Approximate String Matching

PATRICK A. V. HALL

SCICON Consultancy International Limited, Sanderson House, 49 Berners Street, London W1P 4AQ, England

GEOFF R. DOWLING

Department of Computer Science, The City University, Northampton Square, London EC1V OHB, England

Approximate matching of strings is reviewed with the aim of surveying techniques suitable for finding an item in a database when there may be a spelling mistake or other error in the keyword. The methods found are classified as either equivalence or similarity problems. Equivalence problems are seen to be readily solved using canonical forms. For similarity problems difference measures are surveyed, with a full description of the well-established dynamic programming method relating this to the approach using probabilities and likelihoods. Searches for approximate matches in large sets using a difference function are seen to be an open problem still, though several promising ideas have been suggested. Approximate matching (error correction) during parsing is briefly reviewed.

Keywords and Phrases: approximate matching, spelling correction, string matching, error correction, misspelling, string correction, string editing, errors, best match, syntax errors, equivalence, similarity, longest common subsequence, searching, file organization, information retrieval

CR Categories: 1.3, 3.63, 3.7, 3.73, 3.74, 4.12, 5.42

INTRODUCTION

Looking up a person's name in a directory or index is an exceedingly common operation in information systems. When the name is known in exactly the form in which it is recorded in the directory, then looking it up is easy. But what if there is a difference? There may be a legitimate spelling variation, or the name may be misspelled. In either situation the lookup procedure will fail unless some special search is undertaken. Yet this requirement of searching when the string is almost right is very common in information systems.

This paper shows builders of information systems what is possible in finding approximate matches for arbitrary strings. Existing methods are placed within a general framework, and some new techniques are added.

Behind this string matching problem is a yet more general problem of approximately matching arbitrary information items or groups of items. This survey avoids this very general problem, although many of the methods surveyed are applicable. We concentrate instead on the matching of a single string within a set of strings. Strings have special properties, and string matching has many important applications.

Many investigations of string matching have concentrated on searching for a particular string embedded as a substring of another, to satisfy retrieval problems such

CONTENTS

as finding a document whose title mentions some particular word. Methods for finding a substring within another string have culminated in the elegant method of Boyer and Moore [BOYE77, GALI79] where by preprocessing the substring it is possible to make large steps through the string to find a match in sublinear time on average. Rivest [RIVE77] has shown that the worst case behavior must take linear time.

Instead of searching for a single substring one could search for a "pattern." This facility is common in string processing languages (it is illustrated by the language SNOBOL, which is documented in GIMP76) and has been developed by Aho and Corasick [AHO75] and Knuth, Morris, and Pratt [KNUT77]. However, general pattern matching is equivalent to asking whether the string conforms to a grammar, and thus the algorithms involved are parsing algorithms (see, for example, HOPC69).

The basic problem we examine is different. It is as follows.

Problem: Approximate String Matching

Given a string s drawn from some set S of possible strings (the set of all strings composed of symbols drawn from some alphabet A), find a string t which approximately matches this string, where t is in a subset T of S.

The task is either to find all those strings in T that are "sufficiently like" s, or the N strings in T that are "most like" s. The intuitive concepts "approximate," "sufficiently like," and "most like" need elucidation. We shall see two broad categories of the problem in Sections 2 and 3, where the idea of "like" is regarded as "equivalent," or as "different but similar."

A secondary factor in our problem is the representation of the set of strings T. This set can either be represented extensionally as an enumeration of the strings in the set (that is, all the strings are explicitly stored), or intensionally as a set of rules such as a grammar. Most of the discussions in this paper are in terms of extensional sets. Discussion of intensional sets is delayed until Section 4.

1. REASONS FOR APPROXIMATE MATCHING

Before describing the various approaches to approximate matching in Sections 2, 3, and 4, it is worth examining further the reasons for approximate matching. There are two very different viewpoints: "error correction" and "information retrieval."

We can suppose that what should be provided as a search string corresponds precisely to what has been stored in some record or records. The search string does not match because of some corruption process which has changed it. The corruption process has a magnitude associated with it, and we can talk of large corruptions and small corruptions. Furthermore we can imagine that the string gets so badly corrupted that it becomes similar or identical to some other stored string. Thus if the corruptions are larger than the differences between correct strings, we must expect to retrieve falsely, and only if we were to weaken our retrieval criterion, would we expect to be able to retrieve the correct string as an outlying match.

We can think of ourselves as trying to correct the errors introduced by the corruption, with the retrieval process being the attempt to correct the error, and with retrieval of a string which is not relevant being an error. This corruption-correction point of view is adopted in communication theory [PETE61] and pattern recognition [RISE74].

Alternatively, we can take the viewpoint of information retrieval: our search string indicates, as best we can, the information required. We could be unsuccessful in two ways. There is the risk that unwanted records will be retrieved, while required records are missed. In conventional information retrieval these two phenomena are captured by the notion of precision and recall [SALT68, PAIC77]:

- *precision*—proportion of retrieved records that are relevant;
- *recall*—proportion of relevant records actually retrieved.

It is assumed that the relevance of records is known from other sources. These measures are not fully satisfactory, and Paice [PAIC77] suggests refinements. Recently alternative information-theoretic measures were proposed [RADE76]. Nevertheless, precision and recall remain useful conceptually; we see that in general we can trade one against the other. By being less exacting in what is retrieved, recall can be made to approach 100 percent at the expense of precision approaching zero, and vice versa. With retrieval based upon a similarity or difference measure and a threshold, the trade-off can be controlled by varying the threshold; this is covered in Section 3.1. In the information retrieval literature the imagery of precision and recall appears to encourage ad hoc approaches, possibly because a correct analysis is very difficult and something is better than nothing.

Before we move to consider these ideas in further detail, we hope that two facts have been seen emerging. First, we must understand the sources of the corruptions or variability that are requiring us to make approximate matchings, and we must compensate for them accurately. Second, we must know something about the size of the corruptions and adjust our retrieval criterion accordingly, and expect that for large corruptions we will get a degraded performance however we choose to measure it.

2. EQUIVALENCE

2.1 The Equivalence Problem

One notion of "approximate" and "like" is *equivalence*. If two strings which are superficially different can be substituted for each other in all contexts without making any difference in meaning, then they are equivalent.

Common examples of equivalence are alternate spellings of the same word, the use of spaces as formatting characters, optional use of upper- or lowercase letters, and alternative scripts. For example, all the following strings might be considered as equivalent.

Data Base data-base data base database

data base d a t a b a s e Database.

In Arabic and other languages using the Arabic script, there is considerable discretion in how words are typed, associated with the art of calligraphy [HALL78].

Another very different example of equivalence occurs in arithmetic expressions. The same basic calculation can be expressed in many ways by using different orders, bracketing, and repeating arguments in order to give an infinite variety of expressions, all of which are equivalent (see, for example, JENK76).

A very important example in keyword searching in information retrieval [PAIC77] is the treatment of all grammatical variants of a word as equivalent as far as retrieval is concerned. Normally, mechanisms here attempt to reduce words to their stem or root, and then to treat all words that can be reduced to the same stem as equivalent.

In some interpretations [UNES76] synonyms can be viewed as equivalents, but synonyms are more properly considered as similarities and are discussed in Section 3.1.

It is possible that some abbreviations can be viewed as alternative spellings and thus as equivalences—for example, LTD. for LIMITED. In general this is not possible, since several words may have the same abbreviation—for example, ST. for both SAINT and STREET.

The idea of equivalence is well understood in mathematics [BIRK70]. One can talk of an equivalence relation "$\approx$" on the set S of all possible strings, such that for strings r, s, t in S

(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 first two properties are obvious. It is the third property that is important, the property that if r is equivalent to s, and s is equivalent to t, then r is equivalent to t.

We can now reformulate our matching problem for equivalences.

Equivalence Problem

Given s in S, find all t in T such that $s \approx t$.

The equivalence relation divides the set S of all strings into subsets $S_1, S_2, S_3, \ldots$, such that all strings in a subset are equivalent to each other and not equivalent to any string in any other subset. These subsets are called "equivalence classes." We can paraphrase our problem as finding all the elements of T which are in the same equivalence class as the search string s.

Equivalence classes can be characterized by some typical or exemplary member of the class. This exemplary member is frequently known as the *canonical form* for the class [BIRK70], and usually there are rules for transforming any element into its canonical form (that is, the canonical form of its equivalence class). Since there is a one-to-one correspondence between canonical forms and equivalence classes, it gives another formulation of our problem, to find all the elements t in T with the same canonical form as the search string s.

2.2 Storing and Retrieving Equivalent Strings

All methods rely upon the well-established technology of storing and retrieving *exact* matches using a retrieval key, as exemplified by Knuth [KNUT73] or Martin [MART75].

To solve the equivalence problem directly, all strings are separately indexed, and all members of an equivalence class are linked together in some manner. Thus any string indexes into its equivalence class, and all equivalent strings can be retrieved. This method can be used for alternative spellings and for thesauri where synonyms are treated as equivalences [UNES76]. For symbol tables in interpreters and compilers where alternative keywords which are not systematic abbreviations are used (as, for example, in PL/1), these indexes would be predetermined and would be hand optimized in their design. However equivalences sometimes are only obtained as part of the acquisition of data and would be generated dynamically, as happens with EQUIVALENCE statements in programming languages [GRIE71, TARJ75].

(1) Change internal z's to s's when preceded and followed by a vowel or y.

Examples: razor, analyze, realize
Counterexamples: hazard, squeeze

(2) Replace all internal occurrences of 'ph' by 'f'.

Examples: sulphur, peripheral, symphony
Counterexamples: uphill, haphazard

(3) For words of at least six letters, replace a word ending 'our' by 'or'.

Examples: flavour, humour
Counterexample: devour

(4) After removing endings such as 'e', 'ate', and 'ation', replace the endings 'tr' by 'ter'.

Examples: centr(e), filtr(ate)

FIGURE 1. Rules for producing a canonical form for English and American spellings. (From PAIC77.)

The equivalence problem based on canonical forms is much the more common form of the problem encountered, so we first consider methods for reducing a string to a canonical form. Often the transformation is trivial, involving the removal of some extraneous characters and/or the replacement of optional characters by some standard choice [SLIN79]. But in the cases of alternative spellings and the roots of words, the methods are more elaborate. The differences between English and American spelling can mostly be defined by rules which convert words to some standard spelling. Figure 1, taken from Paice [PAIC77], gives a set of possible rules.

The extraction of true roots of words is seldom attempted, but the removal of alternative word endings is common. Truncation is not adequate, and more elaborate methods known as "conflation" are used. Table 1, from Paice, gives a set of "simple" rules; understanding the precise effect of these is helped by the example. Rules like these could be applied to most languages. They will usually be incomplete in that there will be many variants that are not accounted for, and they will also treat as

TABLE 1. PAICE'S CONFLATION RULES FOR REDUCING A FAMILY OF WORDS TO A COMMON ROOT—THE RULES ARE INCOMPLETE, BUT ARE CLAIMED TO BE SATISFACTORY [PAIC77]

Label	*Ending*	*Replacement*	*Transfer*
	—ably	—	goto IS
	—ibly	—	finish
	—ily	—	goto SS
SS	—ss	—ss	finish
	—ous	—	finish
	—ies	—y	goto ARY
	—s	—	goto E
	—ied	—y	goto ARY
	—ed	—	goto ABL
	—ing	—	goto ABL
E	—e	—	goto ABL
	—al	—	goto ION
ION	—ion	—	goto AT
	—	—	finish
ARY	—ary	—	finish
	—ability	—	goto IS
	—ibility	—	finish
	—ity	—	goto IV
	—ify	—	finish
	—	—	finish
ABL	—abl	—	goto IS
	—ibl	—	finish
IV	—iv	—	goto AT
AT	—at	—	goto IS
IS	—is	—	finish
	—ific	—	finish
	—olv	—olut	finish
	—	—	finish

equivalent words that are not equivalent. This leads to degraded performance as measured by precision and recall.

For example, consider the reduction of the word "conducts" to its root "conduct." Starting at the top of Table 1 we compare word endings until the ending "—s" is found. The replacement rule indicates no replacement, so the "s" is deleted. The transfer column indicates that we should continue searching from the label "E." So, starting from label "E," searching continues, matching word endings until the null ending is reached which does match, leading to no replacement and "finish." The root "conduct" has been found.

Having defined the canonical forms for the equivalence classes and rules for transforming arbitrary strings to their canonical forms, these are two ways in which the canonical form can be used.

First, the canonical form can be produced immediately on input, so that the canonical form is the only form that is manipulated in the computer, being stored, used for indexes, and so on. This means that the original string as input is lost, and therefore that strings which are retrieved will in general be different from those which were input. Indeed, they may be unreadable unless some compensating transformation is undertaken to render them readable. This method is invariably used in programming languages for strings representing numerical data—these strings are reduced to a canonical binary form—but it is otherwise of limited applicability. It is used for identifiers in some compilers, for example in FORTRAN where spaces are removed, and in some Arabic systems where the reduction to canonical form is made in the peripherals themselves [HALL78].

Second, the canonical form need only be used where it matters, and the string as input is stored and retrieved. The canonical form is used whenever two strings are compared. If a string item in a record is indexed, the string is reduced to canonical form before searching the index or adding a new entry to the index. When strings are sorted, a sort key consisting of the canonical form is extracted, and this sort key is used in sorting. An example of such a use is given in the system reported by Slinn [SLIN79].

Of course it is possible to store a string as received and on retrieval test all stored strings for equivalence. This is very time consuming for large sets and underutilizes the structure present in the problem. Because the methods of this section use standard searching technology, they are effective for large sets. As will be seen in the next section, searching a database with a keyword similar to the one stored is difficult to do efficiently.

3. SIMILARITY

3.1 The Similarity Problem

By far the most usual understanding of "approximate" or "like" is that of *similarity* between two strings. By some inspection process, two strings can be determined to be similar or not. The important property of similarity which makes it very different from equivalence is that similarity is not necessarily transitive; that is, if r is similar

to s and s is similar to t, then it does not necessarily follow that r is similar to t.

In computer-based information systems, errors of typing and spelling constitute a very common source of variation between strings. These errors have been widely investigated. Shaffer and Hardwich [SHAF68] found in typing that substitution of one letter for another was the most common error, followed by the omission of a letter and then the insertion of a letter. Bourne [BOUR77] has investigated typing errors in a number of bibliographic databases, finding as many as 22.8 percent of its index terms to be misspellings in one database and as low as 0.4 percent in another, with an average of 10.8 percent over all the databases sampled. Litecky and Davis [LITE76] have investigated errors in COBOL programs and found that approximately 20 percent of all errors were due to misspellings and mistypings. All investigations agree that the most common typing mistakes found are single character omissions and insertions, substitutions, and the reversal of adjacent characters. Damerau [DAME64] has reported that over 80 percent of all typing errors are of this type. This has been confirmed by Morgan [MORG70]. Spelling errors, by contrast, may be phonetic in origin. In a study by Masters reported by Alberga [ALBE67], it was found that 65 percent of dictation errors were phonetically correct, and a further 14 percent almost phonetically correct. Phonetic variations are particularly common in transliterations, as in the example "Tchebysheff" and "Chebyshev." Investigations of errors vary in motivation. Bourne [BOUR77] was concerned with the quality of information retrieval and advised better controls to reduce the proportion of these errors. Litecky and Davis [LITE76] and others [JAME73, LYON74, MORG70] have been interested in error recovery and error correction in compilers. Bell [BELL76] was interested in errors as an indicator of programming competence.

Optical character recognizers and other automatic reading devices introduce similar errors of substitutions, deletions, and insertions, but not reversal. The frequency and type of errors are characteristics of the particular device. Pattern recognition researchers seek to "correct" these errors using "context" [RISE74], either by finding the best match among a repertoire of possible inputs (the problem considered in this section) or by using general linguistic structure.

Many approaches to speech recognition deal with strings of phonemes or symbols representing sounds, and attempt to match a spoken utterance with a directory of known utterances [SAKO79, WHIT76, ERMA80]. Variations in strings here can be due to "noise" where one phoneme is substituted for another similar to it, or phonemes are omitted or inserted, but again not transposed. Note that phonemes vary in their similarity to each other, which, for example, makes it more likely that a "d" sound will be misheard as a "t" sound rather than as an "m" or "f" sound [NEWE73, POTT66]. Another source of variation in phoneme strings is the duration of the spoken word. While words and phrases can be spoken at various speeds, speech to phoneme transducers often work at fixed time intervals, and thus slow speakers produce longer sequences of the same or similar phonemes [VELI70].

Synonyms constitute a very different source of variations. In all languages there are many words which mean more or less the same things. If we consider the following example taken from Roget's thesaurus [ROGE61]:

GUN RIFLE CANNON REVOLVER

we might be tempted to think of synonyms as equivalences, but if we look at the example

HOT WARM COOL COLD

we see that synonymity is not transitive; HOT is not synonymous with COLD. However, when synonyms are controlled by a thesaurus, they are often treated as equivalent (see for example the SPINES thesaurus [UNES76]), often referring to the various alternative words as denoting a particular "concept." Thesauri and synonyms are discussed in most books on information retrieval [SALT68, PAIC77].

A problematic example is abbreviations, especially when used in names. Note that

this has the flavor of a similarity relationship, not an equivalence one. For example, "P." might be an abbreviation for both "PATRICK" and "PETER," but Patrick and Peter are certainly not equivalent! A survey of methods for systematically generating abbreviations while retaining discrimination ability has been given by Bourne and Ford [BOUR61]. Although a full treatment of the handling of abbreviations is beyond the scope of this paper, we suggest that an abbreviation denotes a set of strings and thus denotes partial knowledge about the actual string intended. The partial knowledge problem is very close to the problem we are studying here, and the recent pioneering paper by Lipski [LIPS79] is highly recommended.

In all these examples we have been hypothesizing some mechanism for testing whether two strings are similar to each other. Analogous to an equivalence relation, we can define a similarity relation "~" on the set S, such that for r, s, and t in S

(i) $s \sim s$ reflexivity,
(ii) $s \sim t \Rightarrow t \sim s$ symmetry,

but

(iii) $r \sim s$ and $s \sim t \not\Rightarrow r \sim t$ not necessarily transitive.

Our problem now becomes

Similarity Problem

Given s in S, find all t in T such that $s \sim t$.

Now in most examples there is some idea of degree of similarity. There can be one or many typing mistakes; a spelling mistake can be almost right or completely wrong; two spoken utterances can sound very similar or completely different; and even synonyms can have degrees of similarity. Thus we can postulate a similarity function

$$\sigma: S \times S \rightarrow R$$

which for a pair of strings s and t produces a real number $\sigma(s, t)$. This similarity is usually taken to have a value +1.0 for identical objects, and ranges down to 0.0 (or sometimes −1.0) for very different objects. Thus we could solve the similarity problem by finding all strings t such that $\sigma(s, t)$ is greater than some threshold of acceptability, or we could find the N strings, t_1, t_2, ..., t_N such that their $\sigma(s, t_i)$ have the N largest values.

Similarity functions in this form were favored by Alberga [ALBE67] and are very popular in information retrieval [SALT68, PAIC77] and in classification and clustering [CORM71]. The value of +1.0 for an exact match seems to have strong intuitive appeal, and the range of values from −1.0 to +1.0 appears to gain respectability from correlation coefficients and normalized inner products [RAHM68]. For example, Salton gives a similarity function

$$\sum_i \min(v_i, w_i) / \sum_i v_i$$

for the property vectors v and w of two terms. It has the range [0.0, +1.0].

To begin with, we use a difference function

$$d: S \times S \rightarrow R$$

with properties

(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.

It is this triangular inequality which is useful, as seen in Section 3.3. When a difference function satisfies all these properties, we say it is a *metric* [BIRK70]. Thus by using a difference function, we could formulate our problem as finding all the strings t in T which are closer to the search string s than some threshold δ. Alternatively, we could find the N strings t which are closest to s, that is, for which $d(s, t)$ is smallest.

Most string matching problems will of course involve both equivalence and similarity. That is, there is both an equivalence relation on the set of strings, which groups them into equivalence classes, and a similarity function or difference function between strings. Misspellings and mistypings of natural language are of a combined kind. There is an optional variation which is unimportant, for example, the use of spaces for formatting and (perhaps) uppercase letters; and there is variation which must be

classed as error. With this hybrid problem, the similarity must be taken between equivalence classes. Where the similarity function or difference function is between strings, then it should be between canonical forms; this could influence the choice of canonical form.

3.2 Measures of Similarity

How do we assess whether two strings are similar to each other? How do we quantify this similarity or difference?

A very early method for assessing similarity is the Soundex system of Odell and Russell [ODEL18], which reduces all strings to a "Soundex code" of one letter and three digits, declaring as similar all those with the same code. However, the relationship of having the same code is an equivalence relation, but the string matching problem this proposes to solve is a similarity problem. Not suprisingly, the Soundex method and other methods like it can sometimes go very wrong. Yet these approaches can provide significant extra flexibility to systems that use them. The application of the Soundex method in a hospital patient index was recently reported [BRYA76], and a related method has been used successfully in airline reservations [DAVI62].

Let us examine the Soundex method and its shortcomings. The idea is to transform the name into a Soundex code of four characters in such a way that like-sounding names end up as the same four characters. The first character is the first letter of the name. Thereafter numbers are assigned to the letters as follows:

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

Zeros are removed, then runs of the same digit are reduced to a single digit, and finally the code is truncated to one letter followed by three digits. Note that while DICKSON and DIXON are assigned the same code of D25, RODGERS and ROGERS are not assigned the same code. And what of like-sounding names HODGSON and DODGSON?

Related approaches have been taken by Blair [BLAI60] and Davidson [DAVI62]. Both defined rules for reducing a word to a four-letter abbreviation. Davidson, whose application was airline reservations, then appended to the abbreviation of the family name, the letter of the first name. So far these methods are very similar to the Soundex method, but they go further and introduce aspects of similarity. Blair did not allow multiple matches, and if they occurred, used longer abbreviations to resolve the ambiguity. He thus found the best match, provided that it was close enough. Davidson, by contrast, allowed multiple matches but insisted on finding at least one match by approximately matching the abbreviations looking for the longest subsequences of characters in common.

3.2.1 *The Damerau–Levenshtein Metric*

Damerau [DAME64] tackled the problem of misspellings directly, concentrating on the most common errors—namely, single omissions, insertions, substitutions, and reversals. He used a special routine for checking to see if the two given strings differed in these respects. This work stands out as an excellent early work: the author has analyzed the problem clearly, and made his solution fit the problem. Damerau's algorithm has since been used by Morgan [MORG70].

Damerau had only considered strings in which a single change had occurred. The idea can be extended to consider a sequence of changes of substitutions, deletions, insertions, and possibly reversals. By using sequences of such operations any string can be transformed into any other string. We can take the smallest number of operations required to change one string into another as the measure of the difference between them. Given two arbitrary strings, how do we find this difference measure?

Once the problem has been formulated as an optimization problem, standard optimization techniques can be applied. In 1974 Wagner and Fischer [WAGN74a] published a dynamic programming method. To motivate this method, consider the example of ROGERS and HODGE. Assume that somehow you have found the best matches for all the substrings ROGER and HODG

(with difference 4), ROGER and HODGE (with difference 3), and ROGERS and HODG (with difference 5), and you are about to consider the best match for ROGERS and HODGE. If the last two characters are to be matched, then the score will be 5, 4 from the ROGER/HODG match and 1 for the mismatch of S and E. If the S will be unmatched at the end of ROGERS and treated as an insertion/omission, then the score will be 4, 3 from the ROGER/HODGE match and again 1 from the insertion/omission. If the E at the end of HODGE is treated as an insertion/omission then the score will be 6. Thus the best match of ROGERS and HODGE is 4, the smallest of these three alternatives.

Generalizing the idea of this example leads to the dynamic programming method. A function $f(i, j)$ is calculated iteratively using the recurrence relations below: $f(i, j)$ is the string difference for the best match of substrings $s_1 s_2 s_3 \cdots s_i$ and $t_1 t_2 t_3 \cdots t_j$.

$$f(0, 0) = 0$$

$$f(i, j) = \min[f(i-1, j) + 1,\ f(i, j-1) + 1,\ f(i-1, j-1) + d(s_i, t_j)]$$

where

$$d(s_i, t_j) = 0 \quad \text{if } s_i = t_j$$
$$= 1 \quad \text{otherwise.}$$

Here we assume insertion, omission, and substitution are each assessed a "penalty" of 1. This method can be represented as a problem of finding the shortest path in a graph, as is shown in the example of Figure 2. It does not, however, take into account reversals of adjacent characters.

This basic method can be extended in several directions, Lowrance and Wagner [LOWR75] have given an extension to allow general reversals of order. Transposition of adjacent characters is a special case, and the recurrence relation above is quite easily extended to cope with this, by adding to the minimization the term

$$f(i-2, j-2) + d(s_{i-1}, t_j) + d(s_i, t_{j-1}),$$

which allows for the transposed neighbors that do not match exactly. It is clearly also possible to allow multiple character matches, for example CKS and X, but no work known to us makes this extension. Such an extension would be very necessary for comparisons of transliterations, where multiple characters in one language frequently represent one sound or letter in another language.

Another direction of generalization is to allow for substitutions and even insertions and deletions to have different weights, as a function of the character or characters concerned. Thus, for example, d(i, y) could be small while d(i, f) could be large. No table of letter similarities has been pub-

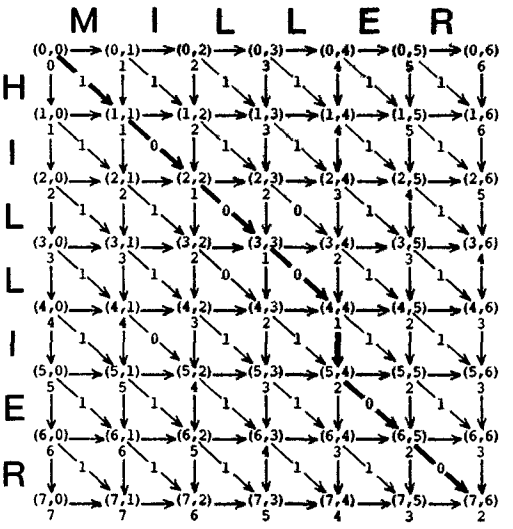

FIGURE 2. (a) Example of the comparison of two strings. The two strings are shown along the top and down the side. Each node of the graph is labeled, (i, j) as appropriate, and below the label is shown the value of $f(i, j)$ for that node. The weights along the diagonal edges are the $d(s_i, t_j)$ values, and along the horizontal and vertical edges they are the penalty values, here set to 1. (b) The best match occurs with a difference of 2, the value of $f(7, 6)$, and the manner of this best match can be deduced from the shortest path, which is drawn in heavy lines.

lished as far as is known, but a table of phoneme similarities was given by Newell et al. [NEWE73]. Instead of modeling phonetic similarities, the difference function could model miskeying by taking into account adjacency on the keyboard—for example, an "a" is often mistyped as an "s."

This distance function and its dynamic programming solution were in fact developed much earlier in the Soviet Union within the fields of coding theory [LEVE66] and speech recognition [VINT68, VELI70].

The primary objective in speech recognition is to compensate for different speeds of speaking and thus stretch or compress the string of phonemes in order to find a best match. This is often called "elastic matching" [DOWL77, SAKO79, WHIT76]. In addition to having the difference between phonemes variable, a penalty can also be introduced for "off-diagonal" matching, to encourage linear matching but still allow elastic matching [ALBE67].

The string difference of Wagner and Fischer satisfies the triangular inequality and thus is a metric. The definition of the difference as the minimum number of changes required to convert one string into the other establishes the triangular inequality. All the variations discussed above also form metrics, although it is important that when nonequal character differences are used, these character differences themselves form a metric. We refer to all distance functions in this general class as Damerau–Levenshtein metrics, after the two pioneering authors in the field [DAME64, LEVE66].

The dynamic programming method takes on the order of n^2 operations to produce its best match where n is the length of the strings being matched. Wong and Chandra [WONG76] have analyzed this in detail, showing that it is the best possible unless special operations are used. As seen below, methods can be derived which are faster in some cases, but these use special methods. The order n^2 processing time is not unduly prohibitive, and one of the authors has used the method in near-real-time speech recognition [DOWL77]. The Damerau algorithm [DAME64, MORG70], which checks just for single errors, is of order n.

One of the by-products of finding the best match between two strings by the Wagner and Fischer method is that it also yields the longest common subsequence. We could also work in the opposite direction: find the longest common subsequence first and then from this compute the difference. A number of techniques other than the dynamic programming method have been published [HUNT77]. These methods have best cases with better than n^2 complexity. Aho, Hirschberg, and Ullman [AHO76] have derived complexity bounds for the longest common subsequence problem and have shown that alphabet size is important. For finite alphabets (as in our problem) an improvement on the n^2 limit should be possible. Heckel [HECK78] has given a method for comparing files which is similar to the methods based on longest common subsequences, but highlights subsequences which have been moved as a body. In some applications, particularly file comparisons, this may be thought to model the real differences and similarities between the two strings more closely.

3.2.2 Similarity as Probability

Another approach to string matching and similarity is through probabilities and likelihoods. This approach has been taken by Fu for error-correcting syntax analysis [FU76]. He follows the conventions of communications theory using conditional probabilities [BACO73, PETE61] to model the production of errors, but there are problems with this approach. We present an alternative formulation.

Let us investigate the joint event {s and t} that string t is "correct" while string s is the observed string. We compute the probability of this event $P\{s \text{ and } t\}$. To do this, let us imagine a generation process which jointly produces s and t from left to right. After this process has created the first i characters of s and the first j characters of t, we can postulate the generation of the next character of s or t or both, with the possible events being (where e is the empty string)

{x and e} = the next character of s is x, and no character of t is generated;

{e and y} = no character of s is generated, and the next character of t is y;

{x and y} = the next character of s is x, and the next character of t is y.

These events exhaust the possibilities, and thus

$$\sum_x \sum_y P\{\text{x and y}\} = 1$$

where we sum over the alphabet including the possibility that x or y is the empty symbol e. Notice that in this generation model we have avoided cause and effect as embodied in conditional probabilities, because of the difficulty of postulating a cause for inserted characters.

With this model of the joint generation of s and t, we can compute a probability for any matching of s and t as the product of the probabilities of the individual generating events. We can compute the best match as the most probable (most likely) matching using our dynamic programming algorithm, recasting the recurrence relations as

$$q(0, 0) = 1$$

$$\begin{aligned} q(i, j) = \max[&q(i-1, j)P\{s_i \text{ and } e\} \\ &q(i, j-1)P\{e \text{ and } t_j\}, \\ &q(i-1, j-1)P\{s_i \text{ and } t_j\}]. \end{aligned}$$

Note that it is the most probable matching that we are finding, so $q(n, m)$ is not $P\{s$ and $t\}$ but $P\{s$ and t and $M\}$ where M is the best match between s and t. If we take logarithms of these recurrence relations and suitably adjust signs, setting

$$f = -\log q,$$

$$\begin{aligned} D &= -\log P\{\text{x and } e\} \\ &= -\log P\{e \text{ and y}\}, \end{aligned}$$

$$d(\text{x, y}) = -\log P(\text{x and y}),$$

we obtain the earlier recurrence relations for differences. However, now the weights, the logarithms of the probabilities, must satisfy certain constraints.

To find $P\{s$ and $t\}$, we must sum over all possible matchings. This can be done iteratively by computing the function

$$\begin{aligned} Q(0, 0) &= 1, \\ Q(i, j) &= Q(i-1, j)P\{s_i \text{ and } e\} \\ &\quad + Q(i, j-1)P\{e \text{ and } t_j\} \\ &\quad + Q(i-1, j-1)P\{s_i \text{ and } t_j\}, \\ P\{s \text{ and } t\} &= Q(n, m). \end{aligned}$$

The similarity to the earlier dynamic programming recurrences is remarkable, although this computation has nothing to do with dynamic programming. To choose the best matching string t, we simply choose the t such that $P\{s$ and $t\}$ is largest. $P\{s$ and $t\}$ is a true similarity function, satisfying the property

$$0 \leq P\{s \text{ and } t\} \leq 1,$$

and generally being close to zero.

In this model the various $P\{$x and y$\}$ can be estimated experimentally by observing errors. Such observations have been made for phonemes [NEWE73] but not for keying errors, and thus there is a need for studies in this area. The model is very appealing but is open to objections because the generation process could generate any pair of strings (unless some of the $P\{$x and y$\}$ are zero), and in real applications the set T is a comparatively small subset of S. However this case can be modeled using regular grammars, and methods for these are surveyed in Section 4.

3.3 Storing and Retrieving Similar Strings

Our problem is to find approximate matches for a given search string s within a set T of strings which are stored explicitly. We must be able to retrieve a record associated with these approximate matching strings and extract associated information.

The primary consideration is the size of the set to be searched. If the set is very small, then all the strings in the set can be tested in turn to see if they satisfy the search criterion (within a threshold δ, or one of the closest N). Often the set is large, perhaps containing millions of entries, and then something must be done to avoid exhaustive searches.

A secondary consideration is the relative importance of the approximate matching necessary. Suppose the problem requires

an exact match if one exists, and otherwise a best match. If exact matches are common, then it could be that the primary requirement is that exact matches be quick to find, while finding approximate matches need not be that efficient. In many applications we can expect 80 percent or more success for exact matches, following the figures of Bourne [BOUR77] and Litecky and Davis [LITE76]. However, in other applications, such as speech recognition, exact matches are most unlikely, and all storage should be structured for approximate matching.

In his review Alberga [ALBE67] made no mention of these search considerations, but three years later Morgan [MORG70] gave a sound discussion of these issues. Morgan's application was searching symbol tables, so he did not consider the extremely large sets that could be encountered in information systems.

There are two basic approaches to searching large sets for approximate matches. The first is to structure the storage of the set T for efficient exact matching, and then when looking for a near match to generate all the strings similar to the search string and test whether these are in the set. The second approach is to structure the storage of the set T with approximate matching in mind using a partitioning strategy. First we look at exhaustive serial searches in order to establish some basis upon which to judge other methods.

3.3.1 Serial Searches

Let us examine simple serial searches and obtain preliminary quantitative figures. We are going to compare and contrast methods by estimating the number of disk accesses required, using a very naive analysis.

Let $|T|$ be the number of strings that are stored, and let m be the (average) number of strings retrieved per disk access. Then a simple serial scan of the set T requires

$$Q_1 = \frac{|T|}{m} \quad \text{disk accesses.}$$

For example, if we take $|T| = 2{,}000{,}000$ and strings have an average length of 10 bytes and are stored on disk pages of 2K bytes, then $m = 200$ and $Q_1 = 10{,}000$. These example figures are used again in later comparisons.

3.3.2 Generating Alternatives

Given a search string s, we can start by testing to see if s itself is in T and an exact match is possible. If this fails, then we can look for a member or members of T close to s by generating all the elements of S in the neighborhood of s and testing each of these in turn to see if it is in T.

The elements of T need to be stored so that searching for an exact match is fast. The technology of exact matching is highly developed [KNUT73, MART75]. Thus testing for membership of T is easy, and can be coupled with the retrieval of the associated record. Suppose we use B-trees for our indexing [COME79]; following KNUT73 (page 476) there will be approximately

$$1 + \log_{\lceil m/2 \rceil} \frac{|T| + 1}{2}$$

disk accesses per index probe, that is, per member of the neighborhood being tested. This is approximately four disk accesses for $|T| = 2{,}000{,}000$ and $m = 200$.

Now if the alternatives to be tested consist only of a few synonyms, then the neighborhood is small, and this method would be very effective. A more common requirement is the correction of misspellings or mistypings involving insertions, deletions, substitutions, and reversals, as discussed in Section 3.2. The members of the neighborhood could be generated, but the neighborhood is now large. A systematic method for generating all the members of the neighborhood needs to be constructed. Riseman and associates [RISE74] have produced such an algorithm, though no details are known. The algorithm would be worth publishing because the generation of the neighborhoods is a nontrivial combinatorial problem.

These neighborhoods are very large. Consider a string s of length n with symbols drawn from an alphabet A of size k. Allowing for insertions, deletions, substitutions, and reversals of adjacent characters, we find the size of the neighborhood of strings with difference 1 from s is

$$\begin{aligned} N(n, 1) &\leq (n + 1)k + n \\ &\quad + n(k - 1) + (n - 1) \\ &= k(2n + 1) + n - 1. \end{aligned}$$

Equality holds provided no two adjacent characters are the same. The size of the neighborhood of distance 2 from s is

$$N(n, 2) \approx N(n, 1)^2.$$

If we consider testing all strings differing by only one error from a string of length 10, with a 26-letter alphabet, the neighborhood size is 565. If we have to use our index and access a disk page for each of these, we then require four disk accesses per string, or a total of

$$Q_2 = 2260 \text{ disk accesses}$$

which is about 4.5 times better than the exhaustive search case. However, to test for up to two errors, we find that

$$Q_2 \approx 1 \text{ million},$$

which is disastrous.

So, at first assessment, the idea of generating all the strings in the neighborhood seems worthless. But suppose we had some simple test which could be used to eliminate most of the members of the neighborhood before accessing the disk to look for the strings in T. All we need is a test for membership of some set X which covers T.

The only published test known to us is that of Riseman and Hanson [RISE74] and Ullman [ULLM77] discussed below. Their approach is ad hoc, but clearly some idea of well-structured strings for English (say) could be derived, since some combinations of letters simply do not occur in English. Any structural test derived from the words or phrases involved would suffice. Structural tests in the form of grammars would provide a very convenient method [GRIE71, HOPC66]. It has been claimed that over 40 percent of possible consecutive letter pairs do not occur in English (Sitar, quoted in RISE74), which suggests that a sensitive test should be possible. Riseman and Hanson review a number of structural tests which are not based on grammars but on checking for the occurrence of sequences of letters within a word or the occurrence of particular letters at particular positions in the words. Their best test can detect simple errors with approximately 99 percent accuracy, but this is only on small vocabularies and is expensive in storage. While Riseman's methods, and those derived from him [ULLM77], may not be ideal for the large sets of strings that concern us here, they do indicate what should be possible. A quick test should at least be able to reduce by an order of magnitude the number of disk accesses required and thus make matching to within a single error by generating alternatives a viable method.

3.3.3 Set Partitioning and Cluster Hierarchies

In the section on exhaustive serial searches, the critical factor was the size of the set T. If we could partition T into subsets T_1, T_2, ..., and select only a few of these subsets for exhaustive searching, we should be able to reduce our number of disk accesses considerably.

Morgan [MORG70] and Szanzer [SZAN69] have suggested partitioning by string length. Assuming that we are only looking for strings differing by only one error, then we need only search strings differing from the length of the search string by 1. This idea will not have a very significant impact but may improve the search cost two- to fivefold. This is because strings in applications, such as name indexes, do not vary much in length and have a very nonuniform distribution in length.

Another idea would be to partition the set on the first letter. There may be no attempt made to compensate for errors in the initial letter, for example, Muth and Tharp [MUTH77]; or the errors in the first letter may be searched for in some separate operation, as proposed by Szanzer [SZAN73].

Ideally any partitioning strategy should produce sets of the same size, and the search efficiency is sensitive to departures from a uniform distribution. The average number of disk accesses for exact matching, assuming each stored string is equally likely, is given by

$$Q_3 = \sum_i \left(\begin{array}{l} \text{Number of disk accesses to search } T_i \\ * \\ \text{Probability of search string being in } T_i \end{array} \right)$$

$$= \sum_i \frac{|T_i|}{m} * \frac{|T_i|}{|T|}$$

$$= \sum_i \frac{|T_i|^2}{m|T|}.$$

If we use some simple rule based on string length or leading letters, we inevitably come up against the uneven distribution of real data. Moreover, the use of data is very uneven. Knuth [KNUT73, pp. 396–398] has a stimulating discussion of this; a useful rule for us here is the 80–20 law, that 80 percent of the activity appears in 20 percent of the file.

Morgan [MORG70] also suggested a technique for partitioning the set based on the first two letters

$$T_{XY} = \{t \text{ in } T \text{ such that } t \text{ begins XY}\}.$$

For the usual 26-letter alphabet, this gives 676 subsets. Of course, following our earlier remarks that 40 percent of pairs do not occur, many of these subsets will be empty. When searching for a string beginning PQ, for example, we would only need to search the 77 subsets where at least one of the defining letters was a P or a Q (that is, subset PQ for an exact match on the first two letters, P? for substitution or deletion of Q or a reversal of the second and third letters, ?Q for substitution of P, QP for reversal of PQ, and Q? for deletion of P). Making the most favorable uniformity assumptions, this means at most a ninefold speedup. Extending the idea to the first three letters, we can hope for as much as a 200-fold speedup on single errors, but the number of partitions is beginning to get out of hand. We could do some hashing however, to randomize and superimpose subsets as Morgan suggested.

So far these methods have not appeared very effective. Though the exact search behavior is not known, they appear to have a search time proportional to $|T|$, since the partitioning strategy is fixed and independent of $|T|$. What we would like ideally is a search behavior of order $\log|T|$, as is found for exact matches.

An interesting search method has been suggested by Shapiro [SHAP77] in the context of general pattern recognition. The method consists of imposing a linear ordering on all the elements of the set of patterns to be searched, finding a most likely match by using binary searching to find a candidate match, and then searching in that neighborhood for a best match. The linear ordering is determined by the difference from some reference point, and it is this difference which gives the means for computing bounds that keeps the search to the neighborhood of the first candidate match. Because the string difference metric does not provide fine discrimination, the method is unlikely to work well for strings.

Knuth [KNUT73] has suggested a method based on the observation that strings differing by a single error must match exactly in either the first or the second half. He does no more than hint at a method of exploiting this observation, but one method might be the following. Index the set using both the first and last halves—the first and last $\lceil n/2 \rceil - 1$ characters—so that the central two characters are omitted from an even-length string to allow for central reversals. For retrieval try the first and last halves, both the $\lfloor n/2 \rfloor$ and the $\lfloor n/2 \rfloor - 1$ first or last characters, so as to allow for insertions and deletions. Thus we retrieve two sets of strings which must be serially searched to find any actual matches to within a single error. (Notation: $\lfloor x \rfloor$ is the greatest integer less than or equal to x, and $\lceil y \rceil$ is the smallest integer greater than or equal to y.) This method will be sensitive to the actual distribution of the strings but does seem very promising. No theoretical or empirical results concerning its effectiveness are known.

$\log|T|$ search behavior is obtained in tree-structured searches. Muth and Tharp's method [MUTH77] forms a character tree, but since they then backtrack up the tree on encountering an error, much of the advantage of the tree is lost. The only substantive gain they do get is by partitioning on the first character, but they do not attempt to correct errors in that first character place.

A general tree-structured approach has been suggested by Salton and his associates for use in information retrieval [SALT68, SALT78]. The method uses the similarity distance function as its basis for partitioning, dividing the set into "clusters" of strings which are simlar to one another. That is, strings within the same subset T_i have $d(s, t)$ small, and strings in different subsets have $d(s, t)$ large. The automatic formation of subsets with these characteristics is known variously as clustering or

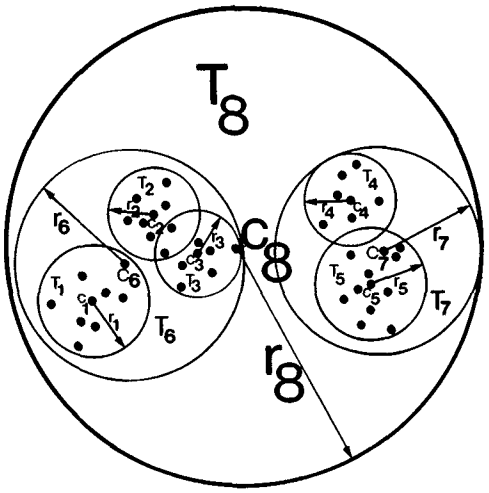

FIGURE 3. A hierarchy of clusters, with T_1, T_2, and T_3 contained in T_6, T_4 and T_5 contained in T_7, and T_6 and T_7 contained in T_8, the whole of T.

classification (see, for example, the review by Cormack [CORM71]). This method shows promise of approaching the $\log|T|$ goal and is described in some detail.

A hierarchy of clusters is formed, and each cluster T_i is described by a center c_i and a radius r_i:

$$T_i = \{t : d(t, c_i) \leq r_i \text{ and } t \text{ in } T\}.$$

Clusters at higher levels contain clusters below them, and clusters at a particular level could overlap. Figure 3 illustrates this.

To search for all the strings within δ of the search string s, we start at the highest level and search within a cluster $T_i(c_i, r_i)$ if and only if $d(s, c_i) \leq r_i + \delta$. This guarantees finding *all* t in T with $d(s, t) \leq \delta$, provided that the difference function satisfies the triangular inequality.

To search for the best match (or N best matches), we use what is basically a branch-and-bound technique [HALL71]. For any subset $T_i(c_i, r_i)$ we have the bounds

$$d(c_i, s) - r_i \leq d(s, t) \leq d(c_i, s) + r_i$$

for all t in T; this follows from the triangular inequality. We search in the most promising subset (the one with the minimum value of $d(c_i, s) - r_i$) to find the best candidate match t^* and then search in the next best remaining subset provided that it could possibly yield an improvement, that is, provided that $d(c_i, s) - r_i \leq d(s, t^*)$. Thus successively better matches t^* are found until it is evident that no further searching is necessary. Both variations of the search use the triangular inequality to guarantee the search algorithms.

Alternatively, the decisions to stop searching can be based upon empirically determined parameters, which is the approach taken by Salton.

Let us illustrate the search process with a small string searching example. Figure 4 shows the differences for a set of 15 names, while Figure 5 shows the set divided into five clusters, including one miscellaneous cluster to accommodate the two strings which did not happily fit into any other cluster. Figure 6 then shows two searches for all matches within a given tolerance. Figure 7 shows a search for a best match. Note that the miscellaneous set is always searched.

It is necessary to define insertion and deletion strategies, as well as the search strategy. Salton and Wong describe an insertion method which they claim provides a good basic clustering method, but they do not describe a deletion method. They liken their approach, quite correctly, to B-trees [KNUT73, COME79] without pursuing the analogy.

Assuming a balanced tree, or reasonably uniform tree, the average depth of the tree will be of order $\log|T|$. But the search methods reported above do not only search a single path from root to leaf, they also try other branches if these are found to be necessary. These other paths could completely destroy the potential $\log|T|$ behavior, and in the worst case lead to an exhaustive search. Salton and Wong suggest limiting the extra searching at each level in the hierarchy to some small number of branches p. Suppose that the tree branches k ways at each level, and at each level we search p of these branches. Clearly p is less than k. We must search p branches at each of the $\log_k|T|$ levels in the tree, and thus must search

$$p^{\log_k|T|} = |T|^{\log_k p}$$

strings at the lowest level. If, for example, $p = \sqrt{k}$, then $\log_k p = \frac{1}{2}$, and we must search

	JOHNSON	ALWOOD	FENLON	BUBENKO	ROGERS	SENKO	ROGET	GOODWIN	WOODRUM	HINTON	HODGES	SLOANE	RODGERS	DODGSON	GOODRUM
JOHNSON	-	6	4	6	6	5	6	5	6	4	6	7	6	3	6
ALWOOD	6	-	5	7	6	5	6	6	5	5	6	5	7	6	6
FENLON	4	5	-	5	6	3	6	6	7	3	6	6	7	5	7
BUBENKO	6	7	5	-	6	3	6	7	7	6	7	6	7	7	7
ROGERS	6	6	6	6	-	5	2	6	5'	6	3	6	1	5	5
SENKO	5	5	3	3	5	-	5	7	7	4	6	5	7	6	7
ROGET	6	6	6	6	2	5	-	6	6	6	3	5	3	5	6
GOODWIN	5	6	6	7	6	7	6	-	4	6	5	6	6	6	3
WOODRUM	6	5	7	7	5	7	6	4	-	7	5	6	6	6	1
HINTON	4	5	3	6	6	4	6	6	7	-	5	6	7	5	7
HODGES	6	6	6	7	3	6	3	5	5	5	-	5	2	4	5
SLOANE	7	5	6	6	6	5	5	6	6	6	5	-	6	7	6
RODGERS	6	7	7	7	1	7	3	6	6	7	2	6	-	4	6
DODGSON	3	6	5	7	5	6	5	6	6	5	4	7	4	-	6
GOODRUM	6	6	7	7	5	7	6	3	1	7	5	6	6	6	-

FIGURE 4. Difference matrix for a set of 15 names. The differences shown are the best match differences, which have been found using the simple dynamic programming approach.

Cluster	Center c_i, Radius r_i	Members
T_1	GOODRUM, 3	WOODRUM, GOODRUM, GOODWIN
T_2	RODGERS, 3	ROGERS, RODGERS, ROGET, HODGES
T_3	SENKO, 4	FENLON, HINTON, SENKO, BUBENKO
T_4	JOHNSON, 3	JOHNSON, DODGSON
T_5	MISCELLANEOUS	ALLWOOD, SLOANE

FIGURE 5. Clusters formed from the names of Figure 4.

$\sqrt{|T|}$ strings. While we do not have a $\log|T|$ law, we have certainly done better than a linear search.

Consider our example where $|T| = 2{,}000{,}000$. Suppose our index branches 100 ways at each level, and the index is arranged with the specification that all 100 ways stored are on one disk page, as in B-trees. Suppose further that we only search at most ten branches at any level. Then our law above says we must search some 1400 strings, or seven pages at 200 strings per page. Including the index, this requires around ten disk accesses in total.

This approach will only be good if the clustering is good and ensures that only a few of the branches need searching. And this clustering must be preserved under insertion and deletion. There is a real need for some research here to determine what clustering property is necessary to ensure a good search behavior, and what insertion and deletion algorithms will guarantee preservation of this property.

At the moment too little is known about this cluster hierarchy method for the use of it to be anything better than a gamble. But if somebody did gamble and validate the method empirically, that would be worth reporting.

4. ERROR CORRECTION USING SYNTACTIC STRUCTURE

Instead of the set of strings being stored explicitly, as has been assumed up to this point, the set could be defined by a collection of structural rules such as a grammar.

		Search 1	Search 2
Search string s		GOODGE	FENKON
Tolerance δ		1	2
Distance of search string from center of cluster	T_1	3	7
	T_2	4	7
	T_3	6	2
	T_4	6	4
Clusters requiring further searching		T_1, T_2 T_5	T_3, T_4 T_5
Strings found		None	FENLON SENKO

FIGURE 6. Two searches of the cluster hierarchy of Figure 5.

If a string fails to conform to the rules, it is in "error." By suitable use of the rules, the error can be "corrected" and the string identified.

Riseman and Hanson [RISE74] describe a set of rules based upon the occurrence or nonoccurrence of particular sequences of letters. The sequences can be either adjacent letters occurring at any point in the string or they can be letters occurring at fixed points in the string. The application is error correction following optical character recognition. Riseman and Hanson acquire their rules directly from the set of strings that are permitted. From the particular rules that are violated, they deduce a correction which when applied will make the string conform to the rules. However a proportion of the strings in error (as high as 30 percent) are "uncorrectable" because no correction can be readily deduced from the violated rules.

All other methods of structural error correction are based upon a grammar. Following Hopcroft and Ullman, we use the notation $G = (V_N, V_T, P, S)$ for a grammar, where V_N are the nonterminals, V_T are the terminals, P the productions, and S the start symbol in V_N [HOPC69]. In error-correcting parsing, instead of rejecting a string found to be in error during parsing, the string is corrected to that member of $L(G)$, the language generated by the grammar G, which is closest to the given string.

A halfway stage to full error correction is

HOODGUS

(a)

Cluster	$d(s, c_i)$	$d(s, c_i) - \delta_i$
T_1	3	0
T_2	4	1
T_3	7	3
T_4	5	2

(b)

Step	Search	Strings found	Differences
1	T_5	SLOANE	6
2	T_1	GOODRUM WOODRUM	3
3	T_2	HODGES	2
4	T_4	Know that we cannot find a closer match, but could find an equal match. Continue only if all best matches required, then STOP.	

(c)

HODGES, difference 2.

(d)

FIGURE 7. Finding a best matching string: (a) search string; (b) results of comparisons with cluster centers; (c) steps in search for the closest match; (d) result.

error detection and error recovery [EGGE72, GRAH75, GRIE71, LITE76, MICK78, SMIT70, WILC76]. In compilers an error should be detected as early as possible, and a meaningful error message output which will help the user to correct the error himself. Then the compiler should recover and keep on parsing so as to subject the complete program to syntax checking. In order to recover, the compiler must compensate for the error and thus make some "correction," but it does not necessarily produce an alternative complete program which is correct and which could be executed.

Error-correction parsing goes the whole way and produces a program string which is valid and can be executed. An illustration at this point may help. Morgan reported an error-correcting system used with the CUPL compiler at Cornell University [MORG70]. The card deck

```
/JOB 2065 MORGAN, H 10S 30P
/CUP6
 READ ROWSUB, COLSUB, NOCOLS
 MATSUB =
   ROWSUB + COLSUB * NOCOLS
 WRITE MATSBU
 STPO
*DATA
 3.0 4, 5
/ENDJOB
```

contains a keypunching error in one job control card and two errors in the program. Their system would have corrected these errors without requiring a rerun. Of course, such corrections may be wrong, and in fact dangerous in that they do not agree with the programmer's original intention, but Morgan's system was used favorably.

Error correction during language analysis has been widely reported, starting with some very early ideas by Irons [IRON63] and Freeman [FREE63]. There was then a lapse of several years before Morgan's work and an ad hoc approach by James and Partridge [JAME73], and then error correction in languages was put on a sound theoretical footing. A theoretical analysis had been given by Hopcroft and Ullman [HOPC66] in 1966, and this was followed by full error-correction techniques for most classes of languages being reported in the literature during the early 1970s.

Regular languages were treated by Wagner in 1974 [WAGN74b] using a dynamic programming approach similar to the one developed with Fischer for simple string matching [WAGN74a]. A function $f(A, i)$ is defined. This measures the best match between the first i symbols of the input string and the strings generated by starting with the start symbol S and ending with the nonterminal A. The recurrence formulas for computing $f(A, i)$ follow directly from the grammar. Terminal productions $A \to a$ are rewritten as $A \to a\not\#$, where $\not\# \notin V_N \cup V_T$. The recurrence relations are

$$f(S, 0) = 0,$$

$$f(A, i) = \min\{ \min_{\text{all } B\ B \to aA} [f(B, j-1) + d(s_i, a)], f(A, i-1) + 1, \min_{\text{all } B\ B \to aA} [f(B, i) + 1]\}$$

$$f(\not\#, n) = \text{difference of the best match,}$$

where again following the notation of Section 3.2

$$d(s_i, a) = 0 \quad \text{if} \quad s_i = a$$
$$= 1 \quad \text{otherwise,}$$

and the insertion/omission penalty is 1.

This algorithm for regular grammars has complexity which is linear in the length of the input string, but of course depends upon some function of the size of the grammar. Independently, and at about the same time, Hall and Dowling discovered the same algorithm and applied it to speech recognition [DOWL74]. Figure 8 gives a matching graph for regular grammars; there is an obvious similarity here with Figure 2.

Context-free languages, because of their importance for programming, have received considerable attention [AHO72, FU76, FUNG75, LEVY75, LYON74, TANA78, WAGN72]. Aho and Peterson [AHO72] add error productions to their grammar, and select derivations with fewest error productions using an Earley parser [EARL70]. Lyons [LYON74] by contrast tackles errors directly using dynamic programming principles, without the need to resort to error productions. He also uses the Earley parser. Levy concentrates on local corrections to

gain parsing speed, while Tanaka and Fu use a Cocke-Younger-Kasami parser and Chomsky normal form. Teitlebaum, by contrast, takes an algebraic approach to language analysis, modeling the error-production process by a weighted sequential transducer, to produce an elegant method for error correction. While recognition of a context-free language theoretically can be done as fast as matrix multiplication ($O(n^{2.61})$ [PAN79]), normal parsing methods are of n^3 complexity [HOPC69, EARL70], and by suitable and careful choice of method, the error-correction parsers can also have complexity n^3 [LYON74, TEIT76, WAGN72]. Fung and Fu [FUNG75] consider only substitution errors, allowing for different probabilities for different substitutions of characters, and thus obtain probabilities for transformations from one string into another. They give an error-correcting parser which returns the string with highest probability.

Using probabilities, one can take into account the frequency with which particular linguistic constructs are used. This is done by Fu, who adds probabilities of use to each production (a stochastic grammar) and adopts the Aho and Peterson algorithm to handle these probabilities, selecting the most likely derivation for a given string. Clearly Fu's method can easily be applied to regular grammars. A regular grammar could be arranged to generate a particular set of strings and thus overcome the objections to probabilistic string similarity at the end of Section 3.2.

Best-match recognition in context-free languages has also been studied in speech recognition. A number of systems are surveyed by Erman et al. [ERMA80]. The process of recognizing connected speech is represented as problem solving using a sequence of "knowledge systems," where the selection of a best match is heuristic. This process is clearly equivalent to a bottom-up parse, and the discussion of the speech recognition methods surveyed could be complemented with a comparison based on language analysis.

Correction for context-sensitive languages has been investigated by Tanaka and Fu [TANA78]. They use the Cocke-Younger-Kasami parser, and although they do not actually use dynamic programming, they formulate their solution in an equivalent way.

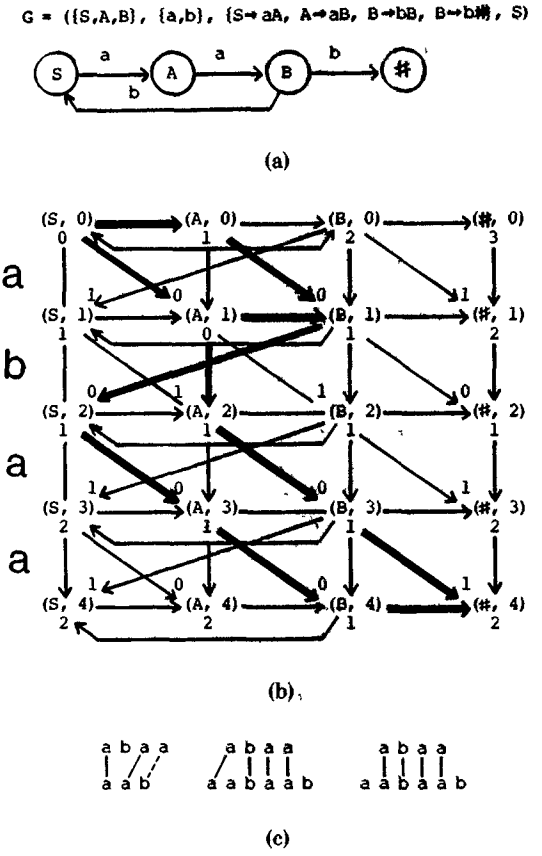

FIGURE 8. Example of error correction in a regular grammar. A graphical representation is used; the regular language is represented by a state-transition graph in (a) and this is combined with the matching string "*abaa*" in (b) to give a graph in which the best match problem has again become a shortest path problem. The edge weights are the mismatch penalties, and all horizontal and vertical paths have a weight of 1. The diagonal edges have $d(s_i, a) = 0$ if $s_i = a$, and 1 otherwise. Each node is labeled with its "coordinates," and the smallest mismatch from $(S, 0)$ to that node. The manner of the three best matches, shown by the heavy lines, is given in (c).

5. SUMMARY

When tackling a string matching problem where retrieval is to be achieved even with nonexact matches, it is important to ascertain the source of the variation leading to the need for nonexact matching.

Variations could be legitimate, as in the use of alternative spellings or formatting

characters. This leads to equivalence classes and the methods reviewed in Section 2. The methods available are highly satisfactory, resting on conventional exact-matching technology. Canonical forms of equivalence classes should be used where possible.

However variation could be due to errors and like processes. Examples are typing mistakes, spelling mistakes due to mishearing, synonyms, and the whole range of encoding problems associated with speech recognition. This leads us into the similarity problems reviewed in Section 3. First it is necessary to construct a difference function which correctly models the source of variation; we have seen a number of powerful difference functions based on dynamic programming and on probabilities.

The real difficulties begin when we must search a large set of strings for an approximate match, guided by our difference function. While there are a number of ideas about how to approach this problem, there are no well-established or generally applicable methods. Most methods described are aimed at compensating for mistyping errors. There is a great need here for research to give the currently proposed methods a firm theoretical foundation and to generate alternative methods.

One class of string matching problems is concerned with finding the best match in a set of strings defined by a grammar. This error-correcting problem is briefly reviewed in Section 4.

Finally, it is interesting to note that during the preparation of this review, a spelling correction program for microcomputer systems was announced in the popular computer press [COMP80]. The program can correct some spelling mistakes it finds in its input text using a 25,000-word list supplied by Oxford University Press and draws attention to those it can recognize but not rectify. After the complete text has been scanned, an operation which processes about 60 words per minute, a list of alterations with references is given. With this facility now available on microcomputer systems, it should not be too long before some of the techniques outlined in this paper do find their way into large commercial information retrieval systems.

ACKNOWLEDGMENTS

The authors would like to thank their employers for providing facilities for this paper. It is a pleasure, too, to thank the Associate Editor, Bruce Weide, and the referees for their help in preparing this manuscript, and in (dare we say) spotting numerous spelling mistakes.

REFERENCES

AHO72 AHO, A. V., AND PETERSON, T. G. "A minimum distance error-correcting parser for context-free languages." *SIAM J. Comput.* 1 (Dec. 1972), 305–312.

AHO75 AHO, A. V., AND CORASICK, M. J. "Fast pattern matching: An aid to bibliographic search," *Commun. ACM* **18,** 6 (June 1975), 333–340.

AHO76 AHO, A. V., HIRSCHBERG, D. S., AND ULLMAN, J. D. "Bounds on the complexity of the longest common subsequence problem, *J. ACM* **23,** 1 (Jan. 1976), 1–12.

ALBE67 ALBERGA, C. N. "String similarity and misspellings," *Commun. ACM* **10,** 5 (May 1967), 302–313.

BACO73 BACON, M. D., AND BULL, G. M. *Data transmission,* Macdonald and Jane's, London, 1973.

BELL76 BELL, D. "Programmer selection and programming errors," *Comput. J.* **19,** 3 (1976), 202–206.

BIRK70 BIRKOFF, G., AND MACLEAN, S. *A survey of modern algebra,* Macmillan, New York, 1970.

BLAI60 BLAIR, C. R. "A program for correcting spelling errors," *Inf. Control* **3** (1960), 60–67.

BOUR61 BOURNE, C. P., AND FORD, D. J. "A study of methods for systemtically abbreviating English words and names," *J. ACM* **8,** 4 (Oct. 1961), 538–552.

BOUR77 BOURNE, C. P. "Frequency and impact of spelling errors in bibiographic data bases," *Inf. Process. Manage.* **13,** 1 (1977), 1–12.

BOYE77 BOYER, R. S., AND MOORE, J. S."A fast string searching algorithm." *Commun. ACM* **20,** 10 (Oct. 1977), 762–772.

BRYA76 BRYANT, J. R., AND FENLON, S. M. "The design and implementation of an on-line index," *Database Technol.* (ON-LINE, 1976).

COME79 COMER, D. "The ubiquitous B-tree," *Comput. Surv.* **11,** 1 (June 1979), 121–138.

COMP80 "Spelling correction program for micros," *Comput. Weekly* (July 3, 1980), 7.

CORM71 CORMACK, R. M. "A review of classification," *Royal Statistical Soc. J.* **134** (Series A, 1971), 321–367.

DAME64 DAMERAU, F. J. "A technique for computer detection and correction of spelling errors," *Commun. ACM* **7,** 3 (March 1964), 171–176.

DAVI62 DAVIDSON, L. "Retrieval of misspelled

names in an airline's passenger record system," *Commun. ACM* **5**, 3 (March 1962), 169–171.

DOWL74 DOWLING, G. R., AND HALL, P. A. V. "Elastic template matching in speech recognition, using linguistic information," in *2nd Int. Joint Conf. Pattern Recognition*, Copenhagen, Aug. 1974, pp. 249–250 (available from IEEE.).

DOWL77 DOWLING, G. R. "Automatic segmentation of continuous speech," Ph.D. Dissertation, The City University, London, 1977.

EARL70 EARLEY, J. "An efficient context-free parsing algorithm," *Commun. ACM* **13**, 2 (Feb. 1970), 94–102.

EGGE72 EGGERS, B. "Error reporting, error treatment and error correction in ALGOL translation, part II," in *2nd Annual Meeting, G.I.*, Karlsruhe, Oct. 1972.

ERMA80 ERMAN, L. D., HAYES-ROTH, F., LESSER, V. R., AND REDDY, D. R. "The HEARSAY-II speech understanding system: Integrating knowledge to resolve uncertainty," *Comput. Surv.* **12**, 2 (June 1980), 213–253.

FREE63 FREEMAN, D. "Error correction in CORC: The Cornell computing language," Ph.D. Dissertation, Cornell University, Ithaca, N.Y., 1963.

FU76 FU, K. S. "Error-correcting parsing for syntactic pattern recognition," in *Data-structures, computer graphics and pattern recognition*, Klinger et al. (Eds.) Academic Press, New York, 1976.

FUNG75 FUNG, L. W., AND FU, K. S. "Maximum-likelihood syntactic decoding," *IEEE Trans. Inform. Theory* **IT-21** (July 1975), 423–430.

GALI79 GALIL, Z. "On improving the worst case running time of the Boyer-Moore string matching algorithms," *Commun. ACM* **22**, 9 (Sept. 1979), 505–508.

GIMP76 GIMPEL, J. F. *Algorithms in SNOBOL 4*, Wiley-Interscience, New York, 1976.

GRAH75 GRAHAM, S. L., AND RHODES, S. P. "Practical syntactic error recovery," *Commun. ACM* **18**, 11 (Nov. 1975), 639–650.

GRIE71 GRIES, D. *Compiler construction for digital computers*, Wiley, New York, 1971.

HALL71 HALL, P. A. V. "Branch and bound and beyond," in *Int Joint Conf. Artificial Intelligence*, Imperial College, London, Sept. 1971, pp. 641–650 (available from British Computer Society).

HALL78 HALL, P. A V., AND HUSSEIN, I. "Design of information systems for Arabic," in *Information systems methodology*, ECI 78 Conference, Venice, Springer-Verlag, New York, 1978, pp. 643–663.

HECK78 HECKEL, P. "A technique for isolating differences between files," *Commun. ACM* **21**, 4 (April 1978), 264–268.

HOPC66 HOPCROFT, J. E., AND ULLMAN, J. D. "Error correction for formal languages," Tech. Rep. 52, Princeton Univ., Princeton, N.J., Nov. 1966.

HOPC69 HOPCROFT, J. E., AND ULLMAN, J. D. *Formal languages and their relation to automata*, Addison-Wesley, Reading, Mass., 1969.

HUNT77 HUNT, J. W., AND SZYMANSKI, T. G. "A fast algorithm for computing longest common subsequences," *Commun. ACM* **20**, 5 (May 1977), 350–353.

IRON63 IRONS, E. T. "An error-correcting parser algorithm," *Commun. ACM* **6**, 11 (Nov. 1963), 669–673.

JAME73 JAMES, E. B., AND PARTRIDGE, D. P. "Adaptive correction of program statements," *Commun. ACM* **16**, 1 (Jan. 1973), 27–37.

JENK76 JENKS, R. D. *1976 ACM Conf. Symbolic and Algebraic Computation* (ACM), 1976.

KNUT73 KNUTH, D. E. *Sorting and searching*, Addison-Wesley, Reading, Mass., 1973.

KNUT77 KNUTH, D. E., MORRIS, J. H., AND PRATT, V. R. "Fast pattern matching in strings," *SIAM J. Comput.* **6** (1977), 323–350.

LEVE66 LEVENSHTEIN, V. I. "Binary codes capable of correcting deletions, insertions, and reversals," *Sov. Phys. Dokl.* **10** (Feb. 1966), 707–710.

LEVY75 LEVY, J. P. "Automatic correction of syntax errors in programming languages," *Acta Inf.* **4** (1975), 271–292.

LIPS79 LIPSKI, W., JR. "On semantic issues connected with incomplete information databases" *ACM Trans. Database Syst.* **4**, 3 (Sept. 1979), 262–296.

LITE76 LITECKY, C. R., AND DAVIS, G. B. "A study of errors, error-proneness, and error diagnosis in COBOL," *Commun. ACM* **19**, 1 (Jan. 1976), 33–37.

LOWR75 LOWRANCE, R., AND WAGNER, R. A. "An extension of the string-to-string correction problem," *J. ACM* **22**, 2 (April 1975), 177–183.

LYON74 LYON, G. "Syntax-directed least-errors analysis for context-free languages: A practical approach," *Commun. ACM* **17**, 1 (Jan. 1974), 3–14.

MART75 MARTIN, J. *Computer data base organization*, Prentice-Hall, Englewood Cliffs, N.J., 1975.

MICK78 MICKUNAS, M. D., AND MODRY, J. A. "Automatic error recovery for LR parsers," *Commun. ACM* **21**, 6 (June 1978), 459–465.

MORG70 MORGAN, H. L. "Spelling correction in systems programs," *Commun. ACM* **13**, 2 (Feb. 1970), 90–94.

MUTH77 MUTH, F. E., JR., AND THARP, A. L. "Correcting human error in alphanumeric terminal input," *Inf. Process. Manage.* **13**, 6 (1977), 329–337.

NEWE73 NEWELL, A., BARNETT, J., FORGIE, J. W., GREEN, C., KLATT, D., LICKLIDER, J. C. R., MUNSON, J., REDDY, D. R., AND WOODS, W. A. "Speech understanding

systems. Final report of a study group," in *Artificial intelligence*, North-Holland Elsevier, New York, 1973.

ODEL18 ODELL, M. K., AND RUSSELL, R. C. U.S. Patent nos. 1,261,167 (1918) and 1,435,663 (1922).

PAIC77 PAICE, C. D. *Information retrieval and the computer*, MacDonald and Jane's Computer Monographs, London, 1977.

PAN79 PAN, V. YA. "Field extension and trilinear aggregating, uniting, and canceling for the acceleration of matrix multiplications," in *Proc. 20th Annual Symp. Foundations of Computer Science*, Oct. 1979, pp. 28–38.

PETE61 PETERSON, W. W. *Error-correcting codes*, Wiley, New York, 1961.

POTT66 POTTER, R. K., KOPP, G. A., AND KOPP, H. G. *Visible speech*, Dover, New York, 1966.

RADE76 RADECKI, T. "New approach to the problem of information system effectiveness evaluation," *Inf. Process. Manage.* **12**, 5 (1976), 319–326.

RAHM68 RAHMAN, N. A. *A course in theoretical statistics*, Griffin, London, 1968.

RISE74 RISEMAN, E. M., AND HANSON, A. R. "A contextual post-processing system for error-correction using binary *n*-grams," *IEEE Trans. Comput.* **C-23**, 5 (1974), 480–493.

RIVE77 RIVEST, R. L. "On the worst-cast behavior of string-searching algorithms," *SIAM J. Comput.* **6**, 4 (Dec. 1977), 669–673.

ROGE61 ROGET *The new Roget's thesaurus*, N. Lewis (Ed.), Putnam, 1961.

SAKO79 SAKOE, H. "Two level DP-matching—A dynamic programming based pattern matching algorithm for connected word recognition," *IEEE Trans. Acoust., Speech, Signal Proc.* **ASSP-27**, 6 (Dec. 1979), 588–595.

SALT68 SALTON, G. *Automatic information organization and retrieval*, McGraw-Hill, New York, 1968.

SALT78 SALTON, G., AND WONG, A. "Generation and search of clustered files," *ACM Trans. Database Syst.* **3**, 4 (Dec. 1978), 321–346.

SHAF68 SHAFFER, L. H., AND HARDWICH, J. "Typing performance as a function of text," *Qt. J. Exper. Psychol.* **20**, 4 (1968), 360–369.

SHAP77 SHAPIRO, M. "The choice of reference points in best match file searching," *Commun. ACM* **20**, 5 (May 1977), 339–343.

SMIT70 SMITH, W. B. "Error detection in formal languages," *J. Comput. Syst. Sci*, 4 (1970), 385–405.

SLIN79 SLINN, C. "Retrieval mechanisms and Arabic strings," in *5th Saudi Arabian Computer Conf.*, Dhahran, Saudi Arabia, March 1979.

SZAN69 SZANZER, A. J. "Error-correcting methods in natural language processing," in *Information processing 68*, IFIP 1969, pp. 1412–1416.

SZAN73 SZANZER, A. J. "Bracketing technique in elastic matching," *Comput. J.* **16**, 2 (1973), 132–134.

TANA78 TANAKA, E., AND FU, K. S. "Error-correcting parsers for formal languages," *IEEE. Trans. Comput.* **C-27**, 7 (July 1978), 605–616.

TARJ75 TARJAN, R. E. "Efficiency of a good but not linear set union algorithm," *J. ACM* **22**, 2 (April 1975), 215–225.

TEIT76 TEITELBAUM, R. "Minimal distance analysis of syntax errors in computer programs," Ph.D. Dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., 1976.

ULLM77 ULLMAN, J. R. "A binary *n*-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words," *Comput. J.* **20**, 2 (1977), 141–147.

UNES76 UNESCO *SPINES thesaurus*, The UNESCO Press, Paris, 1976.

VELI70 VELICHKO, V. M., AND ZAGARUIKO, N. G. "Automatic recognition of 200 words," *Int. J. Man-Mach. Stud.* **2** (1970), 223–234.

VINT68 VINTSYUK, T. K. "Speech discrimination by dynamic programming," *Kibernetika* **4**, 1 (1968), 81–88 (in Russian); translated in *Cybernetics* **4**, 1, 52–58.

WAGN72 WAGNER, R. A. "An n^3 minimum edit-distance correction algorithm for context-free languages," Tech. Rep., Systems and Information Sci. Dept., Vanderbilt Univ., Nashville, Tenn., 1972.

WAGN74a WAGNER, R. A., AND FISCHER, M. J. "The string-to-string correction problem," *J. ACM* **21**, 1 (Jan 1974), 168–178.

WAGN74b WAGNER, R. A. "Order-*n* correction for regular languages," *Commun. ACM* **17**, 5 (May 1974), 265–268.

WHIT76 WHITE, G. M., AND NEELY, R. B. "Speech recognition experiments with linear prediction, bandpass filtering, and dynamic programming," *IEEE Trans. Acoust., Speech, Signal Proc.* **ASSP-24**, 2 (April 1976), 183–188.

WILC76 WILCOX, T. R., DAVIS, A. M., AND TINDALL, M. H. "The design and implementation of a table-driven interactive diagnostic programming system," *Commun. ACM* **19**, 11 (Nov. 1976), 609–616.

WONG76 WONG, C. K., AND CHANDRA, A. K. "Bounds for the string editing problem," *J. ACM* **23**, 1 (Jan. 1976), 13–16.

RECEIVED JANUARY 1980; FINAL REVISION ACCEPTED AUGUST 1980.