Hierarchical File Organization and Its Application to Similar-String Matching

TETSURO ITO and MAKOTO KIZAWA
University of Library and Information Science, Ibaraki, Japan

The automatic correction of misspelled inputs is discussed from a viewpoint of similar-string matching. First a hierarchical file organization based on a linear ordering of records is presented for retrieving records highly similar to any input query. Then the spelling problem is attacked by constructing a hierarchical file for a set of strings in a dictionary of English words. The spelling correction steps proceed as follows: (1) find one of the best-match strings which are most similar to a query, (2) expand the search area for obtaining the good-match strings, and (3) interrupt the file search as soon as the required string is displayed. Computational experiments verify the performance of the proposed methods for similar-string matching under the UNIX™ time-sharing system.

Categories and Subject Descriptors: H.3.2 [**Information Storage and Retrieval**]: Information Storage; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; I.2.7 [**Artificial Intelligence**]: Natural Language Processing

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: File organization, hierarchical clustering, linear ordering, similarity, best match, good match, spelling correction, similar-string, text editor, office automation

1. INTRODUCTION

Workers in an automated office or laboratory manage their tasks of creating, analyzing, copying, transforming, interchanging, and transmitting information with a computer-assisted system [6, 8, 27]. The preparation of documents, which includes the complicated and time-consuming processes of typing, proofreading, updating, and so on, can also be handled with filing, editing, and formatting programs.

One of the most useful facilities in this mode of preparation is that used to analyze documents for the detection and correction of terminal input errors in the written text [11, 15]. A typical error-correcting program looks for the misspellings in the text from beginning to end. Once a misspelled string has been found, the user has several options, for example, try to correct it, mark it as correct, or do nothing [18]. However, when the user does not know the exact spelling of the string to be corrected, he or she can do nothing at all. This can

Authors' address: University of Library and Information Science, Kasuga 1-2, Yatabe-machi, Tsukuba-gun, Ibaraki-ken 305, Japan.

often occur in the course of looking up a person's name in a directory. A spelling program therefore should guide the user to the correct strings by displaying the various probable correct spellings.

Two approaches to file searching can be employed for this purpose [11]. One is to formulate an efficient exact matching algorithm such as that used in the DEC-10 SPELL program. Such an algorithm generates all possible correct strings, and then tests whether or not they are in a dictionary. This is viable only when the probable input errors are fully analyzed beforehand. The other is based on a best-match (or nearest neighbor) technique which appears in general pattern classification problems. The entire set of strings is grouped into affinity classes so that similar strings within a cluster are jointly retrieved. The latter seems more promising for obtaining not only the correct versions of misspelled words but also legitimate variations (e.g., grammatical transformations and abbreviations).

Various authors have devised file organization and search methods for obtaining and processing clusters of the records of interest. (Each record is generally characterized by a set of known attributes. For strings, attributes simply consist of symbols drawn from some alphabet.) Burkhard and Keller [3] and Shapiro [23] have formulated efficient strategies for finding the best-match records based on the triangle inequality of a distance function. Jardine and van Rijsbergen [13] and Salton and Wong [22] have formulated partial file search procedures by constructing classes of related records. Techniques for solving the best-match problem without employing the triangle inequality have been proposed by Bentley, Friedman, and Finkel [1, 2, 10]. These methods, however, need assumptions, such as that the records constitute a metric space, that similar records can be grouped into a cluster, or that the distance function has a monotonic property.

We present here a new method of organizing a file for finding records that are similar to any input query, with the condition that a similarity measure, which has the reflexive and symmetric properties, is given. The file organization steps arrange the records linearly according to their similarity values [12], and group them into hierarchical clusters which are specified by representative records constituting, in turn, the lower level clusters. Thus the given records are stored at the highest level after clustering, and in the lower levels the representative records are stored. An entire configuration, called here an HL-*file* (*H*ierarchically organized file based on a *L*inear ordering), thus takes the form of a *multiway tree* [14].

Consider a spelling correction situation in which the task is to correct misspellings or to find the grammatical variants. Since the input strings are likely to be only minor variants of the correct ones, the problem can be viewed as retrieving the records from a file of strings that are highly similar to the inputs. The file search policy here consists of two processes, *best-match* and *good-match* searches, to speed up the total text editing task. First, one of the match strings showing the largest similarity value to the input is retrieved. The file search continues while the user examines the correctness of the displayed result. It locates the good matches, that is, those showing sufficiently large values of similarity to the unsatisfactory inputs, by visiting the clusters adjacent to the one to which the best match belongs. Thus the time spent in checking the correctness of the displayed result can be overlapped with retrieving further similar strings.

The proposed methods are experimentally examined for a set of about 24,000 strings under the UNIX[1] time-sharing system [17].

2. HIERARCHICAL FILE ORGANIZATION

In this section the notions of similarity measure, linear ordering, and hierarchical clustering, which will be used for organizing HL-files, are explained.

2.1 Structure of HL-Files

HL-files are a class of multiway trees [14] devised for external searching. The generation of HL-files, which employs a hierarchical clustering scheme often used in general pattern classification applications, proceeds by grouping the linearly ordered records into clusters of size not greater than m. *Linear ordering* means that similar pairs of records are placed close together in the file, and dissimilar pairs far from each other. Each cluster obtained is specified by another record, called a *representative*. Representatives are further clustered in the same way, and finally a hierarchical tree structure of a certain height h (≥ 1) is generated as an HL-file.

The file structure has two kinds of nodes, one storing a cluster of input records at the highest level, the other storing that of representatives at the lower levels. Each representative conveys a pointer to its corresponding cluster at the next level up. In the auxiliary memory, a cluster is expressed as a *block* or bucket of size m. Let us refer to the HL-file structure for the records $r0$, $r1$, ..., $r14$ in Figure 1. A given record is expressed as $\times$, and a representative as x or X located in the center of its cluster (size ≤ 4). The distance between two records is assumed to be inversely proportional to the similarity between them. A constructed file of height 2 is seen in Figure 2. The representative rp2:2, for example, specifies the cluster $C2:2 = [r14, r0]$ at level 2, and is specified by rp2 at level 0. It is to be noted that similar records are grouped together. A cluster of input records (at level 2) or representatives (at levels 0 and 1) is stored in a block of size 4, which is shown by a dashed rectangle. Records at level 2 contain no pointer information, since they are terminals.

An HL-file somewhat resembles the clustered file proposed by Salton and Wong [22]. Clustered files, however, have some defects: the clustering process, which needs, for example, a time-consuming density test or an iterative grouping [21], occasionally produces overlapping clusters, and this reduces the storage utilization and file search efficiency. The already existing file structure may not be preserved by the continued insertion and/or deletion of records. The clustering criterion for HL-files, on the other hand, is very simple. Further, since all records are stored after arranging, the insertion and deletion operations could be done in a manner similar to those used with B-trees [5, 14]. (The main difference between HL-files and B-trees is that in the HL-file search the objective is to retrieve additionally any record similar to the query even when an exact match record is not found.)

[1] UNIX is a trademark of Bell Laboratories.

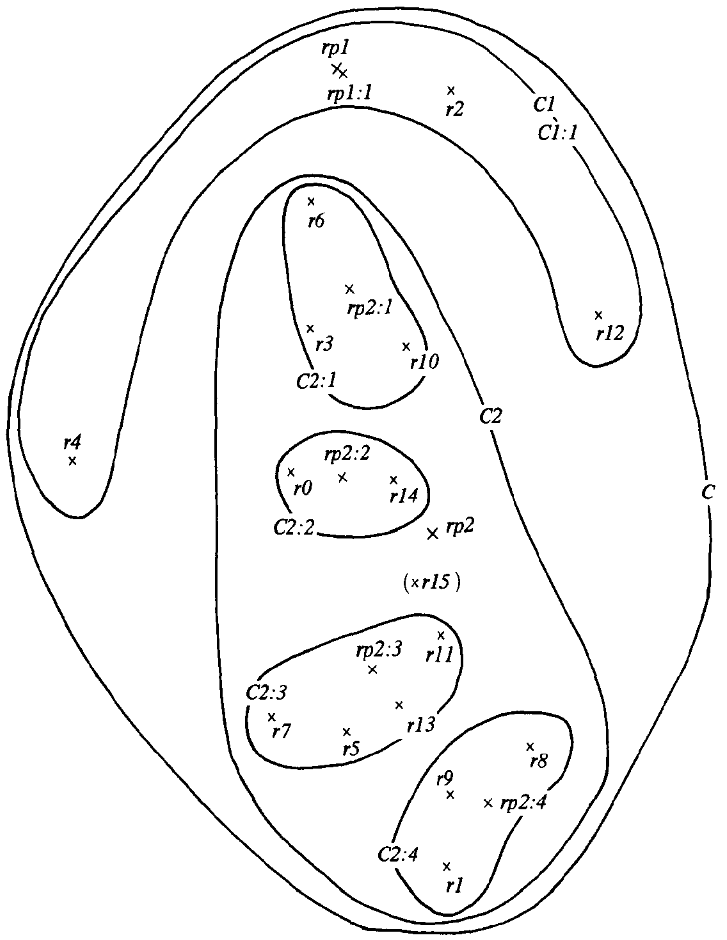


Fig. 1. Hierarchical clusters for the records $r0, r1, \ldots, r14$. The records encircled by a line constitute a cluster.

2.2 Similarity Measure and Linear Ordering

Let $R = \{r_0, r_1, \ldots, r_{n-1}\}$, $n > 0$, be a set of records to be stored. In an information retrieval environment, each record r_i, $0 \le i < n$, is generally characterized by assigning certain attributes, and is expressed as an ω-dimensional ($\omega \ge 1$) weight vector $(\delta_i^1, \delta_i^2, \ldots, \delta_i^\omega)$ of the attributes attached to r_i [22]. Various authors have proposed similarity measures for the purpose, for example, of pattern classification, speech understanding, or database organization. Among such measures a set-theoretical function [19] has often been used because it is a simple process to compute the numbers of common and distinct attributes of any pair of records. The value for r_i and r_j measured by this function becomes

$$\frac{\sum_k \min(\delta_i^k, \delta_j^k)}{\sum_k \max(\delta_i^k, \delta_j^k)}. \tag{1}$$

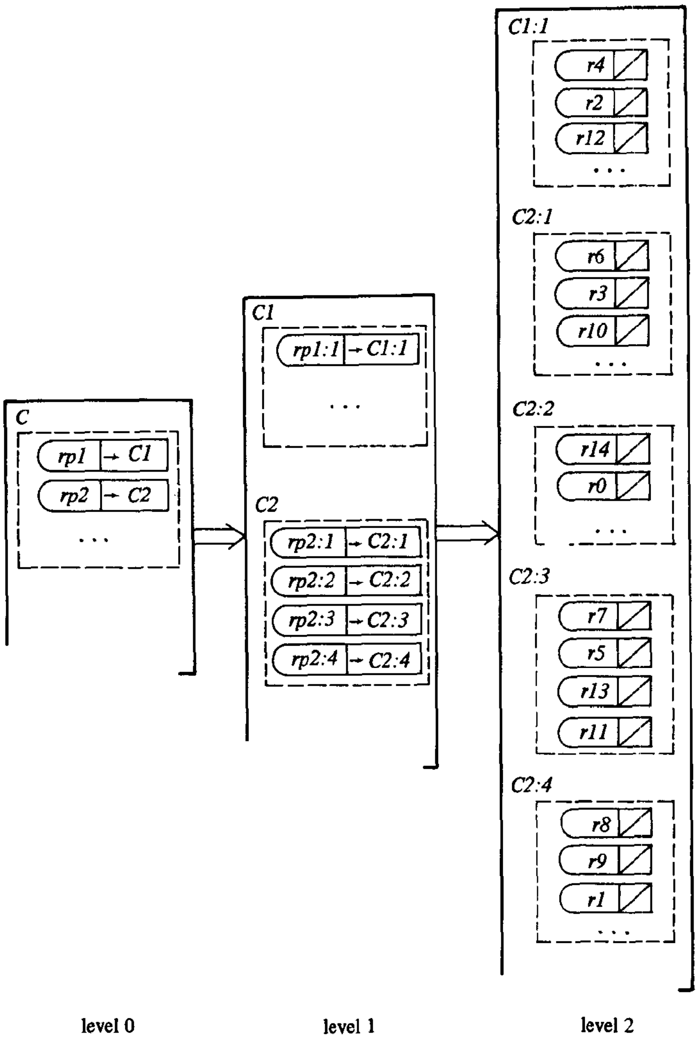


Fig. 2. An HL-file of height 2 for the records $r0, \ldots, r14$ in Fig. 1.

Salton gives a cosine function for clustering document records in [21]. It can be noticed that any reasonable similarity measure s defined on $R \times R$ will satisfy

$$s(r_i, r_j) = \alpha, \qquad \alpha = 1 \quad \text{if} \quad r_i = r_j, \quad \text{reflexivity} \tag{2}$$

and

$$s(r_i, r_j) = s(r_j, r_i), \quad \text{symmetry} \tag{3}$$

for $\alpha \in [0, 1]$ (or sometines $\alpha \in [-1, 1]$). Often s will be formulated to satisfy $s(r_k, r_i) > s(r_k, r_j)$ (or $s(r_k, r_i) \gg s(r_k, r_j)$) when r_k is more (much more) similar to r_i than to r_j. The problem of retrieving similar records can now be stated as follows.

For any query r_q find r_i in the file much faster than r_j, if r_i and r_j satisfy $s(r_q, r_i) \gg s(r_q, r_j)$.

When the records have been arranged linearly, it is easy to solve this problem, but in general the linear ordering characteristic is restrictive for any set of records. Let us now introduce another measure t derived from s. Consider an edge-weighted graph $G(C)$ for a subset C ($|C| = m + 1 \leq n$) of R, where node r_i and weighted edge $r_i - r_j$ correspond to record r_i and node pair (r_i, r_j) with similarity value $s(r_i, r_j)$, respectively. Then $t(r_i, r_j)$ is set to the minimum of the weights of the edges of a path connecting r_i and r_j in a *maximal spanning tree* MST (C) of $G(C)$ [7]. (In any edge-weighted graph G, a sequence of edges joining two nodes is called a path, and a subgraph with no closed path is called a tree. Furthermore, a tree which contains all nodes of G is called a spanning tree of G. If we define the weight of a tree to be the sum of the weights of its constituent edges, then a maximal spanning tree of G is a spanning tree whose weight is maximal among all spanning trees of G [26].)

The following matrix rearrangement procedure [12] (called a two-dimensional sorting operation) is formulated to arrange the records according to their similarity values with respect to t.

[Two-dimensional sorting operation TSO]

Note: Suppose that the records in C are numbered by finite integers 0, 1, . . . , m stored in array $l(l[0] = 0, l[1] = 1, \ldots, l[m] = m)$. (Characters between /* and */ are comments.)

```
Procedure TSO(C);
begin
  for $i := 0$ to $m - 1$ do
    begin      /* Matrix rearrangement */
      {Arrange $[l[i], l[i+1], \ldots, l[m]]$ so that $t(r_{l[i]}, r_{l[i]}) \geq t(r_{l[i]},$
       $r_{l[i+1]}) \geq \cdots \geq t(r_{l[i]}, r_{l[m]})$}
    end;
  for $i := 0$ to $m$ do
    begin
      {Renumber $r_{l[i]}$ as $r_i$}
    end
end;
```

Theorem 1 holds for t.

THEOREM 1. *All the records in C can be arranged linearly by TSO as*

$$\begin{aligned} t(r_i, r_j) &\geq t(r_i, r_k),\ 0 \leq i \leq j \leq k \leq m; \\ t(r_i, r_j) &\leq t(r_i, r_k),\ 0 \leq j \leq k \leq i \leq m. \end{aligned} \tag{4}$$

PROOF. See Appendix.

We can see that the following inequality holds between s and t.

$$s(r_i, r_j) \leq t(r_i, r_j), \qquad \forall\, r_i, r_j \in C. \tag{5}$$

If r_i is most similar to r_j with respect to s (that is, r_i and r_j have the largest s-value), then $s(r_i, r_j)$ equals $t(r_i, r_j)$. Thus t gives the upper bound to the original s. Figure 3 exemplifies the steps of TSO for the set $C = [r14, r10, r3, r6, r0]$. Values of s for all pairs of the records are given in Figure 3a, and those of t in

$$C = [\,r14_0, r10_1, r3_2, r6_3, r0_4\,]$$

$$\begin{array}{rl|ccccc|}
 & & r14_0 & & \cdots & & r0_4 \\ \hline
 & r14_0 & 100 & 16 & 16 & 6 & 22\\
 & r10_1 & & 100 & 18 & 7 & 12\\
s\colon & r3_2 & & & 100 & 20 & 16\\
 & r6_3 & & * & & 100 & 3\\
 & r0_4 & & & & & 100\\ \hline
\end{array}$$

(a)

$$\begin{array}{rl|ccccc|}
 & & r14_0 & & \cdots & & r0_4 \\ \hline
 & r14_0 & 100 & 16 & 16 & 16 & 22\\
 & r10_1 & & 100 & 18 & 18 & 16\\
t\colon & r3_2 & & & 100 & 20 & 16\\
 & r6_3 & & * & & 100 & 16\\
 & r0_4 & & & & & 100\\ \hline
\end{array}$$

(b)

$$Tso(C) = [\,r14_0, r0_1, r10_2, r3_3, r6_4\,]$$

$$\begin{array}{rl|ccccc|}
 & & r14_0 & & \cdots & & r6_4 \\ \hline
 & r14_0 & 100 & \mathbf{22} & 16 & 16 & 16\\
 & r0_1 & & 100 & \mathbf{16} & 16 & 16\\
t\colon & r10_2 & & & 100 & \mathbf{18} & 18\\
 & r3_3 & & * & & 100 & \mathbf{20}\\
 & r6_4 & & & & & 100\\ \hline
\end{array}$$

(c)

$$\begin{array}{rl|ccccc|}
 & & r14_0 & & \cdots & & r6_4 \\ \hline
 & r14_0 & 100 & 22 & 16 & 16 & 6\\
 & r0_1 & & 100 & 12 & 16 & 3\\
s\colon & r10_2 & & & 100 & 18 & 7\\
 & r3_3 & & * & & 100 & 20\\
 & r6_4 & & & & & 100\\ \hline
\end{array}$$

(d)

Fig. 3. Matrix expressions of the similarity values for C. Rearranging matrix (a) results in (d). Each element contains $100 \times s(r_i, r_j)$ or $100 \times t(r_i, r_j)$. The values of (c) in boldface are sufficient for clustering.

Figure 3b. (The edges for an MST(C) are $r14$-$r0$, $r14$-$r10$, $r10$-$r3$, and $r3$-$r6$.) TSO arranges C in order to satisfy inequalities (4), as seen in Figure 3c. Figure 3d shows a matrix representation of the s-values for the arranged records [$r14$, $r0$, $r10$, $r3$, $r6$], where pair ($r14$, $r0$), for example, has a large s-value compared with ($r14$, $r10$), ($r14$, $r3$), and ($r14$, $r6$); and is thus placed adjacently.

Throughout this paper the records showing the largest s-value to r_q are called r_q's *best matches*, and the ones showing sufficiently large s-values are called r_q's *good matches*.

2.3 Hierarchical Clustering

The object of file searching is to find the best matches and good matches, so the file organization may be viewed as consisting of the clustering of similar records. A set C of the records arranged is clustered by the following simple procedure.

[Clustering procedure CLS]

Note: The set $C = [r_0, r_1, \ldots, r_m]$ to be clustered is the output of TSO. Variable thd, storing a threshold value for clustering, is set to 1. CLS returns the position c in C of the record which specifies the subclusters.

```
Procedure CLS($c$, $C$);
begin
  for $i := 0$ to $m - 1$ do
    begin    /* $c \equiv \{i{:}min(t(r_i, r_{i+1}) \mid 0 \le i < m)\}$ */
      if $thd \ge t(r_i, r_{i+1})$ then $c := i$;
      $thd := min(thd, t(r_i, r_{i+1}))$
    end
end;    /* Subclusters are $C_1 \equiv [r_0, \ldots, r_c]$ and $C_2 \equiv [r_{c+1}, \ldots, r_m]$.*/
```

Thus the clustering is attained by examining the t-values of adjacent pairs. Procedure CLS (c, C) operating on the records in Figure 3 results in $C_1 = [r14, r0]$ and $C_2 = [r10, r3, r6]$, and the value of thd is set to 0.16. The best match to both $r10$, and $r6$ is $r3$; all of these are in the same subcluster. The following theorem holds for the resulting subclusters.

THEOREM 2. *An s-value for a pair of records in the different subclusters is smaller than or equal to thd.*

PROOF. By inequality (5). □

It is possible that some of the good matches to any r_i ($\in C$) could be included in the same subcluster as r_i.[2] For the best matches we have the following corollary.

COROLLARY. *Any record r_i in C_1, $|C_1| \ge 2$ (or in C_2, $|C_2| \ge 2$), and its best match, which are adjacent in C, are still adjacent in C_1 (C_2).*

PROOF. If $s(r_i, r_j) <$ thd for any $r_j \in C_1$, then, by (4), $t(r_i, r_j) <$ thd, which contradicts the result of CLS. Thus $s(r_i, r_k) \ge$ thd for some $r_k \in C_1$. By Theorem 2, r_i and its best match are in the same subcluster. □

To determine which clusters contain the best and/or good matches to a query, let us introduce the notion of representatives. A representative rp for a cluster C is another record characterized by the attributes of the records in C. The similarity between rp and r_i is measured by any s_R satisfying

$$s_R(r_i, \mathrm{rp}) \ge s(r_i, r_j), \qquad \forall r_j \in C. \tag{6}$$

Then the *cutoff criterion* for reducing the number of clusters to be examined can be specified as follows.

For a query r_q and a stored record r_i, any cluster C satisfying $s_R(r_q, \mathrm{rp}) \le s(r_q, r_i)$ contains no record r_j such that

$$s(r_q, r_j) > s(r_q, r_i). \tag{7}$$

The repeated clustering of the representatives produces the hierarchically organized clusters shown in Figures 1 and 2, where the given records are in the highest level, and the representatives are in the lower levels.

Some sort of balancing of the cluster is required for efficient file search and organization [22]. According to CLS, the obtained subclusters are disjoint but not

[2] If pairs of the records showing the same t-value in the rearrangement steps of TSO are further arranged according to their s-values, then we can obtain more refined subclusters, including many good matches.

identically sized. There now follows the procedure MRG for merging similar subclusters which are small in size.

[Merging procedure MRG]

Note: The outputs of CLS(c, C) are C_1 and C_2. Let rp_1 and rp_2 be their corresponding representatives. Further, let D be the cluster containing the representative for the (updating) cluster C.

```
Procedure MRG($C_1$, $C_2$);
begin
  {Find a cluster $C'$ whose representative $rp'$ satisfies
   $s(rp_1, rp') = max(s(rp_1, rp^*) \mid rp^* \in D$ and $\mid C_1 \cup C^* \mid \le m)$};
     /* $C^*$ is the cluster represented by $rp^*$ */
  if $C'$ exists then $C_1 := C_1 \cup C'$;
  {Find a cluster $C''$ ($\neq C'$) whose representative $rp''$ satisfies $s(rp_2, rp'') = max(s(rp_2,$
   $rp^*) \mid rp^* \in D$ and $\mid C_2 \cup C^* \mid \le m)$};
  if $C''$ exists then $C_2 := C_2 \cup C''$
end;
```

A set of clusters containing a small number of records results in a significant amount of wasted storage space, since any cluster is stored as a block of fixed size. Therefore, we can save space by merging similar clusters into one. CLS occasionally distributes a record r_i and its best match r_b among different subclusters C_1 and C_2 (which can be caused by the fact that $\mid C_1 \mid = 1$, i.e., $C_1 = [r_i]$). However, when some record is added to C_2 at the later stage of file organization, C_2 will be divided into two, and then by MRG r_i and r_b will be again in the same cluster. This can save time in file searching because these are jointly retrieved.

3. SEARCHING HL-FILES

The discussion above has stated that a record and its best match are to be stored in the same block. The generation of an HL-file is therefore done by adding successively a new record r_q adjacent to its best match. We present a best match search algorithm which specifies recursively the higher level block whose representative is most similar to r_q.

[Best match search procedure BST]

Note: BST is a recursive procedure. One r_b (initially set to a dummy record showing value 0 to any record) of the best matches to r_q is retrieved. Variables v (initially 0) and ptr (initially the location of a block at the lowest level) store the level and the location of a block under search, respectively. A block pointed by ptr at level v is denoted by C^v[ptr].

```
Procedure BST($v$, ptr);
begin
  bs1: while ($v \ge 0$) and ($v < h$) do
         begin
           {Search $C^v$[ptr] for a new record $rp_c$ most similar to $r_q$ with respect to $s_R$};
           if ($rp_c$ exists) and ($s_R(r_q, rp_c) > s(r_q, r_b)$) then
             begin      /* Forward step */
               ptr := location of the block represented by $rp_c$; $v := v + 1$; BST($v$, ptr)
             end
           else
             begin                  /* Backward step. Consider $C^v$[ptr] to be an */
               $v := v - 1$; return    /* already examined block.                    */
             end
         end;
```

```
bs2: if $v = h$ then
        begin
          {Search $C^h[ptr]$ for a candidate best match $r_c$};
          if $s(r_q, r_c) > s(r_q, r_b)$ then $r_b := r_c$;
          $v := v - 1$; return
        end
end;
```

The file structure is updated when r_q is a record to be added.

[File organization procedure ORG]

Note: ORG begins with the block C_b^h containing the best match r_b to r_q. Variables are initialized as follows: rp, rp′, rp″ and rp_1 to dummy record, rp_2 to r_q, ptr to the location of C_b^h, and v to h.

```
Procedure ORG($v, ptr, rp, rp', rp'', rp_1, rp_2$);
begin
  $C := (C^v[ptr] - [rp, rp', rp'']) \cup [rp_1, rp_2]$;
  if $|C| > m$ then
    begin
      if $v = 0$ then
        begin    /* Increase the height of the file */
          $h := h + 1$; $v := v + 1$
        end;
      $CLS(c, TSO(C))$; $C_1 := [r_0, \ldots, r_c]$; $C_2 := [r_{c+1}, \ldots, r_m]$; $MRG(C_1, C_2)$
    end
  else
    begin
      if $v = 0$ then return;
      TSO(C); $C_1 := [r_0, \ldots, r_m]$; $C_2 := \varnothing$
    end;
  {Store $C_1$ and $C_2$ as the blocks at level $v$};
  $rp$ := representative for $C^v[ptr]$;
  $ptr$ := location of the block containing $rp$;
  $rp'$ (or $rp''$) := representative for the block merged into $C_1$ ($C_2$);
  $rp_1$ (or $rp_2$) := representative for $C_1$ ($C_2$);
  $v := v - 1$; $ORG(v, ptr, rp, rp', rp'', rp_1, rp_2)$
end;
```

Let us see how algorithms BST and ORG work by adding $r15$ to the file structure in Figure 2. The position of $r15$ is shown in parentheses in Figure 1. Cluster $C2:3$ containing a candidate $r11$ for the best match is first found in step bs1 which selects in turn the representatives rp2 and rp2 : 3. This step also checks whether or not $r11$ is the true best match, and finally it notices that no further search is needed. After the best match $r11$ is found, ORG works as follows: Record $r15$ is stored in $C2:3$, and is adjacent to $r11$. That is, the cluster to be updated at level 2 becomes [$r7$, $r5$, $r13$, $r15$, $r11$]. This cluster is arranged by TSO as [$r7$, $r5$, $r13$, $r11$, $r15$], and then clustered by CLS as C_1 = [$r7$, $r5$, $r13$, $r11$] and C_2 = [$r15$]. Of these, C_2 is merged by MRG with $C2:2$ = [$r14$, $r0$], since the representative for C_2 shows the largest s-value for rp2 : 2 among rp2 : 1, rp2 : 2, and rp2 : 4 in $C2$ ($|C_2 \cup C2:1|$, $|C_2 \cup C2:2|$, $|C_2 \cup C2:4| \leq 4$). The above C_1 and C_2 = [$r15$, $r14$, $r0$] are stored as two blocks, $C2:3$ and $C2:2'$, respectively, at level 2. The cluster to be updated at level 1 becomes [rp2 : 1, rp2 : 4, rp2 : 3, rp2 : 2′], where rp2 : 2′ is the representative for $C2:2'$. Since the number of the records in this cluster is 4, this is stored, after arranging, as block $C2'$ at level 1. Similarly,

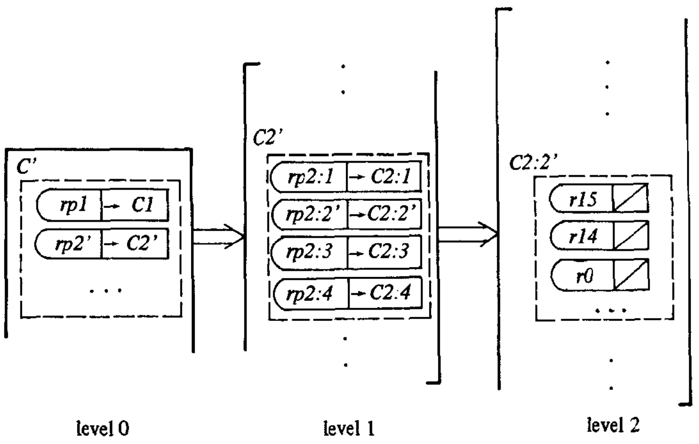


Fig. 4. The updated structure of Fig. 2 caused by the addition of $r15$. The changed clusters and their representatives are written as C' and rp′, respectively.

the block at level 0 becomes [rp1, rp2′] (rp2′ is the representative for $C2'$). The updated structure of Figure 2 is given in Figure 4.

When the obtained best match does not match the user's requirement, the good match file search follows BST. This process begins with the block C_b^h containing r_b, and gradually expands the target blocks to be searched checking the user's satisfaction with the displayed results.

[Good match search procedure GUD]

Note: GUD conveys a variable u ($<h$) to restrict the searching of the extra blocks. The blocks adjacent to C_b^h at a level higher than or equal to u are examined. Array B (initially $B[0] = r_b$ of size κ (>1) stores the required number of the good matches.

```
Procedure GUD($v, ptr, \kappa$);
begin
  gd1: {Same as bs1 except that the first line is "while ($v \geq u$) and ($v < h$) do"};
  gd2: if $v = h$ then
         begin
           {Search $C^h[ptr]$ for $\kappa$ good matches, and merge them into $B$};
           {Display $B$}; $r_b := B[\kappa - 1]$; $v := v - 1$; return
         end;
  gd3: return
end;
```

For query $r15$, six good matches, {$r14$, $r0$} (in $C2:2'$ containing $r15$ at level 2) and {$r7$, $r5$, $r13$, $r11$} (in $C2:3$ adjacent to $C2:2'$ at level 1), are found by GUD ($u = 1$).

A set of representatives works as a *hierarchical directory* to enable the rapid location of the required records. The lower level records are much smaller in number and much more frequently referred to than the higher level ones. Thus, the efficient file search can be attained by employing storage hierarchy hardware [20], where the lower level blocks are stored on the solid state memory and the higher level blocks on the disk memory.

```
johnson
                    j          o          h          n          s          o          n
BTPTN (bin):   0...1...0|0...1...0|0...1...0|0...1...0|0...1...0|0...1...0|0...1...0
               1   10  32 1   15        8          14         19         15     1   14  32

                 j        o        h        n        s        o        n
BTPTN (hex):   00400000|00020000|01000000|00040000|00002000|00020000|00040000
                j  o  h  n  s  o  n
ASCII (hex):   6A|6F|68|6E|73|6F|6E

dodgson
                    d          o          d          g          s          o          n
BTPTN (bin):   0...1...0|0...1...0|0...1...0|0...1...0|0...1...0|0...1...0|0...1...0
                   4          15         4          7          19         15         14

                 d        o        d        g        s        o        n
BTPTN (hex):   10000000|00020000|10000000|02000000|00002000|00020000|00040000
                d  o  d  g  s  o  n
ASCII (hex):   64|6F|64|67|73|6F|6E

{johnson, dodgson}
                 d   j          o         d   h          g  n        s          o          n
BTPTN (bin):   0.1..1..0|0...1...0|0.1..1..0|0..1..1.0|0...1...0|0...1...0|0...1...0
                 4  10        15         4   8          7  14       19         15         14

                 dj       o        dh       gn       s        o        n
BTPTN (hex):   10400000|00020000|11000000|02040000|00002000|00020000|00040000
```

Fig. 5. Examples of BTPTN code for words which are seven characters in length (bin = binary notation; hex = hexadecimal notation).

4. SIMILAR-STRING MATCHING

In this section the spelling problem is discussed from the viewpoint of searching an HL-file for strings highly similar to the misspelled inputs.

4.1 An HL-File for the Spelling Problem

Each character of a string is generally represented by ASCII (or EBCDIC) code within the computer memory. We employ here another representation, named BTPTN code. Let η be a size of character set S. For example, η is 32 for $S = \{$a, b, ..., z, $\sigma_1, \ldots, \sigma_6\}$ ($\sigma_1, \ldots, \sigma_6$ are dummy characters for future extension). Then the BTPTN code for a character σ is an η bit-pattern $\delta^1\delta^2 \cdots \delta^i \cdots \delta^\eta$, where $\delta^i = 1$ and $\delta^1 = \cdots = \delta^{i-1} = \delta^{i+1} = \cdots = \delta^\eta = 0$ if σ is in the ith position in S. Note that δ^i is considered to be a weight of the ith *attribute* for specifying σ. Any string w is a sequence of characters of length ω; thus the BTPTN code for w is an $\omega \times \eta$ bit pattern $\delta^{11}\delta^{12} \cdots \delta^{1\eta}\delta^{21} \cdots \delta^{jk} \cdots \delta^{\omega 1} \cdots \delta^{\omega\eta}$. (N.B.: the subpattern $\delta^{j1}\delta^{j2} \cdots \delta^{j\eta}$ is said to be in the jth character position.) The codes for johnson and dodgson are shown in Figure 5a.

Next let us see the BTPTN code for a representative of a cluster C of strings. Each representative rp is specified by the set of the attributes δ^{jk} assigned to the strings in C. Formally rp is expressed in a way similar to that for strings, except that δ^{jk} is 1 if the jth character of some string in C appears at the kth position in S. Figure 5b depicts the BTPTN code for the representative of the cluster {johnson, dodgson}. Though the BTPTN code requires more computer storage than the ASCII code, no ASCII code exists for a representative. In an HL-file the BTPTN code and the ASCII code are employed to store a representative and a string, respectively.

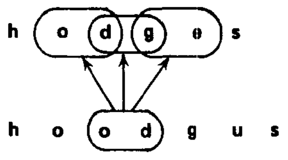


Fig. 6. Computing similarity value s(hoodgus, hodges), where $\gamma(k)$ = od(3).

For the quantitative comparison of two strings, Findler and Leeuwen [9] have defined a measure essentially equivalent to a set-theoretical function. We here modify this to cope with the common spelling mistakes (i.e., single-character omission, insertion and substitution, and reversal of adjacent characters [11, 15]). The similarity s between a pair of records for strings (or representatives) is given by

$$s(r_i, r_j) = \frac{\sum_{\gamma(k)} \min(P_\xi(r_i, \gamma(k)), P_\xi(r_j, \gamma(k)))\,|\gamma(k)|}{\sum_{\gamma(k)} \max(P_\xi(r_i, \gamma(k)), P_\xi(r_j, \gamma(k)))\,|\gamma(k)|}. \tag{8}$$

Here $\gamma(k)$ is a possible subpattern in r_i or r_j specifying a substring, which begins with the kth character position ($|\gamma(k)|$ is the length of the substring). Function $P_\xi(r_i, \gamma(k))$ takes 1 if r_i contains a matched subpattern $\gamma'(k')$ such that $\gamma = \gamma'$ and $k' \in [k + \xi, k + \xi + |\gamma(k)| - 1]$ $(1 \le k \le \omega - |\gamma(k)| + 1, -1 \le \xi \le 1)$, and 0 otherwise ($\xi$ shifts the beginning position of $\gamma(k)$). It is assumed that $\gamma'(k')$ does not match any $\gamma(k)$ in r_j twice. The sum is over all distinct subpatterns in r_i or r_j. (All $\gamma(k)$ in r_i (or r_j) are distinct subpatterns. For $\gamma(k)$ in r_j and $\gamma'(k')$ in r_i they are distinct if $\gamma \ne \gamma'$ or $\min(P_\xi(r_i, \gamma(k)), P_\xi(r_j, \gamma'(k'))) = 0$.) The measure s_R between a string r_i and a representative rp_j is given by

$$s_R(r_i, \mathrm{rp}_j) = \frac{\sum_{\gamma(k)} \min(P_\xi(r_i, \gamma(k)), P_\xi(\mathrm{rp}_j, \gamma(k)))\,|\gamma(k)|}{\sum_{\gamma(k)} P_\xi(r_i, \gamma(k))\,|\gamma(k)|}. \tag{9}$$

The sum is over all subpatterns in r_i. It is evident that s and s_R satisfy inequality (6). Figure 6 depicts schematically the process of computing the s-value for hoodgus and hodges, where subpatterns $\gamma(k)$ ($|\gamma(k)| \le 2$) in hoodgus (or hodges) are h(1) (= h(1)), o(2) (= o(2)), o(3), d(4) (= d(3)), ..., ho(1) (= ho(1)), oo(2), od(3) (= od(2)), ..., us(6). In this figure the specified subpattern $\gamma(k)$ is od(3). P_ξ(hoodgus, od(3)) (or P_ξ(hodges, od(3)) takes 1, when ξ is set to 0 (−1). By summing P_ξ(hoodgus, $\gamma(k)$) and P_ξ(hodges, $\gamma(k)$) according to (8), we have s(hoodgus, hodges) = 0.45. This is still a large value, even if character "o" is inserted, and "u" is substituted for "e".

The text editing process including the correction of misspelled inputs will be done in an interactive fashion using a visual display unit. When the correct string for an input string is displayed on the screen, no further search is needed. In addition, since misspelled strings are only minor variants of the correct ones, a considerable portion of the time for file searching will be spent in checking the retrieved results. By considering these facts, the following file search policy has been devised.

(1) Find a candidate for the best match by setting a rather large threshold value τ to avoid the extra forward steps. In this case, since most of r_c such that $s(r_q, r_c) > s(r_q, r_b)$ satisfy $s_R(r_q, r_c) \ge s_R(r_q, r_b)$, lines 5 and 6 of bsl in BST are

C = [*r0*:**carlson**, *r1*:**goodrum**, *r2*:**alwood**, *r3*:**fenlon**,
r4:**bubenko**, *r5*:**rogers**, *r6*:**senko**, *r7*:**roget**,
r8:**goodwin**, *r9*:**woodrum**, *r10*:**hinton**, *r11*:**hodges**,
r12:**sloane**, *r13*:**rodgers**, *r14*:**johnson**, *r15*:**dodgson**]

	r0	*r1*	...					*r7*			...					*15*
r0	100	0	6	16	2	2	3	0	2	0	12	2	0	0	22	22
r1		100	2	0	0	6	0	3	35	72	0	13	3	15	2	12
r2			100	3	0	0	3	0	2	2	3	0	6	0	2	2
r3				100	2	0	20	0	2	0	18	0	3	0	16	12
r4					100	2	0	3	0	0	2	2	2	2	5	2
r5						100	0	52	3	6	0	23	3	66	6	9
r6							100	0	0	0	7	0	3	0	6	3
r7								100	3	3	3	20	3	33	3	6
r8									100	22	2	13	3	12	5	15
r9										100	0	13	3	15	2	12
r10					*						100	3	3	0	16	12
r11												100	6	45	6	29
r12													100	6	6	2
r13														100	2	22
r14															100	26
r15																100

Fig. 7. Similarity matrix for C. Each matrix element holds 100 × $s(r_i, r_j)$.

changed to:

```
if (rp$_c$ exists) and ($s_R(r_q$, rp$_c) \geq \max(\tau, s_R(r_q, r_b)$)) then
  begin
```

(2) Find the good matches by examining blocks adjacent to the one containing the candidate best match, and display these good matches. In this case τ is set to a rather small value.

(3) Interrupt (2) as soon as the required record is displayed.

Figure 7 shows the s-values for a set of 15 names used by Hall [11] to exemplify Salton's work on a clustered file. (Record $r0$ is added here.) Their relative positions were shown in Figure 1, where the similarity value for two strings is inversely proportional to their separation. Figure 8 demonstrates the steps for retrieving (1) a candidate (or candidates) r_c for the best match that shows the s_R-value greater than 0.50 to the query r_q and (2) three good matches in the blocks that are adjacent to r_c at a level not lower than 0. For hoodgus ($=r_q$) the block C' at level 0 is examined in step bs1 of BST, and rp2$'$ is extracted by considering that $C2'$ will contain the best match. In step bs2 the block $C2{:}3$ whose representative rp2 : 3 is most similar to r_q in $C2'$ is searched to find a candidate best match $r11$ ($=r_c$). It can be noticed that $r11$ is the true best match in the entire file, since $s_R(r_q$, rp2 : 4) ($=s_R(r_q$, rp2 : 2$'$) $>$ $s_R(r_q$, rp2 : 1)) and $s_R(r_q$, rp1) are smaller than $s(r_q, r11)$. Similar steps for finding good matches to hoodgus and best/good matches to fenkon and goodge are summarized in this figure.

		QUERIES					
		hoodgus		fenkon		goodge	
Steps for *Bst/Gud*	level v	rp_c or r_c	s_R (or s)	rp_c or r_c	s_R (or s)	rp_c or r_c	s_R (or s)
B s t *bs1*	0	*rp2'*	0.89	*rp2'*	1.00	*rp2'*	1.00
bs1	1	*rp2:3*	0.63	*rp2:1*	1.00	*rp2:3*	0.68
bs2	2	*r11*	0.57 (0.45)	*r3*	0.68 (0.52)	*r11*	0.62 (0.45)
bs1	1	*rp2:4*	0.42	*rp2:2'*	0.31	*rp2:4* : *rp2:2'*	0.62 : 0.50
bs1	0	*rp1*	0.21	*rp1*	0.25	*rp1*	0.31
G u d *gd2*	2	*r11*	0.57 (0.45)	*r3*	0.68 (0.52)	*r11*	0.62 (0.45)
		r13	0.42 (0.26)	*r6*	0.62 (0.52)	*r13*	0.62 (0.40)
		r5	0.10 (0.06)	*r10*	0.31 (0.18)	*r7*	0.06 (0.03)
gd1	1	*rp2:4*	0.42	*rp2:2'*	0.31	*rp2:4*	0.62
[*gd2*]	[2]	[*r1*	0.42 (0.26)]	[*r14*	0.31 (0.16)]	[*r1*	0.62 (0.40)]
		rp2:2' : *rp2:1*	0.42 : 0.05	*rp2:4*	0.06	*rp2:2'*	0.50
gd3	0			(Return since $v<u$)			
Best & Good Matches		*r11*:hodges		*r3*: fenlon		*r11*:hodges	
		r13:rodgers		*r6*: senko		*r13*:rodgers	
		r1: goodrum		*r10*:hinton		*r1*: goodrum	

Fig. 8. BST and GUD steps for the best·/good·match strings to hoodgus, fenkon, and goodge (τ = 0.50 in BST, and τ = 0.30, κ = 3, and u = 1 in GUD). Here rp_c (or r_c) is the extracted record in step bs1 (bs2) or gd1 (gd2).

4.2 Performance Analysis

The structure of HL-files takes the form of multiway trees. In a multiway tree, a large value of branching, that is, block size m, reduces the number of file accesses to auxiliary memory. But, on the other hand, it increases the time needed to transfer a block into main memory, as well as the time to perform an internal block search. Extra memory is also needed for storing the blocks to be searched.

The searching of a multiway tree is, in general, quite straightforward. It is a top-down or forward process, and only one block at each level is referred against every query. Though the total time T required to retrieve any record in a tree depends on the branching factor m, the characteristics of auxiliary memory, and the length of the records, it is easy to find the optimal multiway tree which minimizes T [14, 25].

Obtaining the optimal HL-file structure is a complicated process, since the file search steps backtrack down the hierarchy to check whether or not the retrieved records are the true best and/or good matches. Though the advantage of the tree structure seems to be diminished in HL-files, the proposed approach is worthwhile if only a few extra blocks need searching.

We attempt in the following an approximate analysis of the expected time to search for a candidate best match. The size of a record (including a pointer) at level v is denoted by $R(v)$, and the size of a page for storing a block or blocks by PGS. Furthermore, the number of blocks visited at level v during the file search is denoted by BA(v), which depends on the clustering criterion CLS. Assuming that any block includes m' records (that is, that any block is m'/m full), the average search time $t(v)$ at level v is given by[3]

$$t(v) = (t_{\text{sim}} \cdot m' + (t_{\text{sk}} + t_{\text{tr}} \cdot \text{PGS}) \cdot \lceil m \cdot R(v)/\text{PGS} \rceil) \cdot \text{BA}(v). \tag{10}$$

Here t_{sim} is the average time for similarity computation against each record, t_{sk} is the average seek time, and t_{tr} is the transfer rate of auxiliary memory employed. Since $\lceil m \cdot R(v)/\text{PGS} \rceil$ is the number of pages to store a block of size m at level v, $\lceil m \cdot R(v)/\text{PGS} \rceil \cdot \text{BA}(v)$ indicates the number of page faults occurring at this level. (*Notation*: $\lceil x \rceil$ is an integer greater than or equal to x and smaller than $x + 1$.) The optimal structure is the one which minimizes the total time:

$$T = \min_{\text{BA}(v)} \sum_{v=0}^{h} t(v) \tag{11}$$

under the condition that $m'^h < n \le m'^{h+1}$.

Next let us see how many blocks are considered to be worth examining in BST. Any input string r_q can be expressed as

$$r_q : a_1, a_2, \ldots, a_\omega, \tag{12}$$

and any representative rp^v at level v as

$$\text{rp}^v : b_1^v, b_2^v, \ldots, b_\lambda^v, b_{\lambda+1}^{v*}, \ldots, b_\omega^{v*}, \qquad 1 \le \lambda < \omega, \tag{13}$$

where a_i $(1 \le i \le \omega)$ and b_i^v $(1 \le i \le \lambda)$ are characters, and b_i^{v*} $(\lambda < i \le \omega)$ is a set of characters. The representative for {johnson, dodgson} in Figure 5 becomes b_1, b_2, b_3, b_4, b_5^*, b_6^*, b_7^* (b_1 = o, b_2 = s, b_3 = o, b_4 = n, b_5^* = {d, j}, b_6^* = {d, h}, and b_7^* = {g, n}).

In the hierarchical structure, rp^v represents many more strings than its descendant rp^u $(v < u < h)$, where

$$\text{rp}^u : b_1^u, b_2^u, \ldots, b_\lambda^u, \ldots, b_\mu^u, b_{\mu+1}^{u*}, \ldots, b_\omega^{u*}, \qquad \lambda < \mu < \omega. \tag{14}$$

[3] The probability that blocks in a page are examined successively is assumed to be small in the equation.

The representative rp_b^v for the cluster containing r_q's best match is considered to show a sufficiently large s_R-value. Therefore the BST search for the most similar record to r_q at each level will specify rp_b^v correctly in the forward step. When an incorrect representative rp^v becomes a candidate for extension, since its descendant rp^u contains in $\{b_i^u : \lambda < i \leq \mu\}$ some characters different from those of r_q, the forward search to the corresponding block is cut off at this level u. Thus BA(v) is said to depend positively on the s_R-value between a representative and an input r_q.

A lower level representative which yields a large s_R-value represents more strings than a higher level one in relation to block size m. It can be concluded that the block access frequency BA depends positively on the number n_m of strings represented by a representative. Though the probability of examining blocks at a lower level seems very high, the total number of the blocks is extremely small. As a result, the distribution of BA is schematically explained as

$$\begin{bmatrix} n_m: & \text{small} & \text{middle} & \text{large} \\ \text{BA:} & \text{low} & \text{high} & \text{low} \end{bmatrix}. \tag{15}$$

The extra space is required for storing a set of representatives as a hierarchical directory. For an HL-file of height h, the size DT of the directory is given by

$$\begin{aligned} \mathrm{DT} &= \sum_{v=0}^{h-1} m \cdot R(v) \cdot m'^{v} \\ &\cong m \cdot R(0) \cdot \frac{m'^{h}}{m' - 1}, \qquad m'^{h} \gg 1, \end{aligned}$$

since $R(0) = R(1) = \cdots = R(h-1)$. On the other hand, the size ST of the space for input strings is given by

$$\mathrm{ST} = m \cdot R(h) \cdot m'^{h}. \tag{17}$$

For a 26-letter alphabet $R(0) \cong 4R(h)$ because the BTPTN code for a character needs 4 bytes. Thus DT will become smaller than ST in an HL-file with $m' > 5$.

4.3 Computational Evaluation

The proposed methods were experimentally examined under the UNIX time-sharing system on a PDP 11/44 computer. The strings were drawn from a set of the words in the dictionary (named */usr/dict/words* in UNIX's file directory) prepared for various test processings. This contains about 24,000 English words.

Using algorithm ORG to update the already existing file structure by the repeated insertion of the records is not so efficient when a large number of strings are to be stored at a time. This is especially true when the given strings are arranged in lexical order. An HL-file in this connection was organized by assuming that any pair of strings physically close in the dictionary would exhibit large similarity values. The best-match search therefore could be eliminated from ORG. (This expedient reveals itself when grammatical transformations of the inputs are being retrieved.) The uppercase characters were mapped into the lowercase ones, all the representatives were truncated to eight characters in length, and the similarity values were measured by eqs. (8) and (9) after setting $|\gamma| \leq 2$.

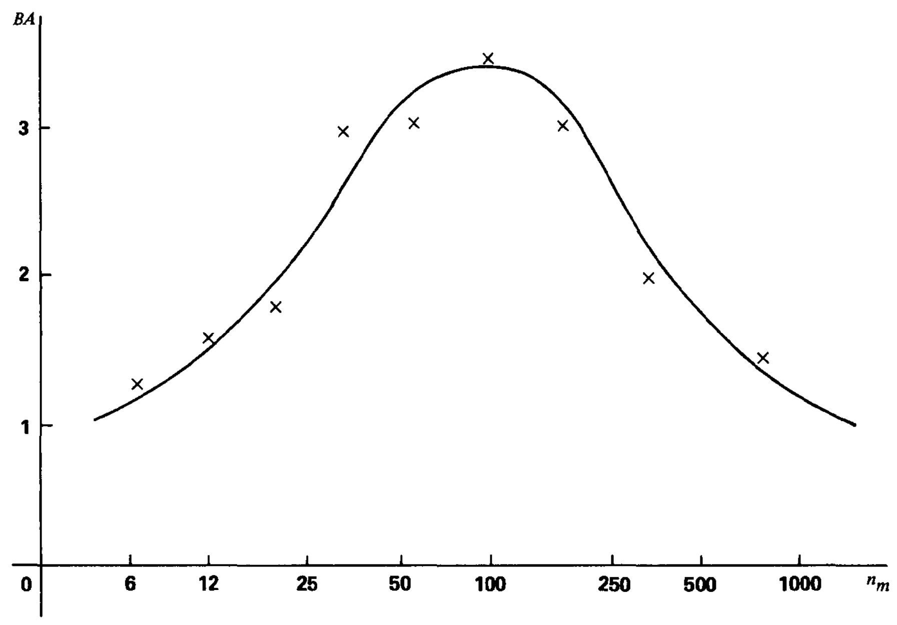


Fig. 9. Block access frequency BA plotted as a function of the number n_m of strings represented by a representative record.

4.3.1 *Optimal Block Size.* It is very time consuming to determine experimentally the optimal block size over a set Z of all 24,000 strings. For an input r_q which begins with a correct character a_1, the representative rp^0 with $b_1^0 = a_1$ will show the largest similarity among others at level 0. In this case the strings with a_1 are examined in BST. On the other hand, for an input with an incorrect first character, some of the representatives are wrongly assumed to include the best match as their descendants. However, as discussed in Sect. 4.2, this process of visiting improper blocks is cut off in the early stage of file searching.

A preliminary experiment for a set A ($|A| \cong 1700$) of the strings which began with the character "a" was made to predict the optimal block size for Z. Figure 9 shows the average block access frequency over 60 spellings with typical mistakes. By this figure, T ($= \sum t(v)$) is computed for constructing the optimal HL-file structure. In our experimental environment, the auxiliary storage device was a DEC RL02 disk cartridge. Table I contains the values of the parameters used in the course of the simulation study. The value of PGS was set to 512 bytes, which is the page size of the UNIX file. Figure 10 gives the estimated values of T for sets A and Z. The optimal structure was obtained for $m \cong 10$ in both cases.

4.3.2 *Best and Good Matches.* Table II shows the number of the records at level v in the optimal HL-files HL_A (for set A) and HL_Z (for set Z) with $m = 12$. Since $m \cdot \mathrm{R}(v) < 512$, each block is included in one page. Both structures are at least $\frac{2}{3}$ full except at the l west level. This indicates the efficient storage utilization of the proposed file design. The size of the space for storing representatives in HL_Z is, by Tables I and II, $466m \cdot \mathrm{R}(0)$ ($= 1057m \cdot \mathrm{R}(4)$), and that for 24,000 strings

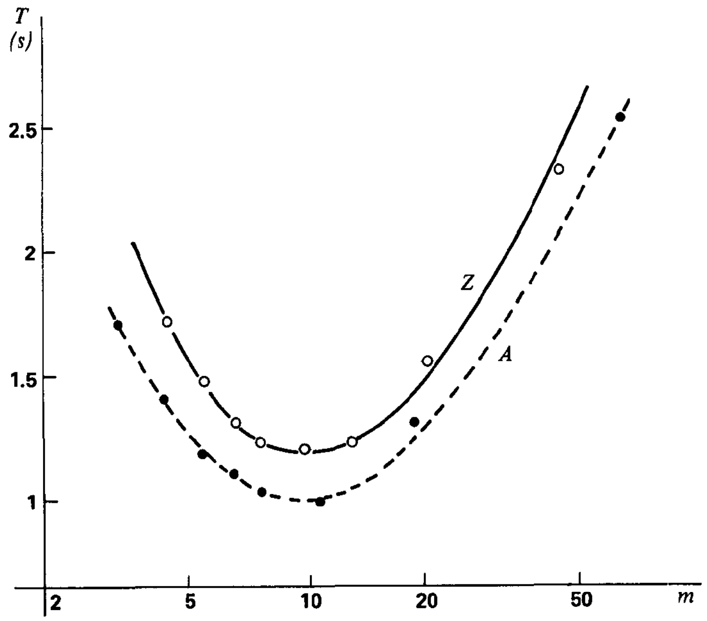


Fig. 10. Total search time T plotted as a function of the block size m for sets A and Z.

TABLE I. The Values of the Parameters for Computing T ($= \sum t(v)$)

Parameter		Value
t_{sim}		10 *ms*[a]
t_{sk}		55 *ms*
t_{tr}		2.5 μs/byte
m'/m		2/3[a]
PGS		512 bytes
$R(v)$	$(v = h)$	15 bytes[b]
	$(v < h)$	34 bytes[b]

[a] Computationally evaluated value.
[b] Representatives are truncated.

TABLE II. The Number of Blocks at Level v in HL_A and HL_Z

	HL file	
v	HL_A	HL_Z
0	1	1
1	3	6
2	28	49
3	221	410
4	*	3134

is $3134m \cdot R(4)$. Thus the directory requires only one-fourth the total space.[4] GUD ($\kappa = 4$) is used to see how many similar records were stored adjacently in a cluster. Experimental results indicated that three good matches (one is the exact match) to a stored record, on average, shared the same block.

Table III shows how BST works well in exact-string matching (i.e., retrieving the strings which exactly match the inputs). The structure of HL-files is analogous

[4] A directory can be reduced in size by employing data compression techniques [16]. There will be a subpattern which is not necessary for distinguishing a representative from others in a cluster (e.g., this is expressed as $b_{\lambda+1}^{v*} \cdots b_{\omega}^{v*}$ in (13)). Furthermore, since strings are given in lexical order, every representative in a cluster will possess an identical subpattern at the front of its BTPTN code. We can apply a data compression scheme to delete these redundant subpatterns.

TABLE III. The Number of Blocks at Level v Visited During the Exact and Best Match Searches to HL_A and HL_Z

	Exact matches		Best matches			
			HL_A		HL_Z	
v	HL_A	HL_Z	Q_1 (*full*)	Q_1	Q_1	Q_2
0	1	1	1	1	1	1
1	1.4	1.0	2.6	1.4	1.0	1.2
2	2.7	1.6	7.3	3.4	1.5	2.6
3	1.2	2.3	1.7	1.6	3.8	4.3
4	*	1.2	*	*	1.5	1.4

to that of B-trees. Both have a hierarchical directory as an index to speed retrieval of the records in the leaf nodes. B-trees are very efficient in maintaining the tree-structured files. A B-tree search algorithm examines only one block at each level of the tree, and then has a search time proportional not to n but to log n [14], n being the number of input records. Algorithm BST for an HL-file instead examines extra blocks at the middle levels. Yet by comparing the result for HL_A and HL_Z, we can see that the number of blocks visited depends on the height of the files. Thus the exact matches are said to be found by BST in the logarithmic time as seen in B-tree searching.

The best-match search results for a set Q_1 of 60 queries are also given in Table III. Each query began with the character "a" and met the common typing mistakes. The "full" for HL_A in the table means that BST searches the listed number of the blocks to find the true best match among possible strings. The other figures are for the number of blocks visited until a candidate best match is found. About half the time is spent checking the validity of the candidate best match. Table III also includes the statistics for HL_Z against Q_1 and another set Q_2 of 60 queries. Each member of Q_2 began with a non-"a" character, and also required the best-match search.[5] Though 14 of 120 best matches are wrongly retrieved, rapid location of the candidate best match, which spent the time independently of the file size, was attained. Thus, an efficient file search is formulated by employing the policy proposed in Section 4.1, at the expense of a small degradation in precision of retrieval. It is noted that BST does not always find the required string because of the difficulty of formalizing a similarity measure reflecting the user's intention. The approach of finding candidate best matches is viable especially in an on-line environment. In the experiment, though the best match r_b to the misspelled query r_q: ansent was hansen, the required spelling r_c: absent is retrieved as a candidate much faster than r_b. Other examples for (r_q, r_b, r_c) are (arrington, barrington, arlington), (chep, chef, cheap), and (thaught, haughty, taught).

[5] In this case, since at least one of the first two characters is correct, the value of $P_\xi(r_i, \gamma(k))$, where $|\gamma(k)| = 1$, was set to 3 for the matched subpattern in the character position 1 or 2 in r_i.

TABLE IV. Number of Blocks at Level v Visited During Good Match Search to HL_Z and Ratio Percentage of Number of Corrected Misspellings to that of Inputs Requiring Good Match Search ($\kappa = 2, u = 1$)

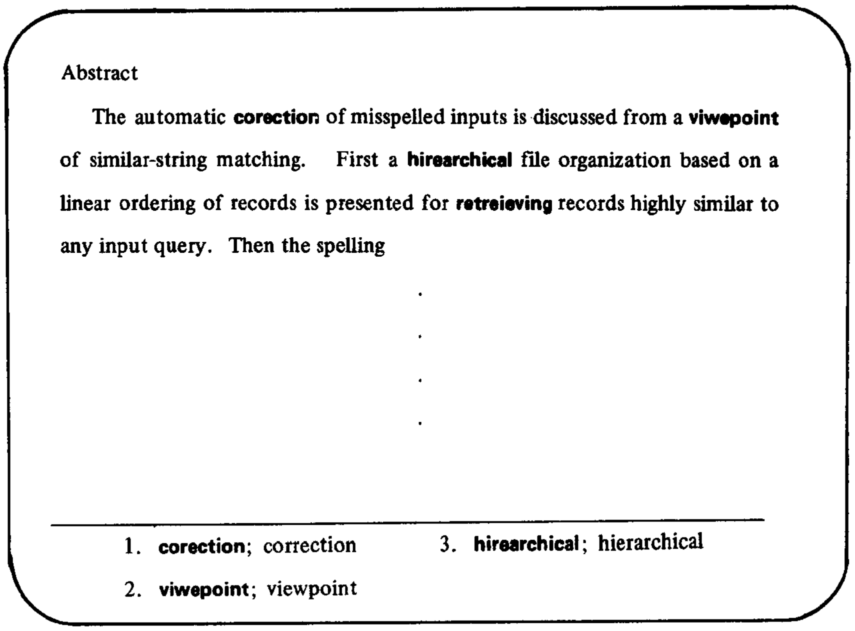
	u		
v	3	2	1
0	*	*	*
1	*	*	*
2	*	*	0.7
3	*	2.0	2.8
4	0.9	1.7	1.9
%	28	64	78

The queries resulting in unsatisfactory searches in BST call the good-match search GUD. The file search statistics are shown in Table IV. Each table element gives the expected number of blocks at level v examined, while GUD expands the search to the blocks adjacent to r_c at a level higher than or equal to u. In this process, misspelling r_q: anie required the search of the additional three blocks at level 4 and one block at level 3 for the correct spelling r_{co}: annie, after r_c: anise was found by BST. Similar examples are (r_q, r_c, r_{co}): (forthy, forth, forty, (supar, supra, super), (intmate, inmate, intimate), and (filfil, filial, fulfil), each of which require the additional block searches (0, 0, 0, 0, 1), (0, 0, 0, 0, 2), (0, 0, 0, 2, 6), and (0, 0, 2, 11, 3) at levels (0, 1, 2, 3, 4), respectively. The percentage value in Table IV is the ratio of the number of the corrected misspellings to that (= 14) of the queries requiring the good-match search. Three of these queries were not corrected even when GUD searched the entire file. Thus 97.5 percent (= 117/120) of the total required strings were found by BST and GUD. (The rest of the uncorrected misspellings were in fact more similar to the other words than they were to their "correct" versions.)

5. USER INTERFACE DESIGN

Though BST and GUD search an HL-file effectively in most cases, some of the inputs call for the examination of many extra blocks. To overcome this defect, a batch processing strategy could be adopted, one in which a few misspelled strings are corrected at a time. While the user examines the alternative candidate words for the misspelled ones, the file search proceeds for the rest of the inputs.

Let us estimate the user's performance using the keystroke-level model proposed by Card and Moran [4]. In this model the task of the user consists of two parts: (1) to assess whether or not a displayed string is a correct one, and (2) to press the specified key following the user's decision. The time for this task was estimated experimentally with 40 pairs of misspellings and their candidate best matches. Ten people were asked to decide the correct versions of the misspellings by pressing the "y" or "n" key. They were told that most of the best matches would be correct. In the course of this experiment, it took the subjects about 4 to



Abstract

The automatic **corection** of misspelled inputs is discussed from a **viwepoint** of similar-string matching. First a **hirearchical** file organization based on a linear ordering of records is presented for **retreieving** records highly similar to any input query. Then the spelling

.

.

.

.

1. **corection**; correction
2. **viwepoint**; viewpoint
3. **hirearchical**; hierarchical

Fig. 11. The screen of the terminal designed for computer-assisted spelling correction. Candidate best matches to the three misspelled words are displayed at the bottom of the screen.

5 seconds to decide the correctness of each displayed string, to find the "y" or "n" key, and to press it. On the other hand, it took about 2 to 3 seconds on average to get the response from BST, which included the best-match search time and the system's overhead of, for example, transmitting results to the terminal, collecting job accounting information, and processing multiuser jobs. The time difference between the two is therefore dedicated to another file search.

Figure 11 depicts the screen of a terminal designed for computer-assisted spelling correction. The misspelled strings are highlighted in the context to show that correction is to take place. Three of the strings being corrected are displayed at the bottom of the screen with numbers 1, 2, and 3 and their candidate best matches. Pressing a preselected function key indicates that either (1) the correct string has been found, or (2) the correct string is to be typed, and so the file search against the string with this number is to be stopped.

6. CONCLUSIONS

A spelling program should guide the user to correct the misspellings by displaying the probable correct strings. The methods presented here solve this problem by (1) finding a candidate string for the best match, (2) expanding the target blocks to be searched for the good matches, and (3) interrupting the search as soon as the user is satisfied with the displayed result. These processes can effectively be executed on the proposed screen-oriented editor.

The associative search in pattern classification, phonemic analysis, speech understanding, natural language analysis, document information retrieval, etc., also necessitates finding records closely related to an incoming query. As presented, HL-files can be applied to those research areas as well as to string matching.

APPENDIX. PROOF OF THEOREM 1.

For simplicity, denote by $t_{i,j}$ the value $t(r_i, r_j)$. From [24] we can see that

$$t_{i,j} \geq t_{i,k},\ t_{j,k} \quad \text{iff} \quad t_{i,k} = t_{j,k}, \quad \forall\ r_i, r_j, r_k \in C. \tag{A1}$$

TSO can arrange $[t_{0,0}, t_{0,1}, \ldots, t_{0,m}]$ so as to satisfy

$$t_{0,0} \geq t_{0,1} \geq \cdots \geq t_{0,m}. \tag{A2}$$

Let us consider the following inequality for any k $(1 < k \leq m)$,

$$t_{j,k-1} \geq t_{j,k} \geq t_{j-1,k}, \qquad 1 \leq j < k. \tag{A3}$$

Consider the situation in which $j = 1$. Since $t_{0,1} \geq t_{0,k}$, by (A1) we have $t_{1,k} \geq t_{0,k}$. If $t_{0,k-1} = t_{0,k}$, then, by TSO, $t_{1,k-1} \geq t_{1,k}$; else if $t_{0,k-1} > t_{0,k}$, then, since $t_{0,1} \geq t_{0,k-1}$, we have $t_{1,k-1} > t_{0,k} = t_{1,k}$. Next, suppose that (A3) holds for any integer smaller than j (≥ 2). By reasoning similar to the above, since $t_{j-1,j} \geq t_{j-1,k}$, we have $t_{j,k} \geq t_{j-1,k}$. If $t_{i,k-1} = t_{i,k}$ for all i $(0 \leq i < j)$, then, by TSO, $t_{j,k,-1} \geq t_{j,k}$; else if $t_{i,k-1} > t_{i,k}$ for some i $(0 \leq i < j)$, then, since $t_{i,i+1} \geq t_{i,k-1}$, we have $t_{i+1,k-1} > t_{i,k} = t_{i+1,k}$ and, hence, $t_{j,k-1} > t_{j,k}$. Thus, by induction, (A3) holds for any j $(1 \leq j < k \leq m)$.

From (A2), (A3), and the reflexive and symmetric properties of t, we can derive inequalities (4). □

ACKNOWLEDGMENTS

The authors wish to express their thanks to the referees for the many helpful suggestions and comments on an earlier version of this paper.

REFERENCES

1. BENTLEY, J.L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (Sept. 1975), 509-517.
2. BENTLEY, J.L., AND FRIEDMAN, J.H. Data structures for range searching. *ACM Comput. Surv. 11*, 4 (Dec. 1979), 397-409.
3. BURKHARD, W.A., AND KELLER, R.M. Some approaches to best-match file searching. *Commun. ACM 16*, 4 (April 1973), 230-236.
4. CARD, S.K., AND MORAN, T.P. The keystroke-level model for user performance time with interactive systems. *Commun. ACM 23*, 7 (July 1980), 396-410.
5. COMER, D. The ubiquitous B-tree. *ACM Comput. Surv. 11*, 2 (June 1979), 121-137.
6. DESOUSA, M.R. Electronic information interchange in an office environment. *IBM Syst. J. 20*, 1 (1981), 4-23.
7. DUNN, J.C. A graph theoretic analysis of pattern classification via Tamura's fuzzy relation. *IEEE Trans. Syst. Man Cybern. SMC-4*, 3 (May 1974), 310-313.
8. ELLIS, C.A., AND NUTT, G.J. Office information systems and computer science. *ACM Comput. Surv. 12*, 1 (March 1980), 27-60.
9. FINDLER, N.V., AND LEEUWEN, J.V. A family of similarity measures between two strings. *IEEE Trans. Patt. Anal. Machine Intell. PAMI-1*, 1 (Jan. 1979), 116-118.
10. FRIEDMAN, J.H., BENTLEY, J.L., AND FINKEL, R.A. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw. 3*, 3 (Sept. 1977), 209-226.

11. HALL, P.A.V., AND DOWLING, G.R. Approximate string matching. *ACM Comput. Surv. 12*, 4 (Dec. 1980), 381-402.
12. ITO, T., AND KIZAWA, M. The matrix rearrangement procedure for graph-theoretical algorithms and its application to the generation of fundamental cycles. *ACM Trans. Math. Softw. 3*, 3 (Sept. 1977), 227-231.
13. JARDINE, N., AND VAN RIJSBERGEN, C.J. The use of hierarchic clustering in information retrieval. *Inf. Stor. Retr. 7* (Dec. 1971), 217-240.
14. KNUTH, D.E. *The Art of Computer Programming, vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973, pp. 471–479.
15. PETERSON, J.L. Computer program for detecting and correcting spelling errors. *Commun. ACM 23*, 12 (Dec. 1980), 676-687.
16. REGHBATI, H.K. An overview of data compression techniques. *Computer 14*, 4 (April 1981), 71-75.
17. RITCHIE, D.M., AND THOMPSON, K. The UNIX time-sharing system. *Commun. ACM 17*, 7 (July 1974), 365-375.
18. ROBINSON, P., AND SINGER, D. Another spelling correction program. *Commun. ACM 24*, 5 (May 1981), 296-297.
19. ROGERS, D.J., AND TANIMOTO, T.T. A computer program for classifying plants. *Science 132* (Oct. 1960), 1115-1118.
20. SALASIN, J. Hierarchical storage in information retrieval. *Commun. ACM 16*, 5 (May 1973), 291-295.
21. SALTON, G. *The SMART Retrieval System.* Prentice-Hall, Englewood Cliffs, N.J., 1971.
22. SALTON, G., AND WONG, A. Generation and search of clustered files. *ACM Trans. Database Syst. 3*, 4 (Dec. 1978), 321-346.
23. SHAPIRO, M. The choice of reference points in best-match file searching. *Commun. ACM 20*, 5 (May 1977), 339-343.
24. TAMURA, S., HIGUCHI, S., AND TANAKA, K. Pattern classification based on fuzzy relations. *IEEE Trans. Syst. Man Cybern. SMC-1*, 1 (Jan. 1971), 61-66.
25. WRIGHT, W.E. Binary search trees in secondary memory. *Acta Inf. 15*, 1 (1981), 3-17.
26. ZAHN, C.T. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Trans. Comput. C-20*, 1 (Jan. 1971), 68-86.
27. ZLOOF, M.M. QBE/OBE: A language for office and business automation. *Computer 14*, 5 (May 1981), 13-22.

Received January 1982; revised May 1982; accepted November 1982