A post processing system for global correction of Ocr generated errors

Bryan Edward Bullard
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Part of the Higher Education Commons

Repository Citation
https://digitalscholarship.unlv.edu/rtds/254

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
A post processing system for global correction of OCR generated errors

Bullard, Bryan Edward, M.S.
University of Nevada, Las Vegas, 1993
A POST PROCESSING SYSTEM FOR
GLOBAL CORRECTION OF OCR
 GENERATED ERRORS

by

Bryan E. Bullard

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

in

Computer Science

Department of Computer Science
University of Nevada, Las Vegas
April 1993
The Thesis of Bryan E. Bullard for the degree of Master of Science in Computer Science is approved.

Kazem Taghva
Chairperson, Kazem Taghva, Ph.D.

Tom Nartker
Examinng Committee Member, Tom Nartker, Ph.D.

Ajoy Datta
Examinng Committee Member, Ajoy Datta, Ph.D.

Ashok Singh
Graduate Faculty Representative, Ashok Singh, Ph.D.

Ronald Smith
Dean of the Graduate College, Ronald Smith, Ph.D.

University of Nevada, Las Vegas
April 1993
ABSTRACT

This thesis discusses the design and implementation of an OCR post processing system. The system is used to perform automatic spelling detection and correction on noisy, OCR generated text. Unlike previous post processing systems, this system works in conjunction with an inverted file database system. The initial results obtained from post processing 10,000 pages of OCR'ed text are encouraging. These results indicate that the use of global and local document information extracted from the inverted file system can be effectively used to correct OCR generated spelling errors.
# TABLE OF CONTENTS

ABSTRACT .................................................................................................................. iii

LIST OF FIGURES ....................................................................................................... v

LIST OF TABLES ........................................................................................................ vi

ACKNOWLEDGEMENTS ........................................................................................... vii

CHAPTER 1  INTRODUCTION ........................................................................... 1
  Background ............................................................................................................. 2

CHAPTER 2 THE STRING MATCHING PROBLEM .................................. 4
  String Equivalence ............................................................................................. 5
  String Similarity ............................................................................................... 10

CHAPTER 3 THE SPELLING DETECTION AND CORRECTION PROBLEM .......... 19

CHAPTER 4 OCR POST PROCESSING SYSTEM ....................................... 30
  System Implementation ....................................................................................... 31
  Future Considerations ....................................................................................... 42

CHAPTER 5 CONCLUSION ............................................................................... 45

BIBLIOGRAPHY ........................................................................................................ 46
LIST OF FIGURES

Figure 1 A brute force string matching algorithm .............................................. 5
Figure 2 "database" equivalence class ................................................................. 7
Figure 3 Illustration of affix analysis ................................................................. 8
Figure 4 "type" equivalence class ................................................................ 9
Figure 5 Example of edit operations ................................................................. 13
Figure 6 Example of Baeza-Yates and Gonnet exact matching algorithm ....... 15
Figure 7 Array R for approximate matching with a single insertion ................. 17
Figure 8 Array R₁ for approximate matching with a single insertion ............... 17
Figure 9 Dictionaries for two-level search strategy ........................................... 25
Figure 10 Global correction post processing system .......................................... 41
Figure 11 Post processing system with planned enhancements ......................... 44
LIST OF TABLES

Table 1  Example of text strings and associated skeleton keys ......................... 26
Table 2  Example of text strings and associated omission keys ........................... 27
Table 3  Experiment 1 cumulative centroid report ........................................... 33
Table 4  Experiment 1 cumulative error report ................................................... 34
Table 5  Experiment 2 cumulative centroid report .............................................. 35
Table 6  Experiment 2 cumulative error report ................................................... 36
Table 7  Partial listing of confusion matrix ......................................................... 37
Table 8  Results of local-info and confusion matrix centroid selection ................. 39
Table 9  Cumulative error reports for experiments 2 and 3 ................................ 40
Table 10 Cumulative percentage of correct changes ............................................ 40
ACKNOWLEDGEMENTS

I would like to thank Dr. Taghva for his insight, expertise and guidance throughout this whole process. I am most grateful to Julie and Allen for their ideas, and the endless hours they spent working on this project. A special thanks to Nancy for her giving up her time to assist in editing the final paper. Finally, to Michelle I thank you for your understanding and patience.
CHAPTER 1

INTRODUCTION

Once reserved for a limited number of applications, full text databases and Information Retrieval (IR) systems are now becoming more and more common in business and government. The need to store, manage and retrieve increasingly large collections of unformatted data, have contributed to the emergence of IR systems in mainstream computing. As businesses seek competitive advantages, IR systems will be relied upon more and more. To facilitate the migration to IR systems, Optical Character Recognition (OCR) devices\(^1\) are being employed to convert existing paper documents to machine readable ASCII text for storage in textual databases.

Like IR systems, OCR until recently was limited to a narrow range of applications. However, newer technologies have improved the speed and quality of OCR devices; OCR is now experiencing rapid growth. Generally, the process of converting paper documents to electronic documents via OCR involves scanning a paper document to obtain an image. The image is then fed into the OCR device for conversion to ASCII text. The ASCII text is then stored and indexed for retrieval. The advantage of using OCR is that it is much faster and cheaper than having to manually retype paper documents into the computer. The drawback to using OCR is that the ASCII text must be manually corrected. The best OCR devices available today are able to achieve upwards of 99 percent conversion accuracy [12]. This translates to approximately 20 - 25 misspellings per page. Correction of these misspellings is both time consuming and expensive. The U.S. Department of Energy

\(^1\)Devices here and throughout this paper refer to either hardware or software OCR implementations.
is considering a large scale OCR project. It is estimated that 60 percent of the cost for this project will be for manual correction of the text [15]. Projects like this demonstrate the need for low cost, efficient ways to correct OCR generated text.

**Background**

Many sophisticated algorithms have been proposed to accurately convert document images into ASCII text [21]. While each of these schemes improves on the other, many variables affect how any one scheme performs. Font type, font size, skew, broken characters and the copy quality of the scanned page are just a few of the attributes that affect the accuracy of an OCR device. Further, the geometric similarity of many characters constitutes a major source of OCR errors even for the best quality print and copy. For example, the character "i" is often mistaken for the characters "1" or "l", and the character "c" is often mistaken for the character "e". For a good quality paper copy, one will find that the majority of errors are caused by this type of substitution, resulting in single error misspellings. Newer OCR devices have spelling error detection and correction algorithms incorporated in them. However, many spelling errors still go uncorrected. The failure of OCR devices to detect and correct errors made during image conversion is due largely to the fact that the OCR process is designed to be fast. In order to maintain high rates of throughput, techniques that could be used to enhance the misspelling detection and correction features of OCR devices are omitted. As a result, other methods for performing automatic correction of OCR generated text must be examined.

In this thesis a post processing system that automatically detects and corrects OCR generated spelling errors is presented. The system uses approximate string
matching and error detection and correction techniques along with empirical data in the form of a confusion matrix. However, the features that separate this system from any previous system of its kind are the following: first, the system is built around an inverted file database system. This allows for the use of local (document level) and global (the entire document collection) information to correctly identify and correct misspellings. Second, the confusion matrix is built as words are corrected by the post processing system. Constructing the confusion matrix during post processing ensures that the error information it contains is a reflection of the document collection being processed.

The system reported in this paper is being used in conjunction with a study to determine how OCR generated spelling errors affect the recall and precision of IR systems[19]. Initial results from the post processing of approximately 10,000 pages of OCR'ed text are encouraging. For word lengths greater than and equal to seven, the cumulative percentage of correct changes is 87 percent.

The remainder of this thesis is organized as follows. In Chapter 2, exact and approximate string matching are reviewed. Particular attention is given to agrep, a pattern and string matching utility that allows the user to specify the edit distance to be used during the search. (Edit distance is discussed in detail in Chapter 2). Chapter 3 discusses spelling correction techniques and provides examples of systems employing such techniques. Chapter 4 discusses the post processing system and provides a breakdown of the results from our experiments.
CHAPTER 2

THE STRING MATCHING PROBLEM

The ability to perform both exact string matching and approximate string matching is vital to any spelling error detection and correction system; if the system cannot locate errors, it cannot correct them. In this chapter, the notions of exact and approximate string matching are surveyed. Particular emphasis is given to several different techniques for performing approximate string matching.

In general, the string matching problem is to find all occurrences of some search string $s$ within a text string $t$. Formally stated, given a search string $s$ whose length is $m$ and a text string $t$ of length $n$ where $n > m$, find all occurrences of $s$ in $t$.

This problem is solved by comparing each character of $s$ with each character of $t$. For example, given the search string $s = (s_1, s_2, s_3, ..., s_m)$ and the text string $t = (t_1, t_2, t_3, ..., t_n)$ where $n > m$, find all occurrences of $s$ in $t$. Figure 1 shows a brute force algorithm that compares all the characters of $s$ with all the characters of $t$, and reports any occurrences of $m$ consecutive matches.

This type of string matching is referred to as exact string matching. Exact string matching is generally very easy to implement and quite useful. However, as the name indicates, exact matching will retrieve information only when the search string and the text string match exactly. The effectiveness of exact matching is diminished in an environment with numerous misspellings.

Approximate string matching is a more general technique that goes beyond exact string matching. It uses the special qualities of strings to match two strings that
are not exact but may be "close". Two different approaches are cited in [4] that try to quantify the "close"ness of two strings. The notion of string equivalence is based on whether or not any two strings in question have the same or nearly the same meaning. The notion of string similarity uses the physical characteristics of any two strings to determine whether or not they are close.

\begin{verbatim}
a = 0
b = 0
while (a < m and b < n) do
    while (s[a] = t[b]) do
        a++
        b++
    endwhile
    if (a = m - 1) then
        report a match
    endif
    a = 0
    b++
endwhile
\end{verbatim}

Figure 1: A brute force string matching algorithm

String Equivalence

"Two strings are considered to be equivalent if their appearance is different but they can be substituted for each other without changing the meaning" [4]. String equivalence is the lesser-known interpretation of approximation. Nonetheless, string equivalence plays a vital role in text retrieval systems where documents may contain several variations of the same word. It is also very important in the use of on-line dictionaries, where the same word may have different forms. The following strings illustrate this point[4]:
database, data-base, Database, Data-Base

While each of these strings looks different, one can indeed be substituted for the other without a change in meaning. When a search is performed using the string "database", each of the strings or documents containing any of the strings in the example above should be retrieved. Similarly, the same set of strings or documents should be retrieved if any of the other strings above is used as a search string. In the best case, however, (assuming case insensitive searching) exact matching will only retrieve records or documents containing the strings "database" and "Database". It will be shown that by utilizing the notion of equivalence classes, string matching problems like the one just discussed can be greatly simplified.

String equivalence is based on the mathematical concept of equivalence classes [1]. By introducing the equivalence relation "≈" on some set S of all possible strings, such that for strings r, s, t in S the following properties are obtained[4]:

(i) \( s \approx s \)                reflexivity
(ii) \( s \approx t \Rightarrow t \approx s \) symmetry
(iii) \( r \approx s \) and \( s \approx t \Rightarrow r \approx t \) transitivity

The reflexivity and symmetry properties are intuitive. A string s is equivalent to itself; and if a string s is equivalent to a string t, then t is equivalent to s. However, the transitivity property makes string equivalence different from other measures of string likeness and allows the string matching problem for equivalence to be restated as the following:

Given s in S, find all t in T such that \( s \approx t \).
The use of equivalence classes is a useful mechanism for grouping or clustering equivalent strings. The equivalence relation divides the set S of all possible search strings into subsets, $S_1, S_2, S_3, \ldots, S_n$, where each $S_i$ is called an equivalence class. All of the strings in an equivalence class are equivalent to each other, and are not equivalent to any string within any other equivalence class. Each equivalence class is identified by a unique string (called a centroid) that is used to identify that equivalence class. This centroid is the canonical form for the class, and it is usually the root (or stem) of all the words contained within that class. The equivalence problem can now be reduced to the following: find all strings $t$ in the equivalence class $T$ such that the centroid of $T$ is equivalent to the canonical form of $s$.

Using this example, the string "database" is already in canonical form. If all of the formatting characters are removed from the other strings, they all reduce to the string "database". Figure 2 illustrates the "database" equivalence class.

![Figure 2: "database" equivalence class](image)

More important than the ability to treat the same word with different formatting styles as equivalent (as in the previous example), is the ability to treat grammatical variants of the same word as equivalent. When one considers that virtually every noun in the English language can be augmented with an "s" and
virtually every verb can be augmented with "re", "ed", and "ing", it is clear that there is a need to somehow establish an equivalence between the different forms of these words. Using more sophisticated techniques than that cited above, equivalence between grammatical variants of strings can be achieved. These techniques are most often employed in full text retrieval systems, as well as spelling error detection and correction systems that use dictionary lookups. In some cases, these techniques are used to support the use of a thesaurus.

The first step in determining equivalence between grammatically different strings is to reduce each word to its root form. This process is called affix analysis. Affix analysis involves identifying and removing prefixes and/or suffixes of a word. Using the string "retyping", the following example illustrates this point.

In this example, prefixes and suffixes are used to reference when the action of "type" is taking place. When a search is performed using the string "retyping", the system should return any records or documents that contain either "retyping" or "type". Furthermore, the system should return all of the words that are in the "type" equivalence class. Figure 4 shows the "type" equivalence class containing the words "typed", "typing", "retype", "retyping" and "retyped". The exact methods used to
perform affix analysis are out of the scope of this paper, but some affix analysis
techniques are discussed in [4] and [9].

As previously stated, the notion of string equivalence is most useful in full text
databases and on-line dictionaries. In the case of a full text database, thousands of
documents and tens of thousands of distinct words are stored. The purpose of a full
text database, aside from storing information, is to retrieve all relevant documents in
response to a user's query. The use of canonical forms aids this process in several
ways. First, the user is not required to be exact in his/her search. Second, by not
having to store each unique word in the index, the use of disk space is minimized. In
addition, search times are decreased as fewer comparisons are required. Hall[4] notes
two primary ways canonical forms and equivalence classes are used within text
retrieval systems.

The first method reduces each word within a document to its canonical form
at the time of input. This method is the most widely used in full text databases.
When the database is loaded, each word (as it is being processed) is stripped of all
formatting characters, prefixes and suffixes and stored in the index in its canonical
form. A pointer is then established to indicate in which document that particular

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{type-equivalence-class.png}
\caption{	extit{"type" equivalence class}}
\end{figure}
word appears. Then, if another word in another document is entered into the database that reduces to the same canonical form, another pointer is established. This new pointer points to the document that was just added to the database. When a query is performed, the search string is reduced to its canonical form; the search then begins. The only drawback in using this method is that the native form of each word (the way the word appears in the document) does not appear in the index.

The other method identified by Hall reduces words to their canonical form at the time the query is performed. Each unique word that is loaded into the database is stored in the index in its native format. At the time that a search is performed, both the search string and the indexed string are reduced to their respective canonical forms prior to the compare. This method is rarely used because of the disk space required to store the index and the overhead associated with reducing each string to its canonical form prior to each comparison.

The notion of string equivalence is very useful in text retrieval systems since it relieves the user from having to be exact in his/her queries. However, this technique for string matching will only work in an environment where spelling errors are few or do not exist. There are several reason for this. First, incorrectly spelled words will fail to be incorporated into an equivalence class. Second, exact matching is the technique most often used to compare the search string and the terms in the index. Therefore, spelling errors may result in relevant information not being retrieved.

**String Similarity**

String similarity is by far the most common interpretation of approximation. As demonstrated in the previous section, equivalence classes provide a very powerful
concept in which strings that have like meanings can be grouped. However, in order for those techniques to work properly, all strings being considered must be spelled correctly. String similarity, on the other hand, uses the patterns of words to approximate how much alike two strings are. Therefore, even if one of the strings being compared is misspelled, it is possible that the patterns of the strings are close enough to make them similar. The following is an often-cited scenario in the area of approximate string matching:

At the time an airline reservation is made for someone, that individual's name is entered into the airline database. At some later time, someone may wish to retrieve the reservation record for that particular person. Three possibilities arise. The first is that the search string used to retrieve the person's record from the database exactly matches the string stored in the index, and the record is retrieved. The second possibility is that the person's name was misspelled when it was initially entered into the database. Consequently, when someone tries to retrieve that person's record using the correctly spelled name, he/she will either retrieve someone else's record or retrieve no records at all. The last scenario is that the individual's name was entered into the database correctly, but the search string is spelled incorrectly. Again, either the wrong record will be retrieved or no records will be retrieved.

The notion of string similarity addresses problems like the example just mentioned. Simply stated, given some search string $s$ and some text string $t$, is the pattern of $t$ "close" enough to $s$ that $t$ should be retrieved? For example, using the airline database example, passenger "Smith" wants to confirm his flight reservation. The operator inadvertently queries the system with the string "Smeth". Without the use of similarity matching, Mr. Smith's reservation will not be retrieved. However,
with the use of some similarity measure, it is possible that Mr. Smith's record might be retrieved.

Several different methods exist for determining string similarity, but the most common methods are based on the Damerau-Levenstien metric [4]. At the heart of the Damerau-Levenstien metric is a difference function which satisfies the triangular inequality. One such function is $d : S \times S \to R$, where $d(s,t)$ produces some real number $k$. This function has the following properties [4]:

(i) $d(s,t) \geq 0$
(ii) $d(s,t)=0$ if and only if $s = t$
(iii) $d(s,t) = d(t,s)$
(iv) $d(s,t) + d(t,r) \geq d(s,r)$ triangular inequality

The use of this function or any other difference function that satisfies the triangular inequality allows the string similarity problem to be stated as follows. Find all occurrences of the search string $s$ in the text $T$, such that the difference $d(s,t)$ between $s$ and the text string $t$ is equal to $k$.

The integer $k$ is known as the edit distance. The edit distance measures how many edit operations are required to change one string to another. The notions of edit distance and edit operations were introduced by Damerau [2] who reported that 80 percent of all typing errors are caused by one of the following typing mistakes: insertion, deletion, substitution or transposition. Following Wagner and Fisher[20], an edit operation is defined as the following: let $\Sigma$ be a finite alphabet. An edit operation is a pair of strings $(a,b) \neq (e,e)$ over $\Sigma$ of length less than or equal to 1. We say string $x$ resulted from $w$ in notation $w \Rightarrow x$, if there are strings $\alpha$ and $\beta$ such that $x = \alpha b \beta$ and $w = \alpha a \beta$. We say $(a,b)$ is a substitution if $a \neq e$ and $b \neq e$, a delete operation if $b = e$ and $a \neq e$, and an insertion operation if $a = e$ and $b \neq e$. For
example, the edit distance between the strings in Figure 5 is 3. The "a" in the first string has no corresponding character (a deletion), the "u" in the first string corresponds to the "o" in the second string (a substitution) and the "s" in the second string corresponds to no character in the first string (an insertion).

\[
\begin{array}{c}
\text{automobile} \\
\mid \mid \mid \\
\text{otomobiles}
\end{array}
\]

Figure 5: Example of edit operations

Two very popular methods of approximate string matching have evolved from the Damerau-Levenstien metric. The first is the problem of \textit{string matching with k mismatches} and the second is \textit{string matching with k differences} (also known as the \textit{k differences} problem). The string matching with k mismatches technique is very limited in its ability to detect mismatches between strings, as it only detects substitution errors. String matching with k differences, on the other hand, is a general approximate string matching technique that will detect any of the edit operations addressed earlier. This is most important in the correction of OCR-generated errors, since any of the edit differences already noted (except transposition) are likely to occur.

Many algorithms exist that address the \textit{k differences} problem. The methods reported in [3], [5] and [6] are just a few. Recently, several papers have been produced, each improving on the previous one. However, Wu and Manber [23] presented an algorithm in 1991 that is by far more flexible than any other approximate string matching algorithm to date. This algorithm is incorporated in the string and pattern matching tool called \textit{agrep}. Agrep allows for searching of strings in a text file
that are exactly $k$ edit operations from the search string. Additionally, agrep allows for wild card searching, searching for ranges of characters (e.g., "0-9") and searching for arbitrary sets of characters (e.g., \{a, e, i, o, u\}). In addition to performing these searches, agrep allows for the replacement of a single character with a set of characters.

To fully understand how agrep works, the original exact matching algorithm of Baeza-Yates and Gonnet [23] must be examined. This algorithm, like most recent string matching algorithms, uses a dynamic programming technique. Let $R$ be a bit array of size $m$ where $m$ is the number of characters in the search string $s$. The array $R_j$ indicates all matches of $S$ up to the $j$th position. More precisely, if the first $i$ characters of $S$ exactly match the last $i$ characters up to $j$ in the text, then $R_j[i] = 1$. For each $R_j[i] = 1$, we need to check whether or not $t_{i+1}$ is equal $s_{i+1}$. If $R_{j+1}[i] = 1$, then a complete match has been found and is returned. If $R_{j}[i] = 0$, then there has been no match up to $i$ and there cannot be a match up to $i+1$.

This process above can be summarized as follows: Initially, $r_0[i] = 0$ for all $i$, where $1 \leq i \leq m$ and $r_0[0] = 1$ to avoid the case of $i = 1$. The transition for $R_{j+1}[i]$ is then seen as the following:

$$R_{j+1}[i] = 1 \quad \text{if } R_j[i-1] = 1 \text{ and } s_i = t_{i+1} \text{ or}$$
$$R_{j+1}[i] = 0 \quad \text{otherwise}$$

If $R_{j+1}[m] = 1$, then return the match.

Figure 6 shows an example of this algorithm, given the search string "process" and the text "post processing". The series of seven diagonal 1's indicate a match.
The approximate string matching algorithm designed by Wu and Manber is an adaptation of the Baeza-Yates and Gonnet algorithm. In this implementation, another binary array, \( R' \), is added. This array indicates all possible matches up to \( t_j \) with at most one insertion, deletion or substitution. Each of these scenarios will be presented separately below; the general problem will be discussed last.

The first case to be examined is insertion. The array \( R'_j[i] = 1 \), if the first \( i \) characters of the search string \( s \) match \( i \) of the last \( i + 1 \) characters up to \( j \) in the text. By maintaining both the \( R \) and the \( R' \) arrays, all exact matches and all matches with at most one insertion will be found. In other words, \( R_j[m] = 1 \) represents an exact match and \( R'_j[m] = 1 \) represents a match with at most one insertion. As was the case for the array \( R \), the transition for \( R' \) must be established. There are two cases for a match with at most one insertion of the first \( i \) characters of the text string \( T \) up to \( t_{j+1} \).

The first case, is an exact match of the first \( i \) characters up to \( t_j \). In this case, inserting \( t_{j+1} \) at the end of the exact match creates a match. In the second case, the insertion is somewhere in the middle of the pattern, or the first \( i-1 \) characters match up to \( t_j \) with one insertion and \( t_{j+1} = s_j \).
In allowing for a single deletion from the search string $s$, the arrays $R$ and $R^1$ are defined as above. As with insertions, there are two cases for a match with at most one deletion. Either all of the first $i-1$ characters of $s$ up to $t_{j+1}$ match (in which case the character $s_i$ is deleted), or the first $i-1$ characters of $s$ up to $t_j$ match with one deletion and $t_{j+1} = s_i$. In this case the deletion occurs in the middle of the string $s$.

The last situation analyzed is substitution. In substitution, one character of $S$ is replaced with one character of $T$. Again, there are two conditions to consider. In the first case, the first $i-1$ characters up to $t_j$ match. Here, substitute $t_{j+1}$ with $s_i$ and match the first $i-1$ characters. In the second case, the substitution occurs in the middle of $T$.

In the general case, that is allowing for substitutions, deletions and insertions instead of one additional array $R^1$, $k$ additional arrays $R^1, R^2, R^3, ..., R^k$ are maintained. Each array $R^d$ stores all possible matches up to $d$ errors. There are four possibilities for obtaining a match of the first $i$ characters with $\leq d$ errors up to $t_{j+1}$. The transitions from array $R^d_j$ to $R^d_{j+1}$ that correspond to the four possibilities represent the following:

1. The first $i-1$ character match with $\leq d$ errors up to $t_j$ and $t_{j+1} = s_i$. This case corresponds to matching $t_{j+1}$.

2. The first $i-1$ character match with $\leq d-1$ errors up to $t_j$. This case corresponds to substituting $t_{j+1}$.

3. The first $i-1$ characters match with $\leq d-1$ errors up to $t_{j+1}$. This case corresponds to deleting $s_i$.
4. The first $i$ characters match with $\leq d-1$ errors up to $t_j$. This case corresponds to inserting $t_{j+1}$.

Figures 7 and 8 [23] show the transitions for the arrays $R$ and $R^1$ respectively, for approximate matching with one insertion. The search string is "aabac" and the text string is "aabaaacaabacab". In Figure 7, the five 1's on the diagonal show that an exact match has been found. Figure 8 shows that a second match is found by substituting an "a" in the text string (column 12) for the "c" in the search string.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>a</th>
<th>c</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 7: Array $R$ for approximate matching with a single insertion

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>a</th>
<th>c</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
<th>c</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 8: Array $R^1$ for approximate matching with a single insertion
As demonstrated in this discussion, agrep is a very powerful string and pattern matching tool. In addition to its flexibility, the designers of agrep have taken great care in ensuring that it performs operations very quickly.

This section has explained how string similarity techniques can be used to perform string matching in an environment where errors are present. In the case of text retrieval systems, however, these techniques are not practical. Text retrieval databases contain large collections of documents with many unique words. It is quite common to find instances where a string containing one error may be within one edit operation of many strings. As a result, a query may return a very large number of documents, many of which may not be relevant to the user's request.
CHAPTER 3

THE SPELLING ERROR DETECTION
AND CORRECTION PROBLEM

The problem of automatically detecting and correcting spelling errors has long received attention from computer scientists. While the original motivation for this work was directed at detecting and correcting errors caused by OCR, and the transmission of Morse Code, error detection and correction today is in widespread use, especially in word processing systems. This section will review some of the techniques that have been proposed to perform automatic spelling error detection and correction.

Spelling error detection and spelling error correction are truly two separate functions. The goal of error detection is to determine whether or not a text string is indeed a word. This is usually done by using a string matching technique to compare the string to a collection of words stored in an on-line dictionary. The goal of error correction is to convert a misspelled term to a correctly spelled term to which it is most similar. Several popular methods exist for doing this.

Like string matching, the methods used to correct spelling errors are broken into two strategies [9]. The first strategy, called absolute, seeks to correct a string strictly from the characteristics of the misspelling. An example of this technique is the string "goood". No word in the English language contains the same character in three consecutive positions, so one of the "o"s can be removed from this string with great
confidence that the resulting string "good" is a valid word. Another absolute method is to store commonly misspelled words in a small dictionary along with the correct spelling of the word. For example, the word "the" might commonly be misspelled as "hte". The string, "hte" is stored in the dictionary along with the correct spelling. When necessary, the dictionary is checked and the proper correction is made if the misspelling appears in the dictionary. Absolute methods of error detection and correction were born out of the need for fast, disk conserving ways to perform spelling error detection and correction. As a result, this method is generally limited in its ability to detect and correct spelling errors; it is only used for very specific applications. However, one absolute technique that is used to perform general spelling error detection and correction is n-grams.

The notion of n-grams was developed out of the need to gain more information than conventional absolute methods could provide and, at the same time reduce the complexities associated with performing spelling error detection and correction using dictionary lookups. An n-gram is a set of $n$ consecutive characters that are extracted from a word and used to identify that word. The integer $n$ can range from 1 to $m$, where $m$ is the length of the word. The two most common n-grams are the digram ($n = 2$) and the trigram ($n = 3$).

Digrams use sequences of character pairs to identify a word; for example, the string $a = a_1, a_2, a_3,...,a_m$ has the set of digrams $a_{1,2}, a_{2,3}, a_{3,4}, ..., a_{m-1,m}$. Trigrams are sequences of three consecutive characters for a given word. Given $a = a_1, a_2, a_3,...,a_m$, the trigram for $a$ is $a_{1,2,3}, a_{2,3,4}, a_{3,4,5}, ..., a_{m-2,m-1,m}$. The string "computer" is represented by the following digram and trigram sets:
digram: -c, co, om, mp, pu, ut, te, er, r-

trigram: -co, com, omp, mpt, pte, ter, er-

The dashes represent the boundaries of the words.

Generally, most work using n-grams has focused on the use of digrams and trigrams. The primary reason is that relatively few of them occur in the English language. There are $28^2$ or 784 possible digrams; this includes blanks and apostrophes along with the 26-letter alphabet. In a study done using a large collection of words, only 500 or 70% of the possible number of digrams were found [4]. Similarly, there are $28^3$ or 21,952 possible trigrams. In the same study, it was found that only 5,488 or 25% of the possible trigrams actually occurred. Therefore, the number of comparisons that need to be made in order to detect an error is much smaller than the number of comparisons required using a dictionary. Another feature of digrams and trigrams is that if an error is found, the location of the error can be easily identified.

An example of a system that was based on the use of trigrams was presented in [24]. This system used trigrams to locate misspelled words, and identify where within the misspelled word the error occurred. The authors deviated from frequency measures generally associated with trigram analysis. Instead of relying on the frequency of occurrence of the trigrams as they would appear in a dictionary, the authors chose to measure the actual error probabilities for each trigram as they appear in the document collection. This way of determining error probability has the implication that any words containing trigrams with significantly high error probabilities are misspelled. The notions of error trigram and valid trigrams are
introduced because a single trigram could result from both a correct spelling and an incorrect spelling. An error trigram is a trigram that is valid, but it appears as the result of a misspelling. A valid trigram is a trigram that exists in a correctly spelled word. If the same trigram is produced from a word that is spelled both correctly and incorrectly, then the trigram is treated as a valid trigram. Therefore, for the word "company" and the misspelling "coopany", the error trigrams are coo, oop and opa; the valid trigrams are -co, com, opm, mpa, pan, any, and ny-. By using the number of occurrences of error trigrams and valid trigrams, the error probability (P) for a given trigram is calculated using the following formula:

\[ P = \frac{E}{E + V} \]

where \( E \) is the number of times a trigram is classified as an error trigram and \( V \) is the number of times it is classified as a valid trigram. A trigram with a probability \( P = 0.40 \) implies that 40 percent of the words that contain that trigram are spelled incorrectly. Trigrams that have not been seen before (those that do not appear in the trigram dictionary) are assumed to be incorrect and are assigned \( P = 1.0 \). The error probability for each trigram is compared to a predetermined threshold \( x \) where \( 0 < x < 1 \). If the error probabilities for two consecutive trigrams within a given string are greater than \( x \), then the word is flagged as a misspelled word. By varying the value of \( x \), the accuracy of the system could be changed. At best, the system was able to identify about 95 percent of the misspellings in the collection. However, the system reported 67 percent of the correctly spelled words as misspelled words.

The successive order of the two incorrect trigrams also serves as the basis for identifying the location of the error within a word. The starting position of the second trigram is returned as the location of the error. For example, given the word
"database" with the trigrams -da, dat, ata, tab, abs, bse and se-, the error trigrams are abs, and bse. The value 6 is returned to denote the position of the error. Using this method, the system was able to locate within two characters the location of the error as much as 96 percent of the time. The system could identify the location within 1 character of where the misspelling occurred 94 percent of the time.

Absolute methods alone have not proven to be very effective in spelling detection and correction. This is generally due to the fact that these methods are not well developed. In the case of n-grams, studies have consistently shown that as the size of a document collection (and the number of unique words in that collection) increase, both the reliability and performance of this technique decrease [14]. For these reasons, absolute techniques are not often used. However, some absolute techniques, such as hashing tables and n-grams, have been used in conjunction with relative strategies because of their ability to identify the location within a word where an error has occurred.

The relative strategy is the most commonly used method of spelling error detection and correction. It has been routinely shown that relative strategies are generally more effective than absolute methods [10] and [11]. Relative strategies involve comparing each word in the text to the entries in a dictionary and changing a misspelled word to a word in the dictionary to which it is most similar. Here, "similar" generally refers to edit distance.

The method of dictionary lookup has long been a successful method of performing automatic spell checking. At the heart of this technique is a machine readable dictionary stored on disk. When a text file is scanned, each word (token) is individually read and compared to each of the words in the dictionary. If a word in
the dictionary matches the token, then the next token is read. If no word in the
dictionary matches the token, then one of two things occurs. Either the misspelled
word is identified for the user (and a list of correct words is presented so that the user
can select the correct word), or the system will choose the word in the dictionary
which is most similar to the token and perform the replacement automatically. The
latter is seldom done because more than one dictionary entry may be equally similar to
the token, and the replacement may result in an incorrect dictionary term being
chosen.

The effectiveness of this method is based largely on how big or small the
dictionary is. Most dictionaries used for this purpose generally contain 20,000 to
50,000 distinct terms. Any dictionary that contains fewer words will report a large
number of misspellings. A dictionary containing more than 50,000 terms is very
accurate; however, it becomes very hard to maintain and results in slow searches[9].

One strategy that is used to overcome both size and speed problems while
maintaining a high degree of accuracy, is the two-level search strategy[16]. The two-
level search strategy is based on dividing the large dictionary into two smaller
dictionaries, along with a large main dictionary. The smallest dictionary (see Figure
9) contains 100 - 200 of the most commonly used terms in the English language. It
was found in the Brown Corpus studies [9] that only 134 words comprise over 50
percent of the words used in everyday English, so one expects that the majority of
tokens occurring in text will be found in this dictionary. The medium size dictionary
contains 1,000 - 2,000 document specific words. These words appear often in a
document, but do not fall into the category of frequently used English words. The
largest dictionary is between 10,000 and 100,000 words and contains words that do
not fall into either of the categories already defined. Research from the Brown Corpus project indicates that only 5 percent of all words that appear in a document fall into this category. The dictionaries would be used in the following manner: given a text file, each token is read individually. The token is first compared with the smallest dictionary. If the token appears in that list, then the next token is read; otherwise, the token is compared to the middle dictionary. If the token appears there, then the next token is scanned; otherwise, the token is compared to the last dictionary. If the token appears there, then the next token is read; if it is not in the largest dictionary, then it is reported to the user as a possible misspelling.

![Figure 9: Dictionaries for two-level search strategy](image)

Even using methods such as the two-level search, dictionary lookup is very costly both in search time and disk space. To counteract these problems, systems have been developed that use a combination of both the absolute and the relative methods. By basing a system on both philosophies, one can get both accuracy of a dictionary system in locating misspelled words, and the ability of techniques such as n-grams for locating where in a word the error occurred to assist in making correct substitutions.
One system that uses a combination of both techniques is SPEEDCOP [11]. SPEEDCOP is an automatic spelling error detection and correction program that makes extensive use of hashing tables (an absolute method) and a dictionary lookup. The purpose of SPEEDCOP is to automatically detect and correct misspelled words that contain a single error (edit distance of 1), and whose correct form is in the dictionary.

At the heart of SPEEDCOP is a similarity key called the skeleton key. This key is formed by extracting the first character of a term and then concatenating each unique consonant in the order that they occur in the string. At the end of this key, each unique vowel is concatenated in the order in which they occur in the word. Table 1 shows an example of some strings and their associated skeleton keys.

<table>
<thead>
<tr>
<th>string</th>
<th>skeleton key</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>dtbsae</td>
</tr>
<tr>
<td>correction</td>
<td>crtnoei</td>
</tr>
<tr>
<td>automatic</td>
<td>tmcauoi</td>
</tr>
</tbody>
</table>

Table 1: Example of text strings and associated skeleton keys

The authors cite several reasons for employing such a scheme. Similarity keys reduce the scatter that is associated with original strings. Therefore, the collating distance between two keys can be used as a similarity measure between the two original strings. Additionally, this scheme preserves the identity of the string because all of the characters in the original string also occur in the key. Preliminary work demonstrated that the skeleton key was very sensitive to incorrect consonants occurring near the beginning of a word. In other words, if a consonant near the
beginning of the word was incorrect or omitted, the system did not reliably detect spelling errors. To correct this, a key called an *omission key* was introduced. The omission key is based on information gathered by the authors indicating that consonants are omitted from words with a distinct frequency. They found that "r" was omitted more often than any other consonant and that "j" was the least omitted consonant. The complete list of consonants in decreasing order of omission represents the following: "rstnlchdpgbymfbywvzqkj". The omission key is built in the following manner: each of the unique consonants in a string is concatenated and sorted in increasing order of frequency. The unique vowels occurring in the string are then concatenated to the end of the key in the order in which they appear. Table 2 lists the strings from Table 1 with their associated omission keys.

<table>
<thead>
<tr>
<th>string</th>
<th>omission key</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>bdtsae</td>
</tr>
<tr>
<td>correction</td>
<td>cntroei</td>
</tr>
<tr>
<td>automatic</td>
<td>mctauoi</td>
</tr>
</tbody>
</table>

Table 2: Example of text strings and associated omission keys

*SPEEDCOP* uses a 40,000 term dictionary in which all words are represented by their skeleton and omission keys. Each of the keys in the dictionary is built in the same manner as already described. Error detection is done by comparing the keys of the text strings with the keys of the dictionary terms. If a match is found, the next string is read. If no match is found, however, a list of dictionary keys that alphabetically collate closest to the incorrect key is built. Once this list is built (starting from the middle of the list and alternating above and below), each list key is compared to the incorrect key. The key in the list that is most similar to the incorrect
key is chosen as the correct key, and the misspelled word is replaced by the word corresponding to the correct key. If two or more keys in the dictionary are equally similar to the incorrect key, then the correct key whose corresponding word appears most often in the text is chosen.

To supplement this scheme, the system also included a misspelling dictionary. While misspelling dictionaries are generally ineffective, the system used this list for fast lookups of terms that had been identified as "commonly" misspelled. This dictionary only contained 256 entries. The words chosen for this dictionary were unique in that every time the misspelling was encountered in the collection, it was always the same misspelling. For example, the only misspelling found in the collection for the string "the" might be "hte".

As with most automatic spelling error correction systems, SPEEDCOP sought to correct only single errors caused by omission, deletion and substitution. Overall, the system was able to detect and correct 71 percent of the misspellings that it encountered. The drawback to SPEEDCOP is that when building the skeleton and omission keys, it is assumed that the first character of the word is correct. If the first character is not correct, the system will fail to accurately identify and change misspellings.

The spelling error detection and correction schemes discussed in this chapter are designed to be used primarily on text that is manually typed. By themselves these techniques are not sufficient for automatic spelling detection and correction of OCR generated text. There are two reasons for this. First, these techniques, generally, perform correction at the word level. In other words, spelling correction is done
based on the entries in the dictionary without regard for other words that may appear in a particular document or in the document collection. As a result, ambiguities cannot be solved automatically. The second reason is that they do not take advantage of information pertinent to OCR generated errors. Errors that are caused by manual entry of text are generally random [9], while a large percentage of errors generated by OCR devices are uniform. Without considering these issues, automatic detection and correction of OCR generated errors is not effective.
CHAPTER 4

OCR POST PROCESSING SYSTEM

The previous chapter discussed techniques for performing spelling error detection and correction on manually typed text. As noted at the end of that chapter, these techniques do not work well for OCR generated text. More information must be incorporated into the correction scheme. Information such as the number of times a term appears in a document or the document collection, as well as the most common errors caused by the OCR device must be considered. This section will describe a post processing system that makes use of such information in order to automatically detect and correct OCR caused spelling errors.

The post processing of OCR'ed text is not new. Two of the earliest attempts to post process OCR'ed text are presented in [13] and [14]. Many reasons can generally be cited for the inadequacies of these systems. The main reason, however, is that these systems did not incorporate enough information into the correction scheme. The system presented in [14] was one of the first attempts to automatically correct multifont, unformatted OCR'ed text. This system was designed to be integrated into the OCR architecture. The system represented terms as vectors and used a dictionary look-up scheme to perform error detection and correction. (Both the OCR'ed terms and the terms in the dictionary were represented as vectors.) If a misspelled word was found, a list of similar, correctly spelled words, was created. The correct word was chosen using Bayes maximum likelihood solutions. The system was able to correctly identify and change 97 percent of the misspellings it
encountered. However, the authors did not divulge the size of the document collection used to perform their experiments. The system presented in [13] used binary n-grams, a modified dictionary lookup and a confusion matrix. This system was very effective at detecting as well as correcting spelling errors. With the use of the confusion matrix, the system was also capable of correcting misspelled words which contained more than one error. However, the tests performed using this system were limited to only 2,755, six-character words.

Our post processing system makes use of some of the methods employed in [13], but one major difference separates this system from any other. The system explained here works in conjunction with an inverted file system. The use of an inverted file system allows the post processor to take advantage of the relationships that exist between words throughout a document or document collection. The rationale behind this method is that for every misspelled word in the database, there exist other correctly spelled occurrences of the same word in the database.

System Implementation

For this study, approximately 10,000 pages from the Licensing Support System (LSS) prototype database[7] were OCR'ed. The selected pages contained a combination of text, pictures and graphics. All of the pages were automatically zoned and processed. No manual correction was done on the OCR'ed documents. The ASCII text was then loaded into an inverted file text retrieval system. After the words in the documents were indexed, the index was dumped into two ASCII files. One of the files, called "centroids," contained words that were spelled correctly and occurred often in the collection. The dictionary, ispell [22], was used to perform the
spell checking. The other file, called "misspelled," contained words that were not in the dictionary or did not occur often in the collection. Along with each of the terms from the index, the number of occurrences corresponding to each term was also written to the files. At this point, phase two of the post processing system began.

The first step of phase two was to cluster each term in the misspelled file with the terms in the centroid file. The clustering was done using agrep to compare each term in the misspelled file with the terms in the centroid file. If the edit distance between a centroid term and a misspelled term was equal to one, they were clustered. Table 3 shows the results of the clustering process for Experiment 1. For this and the following experiments, the clusters were broken down by word length. This was done for two reasons. First, manual verification was easier. Second, it was easier to determine the effectiveness of the system at each word length. The second column in Table 3 shows the total number of centroids that were found in the collection at each character length. The last row of this column indicates the total number of correctly spelled words of length 7 through 18. Column 3 shows the number of centroids that had at least one misspelled term clustered with them. The last column provides the number of centroids that did not have any misspellings clustered with them.

In this first experiment, the correction (step two of phase two) of misspelled terms was done by changing all of the clustered terms to match the centroid term they were clustered with. Table 4 provides a summary of this step. Column 2 shows the cumulative number of clustered misspelled words broken down by word length. Column 3 shows the total number of occurrences of the misspelled words. Because a misspelled term may have appeared in the collection more than one time, the number

---

2 The results shown here and throughout this chapter have also been published in [17] and [18].
of times any misspelling occurred in the collection was maintained. Column 4 represents the number of misspellings that were incorrectly changed to the same spelling as the centroid. The last column shows the total number of occurrences of the words that were incorrectly changed. The last row indicates that there were 8,036 misspellings identified, resulting in 16,367 occurrences. Of the 16,367 occurrences of misspelled terms that were changed, 3,684 or 23 percent were incorrectly changed.

| word length | centroids | non-empty centroids | empty centroids |
|-------------|-----------|---------------------|-----------------
| 18          | 1         | 0                   | 1               |
| 17          | 5         | 2                   | 3               |
| 16          | 8         | 1                   | 7               |
| 15          | 23        | 2                   | 21              |
| 14          | 75        | 23                  | 52              |
| 13          | 200       | 62                  | 138             |
| 12          | 339       | 120                 | 219             |
| 11          | 639       | 223                 | 416             |
| 10          | 946       | 372                 | 574             |
| 9           | 1339      | 551                 | 788             |
| 8           | 1574      | 753                 | 821             |
| 7           | 1766      | 1060                | 706             |

Table 3: Experiment 1 cumulative centroid report

By examining the results from Experiment 1 we discovered three problems with the system. First, there were a number of correctly spelled terms treated as misspelled words because they were not in the dictionary. For example, the word "preconstruction" was considered a misspelling and was incorrectly clustered with "reconstruction". The second problem was that some misspelled words were clustered with more than one centroid. For example, the term "location" was
clustered with "allocation" and "location". The last problem was a result of the assumption that terms that appeared infrequently in the collection are misspelled. In fact, many words that had a low frequency of occurrence were spelled correctly.

<table>
<thead>
<tr>
<th>word length</th>
<th>misspelled terms</th>
<th>occurrences</th>
<th>erroneously-corrected terms</th>
<th>erroneously-corrected occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>9</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>57</td>
<td>105</td>
<td>3</td>
<td>26</td>
</tr>
<tr>
<td>13</td>
<td>183</td>
<td>257</td>
<td>7</td>
<td>32</td>
</tr>
<tr>
<td>12</td>
<td>435</td>
<td>677</td>
<td>19</td>
<td>87</td>
</tr>
<tr>
<td>11</td>
<td>959</td>
<td>1461</td>
<td>62</td>
<td>155</td>
</tr>
<tr>
<td>10</td>
<td>1796</td>
<td>2629</td>
<td>123</td>
<td>321</td>
</tr>
<tr>
<td>9</td>
<td>3124</td>
<td>4933</td>
<td>251</td>
<td>708</td>
</tr>
<tr>
<td>8</td>
<td>5134</td>
<td>9050</td>
<td>460</td>
<td>1524</td>
</tr>
<tr>
<td>7</td>
<td>8036</td>
<td>16367</td>
<td>995</td>
<td>3684</td>
</tr>
</tbody>
</table>

Table 4: Experiment 1 cumulative error report

Experiment 2 included enhancements to the system to compensate for the problems found in Experiment 1. For this experiment the ispell dictionary was enhanced with a local dictionary to help identify more correctly spelled words. The local dictionary contained about 96,000 terms that were specific to this document collection. All of the terms that were originally in the misspelled file were compared to the enhanced dictionary. Terms that were now in the dictionary were added to the

\[\text{The size of the dictionary is misleading since many of the terms in the local dictionary were also in the ispell dictionary.}\]
centroid file. The clustering process was then rerun on the new centroid and misspelled files. Table 5 shows that the changes that were made increased the number of centroids that were found, especially for terms with more than ten characters.

<table>
<thead>
<tr>
<th>word length</th>
<th>centroids</th>
<th>non-empty centroids</th>
<th>empty centroids</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>86</td>
<td>0</td>
<td>86</td>
</tr>
<tr>
<td>17</td>
<td>103</td>
<td>1</td>
<td>102</td>
</tr>
<tr>
<td>16</td>
<td>185</td>
<td>1</td>
<td>184</td>
</tr>
<tr>
<td>15</td>
<td>262</td>
<td>3</td>
<td>259</td>
</tr>
<tr>
<td>14</td>
<td>340</td>
<td>24</td>
<td>316</td>
</tr>
<tr>
<td>13</td>
<td>568</td>
<td>58</td>
<td>510</td>
</tr>
<tr>
<td>12</td>
<td>777</td>
<td>107</td>
<td>670</td>
</tr>
<tr>
<td>11</td>
<td>1117</td>
<td>209</td>
<td>908</td>
</tr>
<tr>
<td>10</td>
<td>1474</td>
<td>344</td>
<td>1130</td>
</tr>
<tr>
<td>9</td>
<td>1873</td>
<td>509</td>
<td>1364</td>
</tr>
<tr>
<td>8</td>
<td>1986</td>
<td>693</td>
<td>1293</td>
</tr>
<tr>
<td>7</td>
<td>2084</td>
<td>960</td>
<td>1124</td>
</tr>
</tbody>
</table>

Table 5: Experiment 2 cumulative centroid report

In Experiment 2, correction of the clustered terms was done the same as in Experiment 1. Table 6 shows the results of the correction phase. It can be determined from the last row of this table that the percentage of incorrectly changed terms (penalties) decreased from 23 percent in Experiment 1 to 17 percent in Experiment 2.

The results gathered from Experiment 2 showed that two of the three problems encountered in experiment 1 had been rectified. However, the problem of
multiple clustered misspelled terms was still present after Experiment 2. The methods proposed for Experiment 3 were developed to work together to address this problem. In Experiment 3 two strategies called, local-info and the confusion matrix, were introduced. These two strategies were invoked at the time the correction process took place.

<table>
<thead>
<tr>
<th>word length</th>
<th>misspelled terms</th>
<th>occurrences</th>
<th>erroneously-corrected terms</th>
<th>erroneously-corrected occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>12</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>9</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>56</td>
<td>68</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>174</td>
<td>213</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>409</td>
<td>548</td>
<td>20</td>
<td>34</td>
</tr>
<tr>
<td>11</td>
<td>905</td>
<td>1267</td>
<td>54</td>
<td>75</td>
</tr>
<tr>
<td>10</td>
<td>1694</td>
<td>2282</td>
<td>111</td>
<td>149</td>
</tr>
<tr>
<td>9</td>
<td>2928</td>
<td>4379</td>
<td>245</td>
<td>508</td>
</tr>
<tr>
<td>8</td>
<td>4767</td>
<td>7724</td>
<td>458</td>
<td>1078</td>
</tr>
<tr>
<td>7</td>
<td>7311</td>
<td>13571</td>
<td>828</td>
<td>2392</td>
</tr>
</tbody>
</table>

Table 6: Experiment 2 cumulative error report

The first method, local-info, made extensive use of the inverted file storage system to determine which of the possible centroids the misspelled word should be changed to. The first step in this method was to locate which document in the inverted file system the misspelling occurred. For that document, the frequency count of the corresponding centroids was then retrieved. The misspelled word was changed to the centroid having the highest frequency in that document. For example, the word "downwar" was clustered with the centroids "downwarp" and "downward". First, the
database is queried to determine in which document the term "downwar" appeared. Subsequent queries of the database show that the word "downwarp" did not appear in that document, but the word "downward" appeared 18 times. Therefore, the term "downwar" was changed to "downward". This method was sometimes not able to identify the correct centroid because more than one of the possible centroids appeared in the document an equal number of times. To resolves these ambiguities, a confusion matrix was used.

The confusion matrix is a table with information pertaining to errors caused by the OCR device. Table 7 shows a partial list of the confusion matrix used for these studies.

<table>
<thead>
<tr>
<th>number of errors</th>
<th>correct</th>
<th>generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>137</td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>109</td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>48</td>
<td>e</td>
<td>c</td>
</tr>
<tr>
<td>41</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>r</td>
<td>t</td>
</tr>
<tr>
<td>25</td>
<td>c</td>
<td>e</td>
</tr>
<tr>
<td>24</td>
<td>e</td>
<td>a</td>
</tr>
<tr>
<td>21</td>
<td>i</td>
<td>t</td>
</tr>
<tr>
<td>18 *</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>17</td>
<td>l</td>
<td>i</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>Z</td>
</tr>
<tr>
<td>16</td>
<td>t</td>
<td>r</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>3</td>
<td>ε</td>
<td>n</td>
</tr>
<tr>
<td>2</td>
<td>t</td>
<td>ε</td>
</tr>
</tbody>
</table>

Table 7: Partial listing of confusion matrix
The purpose of this matrix is to keep a count of substitution, deletion and
insertion errors caused by the OCR device. Since many errors that are generated by
OCR devices are uniform, this information can be confidently used to reverse errors.
The confusion matrix used for this study was built during the global editing phase. As
misspelled terms were corrected, the edit operation and the characters in question
were recorded. The count for each of these substitutions was maintained. For
example, Table 7 shows that the arbitrary substitution of "e" for "a" occurred 24 times
during the global editing stage. The omission of "t" occurred twice and the insertion
of "n" occurred three times. The following demonstrates how the confusion matrix
was used. The term "transporation" is clustered with the centroids "transpiration" and
"transportation". Since both of the centroids appear in the same document as the
misspelled word an equal number of times, the local-info is unable to make a decision.
Using the confusion matrix however, it was determined that in all of the previous
corrections "i" was never substituted for "o", but "t" had been omitted. Therefore,
the term "transporation" was changed to "transportation". Table 8 shows examples of
some terms that were clustered with two centroids and the method, either local-info
or the confusion matrix, that was used to determine the correct spelling of the
misspelled word.

Table 9 shows a comparison of the results obtained from Experiments 2 and 3.
The last two columns show that both the number of erroneously-corrected terms and
the number of occurrences of those terms decreased from Experiment 2. In fact, by
using the local-info and the confusion matrix, the accuracy of the system improved by
four percent. A large number of the words whose spelling was incorrectly changed
during Experiment 3 fell into two categories; either the word was hyphenated at the
end of a line or the word was a proper name. In the first case the inverted file system failed to recognize this condition and treated both ends of the hyphenated word as two words. In the second case, since very few proper names appeared in the dictionary, the system had no way of determining whether or not proper names were spelled correctly.

<table>
<thead>
<tr>
<th>misspelled word</th>
<th>cluster with</th>
<th>result of local-info</th>
<th>result of confusion matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>variation</td>
<td>aviation</td>
<td>variation</td>
<td></td>
</tr>
<tr>
<td>downwar</td>
<td>downwarp</td>
<td>downward</td>
<td></td>
</tr>
<tr>
<td>mountain</td>
<td>fountain</td>
<td>mountain</td>
<td></td>
</tr>
<tr>
<td>location</td>
<td>allocation</td>
<td>location</td>
<td></td>
</tr>
<tr>
<td>construction</td>
<td>constructional</td>
<td>construction</td>
<td></td>
</tr>
<tr>
<td>transporation</td>
<td>transpiration</td>
<td>transportation</td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Results of local-info and confusion matrix centroid selection

Table 10 reports the cumulative percentage of correct changes for each experiment. As seen from this table, there was a marked improvement in correction accuracy at each experiment level. Figure 10 [17] shows a diagram of the post processing system in the configuration that was used for Experiment 3.
<table>
<thead>
<tr>
<th>word length</th>
<th>misspelled terms</th>
<th>erroneous terms Experiment 2</th>
<th>occurrences</th>
<th>erroneous terms Experiment 3</th>
<th>occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>56</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>174</td>
<td>9</td>
<td>10</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>409</td>
<td>20</td>
<td>34</td>
<td>10</td>
<td>22</td>
</tr>
<tr>
<td>11</td>
<td>905</td>
<td>54</td>
<td>75</td>
<td>31</td>
<td>48</td>
</tr>
<tr>
<td>10</td>
<td>1694</td>
<td>111</td>
<td>149</td>
<td>65</td>
<td>93</td>
</tr>
<tr>
<td>9</td>
<td>2928</td>
<td>245</td>
<td>508</td>
<td>148</td>
<td>288</td>
</tr>
<tr>
<td>8</td>
<td>4767</td>
<td>458</td>
<td>1078</td>
<td>276</td>
<td>632</td>
</tr>
<tr>
<td>7</td>
<td>7311</td>
<td>828</td>
<td>2392</td>
<td>575</td>
<td>1827</td>
</tr>
</tbody>
</table>

Table 9: Cumulative error reports for Experiments 2 and 3

<table>
<thead>
<tr>
<th>word length</th>
<th>experiment 1</th>
<th>experiment 2</th>
<th>experiment 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>17</td>
<td>50%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>16</td>
<td>87%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>15</td>
<td>90%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>14</td>
<td>75%</td>
<td>91%</td>
<td>99%</td>
</tr>
<tr>
<td>13</td>
<td>87%</td>
<td>95%</td>
<td>99%</td>
</tr>
<tr>
<td>12</td>
<td>87%</td>
<td>93%</td>
<td>96%</td>
</tr>
<tr>
<td>11</td>
<td>89%</td>
<td>94%</td>
<td>96%</td>
</tr>
<tr>
<td>10</td>
<td>87%</td>
<td>93%</td>
<td>96%</td>
</tr>
<tr>
<td>9</td>
<td>85%</td>
<td>88%</td>
<td>93%</td>
</tr>
<tr>
<td>8</td>
<td>83%</td>
<td>86%</td>
<td>92%</td>
</tr>
<tr>
<td>7</td>
<td>77%</td>
<td>83%</td>
<td>87%</td>
</tr>
</tbody>
</table>

Table 10: Cumulative percentage of correct changes
Figure 10: Global correction post processing system
Future Considerations

At present, the system is only processing words that are seven or more characters in length. However, the success of the system so far and the lessons that have been learned, should be used to plan future enhancements to the system. One of the enhancements already in progress should allow the system to process words that are less than seven characters. For the shorter word lengths, several factors such as proper names and acronyms must be addressed. Since these types of words do not normally appear in a dictionary, other methods must be used to identify them.

In initial experiments, words were considered similar if they were one edit distance away from each other. Procedures that will allow the system to identify similar words with edit distance greater than one should be considered. Through the use of the confusion matrix, the system should attempt to rebuild a word by reversing exactly one of the edit operations in the confusion matrix. For example, the term "IVevada" could be changed to "Nevada" by recognizing that the substitution "IV" for "N" appears in the confusion matrix.

Another enhancement to consider is changing the way that a misspelling and multiple centroids are processed. Currently for a misspelling \( m \) and the centroids \( c_1, c_2, c_3 \ldots c_n \), the clusterings are \((m, c_1), (m, c_2), \ldots (m, c_n)\). Using this method, each (misspelling, centroid) pair is processed by the system without regard for other such pairs. An improvement over this method would be to consider the misspelling and all of the possible centroids. Given the misspelling \( m \) and the centroids \( c_1, c_2, c_3 \ldots c_n \),
the new clustering will be \((m, c_1, c_2, c_3 \ldots c_n)\). This method of clustering should better facilitate the selection of an appropriate change to the misspelling.

If all automatic processing is complete and there are misspellings that still cannot be changed by the system, then an interactive system can be evoked. The misspelling and possible correct words (determined by the system) would be displayed to the user, along with the text of the page where the misspelling occurs. The user could then view the document page and select the correct word. Figure 11 [17] shows a diagram of a post processing system with the above enhancements incorporated in it.
Figure 11: Post processing system with planned enhancements
CHAPTER 5

CONCLUSION

The post processing system described in this paper is a flexible, yet reliable system for correcting OCR errors. While the document set used to perform early studies of the system was relatively small, initial results have so far been encouraging. If these results are any indication, it is possible to automatically correct spelling errors caused by OCR. While 100 percent spelling error detection and correction may not be a realistic or obtainable goal, each step toward that mark could potentially save thousands of dollars in manual correction costs. As OCR becomes more popular, and the need for accurate data becomes more critical, post processing systems such as the one described in this paper will become very valuable.
Bibliography


