INTRODUCTION

When text is typeset using a computer-based system, it can also be checked for spelling errors automatically and efficiently. Several methods of spelling error detection have been proposed. Morris et al. [MORR75] study statistical properties of English words, and describe an algorithm to catch possible typos by examining the relative frequency of trigrams (3-letter combinations). Kernighan et al. [KERN78] report a more conventional approach which looks up each word in a machine readable spelling dictionary. Knuth [KNUT73] comments on dictionary storage, and suggests an organization intended to reduce space requirements. Another, somewhat unsophisticated technique sorts all words in a document and prints those which occur infrequently. Of course, habitually misspelled words escape unnoticed in a frequency based system.

Both [KERN78] and [KNUT73] mention removing suffixes from the spelling words and storing only the stems. When a word is to be looked up, its suffixes are removed by the same method, and only the stem is checked. This method has the advantage of requiring less space, but the disadvantage of not always catching illegal stem and suffix combinations. For example, the typo "computions" would pass the spelling test because removal of the suffix "ions" leaves the valid stem "comput."

Because there are several hundred thousand words in the English language, a complete spelling dictionary is too large to keep in main memory. Fortunately, a complete spelling dictionary has almost no value for catching typos: misspellings commonly turn out to be obscure or archaic terms which appear in the dictionary. Even a dictionary of 40,000 words may be too large to be useful. For example, the 40,000 word dictionary used by Purdue University’s computing center includes terms such as “de,” “hod,” “ila,” “lo,” “moo,” and “pul,” which would probably be typos in technical prose.

While one cannot give an optimum size for an on-line dictionary, it is clear that “the bigger the better” does not apply. Since writers use most words infrequently and a few words very frequently, the best scheme seems to be:

1. Start with a small core of, say, 10,000 commonly used English words.
2. Add new words to the dictionary only as users request them.

This way, an appropriate dictionary evolves without unnecessary or obscure words. A small dictionary, grown by user request, is less likely to accept misspellings, and it can be managed more efficiently as well.

Carrying the evolution strategy further, researchers at Bell Labs derived a spelling dictionary solely from user’s documents [KERN78]. The resulting dictionary is one-third the size of their original spelling dictionary. It has proved to be more useful, however, since it contains all the heavily used technical jargon as well as common words. Of course, if the user population spans a wide variety of interests, each group should probably grow its own augmentation dictionary, keeping the core for common English words. Any term appearing in all augmentation dictionaries should be deleted and moved to the common core.

Small spelling dictionaries often fit into main memory and can be searched without accessing secondary storage. The words themselves usually occupy a large portion of the available space, however, leaving only a little extra space for the program and data structures. This paper explores searching techniques in the context of a small spelling dictionary, and presents a technique which exploits a small amount of extra space to lower access costs. The second section reviews conventional search methods; the third section presents the trie-binary search (which uses a modest amount of memory). The fourth section examines an application to a particular spelling dictionary, and the fifth section presents performance statistics for typical input data. The final section suggests extensions and improvements.

DICTIONARY SEARCHING TECHNIQUES

We view a spelling dictionary \( D \) as a static set of keys \( k_i \), \( 1 \leq i \leq n \), each key being composed of \( m \) letters. For each word \( w \) in the input text, the search procedure must answer the membership question, “is \( w \) in \( D \)?” We assume that:
1. The dictionary fits into main memory and occupies a fraction of the available space, $0 < \alpha < 1$.
2. The dictionary is not to be compressed (see [MORR75] for compression techniques).
3. The dictionary may be preprocessed (e.g., sorted) to decrease search times. In particular, an index may be built to speed retrieval.

The general problem of data storage and retrieval for static sets of keys has received wide attention in the literature. Knuth [KNUT73] provides a summary. When the keys themselves occupy all but a small fraction of the available memory space, only a few candidate strategies emerge:

- **sequential search**—each probe eliminates 1 key.
- **binary search**—each probe eliminates half the keys.
- **interpolation search**—each probe eliminates more than half the keys in a uniform (or nearly uniform) distribution.
- **hash table search**—each probe eliminates most of the remaining keys (given sufficient memory).
- **partial index**—a small, auxiliary data structure is searched to eliminate all but a subset of the keys. Then, one of the above methods is used to search the subset.

The first four methods have been analyzed, and expressions for the expected number of comparisons per look-up have been derived under the assumption of a uniform distribution of keys. Sequential searching examines one-half of the dictionary for each look-up, and is obviously inferior to binary searching, which makes an average of

$$C_{binary} = \log_2 n - 1$$  \hspace{1cm} (1)

comparisons in a dictionary of $n$ words [KNUT73].

Interpolation search requires only $C_i \log_2(n \alpha) + C_s$ comparisons on the average for nearly uniform distributions [YAO76]. In practice, however, the constants make interpolation search too expensive, especially with a highly skewed distribution like that of a spelling dictionary [KNUT73].

The performance of hashing depends on the fraction of the space that is occupied, $\alpha$, and can be estimated as:

$$C_{hash} = (1 + 1/(1-\alpha))/2$$  \hspace{1cm} (2)

comparisons for a successful search, and

$$C_{hash} = (1 + 1/(1-\alpha)^2)/2$$  \hspace{1cm} (3)

comparisons for an unsuccessful search [KNUT73].

Ignoring for the moment partial index methods, one must choose between binary searching and hashing. In the case of an English spelling dictionary which contains less than $2^{16}$ words, binary searching requires less than 15 probes on the average, and 17 probes in the worst case. From (2) we have that hashing is inferior for $\alpha > 0.966$. For example, when only 2 percent of the space is unused, hashing requires over 25 probes for an average successful search, and 1250 probes for an average unsuccessful search.

**TAKING ADVANTAGE OF A LITTLE EXTRA SPACE**

As shown in Figure 1 when only a small fraction of memory is free, binary searching requires fewer comparisons on the average than hashing, even though a binary search uses none of the extra space that is available. This section presents a partial index method that uses only a little extra space; the next section demonstrates that it makes fewer comparisons on the average than a binary search.

Figure 2 illustrates the use of a partial index mechanism. During each look-up, the spelling program searches the index to identify the appropriate subset of the dictionary to search. Presumably, the index identifies the proper subset rapidly. The program then uses a binary search to explore the specified subset.

One particular index method, called a **trie-binary search** keeps the dictionary in sorted form, using a trie index [FRED60] to identify the correct subset to search. As shown in Figure 3, the simplest trie index consists of an array of 26 pointers, one for each letter of the alphabet, which give the starting location of words starting with that letter. Using the first letter of a word to index into the array, one can quickly locate the subset of all words starting with that letter. The array is referred to as a **node** of the trie; many standard English dictionaries provide a 1-node trie in the form of a thumb index to help the user find the appropriate section faster.

A trie index can be extended to more than one level easily by allowing a pointer in the first level to point to another trie node instead of directly into the dictionary. For example, if the level one pointer for "c" pointed to a second node, then the second node would contain the starting locations of the subsets of words beginning with "ca," "cb," "cc," etc.
English Dictionary Searching With Little Extra Space

Figure 2—The idea of an index. A short search in the index data structure identifies some subset of the keys which is then searched for the exact key.

... , "cz." If a third level node were inserted following the pointer in a second level node for "co," then the third level node would give the starting locations of words beginning with "coa," "cob," ..., "coz."

The following terminology will be useful in the sequel. Pointers in the trie which point into the dictionary are called leaf pointers; all others are called trie pointers. The level one node of a trie is called its root, and a node which can be reached by following a pointer from a level i node is said to lie at level i+1.

Searching in the trie begins by indexing into the root node based on the first character. Decisions at subsequent nodes depend on subsequent letters. In general, an ith level node in the trie further restricts the subset of words to be searched by testing the ith letter.

Since storage space is limited, care must be taken to build the trie so that each node allocated reduces the average number of comparisons by the maximum amount possible. For example, if space exists for only two nodes, dividing the set of words starting with letter "s" results in fewer comparisons than dividing the set of words starting with "a." The idea of allocating nodes to produce lowest cost searches leads to the following algorithm for trie construction:

Algorithm A: (construct minimum cost trie using available storage)

1. construct a 1-node trie for the dictionary;
2. while storage is not exhausted do
   a. find dictionary pointer p which yields best improvement in search time when its set is divided;
   b. insert a new node after pointer p
3. end A;

Algorithm A exhibits a high running time unless one employs an efficient method for finding the optimum dictionary pointer to be replaced. For uniformly distributed sets of keys, dividing a largest set is optimal; for non-uniform distributions the size of the set is not always related to the effect that division of that set would have on the cost. However, assuming that division of the largest remaining subset produces a good, if not optimum, trie leads to an efficient, practical algorithm for index construction:

Algorithm B: (construct a low cost trie efficiently)

1. construct a 1-node trie for the dictionary;
2. select a maximum set size, t;
3. while a dictionary pointer, p, points to a subset of more than t keys do
   a. insert a new node after pointer p
4. end B;

Note that the tries constructed by either algorithm do not have fixed depth. Of course, each pointer in a variable depth trie must have an indication of whether it points into the dictionary or to another level of the trie, so an additional bit is required to store the indicator. Furthermore, pointers into the dictionary should include both a starting and ending
location, since finding adjacent pointers in the trie would be time consuming. Even with these additions, the variable depth trie uses far less storage to achieve the same search time as a fixed depth trie.

The choice of the maximum set size, \( t \), is crucial to fitting the trie into a small amount of memory. If \( t \) is too large, not enough nodes will be generated, and search efficiency will be lost. If \( t \) is too small, the algorithm will run out of space before the trie can be completed, and search performance may be poor. The next section discusses the relationship between storage used and the maximum set size, giving data from a sample dictionary.

**PERFORMANCE OF TRIE-BINARY SEARCH**

This section describes the performance of the trie-binary search strategy on a typical spelling dictionary. Usually, such empirical data has limited value because key distributions vary from file to file. In English, the relative distribution of keys remains fairly constant over a large range of dictionary sizes. Thus, the performance study presented here applies directly to other English dictionaries.

The sample dictionary consists of 16,949 words, and was originally formed from computer tapes of newspapers. Gradually, some technical terms have been added, and obscure and nonsense words have been eliminated (although words like "MR", "MRS", "AVE", "ST", and "BLDG" remain).

With space for 26 pointers (1 node), trie-binary, binary, and hash search were implemented, and the actual number of probes necessary to find each word in the dictionary was recorded. Using the collected data, the average and worst case number of probes for a successful search were calculated. Using equations (1), (2) and (3), and assuming that the trie divided the set of words into 26 equal size subsets, predicted average and worst case probes were computed. Table I summarizes the results.

The trie-binary search (using 1 node) requires only 75 percent of the probes needed by a straight binary search, and clearly performs better with only 26 extra storage locations. Two questions arise immediately: how does trie-binary search perform with more space, and what is the relationship between the maximum set size, \( t \), and the number of nodes allocated?

The graph shown in Figure 4 answers the first question in the case of the sample dictionary. It is clear from this plot that the best tradeoff space for time occurs with a small amount of space. For example, Table II summarizes the number of nodes necessary to lower the average number of probes in steps of one. Each successive reduction requires more space.

For the dictionary in question, a choice of 100-120 nodes represents a good compromise and brings the average number of probes for a successful search down to about 6.25 (or 48 percent of that used by a binary search). Doubling the space to 200-240 nodes further reduces the average number of probes by only .5, and hardly seems worthwhile.

The relationship between the maximum set size, \( t \), and the number of nodes allocated, \( p \), is important because the fast trie building algorithm depends on \( t \), while the user can most easily estimate \( p \). Knuth [KNUT73] estimates that

\[
p = n/(t \ln b)
\]

for a file of uniformly distributed keys, where \( b \) is the branch factor of each node. Using a branching of 26 for English yields

\[
p = n/(3.3t)
\]

For the sample spelling dictionary, the estimate turns out to be close to 50 percent low for some \( t \). This discrepancy can be explained easily by the non-uniform distribution of keys. It turns out that an average branching factor of around 8 provides a closer estimate of \( p \):

\[
p = n/(2.08t)
\]

Figure 5 shows both the estimated and actual number of nodes needed as a function of the maximum set size, \( t \). Unfortunately, the number of nodes rises rapidly for small \( t \), the steep slope making selection of optimum \( t \) difficult. However, the cost of trie construction is small, and the task will be performed once in a preprocessing phase, so version A of the trie building algorithm can be used to find optimum \( t \), if necessary. As noted earlier, adding a few extra nodes does not lower the average number of probes drastically. Thus, a conservative estimate of \( t \) will not degrade performance severely when version B of the algorithm is used.

<table>
<thead>
<tr>
<th>TABLE I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary Search</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Observed</td>
</tr>
<tr>
<td>avg.</td>
</tr>
<tr>
<td>worst</td>
</tr>
<tr>
<td>Predicted</td>
</tr>
<tr>
<td>avg.</td>
</tr>
<tr>
<td>worst</td>
</tr>
</tbody>
</table>

The average and worst case search times for 16949 word dictionary, and the predicted values assuming equal size subsets are produced by the trie. The observed values for hashing are omitted because of the excessive computation costs.

---

8 Throughout this paper we count each access in the index as one probe. Thus, if two characters are tested in the trie and three comparisons are made in the dictionary, the lookup requires five probes.

<table>
<thead>
<tr>
<th>TABLE II</th>
</tr>
</thead>
<tbody>
<tr>
<td>total nodes</td>
</tr>
<tr>
<td>allocated</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>77</td>
</tr>
<tr>
<td>236</td>
</tr>
</tbody>
</table>

The number of nodes necessary to lower average probes in steps of 1. Each successive reduction takes increasingly more space.
IMPLEMENTATION PERFORMANCE

To see how trie-binary search performs in practice, algorithm B was implemented and compared to binary search. The implementation was done in Pascal, and the programs shared all code not involving the searching. Each program was run and timed ten times on a CDC 6500, using as input a text file of 7841 words arranged on 1359 lines. The maximum set size for the trie-binary search was $t=125$, which resulted in a trie with 77 nodes (using approximately 3 percent extra storage). The performance from runs with median search times is given in Table III.

Note that trie-binary search requires only about 46 percent as much CPU time during searching as a binary search on the sample input. Furthermore, binary search requires 20 percent more CPU time than trie-binary search, even if the trie is built "on the fly." When the trie is built and stored, the time required to read it is insignificant compared to the time required to read the dictionary.

OTHER INDEX STRUCTURES

This section considers several other index structures, and compares them to trie-binary search. The first variation, called a length-binary search, works as follows: words in the dictionary are sorted by length, and within each length lexicographically. An index is created which consists of a vector of pointers giving the starting location of each length group. Searching proceeds by indexing the length vector to find the locations of words of the appropriate length. Having located the appropriate subset, the spelling program searches it using a binary search. Length-binary search has the advantage that equal length words are stored contiguously so that memory can be compacted. For many computer architectures, a combination of length- and trie-binary search leads to the best use of stor-
Trie-binary search is superior to hashing with less than this amount of extra space.

Figure 5—The relationship between nodes allocated and maximum set size, \(t\), for a spelling dictionary of 16,949 words. The estimate assumes uniform distribution of keys.

At the root node of the trie, the length is used as an index; successive nodes rely on character indexing. Because words are stored compactly, a significant amount of memory is available for the trie.

Another variation is related to the work of Coffman and Eve [COFF70] who consider storage of keys in a tree. To improve the tree balance, they suggest hashing the key first, and using the bits of the hashed value to make decisions in the tree. Figure 6 shows how hashing can be applied to trie-binary search. First, one divides the dictionary into subsets by hashing the words into a fixed number of “buckets.” Each subset is stored contiguously, in lexicographic order, and its starting position is stored contiguously, in a 1-level trie node. During a search operation, the program hashes the input word to obtain an index into the trie node, from which it obtains a pointer to the appropriate subset. In fact, one can view trie-binary searching as a hashed trie search in which the hash code is given by the character ordinal.

One final variation frequently suggested involves a tree-structured index which guarantees the dictionary will be divided into equal size subsets. Figure 7 shows how the method, which is based on the Indexed Sequential Access Method [GHOS69] organizes the tree. By selecting \(K\) keys at equally-spaced intervals in the dictionary and placing them in the root, the tree divides the file into equal size subsets. Of course, a program to search the multi-level tree makes approximately \(\log_2 K\) comparisons at the root before finding the correct pointer to follow (assuming a binary search). If the search continues through \(m\) levels, the method makes an average of:

\[
C_t \approx \log_2 n - 1 - m
\]

comparisons and follows \(m\) pointers, where \(n\) is the number of keys in the dictionary. We assume here that the number of keys promoted into the index is insignificant compared to the total number of keys.

For trie-binary search we assumed that the cost of following one pointer was the same as the cost of performing one comparison. Applying the same assumption here implies that an Indexed Sequential tree costs the same to search as a straight binary search when the entire set of keys resides in main memory. Therefore, trie-binary search is far superior.
CONCLUSIONS

This paper has presented a data structure and algorithm, the trie-binary search for dictionary searching. The method performs well compared to a binary search while using very little extra space beyond that required to store the keys themselves. Trie-binary search has been applied to a typical spelling dictionary, and statistics have been gathered on its performance. For the distribution of words in a typical spelling dictionary, trie-binary search makes 52 percent fewer probes than binary search, while using less than 3 percent extra space. For an input document of roughly 30 pages, trie-binary search required 74 percent as much total CPU time (46 percent as much CPU time during searching) as a binary search.

Several questions remain unanswered. It would be interesting to find out how trie-binary search performs on sets of keys other than spelling dictionaries. One might conjecture...
that typical data are no more skewed with respect to a uniform distribution than a spelling dictionary. If this is the case, trie-binary searching should perform well. There is also no reason to test the letters in left-to-right order. Testing in other orders might produce a better distribution for English. Finally, if a hashed method is considered, choice of an appropriate hash function needs to be considered.

REFERENCES


