Algorithms for Trie Compaction

M. Al-Suwaiyel and E. Horowitz
University of Southern California

The trie data structure has many properties which make it especially attractive for representing large files of data. These properties include fast retrieval time, quick unsuccessful search determination, and finding the longest match to a given identifier. The main drawback is the space requirement. In this paper the concept of trie compaction is formalized. An exact algorithm for optimal trie compaction and three algorithms for approximate trie compaction are given, and an analysis of the three algorithms is done. The analyses indicate that for actual tries, reductions of around 70 percent in the space required by the uncompacted trie can be expected. The quality of the compaction is shown to be insensitive to the number of nodes, while a more relevant parameter is the alphabet size of the key.

Categories and Subject Descriptors: H.3.2 [Information Storage and Retrieval]: Information Storage; I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods and Search; I.1.2 [Algebraic Manipulation]: Algorithms

General Terms: None

Additional Key Words and Phrases: Trie, storage retrieval, data structure, node compaction, approximation algorithm.

1. AN INTRODUCTION TO THE TRIE DATA STRUCTURE

We use the term record to refer to a collection of related pieces of information about some object or entity. Each generic piece of information in a record is called a field. A key is the field that is used to distinguish among the records. A file is a collection of records. This paper has to do with the storage and retrieval of records from a digital computer. The particular storage scheme to be investigated is called a trie. Section 1 of this paper introduces the trie concept and gives a brief review of its advantages and disadvantages. Section 2 concentrates on the issue of the space required by tries, and presents an algorithm which determines a maximal compaction. Section 3 presents and discusses three approximation algorithms for trie compaction. Section 4 gives computer simulation results.
Definition 1.1. A tree is a finite set $T$ of one or more records such that
1. there is a special record called the root;
2. the remaining records are partitioned into $s \geq 0$ disjoint sets $T(1), \ldots, T(s)$, called subtrees of the root, and each one in turn is a tree.

The children of a record in a tree are the records in the roots of the subtrees of the record. The root of a tree is defined to be at level 0, and the children of a record at level $i$ are at level $i + 1$. A leaf (or an external record) is a record with no children.

The trie retrieval method for files of information was proposed by de la Briandais in 1959 [8], and implemented by Fredkin in 1960 [9]. It works by viewing the value in the key field as a sequence of characters. Fredkin used the name trie for this structure, apparently derived from information retrieval.

Definition 1.2. A trie is an $M$-ary tree, whose nodes are $M$-place vectors with fields corresponding to the digits or characters which comprise the keys. Each node on level $i$ represents the set of all keys that start with the same sequence of $i$ characters; this node specifies an $M$-way branch, depending on the $(i + 1)$st character of a key.

Example 1.1. Consider a symbol table constructed for the reserved words of SPARKS, an Algol-like language developed by Horowitz and Sahni [10]. The reserved words are BY, CASE, CYCLE, DO, ELSE, END, ENDCASE, ENDIF, EXIT, FOR, IF, LOOP, PROCEDURE, REPEAT, THEN, TO, UNTIL, WHILE. Figure 1 shows a trie structure for the reserved words of SPARKS.

In this example (Fig. 1) there are six internal nodes, numbered (1), \ldots, (6), each having 27 locations representing A, B, \ldots, Z, and the blank character. The
numbers 1, 2, \ldots, 18 are pointers to an auxiliary table. This table contains the actual identifiers and can be packed together. The arrows are pointers to internal nodes of the trie (e.g., in the C-location of node (1) there is a pointer to node (2), which has pointers to items 2 and 3 in the auxiliary table).

To search for an identifier \( X = A_1, \ldots, A_k \) in a trie one proceeds as follows: in the root, the algorithm examines field \( A_1 \); a null value implies \( X \) is not present; a pointer to the auxiliary table allows one to check \( X \) against one identifier, the result being true or false; or, if a pointer to another node is present, that implies that there is more than one identifier in the trie which begins with \( A_1 \). If the last alternative holds, then one goes to this new node and examines its \( A_2 \) field, repeating the previous steps. Using the internal representation of the characters, the algorithm can access any field of a node in constant time, and this makes trie searching extremely fast. This search procedure will always terminate after examining at most the \( k \) characters of the identifier.

Suppose a file of \( n \) records is represented by a trie. If the largest value in the key field of a record contains \( k \) characters, then no more than \( k \) character comparisons are required to determine if an identifier is present. As one can easily see from Example 1.1, determining that an identifier is not present requires no more than \( k \), and possibly only one character, comparisons. This is in contrast to any of the binary tree representation schemes which require at least \( \log_2 n \) probes to determine if a search is unsuccessful. Moreover, each probe may require several character comparisons. For a trie, the operations of deletion or insertion of an identifier, \( X \), proceed by doing a search to locate the value of \( X \) and then doing the appropriate operation. If the operation was insertion, then we may have to create new nodes if the value of \( X \) is a prefix of, or has as its prefix, an identifier that already exists in the trie. However, the number of internal nodes to be created will not exceed the number of characters in \( X \); hence, for tries, the time to delete or insert an item is proportional to the search time. For details see [10].

As noted in [3, 12] and [4], the high space requirement is the main disadvantage of tries, so in this paper we present several algorithms for reducing the space while trying to preserve the efficiency of the trie search. As first observed by Rotwitt and de Maine [13], one way to reduce the number of nodes of a trie, and thus to reduce the total space required, is to reorder the testing of attributes. As a trivial example of this approach, consider the trie representation for the words WITH, WITS, WIFE. If testing of attributes is done from left to right, the tree will have four internal nodes, and if one right to left, it will have only one internal node.

The problem of finding an ordering of attributes that guarantees a minimum size trie was shown to be NP-complete by Comer and Sethi, who studied the problem of reordering the testing of attributes in detail; see [3, 4] and [6]. They have also studied heuristic algorithms for selecting attributes which minimize the trie size in [5] and [7].

We investigate an alternate approach, namely that of reducing the space by compacting the nodes. This approach was first mentioned in the literature by Knuth [12]. Note that this approach can even be used on a trie constructed by selecting attributes in a special way so as to reduce its size. Thus the approach here and that of Comer-Sethi are complementary.
2. AN EXACT ALGORITHM FOR TRIE COMPACTION

Consider a trie of \( N \) nodes, each node with \( M \) locations. If we represent the presence of a non-null pointer by a 1, and a null pointer by a 0, then each node can be viewed abstractly as a string of ones and zeros. To discuss compaction algorithms, it will be sufficient to regard a trie in this way (i.e., as \( N \) strings of ones and zeros). Let \( S(1), \ldots, S(N) \) be those bit strings.

**Definition 2.1.** Let \( S \) and \( T \) be two strings of zeros and ones. If their lengths differ, then adjoin zeros to the shorter string so that their lengths are equal. If, in the \( i \)th position, \( S \) and \( T \) both contain a one, then these ones are said to overlap. Now let \( S(1), \ldots, S(N) \) be \( N \) strings of zeros and ones, all of the same length. By node compaction we mean the string which results after positioning the strings \( S(1), \ldots, S(N) \), so that there is no overlapping of ones, and taking the logical OR of these strings, thus producing a single string.

We now formally define the problem of optimal node compaction. Let \( S = \{S(1), \ldots, S(N)\} \), where \( S(1), \ldots, S(N) \) are 0–1 strings constructed from the nodes of a trie. Consider every string as a binary number in the range \((0, 2^M - 1)\). Then the optimal node compaction problem asks one to find a set of \( N \) integers \( J = \{j(i), 0 < i \leq N\} \) such that

1. \( S(i) \land 2^{j(i)} = 0 \), for all \( 0 < i \leq N \) and \( 0 < k \leq N, i \neq k \),
2. and that \( K = \max \{|J|\} \) is minimized.

Observe that condition 1 ensures that there is no overlapping of the ones, and condition 2 ensures that a minimum size compacted string is obtained. A smallest compacted trie is gotten by ORing together the strings \( S(i) \land 2^{j(i)} \) for \( 0 < i \leq N \).

**Example 2.1.** Consider again the trie constructed for the SPARKS reserved words given in Figure 1. If we represent the null pointers by zeros, the non-null pointers to internal nodes by parenthesized numbers, and the pointers to the auxiliary table by nonparenthesized numbers, then we can represent the trie of Figure 1 by the nodes in Figure 2.

One way to compact this trie is to place nodes (2), (3), (4), (5), and (6) into one node, since this can be done without overlapping any of the non-null pointers, and then place node (1) on the resulting compacted node starting at location (field) 15. The compacted trie will look like the following:

\[
\begin{array}{cccccccccccccccccccccc}
6 & 2 & 0 & 7 & (6) & 0 & 0 & 0 & 1 & 5 & 8 & 0 & 0 & 5 & 0 & (5) & 1 & 6 & 1 & (2) & 4 & (3) & 1 & 0 & 0 & 0 & 1 & 1 & 9 & 3 & 1 & 2 & 0 & 0 & 0 & 1 & 3 & 0 & 1 & 4 & 0 & (4) & 1 & 7 & 0 & 1 & 8 & 0 & 0 & 0
\end{array}
\]

The compacted trie required 41 locations compared to 162 for the original trie, a saving of about 75 percent. In this compacted trie, node (1) starts at location 15.

and nodes (2) to (6) start at location 1. A table of all nodes and their starting locations should be kept along with the compacted trie.

To search for an identifier $X = A_1, \ldots, A_k$ in a compacted trie, one proceeds as follows.

1. From the Starting Locations Table find the location of the root, and let $i = 1$.
2. Locate field $A_i$ and examine it; one of the following cases should be true:
   
   (a) a null value is found, which implies $X$ is not present (i.e., an unsuccessful search);
   
   (b) a pointer to the auxiliary table is found: this allows one to check $X$ against one identifier, the result being true or false (i.e., a successful or unsuccessful search, respectively);
   
   (c) a pointer to another node is found, implying that there is more than one identifier in the compacted trie which begins with $A_1, \ldots, A_i$.
3. If alternative (c) above holds, then increase $i$ by one. If $i \leq k$ then, from the Starting Locations Table, find the location of the node pointed to and repeat step 2. If $i > k$, then the search ends unsuccessfully.

As can be seen, the only difference between searching a trie and a compacted trie is that in the latter case, when one meets a pointer to an internal node, one has to fetch its location from the Starting Locations Table. This step can be implemented in a constant time. Hence, the worst-case complexity of trie search in both the normal and compacted cases is of the same order. Therefore, compacting a trie does not compromise the search time. This search procedure will always terminate after examining at most $k$ characters of the identifier.

To insert an identifier $Y$ into a compacted trie, one has to search the trie as outlined above. If a search is successful, then there is no need to insert $Y$. If the search is unsuccessful, then it must have failed at step 2(a), 2(b), or 3, above.

- In case of failure at step 2(a), replace the null value by a pointer to a new entry in the auxiliary table and insert $Y$ there.
- Failure at step 2(b) implies that there exists a stored identifier, $X$, that is a prefix of $Y$. In this case, a new internal node has to be created that has two pointers to the auxiliary table to distinguish between $X$ and $Y$. This new node can be compacted along with the compacted trie.
- Failure at step 3 implies that $Y$ is a prefix of a stored identifier $X$. In this case, the field representing the blank in the last node referred to should have a pointer to a new entry in the auxiliary table where $Y$ can be inserted. However, this blank field may be occupied with a non-null pointer of another node. Therefore this insertion may not be possible. An alternative would be to store $Y$ in an overflow list. Thus insertion into an already compacted trie is not always possible.

To delete an identifier $Y$ from a compacted trie, one has to locate $Y$ and then replace it by a null value. The pointers that led to locating $Y$ cannot be removed because in a compacted trie it is not possible to know which node they belong to. In general, it can be seen that compacted tries do not adapt well to insertion and deletion, and this limits the usefulness of compacted tries to static files.
Before describing an optimal trie compaction algorithm, let us adopt the following terminology: by “placing S(i) on S(j) starting at location k,” we mean shifting S(i) k locations to the right of S(j) and then adding or ORing S(i) to S(j). Whenever we add, there will be no overlapping of ones. String S(i) is said “to precede” string S(j) if the starting point of S(i) precedes S(j). If S(i) and S(j) have the same starting point, then S(i) precedes S(j) iff i < j, otherwise S(j) precedes S(i). Thus, after compaction, we can speak about the strings S(1), . . . , S(N) being ordered. An “optimal permutation” of S(1), . . . , S(N) is a permutation which yields a maximal compaction.

The strategy we follow to produce an optimal compaction for a given trie is first to form the strings S(1), . . . , S(N). Then, for each possible ordering of these N strings, we form the compacted string L in the following way: let there be a permutation of 1, 2, . . . , N, say, P(1), . . . , P(N); then, place S(P(1)) at the start. This implies j(1) = 0. The next step is to try to place S(i) at location k of S(i - 1), such that there is no overlapping of the ones, for all k, 0 < k ≤ M and 1 < i ≤ N. This makes j(i) = j(i - 1) + k, 1 < i ≤ N, and the final length of the compacted string L will be: Length(L) = j(N) + M. The fact that it is necessary to try all possible locations of S(i - 1) is shown in Lemma 2.1.

**Lemma 2.1.** Let P(1), . . . , P(N) be a permutation of 1, 2, . . . , N such that S(P(1)), . . . , S(P(N)) is an ordering which has an optimal compaction. Then placing S(P(i + 1)) at the first possible location of S(P(i)) may not yield the smallest compacted trie.

**Proof.** We show that the lemma is true for N = 3 and M = 4, and later generalize for all N > 3 and M > 4, and treat as special the cases for M = 2 and M = 3.

Consider the following nodes given as strings S(1), S(2), and S(3):

S(1) = 1001, S(2) = 1001, and S(3) = 0101.

There are six permutations to consider.

**Case 1.** Compact the nodes in the order S(1), S(2), S(3).

1(a) S(2) on S(1) at location 2 gives: L = 11011101, Length(L) = 8.
1(b) S(2) on S(1) at location 3 gives: L = 1011111, Length(L) = 7.

**Case 2.** Compact in the order S(1), S(3), S(2).

2(a) S(3) on S(1) at location 2 gives: L = 101111001, Length(L) = 9.
2(b) S(3) on S(1) at location 4 gives: L = 100111101, Length(L) = 9.

**Case 3.** Compact in the order S(2), S(1), S(3). (This is the same as the order S(1), S(2), S(3).)

**Case 4.** Compact in the order S(2), S(3), S(1). (This is the same as the order S(1), S(3), S(2).)

**Case 5.** Compact in the order S(3), S(1), S(2).

5(a) S(1) on S(3) at location 3 gives: L = 011111101, Length(L) = 8.
5(b) S(1) on S(3) at location 5 gives: L = 010111101, Length(L) = 9.

**Case 6.** Compact in the order S(3), S(2), S(1). (This is the same as the order S(3), S(1), S(2).)
Now, after examining all permutations of the nodes, we find that, using the first possible location strategy, there are three orderings of the nodes that give the smallest compacted trie (i.e., of length 8). However, for the orderings \( S(1), S(2), S(3) \) and \( S(2), S(1), S(3) \), placing the second node not on the first possible location it fits into but on the next, gives the smallest trie (i.e., of length 7). Hence, placing \( S(P(i + 1)) \) at the first possible location of \( S(P(i)) \) does not give the smallest compacted trie for all the permutations of the nodes including optimal permutations.

This result can be generalized for all \( M > 4 \) by considering the following nodes:

\[
S(1) = 00 \ldots 0001001, \quad S(2) = 00 \ldots 0001001, \quad \text{and} \quad S(3) = 00 \ldots 0000101.
\]

It is clear that the added zeros (the leftmost \( M - 4 \) zeros) will not change the optimal order of compacting the nodes, so the result still holds.

To generalize the result for all \( N > 3 \), we can make \( S(4) = S(5) = \ldots = S(N) = 111, \ldots, 1 \) (i.e., each node has \( M \) ones). Again, it is clear that the added strings will not change the optimal order of compacting the nodes, so the result still holds. For the special case \( M = 2 \), consider the following nodes given as strings \( S(1) = 10, \quad S(2) = 01, \quad \text{and} \quad S(3) = 10 \). For \( M = 3 \), consider \( S(1) = 010, \quad S(2) = 001, \) and \( S(3) = 010 \). For both of these cases the results easily follow. This completes the proof.

It should be mentioned here that this lemma does not establish the problem of optimal compaction to be NP-complete, but it does establish that an obvious "greedy approach" for a given permutation will not always lead to an optimal compaction.

Based on the lemma and the previous discussion, our algorithm to solve the optimal node compaction problem exactly is now given.

```plaintext
procedure EXACT(N, M, NODE(1:N), ANS(1:N*M), FV(1:N))
// This procedure gives an exact solution to the node compaction problem //
// N = the number of nodes, M = the number of locations in a node //
// Node(1:N) has N bit strings of length M representing the N nodes of the trie to be compacted //
// P(1:N) contains a permutation of the integers 1, \ldots, N, representing //
// an ordering of the nodes //
// V(1:N) is the current solution, where V(i) is an integer between //
// 1 and M \cdot N representing the starting point of node P(i) //
// FV(1:N) is the current optimal solution, where FV(i) is an integer between 1 and M \cdot N //
// representing the starting point of node P(i) in the current optimal solution //
// ANS(1:N*M) contains the compacted trie //
for all permutations of 1, \ldots, N do
    ANS := Node(P(1))
    Let P(1:N) contain the next permutation;
    for i := 1 to N do
        V(Node(P(i))) := (i - 1) \cdot M + 1
        FV(Node(P(i))) := (i - 1) \cdot M + 1
    repeat
    // This was initialization of P, V, and FV //
    call COMPACT(2)
    if V(Node(P(N))) < FV(Node(P(N)))
        then FV := V // new solution //
endif
```
repeat
Place the nodes in ANS (1:M• N) with node P(i) starting at FV(i)).
end EXACT

procedure COMPACT(i)
//This procedure tries all different placements of Node(P(i)) on Node(P(i - 1)), 0 < i ≤ N
//global N, M, ANS, P
if i ≤ N
then k := V(Node(P(i - 1))) //k is the starting point of Node(P(i - 1))/
loop
k := PLACE(i, k)
if k = 0 then exit endif //no more positions/
V(Node(P(i))) := k
Update ANS by placing Node(P(i)) on ANS starting at location k
call COMPACT(P(1' + 1))
if K < V(Node(P(i))) + M then k := k + 1 else exit
endif
end COMPACT

procedure PLACE(i, k)
//This procedure returns the smallest integer j which is greater/
//than or equal to k, such that Node(P(i)) can be placed on Node(P(i - 1))/
//starting at j. j = 0 if Node(P(i)) cannot fit anywhere on/
//Node(P(i - 1)) to the right of location k/
//global IV, M, P
for j := k to V(Node(P(i - 1)) + M - 1 do
if (Node(P(i)) can be placed starting at location j over all nodes
occupying ANS(j), . . . , ANS(V(Node(P(i - 1)) + M - 1)
then return(j)
//The above test can be accomplished by transforming the string/
//Node(P(i)) to a string of ones and xs, where an x replaces a zero, and/
//then matching Node(P(i)) with ANS(j), . . . , ANS(V(Node(P(i - 1))) + M - 1).
//The symbol x will match any symbol in the latter string.//
//To match these two strings, we will use the Knuth–Morris–Pratt//
//algorithm for pattern matching, see [10], whose complexity is/ /
//O(the sum of the length of the two strings to be matched)/
repeat
return(0)
end PLACE

As can be seen above, the exact algorithm works by examining all permutations of
the nodes and, for each permutation, it examines all possible placements of a node
on its predecessor. This approach is the best one known by the authors to exactly
solve this problem.

2.1 Time Analysis of the Exact Algorithm

For a given i, the number of invocations of PLACE depends upon the number of
ways to place Node(P(i)) onto the compacted trie. Since PLACE starts where it
left off, the total time for all invocations of PLACE is proportional to M when we
use the Knuth–Morris–Pratt algorithm. Therefore the time for PLACE, denoted
by Tp, is bounded above by 2M.

For procedure COMPACT, if we have only two nodes, then the time it takes,
denoted by Tc, is

\[ Tc(2) = M + 1 \]

in the worst case.
The loop statement may be repeated $M$ times each call, and in each call $N$ is reduced by 1. Therefore,

$$T_{c}(N) = cM + (M + 1)T_{c}(N - 1).$$

The solution of this recurrence relation is

$$T_{c}(N) \leq d(M + 1)^{N-1} + cM \sum_{0 \leq k \leq N-3} (M + 1)^k
\leq d(M + 1)^{N-1} + c(M + 1)^{N-2} = O(M^{N-1}).$$

Finally, in procedure EXACT, the outer for loop is repeated exactly $N!$ times, and the inner for loop adds a cost of $N$, hence the time for EXACT, denoted by $T_e$, is

$$T_e(N) = N!(dN + cM(M^{N-1})/(M - 1)),$$

where $c$ and $d$ are constants, or

$$T_e(N) = O(N N! + N! M^{N-1}).$$

It is possible to analyze the expected complexity of COMPACT. This is done in [1]. The computing time is still exponential in the variable $N$ and, hence, this exact algorithm remains computationally infeasible.

3. APPROXIMATE ALGORITHMS FOR NODE COMPAC TION

With the prohibitive cost of the exact algorithm, as given before, there is a warranted need for a polynomial time algorithm that generates solutions which are at least "close" to optimal. We call such an algorithm an approximation algorithm.

In this section we present three approximation algorithms for solving the node compaction problem. These algorithms differ in their time complexity, one is of order $O(MN^3)$, the second of order $O(MN \log N)$, and the third of order $O(MN)$. We give an analysis of the quality of the solution for each of these algorithms in the worst case and give empirical results obtained by running them on a set of random and realistic tries in Section 4. But, before presenting the algorithms, let us formally define the notion of approximate algorithms.

Let $P$ be a problem such as the optimal trie compaction problem, and let $Z$ be an instance of $P$. Let $A$ be an algorithm that generates a feasible solution to every instance of the problem $P$. Let $F(Z)$ be the value of an optimal solution of $Z$, and $G(I)$ be the value of the feasible solution generated by $A$.

Definition 3.1. $A$ is an absolute approximation algorithm for problem $P$ if and only if for every instance $I$ of $P$, $|F(I) - G(I)| \leq c$, for some constant $c$.

Definition 3.2. $A$ is an $f(n)$-approximate algorithm if and only if for every instance $I$ of size $n$, $|F(I) - G(I)| / |F(I)| < f(n)$, where $F(I)$ is not equal to 0.

Definition 3.3. An $e$-approximation algorithm is an $f(n)$-approximate algorithm for which $f(n) \leq e$, for some constant $e$.

These definitions have now been in use for several years, and can be found for example in [11]. It is clear that an absolute approximation algorithm is the most
does such an algorithm exist for node compaction? The answer is very likely, no, as given by the following lemma.

**Lemma 3.1.** Absolute approximation trie compaction is as hard as optimal trie compaction.

**Proof.** Assume there is a polynomial time absolute approximation algorithm $A$ for trie compaction that guarantees solutions such that $|F(I) \cdot G(I)| \leq k$, where $F(I)$ equals the optimal solution for an instance $I$ of the node compaction problem; $G(I)$ equals a solution given by algorithm $A$ for the instance $I$; and $k$ is a constant. An instance $I$ of the node compaction problem can be stated thus: Given $S(1), \ldots, S(N)$, where $S(i)$ is a $0-1$ string of length $M$, $1 \leq i \leq N$, find a permutation of $1, 2, \ldots, N$ that leads to the smallest compacted trie.

Now, for any instance $I$, we derive another instance $I'$ as follows: produce $T = \{t(1), \ldots, t(N)\}$, where $t(i)$ is a $0-1$ string of length $(k + 1)M$, obtained from $S(i)$ by duplicating every location of $S(i)k + 1$ times, $1 \leq i \leq N$ (e.g., if $S(i) = 011010$ and $k = 1$, then $t(i) = 001110011100$). In instance $I'$ we are asked to find a permutation of the $t(i)$'s that leads to the smallest compacted trie, as before. It is clear that $I$ and $I'$ have the same feasible and optimal solutions because of the size and the relative positions of the ones and zeros.

Now, any solution to $I'$ given by $A$ must satisfy $|F(I') \cdot G(I')| = 0$, and hence: $|F(I') \cdot G(I')| = 0$. This is true because if $|F(I) \cdot G(I)| = c, c \leq k$, then $|F(I') \cdot G(I')| = (k + 1)c$, but $(k + 1)c > k$ for all $c > 0$, and to satisfy $|F(I') \cdot G(I')| \leq k$, we must have $c = 0$, so the equation above follows. Therefore, algorithm $A$ gives a solution that is optimal for both $I$ and $I'$, and this completes the proof. $\Box$

It should be noted here that Lemma 3.1 does not establish optimal trie compaction as an NP-complete problem. We now present three approximate algorithms for trie compaction. These are shown below as algorithms COMPRESS, PAIRS, and LINEAR, and are characterized by the following theorem.

**Theorem 3.1** The worst-case complexities of algorithms COMPRESS, PAIRS, and LINEAR are $O(mn^2)$, $O(mn \log n)$, and $O(mm)$ respectively. Moreover, the ratio of the size of the compacted tries produced by these algorithms to the size of the optimally compacted trie is bounded from above by $f(n) = (mn - n - r - 1)/(n + r - 1)$, where $n$ and $m$ are as described before and $r$ is the number of records indexed by the trie. The bound given by $f(n)$ is tight.

**Proof.** For each of the algorithms the proof of the theorem is given with the discussion of the algorithm. $\Box$

### 3.1 The First Approximation Algorithm—COMPRESS

This algorithm works by selecting the nodes of a trie in any order (to be discussed later) and places the next node at the first possible location in the compacted trie.

**procedure** COMPRESS($N, M, S, FV, COMPACT$)

//This procedure places the next node of the uncompacted trie into the first possible location/
//in the compacted trie, called COMPACT, by searching COMPACT using the//
//Knuth–Morris–Pratt algorithm for pattern matching, shown here as
//procedure MATCH/

Algorithms for Trie Compaction

//N is the number of nodes, M is the number of locations in a node//
//S(1), . . . , S(N) are 0–1 strings representing the nodes of the trie.//
//FV(1:N) contains the starting points of the nodes in COMPACT//
//COMPACT(1*N*M) contains the compacted trie.//
//NEXTNODE(1:M) is a transformation of S(J), 1 ≤ J ≤ N,//
//such that NEXTNODE(i) = 0 if bit i of S(j) is 1 and x otherwise,//
//the x here represents a "don't care" entry that matches any symbol, a0 or a1.//
//The purpose of the transformation is to try to match nextnode//
//with a substring of COMPACT starting at COMPACT(K). If//
//there is a match then S(j) can be placed on COMPACT starting at location K//

for i := 1 to N*M do COMPACT(i) = 0; repeat
//initializing COMPACT//
for j := 1 to N do
  for i := 1 to M do
    if BIT i OF S(j) = 0
      then NEXTNODE(i) := X
    else NEXTNODE(i) := 0
  endif
repeat
  call MATCH(COMPACT, NEXTNODE, i)
//MATCH determines if NEXTNODE is a substring of COMPACT and returns.//
//the smallest index in i where the substring begins.//
  FV(j) := i
repeat
end COMPRESS

To compute the complexity of COMPRESS, we note that at stage i of the computation process we may have to look at M*(i - 1) locations of COMPACT before finding a match, and the complexity of procedure MATCH is known to be O(Mi). (See [10], ch. 4.) Then, for N nodes, each node of length M, the complexity of COMPRESS can be formulated as

T(n) = \sum_{1≤i≤N} Mi = O(MN^2).

Quality of the solution of algorithm COMPRESS in comparison to the optimal solution. Let N be the number of nodes, let M be the number of locations per node, and let R be the ratio of the size of the compacted trie given by COMPRESS to the size of the optimally compacted trie. In the worst case there exists an ordering of nodes such that the size of the compacted trie given by COMPRESS can be as large as M*N. This happens when node i cannot fit anywhere before COMPACT(M*(i - 1)), so it is placed starting at COMPACT(M*(i - 1) + 1), 1 ≤ i ≤ N. On the other hand, the size of the optimally compacted trie can be as small as max{P, M}, where P is the number of non-null pointers. Thus we conclude that

R ≤ M*N/P.

Now let us express P in terms of N and the number of records in the file indexed by the trie. Since every node is pointed to by another node, except the root, there are N – 1 node pointers. In addition to this, the external nodes will have pointers to the actual records (i.e., there are r information pointers, where r is the number of records). Therefore, P = r + N – 1 and R ≤ M*N/(r + N – 1).

The following examples show that the limit given by the last inequality is tight, in the sense that algorithm COMPRESS will produce a compacted trie of

size \( M \times N \), while the trie could be compacted to be of size \( N + r - 1 \). In the examples we use \( M = 4 \), for simplicity, and later generalize the result to \( M > 4 \), and treat as special the cases where \( M = 2 \) or \( 3 \). As these examples are somewhat long, on a first reading the reader may now prefer to skip directly to Section 3.2.

**Example 3.1.** In the following we will assume that \( N \geq 2 \) because \( N = 1 \) is a trivial case, since there are no nodes to compact. Let \( u = N/2 \), if \( N \) is even, otherwise \( u = (N - 1)/2 \). Consider a trie having \( u \) nodes of type 1, shown in Eq. (3.1), and number these nodes from 1 to \( u \). Also let nodes \( u + 1, u + 2, \ldots, u + u = N \) be of type 2 as shown in Eq. (3.1). If \( N \) is odd let node \( N \) be of type 3 as shown in Eq. (3.1).

\[
\text{type 1 node: 1011, type 2 node: 1101, type 3 node: 1100, type 4 node: 0011} \quad (3.1)
\]

A trie can be built from nodes 1, 2, \ldots, \( N \), as constructed above, such that node 1 is the root and nodes 2, \ldots, \( N \) correspond to the level ordering. If the nodes are input to COMPRESS by level (and within a level, use left-to-right), then the resulting compacted trie will be as shown in Eq. (3.2) if \( N \) is even or as in Eq. (3.3) if \( N \) is odd.

\[
10111011, \ldots, 10111101, \ldots, 1101 \quad (3.2)
\]
\[
10111011, \ldots, 10111101, \ldots, 11011100 \quad (3.3)
\]

As can be seen, the size of the trie in Eq. (3.2) or in Eq. (3.3) is \( M \times N \).

The smallest trie will be obtained if the nodes are input in the order \( u + 1, 1, u + 2, 2, \ldots, u + i, i, \ldots, u + u, u \), but if \( N \) is odd then node \( N \) should be read before node \( u + 1 \). This way the compacted trie will have no empty locations, and the size of the smallest trie will be exactly equal to \( P \) (i.e., \( N + r - 1 \)). This example shows that COMPRESS may not compact at all, while a maximum amount of compaction was possible.

**Example 3.2.** Let \( N \) and \( u \) be the same as in the last example, and consider a trie having nodes 1, 2, \ldots, \( u \) of type 1, and nodes \( u + 1, u + 2, \ldots, u + u \) of type 2, if \( N \) is even, otherwise let nodes 2, \ldots, \( u + 1 \) be of type 1, nodes \( u + 2, \ldots, u + u \) of type 2, and node 1 be of type 4 shown in Eq. (3.1).

A trie can be built from nodes 1, 2, \ldots, \( N \), as constructed above, such that node 1 is the root and nodes 2, 3, \ldots, \( N \) correspond to the level ordering. By inputing the nodes in this order to COMPRESS, the resulting trie will be as shown in Eq. (3.4) if \( N \) is even or as in Eq. (3.5) if \( N \) is odd.

\[
10111011, \ldots, 10111101, \ldots, 1101 \quad (3.4)
\]
\[
001110111011, \ldots, 10111101, \ldots, 1101 \quad (3.5)
\]

As can be seen, the size of the trie in Eqs. (3.4) or in (3.5) is \( M \times N \). The smallest compacted trie will be obtained if the nodes were input in the order \( u + 1, 1, u + 2, 2, \ldots, u + i, i, \ldots, 2u \), \( u \) if \( N \) is even or in the order \( u + 2, 2, \ldots, u + i, i, \ldots, 2u \), \( u \), \( 2u + 1, u + 1, 1 \) if \( N \) is odd. This way the compacted trie will have no empty locations and the size of the smallest compacted trie then becomes exactly equal to \( P = r + N - 1 \).

Observe that if we adopt a strategy which orders the nodes for COMPRESS according to the number of non-null pointers, then these two examples show that inputing the nodes which have the fewest non-null pointers either first or last
both fail to produce any compaction. The strategy with the fewest non-null
pointers first will be referred to as least first order, and the one with fewest non-
null pointers last will be referred to as least last order.

3.1.1 Extending the Results to $M > 3$. The constructions below show that the
bound given in Theorem 3.1 is tight.

1. If $M \mod 3 = 1$, then let $x =$ smallest integer $\geq M/3$, $y =$ largest integer $\leq
M/3$, and $z = M - x - y$. If $M \mod 3 = 2$, then let $x =$ (smallest integer $\geq
M/3) + 1$, $y =$ (largest integer $\leq M/3) - 1$, and $z = M - x - y$. If $m \mod 3 =
0$, then let $x = (m/3) + 2$, $y = (m/3) - 1$, and $z = m - x - y$.

2. In type 1 nodes given in Eq. 3.1 let the first $x$ locations be nonempty, the
next $y$ locations be empty, and the last $z$ locations be nonempty.

3. In type 2 nodes let the first $z$ locations be nonempty, the next $y$ locations be
empty, and the last $x$ locations be nonempty.

4. In type 3 nodes let the first $i$ locations be nonempty, $x \leq i \leq M$.

5. In type 4 nodes let the last $i$ locations be nonempty, $x \leq i \leq M$.

Now repeat the constructions in Examples 3.1 and 3.2, with the types of nodes
modified as in (1)-(5) above and the results follow directly.

We should note that this is a worst-case performance—the average case may
be better—and if we restrict the number of non-null pointers in a node to be
$\geq i$, for $1 \leq i \leq M$, then we have the following result, describing the performance
of COMPRESS, under this restriction.

**Lemma 3.2.** If every node has at least $i$ non-null pointers, $1 \leq i \leq M$, then $R \leq
M/i$ for all the strategies above.

**Proof.** Let $R$ be as obtained above, that is,

$$R \leq M \cdot N/P.$$ 

If there are $i$ non-null pointers in every node, then we have

$$P \geq i \cdot N,$$

therefore

$$R \leq M \cdot N/P \leq M \cdot N/i \cdot N = M/i.$$ 

The case where $M = 2$. For this case there can only be three types of nodes,
excluding the empty node, and they are

- type 1: 10,
- type 2: 01,
- type 3: 11.

We can disregard nodes of the third type because they will not change the
outcome, and consider only nodes of the first and second type.

Consider any stage of the compaction process by COMPRESS, we can have
one of two situations.

1. There is an empty location in one of the ends in the compacted trie
COMPACT, so the next node can be placed at the appropriate end.
2. There is no empty location, so the next node will be placed at the rightmost
end of COMPACT.

This leads to the conclusion that the size of the largest compacted trie produced
by COMPRESS for the case $M = 2$ is $P + 1$. 

The optimally compacted trie can be obtained by compacting nodes of the first type together, followed by all nodes of the second type. This guarantees that the empty location in the right end of a type 1 node will be used by the next node, and the empty location in the left end of a type 2 node will be superimposed on the previous node, leaving no empty locations and giving the optimal size as \( P \) (or \( P + 1 \) if all nodes are of the same type). Therefore, \( M = 2: \)

\[
R(\text{Least First}) = R(\text{Least Last}) = R(\text{Level Order}) = (P + 1)/P.
\]

The case where \( M = 3 \). To obtain the worst case, we should maximize the number of zero locations in the compacted trie. If we compress all nodes with only one location, then we will have no more than two empty locations at one end. Therefore, the worst case will result when we are compacting nodes with more than one location (i.e., nodes with only two Is, if we disregard the full nodes, that is, 111).

Now, for Least First, we can have only nodes of the types 001, 101, and 110 (and an 001 node to add two more zeros to the left). A worst case will occur if we compact

\[
011, 101, 011, 101, \ldots, 011, 101,
\]

giving the following compacted trie:

\[
L = 0111011110111101, \ldots, 111101.
\]

To compute the size of \( L \), let \( N \) be the number of 011 nodes. This is also the number of 101 nodes. Then \( P \), the number of ones, is

\[
P = 2N + 2N = 4N.
\]

The number of zeros in \( L \) is the same as the number of 101 nodes, plus one zero from the 011 node in the start (i.e., \( N + 1 \)). Thus the length of the compacted trie given by COMPRESS is

\[
\text{Length}(L) = 4N + N + 1 = 5N + 1.
\]

Now, for the optimally compacted trie, let the nodes be compacted in the order 101, 101, \ldots, 101, 011, 011, \ldots, 011, and let the number of the 101 nodes be even. The resulting compacted trie will be of length \( P \). Thus \( R(\text{Least First}) \leq (5N + 1)/4N \).

For the Least Last and the Level Orders, we can have the same worst case by the same construction. Therefore,

\[
R(\text{COMPRESS}) \leq (5N + 1)/4N, \quad \text{for } M = 3.
\]

Now in order to classify algorithm COMPRESS into one of the classes of the approximation algorithms, we do the following:

1. For \( m \geq 3 \), \( G(I) = M \cdot N \) and \( F(I) = N - 1 + r \), so \( |F(I) - G(I)|/F(I) \leq f(N) \), where \( f(N) = (M \cdot N - N - r + 1)/(N + r - 1) \). Observe that \( f(N) \leq M/2 \).

This shows that algorithm COMPRESS is an \( f(n) \)-approximate algorithm for \( M \geq 3 \).
For $M = 3$, $|F(I) - G(I)|/F(I) \leq f(N)$, where $f(N) = (5N + 2)/(4N - 1) \leq 2$. Thus, for $M = 3$, COMPRESS is an $e$-approximate algorithm where $e = 2$.

For $M = 2$, $|F(I) - G(I)| \leq 1$ (i.e., COMPRESS in an absolute approximate algorithm for $M = 2$).

### 3.2 The Second Approximation Algorithm—PAIRS

The second approximation algorithm works by combining pairs of nodes, then compacting the nodes in pairs, and repeating the process until only a single node remains.

```
procedure PAIRS(N, M, S, FV, COMPACT)
    // A procedure to compact the nodes of a trie in pairs, /
    // until a single node is left. In this procedure we will make /
    // use of procedure COMPRESS as presented before. The /
    // input to COMPRESS will now be a pair of nodes /
    // \text{\textit{N} = the number of nodes. \textit{M} = the number of locations in a node/}
    // \text{\textit{S(1:N)} contains bit strings representing the nodes of the trie to be compacted.} /
    // \text{\textit{S(i) can hold up to M* N bits}}/
    // \text{\textit{FV(1:N)} contains the starting points of \textit{S(1:N)}}. /
    // \text{\textit{COMPACT(1: N*M) contains the compacted trie.} /
    // \text{\textit{R(1) and R(2) are the pair of nodes to be compacted by COMPRESS.} /
    // \text{\textit{NPairs = N, NEWSIZE = M}}
    
    \textbf{while} (NPairs > 1) \textbf{do}
        \textbf{if} (NPairs \mod 2) = 1 \textbf{then} begin \text{NPairs} := NPairs + 1; \text{S(NPairs)} := 00...0 \text{end} // \text{S(NPairs)} has NEWSIZE zeros \textbf{end}
    \textbf{end}
    \textbf{end}

    \textbf{repeat}
    NEWSIZE := 2*NEWSIZE
    NPairs := NPairs/2
    COMPACT := S(1)
    end PAIRS
```

To compute the complexity of PAIRS, let $N = 2^k$ for some $k$. We can consider that compacting the nodes in pairs is done at levels, and there are $k + 1$ such levels. At level $i$, the size of the nodes is $M*2^{k-i+1}$, and there are $2^{i-1}$ nodes.

Therefore at every level the cost of calling COMPRESS is $2^{i-1}*2*M*2^{k-i+1}$.

The `while` statement is repeated $k$ times, hence the total cost of PAIRS is

$$
\sum_{1 \leq i \leq k} 2^{i-1}*2*M*2^{k-i+1} = 2MNk = O(MN \log N).
$$

Quality of the solution of algorithm PAIRS in comparison to the optimal solution. For $M \geq 4$ and $N > 1$, let $R$ be the ratio of the size of the largest
compacted trie given by PAIRS to the optimal. As in COMPRESS, we have $R$ bounded from above by $M \cdot N/P$, where $P$ is the number of non-null pointers in the trie. We show that this limit is tight by the same construction as in Example 3.1.

For $M = 3$ and $N > 1$, by the same construction used for algorithm COMPRESS, we have

$$R(\text{PAIRS}) \leq (5N + 1)/4N.$$  

For $M = 2$ and $N > 1$, consider the following sequence of nodes:

$$(01, 01), (10, 10), (10, 10), \ldots, (10, 10).$$

PAIRS will give a compacted trie of length $P + 2$, where $P$ is the number of ones, while the optimal trie is clearly of length $P$. This is the worst case, and so

$$R(\text{PAIRS}) \leq (P + 2)/P, \text{ for } M = 2.$$  

As in the case of algorithm COMPRESS, we note that PAIRS is an absolute approximate algorithm for $M = 2$, an $\epsilon$-approximate algorithm for $M = 3$, and an $f(n)$-approximate algorithm for $M > 3$.

3.3 The Third Approximation Algorithm—LINEAR

The third approximation algorithm works by placing the next node at the first possible location it fits into of the previous node, and repeating the process until the last node has been placed, giving the compacted trie which we store in COMPACT.

procedure LINEAR(N, M, S, FV, COMPACT)

// $N$ is the number of nodes, $M$ is the number of locations per node/
// $S(1:N)$ contains the 0-1 strings representing the nodes of the trie to be compacted/  
// COMPACT (1:M*N) contains the compacted trie/  
// FV(1:N) contains the starting points of $S(1:N)$/  
// STRING(1:M) is the part of COMPACT where the last node was stored/  
// NEXTNODE(1:M) is a transformation of $S(j)$, 1 < $j$ < $N$, as described in COMPRESS/  

for $j := 1$ to $M$ do

COMPACT(j) := bit $j$ of $S(1)$, // initial COMPACT/  
FV(1) := 1  
repeat

for $k := 2$ to $N$ do

for $j := 1$ to $M$ do

then Nextnode($j$) := $x$

else Nextnode($j$) := 0

endif

for $j := FV(k - 1)$ to $(FV(k - 1) + M)$ do

STRING($j$) := COMPACT($j$)

call MATCH(STRING, NEXTNODE, I)

FV($k$) := I

repeat

end LINEAR

To determine the complexity of LINEAR we note that at stage $k$ of the compaction process we may have to look at all of the $M$ locations of the last node.
placed in COMPACT; the complexity of MATCH is $O(\text{length}(\text{STRING}) + \text{length}(\text{NEXTNODE})) = O(M)$. Thus for $N$ nodes, each of length $M$, the complexity of LINEAR is $O(MN)$.

Quality of solution of algorithm LINEAR in comparison to the optimal solution. For $M > 4$ and $N > 2$, let $R$ be the ratio of the largest compacted trie produced by LINEAR to the size of the optimally compacted trie. As for COMPRESS, we noted that $R \leq M+N/P$, where $P$ is the number of the non-null pointers. Now we can show that this limit is tight by the same example we used for COMPRESS (Example 3.1). As in the case of algorithm COMPRESS, it can be seen that LINEAR is an absolute approximate algorithm for $M = 2$, an $e$-approximate algorithm for $M = 3$, and an $f(n)$-approximate algorithm for $M > 3$.

**COROLLARY.** The three algorithms COMPRESS, PAIRS, and LINEAR, described above, are absolute approximate algorithms for $M = 2$, $e$-approximate algorithms for $M = 3$, and $f(n)$-approximate algorithms for $M > 3$.

**PROOF.** As shown with the description of each algorithm. □

### 4. EMPIRICAL RESULTS

The extent to which procedures COMPRESS, PAIRS, and LINEAR are able to compact the nodes will vary depending upon the order in which nodes are presented to the algorithm. In the hope that one ordering might be significantly better than another, three possible orderings were tested. They are

(1) **Level Order:** Selecting the nodes as they occur in the trie, top level first, left to right.

(2) **Least Last:** The nodes are ordered such that the number of non-null pointers in NODE($i$) is greater than or equal to the number of non-null pointers in NODE($i + 1$) for $1 \leq i < N$; in case of a tie, Level Order is used.

(3) **Least First:** the reverse order of Least Last.

Empirical results were obtained by running COMPRESS, PAIRS, and LINEAR on the set of tries shown in Appendix B and on a set of $N$ random tries, $N$ varying between 10 and 50 by increments of 10 as shown in Appendix B. Each node had a number of locations between 2 and 29 by increments of 3. The value of each node was taken to be a random binary number in the range $(0 : 2^M - 1)$, where $M$ is the number of locations. The results are shown in Appendices A and B. From the statistics in the appendices we observe the following:

(1) The performance of COMPRESS was consistently better—this is to be expected, since it takes more time than the other algorithms.

(2) The differences between the ordering strategies used in this experiment were not significant.

(3) The differences between the performances of the three algorithms decreased as $M$ increased in the random nodes experiment.

(4) Changing $N$ did not have a significant effect on the results in the random nodes case, as can be seen from the small values of the standard deviation in Appendix A.
<table>
<thead>
<tr>
<th>$N$</th>
<th>$M$</th>
<th>Linear</th>
<th>Compress</th>
<th>Pairs</th>
<th>Linear</th>
<th>Compress</th>
<th>Pairs</th>
<th>$*$ Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2</td>
<td>0.6499</td>
<td>0.6499</td>
<td>0.6499</td>
<td>0.6499</td>
<td>0.6499</td>
<td>0.6499</td>
<td>0.6999</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>0.6250</td>
<td>0.6250</td>
<td>0.6250</td>
<td>0.6499</td>
<td>0.6499</td>
<td>0.6499</td>
<td>0.6250</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>0.6166</td>
<td>0.6166</td>
<td>0.6166</td>
<td>0.6166</td>
<td>0.6166</td>
<td>0.6166</td>
<td>0.6166</td>
</tr>
<tr>
<td>40</td>
<td>2</td>
<td>0.5874</td>
<td>0.5874</td>
<td>0.5874</td>
<td>0.5874</td>
<td>0.5874</td>
<td>0.5874</td>
<td>0.5999</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
<td>0.5499</td>
<td>0.5499</td>
<td>0.5499</td>
<td>0.5499</td>
<td>0.5499</td>
<td>0.5499</td>
<td>0.5499</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>0.6799</td>
<td>0.5999</td>
<td>0.6199</td>
<td>0.6199</td>
<td>0.6199</td>
<td>0.6199</td>
<td>0.6999</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>0.6199</td>
<td>0.6099</td>
<td>0.6299</td>
<td>0.6199</td>
<td>0.6199</td>
<td>0.6199</td>
<td>0.6999</td>
</tr>
<tr>
<td>30</td>
<td>5</td>
<td>0.5799</td>
<td>0.5666</td>
<td>0.5799</td>
<td>0.5999</td>
<td>0.5999</td>
<td>0.5999</td>
<td>0.6999</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>0.6099</td>
<td>0.5799</td>
<td>0.6250</td>
<td>0.6149</td>
<td>0.6149</td>
<td>0.6149</td>
<td>0.6999</td>
</tr>
<tr>
<td>50</td>
<td>5</td>
<td>0.5799</td>
<td>0.5399</td>
<td>0.5999</td>
<td>0.5999</td>
<td>0.5999</td>
<td>0.5999</td>
<td>0.5999</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>0.6875</td>
<td>0.6875</td>
<td>0.6875</td>
<td>0.5999</td>
<td>0.5999</td>
<td>0.5999</td>
<td>0.6250</td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>0.6437</td>
<td>0.5874</td>
<td>0.5187</td>
<td>0.5124</td>
<td>0.5124</td>
<td>0.5124</td>
<td>0.6250</td>
</tr>
<tr>
<td>30</td>
<td>8</td>
<td>0.6166</td>
<td>0.5333</td>
<td>0.6250</td>
<td>0.5625</td>
<td>0.5625</td>
<td>0.5625</td>
<td>0.6624</td>
</tr>
<tr>
<td>40</td>
<td>8</td>
<td>0.6406</td>
<td>0.5687</td>
<td>0.6666</td>
<td>0.6187</td>
<td>0.6187</td>
<td>0.6187</td>
<td>0.6666</td>
</tr>
<tr>
<td>50</td>
<td>8</td>
<td>0.6199</td>
<td>0.5549</td>
<td>0.6274</td>
<td>0.6199</td>
<td>0.6199</td>
<td>0.6199</td>
<td>0.6999</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>0.6999</td>
<td>0.6999</td>
<td>0.6999</td>
<td>0.7363</td>
<td>0.7363</td>
<td>0.7363</td>
<td>0.7818</td>
</tr>
<tr>
<td>20</td>
<td>11</td>
<td>0.6480</td>
<td>0.6272</td>
<td>0.6408</td>
<td>0.6772</td>
<td>0.6772</td>
<td>0.6772</td>
<td>0.7818</td>
</tr>
<tr>
<td>30</td>
<td>11</td>
<td>0.6333</td>
<td>0.6242</td>
<td>0.6333</td>
<td>0.6878</td>
<td>0.6878</td>
<td>0.6878</td>
<td>0.7818</td>
</tr>
<tr>
<td>40</td>
<td>11</td>
<td>0.6522</td>
<td>0.6454</td>
<td>0.6522</td>
<td>0.7022</td>
<td>0.7022</td>
<td>0.7022</td>
<td>0.7818</td>
</tr>
<tr>
<td>50</td>
<td>11</td>
<td>0.6290</td>
<td>0.6199</td>
<td>0.6345</td>
<td>0.6690</td>
<td>0.6690</td>
<td>0.6690</td>
<td>0.7818</td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>0.7714</td>
<td>0.7714</td>
<td>0.7714</td>
<td>0.7642</td>
<td>0.7642</td>
<td>0.7642</td>
<td>0.7357</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>0.7285</td>
<td>0.6535</td>
<td>0.7285</td>
<td>0.6999</td>
<td>0.6999</td>
<td>0.6999</td>
<td>0.7357</td>
</tr>
<tr>
<td>30</td>
<td>14</td>
<td>0.7238</td>
<td>0.6690</td>
<td>0.7238</td>
<td>0.6928</td>
<td>0.6928</td>
<td>0.6928</td>
<td>0.7357</td>
</tr>
<tr>
<td>40</td>
<td>14</td>
<td>0.7357</td>
<td>0.6946</td>
<td>0.7357</td>
<td>0.7287</td>
<td>0.7287</td>
<td>0.7287</td>
<td>0.7357</td>
</tr>
<tr>
<td>50</td>
<td>14</td>
<td>0.7271</td>
<td>0.6828</td>
<td>0.7342</td>
<td>0.7057</td>
<td>0.7057</td>
<td>0.7057</td>
<td>0.7357</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>10</td>
<td>17</td>
<td>0.8294</td>
<td>0.8294</td>
<td>0.8068</td>
<td>0.8065</td>
<td>0.7646</td>
<td>0.8862</td>
<td>0.6862</td>
</tr>
<tr>
<td>20</td>
<td>17</td>
<td>0.7441</td>
<td>0.7235</td>
<td>0.7441</td>
<td>0.7235</td>
<td>0.6676</td>
<td>0.7235</td>
<td>0.7558</td>
</tr>
<tr>
<td>30</td>
<td>17</td>
<td>0.7294</td>
<td>0.7117</td>
<td>0.7111</td>
<td>0.7392</td>
<td>0.7019</td>
<td>0.7470</td>
<td>0.7862</td>
</tr>
<tr>
<td>40</td>
<td>17</td>
<td>0.7617</td>
<td>0.7485</td>
<td>0.7705</td>
<td>0.7397</td>
<td>0.7014</td>
<td>0.7397</td>
<td>0.7794</td>
</tr>
<tr>
<td>50</td>
<td>17</td>
<td>0.7470</td>
<td>0.7211</td>
<td>0.7541</td>
<td>0.7341</td>
<td>0.7046</td>
<td>0.7388</td>
<td>0.7517</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>0.8049</td>
<td>0.8049</td>
<td>0.8049</td>
<td>0.8249</td>
<td>0.8249</td>
<td>0.8249</td>
<td>0.7999</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>0.7324</td>
<td>0.7324</td>
<td>0.7424</td>
<td>0.7524</td>
<td>0.7500</td>
<td>0.7524</td>
<td>0.7774</td>
</tr>
<tr>
<td>30</td>
<td>20</td>
<td>0.7416</td>
<td>0.7416</td>
<td>0.7483</td>
<td>0.7533</td>
<td>0.7366</td>
<td>0.7649</td>
<td>0.7833</td>
</tr>
<tr>
<td>40</td>
<td>20</td>
<td>0.7849</td>
<td>0.7849</td>
<td>0.7899</td>
<td>0.7924</td>
<td>0.7749</td>
<td>0.7924</td>
<td>0.7924</td>
</tr>
<tr>
<td>50</td>
<td>20</td>
<td>0.7759</td>
<td>0.7599</td>
<td>0.7799</td>
<td>0.7829</td>
<td>0.7329</td>
<td>0.7789</td>
<td>0.7839</td>
</tr>
<tr>
<td>10</td>
<td>23</td>
<td>0.7912</td>
<td>0.7912</td>
<td>0.7912</td>
<td>0.8043</td>
<td>0.8043</td>
<td>0.8043</td>
<td>0.7304</td>
</tr>
<tr>
<td>20</td>
<td>23</td>
<td>0.7652</td>
<td>0.7652</td>
<td>0.7826</td>
<td>0.7891</td>
<td>0.7891</td>
<td>0.7891</td>
<td>0.7586</td>
</tr>
<tr>
<td>30</td>
<td>23</td>
<td>0.7941</td>
<td>0.7941</td>
<td>0.8057</td>
<td>0.7869</td>
<td>0.7869</td>
<td>0.7869</td>
<td>0.7927</td>
</tr>
<tr>
<td>40</td>
<td>23</td>
<td>0.8132</td>
<td>0.8132</td>
<td>0.8238</td>
<td>0.7847</td>
<td>0.7847</td>
<td>0.7847</td>
<td>0.7815</td>
</tr>
<tr>
<td>50</td>
<td>23</td>
<td>0.8173</td>
<td>0.8095</td>
<td>0.8243</td>
<td>0.7912</td>
<td>0.7860</td>
<td>0.7912</td>
<td>0.7912</td>
</tr>
<tr>
<td>10</td>
<td>26</td>
<td>0.8384</td>
<td>0.8384</td>
<td>0.8384</td>
<td>0.8769</td>
<td>0.8769</td>
<td>0.8769</td>
<td>0.7538</td>
</tr>
<tr>
<td>20</td>
<td>26</td>
<td>0.7846</td>
<td>0.7846</td>
<td>0.7846</td>
<td>0.8038</td>
<td>0.8038</td>
<td>0.8038</td>
<td>0.8038</td>
</tr>
<tr>
<td>30</td>
<td>26</td>
<td>0.8025</td>
<td>0.8025</td>
<td>0.8025</td>
<td>0.7934</td>
<td>0.7934</td>
<td>0.7934</td>
<td>0.7974</td>
</tr>
<tr>
<td>40</td>
<td>26</td>
<td>0.8249</td>
<td>0.8249</td>
<td>0.8249</td>
<td>0.8067</td>
<td>0.8067</td>
<td>0.8067</td>
<td>0.8346</td>
</tr>
<tr>
<td>50</td>
<td>26</td>
<td>0.8169</td>
<td>0.8159</td>
<td>0.8159</td>
<td>0.7976</td>
<td>0.7976</td>
<td>0.7976</td>
<td>0.8222</td>
</tr>
<tr>
<td>10</td>
<td>29</td>
<td>0.8482</td>
<td>0.8482</td>
<td>0.8482</td>
<td>0.8310</td>
<td>0.8310</td>
<td>0.8310</td>
<td>0.7551</td>
</tr>
<tr>
<td>20</td>
<td>29</td>
<td>0.8189</td>
<td>0.8189</td>
<td>0.8189</td>
<td>0.8292</td>
<td>0.8292</td>
<td>0.8292</td>
<td>0.7965</td>
</tr>
<tr>
<td>30</td>
<td>29</td>
<td>0.8202</td>
<td>0.8202</td>
<td>0.8202</td>
<td>0.8126</td>
<td>0.8126</td>
<td>0.8126</td>
<td>0.8496</td>
</tr>
<tr>
<td>40</td>
<td>29</td>
<td>0.8474</td>
<td>0.8474</td>
<td>0.8474</td>
<td>0.8430</td>
<td>0.8430</td>
<td>0.8430</td>
<td>0.8525</td>
</tr>
<tr>
<td>50</td>
<td>29</td>
<td>0.8310</td>
<td>0.8310</td>
<td>0.8310</td>
<td>0.8275</td>
<td>0.8193</td>
<td>0.8275</td>
<td>0.8441</td>
</tr>
</tbody>
</table>
(5) As \( M \) grows larger, the ratio of the compacted trie length to \( M \times N \) grows larger, and this can be explained by the fact that the probability of compacting two nodes into one decreases as \( M \) increases.

(6) The maximum difference between the results of the three algorithms in these experiments was about 6.7 percent between algorithms LINEAR and COMPRESS, for \( M = 11 \) in the random nodes case. This difference could be as high as 10.6 percent if algorithm COMPRESS gave an optimal solution.

(7) In the random nodes experiment, the compacted tries resulted in a savings of about 20 percent in size; and in the realistic tries experiment, the compacted tries saved about 80 percent of the space.

(8) In many cases the performance of PAIRS was bad, even worse than LINEAR. This could be explained by the argument that PAIRS tends to grow large nodes as it compacts, and the probability that two nodes will be compacted in the smallest space decreases as the size of the nodes increases.

From the above observations we conclude the following about the three approximation algorithms:

1. \( M \) is a more crucial factor than \( N \) in the performance of the algorithms.
2. Considering the time each of the three algorithms takes, and the small difference between the results, one might just as well use algorithm LINEAR to compact a trie.
3. Since the difference between the results for the ordering strategies used was not significant, any random ordering of the nodes will suffice.
4. For realistic tries, the space saved by compaction is significant.

APPENDIX A

The results of running the three algorithms on a set of \( n \) random nodes, each node with \( M \) locations, are shown in Table I. \( N \) varied from 10 to 50 by 10, and \( M \) varied from 2 to 29 by 3.

The first two columns of the table give \( N \) and \( M \). Columns 3, 4, and 5 give the ratio of the compacted trie to \( M \times N \), as given by algorithms LINEAR, COMPRESS, and PAIRS, respectively, with the nodes in the order generated, which could be considered as the Level Order.

Columns 6, 7, and 8 give the same results for the nodes ordered in the Least Last Order; columns 9, 10, 11 give the same results for the nodes ordered in the Least First Order.

Column 12 gives a lower bound on the ratio of optimal size to \( M \times N \), where the optimal size is bounded from below by the number of ones in the strings generated.

Column 13 gives a tighter lower bound on the ratio of optimal size to \( M \times N \), FOR \( N = 10 \), AND \( M = 2 \) TO 29 BY 3. This lower bound was obtained by counting the number of zeros that cannot be covered by any combination of the nodes, adding it to the number of ones, and then dividing by \( M \times N \).

APPENDIX B

Examples of actual tries used as tests for the three approximate algorithms are

- **Example 1.** The most used 100 words of English [12].
- **Example 2.** The most used 31 words of English [12].

Table II.

<table>
<thead>
<tr>
<th>Example</th>
<th>M</th>
<th>N</th>
<th>Time</th>
<th>Least Last Order</th>
<th>Least First Order</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>O(N)</td>
<td>O(N²)</td>
<td>O(NlogN)</td>
</tr>
<tr>
<td>1</td>
<td>27</td>
<td>40</td>
<td>.1925</td>
<td>.1451</td>
<td>.2148</td>
</tr>
<tr>
<td>2</td>
<td>27</td>
<td>12</td>
<td>.2222</td>
<td>.2222</td>
<td>.2746</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td>6</td>
<td>.2500</td>
<td>.2500</td>
<td>.2592</td>
</tr>
<tr>
<td>4</td>
<td>27</td>
<td>12</td>
<td>.1573</td>
<td>.1612</td>
<td>.1913</td>
</tr>
</tbody>
</table>

*Values represent the ratio of the compacted trie to $MN$.

**Example 3.** The SPARKS reserved words; see Example 1.1.

**Example 4.** The following example taken from [14]: BACK, BAN, BAND, BANE, BANG, BANK, BAR, BARGE, BARREL, BARB, BARE, BARK, BARN, BANDIT, BARGAIN, BE, BEE, BEEN.

REFERENCES


Received September 1983; accepted November 1983