Succinct Data Structures

Craig Dillabaugh

October 9, 2007

1 Introduction

The paper reviewed here by Guy Jacobson [2] describes space efficient representations of static trees and graphs that nevertheless allow for efficient operations such as accessing the child of a node in a tree or determining the neighbours of a node in a graph. In particular we will examine space efficient (succinct) structures for binary trees, general trees, outer-planar graphs, and general planar graphs.

2 Problem Statement

Data structures for trees and graphs often use pointers to represent links (edges) between vertices or nodes. This representation is very efficient in terms of allowing rapid traversal of these data structures but it is not space optimal since each pointer must address one of n memory locations and thus requires $\log n$ bits. In total we need $O(n \log n)$ bits to store the n pointers in a binary tree in comparison to the theoretical bound of $2n + o(n) = O(n)$ bits to represent such trees. The goal of this research is to represent data structures using approximately the theoretical bound on space occupancy while at the same time supporting efficient $O(\log n)$ bit accesses traversal operations such as finding the parent or child of a node.

3 Techniques

3.1 Rank and Select

Operations in the data structures that follow rely on efficiently performing the operations $Rank_1(m)$, $Rank_0(m)$, $Select_1(m)$, and $Select_0(m)$ on a bit vector B. The subscript value 0 or 1 simply indicates whether the operation is

performed on the 1's or 0's in the vector. Here we will describe the operations in terms of the '1' values as the same operations on zero values are roughly analogous.

$Rank_1(m)$ returns the number of '1's in B up to the position $B[i]$. $Select_1(m)$ returns the index in B of the $i'th$ 1. Consider for example the bit array B in Fig.1. If indexing starts from zero then $Rank_1(5) = 4$ since up to position $B[5]$ we have a total of four 1's in B. $Select_1(3) = 4$ since to select the first three 1's we must go up to index $B[4]$.

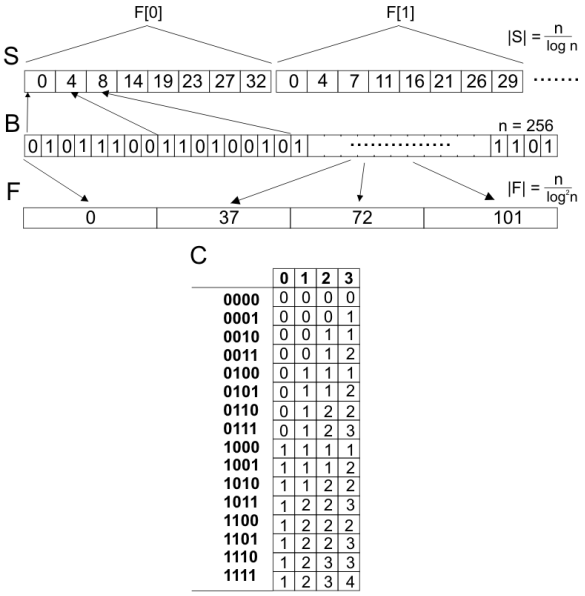
Both queries can be answered by scanning B but this requires $O(n)$ time which is very poor. Jacobsen's paper mentions that both operations can be accomplished in $O(\log n)$ bit access which corresponds to $O(1)$ operations in the standard RAM model, however the paper omits the details. The description that follows, which details how $Rank_1(m)$ is implemented is based on [1]. The Select operation can be performed using a similar technique also achieving the $O(1)$ bound, but the description of this method too lengthy to be included here.

The data structures, or *directories*, required to provide the constant time operations are shown in Fig. 1. First we conceptually divide B into *blocks* of size $(\log n)^2$ and store in the array F a value for each block that corresponds to the rank of the last element in the previous block. The array F must store $\frac{n}{(\log n)^2}$ elements requiring up to $\log n$ bits for a total storage requirement of $\frac{n}{(\log n)^2} \cdot \log n = \frac{n}{\log n}$.

Next we subdivide B conceptually into *sub-blocks* of size $\log n$. Each group of $\log n$ sub-blocks corresponds to a single block. In a manner similar to that for the *blocks* we store in each element of the *sub-block* array S the rank of the last element in the previous *sub-block*, however counting is restarted at each *block* boundary. Thus *sub-blocks* indicate ranks of positions within the *blocks*. While the array S stores a total of $\frac{n}{logn}$ items, each item must record a value of size at most $\log n$ and thus requires $\log \log n$ bits, giving S a total space requirement of $\frac{n \log \log n}{logn}$.

Finally the array C is created. This array stores for all bit vectors of size $\frac{\log n}{2}$ the rank at each position. As there are only $\sqrt{n}$ bit vectors of length $\frac{\log n}{2}$ all possible rankings can be computed and stored in a $\sqrt{n} \times \frac{\log n}{2}$ array. The total number of bits required for this data structure is $\sqrt{n} \cdot \log n \cdot \log \log n$.

With these three auxilliary data structures the query $Rank_1(m)$ can be solved as follows. We simply look up the the values for the *block* and *sub-block* to which m belongs in the F and S arrays respectively, adding the *sub-block* starting value to that of the *block* gives the cumulative rank up to the start of the *sub-block*. Recall that *sub-blocks* are of length $\log n$ so



	0	1	2	3
0000	0	0	0	0
0001	0	0	0	1
0010	0	0	1	1
0011	0	0	1	2
0100	0	1	1	1
0101	0	1	1	2
0110	0	1	2	2
0111	0	1	2	3
1000	1	1	1	1
1001	1	1	1	2
1010	1	1	2	2
1011	1	2	2	3
1100	1	2	2	2
1101	1	2	2	3
1110	1	2	3	3
1111	1	2	3	4

Figure 1: A bit vector B and associated data structures required to answer Rank in $O(1)$ time.

the *sub-block* is divided into two bit-vectors of length $\frac{\log n}{2}$. These values are used directly as row indicies for the array C and we can thus determine with just two more look ups (at most) the rank within the *sub-block* of m. In total we require four look-ups and the additions to calculate the final rank and as such $Rank_1(m)$ is performed in constant time.

3.2 Parentheses Balancing

Given a string, $|S| = n$ of balanced parentheses we wish to implement the function $Find(p)$, which given p, the position of an open or close parenthesis in S, returns the position of q, the close or open parenthesis that matches p. In the description that follows assume p is an open parenthesis and q is a close parenthesis, searching in the other direction is exactly the inverse operation and is implemented in the same manner.

Obviously such queries could be answered trivially if we allowed $O(n \log n)$ bits of storage, as we could simply store for each p a pointer to q. However the data structure and operations described here will support $Find(p)$ efficiently with just $O(n)$ bits of storage.

As was done for rank and select, S is split into $b = \frac{n}{\log n}$ blocks of size

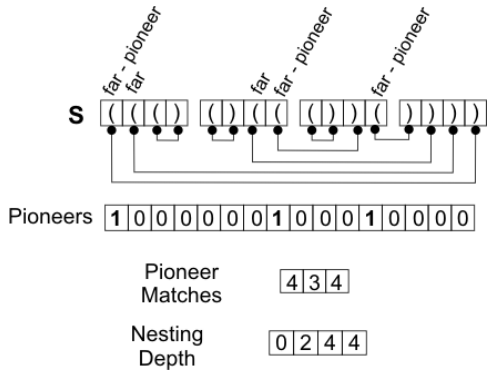


Figure 2: A string S of parentheses divided into blocks of size $\log n (= 4)$ and with matching parentheses indicated.

$\log n$. Next we will define two special types of parenthesis. A *far* open parenthesis has its matching close parenthesis lying in another block. A *pioneer* parenthesis is a *far* parenthesis that has its matching parenthesis lying in a different block than that of the previous far parenthesis in S (see Fig. 2).

In order to support efficient matching we create the following additional tables (also depicted in Fig. 2). First we create a *Pioneer* bitmap which records the position of each pioneer in S, a directory is also built on this structure to support efficient rank and select. The second structure employed is the *Pioneer Matches* table which stores for each *pioneer* parenthesis the block of its matching parenthesis. The third structure stores for each *block* the *nesting depth* of that block where *nesting depth* is the excess of open over close parenthesis in S up to the start of the block.

To find the matching parenthesis q we can perform a linear scan on B if p is not a *far* parenthesis as this will require at most $\log n$ bit accesses. If p is a *far* parenthesis then finding q involves the following steps:

1. Compute $Rank_1(p)$ in the Pioneers bitmap.

2. Use the result of $Rank_1(p)$ as the index into the Pioneer Matches table to determine the block containing p's match.

3. Use the Nesting Depth table to determine the nesting depth at the start of the blocks containing p (b_p) and q (b_q) respectively. From these we can use linear scan to determine the nesting depth of p in b_p and then scan b_q to find the bracket with matching nesting depth, which is q.

Finally, lets consider the space required by the various structures used in the bracket matching process. The bitmap of pioneers requires n bits while the associated ranking directory requires $o(n)$ bits. For the nesting depth array we have $\frac{n}{\log n}$ elements which store a value of size $\log n$ so this array requires n bits. For the Pioneer Matches array we store values of size at most $\log n$ bits. Theorem 1 shows that the length of the Pioneer Matches array does not exceed $2b-3$, so we have in total $(2(\frac{n}{\log n})-3)\cdot \log n = 2n + o(n)$ bits.

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

Proof: Consider each block required to store the string S as if it is a node in a graph G. Arrange the vertices of G such that they lie on a straight line, l in the plane. Now for each pair of nodes in G connect the nodes with an edge if there is one or more *far* parentheses that form a match in their corresponding blocks. Draw these edges in the plane above l. Since the edges represent matching brackets they can be drawn so that they do not intersect. Furthermore since the edges are drawn above l it is clear that each node in G lies on the outer face of G and thus G is outer-planar. The number of edges in an outer-planar graph is at most $2n-3$ (where n is the number of vertices) and as such G has at most $2b-3$ edges. Finally, as the number of *pioneer* parenthesis cannot exceed the number of *far* parenthesis the total number of pioneer parenthesis is bounded by this quantity.

4 Succinct Trees

4.1 Level-Order Binary Marked Tree

Binary trees can be encoded as bit strings using the following simple scheme as depecited in Fig. 3.

1. Label each node in the tree with the value 1.
2. Add external nodes to the internal node so that each internal node has two children. Label these external nodes with the value 0.
3. Traverse the tree from left to right in level order and read the labels to the bit array.

This representation requires n 1 bits and $n+1$ 0 bits for a total of $2n+1$ bits. Each 1 bit in the bit-vector B corresponds to a node in the tree. For the node at $B[m]$ we can use the Rank and Select operations to locate its children and parent as follows:

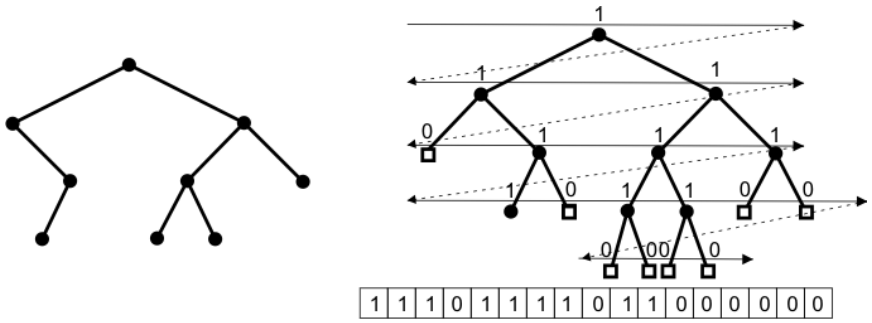


Figure 3: A binary tree (top). On the left hand side the same tree is shown with external nodes (hallow squares) added and binary representations indicated. Arrows indicate the order in which the nodes are visited in producing the binary representation shown at the bottom.

$left - child(m) \leftarrow 2 \cdot Rank_1(m)$
$right - child(m) \leftarrow 2 \cdot Rank_1(m) + 1$
$parent(m) \leftarrow Select_1(\lfloor m/2 \rfloor)$

By using the directory structures described in Section 3.1 we can implement each of these operations in $O(\log n)$ bit access or $O(1)$ time. This adds $o(n)$ bits so we can store B and its *directory* in $2n + o(n)$ bits.

4.2 Level-Order Marked Degree Sequence Tree

For a general binary tree where nodes may have zero, one or more children a slightly different binary encoding is used for each node. We again use level-order left to right traversal of the tree to build up the bit string but the node encoding is a *prefix code* that records a 1 for every child followed by a single 0 bit. Thus a node with three children is encoded as '1110' while a node with no children is encoded as '0', see Fig. 4 for examples of node encodings and the resulting bit string.

The resulting traversal operations now become somewhat more complicated, but can still be accomplished using Rank and Select as with the binary tree.

$first - child(m) \leftarrow 2 \cdot Select_0(Rank_1(m)) + 1$
$next - sibling(m) \leftarrow m + 1$
$parent(m) \leftarrow Select_1(Rank_0(m))$

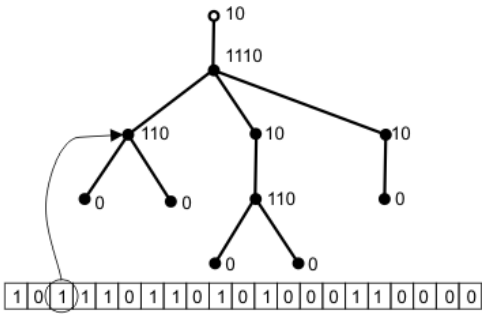


Figure 4: A general tree is shown with its level-order unary degree sequence indicated for each node and the corresponding bit vector shown at the bottom. The root node (hallow) is the *super-root* which is added to ensure that the binary representation stores exactly one 1 per node. Note that the 1 bit corresponding to a node in the bit vector is encoded by the node's parent. Thus the 2nd 1 bit is part of the encoding for root node, but corresponds to the root's first child as indicated by the arrow.

For the degree sequence tree described above we have n 1 bits and $n+1$ zero bits if we include the *super-root* for a total of $2n+1$ bits. Again we can support all traversal operations in $\log n$ bit operations since they use rank and select by adding the $o(n)$ directory structures.

5 Succinct Graphs

Consider the class of graphs known as the *bounded pagenumber graphs*, which is the class of graphs which permit *k page book-embeddings*. Given the graph $G(V, E)$ a single page embedding is an ordering, T, of all $v \in V$ along a straight line in 2d such a subset of the edges E (which are curved) can be drawn above the line without intersecting. A *book embedding* divides the set of edges E into k pages such that given the same node ordering T page can be embedded without any edge crossings. The *pagenumber* of a graph is the minimum number of pages that the graph can be embedded in. For arbitrary graphs finding the *pagenumber* is NP-Complete [3].

5.1 Succinct Single Page Graphs

We will first consider the takes where $k = 1$ and G can be embedded on a single page. Such a graph is outer-planar. Fig. 5 shows the outer-planar graph G and the set of structures used in its succinct representation. First

we see the arrangement of the nodes of G for the page embedding of G. Note that with this representation the nesting of edges corresponds to a string of balanced parentheses. We generate a set of balanced parentheses for the nodes of G using a 3 symbol alphabet starting with a $\bullet$ for each node. Then for each edge $(u, v) \in G$ we add a open parenthesis '(' immediately before node $u + 1$ and a close parenthesis immediately after node v. The resulting string has at most n node symbols and $2n - 3$ brackets.

Next the *node map* is generated. This is a bit vector with a 1 bit corresponding to each node and a 0 bit corresponding to each parenthesis (open or close) in the string of balanced parentheses, and finally we remove the node symbols ($\bullet$) from the string of parentheses and create a bit vector for the parentheses. The *node map* and parenthesis bit-vector used $O(n)$ space, including the directories to enable efficient rank and select and parenthesis matching.

To perform searches on G we define two basic operations both of which are performed on the *node map*. The first of these is *node-to-edge* operation which converts a node number into the index of the first edge out of that node. *Node-to-edge* is calculated as; $node{-}to{-}edge(m) = Rank_0(Select_1(m)+1)$. The second operation is the *edge-to-node* which given the position of one of the edge parentheses in the parenthesis string returns the number of the node adjacent to that edge. *Edge-to-node* is calculated as: $edge - to - node(e) = Rank_1(Select_0(e))$.

Given these simple operations we can visit the neighbours of node m in constant $O(\log n)$ time with the following algorithm:

e $\leftarrow$ node-to-edge(m)
while edge-to-node(e) = m
e' $\leftarrow$ paren-match(e)
visit edge-to-node(e'))
e $\leftarrow$ e + 1

This algorithm performs a constant number of rank, select, and parenthesis matching operations, per neighbour, and can therefore operate in $O(\log n)$ time per neighbor visited.

5.2 Multipage Graphs

For graphs that cannot be embedded in a single page the graph is embedded in k pages, and each page is represented seperately. Operations described for

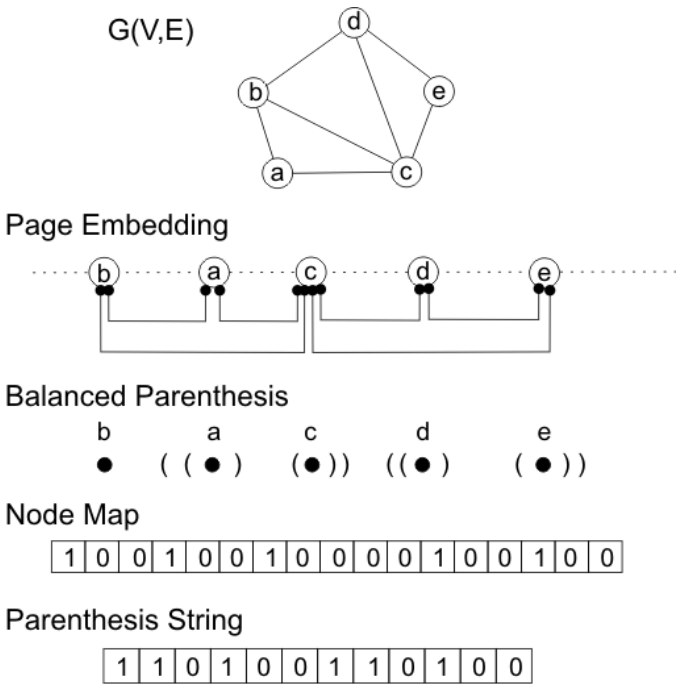


Figure 5: An outer-planar graph $G(V, E)$ and the corresponding structures used in generating and representing its succinct representation.

single page graphs must simply be executed for each of the k pages. Since single pages can be represented in $O(n)$ bits a k page graph requires $O(kn)$ bits. Searching and adjacency testing is performed in $k \log n$ bits.

6 Summary

Jacobsen's paper is was not the first attempt at representing common data structures such as graphs and trees in a succinct manner. However, the key innovation presented in the paper was the efficient traversal operations in such structures. Techniques such as rank and select and parenthesis matching form fundamental operations in current research into succinct data structures.

7 Questions and Answers

7.1 Questions

1. Given a outer-planar graph $G = (V, E)$ describe an algorithm that produces a valid page embedding of the graph.

2. Consider the parenthesis balancing routine outlined in the Jacobsen paper. The paper claims if we know the *nesting depth* of open parenthesis p and the block to which its matching close parenthesis q belongs, we can scan the block b_q to find a parenthesis of matching nesting depth and that this parenthesis will be q. Prove that this is indeed the case and that there cannot be some other parenthesis r in b_q which has the same nesting depth as p.

7.2 Answers

1. Start at any node v and perform a *depth first search* outputing each node the first time it is visited.

2. Proof is by contradiction. Assume there is some such parenthesis r. Since parenthesis are balanced then r which is a close parenthesis and must be matched by some open parenthesis (lets call it s) that appears somewhere in the string after p. The nesting depth of s must be greater than that of p or else p would have matched some parenthesis prior to s. However we have claimed that nesting depth $r = s = p$ which is a contradiction.

References

[1] Eppstein, D.: *Separating Thickness from Geometric Thickness*, Web page: [*http://www.ics.uci.edu/ eppstein/pubs/Epp-GD-02-slides.pdf*], 2002.

[2] Jacobson, E.: *Space-efficient Static Trees and Graphs*, *FOCS*, **42**, (1989), 549-554.

[3] Makinen, V. and Schurmann, KB: *Data Structure Compression*, Web page: [*http://gi.cebitec.uni-bielefeld.de/ teaching/2005summer/datacomp/ dcsummer05 partII.pdf*], 2005.