NCC '82

Integration of bottom-up and top-down contextual knowledge in text error correction

by SARGUR N. SRIHARI, JONATHAN J. HULL, and RAMESH CHOUDHARI
State University of New York at Buffalo
Amherst, New York

ABSTRACT

This paper presents an efficient method for the integration of two forms of contextual knowledge into the correction of character substitution errors in words of text: bottom-up knowledge in the form of character transitional probabilities and top-down knowledge in the form of a dictionary. The method is a modification of the Viterbi algorithm—which maximizes string a posteriori probability by using character confusion and transitional probabilities—so that only legal strings are output. The algorithm achieves its efficiency by using a trie structure representation of a dictionary in the search process. An analysis of the computational complexity and the results of experimentation with the approach are presented.

I. INTRODUCTION

Computer correction of errors in text is important for flexibility in communication between computers and people. The capabilities of present commercial machines for producing correct text by recognizing words in print, handwriting, and speech are very limited. For example, most optical character recognition (OCR) machines are limited to a few fonts of machine print or to text that is handprinted under certain constraints; any deviation from these constraints will produce highly garbled text. Moreover, human beings perform better than these machines by at least an order of magnitude in error rate, although human performance when perceiving a letter or phoneme in isolation is only comparable to that of commercial machines. This is due to human knowledge of contextual factors like letter (or phoneme) sequences, word dependency, sentence structure and phraseology, style, and subject matter as well as associated skills such as comprehension, inference, association, guessing, prediction, and imagination, all of which take place very naturally during the process of reading and hearing.

It is clear that programs that are able to correct errors in text need to be able to use contextual knowledge about the text as well as knowledge about the likely sources of textual errors. A number of programs for using some form of contextual knowledge in text error correction are described in the literature. Among these one can discern two basic approaches: those that are data-driven, or bottom-up, and those that are concept-driven, or top-down.

Data-driven algorithms for text error correction proceed by refining successive hypotheses about an input string. Examples of such an approach are those that use a statistical representation of contextual knowledge—e.g., a Markovian model of text source, which consists of a set of tables representing the probability of occurrence of a letter, given that a set of letters have occurred previously.

Concept-driven algorithms proceed with an expectation of what the input string is likely to be and proceed to fit the data to this expectation. Examples are algorithms that use implicit or explicit representations of dictionaries, syntax, and semantics.

In what follows we describe an algorithm that effectively merges a bottom-up refinement process based on the use of transition probabilities with a top-down process based on searching a trie-structure representation of a dictionary. The algorithm is applicable to text containing an arbitrary number of character substitution errors, such as that produced by OCR machines; thus the method excludes character deletion, insertion, and transposition errors that a typographical error correction algorithm needs to consider.

II. VITERBI ALGORITHM

The Viterbi algorithm (VA) is a method of computing the most probable word that could have caused the observed word. This probability is computed by taking into account the probabilities of confusion between letters and the probabilities of cooccuring n-grams.

Let the observed word be $X = X_0X_1 \ldots X_mX_{m+1}$ where X_0 and X_{m+1} are the delimiters of the m-letter word. The probability that a word $Z = Z_0Z_1 \ldots Z_mZ_{m+1}$ could have caused X is expressed by using Bayes decision theory as

$$P(Z/X) = [P(X/Z)^*P(Z)]/P(X)$$

where $P(X/Z)$ is the probability of observing X when Z is the true word, $P(Z)$ is the a priori probability of Z, and $P(X)$ is the probability of string X. Since $P(X)$ is independent of Z, the word Z that maximizes $P(Z/X)$ can be determined by maximizing the expression

$$G(X/Z) = \log P(X/Z) + \log P(Z).$$

Storing the $P(X/Z)$ distribution in memory is impractical because of the large number of combinatorial possibilities for X and Z. If we assume conditional independence among $X_0, X_1, \ldots, X_{m+1}$, then

$$\log P(X/Z) = \sum_{i=0}^{m+1} \log P(X_i/Z_i).$$

According to this assumption, the observed letters are independent of each other, which is valid for printed text but not necessarily so for cursive script. The probability $P(X_i/Z_i)$ is the probability of observing letter X_i when the true letter is Z_i, which is called the confusion probability.

If we assume that words are generated by an nth-order Markov source, then the a priori probability $P(Z)$ can be expressed as

$$P(Z) = P(Z_{m+1}/Z_{m+1-n} \ldots Z_m) \ldots P(Z_1/Z_0)^*P(Z_0),$$

where $P(Z_k/Z_{k-n} \ldots Z_{k-1})$ is called the nth-order transitional probability, i.e., the probability of observing Z_k when the previous n letters are $Z_{k-n} \ldots Z_{k-1}$.

In the case of n = 1,

$$P(Z) = P(Z_{m+1}/Z_m) \ldots P(Z_1/Z_0)^*P(Z_0)$$

and the word Z with maximum a posteriori probability is one that maximizes

$$G_1(X,Z) = \sum_{i=1}^{m+1} \log P(X_i/Z_i) + \log P(Z_i/Z_{i-1})$$

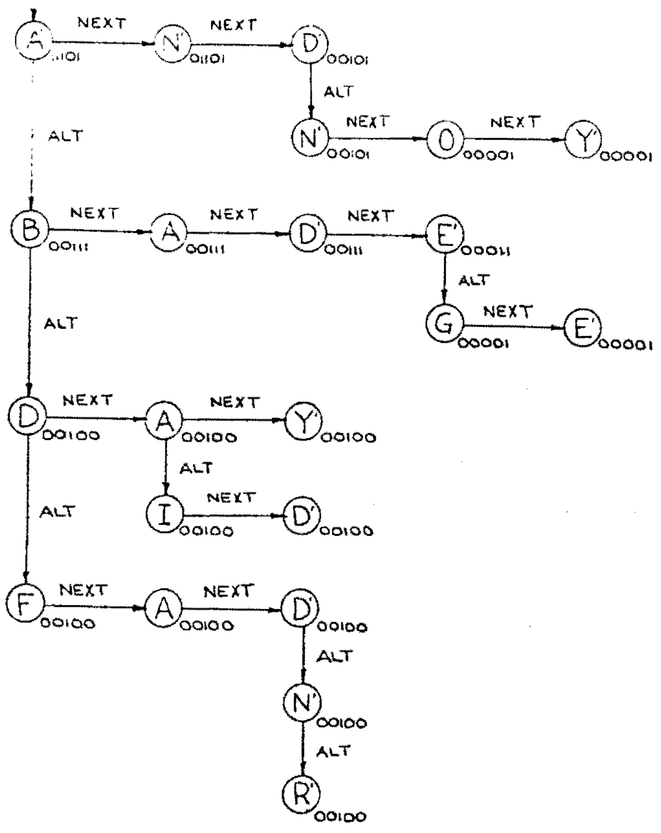

Figure 1—Trie structure representation of the dictionary A, AN, AND, ANN, ANNOY, BAD, BADE, BADGE, DAY, DID, FAD, FAN, FAR. Each node has a 5-bit word-length indicator array and word termination is indicated by a quote mark.

where it is assumed that $P(X_0/Z_0) = P(X_{m+1}/Z_{m+1}) = 1$, i.e., the delimiter symbol is perfectly recognized. In the case of $n = 2$, the corresponding expression is

$$G_2(X, Z) = \sum_{i=1}^{m+1} \log P(X_i/Z_i) + \log P(Z_i/Z_{i-2}Z_{i-1}),$$

where

$$P(Z_1/Z_{-1}Z_0) = P(Z_1/Z_0).$$

The VA is a method of finding the word Z that maximizes $G_i\ (X,Z)$ without having to compute all 26^m possible $G_i\ (X,Z)$. The method is based on a dynamic programming formulation, which leads to a recursive algorithm. Essentially, if L_j, $j = 1, \ldots 26$, represents the jth letter of the alphabet, then $\max[G_i(X_1 \ldots X_k, Z_1 \ldots Z_{k-1}Z_k = L_j)]$ over all possible values of $Z_1 \ldots Z_{k-1}$ can be computed trivially if we know the 26 values corresponding to $\max[G_i(X_1 \ldots X_{k-1}, Z_1 \ldots Z_{k-2}Z_{k-1} = L_r)]$, $r = 1, \ldots, 26$ over all possible values of $Z_1 \ldots Z_{k-2}$. This formulation reduces the complexity of the algorithm to $O(26^2)$, which is superior to $O(26^m)$, required by the exhaustive search. The algorithm can be viewed as a shortest-path algorithm through a directed graph of $26 \times m$ nodes, called a trellis. The negative of the log transitional probabilities is associated with the edges of the trellis, and the negative log confusion probabilities are associated with the nodes. The cost of a path is then the sum of all the edge and node values in the path.

Some generalizations of the VA have used either a fixed number of alternatives that is less than 26, called the modified Viterbi algorithm (MVA),[1] or a variable number of alternatives[2] for each Z_k. These alternatives can be determined by the letters that have the highest confusion probability.

III. THE TRIE

In contrast to the usual lexicographical organization such as that used in a desktop dictionary, several alternative structures have been described.[3] The decision to use such an alternative is based on the search strategy of the text manipulation algorithm and the memory available.

One of the ways to represent a dictionary and the one used here is the trie. The trie and its variations are discussed at length by Knuth,[4] and text enhancement systems that use it as their basis are described by Muth and Tharp[5] and by Kashyap and Oommen.[6]

The trie considers words to be ordered lists of characters, elements of which are represented as nodes in a binary tree. Each node has five fields: a token, CHAR; a word-length indicator array of bits, WL; an end-of-word tag bit; and two pointers labeled NEXT and ALTERNATE.

A node is a NEXT descendant if its token follows the token of its father in the initial substring of a dictionary word. It is an ALTERNATE descendant if its token is an alternative for the father's, given the initial substring indicated by the most immediate ancestor, which is a NEXT descendant (see Figure 1). Without loss of generality it is required that the lexical value of the token of each ALTERNATE descendant be greater than that of its father. The end-of-word bit is set if its token and the initial substring given to reach the token compose a complete dictionary word. The mth bit of the word-length-indicator array is set if the token is on the path of an m-letter word in the trie.

If a dictionary has been given as a trie, with fields initialized as above, the following function determines whether the character ch in a word X of m characters follows an initial substring given by a pointer p to the first possible character following this substring.

```
Function ACCESS-TRIE (var p : ptr; ch : char) : boolean;
begin
    if (p = nil) or (p^. CHAR > ch) or (p^.WL[m] = 0)
      then
        ACCESS-TRIE : = FALSE
      else
        if (p^.CHAR = ch)
          then
            begin
                p: = p^.NEXT;
                ACCESS-TRIE : = TRUE
            end
          else
          ACCESS-TRIE: = ACCESS-TRIE
                          (p^.ALTERNATE, ch)
end; (* ACCESS-TRIE *)
```

A version of this function will be used with the proposed bottom-up and top-down algorithm. It is interesting to note that the maximum number of recursive calls for any given initial substring is 25. If it were assumed that for all positions in any string the possible letters were uniformly distributed over all 26 possibilities, the average number of calls would be 12.5. This assumption is clearly unreasonable because of the nature of the trie and the English language itself. For example, with "AMPLIFYIN" as an initial substring, there is only one possibility for the tenth position. This characteristic is reflected in the experimentation discussed in Section VI where the average number of alternatives for all nodes in a sample trie was 1.62.

IV. DICTIONARY VITERBI ALGORITHM

The MVA is a purely bottom-up approach, whose performance may be unacceptable. For example, in experimentation with the MVA,[7] the best overall word correction rate was 46% when second-order word-length and position-independent (WLPI) statistics were used and 20% when first-order WLPI statistics were used. For an efficient contextual postprocessing system, this performance must be improved. One approach to the problem is to use top-down contextual information, in the form of a dictionary of allowable input words, to aid the bottom-up performance of the MVA.

One such method, known as the predictor-corrector algorithm,[8] uses an extension of the Bledsoe-Browning[9] approach. Given a word output by the MVA, it computes a score for the word. A constrained search and computation procedure is then carried out over the dictionary. This is a two-part method, in which the use of dictionary information is distinct from processing by the MVA. In this section an algorithm is proposed that integrates dictionary information with MVA processing. The resultant dictionary Viterbi algorithm (DVA) offers the advantages of a dictionary method in terms of legibility of output and correction rates but shows no increase in order of complexity from the MVA.

The formal statement of the text enhancement procedure based on the DVA follows.

The Algorithm

```
repeat
    GETWORD($X$);   (* Read next word $X$ *)
    DICTIONARY-VITERBI($X$, $Z$);
    WORD-OUT($Z$);   (* Output word $Z$ *)
until end-of-file;
```

The procedure DICTIONARY-VITERBI, stated below, is for the case of a first-order Markov assumption and a fixed number of alternatives, d, for each letter. This is similar to the MVA and is performed to allow comparison of the complexity of the two algorithms.

Symbols and data structures

$L_1 \ldots L_{26}$ represent symbols A ... Z, and the delimiter symbol is ƀ.

C is a vector of d real numbers called the cost vector.
Q is a vector of d pointers into the trie, initially the root.
S is a vector of d character strings called the survivor vector.
$X = X_1 \ldots X_m$ is the input character string.
$Z = Z_1 \ldots Z_m$ is the output character string.
$\bar{A}$ is a d × m matrix of alternatives whose columns are labeled $A_1 \ldots A_m$.

Primitive functions

MAX($a_1 \ldots a_d$, u) returns the maximum of $\{a_1 \ldots a_d\}$, and the index of the maximum in u.
CONCAT(s,L_j) concatenates character L_j at the end of string s.

```
Procedure DICTIONARY-VITERBI($X_1 \ldots X_m, Z_1 \ldots Z_m$);
    (*given an m-letter string $X = X_1 \ldots X_m$ as input,
      produce an m-letter string $Z = Z_1 \ldots Z_m$ as output*)
    begin
        INITIALIZE($\bar{A}$);
        DICTIONARY-TRACE ($\bar{A}$, C, Q, S, $X_1 \ldots X_m$);
        $Z$ := SELECT($\bar{A}$, C, S);
    end;
```

Procedure INITIALIZE selects the d most likely alternatives for each letter of the input word. This is done by choosing those d letters for which the sum of the log-confusion and log-unigram probabilities is greatest.

Procedure DICTIONARY-TRACE, which follows, returns a set of character strings in Vector S whose costs are defined by Vector C.

```
Procedure DICTIONARY-TRACE($\bar{A}$, C, Q, S, $X_1 \ldots X_i$);
  begin (*C1, S1, Q1, Q2 are local vectors of d
    elements*)
    if i > 1 then begin
        DICTIONARY-TRACE($\bar{A}$, C, Q, S, $X_1 \ldots X_{i-1}$);
        C1 := C; Q1 := Q;
        S1 := S; Q2 := Q;
        for j := 1 to d do begin
          for k := 1 to d do begin
            if ACCESS-TRIE(Q1(k), $A_i$(j))
              then $g_k$ := C1(k) + log P($X_i$/$A_i$(j))
                  + log P($A_i$(j)/$A_{i-1}$(k))
              else $g_k$ := -inf end;
          C(j) := max ($g_1$, ... ,$g_d$, u);
                        Q(j) := Q1(u);
          if (C(j) < > -inf)
            then S(j) := CONCAT(S1(u), $A_i$(j))
            else S(j) := null;
          Q1 := Q2;
          end;
      end
    else begin (*i := 1 *)
      for j := 1 to d do
        if ACCESS-TRIE (Q(j), $A_1$(j))
          then begin
            C(j) := log P($X_1$/$A_1$(j))
              + log P($A_1$(j)/ƀ);
            S(j) := $A_1$ (j); end
```

```
                else begin C(j) := -inf; Q(j) := nil;
                      S(j) := null; end;
            end;
      end; (*DICTIONARY-TRACE *)
```

Function SELECT returns the most likely word by considering the cost of the transition from the final symbol to the trailing delimiter ƀ when the cost vector C and the survivor vector S are used. If all the values in C are equal to minus infinity, *X* is rejected and a null value is returned.

The integration of the dictionary into the Viterbi algorithm is done in Procedure DICTIONARY-TRACE by maintaining a vector of pointers into the trie. Each element corresponds to a survivor string. At each iteration of the k loop the kth element of this vector is passed to ACCESS-TRIE. If the corresponding initial substring concatenated with the letter indicated by the j index is a valid dictionary string (ACCESS-TRIE is true), the probability calculation is carried out. A weight of minus infinity is given to this alternative when a false value is returned in order to preclude the possibility of non-dictionary words being considered. At some iteration in the j loop, if all attempted concatenations fail to produce a valid dictionary string, the survivor for the corresponding node and its pointer are assigned null values. For some value of i, if all survivors are null, the input word is rejected as uncorrectable. This may happen when less than 26 possibilities are considered for each letter, but it will never happen when all candidates are allowed. This phenomenon is discussed in Section VI.

A variation of the DVA would be to use the pointer to the node that corresponds to the alternative chosen at the last step as a substitute for the explicit maintenance of survivor strings. If the trie included son-to-father pointers, this pointer would allow a path to be traced from the indicated node back to the first level of the trie to retrieve the output string. This would yield storage economy when the number of nodes was less than the number of locations required for the survivor strings because of the additional pointer required at each node.

The above algorithm considers a fixed number of d alternatives for each letter of the input string. A modification of the algorithm to include a variable number of alternatives is as follows. Within procedure INTIALIZE only letters for which the sum of the log-confusion and log-unigram probabilities is greater than an a priori threshold value t are chosen as alternatives.

V. COMPUTATIONAL COMPLEXITY

The complexity of the MVA derived by Shinghal and Toussaint[1] will be used to show the additional computation of the DVA. Only the general case of $n > 1$ and $1 < d < 26$ will be discussed here. The overall computation requirement of the DVA can be expressed as a function of n and d, as shown in the following:

$$D(n, d) = D_a(n, d) + D_p(n, d),$$

where $D_a(n, d)$ is the requirement for the selection of alternatives and $D_p(n, d)$ is the requirement for the path tracing. Since the selection of alternatives remains the same as in the MVA,

$$D_a(n, d) = 26n + n \sum_{j=1}^{\min(26-d,\ d)} (26 - j).$$

Path tracing involves two steps. Step 1 is the path tracing itself and step 2 is the evaluation of the last letter to blank transition. A trie look up τ is defined as the number of comparisons necessary in a call to ACCESS-TRIE. An addition and comparison are defined to equal one unit of computation.

Step 1 requires:

$$d^2(n-1)(\tau+3) + d(\tau+1) \qquad \text{units},$$

which is an upper bound occuring when all trie look-ups are successful.
Step 2 requires:

$$(2d-1) \qquad \text{units}.$$

Therefore,

$$D_p(n,d) = d^2(n-1)(\tau+3) + d(\tau+1) + (2d-1).$$

Therefore, the complexity of the DVA is:

$$D(n,d) = 26n + n \sum_{j=1}^{\min(26-d,d)} (26-j) + d^2(n-1)(\tau+3) + d(\tau+1) + (2d-1).$$

The complexity of the MVA[1] is:

$$V(n,d) = 26n + n \sum_{j=1}^{\min(26-d,d)} (26-j) + 3d^2(n-1) - nd + (2d-1).$$

Comparison of V(n,d) and D(n,d) shows no change in order of complexity, with both expressions increasing linearly as a function of n and quadratically as a function of d. The experimentally derived value of the average number of alternatives at a trie node of 1.62 suggests only an increase in the coefficient of d^2.

VI. EXPERIMENTAL RESULTS

To determine the efficiency and performance of the DVA and to compare this with the MVA, a data base was established and experiments were conducted.

English text in the Computer Science domain (Chapter 9 of *Artificial Intelligence*, P.H. Winston, Addison-Wesley, 1977) containing 6372 words was entered onto a disc file. Unigram and first order transition probabilities were estimated from this source. A model reflecting noise in a communications channel was used to introduce substitution errors into a copy of this text and confusion probabilities were estimated from this source. The same probability tables were used for all experiments.

A dictionary of 1724 words containing 12231 distinct letters was extracted from this text and a trie was constructed for use by the DVA. There were 6197 nodes in the trie and the average number of alternates for all nodes was 1.62. The frequency histogram of alternate path lengths is extremely skewed, with 4805 nodes having no alternatives but itself (path length 1) and 701, 240, and 128 nodes having alternate path lengths of 2, 3, and 4, respectively.

An example of input and output text to the DVA follows:

IF WE LOOI AT WHAT HAS PRODUSED LOMPUTER IMTELLIGENCE QO FAR, WE SEE MULTIPLE LAMERS, EACH OF WHICH RESTS ON PRIMITIVES OF THE NAXD TAYFR DOWM, FORMINC A HIERARCFICAL STRUCTURE WITH A GREAT DEAL INTERPOSED BETWEEN THE INTELLIGENT PRPHVEM AND THE TRANSISTORS WHICH ULTIMATELU SUPPODT IT. ALL OF THE CGMPLEXITU OF ONE KEVEL IS SUMMARIZFD ABD DISTILLED DOWN TO A BES SIMPLE ASOMIC NOTIONS WHICH AZE THE PRIMITIVES OE THE NEXT LAMER UP. BUT WITH SO MUCH INSULATIOP, IT CCNNOT POSSMBLY BE THAT THE DETAILFD NATURE OF THE LGWER LEVELS CAN MATTER TO WHAT HAPPENS AFOXE.

IF WE LOOK AT WHAT HAS PRODUCED ******** INTELLIGENCE SO FAR, WE SEE MULTIPLE LAYERS, EACH OF WHICH RESTS ON PRIMITIVES OF THE NEXT ***** DOWN, FORMING A HIERARCHICAL STRUCTURE WITH A GREAT DEAL INTERPOSED BETWEEN THE INTELLIGENT PROGRAM AND THE TRANSISTORS WHICH ULTIMATELY ******* IT. ALL OF THE COMPLEXITY OF ONE LEVEL IS SUMMARIZED AND DISTILLED DOWN TO A BUT SIMPLE ATOMIC NOTIONS WHICH ARE THE PRIMITIVES OF THE NEXT LAYER UP. BUT WITH SO MUCH INSULATION, IT CANNOT POSSIBLY BE THAT THE DETAILED NATURE OF THE LOWER LEVELS CAN MATTER TO WHAT HAPPENS ABOVE.

The output was produced by the DVA using a fixed number of alternatives with the depth of search set at 6; LOMPUTER, TAYFR, and SUPPODT were rejected; and BES was erroneously corrected to BUT instead of FEW. Rejections could be eliminated by increasing the depth of search, because a dictionary word could be located that could not be located previously because of the constrained nature of the trellis.

The performances of the DVA and the MVA were measured by the percentage of garbled words corrected when both algorithms were run on the same piece of text. A fixed and variable number of alternatives were used in both cases.

To contrast the complexity of the DVA and the MVA, a fixed number of alternatives was used for both algorithms, and the CPU time required to process a fixed input text was used for comparison. The same program was used in all cases, the only differences being those necessary to implement the particular version of the algorithm.

The results of applying the algorithm to the entire random sample of garbled text (of 6372 words) are summarized in Table I. Time figures are CPU seconds on a CDC Cyber 174. The DVA in all cases performed significantly better than the MVA without a dictionary: the best-case correction rate for the DVA was 87%; the corresponding figure for the MVA was 35%. To minimize the cost it is necessary to choose the minimum value of the depth of search (d) or the minimum threshold (t) that give the optimum correction rate. These were found to be 8 and −11, respectively. While the time requirement at optimum performance for the DVA differed by about a factor of 1.7 from the MVA, it is interesting to note that the best performance for the DVA in the variable-alternatives case was achieved at a cost significantly less than that using a fixed number of alternatives.

To show the effects of differing levels of contextual information on performance at the optimum parameter settings, the DVA was run with only top-down information by setting all transition probabilities equal; and the MVA was run without the trie, thus using only the bottom-up information provided by the transition probabilities. The correction rates were 82% and 35%, respectively—both less than the 87% provided by the combination approach.

TABLE I—Results of application of algorithm to entire random sample of garbled text

	Fixed Number of Alternatives					Variable Number of Alternatives			
	DVA		MVA			DVA		MVA	
d	% corr.	time (secs.)	% corr.	time (secs.)	t	% corr.	time (secs.)	% corr.	time (secs.)
1	1	770	1	732	−2	0	473	0	463
2	39	853	23	767	−3	0	491	0	479
3	61	946	29	808	−4	0	517	0	490
4	74	1085	33	858	−5	0	522	0	496
5	81	1256	34	916	−6	11	527	8	496
6	85	1474	35	989	−7	44	593	23	500
7	86	1754	35	1082	−8	76	803	33	614
8	87	2122	35	1175	−9	83	1025	35	695
9	87	2536	35	1287	−10	85	1239	35	769
					−11	87	1668	35	819
					−12	87	1668	35	915
					−13	87	1669	35	922

VII. CONCLUSIONS

A new algorithm for text enhancement has been presented that merges processing by the Viterbi algorithm with dictionary information stored in a trie. The results of experimentation with this algorithm were described; they show a correction rate significantly greater than counterparts of the algorithm that do not use a dictionary. An expression for the complexity of this algorithm has been derived and compared with that of one of its counterparts; it shows no increase in the order of complexity due to the addition of the trie. Because of its superior performance, this algorithm is suggested as a low-level word hypothesization component in a system focusing global contextual knowledge on the text enhancement problem.

ACKNOWLEDGMENT

This work was supported in part by National Science Foundation Grant IST-80-10830. We wish to thank Addison-Wesley for permission to use portions of *Artificial Intelligence*, by P.H. Winston, in the experimentation. Thanks are also due to C.T. Chen for the program to build the trie structure from its word list.

REFERENCES

1. Shinghal, R., and G. T. Toussaint. "Experiments in Text Recognition with the Modified Viterbi Algorithm." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1 (1979), pp. 184–192.
2. Doster, W., and J. Schurman. "An Application of the Modified Viterbi-Algorithm in Text Recognition." *Proceedings of International Conference on Pattern Recognition*, Miami, Florida, December 1–4, 1980. Piscataway, New Jersey: IEEE Computer Society, 1980.
3. Peterson, J. L. "Computer Programs for Detecting and Correcting Spelling Errors." *Communications of the ACM*, 23 (1980), pp. 676–687.
4. Knuth, D. E. *Sorting and Searching*. The Art of Computer Programming, vol. 3. Reading, Massachusetts. Addison-Wesley, 1973.
5. Muth, F. E., and A. L. Tharp. "Correcting Human Error in Alphanumeric Terminal Input." *Information Processing and Management*, 13 (1977), pp. 329–337.
6. Kashyap, R. L., and B. J. Oommen. "An Effective Algorithm for String Correction Using Generalized Edit Distances." *Information Sciences*, 23 (1981), pp. 123–142.
7. Srihari, S. N., and J. Hull. "Experiments in Text Recognition with the Binary N-Gram and Viterbi Algorithms." SUNYAB/CS Technical Report 184, Department of Computer Science, State University of New York at Buffalo, July 1981.
8. Shinghal, R., and G. T. Toussaint. "A Bottom-up and Top-down Approach to Using Context in Text Recognition." *International Journal of Man-Machine Studies*, 11 (1979), pp. 201–212.
9. Bledsoe, W. W., and J. Browning. "Pattern Recognition and Reading by Machine." In L. Uhr (ed), *Pattern Recognition*. New York: Wiley, 1966, pp. 301–316.