Space-efficient Static Trees and Graphs

Guy Jacobson

School of Computer Science
Carnegie Mellon University

Abstract: New data structures are developed that represent static unlabeled trees and planar graphs. These structures are more space-efficient than conventional pointer-based representations, but (to within a constant factor) they are just as time-efficient for traversal operations. For trees, the data structures described here are *asymptotically optimal*: there is no other structure that encodes n-node trees with fewer bits per node, as n grows without bound. For planar graphs (and for all graphs of bounded pagenumber), the data structure described uses *linear space*: it is within a constant factor of the most succinct representation.

1 Introduction

Linked data structures often use machine addresses (pointers) to represent the linking relation that exists between nodes. While this provides for rapid traversal, and is a great convenience when the structure is dynamic, it is sometimes wasteful of space.

One reason for this lack of economy is the "fatness" of pointers. For a pointer to have the potential to address n different locations, it must be at least $\lceil \lg n \rceil$ bits wide. A structure with $O(n)$ pointers will therefore occupy $O(n \log n)$ bits in memory. For some classes of linked structures, this is more bits than necessary to distinguish among members of the class, even with any constant fraction of waste.

As an example of such a class, consider unlabeled binary trees. The number of n-node trees is $\frac{1}{n+1}\binom{2n}{n}$. The number of bits needed to differentiate the n-node trees is the logarithm of this quantity, which (by Stirling's Approximation) is $2n + o(n)$. A pointer representation for binary trees would need $O(n \log n)$ bits, but $O(n)$ bits suffice informationally.

One need not look far to find a simple two-bit-per-node representation for binary trees. A simple recursive scheme will work:

$$\text{rep}(T) \;=\; \mathbf{0}\ \text{rep}(\texttt{left-child of } T)\ \mathbf{1}\ \text{rep}(\texttt{right-child of } T)$$

The representation function rep exploits the one-to-one correspondence between binary trees of n nodes and balanced strings of $2n$ parentheses (with **0** and **1** representing open and close parenthesis, respectively). This scheme is *asymptotically optimal*, in the sense that the ratio of the space actually used to the informational lower bound tends to one as n grows without bound. In other words, the fraction of wasted space vanishes.

However, this scheme does not allow efficient tree-traversal. Locating the `right-child` of a node requires a linear scan through the bits to "balance parentheses," taking $O(n)$ time in the worst case. The common pointer representation for binary trees is much faster, if less space-efficient. How can the time-efficiency of pointers be achieved in asymptotically optimal space? That question is answered in this paper.

1.1 Related work

The efficiency of the representations presented here in both time *and* space distinguishes this work from that of other authors who merely seek succinct encodings (R. C. Read [6] gives a good summary of efficient tree-encodings, and Turán [8] gives a linear-space encoding for planar graphs).

Turán's encoding stores a planar graph of n nodes in $12n$ bits. His encoding uses linear space, but it does not allow efficient searching. Kannan *et al.* [5] show how to represent planar graphs implicitly, to allow efficient adjacency testing. Their method makes use of the bounded arboricity of planar graphs. They decompose a planar graph into (at most) three edge-disjoint spanning trees (using a famous theorem of Nash-Williams), and then represent each tree separately. Although they still need $O(n \log n)$ bits for the whole graph and they cannot search efficiently, the beauty of their data structure lies in its implicitness: the graph is fully described by the set of its node indices.

1.2 Metrics for space and time

Let C_n be a of static objects (with natural size parameter n), and let S be a set of query operations that examine a member of C_n. An *implementation* is a way of mapping the elements of C_n into a read-only memory, along with program for each operation in S.

I use the following metrics (functions of the size parameter n) to measure the performance of an implementation:

> **space** will be measured in bits. Simply count the maximum number of bits in the read-only memory.
>
> **time** will be measured in bit-accesses into the read-only memory. This is a cell-probe metric where cells can hold only a single bit.

As a concrete example, let C_n be the class of n-node binary trees, with operation set

$$S = \{\texttt{left-child}, \texttt{right-child}, \texttt{null}\}$$

A pointer representation of C_n has space-complexity $O(n \log n)$, but has a time-complexity of $O(\log n)$, the number of bits in a

CH2806-8/89/0000/0549/$01.00 © 1989 IEEE

single pointer. The **0/1** parenthesis representation presented earlier has space complexity $2n$, but the worst-case time complexity is $O(n)$.

1.3 Organization of this paper

First, I will outline the design of data structure to represent subsets of $1 \ldots n$ efficiently, supporting the operations `rank` and `select`. This will be used as a tool in the linked structures. The two unlabeled tree representation (binary and general) are presented, along with traversal algorithms.

To implement planar graphs in linear space, another tool is needed. This tool, a linear-space parenthesis matcher, is then described. This is used to construct a linear-space representation for k-page graphs that allows efficient traversal and adjacency testing.

2 Ranking and selection

Let C_n be the class of subsets of $1 \ldots n$. If the operation set S consists only of membership testing, it is trivial to build a data structure that is simultaneously optimal in time and space; a simple bit-map will do.

What if a richer set of operations is desired? Two very useful operations on a subset S of $1 \ldots n$ are:

`rank(`m`)` Counts the number of elements in S less than or equal to m.

`select(`m`)` Finds the mth smallest element in S.

These operations are inverses of each other, in the sense that `rank(select(`m`))` = m, for $1 \leq m \leq ||S||$, and `select(rank(`m`))` = m, for $m \in S$. Rank and select can, of course, be performed directly when a bit-map implementation is used, but that would be very inefficient. In general, a linear scan through the bits is required to rank and select, so the worst-case cast of these operations is $O(n)$.

One way to add the operations of ranking and selection to a bit-map implementation of a set data type is to augment the bit-map with an auxiliary structure called a *directory*. This data structure will help make these additional operations efficient. This idea goes back to Elias [3], who used a similar structure to provide good average-case performance for ranking and selecting in multisets.

The scheme used to implement directories for ranking and selection is too complicated to be included here. A detailed description can be found in Jacobson [4]. It uses two-level tables of indices, similar to a data structure described by Tarjan and Yao [7] for storing static sparse sets. The extra space required for my directories is $o(n)$ bits (more precisely, $\Theta(n \log\log n / \log n)$, so the total space for the bit-map and the directory is asymptotic to n bits. The time complexity of `rank` and `select` operations is $O(\log n)$, measured in bit-accesses.

3 Trees in asymptotically optimal space

Now I describe a method, employing rank/select directories, that achieves the asymptotic optimum of two bits per node.

First, consider binary trees.

3.1 Level-order binary marked

When a binary tree is very balanced, it can be represented implicitly by addresses in an array. The root is given the address 1. A node whose address is m has a left child with address $2m$ and a right child with address $2m + 1$. This scheme is an efficient way to represent heaps (see Aho [1, page 87]), since there is no need for explicit pointers. Trees with imperfect balance can also be represented in this way by using these implicit addresses to index an array of bits saying which nodes are present in the tree and which are not. This implicit-bit-map representation of binary trees is shown in figure 1. This representation makes

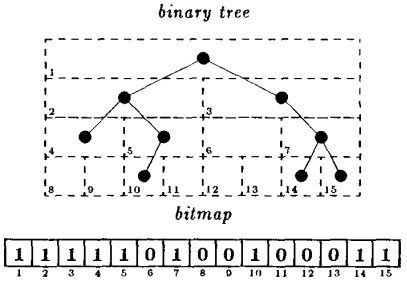

Figure 1: A binary tree and its implicit bit-map.

traversal very cheap, but it has an obvious drawback: unless the tree is extremely well balanced, the number of bits needed will be huge. If the deepest node is at depth d, the space occupied lies between 2^d and $2^{d+1} - 1$ bits.

A modification of this idea can be used to generate a more succinct representation of a binary tree as a string of bits as follows:

1. Mark all the nodes of the tree with **1** bits.
2. Add external nodes to the tree, and mark them all with **0** bits.
3. Read off the bits marking the nodes of the tree in (left-to-right) level-order.

This construction, shown in figure 2, makes it easy to see that the original tree can be reconstructed from the string of bits formed. Each such bit string is therefore associated with a unique tree.

How many bits are there in these level-order mark bit strings? There are n **1** bits (the internal nodes) and $n + 1$ **0**'s, for a total of $2n + 1$ bits. Traversing a tree represented in this way is easy, using the ranking tools developed earlier.

Let an internal node m be represented by the index of where its **1** bit appeared in the level-order mark bit string. Consider the bit string as the bit-map of the set of indices of the (internal) nodes. Now build a ranking directory for this set. Each **1** bit on level d corresponds to a node with two children (some of which may be external nodes) on level $d + 1$, and these two children will correspond to two adjacent bits in the part of the string where the level $d + 1$ nodes appear. Also, left-to-right ordering is maintained from one level to the next: If two nodes, a and b, are on the same level, and a's **1** bit is to the left of b's, then the adjacent pair of bits corresponding to the children of a will occur before b's pair in the string.

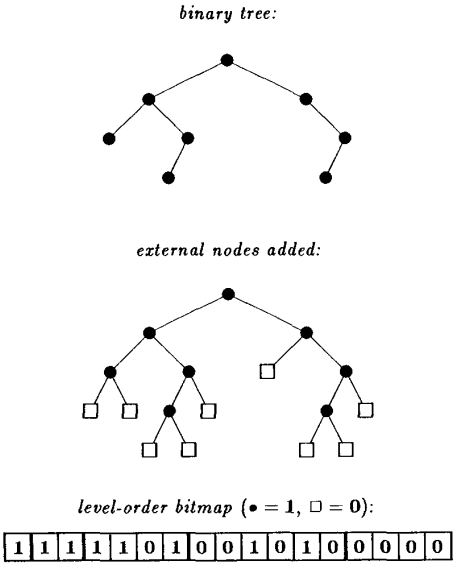

Figure 2: Level-order binary marked representation.

This leads to a very simple algorithm to compute `left-child(`m`)` and `right-child(`m`)`, for (internal) node m.

```
left-child(m) ← 2 · rank(m)
right-child(m) ← 2 · rank(m) + 1
```

Note the strong similarity to the implicit addressing scheme discussed earlier. The value of `null(`m`)` is true exactly when the mth bit of the string is a **0**, since this indicates an external node. The root node has index **1**.

The string itself occupies $2n+1$ bits, and the ranking directory occupies $o(n)$ bits, so the total space required is $2n + o(n)$. This is asymptotically optimal linear space. The tree-traversal operations do a single rank, so they require time $O(\log n)$ time, measured in bit-accesses.

Keeping a selection directory allows efficient computation of `parent(`m`)` too. This is because

$$\texttt{parent}(m) = \texttt{select}(\lfloor m/2 \rfloor)$$

Now, I turn my attention to unlabeled general rooted trees with ordered children. These are equinumerous with binary trees with the same number of nodes, so the optimal lower bound of 2 bits per node applies here as well. I use both ranking and selection, together with another $2n$ bit string scheme (again based on level-order) to represent such trees.

3.2 Level-order unary degree sequence

A rooted, ordered tree can be represented by reporting its degree sequence in any of a number of standard orderings of the nodes. Consider the degree sequence of a tree, ordering the nodes in the left-to-right level order employed in the previous section. This sequence of n integers uniquely identifies the tree. Now encode these integers using a simple binary prefix code R (the "unary" code):

$$\begin{aligned} R(0) &= \mathbf{0} \\ R(k>0) &= \mathbf{1} \cdot R(k-1) \end{aligned} \tag{1}$$

The integer d is represented by the string $\mathbf{1}^d\mathbf{0}$. Take the sequence of degrees encoded in binary and simply concatenate them together to form a bit string. Since the codes are prefix codes, the unique tree associated with a string is easily found.

The number of **1** bits in this bit string is n. Every node except the root is a child of another node, so the number of **0** bits is $n-1$. The total length of the string is thus $2n-1$ bits. Each node is associated with exactly one **0** bit and (except for the root) one **1** bit. To maintain the "one **1** per node" property, add a fake super-root node to the top of the tree, whose only child is the root. Now each node has a unique **1** bit associated with it, and the string is only two bits longer.

This bit-string scheme has much in common with the level-order marked binary scheme described in the previous section. A depiction of a tree and its level-order unary degree sequence is in figure 3.

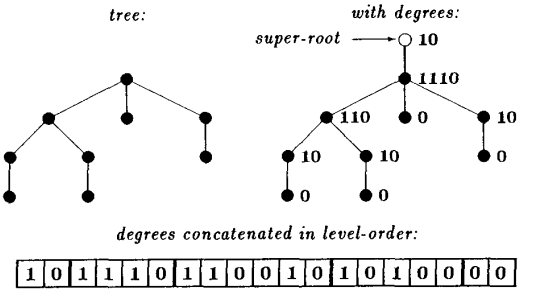

Figure 3: Level-order unary degree sequence representation.

Represent a node m by the index of its corresponding **1** bit in the string, as in the previous section. Also, build ranking and selection directories for the bit-string *and* its bitwise complement. This will permit efficient selection of the mth element *not* present in the set (the mth **0** bit). Use the notation `rank0` and `select0` to refer to the operations of ranking and selection in the complement of the bit-string.

With this representation, `null(`m`)` can be tested, as before, by inspecting the mth bit of the bit-map. The operation `next-sibling(`m`)` is simply an increment of m. This representation also allows access to previous siblings, access to children by number, and counting of children. The basic traversal operations are performed as follows:

```
first-child(m) ← select0(rank(m)) + 1
next-sibling(m) ← m + 1
parent(m) ← select(rank0(m))
```

4 Planar graphs in linear space

Turán's 12-bit-per-node representation shows that the space required to store a planar graph is linear in the number of nodes. Kannan *et al.* [5] represent planar graphs by decomposing them

into three edge-disjoint spanning trees. I will now show how decomposing a planar graph into *pages*, rather than spanning trees, permits a linear-space representation with efficient traversal and adjacency testing. Heavy use will be made of ranking and selection. I will also need one other tool: a parenthesis balancer.

4.1 Parentheses balancing

Given a static balanced string of n parentheses, I will build, in space linear in n, a directory that makes the following operation efficient:

> Find the position in the string of the close (open) parenthesis that matches the open (close) parenthesis in position p.

Obviously it suffices to solve the restricted problem of finding close parentheses that match open ones, because we can build a backwards directory to find the open parentheses that match close ones.

First, break the string of parentheses into blocks of length $\log n$, and let these blocks be numbered starting with 1. A few simple definitions:

Definition 1 *A parenthesis p is called a* far *parenthesis iff p's matching parenthesis lies outside its own block.*

Definition 2 *A far parenthesis is a* pioneer *iff its matching parenthesis lies in a different block that that of the previous far parenthesis in the string.*

Keep the set of pioneer parentheses (as a bitmap) along with a ranking directory for that set. Also, store a table of block numbers recording where the matches for the pioneer parenthesis lie.

If p is not a far parenthesis, its match (call it q) is easily located by linear local search with its own block. Otherwise, find out which block contains q as follows: compute the rank of p in the set of pioneer parentheses, and use the result as an index into the table of block numbers. To find where exactly in this block q is, it is necessary to keep another table on hand: the *nesting depth* of the start every block. (This is just the tally of excess *open* over *close* parentheses in a prefix of the string.) The nesting depth of p itself can be computed by from the depth at the beginning of its block and the parentheses in that block before p (local search). This depth, together with the depth at the beginning the block b_q where q lies, allows location of q by a linear scan through the parentheses in b_q.

This structure, pictured in figure 4, finds matching parentheses in $O(\log n)$ time. The table of nesting depths, the set (bitmap) of pioneer parenthesis, and its ranking directory add up to only $2n + o(n)$ extra bits. But what about the table of matching block numbers for the pioneer parentheses? If there were more than $O(n/\log n)$ pioneers, this table could grow to be more than linear. Any individual block can contribute $\lg n$ pointers into the table. The following theorem shows that the number of pioneers can never be too large:

Theorem 1 *The number of pioneer parentheses in a balanced string divided into b blocks is at most $2b - 3$.*

Proof: Imagine a graph with a node for each block of the string. Lay these nodes out on the plane in a straight line in order of the blocks. Create an edge between two nodes if there is a far parenthesis in one that points into the other. The number of edges in such a graph is at least as great as the number of pioneers in the string, since every pioneer can be mapped to a different edge. But since this string of parentheses is balanced, none of the edges can cross, and the graph is outer-planar (that is, it can be embedded in the plane with all the vertices on a single face). Therefore the number of edges is at most $2b - 3$, by a well-known property of outer-planar graphs. □

Since each pointer in the table is $\lg n$ bits wide, this means the entire table is at most $[2(n/\lg n) - 3] \cdot \lg n = 2n + o(n)$ bits.

4.2 Bounded pagenumber graphs

Instead of describing how to represent planar graphs, I will show how to represent a larger class of graphs of which planar graphs are a subclass. This larger class is the class of *bounded pagenumber graphs*. These are the graphs that have k-page book embeddings, where k is a parameter of the class. First, define the term *book embedding*, following Bernhart and Kainen [2].

A *k-page book embedding* of a graph $G = \langle V, E \rangle$ is a *printing order* of V (a permutation specifying the ordering of the nodes along the spine of a book), plus a partition of E into k *pages*. The edges on a given page must not intersect, and all pages share the same printing order of the nodes.

The *pagenumber* (or *book-thickness*) of a graph G is the minimum number of pages in any book embedding of G. Let $\mathcal{G}_k$ be the class of all graphs whose pagenumber is bounded by k. Given a particular graph $G \in \mathcal{G}_k$, and a k-page embedding of G, I will show how to visit neighbors and test adjacency in G. For G with n nodes, only $O(\log n)$ bit-inspections are performed per operation, using a representation of G with total number of bits linear in n (for fixed k). The number of bits used is actually $O(kn)$.

For simplicity, I will first exhibit a linear-space representation of one-page graphs (these are exactly the outer-planar graphs) and then generalize.

4.2.1 One-page graphs in linear space

The edges on a given page of the graph all lie to one side of the nodes (which are on the spine) and may not cross. When the "book" is drawn so that the spine is horizontal (as shown in figure 5), observe that the nesting structure of the edges is just that of a balanced string of parentheses. Start with a string

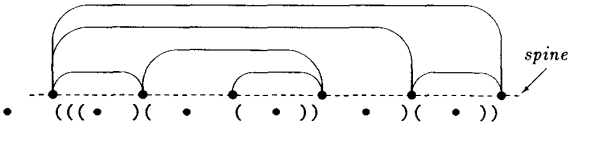

Figure 5: One-page graphs as balanced parentheses

of n node symbols each, denoted by •. For each edge $\langle u, v \rangle$ on the page, insert a (just before the $(u + 1)$st node symbol and a) just after the vth node symbol. (Note that the parentheses in the string remains balanced after each such insertion.) The final result is a string over a 3 symbol alphabet, containing n node symbols and at most $2n - 3$ each open and close parentheses. The parentheses between the mth and the $m + 1$st node symbol correspond to the set of edges out of node m. Now, encode

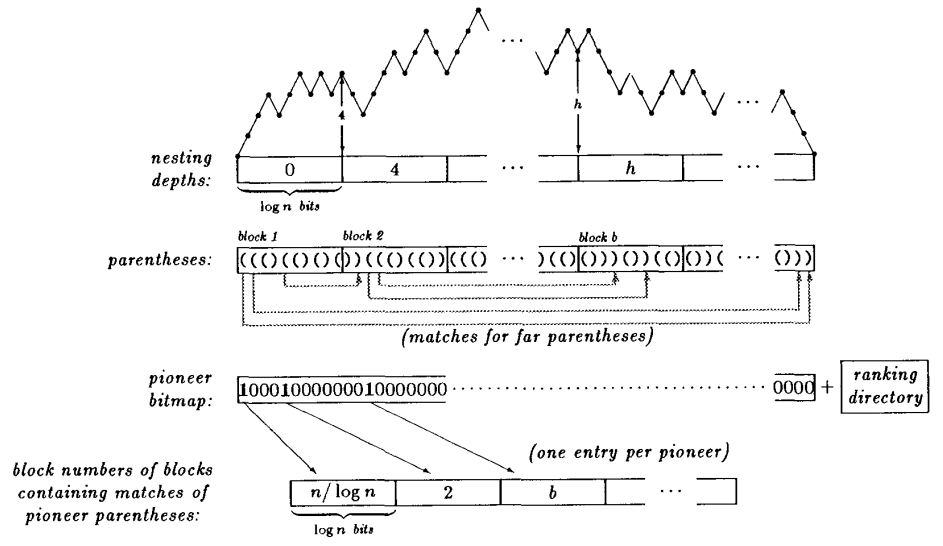

Figure 4: A structure to balance parentheses.

this 3-symbol string into a pair of strings of bits. First, record the sets of positions occupied by node symbols as a bit-map. Term this the *node map*. Next, delete the node symbols and record the remaining parenthesis string in binary. These two strings allow reconstruction of the original graph, and use at most $[n + 2(2n-3)] + [2(2n-3)] = 9n - 12 = O(n)$ bits.

These two bit strings are at the heart of the linear space representation of G. Additionally, construct the following tools:

- A matcher for the parentheses string.
- A rank/select directory for the node-map and its complement.

These tools will require extra space, but the total storage used will still be $O(n)$.

The natural numbering provided by the printing order are the indices of the nodes. Indices into the string of parentheses will serve as edge indices.

4.2.2 Searching and adjacency testing

With these structures, searching around in G is little more than matching parentheses. Each edge in G is associated with a pair of matching parentheses. To follow an edge, given the index of one of its associated parentheses, simply find the matching parenthesis. First, define two simple macros: `node-to-edge` converts a node number into the index of the first edge out of that node, and `edge-to-node` takes an edge index into the parenthesis string and finds the number of the node containing that edge:

```
node-to-edge(m)  =  rank0(select(m) + 1)
edge-to-node(e)  =  rank(select0(e))
```

The algorithm to visit neighbors of a node m is:

```
e ← node-to-edge(m)
while edge-to-node(e) = m
      e' ← paren-match(e)
      visit edge-to-node(e'))
      e ← e + 1
```

This algorithm performs only a constant number of rank, select, and parenthesis matching operations between the nodes it produces. Therefore it uses only $O(\log n)$ bit-inspections per neighbor visited.

Another useful operation is adjacency testing: is there an edge between node u and node v? This operation can also be performed using only $O(\log n)$ bit-inspections; see Jacobson [4] for details.

4.2.3 Graphs of more than one page

In the previous section, I showed how to represent a one-page graph of n nodes in $O(n)$ bits allowing searching and adjacency testing in $O(\log n)$ time. The generalization to multi-page graphs is direct. If graph G is a k-page graph, represent each of its pages (these are one-page graphs) separately. To produce all the neighbors of a particular node m, go through each of the k pages in turn, using the above algorithm to visit neighbors on that page. To test two nodes for adjacency, simply resolve the question for each page.

Therefore any $G \in \mathcal{G}_k$ (the class of k-page graphs) can be represented in $O(kn)$ bits, allowing searching and adjacency testing in $O(k \log n)$ time. For a sub-class of graphs with bounded pagenumber, this becomes $O(n)$ space and $O(\log n)$ time. Yannakakis [9] gives a linear-time algorithm that embeds any planar graph in four pages. Since I have shown the linear-space result for any class of graphs with bounded pagenumber, it follows for planar graphs as well.

References

[1] AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[2] BERNHART, F., AND KAINEN, P. C. "The book thickness of a graph." *Journal of Combinatorial Theory B* **27**:320–331, 1979.

[3] ELIAS, P. "Efficient Storage and Retrieval by Content and Address of Static Files." *Journal of the ACM* **21**(2):246–260, 1974.

[4] JACOBSON, G. *Succinct Static Data Structures.* Ph.D. Thesis, Carnegie Mellon University, 1988.

[5] KANNAN, S., NAOR, M., AND RUDICH, S. "Implicit Representations of Graphs." *Proceedings of the 20th ACM Symposium on Theory of Computing*, pages 334–343, 1988.

[6] READ, R. C. "The coding of various kinds of unlabeled trees." in *Graph Theory and Computing*, pages 153–182. Academic Press, New York, NY, 1972.

[7] TARJAN, R. E., AND YAO, A. C. "Storing a Sparse Table." *Communications of the ACM* **22**(11):606–611, 1979.

[8] TURÁN, G. "Succinct Representations of Graphs." *Discrete Applied Math* **8**:289–294, 1984.

[9] YANNAKAKIS, M. "Four pages are necessary and sufficient for planar graphs." *Proceedings of the 18th ACM Symposium on Theory of Computing*, pages 104–108, 1986.