all probabilities of transitions that start from same state with the same input letter is 1.

Suppose we are given several confusion sets of intended words as below:

Word	Misspellings
do	fo
door	doer, foor

The error model can be defined as below. (Figure 4.4)

In Figure 4.4, all the transitions from the start state are λ/λ, and the probabilities of them are $1/N$, Where N is the size of the dictionary. The probability of each transition after the first transition is assigned based on Mays and Damerau's confusion sets. Therefore, in Figure 4.4 we have that:

$P(do/do) = P1 \times P6 = \alpha/N$ $\qquad$ $P(fo/do) = P2 \times P6 = \frac{1-\alpha}{C(do)-1}/N$

$P(door/door) = P3 \times P7 = \alpha/N$ $\qquad$ $P(doer/door) = P4 \times P7 = \frac{1-\alpha}{C(door)-1}/N$

$P(foor/door) = P5 \times P7 = \frac{1-\alpha}{C(door)-1}/N$

Therefore the probabilities for each pair are divided by N. So that the probabilities of the possible outputs for a given input sum to 1. From a mathematical point of

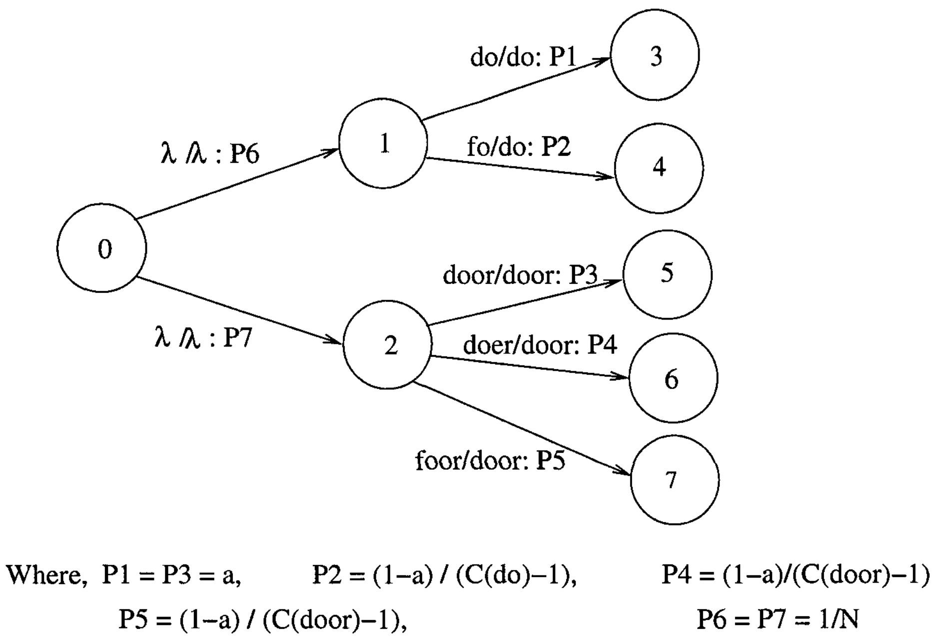


Figure 4.4: Example of Mays and Damerau's Error Model

view, the probabilities have not been changed. We also notice that in Figure 4.4, all probabilities of each transitions from same state with same input sum to 1, as required by our general methodology.

4.5 Improvements in the General Methodology

In Section 4.3, the correction algorithm of using a given channel has been introduced. We can find the correction of misspelling s by finding the string labeling a path of the weighted directed graph $A_s \circ B^{-1} \circ A_D$ with the lowest cost. However, in the general spelling error correction problem, more than one candidate words might be needed. Therefore, finding n strings with the lowest costs in the weighted directed graph becomes important. The algorithm for the n best-strings problem introduced by Mohri and Riley [24] can find the *n-best* distinct words in weighted directed graph easier and faster than the classic n best-string algorithms [4], [30].

N-best distinct words

The problem of determining the n shortest paths of a weighted directed graph is a well studied problem in computer science. The automaton searched may contain in general several paths labelled with the same sequence, thus the problem does not coincide with the classic *n-shortest-paths* problem. In fact, in many applications, the n best paths may be labelled with the same sequence many times.

Mohri and Riley [24] present an efficient algorithm for solving the *n-best-strings* problem in a weighted automaton. This algorithm is based on two general algorithms, the determinization of weighted automata and a general n-shortest-paths algorithm. The authors of [24] use weighted determinization to deal with the problem of several paths labelled with the same string and a single-source shortest paths algorithm to find the n strings with the lowest cost in the result of determinization automaton.

A weighted automaton is a directed weighted graph in which each edge or transition has a label with weight. In the case of spelling error correction in this research, the label is error operation. The weights are interpreted as negative log of probabilities.

The first step of this algorithm consists of computing the shortest distance from each state to the set of final states. After execution of this first step, the algorithm will find the n best paths in the result of a weighted determinization of the automaton.

Weighted determinzation takes as input a weighted automaton A and outputs an equivalent subsequential or deterministic automaton B. The weighted automaton B is deterministic if it has a unique initial state and if no two transitions leaving the same state share the same input label.

The algorithm presented in [24] is a generalization of the classical algorithm of Dijkstra [1]. They assume that the determinization automaton B contains only one final state. This does not affect the generality of this algorithm since one can always complete an automaton by introducing a single final state f to which all previously final states are connected by λ-*transitions.* The pseudo code of this algorithm is shown below, where Q' is the finite set of states, E is the finite set of transitions and F is the set of final states.

for $p \rightarrow 1$ to $|Q'|$ do $r[p] \rightarrow 0$

$z[(i', 0)] \rightarrow NULL$

$S \rightarrow (i', 0)$

While $S \neq \lambda$

    Do $(p, c) \rightarrow head(S)$; Dequeue(S)

      $r[p] \rightarrow r[p] + 1$

      If ($r[p] == n$ and $p \in F$) then exit

      If $r[p] \leq n$

        Then for each $e \in E[p]$

          Do $c' \rightarrow c + w[e]$

            $Z[(n[e], c')] \rightarrow (p, c)$

12 ENQUEUE $(S, (n[e], c'))$

They consider pairs (p, c) of a state $p \in Q'$ and a cost c. The algorithm uses a priority queue S containing the set of pairs (p, c) to examine next. The queue is in increasing order. This algorithm maintains for each state p an attribute r[p] that gives at any time during its execution the number of times a pair (p, c) with first state p has been extracted from S. $r[p]$ is initiated to 0 and incremented after each extraction from S. The priority queue S is initiated to the pair containing the initial state i' of B and the cost 0. Each time through the loop of lines 4-12 a pair (p, c) is extracted from S. For each outgoing transition e of p, a new pair $(n[e], c')$ made of the destination state of e and the cost obtained by taking the sum of c and the weight of e is created. The predecessor of this new pair is defined to be (p, c) and the new pair is inserted in S. The algorithm terminates when the n shortest paths have been found, that is when the final state of B has been extracted from S n times. Since at most n shortest paths may go through any state p, the search can be limited to at most n extractions of any state p. By construction, in each pair (p, c), c corresponds to the cost of a path from the initial state i' to p. Let us use the Figure 7.8 to illustrate this algorithm.

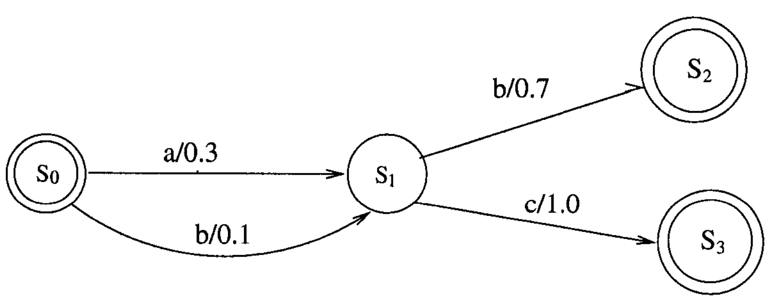


Figure 4.5: Example of Best Path

In Figure 4.5, we have:

(1) S = (S_0,0)

(2) S = (S_1,0.1) (S_1,0.3)

(3) S = (S_1,0.3) (S_2,0.8) (S_3,1.0)

(4) S = (S_2,0.8) (S_2,1.0) (S_3,1.1) (S_3,1.3)

So, if n=2, then the *2-best* paths are:

bb with cost = 0.8

ab with cost = 1.0

4.6 Computing a Channel from Sample Data

In this chapter, the general methodology has been described. It can be applied to many information processing applications, such as spelling error correction, speech recognition, etc. In this research, we are focusing on spelling error correction problem.

The process of computing a channel of the given error model can be defined as below:

— Given sample data and an error model

— Compute a channel of the error model that corresponds to the sample data

Therefore, the computation of a channel of an error model from sample data is important part of our research.

According to our general methodology, we will follow equation (4.1) (see Section 4.2) to compute the desired channel of the error model.

For each state z in the channel, we need a set of sample data consisting of a pair of string sequences (s_1, s_2) such that $s_1[i]$ is the ith string of s_1 and $s_2[j]$ is the jth string of s_2. We shall allow some edit operations on s_2, such as insert a string into s_2, delete a string from s_2, substitute a string in s_2 with another string, as well as other possible edit operations. Thus, a set of pairs $(s_1[i]/s_2[j])$ corresponding to the above edit operations can be generated. Next, we could align each pair $(s_1[i]/s_2[j])$ of this set to obtain a list of sequence pairs (x/y), where x and y are substrings of $s_1[i]$ and $s_2[j]$ with any length. Using statistical data about the sequence pairs and equation 4.1, we shall find the probability of each channel transition. If we have n states in the channel of the error model, then n sets of sample data are needed.

In Chapter 5, the algorithm of generating string pairs from sample data is introduced. This is the first step of computing channels. In Chapter 6, we will see the details about how to compute channels of the Brill and Moore error model.

Chapter 5

Generation Of String Pairs

In this chapter we will develop an algorithm for generating string pairs from sample data. This is also the problem of aligning two sequences of strings, which plays an important role in the area of computing channels.

Given two sequences of strings F_1 and F_2, we assume that F_1 contains m strings and F_2 contains n strings. Let u represent string $F_1(i)$, and v represent string $F_2(j)$, then a *string pair* can be defined as u/v. An *error pair* is a special case of string pair u/v, with $u \neq v$.

Papers [3], [5], [15], [21] introduced different error models for the spelling error correction problem. In order to compute channels of these error models, a set of training data consisting of string pairs is needed. However in these papers, the authors did not describe the method of obtaining the string pairs. In this chapter, we are going to introduce a dynamic programming algorithm to compute the training data set of string pairs. In the beginning of this chapter we will concentrate on the formal and technical aspects of the problem.

5.1 The Edit Distance Between Two Sequences of Strings

Frequently, one wants a measure of the difference between two strings (for example, in spelling correction methods, current molecular biology or textual database retrieval). Various approaches to the problem of string distance measurement have been defined (see [9], [14], [20] and references). A measure of the difference between two *sequences of strings* or between two *files* is also a common requirement for these applications. A file can be viewed as a sequence of strings.

Given a pair of two string sequences, we shall allow the following edit operations on the second sequence: the insertion of a string into the second sequence, deletion of a string from the second sequence, substitution (or replacement) of a string from the second sequence with a string in the first sequence, repetition of a string in the second sequence (which is a special case of insertion), and concatenation of two strings in the second sequence. For example, letting **I** denote the *insertion* operation, **D** denote the *deletion* operation, **S** denote the *substitution* operation, **R** denote the *repetition* operation, **C** denote the *concatenation* operation and **M** the nonoperation of *"match"*. Given the above, the sequence of strings "*error situations permitted in a communication system*" can be edited to "*error permittedin a a data communicaton system*" as follows:

M D C R I S M

error permittedin a a data communicaton system

error situations permitted in a communication system

We now can more formally define the terms sequence of edit transcripts and edit distance. A string over the alphabet I,D,S,C,R,M that describes a transformation of one sequence of strings to another sequence of strings is called an *edit transcript* of the two sequences of strings ([16]).

By examining the above example again, we find that there are several ways to transform the second sequence to the first one.

M D S D M I I S M

error permittedin a a data communicaton system

error situations permitted in a communication system

or

I D D C R I S M

error permittedin a a data communicaton system

error situations permitted in a communication system

However, there exists a best (possibly more than one) way to have edit transcripts between these two sequences. Hence, the *edit distance* between two sequences of strings is defined as the minimum number of edit operations needed to transform the first sequence into the second one where matches are not counted. Therefore, the edit distance problem is to compute the sequence edit distance between two given sequences of strings, along with an optimal edit transcript that describes the transformation.

5.2 Alignment Between Two Sequences of Strings

An edit transcript is one way to represent a particular transformation of one string sequence to another. An alternate way is to display an explicit alignment of the two sequences of strings. This idea is borrowed from the alignment of two strings. In [9], the author describes the concept of string alignment as: *"A global alignment of two strings S1 and S2 is obtained by first inserting chosen spaces, either into or at the ends of S1 and S2, and then placing the two resulting strings one above the other so that every character or space in either string is opposite a unique character or a unique space in the other string."*

An *alignment* of two sequences of strings F_1 and F_2 is a set of string pairs. It is obtained by first inserting the chosen dashes, either into or at the ends of F_1 and F_2, and then placing the two resulting sequences above each other so that every string or dash in either sequence is opposite to a string or a unique dash in the other sequence. As an example of an alignment, considering the alignment between the two sequences we discussed before:

error ——— permittedin a a data communicaton system

error situations permitted in a - —— communication system

In this alignment, the string *communicaton* is matched with *communication*; the strings *situations*, *a* and *data* are opposite dashes; *permittedin* is matched with *permitted in*, and all other strings match their counterparts in the opposite string.

From a mathematical view, an alignment and an edit transcript are equivalent ways to describe a relationship between two sequences of strings. An alignment can be easily converted to the equivalent edit transcript and vice versa. Specifically, two opposing strings that mismatch in an alignment correspond to substitution in the equivalent edit transcript; a dash in an alignment contained in the second sequence corresponds in the transcript to an insertion of the opposing string into the second sequence; a dash in the first sequence corresponds to a deletion of the opposing string from the second sequence; one string from the first sequence with opposing two strings in the second sequence that mismatch in an alignment correspond to the concatenation operation; and the above alignment example also shows that the repetition of a string from second sequence is a special case of the deletion operation.

5.3 Dynamic Programming Method

We now turn to the algorithmic question of how to compute the edit distance between two sequences of strings along with the accompanying alignment by using dynamic programming. We use the method of dynamic programming, which is based on the dynamic programming algorithm for computing the string distance ([9], [20]) – see Section 2.2.4. However our algorithm not only includes *deletion, insertion, substitution*, but also *repetition* and *concatenation*.

The cost of each operation is calculated as below. The concatenation operation between two strings is represented using underscore for seperating these two string. Let u and v be two strings containing at most one underscore.

$$cost(u,v) = \begin{cases} |u|, & \text{if } u \text{ contains no space and } v = \lambda, \text{ (deletion)} \\ |v|, & \text{if } v \text{ contains no space and } u = \lambda, \text{ (insertion)} \\ LD(u,v), & \text{if } u, v \neq \lambda \text{ and contain no space, (substitution)} \\ LD(u_1u_2, vv), & \text{if } u = u_1\_u_2 \text{ and } u_1, u_2, v \text{ contain no space, (repetition)} \\ LD(u, v_1\_v_2), & \text{if } v = v_1\_v_2 \text{ and } u, v_1, v_2 \text{ contain no space, (concatenation).} \end{cases}$$

The edit operation of repetition repeats a string from the source sequence, such as $a \rightarrow a\ a$; the edit operation of concatenation concatenates two strings from the source sequence, such as *permitted in* $\rightarrow$ *permittedin.* According to the definition of *repetition* and *concatenation*, one would expect that *repetition* $= (u, u\_u)$ and *concatenation* $= (u_1\_u_2, u_1u_2)$.

Suppose we are given a pair of sequences (files) F_1 and F_2, then D(i, j) is defined to be the edit distance between F_1[1,...,i] and F_2[1,...,j]. By using this notation, if F_1 has n strings and F_2 has m strings, then the edit distance between F_1 and F_2 is precisely the value of $D(n,m)$. We will compute $D(n,m)$ by solving the more general problem of computing $D(i,j)$ for all combinations of i and j, where i ranges from *0* to n and j ranges from *0* to m. Note that, $F_1[1\ldots 0]$ and $F_2[1\ldots 0]$ represent the empty sequence. The dynamic programming approach has three essential components: the *recurrence relation*, the *tabular computation*, and *the traceback.*

5.3.1 The recurrence relation

The recurrence relation of two files F_1 and F_2 establishes a recursive relationship for the value of $D(i, j)$, for i and j both positive, in terms of the values of D with index pairs smaller than i, j. When there are no smaller indices, the value of $D(i, j)$ must be stated explicitly in what are called the base conditions for $D(i, j)$. Also, we define the length of the strings $F_1[i]$ and $F_2[j]$ as $F_1[i].length$ and $F_2[j].length$, respectively.

The base conditions are: $D(0, 0) = 0$ and

for (i=1 to size of F_1)

D(i, 0) = D(i-1, 0) + $F_1[i].length$

and

for (j=1 to size of F_2)

D(0, j) = D(0, j-1) + $F_2[j].length$

The second base condition is clearly correct because the only way to transform the first i strings of F_1 to the empty sequence is to delete all the i strings of F_1. Similarly, the third base condition is correct because j strings must be inserted to convert the empty sequence to $F_2[1 \ldots j]$.

The recurrence relation for $D(i, j)$ when both i and j are strictly positive is (see

Figure 5.1).

$$
D(i,j) = min \begin{cases} D(i-1,j) + F_1[i].length & (deletion \;\; in \;\; F_2) \\ D(i,j-1) + F_2[j].length & (insertion \;\; in \;\; F_2) \\ D(i-1,j-1) + LD(F_1[i], F_2[j]) & (replace\; in\; F_2, if F_1[i] \neq F_2[j]) \\ \text{D(i-1,j-1)} & (match) \\ \text{D(i-2,j-1)+}LD(F_1[i-1]F_1[i], F_2[j]F_2[j]) & (repetition \;\; in \;\; F_2) \\ \text{D(i-1,j-2)+}LD(F_1[i], F_2[j-1]F_2[j]) & (concatenation \;\; in \;\; F_2) \end{cases} \tag{5.1}
$$

If $i < 2$ $\quad D(i-2, j-1) = \text{INFINITY}$ $(repetition)$

If $j < 2$ $\quad D(i-1, j-2) = \text{INFINITY}$ $(concatenation)$

5.3.2 Tabular computation

The second essential component of any dynamic program is to use the recurrence relation to efficiently compute the value $D(n, m)$. We will first compute $D(i, j)$ for the

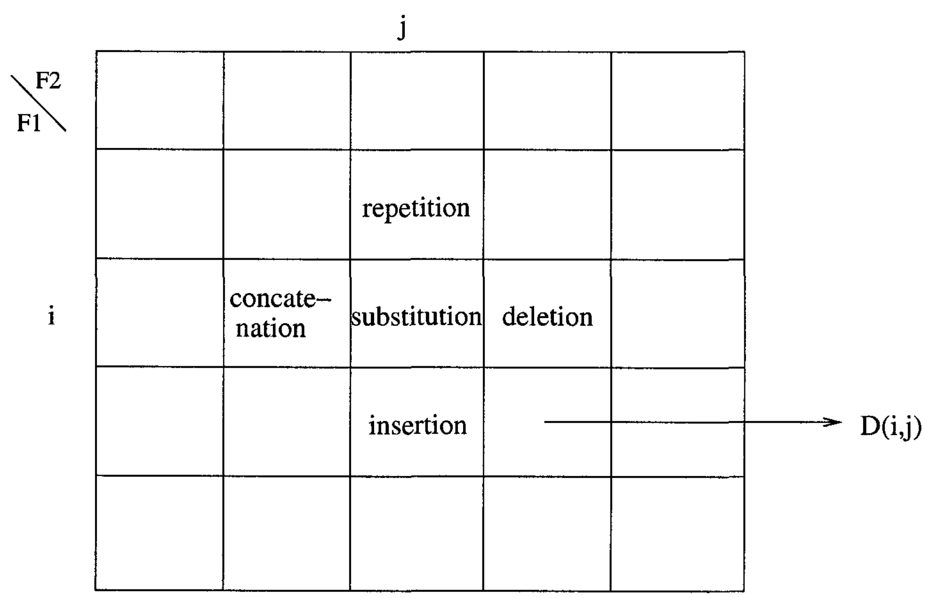


Figure 5.1: Table of Five Edit Operations

smallest possible values for i and j, and then compute values of $D(i,j)$ for increasing values of i and j. Typically, this method is organized with a dynamic programming table of size $(n+1) \times (m+1)$. The table holds the values of $D(i,j)$ for all the choices of i and j. The file F_1 corresponds to the vertical axis of the table and the file F_2 corresponds to the horizontal axis. Because the ranges of i and j begin at zero, the table has a zero row and a zero column. The values in row zero and column zero are filled in directly from the base conditions for $D(i,j)$. After that, the remaining $n \times m$ subtable is filled in one row at time, in order of increasing i. Within each row, the cells are filled in order of increasing j. From Table 5.1, we see that the values for row one can be computed in order of increasing index j. After that, all the values need to be computed in row two are known, and that row can be filled in, in order of increasing j. By extension, the entire table can be filled in one row at a time, in order of increasing i, and in each row the values can be computed in order of increasing j. The detailed tabular example of computing the edit distance between the two sequences as considered earlier is shown in the Table 5.1.

5.3.3 The traceback

Once the value of the edit distance has been computed, we can establish pointers in the table to find the associated optimal edit transcript (that is the alignment between F_1 and F_2). From here, we also can get the string pairs between F_1 and F_2. In each cell of this table, we store two values: $D(i,j)$.cost and $D(i,j)$.ope. $D(i,j)$.cost stores the value of edit distance from $D(0,0)$ to $D(i,j)$ and $D(i,j)$.ope stores the edit operation

D(i,j)	F_2	error	situations	permitted	in	a	communication	system
F_1	0	5	15	24	26	27	40	46
error	5	0	10	19	21	22	35	41
permittedin	16	11	9	12	10	11	24	30
a	17	12	10	13	12	10	22	28
a	18	13	11	14	13	10	22	28
data	22	17	15	18	17	14	21	27
communicaton	34	29	27	24	28	26	15	21
system	40	35	33	30	30	31	21	15

Table 5.1: Table of Dynamic Programming

that corresponds to the calculation in formula 5.1.

In particular, when the value of cell (i, j) is computed, we can consider a set of pointers as follows:

deletion: set a pointer from (i, j) to $(i-1, j)$

if $D(i,j)$.cost $= D(i-1,j)$.cost $+ F_2[j]$.length;

the pair is $D(i,j)$.ope $= F_2[j]/\lambda$;

insertion: set a pointer from cell (i, j) to cell $(i, j-1)$

if $D(i,j)$.cost $= D(i,j-1)$.cost $+ F_1[i]$.length;

the pair is $D(i,j)$.ope $= \lambda/F_1[i]$;

substitution: set a pointer from (i, j) to $(i-1, j-1)$

if $D(i,j) = D(i-1,j-1) + \mathrm{LD}(F_1[i], F_2[j])$;

the pair is $D(i, j).\text{ope} = F_2[j]/F_1[i]$;

match: set a pointer from (i, j) to $(i-1, j-1)$

if $D(i, j) = D(i-1, j-1)$; the operation is $D(i, j).\text{ope} = F_1[i]/F_2[j]$;

concatenation: set a pointer from (i, j) to $(i-1, j-2)$,

if $D(i, j).\text{cost} = D(i-1, j-2).\text{cost} + \text{LD}(F_2[j-1]F_2[j], F_1[i])$

$D(i, j).\text{ope} = (F_2[j-1]\_F_2[j])/F_1[i]$ ('_' refers to space)

repetition: set a pointer from (i, j) to $(i-2, j-1)$

if $D(i, j).\text{cost} = D(i-2, j-1).\text{cost} + \text{LD}(F_2[j]F_2[j]/F_1[i-1]F_1[i])$

$D(i, j).\text{ope} = (F_2[i-1]\_F_2[i])/F_1[j]$

These rules apply to cells in row zero and column zero as well. Hence, each cell in row zero points to the cell to its left, and each cell in column zero points to the cell just above it. The pointers allow one to recover an optimal edit transcript: simply follow the path of pointers from cell (n, m) to cell (0,0). Figure 5.2 shows a detailed example.

According to Figure 5.2, the alignment between these two sequences is as following:

error/error $\quad$ *λ/situations* $\quad$ *permittedin/(permitted in)* $\quad$ *(aa)/a*

data/λ $\quad$ *communicaton/communication* $\quad$ *system/system*

There are **1** deletion error, **1** insertion error, **1** substitution error, **1** concatenation error and **1** repetition error in this sample alignment.

5.4 Time Analysis

We now discuss the time complexity of this algorithm. When computing the value for a specific cell (i, j), only cells $(i-1, j-1), (i, j-1), (i-1, j), (i-2, j-1)$ and $(i-1, j-2)$ are examined, along with the two strings F_1(i) and F_2(j). Hence, to fill in one cell takes a constant number of cell examinations, arithmetic operations, and comparisons. The distance $D(n, m)$.cost can be computed in $\Theta(mn)$ time. The space used for this dynamic algorithm is also $\Theta(mn)$ strings. In practice, a file (sequence) could contain a large number of strings and this algorithm would not be very efficient.

To overcome this we introduce a heuristic method, called the *K-lookahead* method, where K is a positive integer. This method will only test the next K strings of each file every time, and store the first string pair for these K strings. If the pair is a

D(i,j)	F2	error	situations	permitted	in	a	communication	system
F1	0	←5	←15	←24	←26	←27	← 40	←46
error	↑5	↖0 error/error	←10 λ\situations	←19	← 21	← 22	←35	←41
permittedin	↑16	↑ 11	↖9	↖12	10 permittedin/ (permitted in)	←11	← 24	←30
a	↑17	↑ 12	↑10	↑13	↑12	↖10	↖←22	↖←28
a	↑ 18	↑ 13	↑11	↑14	↑13	10 (a a)/a	↖22	←28
data	↑ 22	↑ 17	↑15	↑18	↑17	↑14 data/λ	↖21	← 27
communicaton	↑ 34	↑ 29	↑27	↖24	↖28	↑ 26	15 communicaton/ communication	←21
system	↑ 40	↑ 35	↑33	↑30	↖30	←31	21	15 system/ system

Figure 5.2: Computation Table of Dynamic Programming

deletion of a string $F_2(j)$ from F_2, then the next run will start from $F_2[j+1]$ and $F_1[i]$; if the pair is an *insertion* of a string $F_1[i]$ into F_2, then the next run will start from $F_1[i+1]$ and $F_2[j]$; if the pair is a *match* of or a substitution of the string $F_1[i]$ with the string $F_2[j]$, then the next run will start from $F_1[i+1]$ and $F_2[j+1]$; if the pair is a *repetition* of string $F_1[i]$ with string $F_2[j]$, then the next run will start with $F_1[i+2]$ and $F_2[j+1]$; if the pair is a *concatenation* of string $F_1[i]$ with strings $F_2[j]$ and $F_2[j+1]$, then the next run will start with $F_1[i+1]$ and $F_2[j+2]$. This process will be repeated again and again until the whole file is finished. Figure 5.3, Figure 5.4, Figure 5.5 and Figure 5.6 show a detailed execution of this algorithm($K = 5$):

In these dynamic programming tables, K^2 steps are performed each time for a total of M = size(*longerFile*[1])-K+1 times . So this *K-lookahead* algorithm runs in $\Theta(MK^2)$ time. The space for this dynamic programming algorithm is $\Theta(K^2)$.

[1] *longerFile* is the file that is the longer one of the two input files

D(i,j)		error	situations	permitted	in	a
	0	5	15	24	26	27
error	5	0	10	19	21	22
permittedin	16	11	9	12	10	11
a	17	12	10	13	12	10
a	18	13	11	14	13	10
data	22	17	15	18	17	14

error / error

Figure 5.3: Example of *K-lookahead* Algorithm-Step1

D(i,j)		situations	permitted	in	a	communication
	0	10	19	21	22	35
permittedin	10	8	10	10	11	24
a	11	9	11	11	10	23
a	12	10	12	12	10	22
data	16	14	16	16	14	22
communicaton	28	26	28	27	26	15

λ / situations

Figure 5.4: Example of *K-lookahead* Algorithm-Step2

D(i,j)		permitted	in	a	communication	system
	0	9	11	12	25	31
permittedin	11	2	0	1	14	20
a	12	3	1	0	13	19
a	13	4	2	0	12	18
data	17	8	6	4	11	10
communicaton	29	20	18	16	5	11

permittedin / (permitted in)

Figure 5.5: Example of *K-lookahead* Algorithm-Step3

Sometimes a small K is not sufficient to find the correct string pairs between two files. In this case, a larger K needs to be tested each time until the sufficient K is identified. Binary search has been used for finding the proper K in our training set. First, we choose $K = size(longerFile)/2$. If it is sufficient to find the correct string pairs, then we chose $K = K/2$ to test the program again; otherwise, $K = K + K/2$ will be chosen as the next parameter for the program. The process is repeated until the optimal K has been found. The bigger K we have, the more time is needed to run and the more proper string pairs can be generated.

A number of other alignment alogrithms that save time and space by putting restrictions on the form of the alignment have been described in [9]. The general(original) string generation algorithm is used to derive results in this research.

D(i,j)		a	communication	system
	0	1	14	20
a	1	0	13	19
a	2	0	12	18
data	6	4	11	17
communicaton	18	16	5	11
system	24	20	11	5

(a a) / a, λ / data, communicaton / communication, system / system

Figure 5.6: Example of *K-lookahead* Algorithm-Step4

5.5 Results

We ran the experiments by using 12 pairs of files. Each pair consists of a file that may contain spelling errors and a correct file. Totally these 12 pairs of files have around 7500 string pairs. First, by running the original string generation algorithm in Section 5.3, a set of string pairs has been found. A sample typing file, correct file and their output results are shown below:

Rule based techniques are algorithms that attempt to represent knowledge of common splling error patterns in the form of rules for transforming misspellings into valid word. The candidate generation process consistsof applying all applicable rules ot a misspelled string retainng every valid dictionary word that result. It defines the estimation of probability of having made the particular error that te invoked rule corrected. Yhe candidates identified in the above process thus can be ranked by assigning nmberical scores to them based on the previous estimation.

Rule based techniques are algorithms that attempt to represent knowledge of common spelling error patterns in the form of rules for transforming misspellings into valid words. The candidate generation process consists of applying all applicable rules to a misspelled string and retaining every valid dictionary word that result. It defines the estimation of the probability of having made the particular error that the invoked rule corrected. The candidates identified in the above process thus can be ranked by assigning numerical scores to them based on the previous estimation.

Output: *splling/spelling, word/words, consistsof/(consists of), ot/to, */and, retainng/retaining, */the, te/the, yhe/the, nmberical/numerical*

There are 1042 error pairs in total of 7500 string pairs. Table 5.2 shows the percentage of different error types. *Substitution* error is the most common spelling error made by the specific user – the author of this thesis (84% of the total). It is

Total	Del.	Ins.	Sub.	Con.	Rep.
1042	16	40	876	100	10
100%	1.54%	3.84%	84%	10%	0.62%

Table 5.2: Statistics for Error Types

also the error type that is mostly related to the keyboard.

We have also tested an actual file in this study. A draft file was entered at the beginning. After going through the whole correction procedures, several paragraphs have been added up or cut down from the input file. And a reasonable set of string pairs as illustrated in the following have been identified:

by/, viewd/viewed, */We, */note, */that, */an, */interesting, */product, */construction, */between, */two, */copies, */of, */the, */same, */automaton, */is, */defined, */in, */cite, */for, */the, */purpose, */of, */deciding, */the, */property, */of, */unique, */decodability, */for, */regular, */languages*

From this result, we found that if we insert or delete an entire sentence or paragraph, the output will be a sequence of word insertions or deletions. This nice result shows that this algorithm is useful for generating error pairs.

In this study, we also test the appropriate K for different pair of files. We tested various values for K for each file pair from small to big, until the appropriate $K's$ are found. Table 5.3 shows the appropriate $K's$ we got from the training data sets.

In Table 5.3, the most values of K are satifying. The value of K is around 3 even when the total string of a file is more than 800. However, for files Sample6,7,8, the K

File Name	Total number of strings	K
Sample1	815	3
Sample2	798	3
Sample3	840	3
Sample4	315	2
Sample5	902	3
Sample6	443	280
Sample7	578	205
Sample8	562	243
Sample9	546	3
Sample10	485	3
Sample11	709	3
Sample12	500	3

Table 5.3: Appropriate K for Different Files

is relatively big, $K = 280, K = 205, K = 243$ respectively. Each of these three sample pair files has the situation mentioned before: entire sentences have been deleted from or inserted into the input file. Therefore, the value of K is much bigger.

The optimal value of K is important. Once we found K from the training file, we can use the K as the parameter for our *K-lookahead* algorithm, which will save time and space. From the above training data, K can be chosen as 5 if the file contains strings no more than 800 and there is no significant change between the two files.

Chapter 6

Improving the Brill and Moore Error Model

This chapter focuses on the application of our general methodology to the spelling error correction problem. A set of techniques have been introduced in Chapter 3 for the spelling error correction problem. Among them, the probabilistic method is more interesting to us due to it capability to correct spelling errors in text by using the Bayes Rule and the Noisy Channel Model [13], which has been successfully applied to a wide range of problems, including spelling error correction.

In 2000 Eric Brill and Robert C. Moore introduced a new channel model for spelling error correction [3]. In this chapter, we will have a close look at this model and implement it with several improvements based on our general methodology described in Chapter 4.

6.1 The Brill and Moore Error Model

Usually in an error model, people only consider a single edit operation *(insertion, deletion, substitution)* in the input string s ([5], [6], [15], [21]). Brill and Moore had improved on this by analyzing spelling errors in terms of more general string-to-string edit operations. Therefore, more than one edit operation can be considered in their model. For example, people are more likely to type ***tion*** as ***iton*** rather than ***t*** as ***i*** and ***i*** as ***t***.

Let Σ be an alphabet, s be the input string, w be the output string. The Brill and Moore error model allows all edit operations of the form α/β, where α, β are substrings with any length of s and w respectively, and $\alpha, \beta \in \Sigma^*$. $P(\alpha/\beta)$ is the probability that users intend to type the string α but they type β instead. Note that the edit operations allowed in [5], [6], [15], [21] are properly included by this generic string to string substitutions.

The main idea of this error model can be described as follows.

Generate a word from the input set $-->$ Pick a partition of the characters of that word $-->$ Type each partition, possibly with some mistakes.

Here is an example to illustrate this process. The word ***technical*** is chosen by a person. Then he/she picks a partition from the set of all possible partitions of that word, such as: ***te-ch-ni-cal***. After typing each partition, possibly with errors such as ***ta-k-ni-kal***, and choosing the particular word and partition, the probability of generating the string ***taknikal*** with the partition ***ta k ni kal*** would be $P(ta|te) \times P(k|ch) \times P(ni|ni) \times P(kal|cal)$. Obviously there are many other possible partitions of ***technical.***

In this example, we may notice that neither $P(k|ch)$ nor $P(kal|cal)$ is modeled directly using other error modeling methods ([6], [15], [21]).

A more formal description of this error model can be described:

— Given an alphabet Σ and a string s, where $s \notin D$ (dictionary) and $s \in \Sigma^*$, a partition T of s is a sequence of strings $T = (T_1, T_2, \ldots, T_m)$, such that $T_i \in \Sigma^*$ and $s = T_1 T_2 \ldots T_m$. Let $Part(s)$ be the set of all possible partitions of the string s.

— Given another string $w \in D$, a partition R of w is a sequence of strings $R = (R_1, R_2, \ldots, R_n)$, such that $R_i \in \Sigma^*$, and $w = R_1 R_2 \ldots R_n$. Let $Part(w)$ be the set of all possible partitions of the string w.

If,

The partitions $R = (R_1, R_2, \ldots, R_n)$ and $T = (T_1, T_2, \ldots, T_m)$ can be found such that $n = m$, $|R_i| \leq N$, for some fixed parameter N.

Then,

By only considering the best partitioning of s and w, we can define the error model:

$$P(s|w) \quad = \quad MAX_{R \in Part(w), T \in Part(s), |R|=|T|} \prod_{i=1}^{|R|} P(T_i|R_i), \tag{6.1}$$

where, $|T|$ and $|R|$ are the number of components in T and R, respectively.

The general methodology introduced in Chapter 4 for defining error models can be used to describe the Brill and Moore error model. In the Brill and Moore error model, every channel has only 1 state. Let S be the only state. If given a transition of the channel $S(x/y)S$, the input string x is T_i, output string y is R_i and $H[S, x](x/y, S) =$

$P(T_i|R_i)$. The channel can be viewed in Figure 6.1.

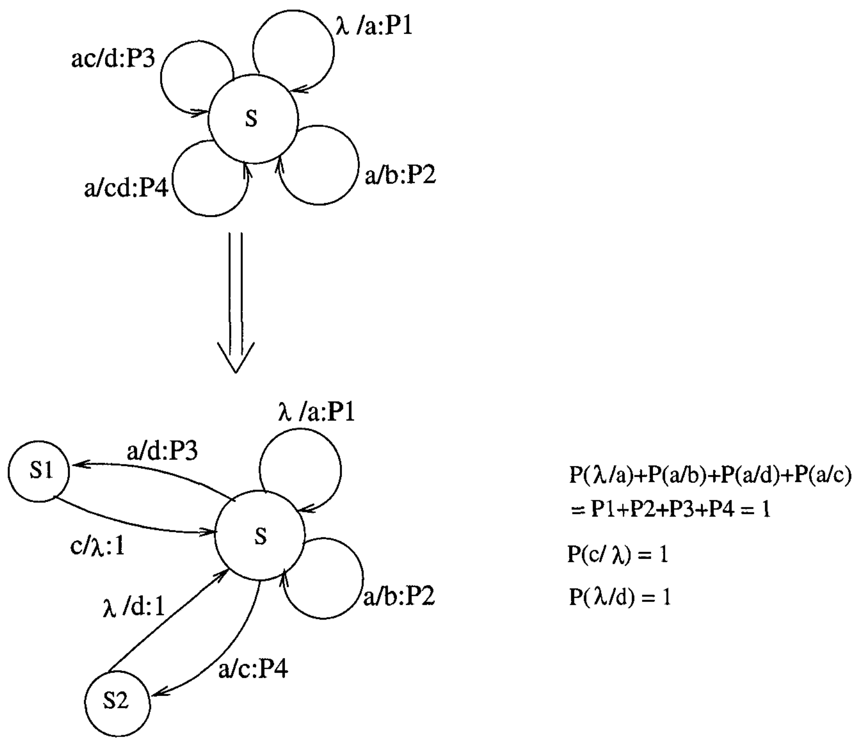


Figure 6.1: Brill and Moore Error Model

6.2 Training the Error Model

In Section 6.1, an useful error model has been described. Our next task is to describe the method of [3] to compute a channel of this error model corresponding to a given set of data. We call this the training problem. To conduct training in this error model, a training set consisting of error pairs (s_i/w_i) is needed. Recall that, in the previous chapter, a string pair generation algorithm was applied to identify a set of error pairs (s_i/w_i), where s_i is a word with possible spelling errors and w_i is a correct word.

Equation 6.1 suggests that we need to find a pair of partitions $R = (R_1, \ldots, R_n)$ and $T = (T_1, \ldots, T_m)$ for w_i and s_i. To achieve this, we begin by aligning the characters in s_i with those in w_i based on minimizing the edit distance between them. Then we expand each substitution edit operation to its left and right to allow general string-to-string edit operations and identify possible partitions of s_i and w_i. After that we can calculate the probability $P(\alpha/\beta)$ of each sequence pair (α/β), where α and β are substrings of s_i and w_i of variable length. The process of training the error model is described below:

- Get an optimal alignment between two strings;
- Expand each substitution edit operation to its left and right to allow string-to-string edit operations;

  For example, the optimal alignment between strings *baa* and *aaca* is b/a, λ/a, a/c and a/a. By expanding substitution edit operation b/a 1 position to it right, (b/aa) is obtained; by expanding substitution edit operation a/c 1 position to its left, (a/ac) is obtained; by expanding a/c to 1 position to its right, (aa/ca) is obtained.
- Generate the training set that contains all the sequence pairs (α/β).
- Compute the fractional count of each sequence pair (α/β);

  Using the same example above, the pairs (ba/aa), (ad/aa), (da/aa), ... need to be counted the total appearing time in the training set.
- Calculate the probability of each pair (α/β) as $P(\alpha/\beta) = count(\alpha/\beta)/count(\alpha)$.

By training the given data, a set that contains all the pairs (α/β) is generated. The quantity $count(\alpha/\beta)$ is simply the total number of (α/β) that appears in this training set. The quantity of $count(\alpha)$ is the number of times that substring α occurs in the input texts.

6.3 Improvements

In the previous section, we reviewed the general process of training the Brill and Moore error model. In order to enhance its applicability to a broader range of spelling errors, several improvements should be considered.

6.3.1 Alignment of string pairs

In the Brill and Moore's paper ([3]), the alignment between s_i and w_i was accomplished based on single character *insertions*, *deletions* and *substitutions*. We describe an example here to illustrate their method of alignment.

The error pair (ot/to) appears a number of times in our training set. But when we are trying to correct the misspelling *'ot'*, the expected correction *'to'* didn't appear in our result. Now, let us analyze the alignment of this error pair. According to the Levenshtein distance, there are different alignments between *'ot'* and *'to'*, but they all have the same minimal edit distance 2:

(1) λ o t

t o λ (one insertion λ/t and one deletion t/λ)

(2) o t λ

λ t o (one deletion o/λ and one insertion λ/o)

(3) o t

t o (two substations o/t and t/o)

Obviously, the third alignment (o/t) (t/o) is the best alignment based on most people's typing habit. However, the first alignment (λ/t) (o/o) (t/λ) is chosen by the program because the authors of [3] didn't consider *transposition* edit operation. In this example, we couldn't get any string-to-string edit operations because there is no substitution error in this alignment. By only using the pairs above (λ/t) (o/o) (t/λ), we are not able to find the correction for *ot*.

The transposition error (ot/to) is more natural than any of the above three alignments. Thus, the fourth edit operation *transposition* has been added to our alignment algorithm (Section 2.2.3) as showing below:

$$C(i,j) = min \begin{cases} C(i-1,j)+1 & \text{deletion,} \\ C(i,j-1)+1 & \text{insertion,} \\ C(i-1,j-1)+G(i,j) & \text{substitution,} \\ C(i-2,j-2)+T(i,j) & \text{transposition.} \end{cases}$$

where $G(i,j) = 0$ if $A(i) = B(j)$,

$G(i,j) = 1$ if $A(i) \neq B(j)$;

$T(i,j) = 1$ if $A(i-1)A(i) = B(j)B(j-1)$,

$T(i,j) = 2$ if $A(i-1)A(i) \neq B(j)B(j-1)$;

After adding *transposition*, the alignment (o/t) (t/o) can be chosen by the program. Therefore by expanding the substitution edit operation, the following sequence pair can be generated: ot/to.

With the above changes, the correction *'to'* for the misspelling *'ot'* can be identified in our result list.

6.3.2 Expand substitution edit operation

In the same paper, Brill and Moore introduced a method that expands each substitution error to incorporate up to N additional adjacent edits, and allow for richer contextual information, where N is the fixed parameter of the model. At each substitution position, the letters are expanded to the left and right.

For the sake of illustration let us consider an example. Suppose we have two strings $s = baada$ and $w = aaaaa$. With Brill and Moore's method the training pair $(baada, aaaaa)$ can be aligned as: $b/a, a/a, a/a, d/a, a/a$. This means when $N = 0$, we have $R = (b, a, a, d, a)$ and $T = (a, a, a, a, a)$. The substitution errors are b/a and d/a. To allow for richer contextual information, we can expand each substitution:

For N=1, the following sequence pairs are generated: $ba/aa, ad/aa, da/aa$;

For N=2, the following sequence pairs are generated: $baa/aaa, aad/aaa, ada/aaa$.

However, if we turn to another example, there is a problem. Given another training pair $(baa, aaca)$, by using the same method shown above, we would generate the following substitutions:

N=0: $b/a, a/c$

N=1: $b/aa, a/ac, aa/ca$

N=2: $ba/aac, ba/aac, aa/aca$

In this example, when N=2, the pair (ba/aac) occurs twice. But when we look back to our original training pair $(baa, aaca)$, the pair (ba/aac) only occurs once. Therefore, we modified this method to allow more suitable sequence pairs. For each no match and matched position in the allignment, we expand the character only to the right. For instance, in the example above, we could regenerate the following sequence pairs:

N=0: $b/a, \lambda/a, a/c, a/a$

N=1: $b/aa, a/ac, aa/ca$

N=2: $ba/aac, aa/aca$

In our new result, ba/aac occurs only once. Therefore, from this example, we could find that the new method gives us more proper sequence pairs.

6.3.3 Assign the probabilities

Fractional count of sequence pairs

In [3], the authors described the method for calculating the probability of each sequence pair (α/β) as $P(\alpha|\beta) = count(\alpha/\beta)/count(\alpha)$. For each (α/β) in the set of sequence pairs we define:

— $count(\alpha/\beta)$ is the number of times that (α/β) occurs in the set of sequence pairs;

— $count(\alpha)$ is the number of times that substring α occurs in the text corpus. In other words, $count(\alpha)$ is the sum of the number of times that $count(\alpha/\beta)$ occurs in the set of sequence pairs, but this time, β could be any letter occurring in that set.

Hence, the probability $P(\alpha/\beta) = count(\alpha/\beta)/count(\alpha)$. We also define the *Distance Cost* between α and β as $C(\alpha/\beta) = -\log P(\alpha/\beta)$.

The Brill and Moore error model only considers α as a non empty letter, but neglects the fact that empty strings are also frequently encountered in insertion errors (λ/β). Therefore, we can redefine the probability calculation method for each pair as below:

If α is the empty string (λ), $count(\alpha)$ is the number of times that λ occurs in the input files. It equals to the sum of each word length plus 1. For example, if the input files contains two words w_1 and w_2 with $|w_1| = 5$ and $|w_2| = 6$, then $count(\alpha) = 6 + 7 = 13$.

Convert the transition

In the Brill and Moore error model, transitions such as $S_0(\alpha/\beta)S_1$ have $|\alpha| = n$ and $|\beta| = m$, where n or m is greater than 1. By using the rule 4.2, we can convert $S_0(\alpha/\beta)S_1$ to a sequence of transitions with single labels. To further demonstrate this, let us look at the following examples:

Given a transition $S_0(xy/ab)S_1$ with $P(xy/ab) = P$, it can be converted to $S_0(x/a)S_1(y/b)S_2$ with $P(x/a) = P$ and $P(y/b) = 1$;

Given a transition $S_0(xy/a)S_1$ with $P(xy/a) = P$, it can be converted to $S_0(x/a)S_1(y/\lambda)S_2$ with $P(x/a) = P$ and $P(y/\lambda) = 1$;

Given a transition $S_0(x/ab)S_1$ with $P(x/ab) = P$, it can be converted to

$S_0(x/a)S_1(\lambda/b)S_2$ with $P(x/a) = P$ and $P(\lambda/b) = 1$.

The Figure 6.1 illustrates this method.

As illustrated in Figure 6.1, $S(a/cd)S$ with $P(a/cd) = P4$ is converted to $S(a/c)S_2(\lambda/d)S$ with $P(a/c) = P4$ and $P(\lambda/d) = 1$; $S(ac/d)S$ with $P(ac/d) = P3$ is extended to $S(a/d)S_1(c/\lambda)S$ with $P(a/d) = P3$ and $P(c/\lambda) = 1$.

Assign the probabilities

In this research, we are going to determine a proper way to calculate the probability $P(\alpha/\beta)$ of pair (α/β).

As we discussed before, if the transition $S_0(\alpha/\beta)S_1$ with $|\alpha|$ or $|\beta|$ is greater than 1, we will convert it to a sequence of transitions with single labels. According to the method of assigning probabilities in Section 4.2, thus, if $\alpha = x_1z, x_1 \in \Sigma, z \in \Sigma^*$, we have

$$\sum_{\beta\in\Sigma^*}\sum_{z\in\Sigma^*} P(x_1z/\beta) = P_1,$$
$$P(\alpha/\beta) = \frac{count(\alpha/\beta)}{count(x_1z)} \times P_1,$$

And,

$$\sum_{\beta\in\Sigma^*} P(\lambda/\beta) = P_2,$$
$$P(\lambda/\beta) = \frac{count(\lambda/\beta)}{count(\lambda)} \times P_2.$$

where,

— $count(x_1z)$ is the number that substring in which start letter is x_1 occurs in the text. In other words, $count(x_1z)$ is the sum of the number times that $count(x_1z/\beta)$ occurs in the set of sequence pairs, where $z \in \Sigma^*, \beta \in \Sigma^*$.

— $0 \le P_1 \le 1, 0 \le P_2 \le 1, P_1 + P_2 = 1$

6.4 Applying the Model

In Section 6.3, we described how to train the error model and how to obtain the set of parameters $P(\alpha/\beta)$, which define the channel. Each $P(\alpha/\beta)$ is the probability that if a substring α is intended, the channel will produce β instead ($\alpha, \beta \in \Sigma^*$). In this section, we will describe Brill and Moore's algorithm to correct spelling errors by applying their error model. In particular, the working process of spelling error correction problem can be described in 3 steps: (1) detecting an error; (2) generating n candidate corrections; (3) ranking the list of candidate corrections. For example, if $n = 3$, then the *3-best* list will contain 3 words w_1, w_2, w_3, such that $w_1 \in D$, $w_2 \in D$, $w_3 \in D$, and they have the minimal distance to s in the order of $C(w_1/s) \le C(w_2/s) \le C(w_3/s)$, where $C(w_1/s) = -logP(w_1/s)$.

Apply the model

In [3], the authors introduced a dynamic programming that correct errors by applying the error model.

In the standard dynamic programming of computing the Levenshtein distance, in order to fill the cell (i, j) in the matrix, we need to only test cells $(i, j-1)$ (*insertion* error), $(i-1, j)$ (*deletion* error) and $(i-1, j-1)$ (*substitution* error). In this research, however, we allow generic edit operations (error pairs) of the form α/β, where each α/β has a cost $C(\alpha/\beta) = -\log P(\alpha/\beta)$. This means that in order to fill in the cell

(i, j) in the edit distance matrix, all cells (a, b) where $a \leq i$ and $b \leq j$ might have to be examined.

Following [3], we precompiled the dictionary into a trie, and store a vector of weights in each node of the trie. Then we consider the standard matrix of computing edit distance between two strings (one is the misspelling s, and the other one is the correct word w in the dictionary). Thus the vector of weights for each node in the trie corresponds to a column in the weight matrix associated with computing the distance between s and the prefix of w ending at that trie node. Therefore the last number stored in the vector of the final nodes in the trie will represent the edit distance between the input string s and the string w in the dictionary reached at that node. Figure 6.2 shows an example that illustrates this dictionary trie, where $s = ann$ and $w = ant$.

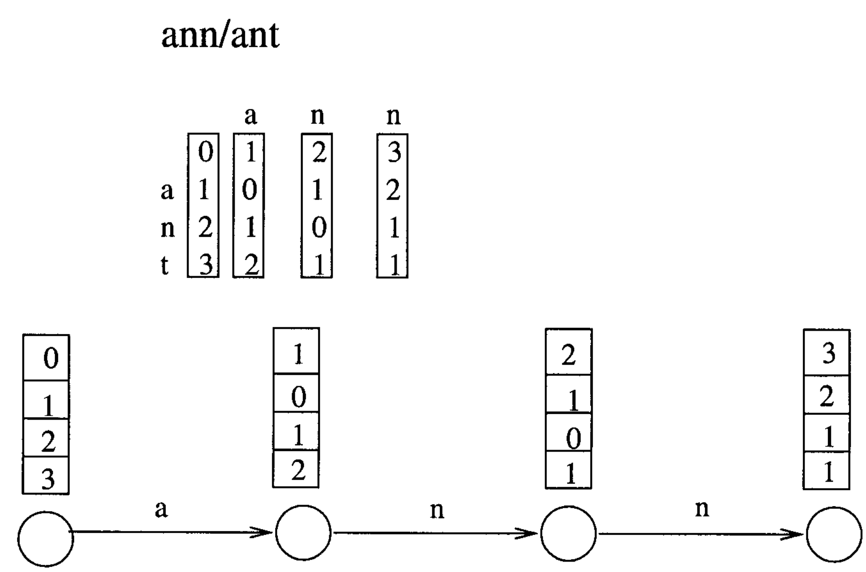


Figure 6.2: Example of Computing Distance In the Trie

We store all the $P(\alpha/\beta)$ parameters in a ternary search trie, each node of which contains a ternary search trie. Figure 6.3 shows the ternary search trie that stores a list of $P(\alpha/\beta) : ac/akg, ad/bc, ab/agh.$

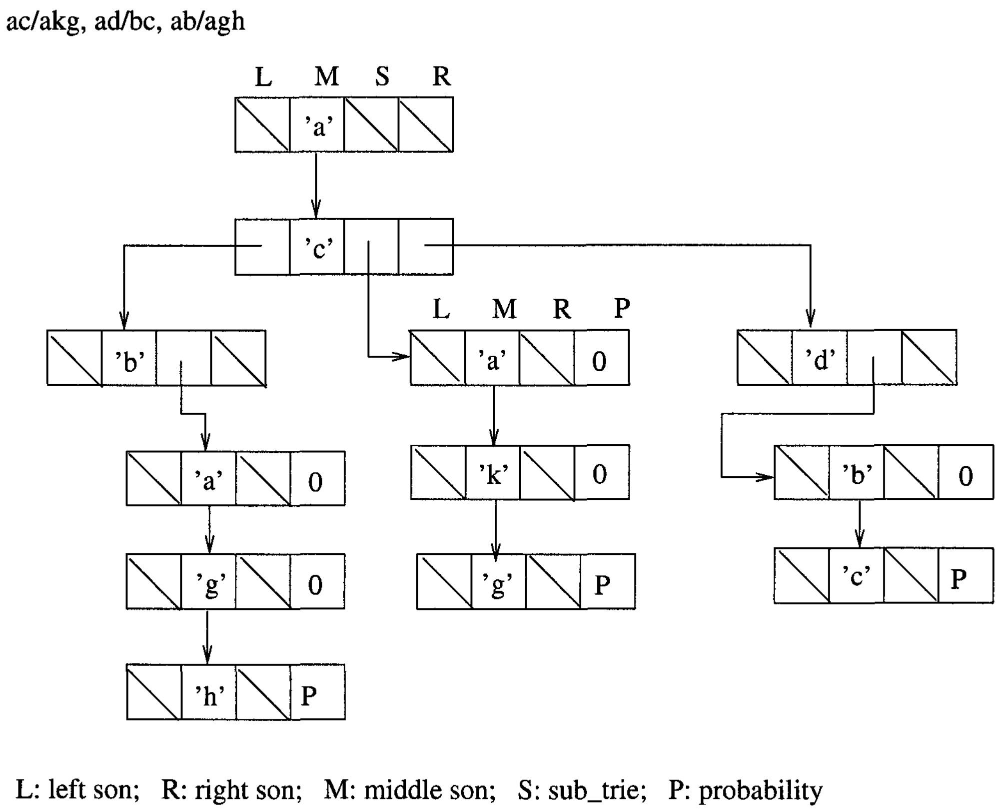


Figure 6.3: Ternary Search Trie to Store Parameters

In particular, we have one ternary trie corresponding to all string pairs that appear on the left hand side (string α) in our parameter set. In this trie, if we reach the end of the string α, then we have a pointer *sub_trie* pointing to a ternary trie that consists of all strings β appearing on the right hand side of the set of $P(\alpha/\beta)$ parameters with α on the left hand side. We will store the substitution probabilities at the terminal nodes of the β ternary trie. Both α and β string will be stored in the reverse order.

We then need to compute edit distance over the entire dictionary one by one.

Chapter 7

Experimentation

In this chapter, we will conduct experimental tests on both the original and the improved Brill and Moore (*improvedBM* for short) error model. We also will compare the experimental results of applying the dynamic programming and the channel error correction algorithm on the *improvedBM* error model. Moreover, four more error models modified from the *improvedBM* error model will be described, tested and the experimental results will be reported.

In order to get reasonable experimental results, a total of 12 pairs of files (containing around 7500 string pairs) have been used for training various error models. A total of 1042 error pairs are generated from the training set to compute the channels of these error models. Our dictionary contains approximately 250,000 correct words, including all words in the training set. For evaluation, we have run experiments using two *testing sets* of *misspellings* (words with common English spelling errors). Both of these sets are generated from the typing mistakes created by the author of this thesis. The first testing set has 134 misspellings and the second one has 91 misspellings. The first testing set contains a number of misspellings that have been used also for training the channel. But no misspellings in the second testing set have been used. In the experimental results tables, *n-best* lists contain *n-candidate* correct words for each misspelling where $n = 1, 2, \ldots$.

In this research, we are interested in computing the channel of given error model that corresponds to a specific typesetter. Therefore, all the training files and testing misspellings are from a specific user (the author of this thesis).

7.1 Comparison of the Original and Improved Brill and Moore Error Models

Testing the original Brill and Moore error models

The Original Brill and Moore error model (without any improvements) has been tested in this section. Totally 2526 sequence pairs (α/β) $(\alpha \neq \beta)$ are generated from 1042 error pairs (s_i/w_i) to compute a channel of this error model. The results on the two testing sets of misspellings are shown in Table 7.1. However, since we don't have large quantity of training data set, the results on *improvedBM* error model don't have the same accuracy level as that has been illustrated in Brill and Moore's paper [3].

Testing the improved Brill and Moore error model

Totally 6692 sequence pairs (α/β), including the case of $\alpha = \beta$, are generated from

	Total	1-best	2-best	3-best
First Set	132	102	117	122
(%)		77.28	88.64	92.42
Second Set	91	61	73	80
(%)		67.03	80.22	87.91

Table 7.1: Result of Original Brill and Moore Error Model

the 1042 error pairs used above to compute a channel of the *improvedBM* error model. The results on the two testing sets of misspellings are shown in Table 7.2.

	Total	1-best	2-best	3-best
First Set	132	114	122	126
(%)		86.36	92.42	95.45
Second Set	91	66	76	82
(%)		72.52	83.51	90.1

Table 7.2: Result of Improved Brill and Moore Error Model

Comparison

As illustrated in Figure 7.1, for both two testing sets the *improvedBM* error model has a better result than the original Brill and Moore error model. In particular, if the misspelling contains more than one errors, the improved error model can find more appropriate corrections. For example, the misspelling *'peoid'* can be corrected to *'period'* in the first candidate word by using the improved model, but can not be corrected in any candidate words by using the original model. Three candidate words prodeced by using the original model are: *Lepid*, *tepid* and *paid*.

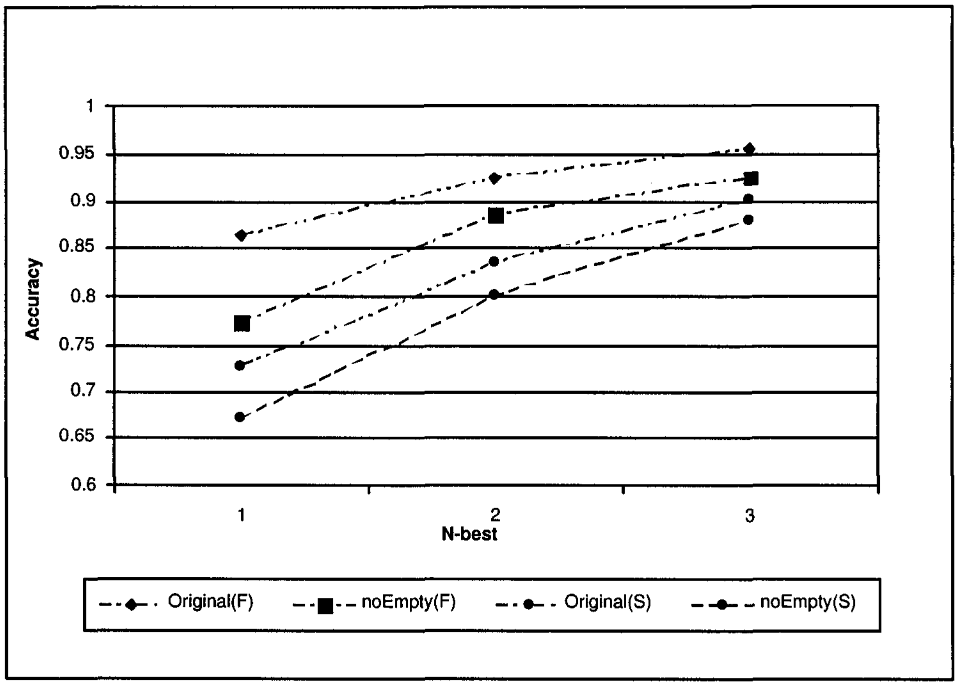


Figure 7.1: Comparison of Original and Improved Brill and Moore Error Model

7.2 Comparison of Dynamic Programming and Channel Correction Algorithms

In Chapter 4, a general error correction algorithm is defined to correct errors for a given channel which is called the channel correction algorithm in this research.

In the channel error correction problem, we are given a channel output s, an NFA A_D that stores the dictionary D, and a wfse-symtem B^{-1} that corresponds to the channel. If s is described as a *DFA* A_s, then the channel correction algorithm addressed in Section 4.3 can be used to find the correction of s.

In this section, we are going to use this algorithm to find out the correction of misspellings if the wfse-system B^{-1} corresponds to the channel of the *improvedBM* error model. Theoretically, the results of finding corrections by using dynamic programming described in Section 6.5 are the same as the results of using the channel correction algorithm. The example in Section 7.2.1 shows that also in practice the results are the same.

The *AT&T* finite state machine tools described in Chapter 2 were used to create the *WFA* A_s, A_D and the *WFST* B^{-1}. This tool also can be used to compute the composition $A_s \circ B^{-1} \circ A_D$.

7.2.1 Example

The example below shows how the corrections of a certain misspelling can be identified by using dynamic programming.

misspelling: *aba*; dictionary word: *abab*

We consider the following 1-state channel of the *improvedBM* error model:

$$P(b/b) : 1.0, P(a/a) : 0.15, P(ab/ab) : 0.5, P(ab/a) : 0.35;$$

As we know, the Distance Cost C is the negative log of the probability P, that is,

$$C(b/b) : 0, C(a/a) : 0.82, C(ab/ab) : 0.3, C(ab/a) : 0.45;$$

Dynamic Programming:

We can create a two-dimensional table as displayed in Figure 7.2

There are two paths shown in this table to reach the final destination:

(1) pairs: $(ab/ab)(ab/a)$

cost: $0.30 + 0.45 = 0.75$ (total)

(2) pairs: $(a/a)(b/b)(ab/a)$

cost: $0.82 + 0 + 0.45 = 1.27$ (total)

In these two paths, the edit operations $(ab/ab)(ab/a)$ has the smallest cost 0.75 to reach the correction.

The next example shows how to determine the corrections of the same misspelling by using the channel correction algorithm.

Create a *weighted finite-state transducer (WFST)* for the dictionary $D = \{abab\}$: Figure 7.3

Create a *WFST* for the misspelling $S = aba$: Figure 7.4

Create a *WFST* for the same channel B^{-1}: Figure 7.5

Create a *WFST* for $X = D \circ B^{-1}$ ($\circ$ is the composition operator): Figure 7.6

Create a *WFST* for $Y = X \circ S$: Figure 7.7

Figure 7.7 illustrates that two paths are able to reach the correction:

$(a/a)(b/b)(ab/a)$ with the total cost of 1.27;

$(ab/ab)(ab/a)$ with the total cost of 0.75.

Then the best path has been found in Y: (Figure 7.8)

Therefore, the pairs $(ab/ab)(ab/a)$ have the smallest cost 0.75 to reach the correction, which is the same result as in the case of dynamic programming.

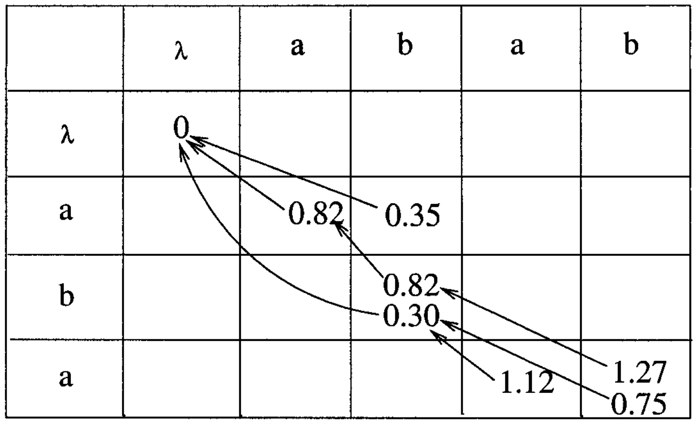


Figure 7.2: Dynamic Programming to Calculate String Correction

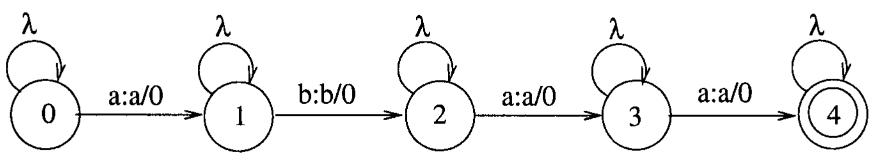


Figure 7.3: $WFST$ D of Dictionary *abab*

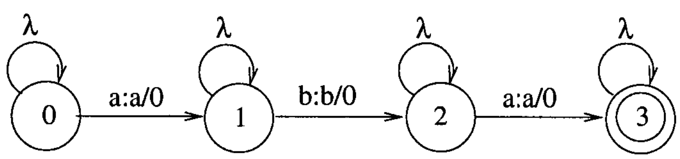


Figure 7.4: $WFST$ S of Misspelling *aba*

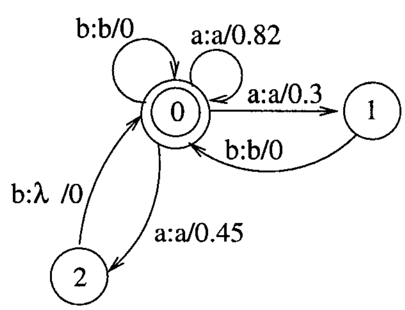


Figure 7.5: Channel $P(b/b) : 0.2, P(a/a) : 0.2, P(ab/ab) : 0.1, P(ab/a) : 0.15$

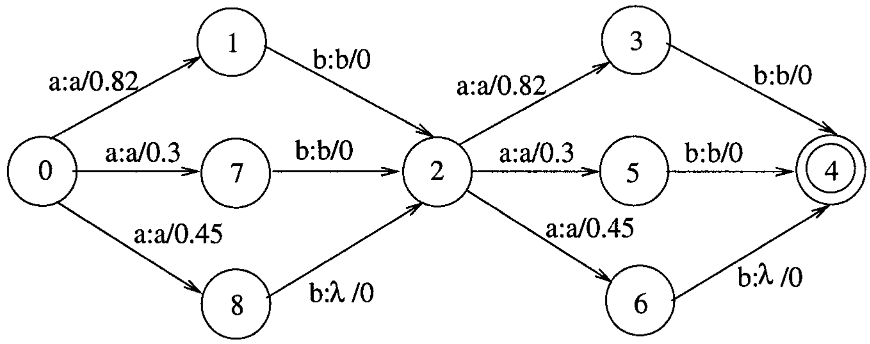


Figure 7.6: $WFST\ X = D \circ B^{-1}$

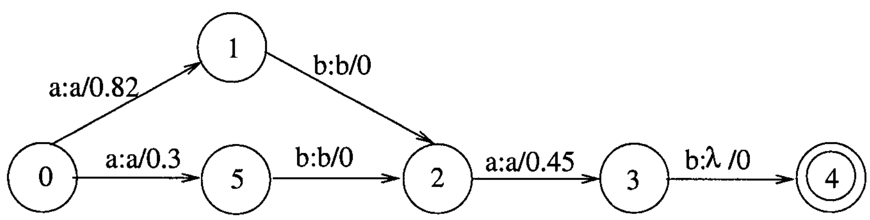


Figure 7.7: $WFST$ Y = X∘ S

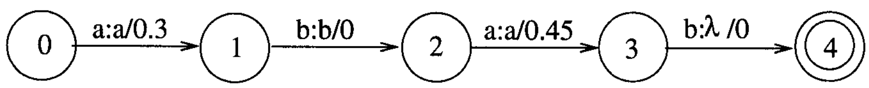


Figure 7.8: The Best Path from X∘ B

7.2.2 Result

In this section, we compare the results by applying the dynamic programming and the channel correction algorithms on the *improvedBM* method. The results are displayed in Table 7.3 and 7.4.

From Table 7.3 and Table 7.4, we can see that the correction of misspellings by using the channel correction algorithm is same as the correction by using the dynamic programming. The corrections are also with essentially the same cost, considering the possibility of computer arithmetic imprecisions.

7.3 Other Error Models

In this research, four more error models will be considered by modifying the *improvedBM* error model. We use the same data to train these error models and use the same sets of misspellings to test these models - see Section 7.1. We will illustrate each of them in the following sections.

7.3.1 Total one model

Ristad and Yianilos presented a stochastic transducer to determine the similarity of two strings [28]. This stochastic transducer allows us to learn a string-edit distance function from a corpus of examples. They modeled string-edit distance as a memoryless stochastic transducer. In this model, each channel has one state and each transition generates either a substitution pair (a/b), a deletion pair (a/λ), an

Misspelling	Correction (Our algorithm)	Cost	Correction (Dynamic Programming)	Cost
one(1-best)	wen	0.734	wen	0.735
(2-best)	fen	0.774	fen	0.776
(3-best)	ten	0.81	ten	0.811
benn(1-best)	bean	0.904	bean	0.906
(2-best)	benny	0.915	benny	0.918
(3-best)	bend	0.94	bend	0.941
ot(1-best)	to	0.655	to	0.655
(2-best)	fo	0.714	ft	0.715
(3-best)	sot	0.734	sot	0.735
peoid(1-best)	peroid	1.133	period	1.134
(2-best)	prid	1.713	prid	1.713
(3-best)	pierid	1.727	pierid	1.729
specity(1-best)	specify	1.104	specify	1.106
(2-best)	specialty	1.774	specialty	1.776
(3-best)	asperity	1.988	asperity	1.99
bu(1-best)	bus	0.741	bus	0.741
(2-best)	but	0.757	but	0.757
(3-best)	bun	0.775	bun	0.775
skils	skis	1.203	skis	1.205

Table 7.3: Comparison Results for the *improvedBM* Error Model

Misspelling	Correction (Our algorithm)	Cost	Correction (Brill and Moore's Method)	Cost
anatmy	anatomy	1.084	anatomy	1.086
contro	control	1.118	control	1.121
goint	going	0.996	going	0.998
detectin	detection	1.271	detection	1.274
speling	spelling	1.156	spelling	1.157
metods	metis	1.868	metis	1.87
sincd	since	1.001	since	1.003
owrd	word	0.668	word	0.668
machanincs	mechanics	1.952	mechanics	1.952
folliwgn	following	1.668	following	1.67
decidng	deciding	1.171	deciding	1.171
correciton	correction	1.211	correction	1.212
bcause	because	1.094	because	1.096
seciton	section	0.909	section	0.909
etecting	detecting	1.326	detecting	1.327
precdeing	preceding	1.35	preceding	1.351
prbability	probability	1.39	probability	1.392
isolatd	isolated	1.224	isolated	1.226

Table 7.4: Comparison Results for the *improvedBM* Error Model

insertion pair (λ/b), or the distinguished termination symbol #. Of course, the null operation (λ/λ) is not included in the alphabet E of edit operations. The sum of the probabilities of all edit transitions is 1.

Ristad and Yianilos use this stochastic transducer to generate strings from a corpus of examples, so they need the termination symbol # to delimit resulted words. However, in our research, a set of input strings is given. We will use error model to find the correction of these input strings. Therefore, in this section, the similar stochastic automaton will be used to define our spelling error correction model with only one difference: the termination symbol # is not considered in the error model.

The error model is illustrated in Figure 7.9, such that the probabilities of all the transitions sum to 1.

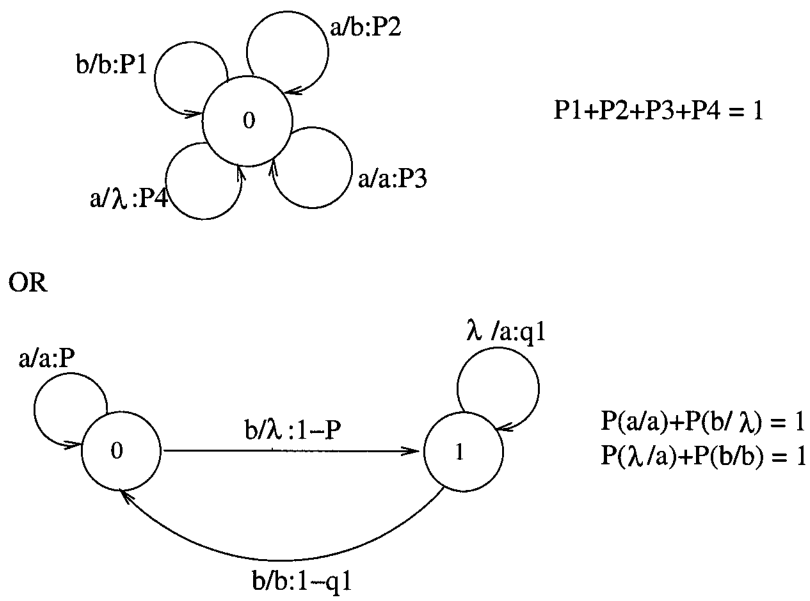


Figure 7.9: The *totalOne* Model

The result of using our training data in this error model is shown in Table 7.5:

	Total	1-best	2-best	3-best
First Set	132	108	115	121
(%)		81.8	87.12	91.67
Second Set	91	64	72	75
(%)		70.32	79.12	82.42

Table 7.5: Result of *totalOne* Model

The result of *totalOne* error model in Table 7.5 shows that it doesn't have the same accuracy as that of *improvedBM* model. In the first testing set, the *1-best* accuracy of *totalOne* model is 81.81%, the *2-best* accuracy is 87.12% and *3-best* accuracy can reach up to 91.67%. In the first testing set, the accuracy of all *n-best* list for this model is lower than the *improvedBM* model. However, a number of misspellings such as *oen* that can be corrected in the *1-best* set of this model cannot be corrected in the *1-best* set of *improvedBM* model. If the correction of The misspelling can not be found in the *1-best* and *2-best* sets in the *totalOne* model, then the chance of the correction appearing in the *3-best* set is limited. In the second testing set, the accuracy of all *n-best* list is lower than that from *improvedBM* model.

7.3.2 Different insertion model

The typing behaviour of people can be very complicated. It depends on the individual's typing skill and knowledge, as well as the layout of keyboard being used. In Chapter 2, we mentioned that Damerau [6] found that approximately 80% of all misspelled words contain a single instance of one of the following four error types: *insertion*, *deletion*, *substitution*, and *transposition*.

In the *improvedBM* error model, when we deal with insertion errors, we consider them independent of the next letter. For example, in the error pair (*thre*/*there*), there is one insertion error λ/e. However, this insertion error may depend on the next letter r. In the most cases, when people are typing, they always think of the next input letter. Therefore the insertion errors may relate to that letter. We modified the *improvedBM* error model to be more specific to insertion errors. In this new error model, we use $\lambda(x)/y$ instead of edit operations λ/y, such that the edit operation $(\lambda(x)/y)$ is applied only if the next input letter is x. Thus, the probability measure $H[S_i, x]$ is now defined in the same manner as before but with the following change:

$H[S_i, x](\lambda(x')/y, S_j) = 0$ for $x' \neq x$ (the probability is 0 if the next input is x with $x \neq x'$)

$H[S_i, x](\lambda(x)/y, S_j) \geq 0$

Figure 7.10 shows the error model we described in this part:

The result of using this error model is shown below:

From the result of this different insertion (*difInsert* for short) error model, we can see that the accuracy is a little better than the *totalOne* model. In the first testing set, the *1-best* set accuracy is 83.33%, the *2-best* set accuracy is 91.42% and the *3-best* set accuracy can reach 93.94%. The result of this error model is similar to but a little worse than the *improvedBM* model. All the misspellings that can not be found by the *improvedBM* model, were not identified by this error model either.

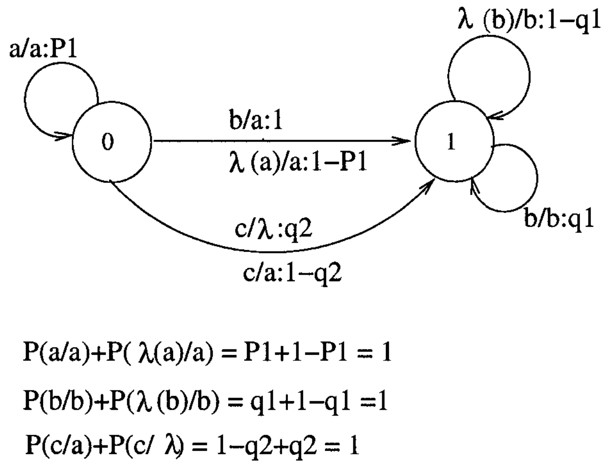


Figure 7.10: The *difInsert* Model

	Total	1-best	2-best	3-best
First Set	132	110	122	124
(%)		83.3	91.42	93.94
Second Set	91	62	68	70
(%)		68.13	74.73	76.92

Table 7.6: Result of *difInsert* Model

In the result of the second set of misspellings, the accuracy is lower than that of the *improvedBM* model and the *totalOne* model. We trained this error model by using the same data set. The result shows that the different insertion error model may not correct as many misspellings as *improvedBM* error model and *totalOne* error model do.

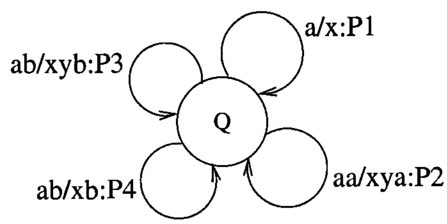
7.3.3 No-empty model

We now consider the no empty model. In this model, we will only consider edit operations α/β with $|\alpha| \geq 1$ and construct the channel using only those edit operations. In other words, this model doesn't consider the single insertion edit operation (λ/y). As before, in this new model we don't want to consider the insertion error individually. Rather we combine the single insertion error with the previous letter. For example, given an error pair $(thre/there)$, the optimal alignment of it is: $(t/t), (h/h), (\lambda/e), (r/r), (e/e)$. However, in this model, the single insertion is not considered. Therefore, the following edit operations are generated by expanding the characters in the alignment: $(th/th), (h/he), (r/er), (re/re), (th/the), (hr/her), (re/ere)$.

Figure 7.11 illustrates this error model with a very simple channel:

The result of using our training data in this error model is shown in Table 7.7:

In the first testing set of *NoEmpty* model, the accuracy of *1-best* set only reach 69.7%, the accuracy of *2-best* is 72.73%, and the accuracy of *3-best* doesn't change much, still less than 75%. The accuracy shown in the second data set is also lower than that of the other error models. The reason of the lower accuracy of this error model is



P(a/x) + P(aa/xya) + P(ab/xyb) + P(ab/xb) = P1+P2+P3+P4 = 1

Figure 7.11: The *noEmpty* Model

	Total	1-best	2-best	3-best
First Set	132	92	96	97
(%)		69.7	72.73	73.48
Second Set	91	46	47	47
(%)		50.55	51.65	51.65

Table 7.7: Result of noEmpty Model

that all the single letter error pairs have been removed, only those edit operation α/β with $|\alpha| \geq 2$ are considered. So the size of the training set is reduced. We know that the more training data we have the more accuracy we can get. However, a number of misspellings that can be found in the *1-best* set by using this error model can not be identified in the *1-best* set by using other models such as *ofd, converence.* Therefore, if there were enough training data for this error model, it might get good result. But in this research, small training data sets are important since we are interested in modeling the error behaviour of a specific typesetter in this study. In practice, it is hard to get big data set from a certain typesetter.

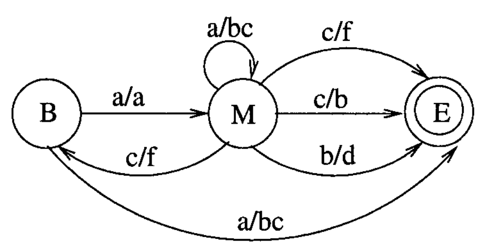
7.3.4 Three state model

The idea of this error model is borrowed from [3]. In this error model, we assign probabilities depending on the position in the string where an edit operation occurs. This can be the start of the word, middle of word and end of word. The position of an edit operation (α/β) is determined by the location of the substring α in the word. Positional information is a powerful conditioning feature for rich edit operations. For example, people rarely mistype *antler* as *entler*, but often mistype *reluctant* as *reluctent.*

Compared with other error models, the *threeState* model has three states rather than only one state. We use three vectors to store all beginning sequence pairs, middle sequence pairs and end sequence pairs and we calculate the probabilities for each of them. For example, for the error pair (aab/acb), the set of beginning sequence pairs is: $\{(a/a), (aa/ac)\}$, the set of middle sequence pairs is: $\{(a/c)\}$, and the set of end sequence pairs is: $\{(b/b), (ab/cb)\}$.

When applying *threeState* error model, we checked different vector according to the position of the substring α in the source (dictionary) word. Similarly, according to the theory in Chapter 4, we define the sum of all probabilities that start from same state with the same input to be 1.

Figure 7.12 illustrates this error model with a simple example:



In State B: P(a/a) + P(a/bc) = 1
In State M: P(a/bc) = 1, P(c/f) = 1, P(b/d) = 1
In State E: P(c/f) + P(c/b) = 1

Figure 7.12: The *threeState* Model

There are 2172 sequence pairs (α/β), including the case of $\alpha = \beta$, generated for the set of beginning pairs; 4824 pairs generated for the set of middle pairs and 2164 pairs generated for the set of end pairs. The result of using our training data in this error model is shown in Table 7.8:

This is the last error model that has been tested. The accuracy result of this *threeState* model shows that it is the good choice for the misspelling correction problem. In the first testing set, the accuracy of *1-best* list is 81.06%, the *2-best* list accuracy result is much better, which is 90.91%, and the accuracy of *3-best* list can

	Total	1-best	2-best	3-best
First Set	132	107	120	126
(%)		81.06	90.9	95.45
Second Set	91	68	79	82
(%)		74.73	86.81	90.11

Table 7.8: Result of *threeState* Model

reach to 95.45%. The accuracy of *1-best* list is lower than the result from the *totalOne* model. But the accuracy of *2-best* set by using this model increases by 9%, which is better than *totalOne* model. The accuracy of the *3-best* list increases by 5%, which is a litter higher than *improvedBM* model. The accuracy of all *1-best, 2-best* and *3-best* lists in the second testing set is higher than all other error models.

7.4 Comparison

In the last part of this chapter, we list all the result of each error model together in Table 7.9 and draw a picture for them.

In Table 7.9, we notice that, in all cases, the accuracy of the second testing set is lower than the accuracy of the first testing set. Recall that, the first testing set contains a number of misspellings that have already been used in the training data set to train the error model. But no misspelling in the second testing set has been used before. Therefore, the result in the first set is better than the result in the second set.

		Total	1-best	2-best	3-best
First Set	improvedBM	132	114	122	126
	(%)		86.36	92.42	95.25
Second Set	improvedBM	91	66	76	82
	(%)		72.53	83.52	90.11
First Set	noEmpty	132	92	96	97
	(%)		69.7	72.73	73.48
Second Set	noEmpty	91	46	47	47
	(%)		50.55	51.65	51.65
First Set	totalOne	132	108	115	121
	(%)		81.82	87.12	91.67
Second Set	totalOne	91	64	72	75
	(%)		70.33	79.12	82.42
First Set	difInsert	132	110	122	124
	(%)		83.33	91.42	93.94
Second Set	difInsert	91	62	68	70
	(%)		68.13	74.73	76.92
First Set	threeState	132	107	120	126
	(%)		81.06	90.91	95.45
Second Set	threeState	91	68	79	82
	(%)		74.73	86.81	90.11

Table 7.9: Table of All Results

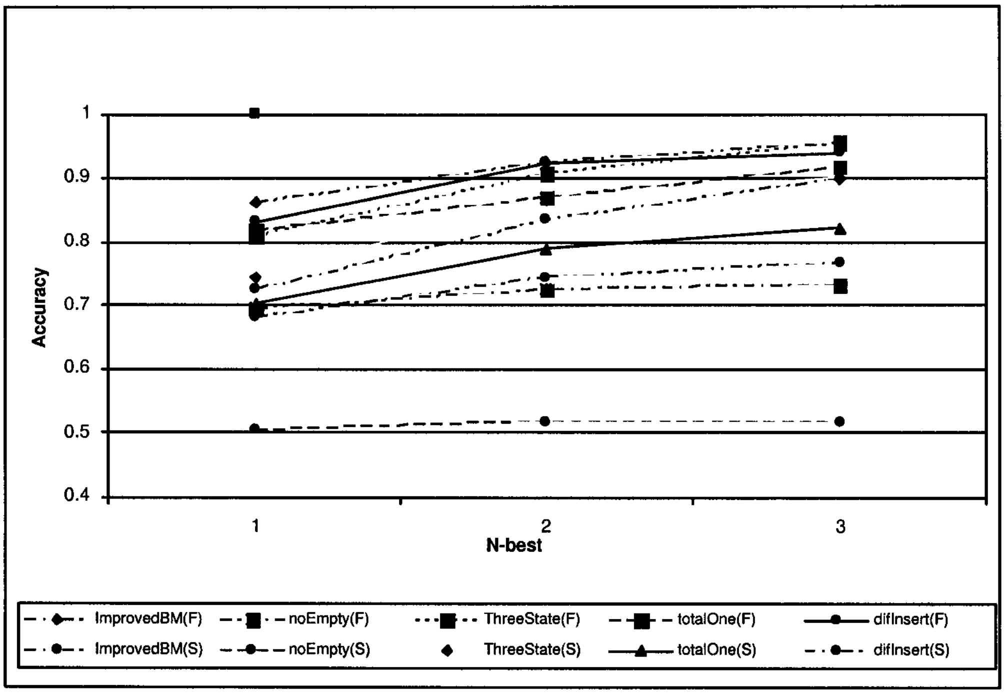


Figure 7.13: Comparison of Five Models

As we can see from the figure 7.13, the *threeState* error model has the highest accuracy in the first testing set and the second testing set since the *threeState* error model consider the positional information which is a powerful conditioning feature. The *noEmpty* error model has the lowest accuracy in both two testing set since it has less training data to train the error model.

Chapter 8

Conclusion and Future Work

In this thesis, we introduced a general methodology to define error models describing different types of errors in information processing application, discussed the channel computation for the specific user with the application to spelling errors, improved the Brill and Moore spelling correction method by employing the theory of the general methodology. Several data structures and algorithms have been used in this thesis to help us apply the general methodology, compute channels and improve the Brill and Moore method:

Data Structures:

- Trie: It was used to store the dictionary
- Binary Search Trie (*bst*): It was used to store all (α/β) pairs

Algorithms:

- String Distance algorithm: The concept of string distance algorithm was used in this thesis to develop the algorithm that generates string pairs from given data

- *N-best* string algorithm: The algorithm was used to find *n-best* distinct words in *WFST*.

Aside from the above data structures and algorithms, the AT&T tools were also used in the thesis to create *WFST*, *WFA* and compute the composition between them.

8.1 Conclusion

As illustrated in Chapters 4 and 6 of the thesis, with its capability of using basic tools from stochastic automata to describe various error situations, our general methodology is able to provide us with a tool to derive different error models (such as Church and Gale error model, Mays and Damerau error model, Brill and Moore error model) in the same way.

The same experimental results from applying the dynamic programming method and the channel correction algorithm to the improved Brill and Moore error model in Chapter 7 of the thesis shows that the latter can be used to correct the errors described by the error model channels.

The better results in *1-best, 2-best* and *3-best* lists from using the improved Brill and Moore error model than the original Brill and Moore model demonstrates that the general methodology with its capability of utilizing the probabilities assignment and transitions conversion method can assist us in creating better error models in the information processing system.

As described in the Chapter 7, among the four modified error models from the improved Brill and Moore error model, the *threeState* error model with the consideration of positional information has the highest accuracy for the *3-best* list in the first testing set and has the highest accuracy for all *1-best, 2-best* and *3-best* lists in the second testing set. The *noEmpty* error model with the less training data has the lowest accuracy for all *1-best, 2-b est* and *3-best* lists in both two testing sets. This model may have a better performance result with a larger data set; however, as we are interested in modeling the error behaviour of a specific typesetter in this research, having small training data sets is essential to the construction and implementation of our methodology. The other factor that affects us in choosing the small training data sets is that it is usually difficult to get large data sets from a certain typesetter in practice.

8.2 Future Work

The possible future work of this thesis might concentrate on the following aspects:

- *Add Source Model P(W)*:

  As mentioned in the thesis, a source model P(W) is a model that describes the probabilities of a word w to be produced by the text generator. It is usually used with the error model $P(s|w)$ together for the probabilistic technique of correcting misspellings. This research focuses on the general methodology of defining an error model, which can be considered as a finite state source model if we omit the output parts of the channel transitions. In the future, we can explore what happens to the results if we consider a source model $P(W)$ independently of the error model.

- *Incorporate the keyboard layout into the string pair generating algorithm*:

  The string pair generating algorithm described in this thesis uses Levenshtein distance to compute the string distance without considering the impact of keyboard layout on the computation. The consideration of the keyboard layout may give a more satisfying result in the string distance computation. In order to get a better result, the author also experimented with the Church and Gale's confusion matrices to generate string pairs with considering of the keyboard layout. However, due to the limited knowledge of statistical data in these four matrices, the results were not satisfying. Therefore, finding an appropriate way that incorporate the keyboard layout into the string pair generating algorithm is our next step of this research.

- *Test more data*:

  In this research, we collected 12 pairs of files to train various error models. Two testing sets with total 225 misspellings were used to test the improved Brill and Moore error model and four other modified error models. All these data are from a specific user — the author of this thesis. To have a more representative data set that covers a wide range of situations, it is necessary in the future work of the research to have testing data sets from a variety of specific users and run the experiments for them.

- *Employ appropriate tools to assist in collecting training data*:

  In this research, a manual approach was employed to collect training data to generate string pairs for the specific user. The author used two Microsoft word documents files to store the original and modified copy of a data set each time. However, in practice, to collect training data sets from a large group of specific users, such an approach may not seem to be realistic. Thus, it could be an appropriate next step in this research to find or develop some sort of tools that will enable us to automatically keep track of every change that a specific user makes to a training data set.

Bibliography

[1] Aho, A. V., Hopcroft, J. E., Ullman, J. D. *The design and analysis of computer algorithms* Addison Wesley publishing company, c1974, pp. 207-209.

[2] Braines, S. N., Napalkov, A. V., Svechinski, W. B. *Neurocybernetics*, Berlin 1964.

[3] Brill, E., Moore, R.C. *An Improved Error Model for Noisy Channel Spelling Correction*, 38th Annual Meeting, Association for Computational Linguistics, 2000.

[4] Chow, Y., Schwartz, R. *The N-Best Algorithm: An Efficien Procedure for Finding top N Sentence Hypotheses*, Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP '90), Albuquerque, New Mexico, April 1990, pp. 81-84.

[5] Church, K. W. and Gale, W. A. *Probability scoring for spelling correction*, Statistics and Computing, 1991, 1:93–103.

[6] Damerau, Fred J. *A technique for computer detection and correction of spelling errors*, Communications of the ACM, 7, 3 (March 1964), pp. 171-176.

[7] Gailil, Z. and Park, K., *An improved algorithm for approximate string matching*, SIAM Journal on Computing, 19, 1990, pg 989- 999.

[8] Grudin, J. T. *Error patterns in novice and skilled transcription typing* in *Cognitive Aspects of Skilled Typewriting*, W. E. Cooper, Ed. Springer-Verlag, New York, 1983.

[9] Gusfield, D., *Algorithms on Strings, Trees, and Sequences: Computer Science and Computation Biology*, Cambridge University Press, New York, NY, USA, 1997, pp. 215-223

[10] Hopcroft, J. E., Motwani, R. and Ullman, J.D. *Introduction To Automata Theory, Languages, And Computation*, Second Edition, Addson-Wesley, 2001

[11] Jelinek, F. *Self-organized Language Modeling for Speech Recognition*, IBM Report, 1985.

[12] Jokinen,P., Ukkonen, E., *Two algorithms for approximate string matching in static texts*, Proceedings of the 16th International Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, 52, 1991, pg 241- 248.

[13] Jurafsky, D., Martin, J. H. *Speech and Language Processing*, Prentice Hall, 2000, pp. 142 199.

[14] Kari, L., Konstantinidis, S., Perron, S., Wozniak, G., Xu, J. *Finite-state error/edit-systems and difference-measures for languages and words*, Technical Report No. 2003-01, 2003, April.

[15] Kernighan, M. D., Church, K. W., and Gale, W. A. *A spelling correction program based on a noisy channel model*, Proceedings of COLING-90, The 13th International Conference on Computational Linguistics, Vol. 2. Hans Karlgren, Ed. pp. 205-210.

[16] Kruskal, J.B. *An Overview of sequence Comparison*, Sankoff and Kruskal (eds.), 1983, pp. 1-44.

[17] Kukich K. *Techniques for Automatically Correcting Words in Text*, ACM Computing Surveys, Vol.24, No. 4, 1992, pp. 377-440.

[18] Landau, G., Vishkin, U., *Fast string matching with k differences*, Journal of Computer and System Sciences, 37, 1988, pp 63-78.

[19] Levenshtein, V. I. *Binary codes capable of correcting deletions insertions and reversals*, Soviet Physics-Doklandy, 10(8), 1966, pp. 707-710.

[20] Manber, U., *Introduction to Algorithms A creative Approach*, Reading Mass., Don Mills, Ont.: Addison Wesley, 1989, pp. 155-158.

[21] Mayes, E., F. Damerau, *Context Based Spelling Correction*, Information Processing and Management, 1991, 27(5): pp. 517-522.

[22] McIlroy, M.D., *Development of a spelling list* IEEE Transactions on communications, 30(1), 1982, pp. 91-99

[23] Mohri, M., Pereira, F., Riley, M. *Weighted Automata in Text and Speech Processing*, 12th European Conference on Artificial Intelligence, 1996.

[24] Mohri, M., Riley, M., *An efficient algorithm for the n-best-strings problem*, Proceedings of the International Conference on Spoken Language Processing 2002 (ICSLP '02). Denver, Colorado, September 2002.

[25] Morris, R., Cherry,L. L. *Computer detection of typographical error*, IEEE Trans. Profess. Commun., 1975, PC-18,1, pp. 54-63.

[26] Pereira, F.C.N., Riley, M.D. *Speech Recognition by Composition of Weighted Finite Automata*, In [29] 1997, pp. 431-453.

[27] Pollock, J. J., Zamora, A. *Automatic spelling correction in scientific and scholarly text*, Commun. ACM 27, 4 (Apr.), 1984, pp. 358-368.

[28] Ristad, E. S., Yianilos, P. N. *Learning String-Edit Distance*, IEEE Transaction on Pattern Analysis and Machine Intelligence, Vol. 20, No. 5, May 1998, pp. 522-532

[29] Roche, E., Schabes, Y. (eds.) *Finite-state Language Processing*, The MIT Press, Cambridge, MA., 1997

[30] Soong, F., Huang, E. F. *A Tree-Trellis Based Fast Search for Finding the N Best Sentence Hypotheses in Continuous Speech Recognition*, Proceedings of the

International Conference on Acoustics, Speech, and Signal Processing (ICASSP '91), Toronto, Canada, 1991, pp.705-708.

[31] Toutanova, K., Moore, R.C. *Pronunciation Modeling for Improved Spelling Correction*, Proceeding of the 40th Annual Meeting of the Association for computation Linguistics (ACL), July 2002, pp. 144-151

[32] Ukkonen,E., *Algorithms for approximate string matching*, Information and control, 64, 1985, pp 100-118.

[33] Ukkonen,E., *Finding approximate patterns in strings*, Journal of Algorithms, 6, 1985, pp 132-137.

[34] Yannakoudakis, E. J., and Fawthrop, D. *The rules of spelling errors*, Information Processing & Management 19,2,1983, pp. 87-99.

[35] Allison, L., *Tries*, http://www.csse.monash.edu.au/ lloyd/tildeAlgDS/Tree/Trie/, Visited on Sep.14, 2003

[36] Black, P.E., *trie*, http://www.nist.gov/dads/HTML/trie.html, Visited on Sep.14, 2003

[37] Bentley, J., Sedgewick, B., *Ternary Search Trees*, http://algorithm.myrice.com/resources/technical_artile/ternary_search_tree/terstree.htm, Visited on Sep.14, 2003

[38] Mohri, M., Pereira, F.C.N., *Non-commercial license agreement and software [binary]*, http://www.research.att.com/sw/tools/fsm/, Visited on Seq.14, 2003