Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports
Department of Computer Science

1977

Heuristics for Trie Index Minimization

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Report Number:
77-224

Comer, Douglas E., "Heuristics for Trie Index Minimization" (1977). *Department of Computer Science Technical Reports.* Paper 164.
https://docs.lib.purdue.edu/cstech/164

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

Heuristics For
Trie Index Minimization

Douglas Comer
Computer Sciences Department
Purdue University
W. Lafayette, Indiana 47907

February, 1977
CSD-TR 224

Abstract

A trie is a particular implementation of a digital search tree in which leaves correspond to records in a file. Searching proceeds from the root to a leaf, where the decision at each node is based on the value of some attribute in the query. Trie implementations have the advantage of being fast, but the disadvantage of achieving that speed at great expense in storage space. Of primary concern in making a trie practical is the problem of minimizing storage. One method for reducing the space required for a trie is to reorder the testing of attributes. Unfortunately, the problem of finding an ordering which guarantees a minimum size trie is NP-complete. In this paper we investigate several heuristics for reordering attribute testing, and derive bounds on the sizes of the worst tries produced by them in terms of the underlying file. Although the analysis is derived for a binary file, the results are extended to files of higher degree.

An alternative representation of a trie, called an O-trie, is examined and is shown to guarantee minimum storage requirements for binary files. For files of higher degree, a bound on the size of the worst O-trie is obtained. For most applications, O-tries are smaller than other implementations of tries, even when heuristics for improving the storage requirements are employed.

1. Introduction:

A trie[1], described by Fredkin [FRED60], is a tree structure in which information is stored character-by-character or digit-by-digit. Figure 1 shows a trie for the words "many", "mane", "map", "mat", and "me". A query is a word which must be looked up in the trie. To search for a word, one begins at the root and follows a path with labels which are the same as the letters of the query. This storage structure has the advantage that the search is quite fast, taking no longer than the number of letters in the query, provided that the time required to decide which path to follow at each node is constant. Fredkin's implementation has the fixed time decision property because each choice costs only as much as an array indexing operation. Sussenguth [SUSS63] proposes an alternative implementation he calls a "doubly chained tree". In the tree implementation, all sons of a node are chained together in a linked list. Thus, the decision about which path to follow cannot be made in constant time but requires searching. In general, following paths which appear early in the lists costs less than following those which appear later. Figure 2 demonstrates this implementation by showing a tree corresponding to the trie shown in Figure 1. While our results are stated in terms of a trie, they apply equally to either implementation.

A trie may be thought of as a storage structure for information from a file. Figure 3 gives a file with $r = 5$ records and $k = 4$ attributes. In this case, each attribute is a single character. There is no reason, however, that an attribute could not take on

---

1 pronounced "try"

values from a larger or smaller set. In practice, the attributes might be fields in a record like "salary" or "employee number". Since the trie implementation requires storage at each node proportional to the size of the set of possible attribute values, it is often convenient to use the individual characters or digits of natural attributes to create smaller attributes.

In comparing the trie in Figure 1 to the file shown in Figure 3, one can see that each node in the trie corresponds to some subset of the records of the file. Each leaf corresponds to exactly one record and each nonleaf node corresponds to the subset of records represented by the leaves of the subtree rooted at that node. We will assume that the trie has no internal node which corresponds to only one record. This definition, called a pruned trie, is primarily made to lessen the amount of storage required by the trie. By "pruning" all chains which lead to leaves, some information from the file may be lost. In most instances a trie will be used as an index for a file, perhaps on secondary storage, and presumably the record of this file will have to be examined to complete the query. A consequence of the definition, which will be used later, is that each internal node must correspond to at least two records in the file (or it would become a leaf).

Since the trie was originally intended for the storage of alphabetic character strings, the order of testing attributes was understandably left-to-right as in Figure 1. When a query is viewed as a k-tuple in which attributes are unrelated, the left-to-right order is no longer natural or necessary. Rotwitt and deMaine [RODE71] note that the order in which attributes are tested may influence the size of the resultant trie. Consider, for example, the trie shown in Figure 4,

constructed from the same file as the trie shown in Figure 1, but by testing the letters from right to left. By changing the order of testing, the size of the trie, measured in internal nodes, decreases from 4 to 2 nodes. Because the chief disadvantage of a trie is the large storage space required, reordering attribute testing to reduce the size is an attractive proposition. We would like to find an ordering of attributes that yields a minimum size trie. While the reordering of attributes also affects the time taken to access a given record, that will not concern us. Others [CASE73, PATT69, STAN70, SUSS63] have explored problems of access time minimization.

Unfortunately, the problem of reordering attributes to produce a minimum size trie is computationally difficult[2] [COSE76]. In this paper we consider alternatives to finding a minimum size trie for a given file. One method employs computationally efficient procedures which produce tries which, while they may not be minimum, are smaller than a randomly ordered trie. Often, such procedures are based on "rule of thumb" practices and will be called heuristics. Rotwitt and deMaine [RODE71] and Yao [YAO76] have proposed two heuristics.

Another approach to minimizing tries uses a modified implementation in which information about the ordering of attributes is contained in the trie itself. Such tries are called O-tries (for Order containing) and are shown to be superior to the heuristics examined.

Definitions of a trie, file, and query are given in [COSE76]. Graph definitions used throughout the paper are standard.

---

[2] the problem was shown to be NP-complete. [AHU74] is a good reference for NP-complete problems.

2. Elimination of Useless Attributes:

We will call a file in which any attribute takes on values from a set of at most n elements an n-ary file or say that the file has degree n. Initially, we will consider the performance of heuristics on binary files. The study of binary files is motivated on two grounds. First, most computer systems store information in binary, so one could think of a binary file as the hardware representation of an arbitrary file. Secondly, examination of the binary case is important for analysis of files with higher degree.

Consider a trie for some binary file as shown in Figure 5. We will say that a node is a binary node if it has exactly 2 sons. The trie in question has some depths[3] at which no binary nodes appear. In terms of the file, some attribute was tested which did not further divide the sets of records. An attribute which produces no binary nodes when tested is said to be useless because its omission would result in a smaller trie. Note that the property of being useless is related to the testing of an attribute in a particular trie and cannot be determined from examination of the attribute values *a priori*.

The first heuristic for minimizing tries is one which eliminates useless attributes. Relating this to the reordering of attribute testing, we may think of the useless attributes as being moved to the end of the order (where they are never reached during a search).

HEURISTIC 1 (Elimination of Useless Attributes): When building a trie, select at each depth an attribute which adds at least one binary node. ▯

---

[3] the root of a trie lies at depth 0, and the sons of a node at depth i lie at depth i+1.

The size of tries produced by this heuristic may vary from the size of an optimal trie for the file. Let S_h denote the size of a trie produced by some heuristic, and let S_o denote the size of an optimal (smallest) trie for the same file. Then the cost criterion for the heuristic will be

$$\text{Cost} = S_h/S_o$$

Heuristics with low cost are desirable. For any heuristic, it is important to know the largest cost that can be incurred.

In order to bound the cost of Heuristic 1, we will define a binary tree called an (r,k)-WIDE tree and show that it is as large as any trie for a binary file of r records and k attributes produced by Heuristic 1. To see how the definition of an (r,k)-WIDE tree arises, consider the largest trie for a binary file from which all useless attributes have been removed. For the present, assume that r is even. Recall that each internal node in the trie must correspond to at least 2 records in the file. Therefore, there can be no more than $r/2$ nonleaf nodes at any depth. Suppose that at some depth, q, $r/2$ nodes do appear. Since at least one of these must be a binary node, there can be at most $r/2 - 1$ internal nodes at depth $q + 1$. Similarly, there can be at most $r/2 - 1$ internal nodes at depth $q - 1$ (because at least one must have been a binary node and no leaves had appeared by depth q). By a simple induction argument, there can be at most $r/2 - i$ internal nodes at depths $q + i$ and $q - i$. Thus, the trie will display linear growth until a depth is reached with $r/2$ internal nodes. After that, linear shrinkage must follow. An immediate consequence is stated in Lemma 1.

LEMMA 1: If F is a binary file of r records and k attributes, and T is a trie for F produced by Heuristic 1, then T can have no leaf at depth greater than $r - 1$.

PROOF: From the above discussion, no depth can have more than $r/2$ nodes, and preceding depths must have one less node each. Let q denote the depth with $r/2$ nodes. Since the root lies at depth 0, $q \leq r/2 - 1$. Including the leaves, there can be at most $r/2$ depths following q, so the Lemma holds. $\square$

Of course, it may be that $k < r - 1$ in which case the trie would be still shorter.

Since we wish to find the largest trie allowed by Heuristic 1, and there is a limit on the height, we must include the "widest" parts possible. Figure 6 shows what happens as k grows smaller. The first depths of the trie shown there form a complete binary tree with 2^t nodes at depth t. Then the linear growth to $r/2$ nodes begins as when k is unbounded. After reaching a depth where there are $r/2$ internal nodes, successive depths shrink linearly until the last possible depth at which all leaves appear. The parameter t is chosen to be the minimum height binary tree needed to distinguish all nodes.

If h is the height of the trie shown in Figure 6, then there will be $h - t - 1$ depths remaining after the complete binary tree, not including the depth which has $r/2$ nodes. Thus the number of depths of linear growth is limited to $(h-t-1)/2$. At each one, there is one more node than at the last, and the sequence terminates in $r/2$ nodes. Thus, the first depth in the sequence must have $r/2 - (h-t-1)/2$ nodes. This expression is called "p" in the definition. To define t, we must have that t is at least large enough to guarantee $2^{t+1} \geq r/2 - (h-t-1)/2$.

With some rearrangement, this yields the following definition.

DEFINITION 1: Let r, k be integers s.t. $1 < \lceil \log_2 r \rceil \leq k$, and let t be the least nonnegative integer satisfying $2^{t+2} - (t+2) \geq r - k$. Let h be the min(k-1,r-2), and let $p = \lceil r/2 - (h-t-1)/2 \rceil$. Then an <u>(r,k)-WIDE tree</u> is a binary tree s.t.

1. Each node at depth d, $0 \leq d < t$ has 2 sons.
2. At depth t, $p - 2^t$ nodes have 2 sons and all other nodes have 1 son (i.e. there are p nodes at depth t+1).
3. Exactly one node at depth d, $t+1 \leq d < h$ has 2 sons, all other nodes at depth d have 1 son. After a depth with r/2 internal nodes, the sons of the binary node are both leaves.
4. All nodes at depth h have 2 sons. □

We can now establish that an (r,k)-WIDE tree is as large as any trie for a binary file produced by Heuristic 1.

LEMMA 2: Let r, k be integers s.t. $1 < \lceil \log_2 r \rceil \leq k$, let F be a binary file of r records and k attributes, and let T be a trie for F produced by Heuristic 1. If W is an (r,k)-WIDE tree, then

$$|W| \geq |T|$$

where $|W|$ denotes the size of W measured in internal nodes.

PROOF: Suppose that $|T| > |W|$. From Lemma 1 and the definition of W, T can have no leaf at depth greater than the depth of a leaf in W. Therefore, it must hold that T has more nodes than W at some depth. Let d be the first depth at which T has more nodes than W, and let t be as in the definition of W. Then two cases arise.

<u>Case 1</u> ($d \leq t+1$): Since each node at depths 0 to t in W has 2 sons,

T cannot have more nodes at depth d than W. Therefore, case 2 must hold.

Case 2 ($d > t+1$): Consider the sequence given by the number of nodes at depths t+1, t+2, ... in W. We will call this sequence the profile of W. Beginning at depth t+1 W has, say, p nodes. Thus the profile is:

$$P_W = p, p+1, \ldots, j, j+1, j+2, \ldots, r/2-1, r/2, r/2-1, \ldots, p+1, p$$

Since T has more nodes at depth d, it must have a profile

$$P_T = p, p+1, \ldots, j, j+n, j+n+1, \ldots, r/2-1, r/2, r/2-1, \ldots, p-n-1, p-n$$

where depth d is shown to have j+1 nodes in W and j+n nodes in T. The point to note is that by choosing a larger value than j+1, T cannot have as large values near the end of the profile as W. Comparing the two profiles we find that

$$\sum_{i=1}^{n} (j+i) > \sum_{i=1}^{n} (p+i)$$

so $P_W > P_T$.

Thus, the assumption that $|T|>|W|$ was false and the Lemma holds. []

From Lemma 2 we know that a bound on the size of an (r,k)-WIDE tree is also a bound on the worst case trie produced by Heuristic 1. The size of an (r,k)-WIDE tree can be computed by summing over the number of nodes at each depth. From the definition, there are several possible contributions depending on r and k. For depths 0 through t there are $2^{t+1}-1$ nodes because these depths form a complete binary tree. As shown in the proof of Lemma 2, the number of nodes at depths t+1,... is given by the profile of W, with p as in the definition. Thus, the size of an (r,k)-WIDE tree is:

$$W = (2^{t+1}-1) + r/2 + 2\sum_{i=p}^{r/2-1} (i) + f(t) \qquad (1)$$

where t depends on r and k as in the definition. The function f is either 0 or p depending on whether there are an even or odd number of depths after depth t in the tree. With an even number of depths, the last one will contain only p nodes.

To see at what values of k this expression is maximized, we need only make a few observations. From the definition of an (r,k)-WIDE tree we know that t is the least integer s.t. $2^{t+2} - (t+2) \geq r - k$. Thus, t is maximized when r - k is maximized. Since $k \geq \lceil \log_2 r \rceil$, r - k is at most $r - \lceil \log_2 r \rceil$. Thus, $2^{(t-1)+2} - ((t-1)+2) < r - \lceil \log_2 r \rceil$, from which it follows that $2^{t+2} = O(r)$. So in the worst case, the first term in (1) is proportional to r.

The second part of (1) is more interesting. To maximize the sum from p to r/2-1, it is necessary to minimize p. Recall that p is given by $\lceil r/2 - (h-t-1)/2 \rceil$. When $k > r - 1$, t = 0 and h achieves its maximum of r - 2 . Thus, the minimum value for p is 2 and corresponds to a trie with linear growth and linear shrinkage. The size of the worst case trie is bounded as follows.

$$|W| \leq 1 + r/2 + 2 \sum_{i=2}^{r/2-1} (i) + 1 = r^2 / 4 \qquad (2)$$

To see that this bound is the best possible, consider the file shown in Figure 7. Testing attributes left to right produces a trie with size $r^2 / 4$, in which each depth has at least one binary node. Testing attributes from right to left in the same file produces a trie of only r - 1 internal nodes. Recall that a binary tree with r leaves must have r - 1 internal nodes. Thus, we can conclude Theorem 1.

THEOREM 1: The cost of Heuristic 1 for binary files is

$$S_{H1} / S_0 \leq (r^2/4)/(r-1)$$

PROOF: From Lemma 2, an (r,k)-WIDE tree is as large as any trie for a binary file produced by Heuristic 1. From the above analysis, the size of an (r,k)-WIDE tree is bounded by (2). The Theorem follows. ▯

Note that the ratio $r^2/4(r-1)$ is not bounded by a constant, but grows as the number of records in the file.

3. Other Heuristics:

In Heuristic 1 an attempt was made to reduce the space requirements of a trie by eliminating useless attributes. As an extension to that, suppose one were to choose an attribute which gave the most nodes at each depth 2 sons. This would tend to break up the sets of records very fast, and might yield leaves earlier in the trie. In order to insure that leaves are distinguished as early as possible, we will further stipulate that is two or more attributes would add the same number of nodes at the next depth, then an attribute which distinguishes the most leaves should be selected from among them. These ideas are encompassed in the Splitting heuristic.

HEURISTIC 2 (Splitting Heuristic): When building a trie select at each depth an attribute which adds the most nodes (including leaves). Among all attributes adding the maximum number of nodes, select one which adds the most leaves. ▯

Note that the Splitting heuristic must choose an attribute which

distinguishes at least one node at each depth. From Theorem 1, no trie produced by the Splitting heuristic can have more than $r^2/4$ nodes. We will show that the Splitting heuristic is $O(r^2)$ in the worst case.

Consider the file shown in Figure 8. Observe that testing the attributes in this file from right to left yields a trie of size $r - 1$. We will see that testing left to right produces a trie of $O(r^2)$ nodes and that the Splitting heuristic allows this order. To see how the Splitting heuristic can make poor choices, observe that no attribute can produce a leaf on the first selection, and that all attributes will add 2 nodes at depth 1 . Choosing the leftmost attribute divides the records into sets consisting of records 1 - 4, and the rest of the file. For this division we again observe that no attribute will add any leaves and any attribute will cause 3 nodes to appear at depth 2. Selections can be made left to right until there are $r/4$ nodes at depth $r/4 - 1$, corresponding to 4 records each. The remaining choices are made in pairs since after a choice places two nodes in a set the heuristic causes a leaf to appear.

We need only consider the first $r/4$ depths of the trie for the file shown in Figure 8 to establish a lower bound on the size. Since there is linear growth, the size of the trie can be bounded from below by

$$\text{Size} \geq \sum_{i=1}^{r/4} i = (r^2 + 4r)/32$$

From this we can conclude the cost of the Splitting heuristic.

THEOREM 2: The cost of Heuristic 2 for binary files is

$$S_{H2} \;/\; S_o = O(r)$$

PROOF: From Theorem 1, Heuristic 2 has $S_{H2} \leq r^2/4$. A trie produced by Heuristic 2 for the file shown in Figure 8 has at least $c \cdot r^2$ nodes,

where c is a constant. Since the optimal trie for this file has r - 1 nodes, $S_{H2} / S_o = O(r)$. ▯

We will now turn our attention to two heuristics which extend the idea of generating leaves used in the Splitting heuristic. One of the Greedy heuristics simply reverses the criteria used in the Splitting heuristic and chooses an attribute at each depth which yields the most leaves, selecting from among all those attributes adding a maximum number of leaves one which adds the most internal nodes. Although the second Greedy heuristic chooses an attribute which adds the most leaves, as a secondary consideration it will choose an attribute adding the least internal nodes. The ideas for these heuristics come from the Splitting heuristic where we attempted to divide the sets of records as fast as possible to generate leaves early in the trie. On one hand, it might seem reasonable to try to divide the internal nodes as fast as possible given two attributes which would both add the same number of leaves. On the other hand, it might turn out that by generating many internal nodes the trie would become too "wide".

Unfortunately, neither of these methods has a better worst case than the Splitting heuristic. We will demonstrate their performance after giving the definitions.

HEURISTIC 3 (Greedy Heuristic): When building a trie select at each depth an attribute which adds the most leaves. Among all those attributes adding the maximum number of leaves, select one which adds the most internal nodes. ▯

HEURISTIC 4 (Leaf Greedy Heuristic): When building a trie select at each depth an attribute which adds the most leaves. Among all

attributes adding the maximum number of leaves, select one which adds the least internal nodes. []

Because both of the Greedy heuristics will always force at least one binary node at each depth, they are each at least as good as Heuristic 1. We will demonstrate, however, that both Greedy heuristics can produce tries which require $O(r^2)$ space in the worst case by again using the file shown in Figure 8. To see that Heuristic 3 can perform badly on this file, consider the first choice. Since the first choice can add no leaves, the leftmost attribute can be selected. Continuing to select attributes left to right until $r/4$ nodes appear is allowed because after each selection, no leaves can be generated and any attribute will add at most one internal node to the next depth. From the proof of Theorem 2 we can conclude the following.

COROLLARY 1: The cost of Heuristic 3 for binary files is

$$\text{Cost} = O(r)$$

PROOF: Immediate from the above discussion. []

COROLLARY 2: The cost of Heuristic 4 for binary files is

$$\text{Cost} = O(r)$$

PROOF: We need only point out that at each of the first $r/4$ depths of a trie for the file shown in Figure 8, the least number of nodes that can be added is 1. Therefore a left to right ordering of attribute testing is also allowed by Heuristic 4. The result follows. []

Note that the sample file is used here only to establish an asymptotic bound. Other files will have costs with larger constants.

4. O-Tries, An Alternative Implementation:

In considering heuristics for minimizing the space required by a trie we have assumed a global ordering of attributes was necessary, and that the ith attribute in this ordering would be tested at a node at depth i in the trie. If we are willing to relax that requirement, then the size of the trie can be further reduced. At a cost of a small amount of extra space in each node, information specifying which attribute to test upon reaching the node could be stored in it. The ordering of attribute testing being made explicit would allow different orders along different paths from the root to a leaf. This implementation of a trie will be called an O-trie or Order-containing trie. Figure 9 shows one possible O-trie for the strings in the trie of Figure 1. Numbers in the nodes of the O-trie are the positions of letters which should be tested. Since there are k possible positions, $\log_2 k$ extra bits would be needed in each node.

One way to build an O-trie is to start with an arbitrary attribute order, construct a trie, and then reorder attribute testing within the various subtrees to reduce the size. For the binary case, an obvious generalization of Heuristic 1 (Elimination of Useless Attributes) works well in O-tries. We can state the procedure as a heuristic.

HEURISTIC 5 (Elimination of Useless Attributes for O-Tries): When building an O-trie select at each node an attribute which causes the node to have at least 2 sons. □

This heuristic may be viewed as the elimination of internal chains. It was exactly the problem of internal chains which gave us a large worst

case for Heuristics 1 - 4 . With the chains removed, we have that

THEOREM 3: Any 0-trie for a binary file produced by Heuristic 5 is optimal.

PROOF: Since each node in the trie has 2 sons, there are at most r - 1 internal nodes. Since there are r leaves, $S_{H5} / S_o = 1$. □

5. The Performance of Heuristics on Files of Degree > 2:

Recall that a file is of degree n if the maximum of the sizes of the value sets of its attributes is n. It should be clear that if only one attribute has a value set with n items and all others are binary, then the performance of Heuristics 1 - 4 will be asymptotically the same. Even if we require every attribute to take on n values, Heuristics 1 - 4 can be "fooled" by relegating the higher values to a set of n records appended to an otherwise binary file. After distinguishing the n records, the heuristics will then generate a trie on the remaining subfile as shown before.

Let us direct our attention now to the performance of Heuristic 5 on files of higher degree. We have already seen that, in the binary case, it can produce optimal tries. Although it will not hold that tries for higher degree files remain optimal, the cost of such tries will be bounded by n and will not grow as the number of records. To derive this bound, we first observe that an 0-trie produced by Heuristic 5 will have at most r - 1 internal nodes for a file of r records. Suppose that the file for some 0-trie was of degree n, n > 2. The size of a complete n-ary tree of r leaves gives the size of the smallest possible trie for F. If there are r leaves in the trie we

we will assume that $r = n^p$. At depth i in the complete n-ary tree there are n^i nodes, so the size is:

$$S_o = \sum_{i=0}^{p-1} n^i = (n^p - 1)/(n - 1)$$

and this gives

$$S_{H5} / S_o = (r - 1)/((n^p - 1)/(n - 1)$$

and since $r = n^p$,

$$S_{H5} / S_o = (r - 1)(n - 1)/(r - 1) = n - 1 \qquad (3)$$

We summarize this in the following Theorem.

THEOREM 4: The cost of Heuristic 5 for files of degree n is at most $n - 1$.

PROOF: Given in the analysis above. ▯

A few remarks are in order about this bound. Unlike the other bounds stated in this paper, this bound is loose in the sense that we have not shown a set of files on which it is always attainable. Whether such a set exists is an open question at this point. More important, however, is the fact that the bound does not depend on the size of the input, r or k, but only on the maximum value that is stored in the file. Since in practice, one would expect n to be small compared to r, the O-trie can be recommended as a space-efficient implementation.

6. Conclusions:

We have examined several heuristics for space minimization in

In the worst case, tries produced by four obvious heuristics required space proportional to r^2, where r is the number of records in the underlying file. The optimal trie for the same files required only $r - 1$ records, so the cost of the heuristics was proportional to r. In each case it was demonstrated that the asymptotic bound could be attained, making these bounds the best possible. Moreover, the worst case example used a binary file so the results hold for files of any degree d, $d > 2$.

Turing to an alternative implementation of a trie, called an O-trie, it was shown that a simple method of constructing O-tries for binary files could produce tries the minimum space in terms of the number of nodes. The size of each node in an O-trie, however, must be slightly larger than in other implementations. It was also shown that a bound on the size of the worst case O-tries for files of higher degree could be obtained which depended only on the degree of the file and not on the number of records. The O-trie implementation is therefore recommended especially when the number of records in the file is large compared to the degree of the file.

References

[AHU74] Aho, A.; Hopcroft, J.; and Ullman, J., The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.

[CASE73] Casey, R. G., "Design of Tree Structures for Efficient Querying," CACM, vol. 1:9 (Sept 1973), 549-536.

[COSE76] Comer, D.; and Sethi, R., "Complexity of Trie Index Construction," JACM, to appear.

[FRED60] Fredkin, E., "Trie Memory," CACM, vol 3:9 (Sept 1960), 490-499.

[PATT69] Patt, Y., "Variable Length Tree Structures Having Minimum Average Search Time," CACM, vol. 12:2 (Feb 1969), 72-76.

[RODE71] Rotwitt, T.; and deMaine, P. A. D., "Storage Optimization of Tree Structured Files Representing Descriptor Sets," Proc 1971 ACM-SIGFIDET Workshop on Data Description Access and Control, 207-217.

[STAN70] Stanfel, L., "Tree Structures for Optimal Searching," JACM, vol. 17:2 (July 1970), 508-517.

[SUSS63] Sussenguth, E. H., "Use of Tree Structures for Processing Files," CACM, vol. 6:5 (May 1963), 272-279.

[YAO76] Yao, S. B., "A Model for Combined Attribute Index Organization," Proc Fifth Texas Conference On Computing, Austin, 1976, 127-130.

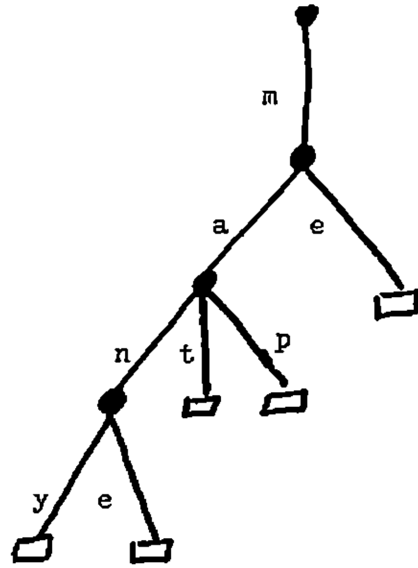

Figure 1. A trie for the strings "many", "map", "mat", "mane", and "me".

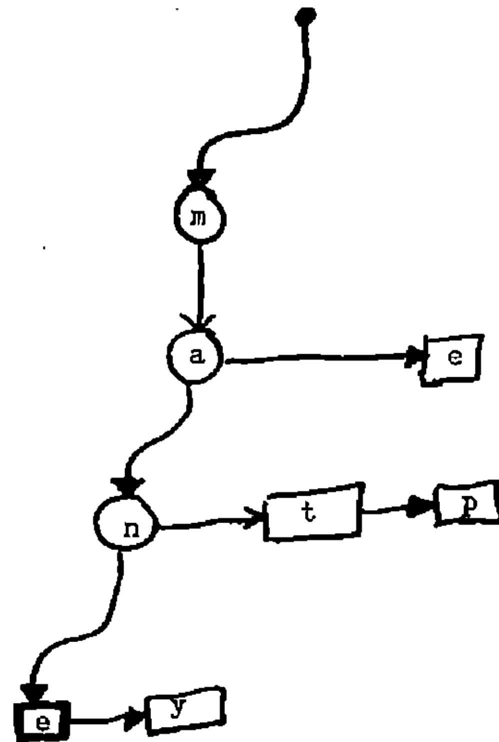

Figure 2. The Tree implementation of the strings shown in Figure 1.

records		attributes			
	1	m	a	p	b̸
	2	m	a	t	b̸
	3	m	a	n	y
	4	m	e	b̸	b̸
	5	m	a	n	e

Figure 3. A file of 5 records and 4 attributes.

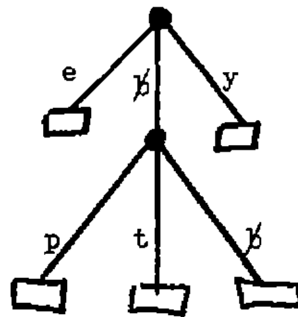

Figure 4. A trie for the strings in the file of Figure 3 formed by testing attributes right to left.

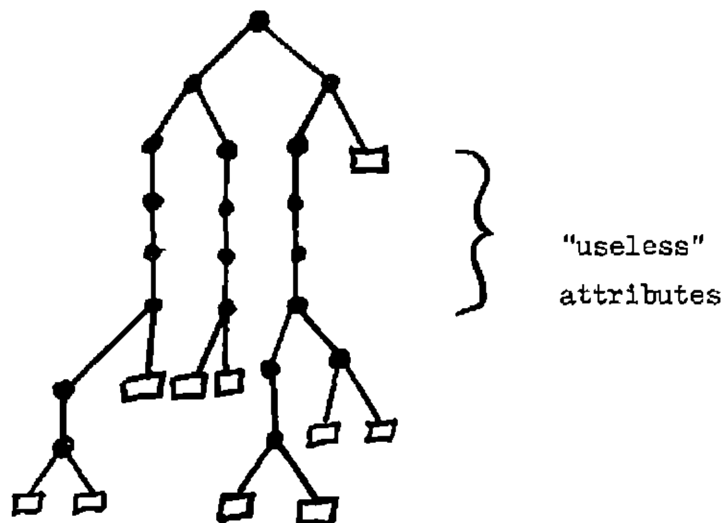

Figure 5. An aribtrary trie for a binary file. Some depths have useless attributes which can be eliminated.

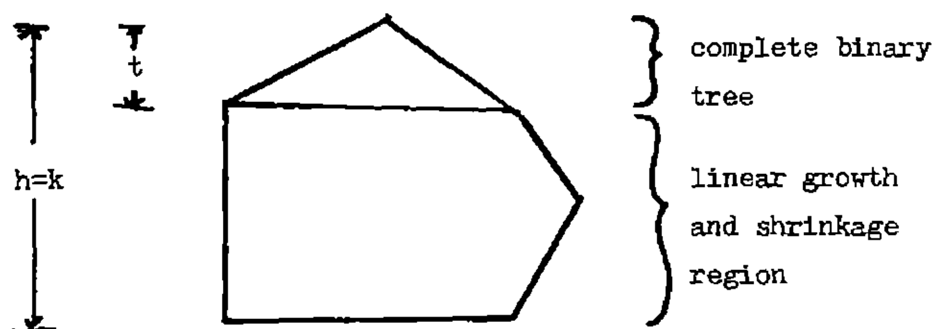

Figure 6. The shape of a worst case trie for a binary file produced by Heuristic 1.

<table>
<tr><td>1
1</td><td></td><td></td><td></td><td>1
0</td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>1
1</td><td></td><td></td><td></td><td>1
0</td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td>1
1</td><td></td><td></td><td></td><td>1
0</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></td><td></td><td></td><td>1
1</td><td></td><td></td><td></td><td></td><td>1
0</td></tr>
</table>

Figure 7. A file for which Heuristic 1 performs badly. Note that this file can be extended to any number of records. All values not specified are 0.

1 1 1 1					1 1 0 0	1 0 1 0							
	1 1 1 1						1 1 0 0	1 0 1 0					
		1 1 1 1							1 1 0 0	1 0 1 0			
			⋱								⋱		
				1 1 1 1								1 1 0 0	1 0 1 0

Figure 8. A worst case file for Splitting and Greedy Heuristics. All values not shown are 0. Note that the same shape file can be used for any r a multiple of 4. Other values of r are not needed, but a suitable modification can be made for the odd records.

Figure 9. One possible O-trie for the file shown in Figure 3.