Programming Languages T.A. Standish Editor

Order-*n* Correction for Regular Languages

Robert A. Wagner
Vanderbilt University

A method is presented for calculating a string B, belonging to a given regular language L, which is "nearest" (in number of edit operations) to a given input string α. B is viewed as a reasonable "correction" for the possibly erroneous string α, where α was originally intended to be a string of L. The calculation of B by the method presented requires time proportional to $|\alpha|$, the number of characters in α. The method should find applications in information retrieval, artificial intelligence, and spelling correction systems.

Key Words and Phrases: error correction, regular languages, regular events, finite state automata, compiler error recovery, spelling correction, string best match problem, correction, corrector, errors, nondeterministic finite-state automata

CR Categories: 4.12, 4.20, 5.22, 5.23, 5.42

This research was supported by NSF grant GJ-33014 with Vanderbilt University. Author's address: Department of Systems and Information Sciences, Vanderbilt University, Box 6147 Station B, Nashville, TN 37235.

[1] This set of edit operations was apparently first used by Irons [10] in 1963 and has been subsequently used in several compilers [1, 2] and compiler-writing systems [9, 10, 11].

All compilers must be prepared to accept "erroneous" or syntactically illegal source strings. Most compilers handle such strings by a process called "error recovery." Error recovery usually proceeds by announcing the discovery of an "error" in the input string as soon as the leading substring of the input is determined to be the "head" of no valid sentence of the language. Next, input string characters are skipped, until a character can be added to the last previously-acceptable leading source substring. This process may overlook errors (in the skipped portion of the string), and may also announce the discovery of many more "errors" than actually exist. La France [9, Sec. 1.2] gives an extensive discussion of this point.

We have recently concentrated on developing error recovery or "correction" techniques based on a different principle. Rather than build error recovery into existing parsing algorithms as an after-the-fact adjustment, we have studied the possibility of performing error correction in advance of parsing. Thus, an error correction algorithm can be thought of as a preprocessor which accepts the (possibly illegal) source string and translates that source string into a guaranteed syntactically legal string. An error-correction criterion is defined which has the property that a legal input string will be unchanged by the error correction algorithm; illegal strings will, of course, be modified during error correction. The general approach was developed at Cornell, in connection with the CORC [1] and PL/C [2] compilers.

The principal error correction criterion we have investigated we term the "minimum edit distance" criterion. We define a set[1] of "edit operations" which can be applied to a source string to modify it. One such set of edit operations is a subset of those suggested by Morgan in [3].

1. *Changing* any single input character into any other single character.
2. *Inserting* any single character into the source string.
3. *Deleting* any single source character.

The minimum edit distance criterion then seeks to find a translation of the source string: (a) which is syntactically legal; and (b) which can be generated from the source string in the fewest possible edit operations. Each application of an edit operation to the source string is accompanied by an error indication; so this criterion minimizes the number of error messages produced. Of course, if the input string is already legal, no error messages will be produced, and the translation produced by error correction will be identical to the original source string.

Apparently, the more complex (in the Chomsky hierarchy of languages) the underlying language is, the slower the error correction algorithm. Thus, we have developed an error correction algorithm for context free languages [4] which requires time of order n^3, where n characters appear in the input string. The present paper describes an error correction method for the regular languages (those recognizable by a finite automaton) which requires time of order n, given an input of n characters. The algorithm is applicable to Morgan's spelling correction, certainly for what he terms "keyword spelling correction," and with some difficulty, to the general problems of correcting the spelling of either keywords or programmer-introduced names for variables, labels, and functions. To apply it, one would construct a finite state automaton which would "accept" any valid keyword. One would then apply the correction algorithm, guided by this automaton, to the given input string. Other applications, for example to the problem of retrieving the "nearest" of a finite set of strings to the given input, are also possible.

The error correction problem has been deliberately couched in terms suggestive of a constrained optimization problem. The "variables" of this problem are the edit operations (made specific as to where each applies in the source string, and what each does). The "constraints" of the problem require that a sequence of edit operations be chosen which changes the given input into a syntactically valid string. The criterion function is the number of edit operations required. Obviously, the problem variables are all discrete (integers). Nevertheless, this formulation of the problem is profitable because it suggests the use of "dynamic programming" [5] to achieve a reasonably efficient correction algorithm. Dynamic programming will be applied to the input, one character at a time, to compute, in effect, the minimum edit distance from the given input string to some valid string of the language. The actual edit operations which achieve correction in this minimum number of steps will be "reconstructed" by proceeding backward over the string, using edit distance information and "choice" information retained from the first "forward" pass over the input.

Dynamic Programming and Correction of Regular Languages

A regular language is characterized by the fact that its sentences are precisely the set of sentences acceptable to some finite-state automaton (FSA). In turn, an FSA has a peculiarly useful property. Suppose the FSA has scanned the first j characters of an input string. Then the only information the FSA retains about the characters already scanned is contained in its "state." Furthermore, regardless of the input, the FSA can be in only one of a finite number of possible states. (For the moment, we will restrict our attention to deterministic FSA's. This restriction will be relaxed later, permitting us to avoid the construction of an equivalent deterministic FSA from a nondeterministic FSA.)

Let us take advantage of these properties of an FSA by defining a generalization of the criterion function for the optimization problem.

Let $F(j,S)$ equal the minimum number of edit operations needed to change the first j characters of input string α into some string β which will cause the FSA which accepts our language to enter state S after reading β.

It will develop that $F(j,S)$ can be computed, given $F(j-1,T)$ (for all states T which the FSA can be in), together with the knowledge of $\alpha\langle j\rangle$ (the jth character of input string α), and some information which depends on the language but not on α. If there are $|\alpha|$ characters in α and we proceed with the computation until $F(|\alpha|,S)$ is available for all states S, then the number

$$F(|\alpha|,R) = \min_{S\in A} F(|\alpha|,S), \tag{1}$$

where A is the set of "accepting" states of our FSA, gives the edit distance from α to the nearest string β acceptable to the FSA. The edit operations themselves can be specified by examining successively information associated with the computation of $F(|\alpha|,R)$ to determine which state T was involved in computing $F(|\alpha|,R)$ from $F(|\alpha|-1,T)$, then finding that state U involved in calculating $F(|\alpha|-1,T)$ from $F(|\alpha|-2,U)$, and so on. This results in tracing out, backward, the successive states that α "should" have forced the FSA through, as each character of α was scanned. β is directly computable during this back-trace, as is the sequence of edit operations needed to change α into β.

Notation. Let $x\langle j\rangle$ be the jth character of string x, and $x\langle i{:}j\rangle$ be the string $x\langle i\rangle\cdots x\langle j\rangle$, when $j\geq i$, and the null string if $j<i$. ($x\langle i{:}j\rangle$ represents the substring of x consisting of characters i, $i+1$, ..., j of x.)

Computation of $F(j,S)$

We claim that, if $j \geq 1$,

$$F(j,S) = \min_{T} F(j-1,T) + V(T,S,\alpha\langle j\rangle), \tag{2}$$

where $V(T,S,c)$ is equal to the smallest number of edit operations which will change the single character c into a string $w(T,S)$ which will force the FSA from state T to state S. For the case $j=0$ (the initial condition), we claim that

$$F(0,S) = \begin{cases} 0, & \text{if } S \text{ is the "start" state of the FSA,} \\ \infty, & \text{otherwise.} \end{cases} \tag{3}$$

Let $G = \min_T F(j-1,T) + V(T,S,\alpha\langle j\rangle)$ for any j. Then, if $F(j-1,T)$ and $V(T,S,\alpha\langle j\rangle)$ are correctly computed, surely $F(j,S) \leq G$. For the correctness of $F(j-1,T)$ shows that $\alpha\langle 1{:}j-1\rangle$ can be changed into β, a string which places the FSA in state T, in $F(j-1,T)$ operations. $\alpha\langle j\rangle$ can be changed to a string which takes the FSA from state T to state S in $V(T,S,\alpha\langle j\rangle)$ operations. So $\alpha\langle 1{:}j\rangle$ can be changed into a string γ which forces the FSA into state S in G operations. So $F(j,S) \leq G$. Now suppose $F(j,S) < G$. Then there must exist a string δ which forces the FSA into state S such that fewer than

G operations are needed to change $\alpha\langle 1{:}j\rangle$ into δ. By a result of Wagner [6], δ can be divided into two substrings δ_1 and δ_2 such that $\delta = \delta_1\delta_2$ and $F(j,S) = D(\alpha\langle 1{:}j\rangle,\delta) = D(\alpha\langle 1{:}j-1\rangle,\delta_1) + D(\alpha\langle j\rangle,\delta_2)$, where $D(\alpha,\beta)$ is equal to the smallest number of edit operations needed to change string α into string β.

Now after reading δ_1, the FSA must be in some state, say state T. δ_1 is then a correction of $\alpha\langle 1{:}j-1\rangle$ which forces the FSA into state T. So $D(\alpha\langle 1{:}j-1\rangle,\delta_1) \geq F(j-1,T)$ by definition of $F(j-1,T)$. Also, δ_2 forces the FSA from state T to state S. So $D(\alpha\langle j\rangle,\delta_2) \geq V(T,S,\alpha\langle j\rangle)$. We have $G > F(j,S) = D(\alpha\langle 1{:}j\rangle,\delta) \geq F(j-1,T) + V(T,S,\alpha\langle j\rangle) \geq G$ for some state T, a contradiction. Hence $G = F(j,S)$.

For the purposes of this algorithm, we assume that the numbers $V(T,S,c)$ are stored in random-access memory, for each character c, and for each ordered pair of (possibly identical) states T and S. To show the practicality of this algorithm, we must show that only a finite (hopefully small) amount of space must be devoted to the storage of $V(T,S,c)$ values, and we must also show that these values are independent of the particular input string read. This will permit the $V(T,S,c)$ values to be computed once, and then used to correct many different input strings.

Storing the $V(T,S,c)$ Information

$V(T,S,c)$, for states T and S and input character c, gives the number of edit operations needed to change c into a string β which will force the FSA from state T to state S. Although there are an infinity of characters c which may possibly appear in the input string, we can easily show that only the finite input alphabet I of the FSA, plus one representative of the infinity of characters not in I, needs to be considered.

A theorem of Wagner [6] shows that a single character x can be changed to a string β in max $(|\beta|,1)$ edit operations, if $x \notin \beta$, and in $|\beta|-1$ edit operations if $x \in \beta$. (If $x \notin \beta$, then x can be changed to the first character of β, and the other characters of β can be inserted following the modified x. If $|\beta| = 0$, so that β is the empty string, this yields one edit operation (a deletion); otherwise it yields $|\beta|$ edit operations. If $x \in \beta$, so that $x = \beta\langle i\rangle$, say, insert $\beta\langle 1{:}i-1\rangle$ before x, $\beta\langle i+1{:}|\beta|\rangle$ after x, for a total of $|\beta|-1$ edit operations.) So, if $x \notin I$ and $y \notin I$, neither x nor y can be accepted by any state of the FSA. Hence, in any string β which drives the FSA from state T to state S appears neither x, nor y. So $|\beta|$ edit operations are needed to correct either x or y to β, so that $V(T,S,x) = V(T,S,y)$. Obviously, only one of these numbers needs to be stored.

Reasoning similar to that outlined above suggests that $V(T,S,c)$ can be reduced to an integer $P(T,S)$, plus a one-bit indicator $L(T,S,c)$ for each character in the set $I \cup \{z\}$, where z is a character not in I. For we argue that, if $P(T,S)$ is equal to the length of the shortest string which forces the FSA from state T to state S, then $V(T,S,c)$ equals $P(T,S)$ if $P(T,S) > 0$ and if c appears in none of the possible shortest strings which force the FSA from state T to state S; $V(T,S,c) = 1$, if $P(T,S) = 0$; and $V(T,S,c) = P(T,S) - 1$, otherwise. So if we let $L(T,S,c) = 1$ if c appears in at least one of the shortest length strings which force the FSA from state T to state S, and $L(T,S,c) = 0$, otherwise, then

$$V(T,S,c) = \begin{cases} 1, & \text{if } P(T,S) = 0, \\ P(T,S) - L(T,S,c), & \text{otherwise.} \end{cases}$$

Note that even if $c \in \gamma$, where $|\gamma| > |\beta| > 0$, where both γ and β force the FSA from T to S, $|\gamma|-1$ edit operations are needed to change c into γ, while $|\beta|$ edit operations suffice to change c into β. Since $|\gamma| > |\beta|$, $|\gamma|-1 \geq \beta$, so it is no worse to correct c to β than to γ. Thus, no string γ whose length is longer than the shortest nonnull string β which forces the FSA from state T to state S need be considered.

Computation of $P(T,S)$ and $L(T,S,c)$

The computation of $P(T,S)$ and $L(T,S,c)$ can be accomplished using a slight modification of a standard algorithm for computing the shortest distance in a graph between all pairs of nodes. Let the length of an arc in the FSA's transition diagram, G, be equal to the number of input symbols read during a transition along that arc. Define $P^k(T,S)$ as equal to the length of the shortest path from state T to state S in G, passing only through states numbered k or less. Then $P^0(T,S)$ is equal to the one-arc distance from T to S (infinite if no arc of G connects T directly to S). $P^{k+1}(T,S) = \min(P^k(T,S), P^k(T,k+1) + P^k(k+1,S))$.

Define

$$L^k(T,S,c) = \begin{cases} 1, & \text{if character } c \text{ is accepted by some arc along a path of length } P^k(T,S) \text{ from } T \text{ to } S, \\ 0, & \text{otherwise.} \end{cases}$$

Then $L^{k+1}(T,S,c)$ can be computed during the computation of $P^{k+1}(T,S)$.

$$L^{k+1}(T,S,c) = \begin{cases} L^k(T,k+1,c) \vee L^k(k+1,S,c), \\ \quad \text{if } P^k(T,S) > P^k(T,k+1) + P^k(k+1,S); \\ L^k(T,k+1,c) \vee L^k(k+1,S,c) \vee L^k(T,S,c), \\ \quad \text{if } P^k(T,S) = P^k(T,k+1) + P^k(k+1,S); \\ L^k(T,S,c), \\ \quad \text{otherwise.} \end{cases}$$

Here, $a \vee b = 0$, if $a = b = 0$, 1, otherwise.

These formulas can be used to compute $P^1(X,Y)$ and $L^1(X,Y,c)$ for all states X and Y and all characters c, then $P^2(X,Y)$ and $L^2(X,Y,c)$, and ultimately $P^t(X,Y)$ and $L^t(X,Y,c)$, where there are t states in the FSA. $P(X,Y) = P^t(X,Y)$, and $L(X,Y) = L^t(X,Y)$, by definition of $P^k(X,Y)$ and $L^k(X,Y,c)$. This technique closely follows the strategy of Floyd's algorithm [7].

Modifications Necessary To Handle NDA

Automated techniques for constructing FSA's from other descriptions of languages often produce nondeterministic finite state automata (NDA's) as a first step. (See, for example, Gries [8 pp. 40–42].) Algorithms are known for transforming NDA's into FSA's, which accept an identical set of strings. However, these algorithms may sometimes increase drastically the number of states in the automaton. Since time and space required by our algorithm grows proportional to P^2, where P is the number of states of the FSA, adaption of the algorithm to handle an NDA seemed worthwhile.

It developed that very little change in the basic algorithm was necessary for this adaption. For an NDA differs from an FSA primarily because it allows the same input character to be accepted along several arcs leading out of state T to other states S, U, etc. A recognizer, which "runs" the automaton, can thus be faced with an ambiguity as to which state to enter next. But in our case, we seek only to calculate the edit-distance required to force the automaton into each possible state; we don't care, during the forward pass of the algorithm, which state the automaton actually enters. Thus, the only change necessary to adapt our algorithm to an NDA lies in the eq. (3). This, we must rewrite as

$$F(0,S) = \begin{cases} 0, & \text{if } S \text{ is any of the possible start} \\ & \text{states of the NDA,} \\ \infty, & \text{else.} \end{cases} \tag{3'}$$

All other calculations remain unchanged. The "backward" pass then traces out a completely valid sequence of state transitions, related to states of the NDA directly. This amounts to a "parse" of the (corrected) input string.

Conclusion

We have sketched an algorithm which, given a string α and an FSA Q whose input alphabet size is $|I|$ and number of states is $|Q|$, can correct α to a string acceptable to Q in time of order $|\alpha|*|Q|^2$. Space proportional to $|Q|^2*|I| + |\alpha|*|Q|$ is also needed by this algorithm. The algorithm can be easily adapted to handle nondeterministic FSA's. This makes it unnecessary to transform an NDA to its equivalent FSA, a transformation which usually increases the number of states in the automaton. The algorithm as adapted for nondeterministic finite state automata serves equally well as a "recognizer" for the set of strings acceptable to the automaton. An input string α must be accepted if and only if the algorithm calculates an edit-distance of 0 for α.

Acknowledgments. The author wishes to acknowledge the contribution of Jean-Pierre Levy to this work. During a conversation with him, the existence of algorithms for efficiently correcting several classes of formal languages became clear. Mr. Levy's development of these ideas appears in [12].

Received November 1972; revised June 1973

References

1. Conway, R.W., and Maxwell, W.L. CORC—the Cornell computing language. *Comm. ACM 6*, 9 (Sept. 1966), 317–321.
2. Morgan, H.L., and Wagner, R.A. PL/C—A high performance compiler for PL/I. Proc. 1971 SJCC, Vol. 38, AFIPS Press, Montvale, N.J., pp. 503–510.
3. Morgan, H.L. Spelling correction in systems programs. *Comm. ACM 13*, 2 (Feb. 1970), 90–94.
4. Wagner, R.A. An n^3 minimum edit distance correction algorithm for context free languages. Tech. Rep., Systems and Information Science Dep., Vanderbilt U., Nashville, Tenn., 1972.
5. Nemhauser, G.L. *Introduction to Dynamic Programming*. Wiley, New York, 1966.
6. Wagner, R.A. The string-to-string correction problem. Tech. Rep., Systems and Information Sciences Dep., Vanderbilt U., Nashville, Tenn., 1971.
7. Floyd, R.W. Algorithm 97—Shortest path. *Comm. ACM 5*, 6 (June 1962), 345.
8. Gries, D. *Compiler Construction for Digital Computers*. Wiley, New York, 1971.
9. LaFrance, J.E. Syntax-directed error recovery for compilers. Ph.D. Th., Rep. No. 459, Dep. of Comput. Sci., U. of Illinois at Urbana-Champaign, Urbana, Ill., June 1971.
10. Irons, E.T. An error-correcting parse algorithm. *Comm. ACM 13*, 11 (Nov. 1963), 669–673.
11. Leinus, R. Error detection and recovery in syntax-directed compilers. Ph.D. Th., U. of Wisconsin, Madison, Wis., 1970.
12. Levy, J-P. Automatic correction of syntax errors in programming languages. Ph.D. Th., Dep. of Comput. Sci., Cornell U., Ithaca, N.Y. 1971.