Tables

J. Ian Munro *
Department of Computer Science
University of Waterloo
Waterloo, ON
Canada N2L 3G1

Abstract. In representing a subset of a large finite set, or an index for text search, one is faced with the need for both time and space efficiency. In this paper, we look at some approaches that have been applied to these problems to represent objects in near minimum space and still permit queries to be performed in constant time. It is hoped that this paper will draw attention to techniques for representing large (mostly static) structures.

1 Introduction

Computer memories are growing dramatically, but problem size is growing even more quickly. As a consequence, space efficient representations of large objects are even more important than they were in the past. For example, given a large text file, we would like to preprocess the file and produce an index, so that given an arbitrary phrase one can quickly discover every occurrence of the query phrase in the text file. One natural approach is to use a search trie, say a Patricia tree [10] as a suffix trie [15, 5, 9]. The leaves of the tree are pointers to locations in the text and one descends the trie, branching according to the bit pattern of the input phrase. At a leaf one has discovered the unique location in the text in which (the prefix of) the query phrase occurs. If the phrase search stops higher in the tree, then an entire subtree indicates all occurrences of the phrase. Such a suffix trie approach permits searches in time bounded by the length of the input query, and in that sense is independent of the size of the database. However, assuming our phrase queries start at the beginning of any word and that words of text on average 5 or 6 characters in length, we have an index about 3 times the size of the text. That the index contains a reference to each word of the text accounts for less than a third of this overhead. Most of the index cost is in storing its tree structure. Indeed this is the main reason for the proposal [5, 9] of simply storing an array of references to positions in the text rather than the valuable structure of the tree.

In this paper we briefly review some techniques for representing (primarily static) information so that one can navigate through the structure efficiently. The general goal is to take as little space as possible and still perform basic operations

* imunro@uwaterloo.ca This work was supported by the Natural Science and Engineering Research Council of Canada.

in constant time. We will focus on two problems. The first is the representation of binary and ordered rooted trees. The second is the representation of moderately sparse sets. We view these relatively direct static structures as tables. While some of the higher level structures have variable sized fields, and so deviate from the most intuitive notion of a table, others are precisely simple tables of answers to all possible queries over the given domain. The goal of this paper is to point to a useful, though certainly not new, approach by way of a couple of examples. It is neither a survey of applications, nor is it a detailed treatment of the specific problems discussed.

Our model of computation is a random access machine in which each word consists of $\lg n$ bits. Here n can be the number of nodes in a tree or the size of the universe from which a set is constructed. Putting it another way, to identify an arbitrary basic object takes a word. On the other hand, we are not constrained to use representations of that sort as a word could be used as a bit vector to identify the presence or absence of $\lg n$ previously specified objects.

2 Representing Trees

In terms of clean representation of a tree, the heap of Williams[16] is the epitome of elegance. The structure is implicit [11, 12] in that only the size of the object and the records associated with particular nodes is retained. The structure represented as a heap is a balanced binary tree in which all leaves on the bottom level are shifted to the left. The nodes are numbered level by level from left to right. Hence the root is node 1, and the left and right children of node i are $2i$ and $2i+1$ respectively. We can therefore navigate from parent to child and back through the tree by simple calculation of the index. It is also straightforward to calculate the size of a subtree rooted at node i, given only i and n. Insertion or deletion of the next/last node on the bottom now is trivial. As a consequence the structure is an absolute delight for the data type priority queue.

All this comes, in large measure, because there is only one tree on n nodes. In the case of arbitrary binary trees, we have the nth Catalan number, $C_n = \binom{2n}{n}/(n+1)$ or about 4^n, to distinguish among. In supporting a specific abstract data type, say dictionary, we can go to implicit representations of AVL trees, using the relative values of keys to encode pointer information [11]. This obviously stretches the limits of practicality in time space tradeoffs. It is also outside our realm of interest here, as we are concerned primarily with the representation of a tree without any assumption regarding the relative values in the nodes. As a consequence, just to say what tree we are dealing with requires at least $\lg C_n$ or close to $2n$ bits.

Jacobson [6, 7] proposed an encoding based on the construction of Zaks' sequences [17] and the work of others including Lee et al [8] to map the nodes of a tree onto the integers $1..n$. The idea is essentially the same as that of the heap ordering, appropriately extended to handle arbitrary binary trees. Each node is mapped to its position in the level ordering of the (basic) tree. To indicate the structure of the tree we extend the n node tree to one of $2n+1$ nodes

by appending leaves in all positions in which a child is missing in the original tree. We now consider the level ordering of the nodes in this extended tree and represent the tree by the $2n+1$ bit pattern with a 1 for an internal (original) node and a 0 for each newly appended leaf. If we apply this procedure to a tree with the heap shape, we have the encoding 1^n0^{n+1}. The navigation in the general case is surprisingly like that of a standard heap, with a little help from the bit encoding. We need, however, a couple of operations on bit vectors:

- $rank(x)$ = the number of 1's up to and including position x
  and
- $select(x)$ = the position of the xth 1.

($xrank0$ and $select0$ can be defined analogously to deal with the 0's, and will be of use in encoding more general trees.)

Now we see that children and parents of node x can be found as

- $leftchild(x) = 2\ rank(x)$
- $rightchild(x) = 2\ rank(x) + 1$
- $parent(x) = select(\lfloor x/2 \rfloor)$.

2.1 Representing Ordered Rooted Trees

By an ordered rooted tree we mean a tree with a distinguished root and an ordering on the children of any node. As is well known, the added feature of a node having an arbitrary number of children precisely balances with the loss of the feature that a single child is not distinguished as left or right. These two classes of trees on n nodes are isomorphic, mapping a left child to a first child and a right child to a next sibling. Unfortunately, this mapping is not convenient for finding the ith child or its parent. The binary tree representation suggested above would infer i rank/select operations for the task. Instead Jacobson proposed an encoding due to Read [13], again based on level orderings. For convenience a super-root with one child, the root of the given tree, is added. We represent a node with k children by k 1's and a 0. Concatenating these node representations by level order of the tree, we see the ith 1 is the indication of node i's "birth" by its parent, while the ith 0 indicates its "death", having indicated all its children. Now we have

- $degree(x) = select0(rank(x) + 1) - select0(rank(x)) - 1$
- $child(x) = select0(rank(x)) + i$
- $parent(x) = select(rank0(x))$

2.2 Implementing Rank and Select

Jacobson's goal was to perform basic tree navigation by inspecting $O(\lg n)$ bits per step. As a consequence, the rank operation part of his construction translates to $O(1)$ time under a $\lg n$ bit word model. Jacobson's select, however, takes $\Theta(\lg \lg n)$ time as a consequence of a binary search over $(\lg n)^2$ bits, and so

requires reworking. Clark and the author [2, 3] provide the enhancement. The implementation of both operations involve our major technical theme, the use of subtables. Jacobson's rank algorithm uses two subtables of about $n \lg\lg n / \lg n$ extra bits each. The first suffices to give the rank of each $(\lg n)^2$rd bit in the vector. The second uses $\lg\lg n$ bits to record the rank of each $(\lg n)$th bit within these subranges. This reduces to ranges of size $\lg n$, and so, we can finish off the process either by a careful coding or by using tables of size $O(n^{1/2})$ to record all possible answers on half-words. Handling the select operation in constant time is a bit more subtle, but again is based on a subtable approach. Summarizing these results:

> A binary tree on n nodes, or an ordered rooted tree on n nodes, can be represented in $2n+o(n)$ bits so that each node is mapped to a distinct integer location in the range 1 to n, and that given the location associated with a node, one can determine to location of its parent, the number of children it has, and the location of any specified child in constant time.

The author's interest in compact representations of trees arose from the problem of finding compact encodings of suffix tries. The method presented here leads to a near optimal representation and constant time navigation. From an implementation point of view, it has the drawback of complication. For the particular application it has another difficulty, namely that when one does a search for a phrase, a successful search gives a subtree corresponding to all matches. If a phrase has large number of matches, one may first like to know how many there are. Hence the size of the subtree at a given node is a very useful operation. It would not appear that the representation we have outlined lends itself to this. As a consequence of both these reasons, Clark [2] chose to implement a different encoding, also explored by Jacobson, although it does not support the operation of finding a parent directly. Using this approach we store each subtree by encoding

- *small*: a flag to denote which subtree is smaller
- *subsize*: the size of the smaller subtree using some prefix encoding
- *Leftsubtree*: recursive encoding of the left subtree, of it has more than 1 node
- *Rightsubtree*: recursive encoding of the right subtree, of it has more than 1 node.

In interpreting such an encoding it is absolutely imperative that the space taken for any tree of a given size be the same, and that this size be easily determined. This is crucial, for example, in determining where the encoding of a right subtree begins. The interesting part, however, is the representation of *subsize*. First note that if we were simply to encode the size of the left subtree, then the degenerate tree that runs to the left would take $\Theta(n \lg n)$ bits. The flag *small* eliminates this problem. However, *subsize* must be represented in some manner so that small values are encoded very succinctly, and the code grows slowly with the value of *subsize*. Given that it will not be known how many bits

are in this encoding, it seems well advised to use a prefix code, one in which the code for a value cannot be the prefix of the code for another. Using the simple approach of storing the subsize k with $\lg k$ zeroes followed by the number in binary (the latter must start with a 1), the entire tree can be represented in about $3n$ bits. One is tempted to try encoding $\lg k$ itself in binary and ultimately getting to the well known game of trying to guess a number, upon which we have no a priori upper bound, in the smallest number of binary questions. The optimal solution (for large k) requires $\sum \lg^{(i)} k$ guesses or bits in the problem at hand (Here $\lg^{(i)}$ denotes the lg function iterated i times, e.g. $lg^{(3)}k = \lg\lg\lg k$). Unfortunately this does not help. The problem is in encoding small values well. Suppose we do something silly, like take 3 bits to encode the value zero, then with the *small* flag we take 4 bits to say a tree of size n has a left subtree of size $n-1$, leading to a $4n$ encoding of the tree. With care in the encoding, particularly of the small values, we can however, drop the $3n$ bit encoding to about $2.75n$ bits. Perhaps this is not enough to justify the complication, but it gives a feel for the limits to the gains. This issue of careful encoding of small numbers occurs in a number of problems. One interesting example is in trying to tighten the $O(n)$ term in the Schaffer-Sedgewick [14] average case analysis of Floyd's version [4] of heapsort.

3 Representing Subsets of a Bounded Universe

The other problem noted in the introduction is that of representing a simple subset of a finite universe, k elements from a universe of size n. As we noted, a bit vector is ideal if about half the elements are present. Fully encoding each element in $\lg n$ bits is near optimal for a sparse table and indeed perfect hashing techniques can be tuned to achieve both constant time and almost minimal space. The ground in between was explored by Brodnik and the author [1]. The approach taken is to use the two obvious methods when the set to be stored is, respectively, large or small. When k is in the range $n^{1-\epsilon}$ to $n/\lg n$ or so, elements are mapped to $k/\lg n$ buckets corresponding to $\lg n$ consecutive values in the universe. One quickly considers recursing on the universe of size $\lg n$, and one would appear to be off to a $\lg^* n$ solution by recursing until the problem size is down to a constant. However, after just two iterations the entire universe is down to $\lg n$ elements. At this point any tight encoding of the subset to be represented can be used as an index into one fixed and very small table. As a consequence we get a solution in constant time and essentially the minimal space of $\lg \binom{n}{k}$ bits.

4 Conclusion

In this brief overview we have discussed two examples of representing static structures succinctly while still permitting the basic search and navigation operations to be performed in constant time. Such problems and some of the approaches

suggested for their solutions may well be of increasing importance as the size of the problems we attack continues to grow. While the focus has been on asymptotically large problems, the essence of the methods can be implemented to show real benefit on problems of even moderate size. On the other hand, problem sizes no longer seem moderate. For years I had told my students that $\lg \lg n$ was 5, as one found it hard to imagine storing a set of more than 4 billion (U.S.) elements. Well, we passed that threshold a few of years ago. $\lg \lg n$ is now 6, I don't expect to see 7.

References

1. A. Brodnik and J. I. Munro, *Membership in Constant Time and Minimum Space*, Proc. Algorithms - ESA '94, LNCS 855 (1994) 72-81.
2. D. R. Clark, Compact Pat Trees, manuscript, University of Waterloo (1996).
3. D. R. Clark and J. I. Munro, Succinct Representation of Trees, in preparation, University of Waterloo (1996).
4. R. W. Floyd, *Algorithm 245: Treesort 3*, Communications of the ACM, **7** (1964) 701.
5. G. H. Gonnet, R. A. Baeza-Yates and T. Snider, Lexicographic Indices for Text: Inverted Files vs. Pat Trees, Tech. Rpt. OED-91-01, Centre for the New OED, University of Waterloo (1991).
6. G. Jacobson, *Space Efficient Static Trees and Graphs*, Proc. 30th IEEE Symp. on Foundations of Computer Science, (1989) 549-554.
7. G. Jacobson, Succinct Data Structures, Tech. Rpt. CMU-CS-89-112, Carnegie Mellon University (1989).
8. C. C. Lee, D. T. Lee and C. K. Wong, *Generating Binary Trees of Bounded Height*, Acta Informatica, **23** (1986) 529-544.
9. U. Manber and G. Myers, *Suffix Arrays: A New Method for On-Line String Searches*, SIAM Journal on Computing, **22**(5) (1993) 935-948.
10. D. R. Morrison, *Patricia - Practical Algorithm to Retrieve Information Coded in Alphnumeric*, Journal of the ACM, **15**(4) (1968) 514-534.
11. J. I. Munro, *An Implicit Data Structure Supporting Insertion, Deletion, and Search in $O(log^2 n)$ Time*, J. Computer and System Sciences, **33**, (1986) 66-74.
12. J. I. Munro and H. Suwanda, *Implicit Data Structures for Fast Search and Update*, J. Computer and System Sciences, **21**, (1980) 236-250.
13. R. C. Read, *The Coding of Various Kinds of Unlabelled Trees*, In R. C. Read (ed.) Graph Theory and Computing, Academic Press, (1972) 153-182.
14. R. Schaffer and R. Sedgewick, *The Analysis of Heapsort*, Journal of Algorithms, **15**(1), (1993) 76-100.
15. P. Weiner, *Linear Pattern Match Algorithms*, Proc. 14th IEEE Symp. on Switching and Automata Theory, (1973) 1-11.
16. J. W. J. Williams, *Algorithm 232, Heapsort*, Communications of the ACM, **7** (1964) 347-348.
17. S. Zaks, *Lexicographic Generation of Ordered Trees*, Theoretical Computer Science, **10**(1), (1980) 63-82.