Integrating Diverse Knowledge Sources in Text Recognition

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

A new algorithm for text recognition that corrects character substitution errors in words of text is presented. The search for a correct word effectively integrates three knowledge sources: channel characteristics, bottom-up context, and top-down context. *Channel characteristics* are used in the form of probabilities that observed letters are corruptions of other letters; *bottom-up context* is in the form of the probability of a letter when the previous letters of the word are known; and *top-down context* is in the form of a lexicon. A one-pass algorithm is obtained by merging a previously known dynamic programming algorithm to compute the maximum a posteriori probability string (known as the Viterbi algorithm) with searching a lexical trie. Analysis of the computational compexity of the algorithm and results of experimentation with a PASCAL implementation are presented.

CR Categories and Subject Descriptors: H.4.1 [**Information Systems Applications**]: Office Automation—*word processing*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search—*dynamic programming, graph and tree search strategies*; I.5.4 [**Pattern Recognition**]: Applications—*text processing*; I.7.1 [**Text Processing**]: Text Editing—*spelling*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Knowledge integration

1. INTRODUCTION

Computer correction of errors in text is important for flexibility in man–computer communication. The capabilities of present commercial machines for producing correct text by recognizing words in print and handwriting 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 produces highly garbled text. Moreover, human beings perform better than these machines by at least an order of magnitude in error rate, although their 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 such as letter sequences, word dependency, sentence structure and phraseology, style and subject matter, as well

This work was supported in part by the National Science Foundation under Grant IST-80-10830.
Authors' address: Department of Computer Science, State University of New York at Buffalo, Amherst, NY 14226.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1982 ACM 0734-2047/83/0100-0068 $00.75

as comprehension, inference, association, guessing, prediction, and imagination, all of which take place very naturally during the process of reading.

It is clear that if computer programs are to reach the expert level of human ability in text recognition, they need to be able to effectively integrate diverse contextual knowledge sources about the text, as well as knowledge about the kinds of textual errors that are likely (i.e., characteristics of the text transmission channel that introduces errors). A number of programs that utilize only a few knowledge sources in text error correction are described in the literature; tutorial surveys of these methods have been made by Toussaint [18], Peterson [13], and Hall and Dowling [7]. Some of these methods (viz., text recognition algorithms) are directly applicable to a set of image vectors representing characters of text; others (viz., text error correction algorithms) are applicable only to previously decoded text. A majority of these methods can be characterized as 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. An example is a program that utilizes a statistical (Markovian) representation of contextual knowledge in the form of a table of *transitional probabilities*, that is, the probability for each 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 fit the data to meet this expectation. Examples are algorithms that use implicit or explicit representations of dictionaries, syntax, and semantics.

The algorithm described in this paper effectively merges a bottom-up refinement process that is based on the utilization of transitional probabilities with a top-down process based on searching a trie structure representation of a dictionary that is applicable to text containing an arbitrary number of character substitution errors such as those produced by OCR machines. Note that although errors due to insertion and deletion of letters are possible, they are unlikely with machine and handprinting. The method utilizes the characteristics of the OCR channel in the form of a table of *confusion probabilities*. Each entry in this table represents the probability that the OCR channel assigns a given letter to another (possibly the same) letter owing to ambiguities in the shape features used to classify shapes into character classes.

A preliminary version of the algorithm described here was given by Srihari, Hull, and Choudhari [17]; it is refined and presented with further experimental results in this paper. A method for handling textual errors due to crowding of letters (or merging errors) and errors due to incorrect segmentation boundaries (or splitting errors) is described by Srihari and Bozinovic [16]; in this method channel characteristics are modeled by a probabilistic finite state machine, and the dictionary is represented by a trie structure. There exists a fairly large literature on feature extraction and classification methods for OCR; see in particular the companion papers by Munson [10] and Duda and Hart [5].

2. PROBLEM FORMULATION

Let the observed word be $X = X_0 X_1 \cdots X_m X_{m+1}$, where the letters $X_i\,(1 \leq i \leq m)$ belong to an alphabet $L = \{L_1, \ldots, L_r\}$, and X_0 and X_{m+1} are special letters

known as delimiters (blank, hyphen, etc.) which are used to separate different words of text. The probability that a word $Z = Z_0 Z_1 \cdots Z_m Z_{m+1}$ could have caused X is given from Bayes decision theory as the a posteriori probability

$$P(Z \mid X) = \frac{P(X \mid Z) * P(Z)}{P(X)},$$

where $P(X \mid Z)$ 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 $P(Z \mid X)$ can be determined by maximizing the expression

$$G(X \mid Z) = \log P(X \mid Z) + \log P(Z). \tag{1}$$

The objective is to determine the word Z in a set of words D that maximizes eq. (1). The set D itself is determined by contextual constraints and is known as a *dictionary* or *lexicon*.

Storing the $P(X \mid Z)$ distribution in memory is impractical, due to 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

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

According to this assumption the observed letters are independent of each other, which is valid for printed text but not necessarily for cursive script. The probability $P(X_i \mid Z_i)$ 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

$$P(Z) = P(Z_{m+1} \mid Z_{m+1-n} \cdots Z_m) \cdots P(Z_1 \mid Z_0) * P(Z_0), \tag{2}$$

where $P(Z_n \mid Z_{k-n} \cdots Z_{k-1})$ is called the *nth-order transitional probability,* that is, the probability of observing Z_k when the previous n letters are $Z_{k-n} \cdots Z_{k-1}$.

In the case of $n = 1$, $P(Z) = P(Z_{m+1} \mid Z_m) \cdots P(Z_1 \mid Z_0) * P(Z_0)$, and the word Z with maximum a posteriori probability maximizes

$$G_1(X, Z) = \sum_{i=1}^{m+1} \log P(X_i \mid Z_i) + \log P(Z_i \mid Z_{i-1}), \tag{3}$$

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

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

where $P(Z_1 \mid Z_{-1} Z_0) = P(Z_1 \mid Z_0)$.

The *Viterbi algorithm* (VA) [6, 12] is a method of finding the word Z that maximizes $G_i(X, Z)$ over all possible Z, not necessarily those in a dictionary D. Efficiency of the VA can be improved by using either a fixed number of d ($d < r$) alternatives per input letter, called the modified Viterbi algorithm (MVA) [14], or a variable number of alternatives [4]. The VA with a variable number of

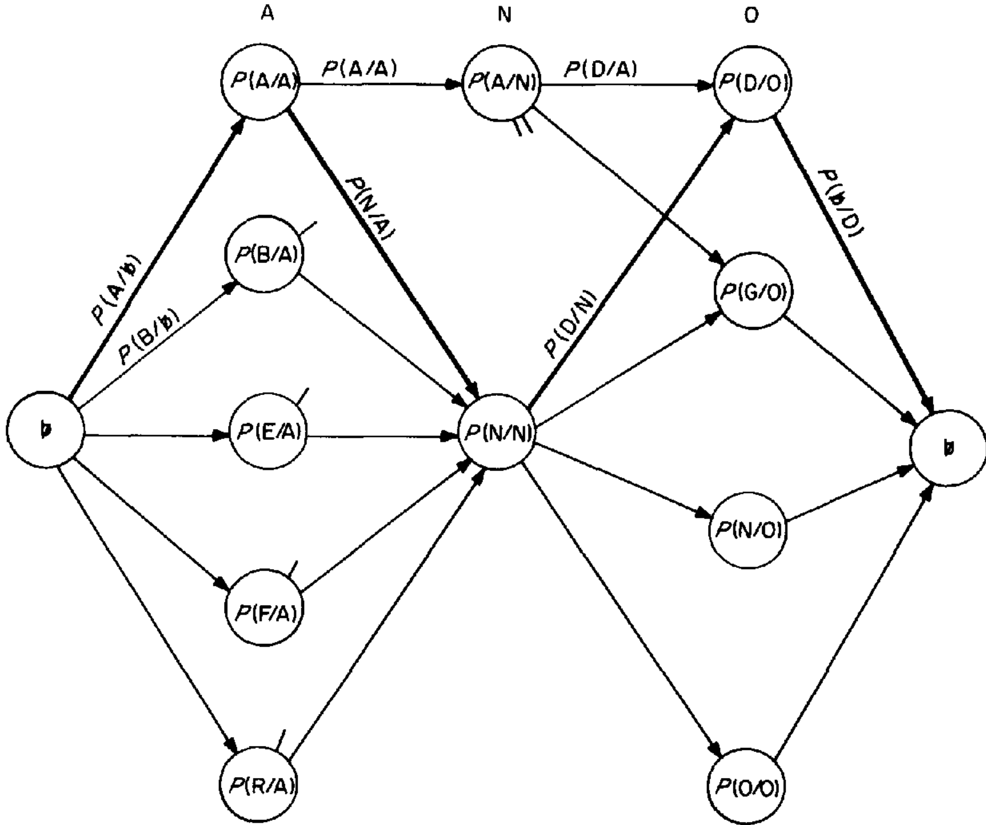

Fig. 1. The Viterbi algorithm finds the maximum cost path through a trellis.

alternatives per input letter is illustrated by an example using the ten-letter alphabet $L = \{A, B, D, E, F, G, N, O, R, Y\}$. Given the input word bANOb, where b indicates a delimiting blank, the ten alternatives for each input letter are sorted by confusion probability and only those that exceed a threshold t are considered. The directed graph of Figure 1, also called a trellis, shows the information used in deciding the output string when there are five, two, and four alternatives that exceed a threshold for the letters A, N, and O, respectively. Each of the nodes (except the start and end nodes) and edges have values associated with them. The node values are confusion probabilities and the edge values are transitional probabilities. Any path from the start node to the end node represents a sequence of letters; note that by considering a variable number of alternatives we have 40 possible paths rather than 1000. Considering the dark path in Figure 1, if we add the logarithms of all the edge and node probabilities encountered on that path, we get the cost of the path as

$$G_1(\text{ANO}, \text{AND}) = \log[P(\text{A} \mid \text{A})P(\text{N} \mid \text{N})P(\text{D} \mid \text{O})] + \log[P(\text{A} \mid b)P(\text{N} \mid \text{A})P(\text{D} \mid \text{N})P(b \mid \text{D})],$$

which is equivalent to (3). Thus the problem is one of finding the maximum cost

path in the trellis. The VA achieves its efficiency by dynamic programming. At each stage in the sequence of letters the most likely string ending with each of the possible alternatives is computed. Only these strings are used in the computation for the next stage. At the final stage the most likely sequence is chosen.

The VA (and its variations) is a purely bottom-up approach whose performance may be unacceptable due to the fact that the resulting strings do not necessarily belong to a dictionary. For example, in experimentation with the MVA [8], the best overall word correction rate was 46 percent using second-order word length and position-independent (WLPI) statistics (transitional probabilities), and 20 percent using first-order WLPI statistics (i.e., 46 percent and 20 percent of the garbled words were transformed into correct dictionary words, while the remainder were either transformed into other garbled words or incorrect dictionary words). One approach to this 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.

A method of using lexical knowledge to improve the performance of the VA, known as the *predictor-corrector algorithm*, was proposed by Shinghal and Toussaint [15] and improved upon by Bouchard and Toussaint [2]. Given a word Z that is output by the MVA (i.e., the predicted word), a *value* is computed for Z as $\log P(Z)$ according to (2) and a first-order assumption. If there is no dictionary word with the same value, Z is assumed to be incorrect. The correction is achieved by determining the dictionary word whose *score* is nearest to that of Z; thus the correction procedure is similar to that of Bledsoe and Browning [1]. The score is computed according to (3) over a heuristically determined subset of the dictionary. The predictor-corrector algorithm is a two-step approach where bottom-up processing by the MVA is distinct from the top-down dictionary search. Moreover, the resulting word is not necessarily the most likely (in the sense of a posteriori probability) of the dictionary; thus the predictor-corrector algorithm is nonadmissible. Our objective is to explore the integration of dictionary information into the VA trellis search so as to retain VA efficiency while guaranteeing the most likely dictionary word as output.

3. LEXICAL ORGANIZATION

The data structure used to represent the lexicon and the method used to access words in it are critical to the efficiency of any text correction algorithm. Several alternative structures are described by Peterson [13]; the choice has to be based on the search strategy of the algorithm and the memory available.

A data structure that is suitable for determining whether a given string is an initial substring, or *prefix*, of a word in the lexicon is known as the *trie*. The trie and its variations are discussed at length by Knuth [9], and a text error correction system that uses this data structure has been described by Muth and Tharp [11]. Since the VA proceeds by computing for a given length the most likely prefix, the trie is an attractive data structure. Essentially, the trie considers words as ordered lists of characters, elements of which are represented as nodes in a binary tree. Each node has 5 fields: a token, CHAR; a word-length indicator array of bits, WL; an end-of-word tag bit, E; and two pointers labeled NEXT and ALTERNATE. An example of a letter trie representing a lexicon of 13 words based on the 10-letter alphabet used in Figure 1 is given in Figure 2.

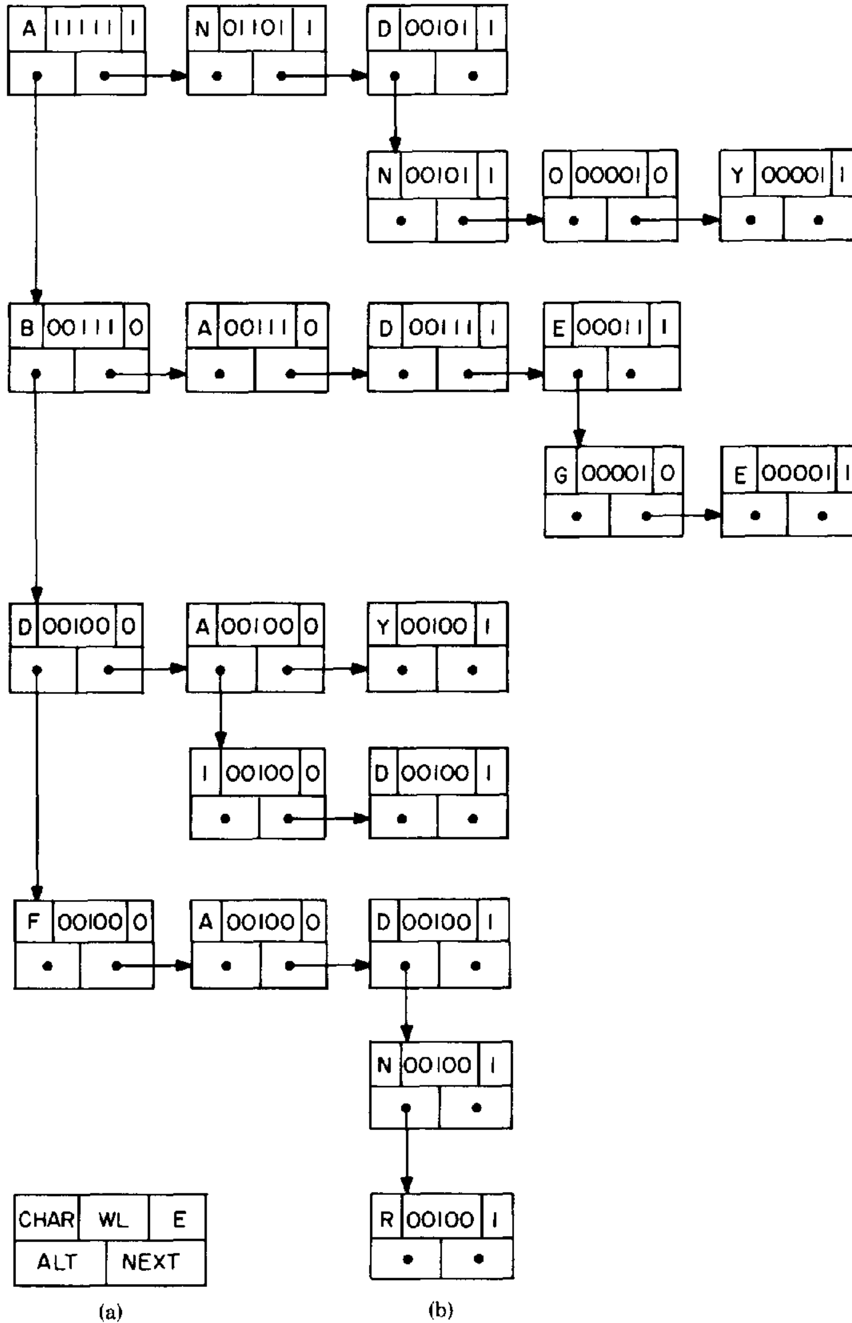

Fig. 2. Trie structure representation: (a) the fields of a typical record, and (b) trie of the dictionary A, AN, AND, ANN, ANNOY, BAD, BADE, BADGE, DAY, DID, FAD, FAN, FAR.

A node with token L_j is a NEXT descendant (son) of a node with token L_i if L_iL_j is a substring of a dictionary word. A node with token L_j is an ALTERNATE descendant of a node with token L_i if L_j is an alternative for L_i in a dictionary word, where the letters preceding L_j (and L_i) are specified by the most immediate ancestor node which is a NEXT descendant. Further, it is required that if a node with token L_j is an ALTERNATE descendant of a node with token L_i, then L_j is ordinally greater than L_i; that is, L_j appears later than L_i in an ordering of the elements of L. The end-of-word bit is set if its token and the initial substring given to reach the token comprise 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.

Given a trie with fields initialized as above, the following PASCAL function, *access-trie*, is used to determine whether a given string is a prefix of an m-letter dictionary word. Specifically, *access-trie* determines whether or not character *ch* appears in an m-letter dictionary word following an initial substring (or prefix) specified by a pointer p to the first possible character following the substring. If *ch* does appear in the trie, then, as a side-effect of the function, p points to the successor node of the new prefix.

```
function access-trie(var p:ptr; ch:char):boolean;
  begin
    if (p = nil) or (p ∧ .CHAR > ch) or (p ∧ .WL[m] = 0)
      then
        begin
          p := nil;
          access-trie := false
        end
      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*)
```

Function *access-trie* is an efficient way of determining the legality, with respect to the dictionary, of a given string. Although the maximum number of recursive calls for any given initial substring is $r - 1$, the average number of calls is considerably fewer for an English language dictionary. If it were assumed that for all positions in any string the possible letters were uniformly distributed over all r possibilities, the average number of calls would be $(r - 1)/2$. Clearly this assumption is unreasonable, for if "AMPLIFYIN" is the initial substring, then there is only one possibility for the tenth position. This characteristic is reflected in the experimentation discussed in Section 6, where the average number of alternatives for all nodes in a trie representing 1724 English words is 1.62.

4. THE ALGORITHM

Simultaneous search of the VA trellis using a variable number of alternatives and the trie structure representation of a dictionary can be achieved using an $r \times m$

binary array A. Element $A[j, i]$ is set to 1 if L_j is a possible correction for X_i, and 0 otherwise. One way of initializing A is by examining the letter confusion probabilities and setting $A[j, i]$ to 1 only if the probability of recognizing L_j as X_i is greater than some predetermined threshold. Thus the paths of the trellis that need to be evaluated are only those that begin at the 1's of the first column of A and proceed through the 1's of the subsequent columns. Before evaluating a path that reaches a node from the nodes of the previous column, that path is determined to be legal with respect to the trie by using function *access-trie*. For example, consider the node labeled $P(\mathrm{N}|\mathrm{N})$ at the second stage in Figure 1. In determining the most likely lexical string reaching that node, we only need to evaluate the path from the node labeled $P(\mathrm{A}|\mathrm{A})$ since the other four paths are invalid with respect to the trie of Figure 2.

If $A[j, i] = 1$ for $1 \le j \le r$ and $1 \le i \le m$ (i.e., all r alternatives are considered for each of the m letters), then the above process guarantees that the resulting string is the most likely word of the dictionary. The optimality follows from the fact that the VA guarantees the optimal string and that the only paths of the trellis we ignore at each stage are those that do not correspond to prefixes of words in the dictionary. A formal statement of the algorithm, called procedure *text-recognize*, follows.

Symbols and Data Structures

$X = X_1 \cdots X_m$ is the input character string.
$Z = Z_1 \cdots Z_m$ is the output character string.
A is an $r \times m$ two-dimensional Boolean array.
C is a vector of r real numbers called the *cost* vector.
Q is a vector of r pointers each pointing to a record of the trie (initially all point to the root).
S is a vector of r character strings called the *survivor* vector.
$\not{b}$ is assumed to be the delimiter symbol.

```
procedure text-recognize; (*the main algorithm*)
  begin
    repeat
      getword(X);  (*read next word X*)
      dictionary-viterbi(X, Z);  (*find most likely legal word Z*)
      word-out(Z)  (*output word Z*)
    until end-of-file
  end;  (*text-recognize*)

procedure dictionary-viterbi(X1 ··· Xm; Z1 ··· Zm);
    (*given an m-letter string X = X1 ··· Xm as input, produce an m letter string Z = Z1
    ··· Zm as output*)
  begin
    initialize(A);
    dictionary-trace(A, C, S, Q, X1 ··· Xm);
    select(A, C, S, Z)
  end;  (*dictionary-viterbi*)
```

We describe in order the procedures *initialize* and *dictionary-trace*, and the function *select*. 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. This procedure refers to the following primitive function: *max*(G:**array**[1..r] **of real**; **var**

u:**integer**), which returns the maximum of the elements of G, with the side-effect that u contains the index of the maximum element.

```
procedure initialize(A);
var Y:real;
  begin
    for i := 1 to m do
      begin
        for j := 1 to r do
          begin
            A[j, i] := false;
            W[j] := log P(X_i | L_j) + log P(L_j)
          end;
        for k := 1 to d do
          begin
            Y := max(W, u);   (*determine index of maximum element u*)
              W[u] := -inf;   (*a large negative number*)
              A[u, i] := true
          end
      end;
  end;  (*initialize*)
```

An alternative way of implementing *initialize* is to select a variable number of alternatives for each letter based on a threshold t. In this case the k loop above is replaced by

```
for k := 1 to r do
  if W[k] > t then A[k, i] := true;
```

Procedure *dictionary-trace*, described next, returns a set of character strings in vector S whose costs are defined by vector C. In addition to calling function max, as defined for procedure *initialize*, this procedure refers to the following primitive procedure: *concat*($s1$, $s2$: string; L_j: **char**), which concatenates character L_j at the end of string $s2$ and returns the result in $s1$.

```
procedure dictionary-trace(A, C, S, Q, X_1 ··· X_i);
var (*C1, S1, Q1, Q2, G are local vectors of r elements*)
  begin
    if i > 1 then begin
      dictionary-trace(A, C, S, Q, X_1 ··· X_{i-1});
      C1 := C; S1 := S; Q1 := Q; Q2 := Q;
      for j := 1 to r do begin
        if A[j, i] then begin
          for k := 1 to r do
  a:          if A[k, i - 1] and access-trie(Q1[k], L_j)
                then G[k] := C1[k] + log P(X_i | L_j) + log P(L_j | L_k)
                else G[k] := -inf;
          C[j] := max(G, u);
          Q2[j] := Q1[u]; Q1 := Q;
          if (C[j] ⟨ ⟩ -inf)
            then concat(S[j], S1[u], L_j)
            else S[j] := null
        end; Q := Q2
      end
    else  (*i = 1*)
        for j := 1 to r do
          if A[j, 1] and access-trie(Q[j], L_j)
```

```
            then begin
              C[j] := log P(X1 |Lj) + log P(Lj | b̸);
              S[j] := Lj end
            else begin
              C[j] := -inf; Q[j] := nil;
              S[j] := null end
end;  (*dictionary-trace*)
```

Procedure *select* returns the most likely word by considering the cost of the transition from the final symbol to the trailing delimiter $b\!\!/$ using the cost vector C and the survivor vector S. If all the values in C are equal to minus infinity, X is rejected and a null value is returned.

```
procedure select(A, C, S, Z)
var y:real;
  begin
    for k := 1 to r do
        if A[k, m]
          then
      b: C[k] := C[k] + log P(b̸ | Lk)
        else
          C[k] := -inf;
    y := max (C, u);
    if (y > -inf) then Z := S[u]
                else Z := ' _ '   (*reject string*)
  end;  (*select*)
```

The integration of the dictionary into the VA 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 to preclude the possibility of nondictionary 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 occur when fewer than r possibilities are considered for each letter but will never happen when all candidates are allowed. This phenomenon is discussed further in Section 6.

4.1 Algorithm Modifications

A variation of procedure *dictionary-viterbi* is not explicitly to maintain survivor strings but to derive them from a vector of pointers Q. However, this would necessitate son-to-father pointers to allow a path to be traced from each node back to the first level of the trie to retrieve the string. This would yield a storage economy when the number of nodes was less than the number of locations required for the survivor strings due to the additional backward pointers.

Procedure *dictionary-viterbi* can be modified to handle the second-order Markovian assumption by making two simple changes. In step a of procedure *dictionary-trace*, replace $P(L_j \mid L_k)$ by $P(L_j \mid (S[k])^{i-1}, L_k)$, where $(S[k])^{i-1}$ de-

notes the $(i-1)$th letter of the kth string in vector S. In step b of function *select*, replace $P(b \mid L_k)$ by $P(b \mid (S[k])^{m-1}, L_k)$.

Finally, we note that procedure *text-recognize* can be implemented either as a postprocessor for text that is output by an OCR machine or can be directly applied to feature vectors that represent letter shapes. In the latter case, X consists of a set of m feature vectors instead of hard decisions output by a character classifier (see [12, 14]).

5. COMPUTATIONAL COMPLEXITY

The computational complexity of procedure *dictionary-viterbi* is derived for a fixed number of d alternatives and a fixed alphabet size r. Only the general case of $m > 1$ and $1 < d < r$ is discussed here. A *trie lookup* c is defined as the maximum number of comparisons in a call to *access-trie*. Its maximum value is $r * 4$ for an alphabet of size r, which occurs when a node has $r-1$ alternatives and the last alternative in this list is the token sought. The overall computation requirement can be expressed as a function of m, d, r, and c as

$$D(m, d, r, c) = D_a(m, d, r) + D_p(m, d, r, c), \tag{4}$$

where $D_a(m, d, r)$ is the requirement for the selection of alternatives (procedure *initialize*) and $D_p(m, d, r, c)$ is the requirement for the path tracing (procedures *dictionary-trace* and *select*). If addition and comparison are each defined as requiring one unit of computation, then

$$D_a(m, d, r) = m(r + (r-1)d) \quad \text{units.} \tag{5}$$

Path tracing involves two steps: *dictionary-trace* is the path tracing itself, and *select* is the evaluation of the last letter to blank transition. Procedure *dictionary-trace* requires

$$m + (m-1)[r + (2+c)dr + 2d^2] + r(1+c) + d \quad \text{units,}$$

which is an upper bound occurring when all trie lookups are successful. Procedure *select* requires exactly $(2r + d - 1)$ units. Therefore,

$$D_p(m, d, r, c) \le m + (m-1)[r + (2+c)dr + 2d^2] + r(3+c) + 2d - 1. \tag{6}$$

Combining (4)–(6),

$$D(m, d, r, c) \le m(r + (r-1)d + 1) + (m-1)[r + (2+c)dr + 2d^2] + r(3+c) + 2d - 1.$$

This bound on the complexity of procedure *dictionary-viterbi* increases linearly with m, r, and c, and quadratically with d, when the other three parameters are fixed. When r is fixed (26 for the English alphabet), the constant terms are high enough to suggest that judicious selection of parameters is called for to minimize the run-time cost of the algorithm. Since values of m, r, and c are dictated by the text, alphabet, and dictionary, respectively, the number of alternatives d is the only parameter that can be controlled. The optimal d can be empirically determined as described in Section 6.

6. EXPERIMENTAL RESULTS

To determine the performance and efficiency of the dictionary-viterbi algorithm (DVA) with actual text and to compare this with variations of the VA, a database was established and experiments were conducted.

English text in the computer science domain (Chapter 9 of *Artificial Intelligence*, P. H. Winston, Addison-Wesley, Reading, Mass., 1977) containing 6372 words was entered onto a disk file. Unigram and first-order transitional probabilities were estimated from this source; these probabilities were computed using all word lengths and positions. The log-transitional probabilities are given in Table I. 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 log-confusion probabilities given in Table II were used for all experiments.

A dictionary of 1724 words containing 12231 distinct letters was extracted from the correct text and a trie was constructed for use by the DVA. The size of this dictionary would be reasonable for a more general text enhancement system since it has been noted [3] that 1000 words can summarize 78 percent of written English. 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 themselves (path length 1), and 701, 240, and 128 nodes having alternate path lengths of 2, 3, and 4, respectively.

The approximate number of words required to load the DVA program and its data structures on a CDC Cyber 730 are: program (10K), trie (18K), confusion and first-order transitional probability tables (1.5K)

An example of garbled text is given in Figure 3a, and its correction produced by the DVA with $d = 8$ is given in Figure 3b. It can be observed that the text produced by the DVA is significantly better than the text input to it. We also note the shortcomings of the DVA in that the word "lomputer" was rejected, the garbled words "tayfr," "bm," and "bes" were erroneously corrected to "sites," "by," and "but" (instead of "layer," "be," and "few"), respectively, and the dictionary word "come" was not corrected to "some." Rejections could be eliminated by increasing the depth of search because a dictionary word that was not possible due to the constrained nature of the trellis would be located. However this would probably produce an erroneous correction in place of the rejection because, when increasing d past 6, it is highly likely that the correct alternatives for each input letter already exist in the trellis; it is just that one or more of them are excluded from a path prior to rejection-termination.

To contrast the performance and cost of the DVA and VA when applied to the complete sample of garbled text, experimentation was performed using a fixed and variable number of alternatives for both algorithms. The results are summarized in Tables III and IV where performance and cost are measured by the percentage of garbled words corrected to their original form and CPU seconds on a CDC Cyber 730, respectively. The minimum value of the depth of search (d) and the minimum threshold (t) that yield the optimum correction rate (maximum correction rate of 87 percent with minimum cost) were found to be 8 and -11,

Table I. First-order transitional probabilities. Entries are log $P(a|b)$ where rows correspond to values of b.

	A/O	B/P	C/Q	D/R	E/S	F/T	G/U	H/V	I/W	J/X	K/Y	L/Z	M/.	N
A	-INF -INF	-3.119 -4.017	-2.701 -7.801	-4.137 -2.138	-6.009 -2.591	-6.415 -1.843	-4.805 -4.400	-INF -4.112	-3.382 -5.855	-INF -7.801	-4.217 -3.909	-2.394 -7.801	-3.128 -2.602	-1.712
B	-2.491 -2.343	-6.275 -6.275	-INF -INF	-INF -2.841	-1.291 -3.017	-6.275 -4.665	-5.176 -2.841	-INF -INF	-2.561 -INF	-4.195 -INF	-INF -2.586	-1.690 -INF	-INF -4.888	-INF
C	-2.097 -1.872	-INF -INF	-4.442 -7.215	-INF -3.604	-1.926 -5.423	-INF -1.658	-INF -3.302	-1.882 -INF	-2.640 -INF	-INF -INF	-3.813 -5.605	-3.718 -INF	-INF -3.526	-5.828
D	-3.937 -2.839	-5.372 -6.982	-6.982 -INF	-4.149 -3.986	-1.588 -3.763	-6.982 -INF	-3.090 -2.975	-INF -5.372	-2.338 -6.289	-INF -INF	-INF -4.679	-5.190 -INF	-INF -0.827	-6.982
E	-3.018 -5.107	-5.928 -4.743	-3.053 -4.892	-2.656 -2.126	-4.050 -2.429	-5.030 -3.573	-5.687 -5.554	-5.761 -4.092	-4.892 -5.436	-8.326 -4.022	-7.228 -5.331	-3.436 -INF	-3.370 -1.099	-2.443
F	-2.677 -1.550	-INF -INF	-INF -INF	-INF -2.677	-2.745 -INF	-3.867 -3.733	-INF -3.328	-INF -INF	-2.301 -INF	-INF -INF	-INF -5.407	-4.560 -INF	-INF -0.941	-INF
G	-3.061 -3.397	-INF -INF	-INF -INF	-INF -2.083	-1.504 -4.670	-INF -5.923	-4.131 -3.358	-2.609 -INF	-2.953 -INF	-INF -INF	-6.616 -4.537	-4.313 -INF	-5.923 -1.127	-3.438
H	-2.049 -2.583	-INF -INF	-INF -INF	-INF -4.384	-0.745 -5.888	-INF -4.183	-INF -3.691	-INF -INF	-2.218 -7.274	-INF -INF	-INF -4.016	-6.581 -INF	-7.274 -2.038	-5.195
I	-3.512 -2.349	-4.611 -5.191	-2.583 -7.136	-3.255 -3.640	-3.567 -1.986	-3.625 -2.292	-3.185 -7.830	-INF -3.918	-INF -INF	-INF -7.136	-5.057 -INF	-3.341 -4.497	-3.553 -6.220	-1.277
J	-INF -1.386	-INF -INF	-INF -INF	-INF -INF	-0.693 -INF	-INF -INF	-INF -1.569	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -3.178	-INF
K	-4.060 -INF	-INF -INF	-INF -INF	-INF -INF	-1.597 -2.502	-INF -5.447	-4.754 -5.447	-INF -INF	-2.228 -3.655	-INF -INF	-INF -4.060	-4.348 -INF	-INF -1.439	-1.272
L	-2.846 -2.929	-INF -4.404	-6.484 -INF	-3.285 -7.177	-1.354 -4.132	-5.231 -3.681	-7.177 -3.776	-7.177 -3.919	-2.096 -INF	-INF -INF	-5.098 -2.228	-2.228 -INF	-INF -1.946	-INF
M	-1.809 -2.683	-3.271 -2.135	-6.827 -INF	-INF -INF	-1.829 -2.749	-6.827 -INF	-5.217 -3.042	-INF -INF	-2.470 -INF	-INF -INF	-INF -3.020	-6.827 -INF	-3.831 -1.777	-5.035

N	-3.524 -2.634	-6.659 -5.678	-2.898 -7.758	-2.190 -7.758	-2.532 -2.599	-4.261 -2.182	-2.130 -4.622	-7.065 -4.985	-3.183 -7.758	-7.758 -INF	-4.985 -4.985	-4.713 -INF	-7.065 -1.441	-5.050
O	-5.419 -3.739	-3.809 -4.205	-4.179 -INF	-3.422 -1.939	-5.419 -3.827	-2.374 -3.191	-3.295 -2.652	-7.816 -4.415	-4.382 -2.833	-6.718 -7.816	-5.419 -6.718	-3.398 -7.123	-2.575 -2.374	-1.714
P	-2.784 -2.228	-INF -3.267	-INF -INF	-INF -1.505	-1.481 -3.927	-INF -3.457	-INF -2.664	-3.499 -INF	-3.744 -6.635	-INF -INF	-INF -5.536	-2.102 -INF	-INF -3.233	-6.635
Q	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -0.027	-3.638 -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF
R	-2.246 -2.027	-5.114 -6.070	-4.635 -INF	-3.966 -4.789	-1.473 -2.961	-4.846 -2.592	-4.042 -3.335	-6.070 -5.887	-2.629 -5.281	-INF -INF	-4.635 -3.829	-5.114 -INF	-3.851 -1.924	-4.347
S	-4.212 -3.011	-INF -3.533	-4.155 -5.792	-INF -7.738	-2.410 -3.307	-6.639 -1.866	-INF -3.295	-4.049 -INF	-2.568 -5.340	-INF -INF	-5.253 -4.100	-5.340 -INF	-4.603 -0.839	-7.738
T	-2.846 -2.587	-INF -7.379	-6.686 -INF	-INF -2.907	-2.090 -3.529	-7.379 -4.488	-INF -3.075	-1.335 -INF	-2.126 -5.433	-INF -INF	-INF -4.029	-4.083 -INF	-6.280 -1.739	-INF
U	-3.264 -6.977	-3.842 -3.982	-2.141 -INF	-3.982 -1.757	-3.193 -2.043	-6.284 -2.057	-3.264 -INF	-INF -6.977	-3.422 -INF	-INF -INF	-INF -INF	-2.304 -6.284	-2.788 -5.031	-2.660
V	-1.964 -2.794	-INF -INF	-INF -INF	-INF -INF	-0.534 -INF	-INF -INF	-INF -INF	-INF -INF	-1.585 -INF	-INF -INF	-INF -INF	-INF -INF	-INF -4.934	-INF
W	-1.916 -2.581	-INF -INF	-INF -INF	-INF -3.995	-1.988 -3.794	-INF -5.499	-INF -INF	-1.704 -INF	-1.810 -INF	-INF -INF	-INF -INF	-2.342 -INF	-INF -2.033	-3.794
X	-1.792 -INF	-INF -0.596	-3.664 -INF	-INF -INF	-INF -INF	-INF -1.959	-INF -INF	-3.664 -INF	-3.258 -4.357	-INF -INF	-INF -4.357	-INF -INF	-INF -3.664	-INF
Y	-INF -3.982	-5.591 -3.394	-2.818 -INF	-6.284 -INF	-3.576 -2.314	-INF -3.886	-6.284 -INF	-INF -INF	-4.205 -5.591	-INF -INF	-INF -INF	-5.591 -5.186	-4.338 -0.376	-5.186
Z	-3.045 -INF	-INF -INF	-INF -INF	-INF -INF	-0.442 -INF	-INF -INF	-INF -INF	-INF -INF	-1.540 -INF	-INF -INF	-INF -INF	-3.045 -3.045	-INF -INF	-INF
.	-2.207 -2.606	-3.226 -3.008	-2.751 -5.251	-3.555 -3.704	-3.475 -2.535	-3.191 -1.836	-4.282 -4.270	-3.672 -5.192	-2.386 -3.157	-6.350 -INF	-4.391 -5.752	-3.880 -7.361	-3.271 -INF	-3.737

Table II. Log-confusion probabilities.

	A/N	B/O	C/P	D/Q	E/R	F/S	G/T	H/U	I/V	J/W	K/X	L/Y	M/Z
A	-0.061 -INF	-4.623 -INF	-4.304 -INF	-7.801 -4.469	-4.400 -7.801	-INF -INF	-INF -INF	-INF -7.801	-4.505 -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF
B	-4.195 -INF	-0.084 -INF	-INF -INF	-4.665 -INF	-INF -3.972	-3.790 -INF	-INF -INF	-INF -INF	-INF -INF	-4.483 -INF	-INF -INF	-5.582 -INF	-INF -INF
C	-4.123 -INF	-INF -INF	-0.074 -INF	-4.730 -7.215	-INF -INF	-INF -4.079	-4.381 -INF	-7.215 -INF	-INF -INF	-INF -7.215	-4.270 -INF	-7.215 -INF	-INF -INF
D	-6.982 -INF	-4.343 -INF	-4.679 -INF	-0.076 -INF	-INF -INF	-INF -6.982	-6.982 -4.149	-4.149 -INF	-INF -INF	-INF -INF	-INF -6.982	-4.149 -INF	-INF -INF
E	-4.394 -INF	-INF -INF	-7.633 -INF	-INF -7.228	-0.069 -INF	-4.542 -INF	-4.663 -INF	-INF -4.022	-7.633 -INF	-INF -7.228	-INF -INF	-INF -INF	-4.249 -INF
F	-INF -4.560	-4.896 -INF	-INF -INF	-INF -INF	-3.867 -INF	-0.065 -INF	-INF -INF	-4.560 -INF	-INF -4.309	-INF -INF	-INF -INF	-INF -INF	-INF -INF
G	-INF -INF	-INF -4.670	-4.051 -6.616	-INF -INF	-3.977 -INF	-INF -INF	-0.081 -INF	-4.218 -INF	-INF -INF	-INF -4.131	-INF -INF	-INF -INF	-INF -INF
H	-INF -7.274	-INF -INF	-INF -4.330	-4.183 -INF	-INF -INF	-4.016 -INF	-4.096 -INF	-0.081 -INF	-INF -INF	-INF -INF	-INF -4.330	-7.274 -INF	-INF -INF
I	-4.303 -INF	-INF -INF	-7.830 -INF	-INF -7.136	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-0.076 -INF	-4.192 -INF	-4.192 -INF	-INF -4.396	-4.166 -INF
J	-INF -INF	-3.178 -INF	-INF -3.178	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-0.182 -INF	-INF -INF	-3.178 -INF	-INF -3.178
K	-INF -INF	-INF -4.348	-3.837 -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-3.367 -INF	-INF -INF	-0.085 -INF	-4.348 -INF	-INF -INF
L	-INF -INF	-INF -6.484	-INF -4.287	-4.404 -INF	-INF -INF	-INF -INF	-INF -7.177	-INF -INF	-INF -INF	-4.469 -INF	-4.612 -INF	-0.051 -INF	-INF -INF
M	-INF -4.881	-INF -4.629	-INF -INF	-INF -INF	-3.736 -INF	-INF -INF	-INF -INF	-INF -INF	-4.187 -INF	-INF -INF	-INF -INF	-INF -INF	-0.058 -INF

N	-INF -0.058	-7.065 -INF	-INF -4.357	-INF -INF	-INF -INF	-4.357 -INF	-INF -INF	-INF -INF	-7.758 -7.758	-4.391 -INF	-INF -INF	-INF -INF	-4.147 -7.758
O	-INF -7.816	-INF -0.050	-INF -4.558	-INF -INF	-7.816 -INF	-INF -INF	-4.598 -INF	-7.123 -INF	-7.816 -INF	-INF -INF	-4.179 -INF	-INF -INF	-4.484 -INF
P	-INF -3.927	-INF -4.070	-INF -0.061	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-4.555 -INF	-INF -INF	-INF -INF	-INF -INF	-4.437 -INF	-INF -INF
Q	-2.944 -INF	-INF -INF	-INF -INF	-INF -0.156	-INF -3.638	-INF -INF	-INF -INF	-INF -2.721	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF
R	-6.986 -INF	-4.183 -INF	-INF -INF	-7.679 -4.124	-INF -0.076	-INF -7.679	-INF -4.245	-INF -6.986	-INF -4.544	-INF -INF	-INF -7.679	-INF -INF	-INF -4.312
S	-INF -INF	-INF -INF	-4.212 -INF	-INF -4.155	-INF -7.738	-INF -0.056	-7.738 -4.742	-INF -7.738	-INF -INF	-INF -4.304	-INF -INF	-INF -INF	-INF -INF
T	-INF -INF	-8.072 -INF	-INF -INF	-4.029 -INF	-INF -4.516	-INF -4.222	-INF -0.062	-6.973 -INF	-INF -INF	-INF -INF	-INF -4.180	-INF -INF	-INF -INF
U	-INF -INF	-INF -INF	-INF -INF	-INF -4.144	-4.087 -6.977	-INF -6.977	-INF -INF	-INF -0.060	-INF -4.412	-INF -4.675	-INF -6.977	-INF -INF	-6.977 -INF
V	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -3.836	-3.836 -INF	-INF -INF	-INF -4.018	-INF -0.090	-INF -INF	-INF -3.682	-INF -INF	-INF -INF
W	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -3.995	-4.246 -INF	-INF -4.113	-INF -INF	-INF -0.070	-INF -3.995	-INF -INF	-INF -INF
X	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-3.664 -INF	-INF -INF	-INF -4.357	-INF -0.039	-INF -INF	-INF -INF
Y	-INF -INF	-INF -4.675	-INF -INF	-INF -4.492	-4.898 -INF	-INF -INF	-INF -INF	-INF -1.137	-INF -4.898	-INF -5.591	-INF -INF	-INF -1.197	-1.108 -4.898
Z	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF -INF	-INF 0.000

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. Figure 9-8 illustratms.

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. This argues egainqt dhe idea that studming neurons cap lead to muah of an understanding about intelligence. Understandinw them beautifullu and entirelu cbn no more pvoduse an uncerstanding of intelligendc than a complete undetstanding of transistors can uyeld insight into how a computer can understand scenes or reqpknds to english. People cannot think ib we pluci the neurons out of their brains but if we studu only neurons, we have onlm a slender chance of getting at intellkgence.

Still, come critics argum that aomputerc cannkt by intelligenx becavse digital hardware made of silicom can never do what braips made of neurons do. Their pocition is weakened bu the hierarchu argument and the lack of solid knowledge about what the ufthynkablm tanglad neuropil does.

If we look at what has produced ——— intelligence so far, we see multiple layers, each of which rests on primitives of the next sites down, forming a hierarchical structure with a great deal interposed between the intelligent program and the transistors which ultimately support it. Figure 9-8 illustrates.

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. This argues against the idea that studying neurons can lead to much of an understanding about intelligence. Understanding them beautifully and entirely can no more produce an understanding of intelligence than a complete understanding of transistors can yield insight into how a computer can understand scenes or responds to english. People cannot think if we pluck the neurons out of their brains but if we study only neurons, we have only a slender chance of getting at intelligence.

Still, some critics argue that computers cannot be intelligent because digital hardware made of silicon can never do what brains made of neurons do. Their position is weakened by the hierarchy argument and the lack of solid knowledge about what the unthinkably tangled neuropil does.

Fig. 3. Example of DVA performance: (a) garbled text, and (b) corrected text.

respectively. The cost of these runs was 1896 and 1492 seconds, respectively. Thus the optimum performance of the DVA was achieved with variable alternatives at 79 percent of the cost required with a fixed number of alternatives. The average number of alternatives per letter with $t = -11$ was 7.25, with a maximum of nine alternatives (for H, Q, U), and a minimum of two alternatives (for Y).

To show the effects of differing levels of contextual information on performance at the optimum parameter settings ($d = 8$), the DVA was run using only top-down information, by setting all transitional probabilities equal, and without the trie, thus using only the bottom-up information provided by the transitional probabilities. The correction rates were 82 and 35 percent, respectively, both less than the 87 percent provided by the combination approach. Performance of the DVA and VA using second-order transitional probabilities and a variable number of alternatives was also considered. The results are given in Table V. Whereas

Table III. Experimental Results Using First-Order Statistics and a Fixed Number of d Alternatives[a]

	DVA		VA	
d	Percent correct	Time (seconds)	Percent correct	Time (seconds)
1	1	687	1	660
2	39	763	23	694
3	61	859	29	720
4	74	988	33	761
5	81	1147	34	816
6	85	1340	35	882
7	86	1588	35	960
8	87	1896	35	1057
9	87	2261	35	1148

[a] Entries show percent if garbled words corrected and the time to process the text of 6372 words on a CDC Cyber 730.

Table IV. Experimental Results Using First-Order Statistics and a Variable Number of Alternatives

	DVA		VA	
t	Percent correct	Time (seconds)	Percent correct	Time (seconds)
−2	0	437	0	433
−3	0	450	0	428
−4	0	474	0	451
−5	0	479	0	452
−6	11	486	8	455
−7	44	540	23	481
−8	76	739	33	547
−9	83	946	35	642
−10	85	1145	35	688
−11	87	1517	35	820
−12	87	1492	35	821
−13	87	1492	35	831

Table V. Experimental Results Using Second-Order Statistics and a Variable Number of Alternatives

	DVA		VA	
t	Percent correct	Time (seconds)	Percent correct	Time (seconds)
−3	0	495	0	493
−4	0	519	0	506
−5	0	525	0	504
−6	11	530	11	507
−7	44	596	36	532
−8	77	809	58	629
−9	84	1027	62	724
−10	86	1244	63	807
−11	87	1649	64	955
−12	87	1655	64	978
−13	87	1651	64	972

the VA performance improves from 35 percent to 64 percent, DVA performance does not significantly improve beyond 87 percent. This leads to the conclusion that second-order statistics contribute significant contextual information beyond first-order statistics only in the absence of top-down lexical information.

7. SUMMARY AND CONCLUSIONS

The proposed algorithm for text recognition is able to utilize top-down information in the form of a lexicon of legal words (represented as a trie), channel characteristics in the form of probabilities that observed letters are corruptions of other letters (confusion probability table), and two types of bottom-up information: letter shapes (represented as vectors) and the probability of a letter when the previous letters are known (transitional probability table). Results of experimentation with this algorithm show a significant increase in correction rate over its predecessors that do not use lexical information and show no increase in the order of complexity. In the presence of a lexicon, second-order transitional probabilities do not contribute significant contextual information beyond first-order transitional probabilities. Owing to its superior performance, this algorithm is suggested as a word hypothesization component in a system focusing global contextual knowledge on the text recognition problem.

ACKNOWLEDGMENTS

We wish to thank Addison-Wesley for permission to utilize portions of *Artificial Intelligence* by P. H. Winston in the experimentation. We would also like to thank C. T. Chen for the program to build the trie structure from its word list.

REFERENCES

1. BLEDSOE, W.W., AND BROWNING, J. "Pattern recognition and reading by machine," in *Pattern Recognition*, L. Uhr (Ed.), Wiley, New York, 1966, pp. 301-306.
2. BOUCHARD, D.C., AND TOUSSAINT, G.T. "Heuristic search methods for efficient use of dictionary information in text recognition," Tech. Rep. SOCS 80.5, School of Computer Science, McGill Univ., May 1980.
3. DEWEY, G. *Relative Frequency of English Speech Sounds*. Harvard Univ. Press, Cambridge, Mass., 1923.
4. DOSTER, W., AND SCHURMAN, J. "An application of the modified Viterbi algorithm," in *Proc. 5th Int. Conf. on Pattern Recognition* (Miami Beach, Fla.), IEEE, New York, 1980, pp. 855-863.
5. DUDA, R.O., AND HART P.E. "Experiments in the recognition of hand-printed text: Part II—Context analysis," in *Proc. Fall Joint Computer Conference* (San Francisco, Calif. Dec 9-11), Thompson Book Co., Washington, D.C., 1968, pp. 1139-1150.
6. FORNEY, G.D., JR. The Viterbi algorithm. *Proc. IEEE 61* (March 1973), 268-278.
7. HALL, P.A.V., AND DOWLING, G.R. Approximate string matching. *Comput. Surv. 12*, 4 (Dec. 1980), 381-402.
8. HULL, J.J., AND SRIHARI, S.N. "Experiments in text recognition with the binary *n*-gram and Viterbi algorithms," *IEEE Trans. Pattern Anal. Mach. Intell. PAMI 4* (Sept. 1982), 520-530.
9. KNUTH, D.E. *Sorting and Searching*. The Art of Computer Programming, vol. 3. Addison-Wesley, Reading, Mass., 1973.
10. MUNSON, J.H. "Experiments in the recognition of hand-printed text: Part I—Character recognition," in *Proc. Fall Joint Computer Conference* (Dec. 9-11, 1968, San Francisco, Calif.), Thompson Book Co., Washington, D.C., pp. 1125-1138.
11. MUTH, F.E., AND THARP, A.L. Correcting human error in alphanumeric terminal input. *Inf. Process. Manage. 13*, (1977), 329-337.

12. NEUHOFF, D.L. The Viterbi algorithm as an aid in text recognition. *IEEE Trans. Inf. Theory IT-21* (March 1975), 222–228.
13. PETERSON, J.L. Computer programs for detecting and correcting spelling errors. *Commun. ACM 23*, 12 (Dec. 1980), 676–687.
14. SHINGHAL, R., AND TOUSSAINT, G.T. Experiments in text recognition with the modified Viterbi algorithm. *IEEE Trans. Pattern Anal. Mach. Intell. PAMI-1* (April 1979), 184–192.
15. SHINGHAL, R., AND TOUSSAINT, G.T. A bottom-up and top-down approach to using context in text recognition. *Int. J. Man-Mach. Stud. 11* (1979), 201-212.
16. SRIHARI, S.N., AND BOZINOVIC, R. "A string correction algorithm for cursive script recognition," in *Proc. 6th Int. Conf. on Pattern Recognition* (Munich, W. Germany, Oct. 19–22), IEEE, New York, 1982
17. SRIHARI, S.N., HULL, J.J., AND CHOUDHARI, R. "Integration of bottom-up and top-down contextual knowledge in text error correction," in *Proc. AFIPS Nat. Computer Conf.*, AFIPS Press, Arlington, Va., pp. 501–508.
18. TOUSSAINT, G.T. The use of context in pattern recognition. *Pattern Recogn. 10*, (1978), 189–204.

Received June 1982; revised August 1982; accepted August 1982