UO-CIS-TR-82/3

FAST ENUMERATION, RANKING, AND UNRANKING OF BINARY TREES

by

Andrzej Proskurowski and Ekaputra Laiman
Department of Computer and Information Science
University of Oregon, Eugene OR

ABSTRACT

A linear representation of binary trees proposed earlier is analyzed with respect to the complexity of ranking and unranking operations. Simple algorithms are given that perform these operations in time proportional to the order of the binary tree, provided that a table of values of certain integer function is precomputed. Also, the complexity of generating representations of all binary trees of given order is discussed. An algorithm is given which updates the representation of a tree to yield the representation of the next tree in a time averaging to a constant over all generated binary trees. The larger the order of the generated trees, the smaller the constant.

1. Introduction

Generation of random data, or all possible data for a given algorithm may be a way to test or analyze the algorithm for its correctness, or computational complexity. In the past few years the subject of generating combinatorial structures in general, and trees in particular has been fairly extensively treated in the literature [1,5,6,7]. The fact that trees are often enumerated in a certain order causes consideration of functions ranking trees according to this particular ordering. Also, the inverse function producing the tree which occupies a given place in the ordered list of all such trees would allow an easy random selection from the list [3]. Thus, the computational complexity of algorithms enumerating, ranking and unranking trees is of importance.

In this paper we deal with enumeration of binary trees with n nodes. A _binary tree_ is either empty or consists of a root node and two subtrees, left and right, which are binary trees. An _extended binary tree_ is either a single external node (a leaf), or consists of an (internal) root node and two extended binary trees as left and right subtrees. There is an obvious bijective correspondence between the set of all binary tree with n nodes and all extended binary trees with n internal nodes (and n+1 external nodes). An _ordering traversal_ of an (extended) binary tree visits every node of the tree exactly once thusly ordering the nodes of the tree. A _monotonic ordering traversal_ visits nodes of the root of a subtree before the nodes of its both subtrees, for all subtrees of the traversed tree.

A binary tree T with n nodes will be represented by a string of n 1's and n+1 Ø's reflecting a traversal of the corresponding extended binary tree T' according to some fixed monotonic traversal order. In this string, a 1 corresponds to visiting an internal node of T' and a Ø corresponds to visiting an external node of T' (a leaf). It has been shown in [4] that a recursive "leaf-expansion" algorithm applied iteratively to intermediate extended binary trees (with less than n internal nodes) in the order determined by a fixed monotonic traversal, generates exactly the class of all binary strings representing binary trees with n nodes. In the next section we present two simple algorithms, FIRST and NEXT, which will effectively generate this class of strings in time proportional to its size. Namely, FIRST produces the lexicographically smallest such string, and NEXT modifies the string representation t to produce the tree representation succeeding t (unless t is the lexicographically largest string). Analysis of performance of the algorithm NEXT will be based on the total time requirements of computing NEXT^{i} for all $i = 1,\ldots,C_{n-1}$, where $\mathrm{NEXT}^{0} =$ FIRST, $\mathrm{NEXT}^{i} = \mathrm{NEXT} \circ \mathrm{NEXT}^{i-1}$, and C_n is the n-th Catalan number (counting the binary trees with n nodes). The subsequent section gives an algorithm RANK(t) which computes the ordinal of a string t representing a tree with n nodes in the lexicographically ordered list of all such strings. Accordingly, $\mathrm{RANK}(\mathrm{NEXT}^{i}) = i + 1$. The algorithm uses values of an integer function which can be computed either following a recurrence relation, or from a closed form involving binomial coefficients. Assuming constant time availability of these values (through, for instance, a pre-computation process), RANK requires only a linear amount of time. The algorithm UNRANK(i) performing the inverse function produces the string representation of the i-th binary tree with n vertices (according to the lexicographical ordering of the corresponding strings). Thus, UNRANK(i+1) = NEXT^i for i = $1,\ldots,C_{n-1}$. The time complexity of UNRANK is linear with n, again assuming constant time access to the values of the integer function mentioned above.

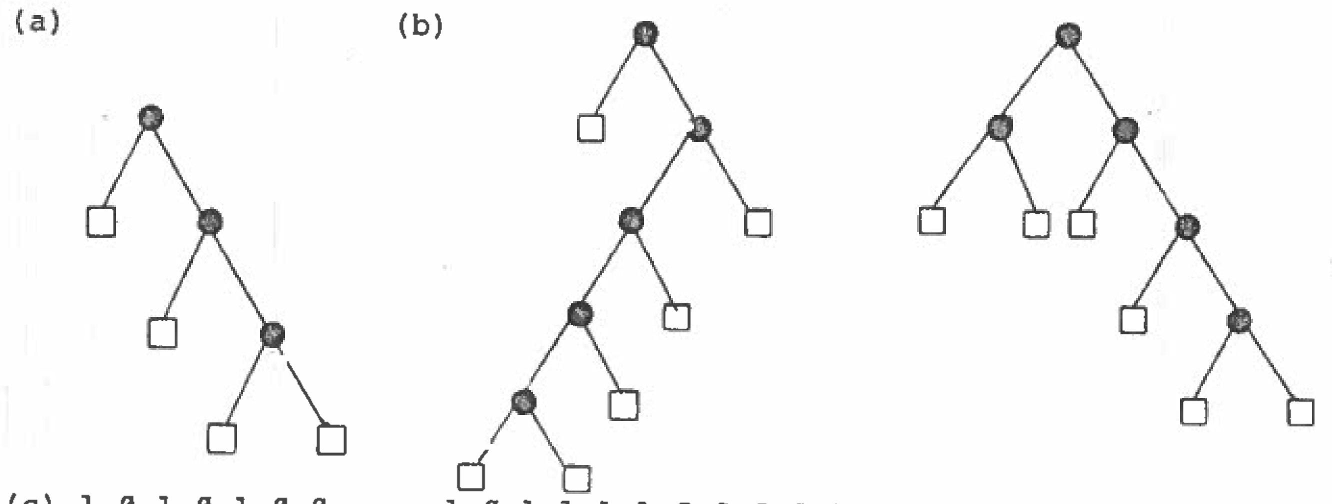
2. Enumeration: algorithms and analysis.

We will assume that a binary tree is represented by a global record consisting of an array <u>tree</u>[1..2n+1] and an integer value <u>last</u> pointing to the rightmost 1 in that array. The strings representing binary trees with n nodes will be generated in lexicographical order through initialization of the array <u>tree</u> (procedure FIRST) and the consecutive updating of some of the entries in the array (procedure NEXT). Although both procedures may spend as much as O(n) time in a single execution, the time complexity of updating the array <u>tree</u> averaged over all binary trees with n nodes will be shown to be bounded by a constant.

```
procedure FIRST;
  var i:integer;
  begin for i := 1 to n do
        begin tree[2i-1] := 1;
                tree[2i] := 0 end;
  tree[2n+1] := 0;
  last:=2n-1
  end; (* FIRST *)
```

Assuming the preorder traversal controlling the interpretation of strings representing binary trees, the first tree on 3 nodes is shown in Figure 1a. The computational complexity of FIRST is O(n).

In [2] the generation of strings representing binary trees with n nodes is defined recursively. Two consecutive (lexicographically ordered) strings may be obtained from the same string representing a binary tree with n-1 nodes by expanding into the substring "100" each of two consecutive zeroes in the block of rightmost zeroes. (This corresponds to expanding -- through the promotion to the internal node status -- the corresponding leaves of the extended binary tree.) The resulting strings differ only by the position of the rightmost 1's which are relatively shifted by one position. In the alternative situation, the two consecutive representations are obtained from two consecutive representations of trees with n-1 nodes, as the result of expanding the last zero in one, and the first zero (in the rightmost block) in the other string. Applying this description recursively, we find that any two consecutive strings differ in the suffix positions containing in the lexicographically preceding string the rightmost block of $j < n$ ones and the preceding zero. This suffix is replaced in the other string by a leading 1 followed by zeroes and the representation of the first binary tree on j-1 nodes. An example with n=5 and j=4 is given in Figure 1b.



(c) 1 Ø 1 Ø 1 Ø Ø   1 Ø 1 1 1 1 Ø Ø Ø Ø Ø   1 1 Ø Ø 1 Ø 1 Ø 1 Ø Ø

Figure 1 (a) The first tree with 3 nodes, (b) two consecutive trees on 5 nodes, and (c) their representations.

In the following, we describe a charging scheme for the cost of updating tree representation in the operation NEXT. The unit charge is the cost of setting two entries in the tree representation. When NEXT is applied to a representation t in which the rightmost block of 1's has width w ($1 \leq w < n$), then the following entries have to be set: the one immediately preceding that block (to 1), the w entries of the block itself (to Ø), and the w-1 entries in the suffix representing the first tree with w-1 nodes (to 1). The prefix of the tree representation and the other Ø's in its suffix remain unchanged. Thus, the total charges amount to w units (1 + w + w - 1 changes). These are w − 1 changes). These are apportioned to t itself and to its w-1 predecessors. Each of these predecessors incurs only one unit charge as arguments of NEXT, as the corresponding values of w are all 1. Indeed, the rightmost block of w 1's in t must be followed by at least w+1 Ø's to satisfy the feasibility of the representation. For each of the w-1 representations of trees immediately preceding t, an application of NEXT requires only a change of the rightmost 1. Thus, over all applications of NEXT to every tree representation once, no tree is charged more than two units.

Actually, the total charges for the trees with n nodes are less than $2C_n$. This follows from the fact that the rightmost block of w 1's in a tree representation may be, and frequently is, followed by more than w+1 Ø's. We conclude this section by an implementation of the procedure NEXT.

```
procedure NEXT;
  var i,w: integer;
  begin w:=Ø; i:=last;
        while tree[i] = 1 do
        begin tree[i]:=Ø; w:=w+1; i:=i-1 end;
        tree[i]:=1;
        for i:=1 to w-1 do tree[2(n-i)+1]:=1;
        last:=2n-1
        end; (* NEXT *)
```

3. Ranking of feasible strings

Iterative application of the procedure NEXT from the preceding section produces the master list of binary strings representing binary trees with n nodes. The interpretation of these strings depends on the chosen monotonic traversal order.

Henceforth, by a string we will understand a <u>feasible string</u>, i.e., one that is on our master list. A feasible string representing a binary tree has no proper prefix in which the number of Ø's exceeds that of 1's. We will now concentrate on determining the position of a given string t on our master list, and denote this position by RANK(t). Thus, RANK(FIRST) = 1, and RANK(NEXT(t)) = 1 + RANK(t) for all strings t but the last on the master list.

We observe that the position of a string on the master list depends directly on the displacement of its 1's from their original positions ($2(n-j)+1$ for the jth rightmost 1). Denoting by $d(t,j)$ the displacement of the jth 1 in the string t, we have $d(t,n)=0$, and $d(t,j) \leq d(t,j+1) + 1 \leq n-j$, for $j=1..n-1$. Consider a string t' which has $d(t',i)=0$ for all $i<j$, and $d(t',j)=k$ for some fixed j, $k \leq n$ and such that $k \leq n-j$. Consider another string t" such that $d(t",i)=d(t',i)$ for all $i \neq j$ and $d(t",j)=0$. We define the <u>displacement function</u> <u>g(j,k)</u> as the distance between these two strings on our master list:

$$g(j,k) = \text{RANK}(t') - \text{RANK}(t")$$

We note that values of the displacement function do not depend on the number of nodes, n. Figure 2 illustrates the definition.

From this definition it follows that $g(j,0)=0$ and that $RANK(t') = g(j,k) + g(j+1,d(t'',j+1)) + RANK(t''')$, where $d(t''',i)=d(t'',i)$ for all $i>j+1$, and $d(t''',i)=0$ for $i \leq j+1$. By induction, for any string t we have

$$(3.1) \qquad RANK(t) = 1 + SUM(\ g(i,d(t,i)) \mid 1 \leq i \leq n\ )$$

```
t''':      1 0 1 0 1 0 1 0 1 0 0
           . . . . . . . . . . .
t"  :      1 1 0 0 1 0 1 0 1 0 0
           1 1 0 0 1 0 1 1 0 0 0
           1 1 0 0 1 1 0 0 1 0 0
           1 1 0 0 1 1 0 1 0 0 0
           1 1 0 0 1 1 1 0 0 0 0
t'  :      1 1 0 1 0 0 1 0 1 0 0
```

<u>Figure 2</u> Representations of t', t", and t''' when $n=5$, $j=3$, and $k=1$.

By definition of the procedure NEXT, the string *t' immediately preceding t' on our master list has $d(*t',i)=k+j-i-1$ for $i \leq j$, $d(*t',i)=d(t',i)$ for $i>j$, and $RANK(t') = RANK(*t') + 1$. From this and (3.1) we have the recurrence relation defining g:

$$g(j,k) = 1 + SUM(\ g(i,k+j-i-1) \mid 1 \leq i \leq j\ ) \text{ for } 1 \leq k \leq n-j,\ j<n$$
$$g(j,0) = 0$$

Following Dershowitz and Zaks [2], we find the closed form solution to this recurrence relation.

(3.2) $g(j,k) = \binom{2j+k-1}{j} - \binom{2j+k-1}{j-1}$ for $j>0$, $k\geq 0$.

The formula (3.1) suggests a simple algorithm computing RANK(t) for a given string t.

```
function RANK( t:string ): integer;
  var sum, j: integer;
  begin sum:=1; j:=1;
        for i:=last downto 2 do
        if tree[i]=1
        then begin sum:=sum+g(j,2(n-j)-i-1);
                j:=j+1
                end;
        RANK:=sum
        end; (* RANK *)
```

Assuming a constant time access to the values of g(j,k), the complexity of the algorithm RANK is obviously linear with the length of the string (with the number of nodes in the represented by it tree). However, as the formula (3.2) contains binomial coefficients, the related computations may take substantial amount of time because of the magnitude of the factorial values involved. A much more realistic assumption is that the table of values g(j,k) will be precomputed using the recurrence relation (3.1). This is an $O(N^2)$ process which may serve many ranking and unranking requests, for trees with different numbers of nodes n, bounded by some fixed value N. Figure 3 gives an example of such a table.

7	7	35	154					
6	6	27	110	429				
5	5	20	75	275	1001			
4	4	14	48	165	572	2002		
3	3	9	28	90	297	1001	3432	
2	2	5	14	42	132	429	1430	4862
1	1	2	5	14	42	132	429	1430
k/j	1	2	3	4	5	6	7	8

Figure 3 Table of the displacement function g(j,k).

The unranking procedure consists of the reconstruction of the values d(t,j) from their composite representation as RANK(t). From the recurrence relation defining the displacement function g(j,k), we have that for any string t and its displaced jth rightmost 1

$$g(j,1+d(t,j)) > \mathrm{SUM}(\ g(i,d(t,i))\ \mid\ 1 \le i < j\ )$$

This follows from the constraints that the position of the jth 1 puts on the displacement of the other 1's (to its right in t). We can thus determine the displacement of 1's of t by comparing the ranking number (or the remainder thereof) with the values of the displacement function, from the leftmost 1 subject to displacement (j = n-1) through the rightmost one. The restriction $d(t,j-1) \le d(t,j) + 1$ allows a linear algorithm UNRANK given below.

```
procedure UNRANK(x:integer);
  var i,j,k: integer;
  begin tree[1]:=1; k:=1; i:=2;
        for j:=n-1 downto 1 do
        begin while g(j,k)>x do k:=k-1;
                while i≤2(n-j)-k do
                begin tree[i]:=0; i:=i+1 end;
                tree[i]:=1; i:=i+1;
                x:=x-g(j,k); k:=k+1
                end;
        last:=i-1;
        while i≤2n+1 do begin tree[i]:=0;
                                i:=i+1 end
        end; (* UNRANK *)
```

4. Conclusions

We have proposed an algorithm enumerating a sequence of binary strings representing (extended) binary trees. The algorithm performs consecutive updates of of the strings in constant time, when averaged over all string updates. Based upon this enumeration algorithm, we also propose algorithms computing the rank of a tree representation, and computing the tree given its rank. These algorithms are very efficient, under assumption of a fast access to values of a two-argument function closely related to functions counting grid paths from the origin to a given point. Dershowitz and Zaks [2] eloquently present the combinatorial background to those counting results.

5. References

[1] T. Beyer and S.M. Hedetniemi, "Constant Time Generation of Rooted Trees", Computer and Information Science Department Technical Report University of Oregon.

[2] N. Dershowitz and S. Zaks, "Enumeration of Ordered Trees", Discrete Mathematics 31, 1980, pp. 9-28.

[3] G.D. Knott, "A Numbering System for Binary Trees", Communications of the ACM 2Ø, 2(1977), pp. 113-115.

[4] A. Proskurowski, "On the Generation of Binary Trees", Journal of the ACM 27, 1(198Ø), pp. 1-2.

[5] R.C. Read, "How to Avoid Isomorphism Search when Cataloguing Combinatorial Configurations", Ann. Disc. Math. 2, 1978, pp.1Ø7-12Ø.

[6] F. Ruskey and T.C. Hu, "Generating Binary Trees Lexicographically", SIAM J. Computing 6, 4(1977), pp. 745-758.

[7] M. Solomon and R.A. Finkel, "A Note on Enumerating Binary Trees", Journal of the ACM 27, 1(198Ø), pp. 3-5.