MEASURING THE ACCURACY

OF PAGE-READING SYSTEMS

by

Stephen V. Rice

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

Doctor of Philosophy

in

Computer Science

Department of Computer Science
University of Nevada, Las Vegas
December 1996

The Dissertation of Stephen V. Rice for the degree of Doctor of Philosophy in Computer Science is approved.

_____

Chairperson, Thomas A. Nartker, Ph.D.

_____

Examining Committee Member, Laxmi P. Gewali, Ph.D.

_____

Examining Committee Member, Lawrence L. Larmore, Ph.D.

_____

Examining Committee Member, John T. Minor, Ph.D.

_____

Graduate Faculty Representative, Shahram Latifi, Ph.D.

_____

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

University of Nevada, Las Vegas
December 1996

ABSTRACT

Given a bitmapped image of a page from any document, a page-reading system identifies the characters on the page and stores them in a text file. This “OCR-generated” text is represented by a string and compared with the correct string to determine the accuracy of this process. The string editing problem is applied to find an optimal correspondence of these strings using an appropriate cost function. The ISRI annual test of page-reading systems utilizes the following performance measures, which are defined in terms of this correspondence and the string edit distance: character accuracy, throughput, accuracy by character class, marked character efficiency, word accuracy, non-stopword accuracy, and phrase accuracy. It is shown that the universe of cost functions is divided into equivalence classes, and the cost functions related to the longest common subsequence (LCS) are identified. The computation of a LCS can be made faster by a linear-time preprocessing step.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

Funding for this project was provided by the U.S. Department of Energy, the U.S. Department of Defense, and the following developers of page-reading systems: Adobe, Caere, Calera, Hewlett Packard, Maxsoft-Ocron, Recognita, and Xerox. This project has been enriched by input from many people, including Dave Abercrombie, Carl Alsing, Mindy Bokser, Roger Bradford, Mitch Buchman, Richard Casey, Barbara Cerny, Phil Cheatle, Dan Graser, Donna Harman, Frank Jenkins, Shahram Latifi, Michael O'Donnell, Ashok Singh, Kazem Taghva, Zoltan Urban, Ben Wittner, and especially, Horst Bunke and Junichi Kanai. My "co-advisors," George Nagy and Tom Nartker, were a source of valuable advice and encouragement throughout this project. My greatest motivation has been my family — my dear wife, Sheila, and our wonderful children, Christopher, Jessica, Stephanie, and Curt.

CHAPTER 1

INTRODUCTION

A page-reading system, or "page reader," is a computer system for converting hard-copy documents to electronic form. Given a bitmapped image of a page from any document, the system will identify the machine-printed characters on the page and store them in a text file, which may be edited or searched. A page reader is a type of optical character recognition (OCR) system; other types include form readers and postal address readers.

The first commercial page-reading system appeared in 1959 and could recognize only one font (Bokser, 1992). Today, many "omnifont" systems are available that can identify characters in English and most European languages. Also, systems are now appearing that can handle complex scripts such as Arabic or Chinese.

Tests of Page-Reading Systems

Unlike a word processor or spreadsheet program, the behavior of an OCR system is complex and unpredictable. Like other pattern recognition systems, an OCR system is "trained" using a set of data. Its performance when processing other data is not known a priori, and must be measured empirically.

Testing is essential for technological improvement in OCR. Traditionally, OCR systems have been tested on isolated characters (Kanai, Nartker, Rice, & Nagy, 1993). For each character of the test set, the OCR system is given a bitmapped image containing only that character and asked to identify it. The system may "reject" the character, i.e., choose not to identify it. The percentage of characters that are rejected is termed the *reject rate*. For characters that are not rejected, the percentage that are identified incorrectly is termed the *error rate*. Typically, the error rate for various reject rates is plotted in an *error-reject curve* (Chow, 1994).

The input to a page-reading system is an image of an entire page of text. The user may supply the image coordinates of blocks of text (called *zones*), or may ask the system to locate the blocks by a process known as

automatic zoning. The system isolates the characters in a block by first finding lines, and then dividing each line into words and characters. Thus, a page reader must locate and identify the characters on a page.

The text file produced by a page-reading system can be represented as a single string of characters by concatenating its lines. To evaluate the accuracy of the system, this string of *OCR-generated* text can be compared with the string of correct text, or *ground-truth*, for this page. This dissertation focuses on how these strings can be matched to derive measures of performance.

Conducting a meaningful test of page-reading systems is a complex task. In addition to the string-matching problem, careful consideration must be given to the selection, preparation, and representation of test data (see Nagy, 1995; Rice, Kanai, & Nartker, 1993b; Phillips, Chen, Ha, & Haralick, 1993; and RAF Technology, Inc., 1995). A large number of pages must be processed to obtain statistically significant results. Thus, software tools are needed to automate the test. Through automation, human error and bias are avoided, and tests are reproducible.

Reviews of page-reading systems in computer magazines often

include the results of a simple test. Table 1 lists some examples of these, indicating the number of systems tested, the number of pages on which the systems were tested, and the performance measures that were reported. Unfortunately, these tests were conducted on too few pages for the results to be meaningful. Accuracy is reported in terms of the number of misrecognized characters or words, but the error-counting method is not precisely specified. Of the tests listed in Table 1, only the *Byte* article indicates that an automated tool was used to compute accuracy. Presumably, errors were counted manually in the other tests, which may explain the smaller number of test pages.

Since 1992, the Information Science Research Institute (ISRI) at the University of Nevada, Las Vegas, has conducted an annual test of page-reading systems. The goal of the test is to provide useful information regarding the performance of these systems to users and vendors of this technology, and OCR researchers. As shown in Table 2, the test has expanded each year to include more data and more measures of performance. Programs to compute these measures are part of a suite of software tools called the OCR Experimental Environment (Rice, 1993).

Table 1 Evaluations of Page-Reading Systems Appearing in Computer Magazines

	Systems	Pages	Performance Measures
Byte (Diehl & Eglowstein, 1991)	14	49	word accuracy and throughput
MacWorld (McClelland, 1991)	10	5	character accuracy and speed
PC Magazine (Jones, 1992)	6	?	word accuracy and throughput
MacUser (Welch, 1993)	8	19	character accuracy
SunWorld (Raucci, 1993)	3	4	word accuracy and speed
Imaging Magazine (Gilder & Neilson, 1993)	4	4	character accuracy and speed

Table 2 The ISRI Annual Test of Page-Reading Systems

	Systems	Pages	Performance Measures
First annual test (Rice, Kanai, & Nartker, 1992)	6	132	character accuracy
Second annual test (Rice, Kanai, & Nartker, 1993a; Kanai, Rice, & Nartker, 1993; Nartker, Rice, & Kanai, 1994)	8	460	character accuracy, marked character efficiency, word accuracy, non-stopword accuracy, and automatic zoning
Third annual test (Rice, Kanai, & Nartker, 1994; Nartker & Rice, 1994)	6	660	character accuracy, marked character efficiency, word accuracy, non-stopword accuracy, phrase accuracy, and automatic zoning
Fourth annual test (Rice, Jenkins, & Nartker, 1995; Nartker, Rice, & Jenkins, 1995)	8	1529	character accuracy, throughput, accuracy by character class, marked character efficiency, word accuracy, non-stopword accuracy, phrase accuracy, and automatic zoning

Definitions and algorithms for these measures are presented in this dissertation, with the exception of the automatic zoning measure which has been documented by Kanai, Rice, Nartker, and Nagy (1995).

Few other tests of page-reading systems have been reported. Some tests were conducted to measure the effectiveness of combining page readers in a voting system (Handley & Hickey, 1991; Bradford & Nartker, 1991; Rice, Kanai, & Nartker, 1992, 1994). Chen, Subramaniam, Haralick, and Phillips (1994) evaluated two page-reading systems using the 1,000-page University of Washington database. Kanai, Liu, Rice, and Nartker (1994) tested three Chinese page readers.

This dissertation presents a methodology for the evaluation of page-reading systems. The string editing problem is introduced in the next section, and is the cornerstone of this methodology. In Chapter 2, character accuracy and related measures are presented. In Chapter 3, a theoretical result is derived that provides new insight into the string editing problem. This result is used in the following chapter on word accuracy. A summary of the contributions of this dissertation appears in Chapter 5.

The String Editing Problem

We wish to find a correspondence between the characters of the OCR-generated string and the correct string. Mistakes made by the page-reading system can result in missing or extraneous characters in the generated string. Hence, it cannot be assumed that the ith character of the generated string corresponds to the ith character of the correct string. However, a well-known method for aligning two strings can be used. It is possible to find a correspondence that identifies a sequence of edit operations for transforming one string into the other with the minimum cost. Finding this "optimal" correspondence is known as the "string editing problem" (or "string-to-string correction problem").

Let $A = a_1 a_2 \ldots a_m$ be a string of m symbols from an alphabet Σ. In the string editing problem, three types of edit operations can be applied to A:

1. *insertion* – any symbol $x \in \Sigma$ can be inserted before a_1, after a_m, or between a_i and a_{i+1} $(1 \leq i < m)$;
2. *deletion* – the symbol a_i can be deleted $(1 \leq i \leq m)$;
3. *substitution* – the symbol a_i can be replaced by any symbol $x \neq a_i$

$(1 \leq i \leq m)$.

A *cost function* $\gamma = (\gamma_I, \gamma_D, \gamma_S)$ specifies the cost of each insertion, deletion, and substitution, respectively, where γ_I, γ_D, and γ_S are non-negative real numbers.

Levenshtein (1966) was the first to express the similarity of two strings in terms of the minimum number of edit operations needed to transform one string into the other. Wagner and Fischer (1974) generalized this idea by assigning a cost to each edit operation, and presented an algorithm based on dynamic programming for finding a minimum cost sequence of edit operations for transforming one string into the other. This algorithm is considered the standard solution for this problem, and takes *O(mn)* time and space, where *m* and *n* are the lengths of the strings. Similar algorithms were discovered earlier in the areas of speech recognition (Vintsyuk, 1968) and molecular biology (Needleman & Wunsch, 1970). Even earlier, Damerau (1964) presented an algorithm that determines whether two strings differ by one edit operation.

The minimum cost of transforming one string into another is equal to the minimum number of edit operations when the cost of each operation

is one, i.e., the cost function is (1,1,1). For some applications, it is desirable to exclude substitutions from consideration and find the minimum number of insertions and deletions needed to transform the string. This can be achieved by using the cost function (1,1,2), because a substitution can be performed by one insertion and one deletion.

If a string *A* can be transformed into a string *C* by performing zero or more deletions, then *C* is a *subsequence* of *A*. If *C* is a subsequence of *A* and a subsequence of string *B*, then *C* is a *common subsequence* of *A* and *B*. If *C* is a common subsequence of *A* and *B*, and there does not exist a common subsequence of *A* and *B* having more symbols than *C*, then *C* is a *longest common subsequence* (*LCS*) of *A* and *B*. Let $L_{A, B}$ denote the number of symbols in a LCS of *A* and *B*. If $A = yxzyyx$ and $B = yyxx$, then the string *yzyx* is a subsequence of *A*, and the string *xx* is a common subsequence of *A* and *B*. The strings *yxx* and *yyx* are longest common subsequences of *A* and *B*, and $L_{A, B} = 3$. Wagner and Fischer (1974) have shown that finding a LCS of two strings is equivalent to finding an optimal correspondence using the cost function (1,1,2). Thus, the LCS problem is a special case of the string editing problem.

Let $d_{A, B, \gamma}$ denote the minimum cost of transforming string *A* into string *B* using cost function γ. When γ is (1,1,1) or (1,1,2), $d_{A, B, \gamma}$ is called the *Levenshtein distance* between *A* and *B*. This distance function is termed the *Levenshtein metric* because it satisfies the properties of a metric:

1. $d_{A, B, \gamma} > 0$ if $A \neq B$; $d_{A, A, \gamma} = 0$;
2. $d_{A, B, \gamma} = d_{B, A, \gamma}$;
3. $d_{A, C, \gamma} \leq d_{A, B, \gamma} + d_{B, C, \gamma}$.

For an arbitrary cost function $\gamma = (\gamma_I, \gamma_D, \gamma_S)$, $d_{A, B, \gamma}$ is called the *edit distance* (Wagner & Fischer, 1974) or *weighted Levenshtein distance* (Okuda, Tanaka, & Kasai, 1976) between *A* and *B*. This distance function is a metric when $\gamma_I = \gamma_D > 0$ and $\gamma_S > 0$.

The string editing problem can be characterized as a shortest path problem. Let $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ be strings of symbols where $m \geq 0$ and $n \geq 0$. The *edit graph* of *A* and *B*, denoted $G_{A, B}$, is a directed, acyclic graph having $(m + 1)(n + 1)$ vertices, which are denoted by $v_{i, j}$ for $0 \leq i \leq m$ and $0 \leq j \leq n$. The arcs of $G_{A, B}$ are divided into three groups:

1. *horizontal arcs* – $(v_{i, j-1}, v_{i, j})$ for $0 \le i \le m$ and $0 < j \le n$;

2. *vertical arcs* – $(v_{i-1, j}, v_{i, j})$ for $0 < i \le m$ and $0 \le j \le n$;

3. *diagonal arcs* – $(v_{i-1, j-1}, v_{i, j})$ for $0 < i \le m$ and $0 < j \le n$.

If $a_i = b_j$, then the diagonal arc $(v_{i-1, j-1}, v_{i, j})$ is said to be a *matching arc*; otherwise, it is termed a *non-matching arc*.

An *edit path* is any directed path in $G_{A, B}$ from $v_{0, 0}$ to $v_{m, n}$. It indicates a sequence of edit operations for transforming *A* into *B*. A horizontal arc $(v_{i, j-1}, v_{i, j})$ in the path specifies that b_j is to be inserted, while a vertical arc $(v_{i-1, j}, v_{i, j})$ indicates that a_i is to be deleted. A non-matching arc $(v_{i-1, j-1}, v_{i, j})$ specifies that b_j is to be substituted for a_i, and a matching arc implies no operation. In fact, the matching arcs of the path identify a common subsequence of *A* and *B*.

If the weight of each arc is equal to the cost of the editing operation it implies (zero for a matching arc), then an edit path having the least total weight specifies an optimal correspondence. Such a path is called a *shortest edit path*, and its total weight, or length, is the edit distance between the two strings. Note that there can be many shortest edit paths having the same length.

Figure 1 displays the edit graph for strings $A = zxy$ and $B = xyxz$. Each arc that implies an edit operation is labelled: *I* for an insertion, *D* for a deletion, or *S* for a substitution. Matching arcs are unlabelled. Figure 2 illustrates two edit paths in this edit graph. Each path identifies a sequence of edit operations that transforms *A* into *B*. The dashed path specifies one deletion and two insertions, while the dotted path indicates one insertion and two substitutions. Using cost function (1,1,2), the dashed path is a shortest edit path, and the edit distance is three. Using cost function (2,1,1), the dotted path is a shortest edit path, and the edit distance is four.

There are many applications and variations of the string editing problem (Sankoff & Kruskal, 1983). Edit distance has been used to find "close" inexact matches of a pattern within a string (Hall & Dowling, 1980) and to correct misspelled words by replacing each with the "nearest" word from a dictionary (Kukich, 1992). Handley and Hickey (1991) introduced the idea of measuring the accuracy of OCR-generated text in terms of its edit distance from the correct text. Srihari (1985) gives other applications in OCR.

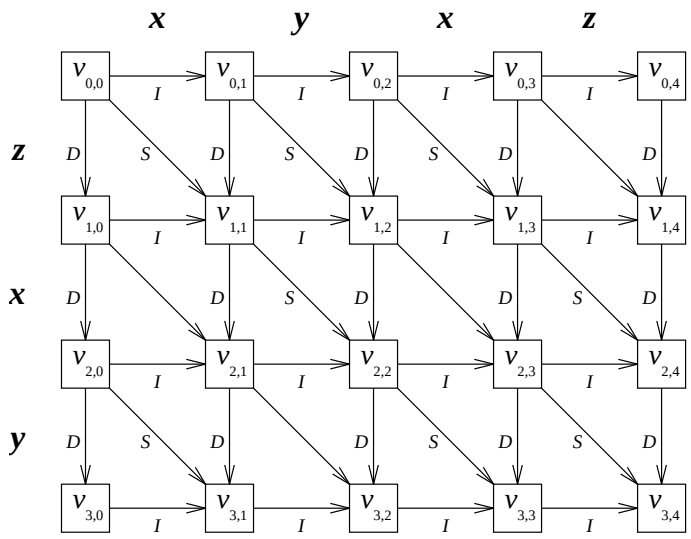


Figure 1 Example of an Edit Graph

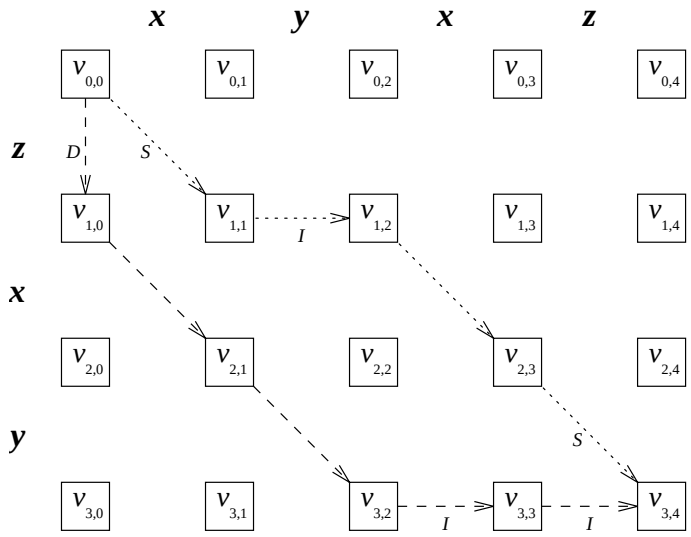


Figure 2 Examples of Edit Paths

Some lower bounds have been determined for the time complexity of algorithms for the string editing problem. For strings of length n composed of symbols from an infinite alphabet, a lower bound of $\Omega(n^2)$ has been established for algorithms restricted to "equal–not equal" comparisons (Wong & Chandra, 1976). But for a finite alphabet, Masek and Paterson (1980) developed an algorithm requiring only $O(n^2/\log n)$ time in the worst case. For the LCS problem, the following lower bounds have been established:

1. $\Omega(ns)$ for a finite alphabet of size s and "equal–not equal" comparisons (Aho, Hirschberg, & Ullman, 1976);
2. $\Omega(n\log n)$ for an infinite alphabet and "less than–equal to–greater than" comparisons (Hirschberg, 1978).

There are many algorithms for the string editing problem, although most are for the LCS case. A list appears in Table 3. The time and space complexity is given in terms of n, the length of each string, d, the edit distance, and L, the length of a LCS. Also, R denotes the number of "matches" as defined by Hunt and Szymanski (1977), and R' denotes the number of "dominant matches" (Apostolico, 1986), where $R' \le R \le n^2$.

Table 3 Algorithms for the String Editing Problem

	Cost Function	Time	Space
Wagner & Fischer, 1974	any	$O(n^2)$	$O(n^2)$
Hirschberg, 1975	(1,1,2)	$O(n^2)$	$O(n)$
Hunt & Szymanski, 1977	(1,1,2)	$O((R+n)\log n)$	$O(R+n)$
Hirschberg, 1977	(1,1,2)	$O(nL+n\log n)$	$O(nL)$
	(1,1,2)	$O(dL\log n)$	$O(d^2+n)$
Masek & Paterson, 1980	any	$O(n^2/\log n)$	$O(n^2/\log n)$
Mukhopadhyay, 1980	(1,1,2)	$O((R+n)\log n)$	$O(R+n)$
Nakatsu, Kambayashi, & Yajima, 1982	(1,1,2)	$O(nd)$	$O(nd)$
Fickett, 1984	any	$O(nd)$	$O(n^2)$
Hsu & Du, 1984; Apostolico, 1987	(1,1,2)	$O(nL+R'\log(nL/R'))$	$O(nL)$
Ukkonen, 1985	any	$O(nd)$	$O(nd)$
	(1,1,1)	$O(nd)$	$O(d^2+n)$
Myers, 1986	(1,1,2)	$O(nd)$	$O(n)$
Apostolico, 1986	(1,1,2)	$O(n\log n+R'\log(n^2/R'))$	$O(R'+n)$
Allison & Dix, 1986	(1,1,2)	$O(n^2)$	$O(n)$
Kumar & Rangan, 1987	(1,1,2)	$O(nd)$	$O(n)$
Apostolico & Guerra, 1987	(1,1,2)	$O(nL)$	$O(R'+n)$
Hadlock, 1988	(1,1,1)	$O(nd)$	$O(nd)$
Kuo & Cross, 1989	(1,1,2)	$O(R+nL+n\log n)$	$O(R+n)$
Raiha, 1990	any	$O(nd)$	$O(nd)$

Table 3 Algorithms for the String Editing Problem (Continued)

	Cost Function	Time	Space
Chin & Poon, 1990	(1,1,2)	$O(n\log n + \min\{nR', nL\})$	$O(R' + n)$
Wu, Manber, Myers, & Miller, 1990	(1,1,2)	$O(nd)$	$O(n)$
Apostolico, Browne, & Guerra, 1992	(1,1,2)	$O(nd)$	$O(n)$
Eppstein, Galil, Giancarlo, & Italiano, 1992	(1,1,2)	$O(n\log n + R'\log\log\min\{R', n^2/R'\})$	$O(R' + n)$

Ukkonen's algorithm for cost function (1,1,1) can be used to compute the character accuracy of a page-reading system; this algorithm is presented in Chapter 2 with optimizations. Algorithms for the LCS problem are listed in Table 3 as solving the string editing problem for cost function (1,1,2). But in Chapter 3, we prove that these algorithms can solve the string editing problem for many more cost functions, including (1,0,1), which is used in the computation of word accuracy. All of these LCS algorithms can benefit from the linear-time preprocessing step presented in Chapter 4.

CHAPTER 2

CHARACTER ACCURACY

Definition and Philosophy

The primary goal of a page-reading system is to identify the characters on a page with accuracy. The degree to which it accomplishes this goal is the most fundamental measure of its performance. There are many ways to quantify the deviation between OCR-generated and ground-truth text, but one meaningful approach is to estimate the cost of correcting the OCR-generated text. For many applications, a human editor must correct the generated text, at least partially, before it can be utilized; therefore, the cost of correction is important.

The string editing problem is apropos. An editor performs character insertions, deletions, and substitutions to correct the OCR-generated text. Thus, the edit distance between the OCR-generated and correct

strings is an estimate of the cost of correction. Since each edit operation can be performed by one keystroke, it is reasonable to use cost function (1,1,1). This idea was originated by Handley and Hickey (1991).

More formally, let $A = a_1 a_2 \ldots a_m$ be an OCR-generated string and $B = b_1 b_2 \ldots b_n$ be the correct string for a given page. The number of *errors* made by the page reader on this page is given by

$$E = d_{A, B, \gamma}$$

where γ = (1,1,1). That is, the number of errors is the Levenshtein distance between *A* and *B*, or the minimum number of edit operations needed to correct the OCR-generated text.

The *character accuracy* for this page is given by

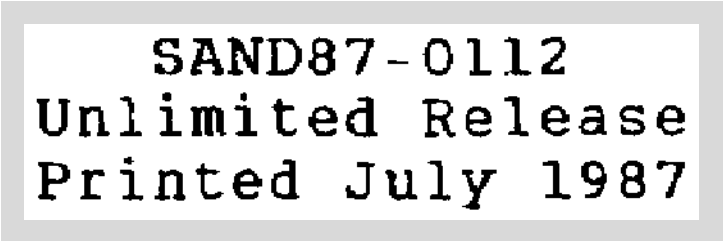
$$\frac{n - E}{n}.$$

If no errors were made, i.e., $E = 0$, the character accuracy is 100%. If 23 errors were made on a page containing 1,000 characters, the character accuracy is 97.7%. If *A* and *B* have no characters in common, then $E = \max\{m, n\}$. If $m > n$, then the character accuracy can be negative, indicating the extreme situation in which the entire correct string can be entered from scratch using fewer keystrokes than are needed to correct the

generated string.

Figure 3 displays a small page image followed by the correct string and an OCR-generated string for this page. Blanks and end-of-line characters are indicated in this example by underscores and asterisks, respectively. Notice that an end-of-line character is treated like any other character. If there were a one-to-one correspondence between the lines of the correct and generated strings, then the Levenshtein distance could be computed on a line basis, rather than a page basis. But this correspondence cannot be assumed because the process of locating lines, known as *text-line extraction*, can result in missing or extraneous lines, as well as incorrect line breaks.

Figure 3 shows the optimal correspondence between the correct and generated strings, which partitions each string into an alternating sequence of matching and non-matching substrings. The non-matching substrings describe a set of *confusions*. A confusion associates a substring of the generated string, denoted $g_1 g_2 \ldots g_p$, with a substring of the correct string, denoted $c_1 c_2 \ldots c_q$. One of these substrings, but not both, may be the empty (zero-length) string, denoted by ε. The deviation indicated by



Correct string

S A N D 8 7 - 0 1 1 2 * U n l i m i t e d _ R e l e a s e * P r i n t e d _ J u l y _ 1 9 8 7 *

S ~ l V D 8 7 0 1 1 2 * U n l i r n i t e d _ R e l e a 5 e * P r . i n t e d _ J u v _ 1 % B 7 *

OCR-generated string

Confusion			Insertions	Deletions	Substitutions
~lV	→	AN	0	1	2
ε	→	-	1	0	0
rn	→	m	0	1	1
5	→	s	0	0	1
.	→	ε	0	1	0
v	→	ly	1	0	1
%B	→	98	0	0	2
	Total		2	3	7

$n = 48$

$E = 12$

character accuracy = 75%

Figure 3 Character Accuracy Example

a confusion is corrected by

$q - \min\{p, q\}$ insertions,

$p - \min\{p, q\}$ deletions, and

$\min\{p, q\}$ substitutions,

for a total of

$\max\{p, q\}$ edit operations or errors.

It is possible to extend this error-counting method to allow wildcards or "don't care" symbols in the correct string. A page may contain one or more symbols that the page-reading system is not expected to recognize, such as a Greek letter (ζ) or bullet symbol ($\bullet$). These symbols can be represented in the correct string by one or more wildcards, where each wildcard allows the system to generate zero or one arbitrary character without being charged an error (Rice, Kanai, & Nartker, 1993b). When an optimal correspondence is computed, each wildcard will appear in a non-matching substring of the correct string, and one insertion or one substitution will be charged unless an adjustment is made to the error-counting procedure. Let w be the number of wildcards in the correct substring of a confusion ($w \leq q$). It can be easily shown that the desired

adjustment is to count

$q - \max\{w, \min\{p, q\}\}$	insertions,
$p - \min\{p, q\}$	deletions (as before), and
$\min\{p, q\} - \min\{p, w\}$	substitutions,

for a total of

$\max\{p, q\} - w$	errors.

Since the page reader should not be given credit for recognizing the wildcards, the value of *n* used to compute character accuracy should be decremented by the number of wildcards in the correct string.

To avoid charging errors for unimportant deviations in formatting, extraneous spacing characters should be removed from both strings before computing an optimal correspondence. Specifically, blank lines, and leading and trailing blanks on a line, should be discarded. Multiple consecutive blanks within a line should be replaced by a single blank.

Although character accuracy is determined for an individual page, an overall character accuracy for a set of pages can be computed by

$$\frac{\sum n_i - \sum E_i}{\sum n_i}$$

where n_i is the number of characters and E_i is the number of errors made on the ith page. This formula computes a weighted average in which the character accuracy for each single page is weighted by the number of characters on that page.

By dividing a set of pages into groups according to some page property, the overall character accuracy for each group can be determined and contrasted with one another to observe the effect of this property on accuracy. In this way, the effects of page quality, font features, skew, and resolution have been investigated (Rice, Kanai, & Nartker, 1992, 1993a, 1994; Chen, Subramaniam, Haralick, & Phillips, 1994; Rice, Jenkins, & Nartker, 1995).

Three other approaches to character accuracy will now be examined. The first approach focuses on identifying mistakes made by the page reader. The second method takes advantage of geometric information to align the OCR-generated and correct text. The third approach rewards the correct identification of ground-truth characters while excusing the generation of extraneous characters.

Identifying mistakes. A difference between the OCR-generated

string and the correct string implies that at least one incorrect decision was made by the page-reading system. The developer of the system would like to know as precisely as possible what mistakes were made by the system. Although the number of edit operations needed to correct the generated text is termed the number of "err ors," the actual number of mistakes is usually less than this quantity because it may take several edit operations to correct the deviation resulting from a single mistake. For example, if the text-line extraction process fails to locate a line containing *q* characters, then *q* errors will be charged for this one mistake because each character of the line needs to be inserted.

The mistakes made by a page-reading system can often be inferred reliably from confusions. For example, if a confusion indicates that *c* has been generated for *e*, then it is likely that the system has misinterpreted the shape of an *e*. If *cl* has been generated for *d*, then it is likely that the system has "segment ed" or divided the *d* into two components. But it is unclear what has happened when *mm* has been generated for *nin*, or when *v:'tr* has been generated for *Also*. Indeed, it is not always possible to determine with precision the mistakes made by the system.

A correspondence between the correct and generated strings identifies a set of confusions from which mistakes can be inferred. Perhaps a different correspondence, i.e., one that is not optimal with respect to minimizing the number of edit operations, will allow mistakes to be determined more precisely. Esakov, Lopresti, Sandberg, and Zhou (1994) presented an algorithm for finding a correspondence that they believe is well suited for identifying mistakes. Concepcion and D'A mato (1993) noted that if the probability of each type of mistake were known, and if mistakes were independent events, then a correspondence could be computed having the *maximum posterior probability distance*. Such a correspondence identifies the most likely set of mistakes explaining the deviations. Unfortunately, the probabilities are rarely (if ever) known, and it is clear from ISRI tests that mistakes on a given page are not independent events.

Utilizing geometry. If a page-reading system reports the image coordinates of each character it generates, and if the coordinates of each ground-truth character are available, then a correspondence could be obtained using a strictly geometric approach and there would be no need

for string matching. The mistakes made by the system could be identified more precisely. But to date, there are few page readers that produce coordinate information, and of these, some provide only line or word locations. Furthermore, it must be remembered that this information can be erroneous, which must be factored into the design of a geometric alignment algorithm. Also, the task of ground-truth data preparation is already expensive and time-consuming. If ground-truth coordinate information must also be collected, the cost of preparing a page for testing could increase tenfold.

Ignoring deletions. Some observers of the ISRI testing program have suggested a more intuitive definition of character accuracy: the percentage of ground-truth characters that are correctly identified. But under this definition, there is no cost associated with deletions, i.e., no penalty for generating extraneous characters. It is therefore possible to construct a *universal generated string* that is 100% correct for every page. Let $\Sigma = \{a_1, a_2, \ldots, a_s\}$ be a finite alphabet from which generated strings are composed, and let $A = a_1 a_2 \ldots a_s$ be a string containing every symbol of Σ. Let Υ be the generated string containing t occurrences of A, i.e.,

$\Upsilon = AA\ldots A = A^t$, where t is the largest number of symbols in a ground-truth string. The string Υ is 100% correct for every page because the first ground-truth character can be found in the first occurrence of *A*, the second ground-truth character can be found in the second occurrence of *A*, and so on. Figure 4 illustrates the universal generated string for the ASCII alphabet. While it is unlikely that a page reader would ever produce such a string, it does underscore the importance of counting deletions.

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
                .
                .
                .
```

Figure 4 Universal Generated String

Ukkonen's Algorithm

An algorithm for the string editing problem that takes $O(n^2)$ time or space is impractical for finding an optimal correspondence between an OCR-generated string and a correct string, because each string can

contain as many as 10,000 characters. Since these strings are usually very similar, i.e., the edit distance is small, an algorithm requiring $O(nd)$ time and space is much preferable. The algorithm by Ukkonen (1985) for cost function (1,1,1) is ideal, needing $O(nd)$ time and only $O(d^2 + n)$ space. This algorithm is shown in Figure 5. Given strings $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$, it returns the edit distance, *d*, and an array, *row*, from which an optimal correspondence (shortest edit path) can be extracted using the algorithm given in Figure 6.

The array, *row*, is a sparse, two-dimensional array for which only the defined elements are saved to realize the $O(d^2 + n)$ space complexity. Undefined elements are assumed to have the value (-1). The diagonal *k* of the edit graph, $G_{A, B}$, contains every vertex, $v_{i, j}$, such that $j - i = k$; thus, the diagonals of the edit graph range from $-m$ to *n*. A defined element of the array, *row*[*k*, *d*], indicates the index of the furthest row of the edit graph that can be reached by a path of length *d* from $v_{0, 0}$ to a vertex on diagonal *k*. Such a path has been termed a *furthest reaching path* by Myers (1986), who presented a similar algorithm for cost function (1,1,2).

Ukkonen's algorithm first determines how far it can "reach" from

Input: strings $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$

Output: d = edit distance between A and B using cost function (1,1,1);
row = array from which shortest edit paths can be recovered

```
procedure compute_row(k, d)
begin
    i ← max {row [k – 1, d – 1], row [k + 1, d – 1] + 1, row [k, d – 1] + 1} ;
    j ← i + k;
    while i < m and j < n and a_{i+1} = b_{j+1} do
        i ← i + 1;
        j ← j + 1
    end while;
    row [k, d] ← i
end procedure;

begin
    d ← –1;
    while row [n – m, d] ≠ m do
        d ← d + 1;
        r ← d – min {m, n} ;
        for k ← max {–m, –d} to min {–1, –r} do
            compute_row(k, d)
        end for;
        for k ← max {0, r} to min {n, d} do
            compute_row(k, d)
        end for
    end while
end.
```

Figure 5 Ukkonen's Algorithm

$v_{0,0}$ in a path of length 0. It need only consider diagonal 0 in this case. Then for diagonals –1, 0, and +1, it finds the furthest reaching paths of length 1. Using these paths, it computes the furthest reaching paths of length 2 for diagonals –2 through +2, and so on. Eventually, a furthest

Input: *m*, *n*, *d*, and *row*
Output: a shortest edit path

begin
  $i \leftarrow m$;
  $j \leftarrow n$;
  while $i > 0$ **or** $j > 0$ **do**
    $k \leftarrow j - i$;
    if $i = row[k-1, d-1]$ **then**
      horizontal arc $(v_{i, j-1}, v_{i, j})$ is in the path;
      $j \leftarrow j - 1$;
      $d \leftarrow d - 1$
    else if $i = row[k+1, d-1] + 1$ **then**
      vertical arc $(v_{i-1, j}, v_{i, j})$ is in the path;
      $i \leftarrow i - 1$;
      $d \leftarrow d - 1$
    else if $i = row[k, d-1] + 1$ **then**
      non-matching arc $(v_{i-1, j-1}, v_{i, j})$ is in the path;
      $i \leftarrow i - 1$;
      $j \leftarrow j - 1$;
      $d \leftarrow d - 1$
    else
      matching arc $(v_{i-1, j-1}, v_{i, j})$ is in the path;
      $i \leftarrow i - 1$;
      $j \leftarrow j - 1$
    end if
  end while
end.

Figure 6 Recovering a Shortest Edit Path

reaching path for diagonal $(n - m)$ encounters $v_{m, n}$ and the algorithm terminates. The length of this path is the edit distance, and *row* contains the information needed to recover the shortest edit paths.

Normally, *row*[*k*, *d*] is computed for each *k* in the range $-d$ to $+d$. But some of these diagonals can be excluded because they have no bearing on the final result. Using the variable *r*, Ukkonen's algorithm avoids some diagonals known prior to execution to be unnecessary. But the intermediate results of the algorithm can be used to exclude many more diagonals.

Let v_{i_1, j_1} and v_{i_2, j_2} be vertices of the edit graph $G_{A, B}$ such that $0 \le i_1 \le i_2 \le m$ and $0 \le j_1 \le j_2 \le n$. Let $D(v_{i_1, j_1} \to v_{i_2, j_2})$ denote the length of a shortest path from v_{i_1, j_1} to v_{i_2, j_2}. It can be easily shown that

$$\left|(i_2 - i_1) - (j_2 - j_1)\right| \le D(v_{i_1, j_1} \to v_{i_2, j_2}) \le \max\{i_2 - i_1, j_2 - j_1\}. \qquad (2.1)$$

Note that *row*[*k*, *d*] = *i* means that

$$D(v_{0, 0} \to v_{i, i + k}) = d.$$

This intermediate result can be used to obtain an upper bound, β, on the edit distance between *A* and *B*. From (2.1), we know that

$$D(v_{i, i + k} \to v_{m, n}) \le \max\{m, n - k\} - i.$$

Thus,

$$\begin{aligned} D(v_{0, 0} \to v_{m, n}) &\le D(v_{0, 0} \to v_{i, i + k}) + D(v_{i, i + k} \to v_{m, n}) \\ &\le d + \max\{m, n - k\} - i = \beta. \end{aligned}$$

For any diagonal *k*, we can derive from (2.1) the following lower bound on the length of an edit path containing a vertex on this diagonal:

$$D(v_{0,0} \rightarrow v_{i,i+k}) + D(v_{i,i+k} \rightarrow v_{m,n}) \geq |k| + |k - (n-m)|.$$

If $|k| + |k - (n-m)| > \beta$, then diagonal *k* can be disregarded since it cannot contribute a vertex to a shortest edit path. This implies that every diagonal *k* such that

$$k < \frac{n-m-\beta}{2} \qquad \text{or} \qquad k > \frac{n-m+\beta}{2}$$

can be excluded.

Figure 7 shows an optimized version of Ukkonen's algorithm. It utilizes variables *lower* and *upper* to restrict the range of diagonals for which *row*[*k*, *d*] is computed. It incorporates the aforementioned optimization, and a second, simpler one, which has also been used to optimize Myers' algorithm (Miller & Myers, 1985). When a furthest reaching path encounters a vertex $v_{m,m+k}$, then it has reached the last row of the edit graph and only diagonals greater than *k* need to be considered henceforth. Similarly, when a furthest reaching path encounters a vertex $v_{n-k,n}$, then it has reached the last column of the edit graph and only diagonals less than *k* need to be examined in the future.

Input: strings $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$

Output: d = edit distance between A and B using cost function (1,1,1);
row = array from which shortest edit paths can be recovered

procedure *compute_row*(k, d)
begin
$i \leftarrow \max \{ row[k-1, d-1], row[k+1, d-1] + 1, row[k, d-1] + 1 \}$;
$j \leftarrow i + k$;
while $i < m$ **and** $j < n$ **and** $a_{i+1} = b_{j+1}$ **do**
$i \leftarrow i + 1$; $j \leftarrow j + 1$
end while;
$row[k, d] \leftarrow i$;
if $i = m$ **then**
$lower \leftarrow k + 1$
end if;
if $j = n$ **then**
$upper \leftarrow k - 1$
end if;
$\beta \leftarrow d + \max \{ m, n-k \} - i$;
$lower \leftarrow \max \{ lower, \left\lceil \frac{n-m-\beta}{2} \right\rceil \}$;
$upper \leftarrow \min \{ upper, \left\lfloor \frac{n-m+\beta}{2} \right\rfloor \}$
end procedure;

begin
$lower \leftarrow -m$; $upper \leftarrow n$;
$d \leftarrow -1$;
while $lower \leq n - m$ **do**
$d \leftarrow d + 1$;
if $m \leq n$ **then**
for $k \leftarrow \min \{ n-m, d \}$ **downto** $\max \{ lower, -d \}$ **do**
compute_row(k, d)
end for;
for $k \leftarrow n - m + 1$ **to** $\min \{ upper, d \}$ **do**
compute_row(k, d)
end for
else
for $k \leftarrow \max \{ n-m, -d \}$ **to** $\min \{ upper, d \}$ **do**
compute_row(k, d)
end for;
for $k \leftarrow n - m - 1$ **downto** $\max \{ lower, -d \}$ **do**
compute_row(k, d)
end for
end if
end while
end.

Figure 7 Optimized Ukkonen's Algorithm

These optimizations do not improve the asymptotic time or space complexity of Ukkonen's algorithm, but experiments using OCR-generated and correct strings have revealed that on average, the optimized version computes and stores 25% fewer elements of the *row* array than the unoptimized version.

Throughput

For most applications, the accuracy of OCR-generated text is much more important than the speed at which the page-reading system produces it. Indeed, there is little use for a system that quickly generates text that is mostly gibberish. But given systems of comparable accuracy, speed can be an important discriminator.

The raw speed of a page-reading system is usually expressed in terms of the number of characters processed per second, or the number of words processed per minute. A more meaningful measure though is *throughput*, which incorporates a penalty for errors. Let n be the number of characters on a page, E be the number of errors made on the page, and S be the time it took to process the page in number of seconds. Throughput is defined as a function of P, the penalty assigned to each error:

$$\frac{n - P \times E}{S}.$$

For a set of pages, throughput is given by

$$\frac{\sum n_i - P \sum E_i}{\sum S_i}.$$

When $P = 0$, throughput is equal to the raw speed in terms of characters per second. Diehl and Eglowstein (1991) and Jones (1992) expressed throughput as "correct words per minute." When $P = 1$, we obtain in essence, "correct characters per second." But this is a rather insignificant penalty for errors. For most applications, a larger value, such as $P = 10$, is a better reflection of the real-world trade-off of speed versus accuracy.

Accuracy by Character Class

It is useful to determine how well a page-reading system identifies the characters belonging to a particular class. For example, we may want to know what percentage of the lowercase letters, or decimal digits, were correctly recognized. For the Japanese and Chinese character sets, each containing several thousand symbols, it is especially valuable to analyze

the performance of the system on various subsets (J. Kanai, personal communication, 1994).

Let Σ be the alphabet from which ground-truth strings are composed. A *character class* C is any subset of Σ. Let A be an OCR-generated string and B be the correct string for a given page, and assume that an optimal correspondence between A and B has been computed using cost function (1,1,1). Let n' be the number of symbols in B belonging to class C. Let E' be the number of symbols in B that belong to C and are part of a non-matching substring induced by the optimal correspondence. The accuracy for class C on this page is given by

$$\frac{n' - E'}{n'},$$

and for a set of pages by

$$\frac{\sum n_i' - \sum E_i'}{\sum n_i'}.$$

For the example in Figure 3, the accuracy achieved on lowercase letters was 82.6% ($n' = 23$, $E' = 4$), and 80% of the decimal digits were correctly identified ($n' = 10$, $E' = 2$).

To determine the percentage of occurrences of the letter *e* that were correctly recognized, a class containing only this character is utilized.

Marked Character Efficiency

Finding and correcting errors in OCR-generated text is a tedious and expensive process for users. But a page-reading system provides some assistance by flagging the generated characters that it believes are most likely in error. A *reject character*, usually a tilde (~), is placed in the output whenever the system is unable to recognize a character. In addition, if a character has been generated with low confidence, the system marks the character as "suspect." This may be accomplished by placing a meta-character, known as a *suspect marker* (often the "^" symbol), immediately preceding the character in question. Reject characters, and characters marked as suspect, are called *marked characters*.

This sentenc~ conta^lns reject charact~rs an^d suspect markars.

Not all errors should be considered equal. An error that is flagged is termed a *marked error.* Clearly, this type of error is much less costly to correct than an *unmarked error.* In the above example, there are three

marked errors: the two reject characters and the *l* in "contains." There is one unmarked error: the second *a* in "markers." The marked *d* in "and" is an example of a *false mark*, which is a correctly-generated character that has been marked as suspect. It takes time to verify the correctness of false marks, which adds to the overall cost of correction. The page-reading system tries to mark as many of its errors as possible while minimizing the number of false marks.

Let $A = a_1 a_2 \ldots a_m$ be an OCR-generated string and $B = b_1 b_2 \ldots b_n$ be the correct string for a given page. The generated character a_i is a marked character if it is a reject character (i.e., $a_i = \sim$), or if it is marked as suspect by a meta-character. (The meta-character is not one of the symbols of *A*.)

Assume that an optimal correspondence between *A* and *B* has been established using cost function (1,1,1). If a marked character is part of a matching substring induced by this correspondence, then it is a false mark. Otherwise, it is part of a non-matching substring and flags a confusion. One marked character in a non-matching substring is considered enough to mark all of the errors associated with the confusion. For example, if

r^n has been generated for *m*, then both errors are considered to be marked even though only one marked character is present. Thus, unmarked errors are counted only for those confusions having no marked characters.

Consider the process of examining the marked characters and correcting the marked errors. If *U* is the number of unmarked errors, then the accuracy of the text after this process, called the *character accuracy after correction*, is

$$\frac{n - U}{n}$$

for one page, and

$$\frac{\sum n_i - \sum U_i}{\sum n_i}$$

for a set of pages. The efficiency of this process depends on the gain in accuracy relative to the amount of work involved. If a large percentage of the marked characters are false marks, then much effort is expended to obtain a small increase in accuracy.

A *marked character efficiency curve*, which bears some similarity

to the error-reject curve (Chow, 1994), reflects the efficiency of this correction process. An example is presented in Figure 8. The *x*-dimension indicates the amount of work, in terms of the number of marked characters examined, expressed as a percentage of the total number of ground-truth characters. The *y*-dimension shows the character accuracy after correction.

The first point on the curve ($x = 0.0$) indicates the base character accuracy, i.e., the accuracy of the generated text before any corrections are made. The second point shows the character accuracy after correcting the errors identified by reject characters. The slope of the line segment connecting the first and second points is high, indicating that this is a very efficient operation. Indeed, a reject character can never be a false mark.

The third point of the curve shows the character accuracy after correcting the errors identified by either reject characters or the first "level" of suspect markers. Most page readers allow the user to control the number of suspect markers produced by specifying a discrete level, or a value on a continuous scale. The slope of the second line segment reflects the efficiency of examining the first-level suspect markers. The slope of each

subsequent line segment indicates the efficiency of processing an additional level of suspect markers. The flattening of the curve in Figure 8 is a typical pattern: as the number of marked characters increases, the percentage of false marks increases, and the correction process becomes less efficient.

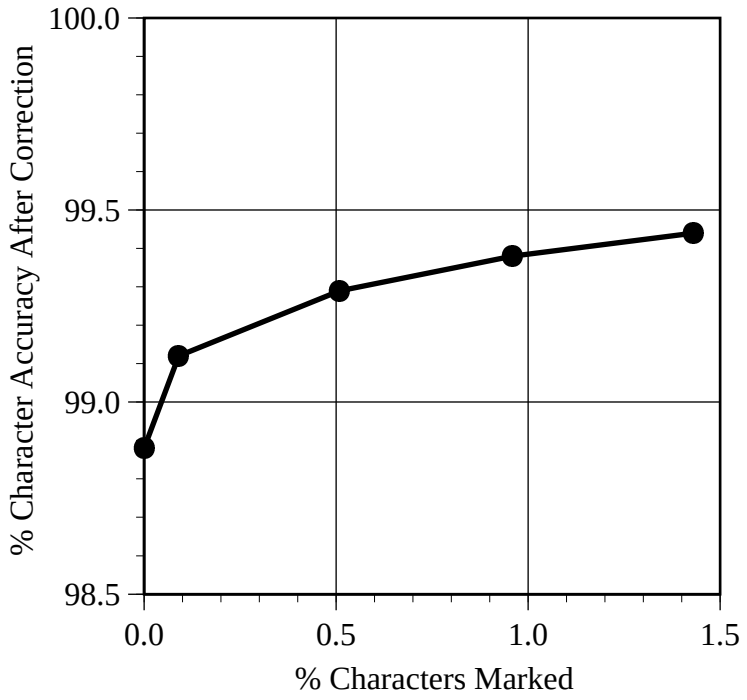


Figure 8 Marked Character Efficiency Curve

CHAPTER 3

CLASSES OF COST FUNCTIONS

In this chapter, we digress from the presentation of performance measures to develop a theoretical result pertaining to the string editing problem. This result will be used in Chapter 4 in the computation of word accuracy.

We begin with the following observation: a correspondence between two strings is optimal using cost function $\gamma = (\gamma_I, \gamma_D, \gamma_S)$ if and only if it is optimal using cost function $\gamma' = (c\gamma_I, c\gamma_D, c\gamma_S)$, where c is a positive, real constant. Uniform scaling of the edit operation costs means only that the unit of measurement has changed, which cannot affect whether an edit path is shortest. Cost functions γ and γ' are considered to be "equivalent" because they yield the same optimal correspondences.

Recent work in parametric sequence alignment has addressed the

problem of determining which cost functions produce the same optimal correspondences for a specific pair of input strings (Gusfield, Balasubramanian, & Naor, 1994; Bunke & Csirik, 1995). In this chapter, we identify the cost functions that are equivalent for every pair of input strings.

Let $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ be strings, and let P be an edit path in $G_{A,B}$. Let K_P and L_P denote the number of diagonal arcs and matching arcs, respectively, in P. It can be easily shown that

$$0 \leq L_P \leq K_P \leq \min\{m, n\},$$

and that P has $(n - K_P)$ horizontal arcs and $(m - K_P)$ vertical arcs.

Let $d_{P,\gamma}$ denote the length of P using cost function $\gamma = (\gamma_I, \gamma_D, \gamma_S)$. Since P implies a sequence of edit operations consisting of $(n - K_P)$ insertions, $(m - K_P)$ deletions, and $(K_P - L_P)$ substitutions, then

$$d_{P,\gamma} = (n - K_P)\gamma_I + (m - K_P)\gamma_D + (K_P - L_P)\gamma_S.$$

If there does not exist an edit path P' in $G_{A,B}$ such that $d_{P',\gamma} < d_{P,\gamma}$, then P is a shortest edit path.

Let $\Phi_{A,B,\gamma}$ denote the set of shortest edit paths in $G_{A,B}$ using γ. The edit distance between A and B is equal to the length of a shortest edit path, i.e.,

$$d_{A,B,\gamma} = d_{P,\gamma} \text{ where } P \in \Phi_{A,B,\gamma}.$$

Cost functions γ and γ' are *equivalent* if $\Phi_{A,B,\gamma} = \Phi_{A,B,\gamma'}$ for every pair of strings *A* and *B*. A *class of cost functions* is a set of cost functions Γ such that if γ and γ' are any two elements of Γ, then γ and γ' are equivalent.

Given an edit path *P*, and a cost function $\gamma = (\gamma_I, \gamma_D, \gamma_S)$ such that $\gamma_S > 0$, let

$$F_{P,\gamma} = K_P\left(\frac{\gamma_I + \gamma_D}{\gamma_S} - 1\right) + L_P.$$

The length of an edit path can be expressed in terms of *F*, as indicated by the following lemma.

LEMMA 3.1. *Given strings* $A = a_1 a_2 \ldots a_m$ *and* $B = b_1 b_2 \ldots b_n$, *let P be an edit path in* $G_{A,B}$, *and let* $\gamma = (\gamma_I, \gamma_D, \gamma_S)$ *be a cost function with* $\gamma_S > 0$. *Then*

$$d_{P,\gamma} = n\gamma_I + m\gamma_D - \gamma_S F_{P,\gamma}.$$

PROOF. We know that

$$\begin{aligned} d_{P,\gamma} &= (n - K_P)\gamma_I + (m - K_P)\gamma_D + (K_P - L_P)\gamma_S \\ &= n\gamma_I + m\gamma_D - K_P(\gamma_I + \gamma_D - \gamma_S) - L_P\gamma_S \\ &= n\gamma_I + m\gamma_D - \gamma_S F_{P,\gamma}. \ \square \end{aligned}$$

Since $\gamma_S > 0$, it is clear from Lemma 3.1 that an edit path that minimizes *d* is an edit path that maximizes *F*. This observation is formalized in the following lemma.

LEMMA 3.2. *Given strings* $A = a_1 a_2 \ldots a_m$ *and* $B = b_1 b_2 \ldots b_n$, *let P be an edit path in* $G_{A,B}$, *and let* $\gamma = (\gamma_I, \gamma_D, \gamma_S)$ *be a cost function with* $\gamma_S > 0$. *Then* $P \in \Phi_{A,B,\gamma}$ *if and only if there does not exist an edit path* P' *in* $G_{A,B}$ *such that* $F_{P',\gamma} > F_{P,\gamma}$.

PROOF. Let $P \in \Phi_{A,B,\gamma}$. Then there does not exist an edit path P' in $G_{A,B}$ such that $d_{P',\gamma} < d_{P,\gamma}$. Suppose there exists an edit path P' in $G_{A,B}$ such that $F_{P',\gamma} > F_{P,\gamma}$. Since $\gamma_S > 0$, we know that

$$n\gamma_I + m\gamma_D - \gamma_S F_{P',\gamma} < n\gamma_I + m\gamma_D - \gamma_S F_{P,\gamma}.$$

By Lemma 3.1, this implies that $d_{P',\gamma} < d_{P,\gamma}$, which is a contradiction.

Now let us assume that there does not exist an edit path P' in $G_{A,B}$ such that $F_{P',\gamma} > F_{P,\gamma}$. Suppose that $P \notin \Phi_{A,B,\gamma}$. Then there exists an

edit path P' in $G_{A,B}$ such that $d_{P',\gamma} < d_{P,\gamma}$. By Lemma 3.1, we know that

$$n\gamma_I + m\gamma_D - \gamma_S F_{P',\gamma} < n\gamma_I + m\gamma_D - \gamma_S F_{P,\gamma}.$$

Since $\gamma_S > 0$, this implies that $F_{P',\gamma} > F_{P,\gamma}$, which is a contradiction. □

Thus, whether an edit path P is shortest depends on $F_{P,\gamma}$, which depends not on γ_I, γ_D, and γ_S individually, but on the ratio of $\gamma_I + \gamma_D$ to γ_S. This motivates the following definition: let

$$\Gamma_r = \{ (\gamma_I,\gamma_D,\gamma_S) \mid \gamma_S > 0 \text{ and } \frac{\gamma_I + \gamma_D}{\gamma_S} = r \}$$

where r is a non-negative real number. In the following theorem, we show that Γ_r is a class of cost functions.

THEOREM 3.1. *Let* $A = a_1 a_2 \ldots a_m$ *and* $B = b_1 b_2 \ldots b_n$ *be strings, and let* $\gamma = (\gamma_I,\gamma_D,\gamma_S)$ *and* $\gamma' = (\gamma'_I, \gamma'_D, \gamma'_S)$ *be cost functions in* Γ_r, *where r is a non-negative real number. Then* $\Phi_{A,B,\gamma} = \Phi_{A,B,\gamma'}$, *and*

$$d_{A,B,\gamma'} = \frac{n(\gamma'_I\gamma_S - \gamma_I\gamma'_S) + m(\gamma'_D\gamma_S - \gamma_D\gamma'_S) + \gamma'_S d_{A,B,\gamma}}{\gamma_S}. \quad (3.1)$$

PROOF. First we show that $\Phi_{A, B, \gamma} = \Phi_{A, B, \gamma'}$. Let $P \in \Phi_{A, B, \gamma}$. Then by Lemma 3.2, we know that there does not exist an edit path P' in $G_{A, B}$ such that $F_{P', \gamma} > F_{P, \gamma}$. Suppose that $P \notin \Phi_{A, B, \gamma'}$. Then by Lemma 3.2, we know that there exists an edit path P' in $G_{A, B}$ such that $F_{P', \gamma'} > F_{P, \gamma'}$. But

$$F_{P', \gamma'} = K_{P'}(r-1) + L_{P'} = F_{P', \gamma},$$

and similarly, $F_{P, \gamma'} = F_{P, \gamma}$. This implies that $F_{P', \gamma} > F_{P, \gamma}$, which is a contradiction. Thus, $P \in \Phi_{A, B, \gamma'}$, and we know that $\Phi_{A, B, \gamma} \subseteq \Phi_{A, B, \gamma'}$. Using the same argument, we can show that $\Phi_{A, B, \gamma'} \subseteq \Phi_{A, B, \gamma}$; hence, $\Phi_{A, B, \gamma} = \Phi_{A, B, \gamma'}$.

We will now prove that equation (3.1) holds. Let $P \in \Phi_{A, B, \gamma}$. Then $d_{A, B, \gamma} = d_{P, \gamma}$, and by Lemma 3.1, we know that

$$d_{A, B, \gamma} = n\gamma_I + m\gamma_D - \gamma_S F_{P, \gamma}.$$

Thus,

$$F_{P, \gamma} = \frac{n\gamma_I + m\gamma_D - d_{A, B, \gamma}}{\gamma_S}. \tag{3.2}$$

Since $P \in \Phi_{A, B, \gamma}$, we know that $P \in \Phi_{A, B, \gamma'}$ and $d_{A, B, \gamma'} = d_{P, \gamma'}$. Again by Lemma 3.1, we have

$$d_{A,\,B,\,\gamma'} = n\gamma'_I + m\gamma'_D - \gamma'_S F_{P,\,\gamma'}.$$

But $F_{P,\,\gamma'} = F_{P,\,\gamma}$; hence,

$$d_{A,\,B,\,\gamma'} = n\gamma'_I + m\gamma'_D - \gamma'_S F_{P,\,\gamma}. \qquad (3.3)$$

Substituting equation (3.2) into equation (3.3) yields equation (3.1). □

Thus, for every non-negative real number *r*, there exists a class of cost functions, Γ_r. The set of shortest edit paths in $G_{A,\,B}$ is the same for every cost function in Γ_r, but the edit distance will vary. However, given the edit distance for one cost function in Γ_r, the edit distance for any other cost function in Γ_r can be computed using equation (3.1). Hence, an algorithm for the string editing problem that works for cost function (1,1,1) (such as Ukkonen's algorithm) can be used for any cost function in Γ_2. Also, an algorithm for the LCS problem (cost function (1,1,2)) can solve the string editing problem for any cost function in Γ_1.

We have shown that every cost function $\gamma = (\gamma_I, \gamma_D, \gamma_S)$ with $\gamma_S > 0$ belongs to a class of cost functions. Now let us consider the remaining cost functions. Let

$$\Gamma_\infty = \{\ (\gamma_I, \gamma_D, \gamma_S) \mid \gamma_I + \gamma_D > 0 \text{ and } \gamma_S = 0\ \}.$$

It can be easily shown that Γ_∞ is a class of cost functions. If

$A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ are strings, and $\gamma = (\gamma_I, \gamma_D, \gamma_S)$ is any cost function in Γ_∞, then $\Phi_{A, B, \gamma}$ is the set of edit paths in $G_{A, B}$ having the maximum number of diagonal arcs, i.e.,

$$\Phi_{A, B, \gamma} = \{ P \mid P \text{ in } G_{A, B} \text{ and } K_P = \min\{m, n\} \}.$$

Furthermore, the edit distance is obtained directly by

$$d_{A, B, \gamma} = \begin{cases} (n - m)\,\gamma_I & \text{if } m \le n \\ (m - n)\,\gamma_D & \text{if } m > n. \end{cases}$$

Now only the degenerate cost function $\gamma = (0,0,0)$ remains. In this trivial case, for every pair of strings A and B, the set $\Phi_{A, B, \gamma}$ contains every edit path in $G_{A, B}$, and $d_{A, B, \gamma} = 0$. The singleton set containing this cost function is a class, which we denote by $\Gamma_{0/0}$.

Having partitioned the universe of cost functions into classes, we now consider whether any of these classes can be combined to form a larger class. That is, can we find real numbers r and r' such that $0 \le r < r'$ and $\Gamma_r \cup \Gamma_{r'}$ is a class? Let

$$\Gamma_{(0, 1)} = \bigcup_{0 < r < 1} \Gamma_r = \left\{ (\gamma_I, \gamma_D, \gamma_S) \;\middle|\; \gamma_S > 0 \text{ and } 0 < \frac{\gamma_I + \gamma_D}{\gamma_S} < 1 \right\}.$$

We will show that $\Gamma_{(0, 1)}$ is a class, and that no other classes can be

formed.

LEMMA 3.3. *Given strings A and B, let P be an edit path in* $G_{A, B}$, *and let* $\gamma = (\gamma_I, \gamma_D, \gamma_S)$ *be a cost function in* $\Gamma_{(0, 1)}$. *If* $P \in \Phi_{A, B, \gamma}$, *then* $K_P = L_P$.

PROOF. Let $P \in \Phi_{A, B, \gamma}$. By Lemma 3.2, we know that there does not exist an edit path P' in $G_{A, B}$ such that $F_{P', \gamma} > F_{P, \gamma}$. Suppose $K_P > L_P$. Then there exists a non-matching arc $(v_{i-1, j-1}, v_{i, j})$ in P. Let P' be the edit path in $G_{A, B}$ formed by replacing $(v_{i-1, j-1}, v_{i, j})$ in P by the horizontal arc $(v_{i-1, j-1}, v_{i-1, j})$ and the vertical arc $(v_{i-1, j}, v_{i, j})$. Then $K_{P'} = K_P - 1$ and $L_{P'} = L_P$. Thus,

$$\begin{aligned} F_{P', \gamma} &= K_{P'}\left(\frac{\gamma_I + \gamma_D}{\gamma_S} - 1\right) + L_{P'} \\ &= (K_P - 1)\left(\frac{\gamma_I + \gamma_D}{\gamma_S} - 1\right) + L_P \\ &= F_{P, \gamma} + 1 - \frac{\gamma_I + \gamma_D}{\gamma_S}. \end{aligned}$$

Since $\gamma \in \Gamma_{(0, 1)}$, then $\frac{\gamma_I + \gamma_D}{\gamma_S} < 1$, and $F_{P', \gamma} > F_{P, \gamma}$, which is a contradiction. Hence, $K_P = L_P$. □

THEOREM 3.2. *Let r and r' be real numbers such that $0 \le r < r'$, and let $\gamma = (\gamma_I, \gamma_D, \gamma_S)$ and $\gamma' = (\gamma'_I, \gamma'_D, \gamma'_S)$ be cost functions in Γ_r and $\Gamma_{r'}$, respectively. Then γ and γ' are equivalent if and only if $0 < r < r' < 1$.*

PROOF. Let γ and γ' be equivalent. Then by definition, there does not exist a pair of strings A and B such that $\Phi_{A, B, \gamma} \neq \Phi_{A, B, \gamma'}$. Suppose $0 < r < r' < 1$ is not true. We consider three cases.

Case 1: $r = 0 < r'$. Let $A = B = x$, and let P and P' be edit paths in $G_{A, B}$, where P consists of one horizontal and one vertical arc, and P' consists of one matching arc. Since $r = 0$, we know that $d_{P, \gamma} = 0$ and $P \in \Phi_{A, B, \gamma}$. But $r' > 0$ implies that $d_{P, \gamma'} > 0 = d_{P', \gamma'}$; thus, $P \notin \Phi_{A, B, \gamma'}$.

Case 2: $r < 1 \le r'$. Let $A = x$ and $B = y$ with $x \neq y$, and let P and P' be edit paths in $G_{A, B}$, where P consists of one horizontal and one vertical arc, and P' consists of one non-matching arc. Since $r' \ge 1$, we know that $d_{P', \gamma'} \le d_{P, \gamma'}$ and $P' \in \Phi_{A, B, \gamma'}$. But $r < 1$ implies that $d_{P, \gamma} < d_{P', \gamma}$; hence, $P' \notin \Phi_{A, B, \gamma}$.

Case 3: $1 \le r < r'$. Let $\frac{t}{u}$ be a rational number such that $r < \frac{t}{u} < r'$,

where t and u are positive integers, and let $x, y, z_1, z_2, \ldots, z_{(t-u)}$ be distinct symbols. Let A be the string consisting of u consecutive x's followed by $z_1 z_2 \ldots z_{(t-u)}$, and let B be the string consisting of $z_1 z_2 \ldots z_{(t-u)}$ followed by u consecutive y's. That is,

$$A = \overbrace{xx\ldots x}^{u} z_1 z_2 \ldots z_{(t-u)}, \text{ and}$$

$$B = z_1 z_2 \ldots z_{(t-u)} \underbrace{yy\ldots y}_{u}.$$

Let P and P' be edit paths in $G_{A,B}$, where P consists of u horizontal arcs, u vertical arcs, and $(t-u)$ matching arcs, and P' consists of t non-matching arcs. Thus,

$$\begin{aligned} d_{P,\gamma} &= u(\gamma_I + \gamma_D), \\ d_{P',\gamma} &= t\gamma_S, \\ d_{P,\gamma'} &= u(\gamma'_I + \gamma'_D), \text{ and} \\ d_{P',\gamma'} &= t\gamma'_S. \end{aligned}$$

Since

$$\frac{\gamma_I + \gamma_D}{\gamma_S} = r < \frac{t}{u} < r' = \frac{\gamma'_I + \gamma'_D}{\gamma'_S},$$

it follows that $d_{P,\gamma} < d_{P',\gamma}$ and $d_{P',\gamma'} < d_{P,\gamma'}$. Thus, $P \notin \Phi_{A,B,\gamma'}$, yet it

can be easily shown that $P \in \Phi_{A, B, \gamma}$.

In each case, we have contradicted the fact that γ and γ' are equivalent by identifying strings A and B such that $\Phi_{A, B, \gamma} \neq \Phi_{A, B, \gamma'}$; hence, we know that $0 < r < r' < 1$.

Now we assume that $0 < r < r' < 1$, and will show that γ and γ' are equivalent. Let A and B be any pair of strings, and let P and P' be edit paths in $G_{A, B}$ such that $P \in \Phi_{A, B, \gamma}$ and $P' \in \Phi_{A, B, \gamma'}$. By Lemma 3.3, we know that $K_P = L_P$ and $K_{P'} = L_{P'}$; thus,

$$
\begin{aligned}
F_{P, \gamma} &= rL_P , \\
F_{P', \gamma} &= rL_{P'} , \\
F_{P, \gamma'} &= r'L_P , \text{ and} \\
F_{P', \gamma'} &= r'L_{P'} .
\end{aligned}
$$

Suppose $P \notin \Phi_{A, B, \gamma'}$. Then by Lemma 3.2, we know that $F_{P', \gamma'} > F_{P, \gamma'}$, and $L_{P'} > L_P$. But this implies that $F_{P', \gamma} > F_{P, \gamma}$, which contradicts the fact that $P \in \Phi_{A, B, \gamma}$; hence, $P \in \Phi_{A, B, \gamma'}$ and $\Phi_{A, B, \gamma} \subseteq \Phi_{A, B, \gamma'}$. Using the same argument, we can show that $\Phi_{A, B, \gamma'} \subseteq \Phi_{A, B, \gamma}$; thus, $\Phi_{A, B, \gamma} = \Phi_{A, B, \gamma'}$, and γ and γ' are equivalent. □

From Theorem 3.2, we know that $\Gamma_{(0,\,1)}$ is a class, and that no other classes can be formed. Thus, the following represents the complete set of equivalence classes induced by the equivalence relation:

Γ_0

$\Gamma_{(0,\,1)}$

Γ_r for every real number $r \geq 1$

Γ_∞

$\Gamma_{0/0}$.

Some of these classes are depicted in Figure 9.

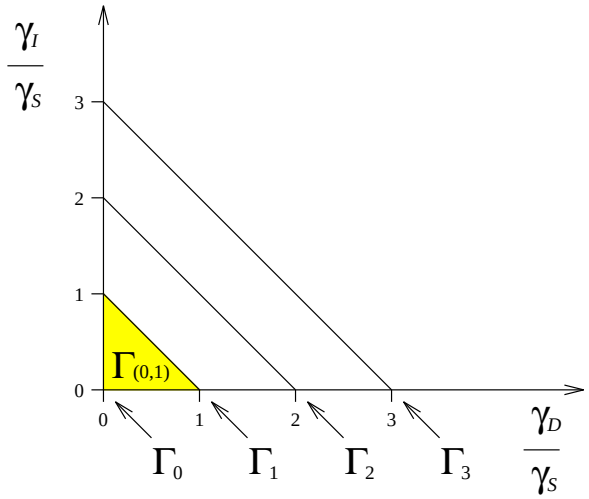


Figure 9 Classes of Cost Functions

Let $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ be strings. Every edit path in $G_{A, B}$ identifies a common subsequence of *A* and *B*, where each matching arc corresponds to one symbol of the common subsequence. Also, every common subsequence of *A* and *B* is identified by one or more edit paths in $G_{A, B}$. If *P* is an edit path in $G_{A, B}$ such that $L_P = L_{A, B}$, then *P* identifies a LCS of *A* and *B*.

Wagner and Fischer (1974) have shown that if $\gamma = (1,1,2)$, then every edit path in $\Phi_{A, B, \gamma}$ identifies a LCS of *A* and *B*, and

$$d_{A, B, \gamma} = n + m - 2L_{A, B}.$$

We extend this result to a set of cost functions, denoted by Γ_{LCS}, which itself is not a class, but is the following union of classes:

$$\Gamma_{LCS} = \Gamma_{(0, 1)} \cup \Gamma_1.$$

THEOREM 3.3. *Given strings* $A = a_1 a_2 \ldots a_m$ *and* $B = b_1 b_2 \ldots b_n$, *let P be an edit path in* $G_{A, B}$, *and let* $\gamma = (\gamma_I, \gamma_D, \gamma_S)$ *be a cost function in* Γ_{LCS}. *If* $P \in \Phi_{A, B, \gamma}$, *then* $L_P = L_{A, B}$, *and*

$$d_{A, B, \gamma} = n\gamma_I + m\gamma_D - (\gamma_I + \gamma_D)L_{A, B}.$$

PROOF. Let $P \in \Phi_{A, B, \gamma}$. By Lemma 3.2, we know that there does not exist an edit path P' in $G_{A, B}$ such that $F_{P', \gamma} > F_{P, \gamma}$.

Case 1: $\gamma \in \Gamma_{(0,\,1)}$. Let $r = \dfrac{\gamma_I + \gamma_D}{\gamma_S}$. By Lemma 3.3, we know that $K_P = L_P$, and $F_{P,\,\gamma} = rL_P$. Suppose $L_P < L_{A,\,B}$. Clearly, there exists an edit path P' in $G_{A,\,B}$ such that $K_{P'} = L_{P'} = L_{A,\,B}$. But

$$F_{P',\,\gamma} = K_{P'}(r-1) + L_{P'} = rL_{A,\,B} > rL_P = F_{P,\,\gamma},$$

which is a contradiction. Hence, $L_P = L_{A,\,B}$. Also,

$$\begin{aligned} d_{A,\,B,\,\gamma} &= d_{P,\,\gamma} \\ &= (n - K_P)\gamma_I + (m - K_P)\gamma_D + (K_P - L_P)\gamma_S \\ &= n\gamma_I + m\gamma_D - (\gamma_I + \gamma_D)K_P \\ &= n\gamma_I + m\gamma_D - (\gamma_I + \gamma_D)L_{A,\,B}. \end{aligned}$$

Case 2: $\gamma \in \Gamma_1$. We observe that $F_{P,\,\gamma} = L_P$. Since there does not exist an edit path P' in $G_{A,\,B}$ such that $F_{P',\,\gamma} = L_{P'} > L_P = F_{P,\,\gamma}$, it is clear that $L_P = L_{A,\,B}$. Using Lemma 3.1, we know that

$$\begin{aligned} d_{A,\,B,\,\gamma} &= d_{P,\,\gamma} \\ &= n\gamma_I + m\gamma_D - \gamma_S F_{P,\,\gamma} \\ &= n\gamma_I + m\gamma_D - \gamma_S L_{A,\,B} \\ &= n\gamma_I + m\gamma_D - (\gamma_I + \gamma_D)L_{A,\,B}. \end{aligned}$$ □

Thus, the string editing problem is equivalent to the LCS problem when using any cost function in Γ_{LCS}. This equivalence does not hold for cost functions that are not in Γ_{LCS}; this can be shown trivially for Γ_0,

Γ_{∞}, and $\Gamma_{0/0}$, and by using the construction presented in Case 3 of the Theorem 3.2 proof for Γ_r where $r > 1$.

CHAPTER 4

WORD ACCURACY

Definition and Philosophy

An important application of page-reading systems is to build a text database from a collection of hard-copy documents. Information retrieval techniques can then be applied to find documents of interest. Typically, a document is located based on the words it contains. Hence, in this application, the accurate recognition of words is more important than the correct identification of numbers or punctuation.

An OCR-generated or correct string of characters can be parsed to extract an ordered list of words. The parsing depends on the definition of a word. We define a word to be simply any sequence of one or more letters, but a more complicated definition could be used. By considering each word to be one symbol, the list of words can be represented by a

string.

Let $A = a_1 a_2 \ldots a_m$ be an OCR-generated string of words and $B = b_1 b_2 \ldots b_n$ be the correct string of words for a given page. To correct the OCR-generated string, insertions, deletions, and substitutions of words can be performed. An insertion is needed to enter a missing word; a deletion is used to remove an extraneous generated word; and a substitution replaces an incorrect word with the correct one.

We do not wish to penalize the page reader for generating extraneous words because if the user chooses not to delete them and leaves them in the text database, there would be little impact on retrieval effectiveness. Thus, the appropriate cost function to use when aligning strings *A* and *B* is (1,0,1). The edit distance between *A* and *B* using this cost function is the number of "ess ential" edit operations (i.e., insertions plus substitutions). It is also the number of misrecognized words because each insertion or substitution corresponds to one misrecognized word. Let *E* denote this value, i.e.,

$$E = d_{A, B, \gamma}$$

where γ = (1,0,1). The *word accuracy* for this page is given by

$$\frac{n - E}{n}$$

and is the percentage of words that are correctly recognized.

Since $\gamma \in \Gamma_{LCS}$, we know from Chapter 3 that we can use any LCS algorithm to align strings *A* and *B*. From Theorem 3.3, we know that

$$E = n - L_{A, B};$$

hence, word accuracy can be expressed as

$$\frac{L_{A, B}}{n}.$$

Indeed, a LCS of *A* and *B* is composed of the correctly recognized words. For a set of pages, word accuracy is computed by

$$\frac{\sum n_i - \sum E_i}{\sum n_i} \quad \text{or} \quad \frac{\sum L_i}{\sum n_i}$$

where L_i is the length of a LCS for the *i*th page.

Figure 10 displays a small page image followed by the correct string and an OCR-generated string of characters for this page. Beneath these strings is a diagram showing the correct and generated strings of words, and their longest common subsequence, which identifies the

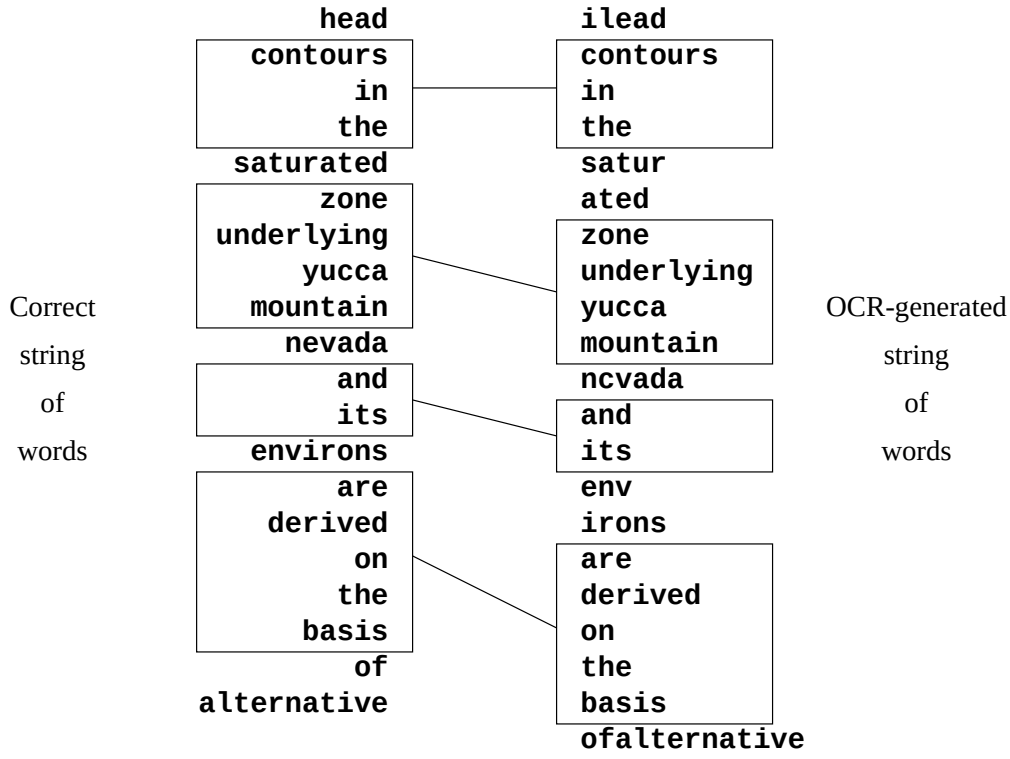
Head contours in the saturated zone underlying Yucca Mountain,
Nevada, and its environs are derived on the basis of alternative

Correct string of characters

Head contours in the saturated zone underlying Yucca Mountain, Nevada, and its environs are derived on the basis of alternative

OCR-generated string of characters

Ilead contours in the satur ated zone underlying yucca Mountain. Ncvada. and its env irons are deriVed on the basis ofalternative



$n = 20$

$E = 6$

word accuracy = 70%

Figure 10 Word Accuracy Example

correctly recognized words. Notice that each word is one symbol. Symbols are compared for equality on a case-insensitive basis. Since full-text searching is usually insensitive to case, it is appropriate to accept as correct, "yu cca" generated for "Yucca," and "deriVed" generated for "der ived." Notice also that errors in punctuation (periods generated for commas) are not penalized; indeed, this type of error has no significant effect on retrieval. The generated string in this example is corrected by one insertion, two deletions, and five substitutions, but $E = 6$ since deletions are ignored. (Although we ignore deletions in the computation of word accuracy, a universal generated string, which is 100% correct for every page, cannot be constructed because the alphabet of words is infinite.)

LCS Preprocessing

Let $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ be strings of symbols from an alphabet Σ. Suppose a symbol $x \in \Sigma$ occurs one or more times in *A*, but does not occur in *B*. Then *x* cannot be part of a longest common subsequence of *A* and *B*. Removing each occurrence of *x* from *A* before computing a LCS of *A* and *B* will speed up the computation without

changing the result. This leads to the following preprocessing step for LCS algorithms: remove each symbol from *A* that does not appear in *B*, and remove each symbol from *B* that does not appear in *A*.

Figure 11 shows one algorithm that accomplishes this task. First, the distinct symbols of *A* are stored in the data structure, T_A, and likewise, the distinct symbols of B are saved in T_B. Then each symbol of *A* that is found in T_B is retained to produce a smaller string, A'. Similarly, each symbol of *B* that is found in T_A is saved to create the smaller string, B'. A LCS of *A* and *B* can now be computed more efficiently by operating on these smaller strings.

If the data structure chosen for T_A and T_B allows "insert" and "lookup" operations to be performed in constant time, then the running time of this algorithm is linear, i.e., $O(m + n)$. If the alphabet is small, this data structure can be an array having one element for each symbol of the alphabet, and the symbol value can be used to index the array directly. If the alphabet is large, then a hash table can be utilized instead, with the symbol value as the hash key. In either case, the desired operations can be performed in constant time, although for a hash table lookup, this is

Input: strings $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$

Output: strings $A' = a'_1\, a'_2 \ldots a'_{m'}$ and $B' = b'_1\, b'_2 \ldots b'_{n'}$
where $m' \le m$ and $n' \le n$

```
begin
    T_A ← empty;
    for i ← 1 to m do
        if a_i not in T_A then
            insert a_i into T_A
        end if
    end for;
    T_B ← empty;
    for i ← 1 to n do
        if b_i not in T_B then
            insert b_i into T_B
        end if
    end for;
    j ← 0;
    for i ← 1 to m do
        if a_i in T_B then
            j ← j + 1;
            a'_j ← a_i
        end if
    end for;
    m' ← j;
    j ← 0;
    for i ← 1 to n do
        if b_i in T_A then
            j ← j + 1;
            b'_j ← b_i
        end if
    end for;
    n' ← j
end.
```

Figure 11 Preprocessing Step for LCS Algorithms

expected, not worst-case time.

By reducing the size of the input strings, the preprocessing step can speed up every LCS algorithm listed in Table 3 of Chapter 1. Those algorithms whose running time depends on the edit distance are helped further because the edit distance is reduced by one for each symbol that is removed. If u symbols are removed from each string, the combined running time of the preprocessing step and an $O(nd)$ LCS algorithm is

$$O(n + (n - u)(d - 2u)).$$

The preprocessing step is especially helpful when computing word accuracy. When a page reader misrecognizes a word, it is likely to generate a word that does not appear anywhere in the correct string; hence, it will be removed from the generated string by the preprocessing step. Furthermore, unless the misrecognized word was generated elsewhere on the page, it will be removed from the correct string. In an extreme case, it is possible that only those words belonging to a LCS will remain after the preprocessing. This is the case for the example in Figure 10.

Non-stopword Accuracy

In text retrieval, common words known as *stopwords* are normally not indexed because they offer little or no retrieval value. Here are some examples of stopwords from three different languages:

- in English, *the*, *of*, *and*, *to*, *a*;
- in Spanish, *de*, *la*, *el*, *y*, *en*;
- in German, *der*, *die*, *in*, *und*, *von*.

Words that are not stopwords, called *non-stopwords*, are indexed. Typically, users search for documents containing one or more non-stopwords. Hence, the accuracy with which a page-reading system identifies non-stopwords is especially relevant in a text retrieval application.

Let A be an OCR-generated string of words and B be the correct string of words for a given page. Assume that a LCS of A and B has been computed, which identifies the symbols of B that have been correctly recognized. Let S be a set of stopwords, and let n' be the number of symbols in B that are non-stopwords, i.e., not in S. Let E' be the number of non-stopwords in B that were misrecognized. The *non-stopword accuracy* for this page is given by

$$\frac{n' - E'}{n'}$$

and is the percentage of non-stopwords that were correctly recognized. For a set of pages, non-stopword accuracy is computed by

$$\frac{\sum n_i' - \sum E_i'}{\sum n_i'}.$$

For the example in Figure 10, if the set of stopwords is

$$S = \{a, and, are, in, its, of, on, the, to\},$$

then the non-stopword accuracy is 58.33% ($n' = 12$, $E' = 5$).

Phrase Accuracy

In text retrieval, users also search for documents containing specific phrases. We define a phrase of length *k* to be any sequence of *k* consecutive words. Phrases may overlap; thus, a correct string of words, $B = b_1 b_2 \ldots b_n$, has $(n - k + 1)$ phrases of length *k*, namely

$b_1 b_2 \ldots b_k$

$b_2 b_3 \ldots b_{k+1}$

…

$b_{n-k+1} b_{n-k+2} \ldots b_n$,

where $1 \le k \le n$.

Let $A = a_1 a_2 \ldots a_m$ be an OCR-generated string of words and $B = b_1 b_2 \ldots b_n$ be the correct string of words for a given page. Assume that a LCS of *A* and *B* has been computed, which identifies the symbols of *B* that have been correctly recognized. A phrase of length *k* is considered to be correctly recognized provided all *k* words of the phrase have been correctly identified. Let $n' = n - k + 1$ be the number of phrases of length *k* in *B*, and let E' be the number of those phrases that were misrecognized. The *phrase accuracy* for this page is given by

$$\frac{n' - E'}{n'}$$

and is the percentage of phrases of length *k* that were correctly recognized. For a set of pages, phrase accuracy is computed in the usual way:

$$\frac{\sum n_i' - \sum E_i'}{\sum n_i'}.$$

For the example in Figure 10, the phrase accuracy for phrases of length 4 is 17.6% ($n' = 17$, $E' = 14$). The only correct phrases are "zone underlying yucca mountain," "ar e derived on the," and "derived on

the basis."

Typically, we compute phrase accuracy for $k = 1, 2, \ldots, 8$ and plot the results. An example is given in Figure 12. As one would expect, phrase accuracy decreases as k increases. Note that the phrase accuracy for $k = 1$ is equal to the word accuracy.

Phrase accuracy provides a useful measure of "error bunching." Suppose that page-reading systems X and Y have misrecognized the same number of words (and thus, have the same word accuracy), but the phrase accuracy of X is higher than the phrase accuracy of Y. This means that the words missed by X are more closely "bunched" or "clustered" than the words missed by Y, and hence, are easier to correct.

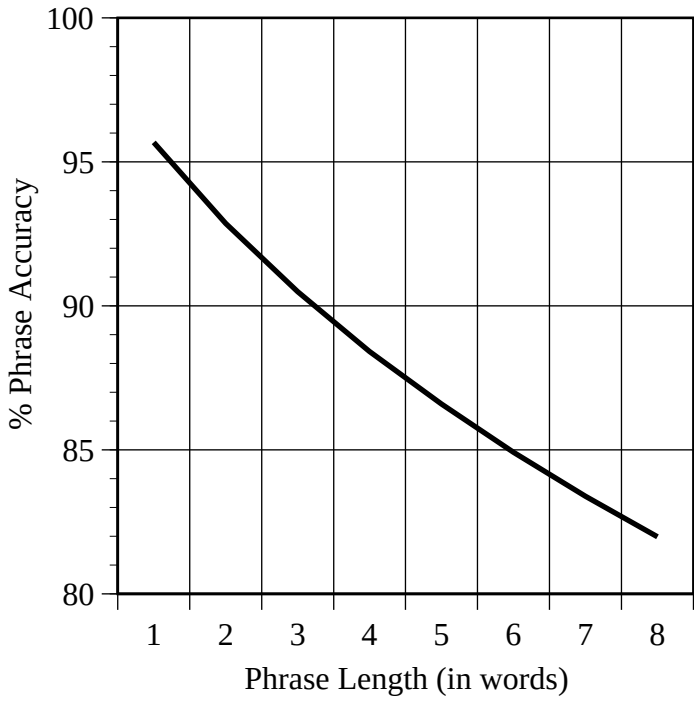


Figure 12 Phrase Accuracy

CHAPTER 5

CONCLUSION

The behavior of page-reading systems is complex and unpredictable. It is necessary to conduct meaningful tests to gain insight, to identify problems, and ultimately, to make improvements. Large-scale, automated tests are needed in which expressive and precise measures of performance are computed. In this dissertation, we have presented definitions and algorithms for such measures. These measures are utilized in the ISRI annual test, which benefits users, vendors, and researchers.

1. *Character accuracy* indicates how well the page-reading system identifies characters, and is expressed in terms of the editing effort needed to correct the generated text.
2. *Throughput* combines raw processing speed with character accuracy in a single measure of performance.

3. *Accuracy by character class* shows how well a subset of the characters is recognized, such as digits (0-9) or uppercase letters (A-Z).
4. *Marked character efficiency* gauges the extent to which reject characters and suspect markers assist users in locating errors in generated text.
5. *Word accuracy* indicates how well the page-reading system identifies words, and is motivated by the popular text retrieval application.
6. *Non-stopword accuracy* focuses on the correct identification of non-stopwords, which is especially relevant to text retrieval.
7. *Phrase accuracy* is the percentage of phrases that are correctly recognized, but can also indicate whether errors are bunched or scattered in the generated text.

For all of these measures, the string editing problem provides the method for aligning correct and generated strings. Some new insights and optimizations have been presented in this dissertation that benefit this and other applications of the string editing problem.

1. The universe of cost functions is divided into equivalence classes.

2. An algorithm for a specific cost function can also be used for any equivalent cost function.

3. The string editing problem is equivalent to the LCS problem when using any cost function $(\gamma_I, \gamma_D, \gamma_S)$ that satisfies $0 < \gamma_I + \gamma_D \leq \gamma_S$.

4. An algorithm by Ukkonen (1985) can be optimized by exploiting its intermediate results.

5. The computation of a LCS can be made faster by a linear-time pre-processing step in which symbols that occur in only one input string are removed.

Additional measures of page reader performance continue to be developed. Each new measure provides a unique perspective on the page-reading process. Progress in page-reading technology depends on thoughtful, multi-faceted evaluation.

BIBLIOGRAPHY

Aho, A. V., Hirschberg, D. S., & Ullman, J. D. (1976). Bounds on the complexity of the longest common subsequence problem. *Journal of the ACM*, *23*(1), 1-12.

Allison, L., & Dix, T. I. (1986). A bit-string longest-common-subsequence algorithm. *Information Processing Letters*, *23*(6), 305-310.

Apostolico, A. (1986). Improving the worst-case performance of the Hunt-Szymanski strategy for the longest common subsequence of two strings. *Information Processing Letters*, *23*(2), 63-69.

Apostolico, A. (1987). Remark on the Hsu-Du new algorithm for the longest common subsequence problem. *Information Processing Letters*, *25*(4), 235-236.

Apostolico, A., Browne, S., & Guerra, C. (1992). Fast linear-space computations of longest common subsequences. *Theoretical Computer Science*, *92*, 3-17.

Apostolico, A., & Guerra, C. (1987). The longest common subsequence problem revisited. *Algorithmica*, *2*, 315-336.

Bokser, M. (1992). Omnidocument technologies. *Proceedings of the IEEE*, *80*(7), 1066-1078.

Bradford, R., & Nartker, T. (1991). Error correlation in contemporary OCR systems. In *Proceedings of the First International Conference on Document Analysis and Recognition* (pp. 516-524). Saint-Malo, France.

Bunke, H., & Csirik, J. (1995). Parametric string edit distance and its application to pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, *25*(1), 202-206.

Chen, S., Subramaniam, S., Haralick, R. M., & Phillips, I. T. (1994). Performance evaluation of two OCR systems. In *Proceedings of the Third Annual Symposium on Document Analysis and Information Retrieval* (pp. 299-317). Las Vegas: University of Nevada, Information Science Research Institute.

Chin, F. Y. L., & Poon, C. K. (1990). A fast algorithm for computing longest common subsequences of small alphabet size. *Journal of Information Processing*, *13*(4), 463-469.

Chow, C. K. (1994). Recognition error and reject trade-off. In *Proceedings of the Third Annual Symposium on Document Analysis and Information Retrieval* (pp. 1-8). Las Vegas: University of Nevada, Information Science Research Institute.

Concepcion, V. P., & D'Amato, D. P. (1993). Synchronous tracking of outputs from multiple OCR systems. In D. P. D'Amato (Ed.), *Proceedings of SPIE: Vol. 1906. Character recognition technologies* (pp. 218-228). San Jose, CA: SPIE.

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

Diehl, S., & Eglowstein, H. (1991, April). Tame the paper tiger. *Byte*, pp. 220-238.

Eppstein, D., Galil, Z., Giancarlo, R., & Italiano, G. F. (1992). Sparse dynamic programming I: Linear cost functions. *Journal of the ACM, 39*(3), 519-545.

Esakov, J., Lopresti, D. P., Sandberg, J. S., & Zhou, J. (1994). Issues in automatic OCR error classification. In *Proceedings of the Third Annual Symposium on Document Analysis and Information Retrieval* (pp. 401-412). Las Vegas: University of Nevada, Information Science Research Institute.

Fickett, J. W. (1984). Fast optimal alignment. *Nucleic Acids Research, 12*(1), 175-179.

Gilder, J. H., & Neilson, M. (1993, March). Real-world OCR tests. *Imaging Magazine*, pp. 72-81.

Gusfield, D., Balasubramanian, K., & Naor, D. (1994). Parametric optimization of sequence alignment. *Algorithmica*, *12*, 312-326.

Hadlock, F. (1988). Minimum detour methods for string or sequence comparison. *Congressus Numerantium*, *61*, 263-274.

Hall, P. A. V., & Dowling, G. R. (1980). Approximate string matching. *ACM Computing Surveys*, *12*(4), 381-402.

Handley, J. C., & Hickey, T. B. (1991). Merging optical character recognition outputs for improved accuracy. In *Proceedings of the RIAO 91 Conference* (pp. 160-174). Barcelona, Spain.

Hirschberg, D. S. (1975). A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, *18*(6), 341-343.

Hirschberg, D. S. (1977). Algorithms for the longest common subsequence problem. *Journal of the ACM, 24*(4), 664-675.

Hirschberg, D. S. (1978). An information-theoretic lower bound for the longest common subsequence problem. *Information Processing Letters*, *7*(1), 40-41.

Hsu, W. J., & Du, M. W. (1984). New algorithms for the LCS problem. *Journal of Computer and System Sciences*, *29*, 133-152.

Hunt, J. W., & Szymanski, T. G. (1977). A fast algorithm for computing longest common subsequences. *Communications of the ACM, 20*(5), 350-353.

Jones, M. (1992, July). From print to PC: OCR gets you from there to here. *PC Magazine*, pp. 267-319.

Kanai, J., Liu, Y., Rice, S. V., & Nartker, T. A. (1994). *A preliminary evaluation of Chinese OCR systems* (Tech. Rep. No. 94-04). Las Vegas: University of Nevada, Information Science Research Institute.

Kanai, J., Nartker, T. A., Rice, S. V., & Nagy, G. (1993). Performance metrics for document understanding systems. In *Proceedings of the Second International Conference on Document Analysis and Recognition* (pp. 424-427). Tsukuba Science City, Japan.

Kanai, J., Rice, S. V., & Nartker, T. A. (1993). *A preliminary evaluation of automatic zoning* (Tech. Rep. No. 93-02). Las Vegas: University of Nevada, Information Science Research Institute.

Kanai, J., Rice, S. V., Nartker, T. A., & Nagy, G. (1995). Automated evaluation of OCR zoning. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 17*(1), 86-90.

Kukich, K. (1992). Techniques for automatically correcting words in text. *ACM Computing Surveys*, *24*(4), 377-439.

Kumar, S. K., & Rangan, C. P. (1987). A linear space algorithm for the LCS problem. *Acta Informatica*, *24*, 353-362.

Kuo, S., & Cross, G. R. (1989). An improved algorithm to find the length of the longest common subsequence of two strings. *ACM SIGIR Forum*, *23*, 89-99.

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

Masek, W. J., & Paterson, M. S. (1980). A faster algorithm computing string edit distances. *Journal of Computer and System Sciences, 20*(1), 18-31.

McClelland, D. (1991, November). Teaching your Mac to read. *Mac-World*, pp. 169-175.

Miller, W., & Myers, E. W. (1985). A file comparison program. *Software—Practice and Experience, 15*(11), 1025-1040.

Mukhopadhyay, A. (1980). A fast algorithm for the longest-common-subsequence problem. *Information Sciences, 20*, 69-82.

Myers, E. W. (1986). An *O(ND)* difference algorithm and its variations. *Algorithmica*, *1*, 251-266.

Nagy, G. (1995). Document image analysis: Automated performance evaluation. In A. Dengel & A. L. Spitz (Eds.), *Document analysis systems*. Singapore: World Scientific.

Nakatsu, N., Kambayashi, Y., & Yajima, S. (1982). A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, *18*, 171-179.

Nartker, T. A., & Rice, S. V. (1994, September). OCR accuracy: UNLV's third annual test. *Inform*, Association for Information and Image Management, pp. 30-36.

Nartker, T. A., Rice, S. V., & Jenkins, F. R. (1995, July). OCR accuracy: UNLV's fourth annual test. *Inform*, Association for Information and Image Management, pp. 38-46, 55.

Nartker, T. A., Rice, S. V., & Kanai, J. (1994, January). OCR accuracy: UNLV's second annual test. *Inform*, Association for Information and Image Management, pp. 40-45.

Needleman, S. B., & Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology, 48,* 443-453.

Okuda, T., Tanaka, E., & Kasai, T. (1976). A method for the correction of garbled words based on the Levenshtein metric. *IEEE Transactions on Computers*, *C-25*(2), 172-178.

Phillips, I. T., Chen, S., Ha, J., & Haralick, R. M. (1993). English document database design and implementation methodology. In *Proceedings of the Second Annual Symposium on Document Analysis and Information Retrieval* (pp. 65-104). Las Vegas: University of Nevada, Information Science Research Institute.

RAF Technology, Inc. (1995). *DAFS: Document attribute format specification*. Redmond, WA.

Raiha, L. (1990). Approximate sequence comparison: A study with histograms. *Pattern Recognition*, *12*(1/2), 159-169.

Raucci, R. (1993, January). OCR moves into the mainstream. *SunWorld*, pp. 47-52.

Rice, S. V. (1993). *The OCR experimental environment, version 3* (Tech. Rep. No. 93-04). Las Vegas: University of Nevada, Information Science Research Institute.

Rice, S. V., Jenkins, F. R., & Nartker, T. A. (1995). *The fourth annual test of OCR accuracy* (Tech. Rep. No. 95-04). Las Vegas: University of Nevada, Information Science Research Institute.

Rice, S. V., Kanai, J., & Nartker, T. A. (1992). *A report on the accuracy of OCR devices* (Tech. Rep. No. 92-02). Las Vegas: University of Nevada, Information Science Research Institute.

Rice, S. V., Kanai, J., & Nartker, T. A. (1993a). *An evaluation of OCR accuracy* (Tech. Rep. No. 93-01). Las Vegas: University of Nevada, Information Science Research Institute.

Rice, S. V., Kanai, J., & Nartker, T. A. (1993b). *Preparing OCR test data* (Tech. Rep. No. 93-08). Las Vegas: University of Nevada, Information Science Research Institute.

Rice, S. V., Kanai, J., & Nartker, T. A. (1994). *The third annual test of OCR accuracy* (Tech. Rep. No. 94-03). Las Vegas: University of Nevada, Information Science Research Institute.

Sankoff, D., & Kruskal, J. B. (Eds.). (1983). *Time warps, string edits, and macromolecules: The theory and practice of sequence comparison*. Reading, MA: Addison-Wesley.

Srihari, S. N. (Ed.). (1985). *Computer text recognition and error correction*. Silver Springs, MD: IEEE Computer Society Press.

Ukkonen, E. (1985). Algorithms for approximate string matching. *Information and Control*, *64*, 100-118.

Vintsyuk, T. K. (1968). Speech discrimination by dynamic programming. *Cybernetics*, *4*(1), 52-58.

Wagner, R. A., & Fischer, M. J. (1974). The string-to-string correction problem. *Journal of the ACM*, *21*(1), 168-173.

Welch, E. M. (1993, January). Can you read this? OCR software. *MacUser*, pp. 152-184.

Wong, C. K., & Chandra, A. K. (1976). Bounds for the string editing problem. *Journal of the ACM*, *23*(1), 13-16.

Wu, S., Manber, U., Myers, G., & Miller, W. (1990). An *O(NP)* sequence comparison algorithm. *Information Processing Letters*, *35*(6), 317-323.