Compressing Tries for Storing Dictionaries

Titus D. M. Purdin

University of Arizona
Management Information Systems Department

Abstract

This work describes a technique for compressing static dictionaries using *trie* structures. Our goal is to achieve reasonable compression of such dictionaries while preserving the ability to travel both up and down in the resulting trie. We investigate the trade-offs necessary to accomplish this; and conclude with a discussion of the utility of trie structures thus constructed.

Overview

It is quite easy to convert a *static dictionary* of words or phrases into a tree structure. Furthermore, we can conspire to construct such a tree so that it can be traversed forward and backward—forward from the root and backward from any leaf. Such trees can be associated in various ways with one another at the leaf level to construct some very interesting and complex structures. If the trees are built so that each node has but one child pointer (i.e. a *trie*) we avail ourselves of several additional possibilities; namely, the ability to compress the resulting structure and the ability to traverse the trie in an upward direction while storing far fewer pointers.

In this paper we explore, in detail, an approach to compressing static dictionaries using trie structures. The limitation of the range of input to static dictionaries allows the application of much more restrictive compression techniques than are possible in the case of textual input[3]. Some of these compression techniques (e.g. suffix elimination), however, make it impossible to travel upward in the trie. Where it is possible to travel upward in a compressed trie, multiple tries can be associated at the leaf level to form more complex structures. This allows the construction of translators, thesauri, and other dependent relationships.

We look first at the relationship between dictionaries and tries; then at the specifics of compressing such tries. The utility of suffix elimination is discussed specifically because of its impact on traversing tries in a upward direction. Finally, we demonstrate the general utility of associating tries.

Constructing Tries

Static dictionaries consist of a list of unique words or phrases in no particular order. While it does happen to be in alphabetic order, the UNIX dictionary associated with the *spell* and *look* commands is a good example. Like that dictionary, static dictionaries do not allow the dynamic addition or deletion of entries in the dictionary.

For the purposes of exposition we make certain other constraining assumptions here. It is assumed that words in the dictionary are composed of the 26 lower case alphabetic characters (i.e. no spaces or punctuation marks). The alphabet is, however, extended to include a distinguished *end-of-word* character. With the exception of the restriction to a static dictionary, all of these assumptions are merely simplifying in nature. The techniques are applicable to alphabets of arbitrary complexity.

A static dictionary is converted to a tree structure in which each node represents the set of characters available in the subject alphabet. An entry in a node is *filled* if it represents the value of a character in a word in the input dictionary on a path from the root of the tree, taking the characters from the input word one-at-a-time. Other node entries are *empty*.

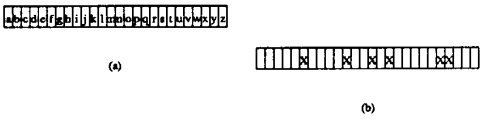

Figure 1

Figure 1.a shows a simplified tree node with lower case alphabetic characters holding their representative places. Figure 1.b shows a simplified tree node with X's indicating filled entries. A filled entry, of course, contains a pointer to its subordinate node. The fixed size of the input alphabet makes it possible to represent letter values by location within the node. That individual letters are not nodes in themselves corrupts somewhat the "pure" notion of a trie, but serves as a shorthand for the 26 nodes and sibling pointers necessary to realize a "real" trie. This can be thought of as an initial compression step.

That the *tree* described thusfar is in fact a *trie* can be seen in Figure 2. Each node in the tree represents a "level" in the trie, and each entry in a node represents a "node" in the trie. Each of these trie "nodes" has only one child pointer. It points to the subordinate "level" of the trie. The resulting data structure allows the lookup of a given word in time proportional to the length of the word by virtue of the underlying trie construct.

As presented, the trie structure does not contain sufficient information to allow it to be traversed from leaf to root. To correct this deficiency we prepend a single "backpointer" to each node. This pointer identifies the common ancestor for all of the letters represented in a particular node. To be specific, what is needed is actually two pieces of information: the location of the parent node and the identity of the relevant entry in that node. For convenience we combine these into a single entity.

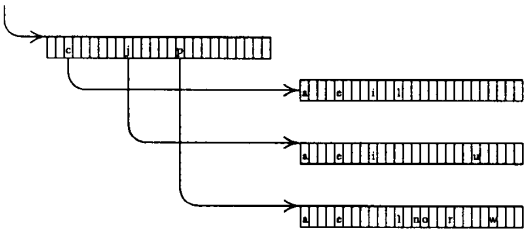

Figure 2

Combining Tries Into Complex Structures

To see the utility of combining tries of the sort described above, imagine a trie constructed from a dictionary of English words and a second trie constructed from Modern Greek words. If we associate the two tries at the leaf level appropriately, it is possible to traverse the English trie from the root, tracing the word *b-r-e-a-d*, for example. The terminal *d*, is linked to a leaf in the Modern Greek trie. Moving to it and traversing that trie from leaf to root, we obtain the elements ι-μ-o-σ-π. Taken in reverse to be the word $\pi\sigma o\mu\iota$. This process is, of course, reversible as depicted in Figure 3.

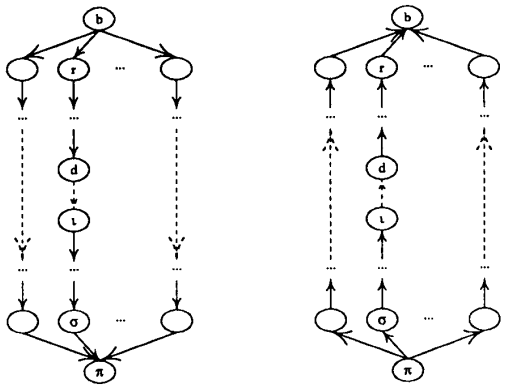

Figure 3

In the example above, the leaves of two tries are associated in a one-to-one fashion. It is easy to imagine each leaf of one English word trie being associated in a one-to-many arrangement with multiple leaves of a second English word trie to form a thesaurus. If we extend the notion of "leaf" in this thesaurus example, it is plausible to think of each such "leaf" in a trie being associated with a special "clearing-house" node in such a way that a single English word trie would suffice to implement our thesaurus using a many-to-many connection scheme.

We can find no specific references to such structures in the literature; but find it highly unlikely that such useful structures have not occurred in many practical applications. Our present focus is on techniques that permit these structures to be stored in minimum space.

Compressing the Trie

The fairly specific nature of the input and desired output allows us to concentrate on a number of particular space saving ideas. This includes some very general approaches, such as prefix folding and node compaction. It also includes some more specific approaches, such as positional (bitmap) representation of nodes, bitmap compression, node entry compression, and node entry ordering. Each of these is discussed individually below.

The most important accolade for these techniques, taken together, is that they preserve the capability of traveling both forward and backward in the trie structure.

They do, however, impose the restriction that the compression process be a static rather than a dynamic process.

Prefix Folding and Node Compaction

The combining of common prefixes in the input into a single instance, called prefix folding here, is obtained automatically as a result of using a trie (or tree) structure to represent the input. Every word in the input that begins with a *b* uses the *b* entry found in the root node. The words *bend* and *bent* share their first three letters. A root to leaf traversal for these two words will encounter exactly the same first three nodes. The final node (pointed to by the appropriate *n*) will contain (at least) entries for *d* and *t*. This provides a substantial savings in space.

It seems plausible that an equally great savings can be obtained through the folding of suffixes. And it has been demonstrated that suffix folding can be accomplished in linear time [2]. Unfortunately, it makes any backward traversal in the trie impossible. This makes it interesting in terms of seeing just how much compression can be achieved, but useless in the present circumstances. Space savings of $\approx$ 50% have been obtained incorporating suffix folding.

As described above, our basic trie encoding includes all of the letters in a particular "level" of a trie (i.e. all letters with a common parent letter) in a single node. This avails us two things. It allows us to use child pointers to represent the letters in the current node positionally and it allows us to associate a single "backpointer" with each node, based on the common parentage of such a node. As a bonus, it contributes considerably to making the resulting structure smaller.

Node Encoding and Specific Compressions

The nodes in the tries we are constructing become sparse very quickly as one descends in the trie. This sparseness is an important source of space savings. The only question is how best to take advantage of it.

Several approaches are available reduce the empty space in the nodes of the trie. [0], for example, suggests a scheme in which individual nodes are aligned at the left edge with their parent and then shifted right until the filled locations in the child node do not overlap any filled locations in the parent. The child node is then "merged" with the parent node. The result is a structure that grows constantly to the right, while attempting to use as little "new" space as possible.

This approach requires additional information in each node location to associate that location with a particular node. It also places an arbitrary limit on the size of relative pointers and, consequently, presumes a certain amount of sparseness. It can be easily demonstrated that for certain node configurations no interleaving pattern exists that yields acceptable pointer values. Consider a dictionary of nonsense words in which every letter of the alphabet can and does follow every letter of the alphabet. This results in nodes with no empty entries and, consequently, no sparseness of which to take advantage. A complete set of such words of length three will defeat the single byte relative pointer scheme.

We have implemented a bitmap approach for the representation of each node. Using this technique, it is possible to eliminate entirely the unused alphabet locations within the node. The corresponding cost is a bitmap associated with each node to preserve information concerning which letters are contained in the node.

A node consists of its bitmap header and <= 26 pointers to subordinate nodes. Our 26 letter alphabet allows us to use 4 bytes for the bitmap and, let us suppose, 4 bytes for each pointer. The resulting node format is pictured in Figure 4 (the backpointer for the node has been omitted).

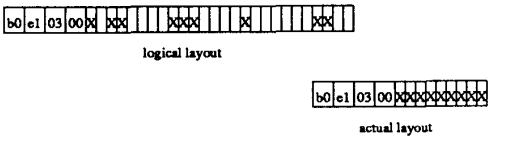

Figure 4

The act of converting the nodes of the trie into such a format amounts to converting it into linear structure that mimics the memory representation of the trie. This linear structure allows conversion of memory address pointers to "relative" pointers, which will necessarily be much smaller.

The addition of a four byte bitmap to the beginning of each node appears, at first glance, to be rather wasteful in terms of space. Also the generality of four byte relative pointers is achieved at considerable apparent expense in terms of space. To avoid the overhead imposed by these two two choices we employ a *second-level* compression scheme. Both the bitmap and the associated pointers are compressed; however, in very different ways.

In the case of the bitmap it is possible for much of the four bytes available to be unused. Consider the case of an extremely sparsely populated node, say one containing only the character 'a'. We will eliminate all of the sparseness among the pointers as described above, but just one bit of the 32 available bits in the bitmap will be turned on. There will be three all-zero

bytes. To reduce this overhead we omit such all-zero bytes. The first three bits of the bitmap are used to indicate which of the four bytes are actually present. Any that are not present are assumed to contain only zeros. Only three bits are required, because the first byte of the bitmap contains these *indicator* bits and will *always* be present.

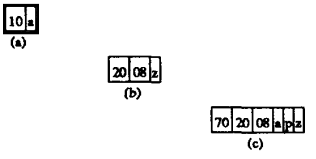

Figure 5

Figure 5 shows the results of this bitmap compression applied to a node containing only an 'a', a node containing only a 'z', and a node containing the letters 'a', 'p', and 'z'. Recall that the use of the bitmap produces a net savings over the original coding scheme in those cases where there are more than four active entries in a node. This bitmap compression is most effective when it is applied to very sparse nodes, which is precisely where the heaviest overhead occurs.

For the node entries themselves a similar compression approach is used. These relative pointers can require anywhere from one to four bytes. Although experience shows that values requiring more than two bytes are unlikely. Three specific bits are used for bitmap compression because arbitrary combinations of four bytes are possible. For the pointers, however, we need to indicate only how many bytes are being used—not which ones. For this reason we are able to use just one bit.

The first bit of each byte in a pointer is used as a "continuation" bit to indicate that the following byte is also part of the pointer. This approach allows seven bits of each byte to be used for the relative address value. More importantly, it allows the largest possible value (127) to fit in the first byte. The goal, after all, is to minimize the instances in which additional bytes are required.

As an example, consider a pointer that indicates a relative distance of 59. This value fits comfortably in the seven available bits and the resulting pointer entry, shown in Figure 6.a, consists of a single byte with a high order bit of zero. A relative pointer value of 213 cannot be contained in the bits available in a single byte. In this case the value is spread between the seven bits of the first byte and the seven of a second byte. The high order bit of the first byte is one and that of the second byte zero. Figure 6.b shows the details of this case.

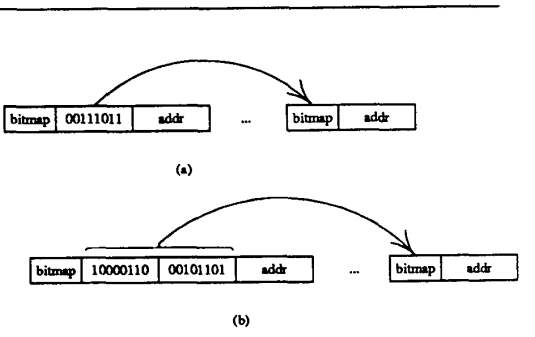

Figure 6

These compression techniques produce an appreciable savings in the space required to store the contents of a static dictionary. Except for pathological cases the resulting structure is smaller not only than the plain character input file, but smaller than the result of node merging,which is described above, as well. The actual savings realized are, of course, extremely input dependent. For a typical UNIX *spell* dictionary the space reduction is 25% to 28% over that required for a plain character file.

Entity Ordering and Node Ordering

This encoding scheme performs well in terms of eliminating unused entries from a node. It is clear, though, that it is very sensitive to the order in which letters (entries) actually occur in a node. The bitmaps, in particular, show this bias.

As noted, every bitmap includes at least one byte, the first byte, because it contains the map of the bitmap itself. Whenever a letter at the "end" of the alphabet is recorded the bitmap must include its fourth byte as well (see Figure 5.a and 5.b for an example). This suggests that some sort of *etaonirs* (the descending order of frequency with which letters of the English alphabet occur in *average* English text) ordering can be effectively applied.

As our algorithm makes two passes, one to construct the tries and one to convert them to a linear format, it is a simple aside to compute the actual frequency of letters in a specific input during the first pass and then apply that ordering to the alphabet as nodes are converted in the second pass. The effective result of this is to minimize the size of the most frequently occurring bitmap configurations.

It has also been suggested by Durre (see [0]) in a private communication that the underlying trie structure can be balanced to effectively minimize the size of relative pointers needed. The linear structure produced in the second pass grows only to the right, resulting in constantly increasing relative distances from parent node entries. One can imagine building the linear structure with the root node in the "middle" and alternate entries to the right and then to the left. We have not yet implemented such a balancing scheme.

Special Cases

The initial encoding of the input dictionary as a trie uses a special "end-of-word" character to allow the representation of words that are completely subsumed by other, longer words. For example, the words *be* and *bee* occur within the prefix of the word *been*. In the third node encountered in tracing *been*, in addition to the entry for *e* that points to the node containing the *n*, there will be an entry for the "end-of-word" character. Its presence indicates that the path *be* is terminal. This "end-of-word character would also be present in the node containing the *n* in *been*.

This demonstrates the necessity for inclusion of such a special character, but it overlooks the wasteful condition in which the *n* entry for the word *been* points to a child node that contains only an entry for the "end-of-word" character. This condition can be detected with a simple counter in the first pass and eliminated by producing a null pointer in the second pass.

Results

We have implemented a small (500 word) translator and a small (800 word) thesaurus. The thesaurus was built both in the trie-to-trie fashion and in the trie-to-itself fashion. The latter uses the special nodes, called "clearing house" nodes mentioned earlier.

The representation of a trie structure representing the input dictionary, using allocated memory and pointers, is considered a strictly internal representation. It is of no utility in representing the trie in an externally useful form. It is for this reason that the trie is converted to a linear form during a second pass. The various compressions (with the exception of prefix folding) are applied during the second pass.

Furthermore, comparisons between the internally represented trie and the resulting linear trie are meaningless. This leaves us with only the measure of the size of the input file and the size of the compressed output file. But these can no longer be compared directly, because we have added functionality to the output by structuring it as a trie and by providing the information necessary to travel upward (leaf-to-root) in the trie. Our compressions are carefully applied to preserve this added functionality.

Nonetheless, as a measure of the compression that is achieved, we have observed the following. A 20,000 word English dictionary was converted to a linear trie structure and stored in 28.7% less space than the input file. A linear trie structure to which none of these compactions is applied (except prefix folding, which is automatic), for the same dictionary, produces a file that is 136.5% the size of the input file. We estimate that the file would be approximately 150% the size of the input file if prefix folding were not present.

These figures allow us to compare, at least, the various compressions being applied to the input to each another. Using this measure, we can say that individual compression techniques have the following approximate relative values:

node compaction	28.3%
bitmap compression	9.4%
address compression	17.8%
elim "eow" nodes	4.1%
alphabet ordering	5.6%

Conclusions

The interesting structures that can be constructed by juxtaposing tries (or trees) is in no way limited by efforts to compress the structures—as long as certain information is retained by the structure. The overall act of compression, then, is a trade-off between the amount of compression that can be achieved and the amount of information that must be preserved. We believe that the techniques described here result in a very reasonable savings in storage while preserving the interesting character of the trie storage structure.

The two small demonstration systems that were built in conjunction with these explorations have proven quite interesting in their own right. We intend to expand them and to investigate the class of tools that might be built or built better out of such structures.

References

[0]Durre, Karl P., Storing Static Tries. Proceedings of the 10th International Workshop on Graphtheoretical Concepts in Computer Science, Universitätsverlag Rudolf Trauner, Linz, Austria, June 1984, 725–737.

[1]Fiala, Edward R. and Greene, Daniel H., Data Compression with Finite Windows. *Communications of the ACM* Vol 32, No 4, April 1989, 490–505.

[2]McCreight, E. M. A Space-economical Suffix Tree Construction Algorithm *Journal of the ACM* Vol 23, No 2, 1976, 262–271.

[3]Ziv, J. and Lempel, A. A Universal Algorithm for Sequential Data Compression *IEEE Transactions on Information Theory* Vol 23, No 3, 1977, 337–343.