A RANDOM BINARY TREE GENERATOR

Harold W. Martin [*] and Bonnie J. Orr [*]

Department of Computer Science and Mathematics
Northern Michigan University

ABSTRACT.

Let B(N) denote the set of all binary trees that have N nodes. A procedure for randomly generating the trees in B(N) such that each tree is equally likely to occur, that is an unbiased random generator, is given which runs in O(N) time, requires very little storage, and uses a system of arithmetic no larger than is required to represent the number N itself. Previous unbiased random binary tree generators, based on inverse rank functions, ran in O(N log N) time and required multiple precision arithmetic capable of handling numbers of the order of magnitude of the cardinality of B(N).

1. INTRODUCTION.

Let B(N) denote the set of all binary trees that have exactly N nodes and let C(N) denote the Catalan number $\binom{2N}{N}/(N + 1)$. It is well known that the number of elements in B(N) is exactly C(N). A rank function is a one-to-one function R from B(N) onto the set $\{1, 2, \ldots, C(N)\}$ of positive integers equal to or less than C(N). In [3], G. Knott gave algorithmic constructions for a rank function R and its inverse R^{-1} and pointed out that R^{-1} could be used to construct an unbiased random binary tree generator, that is, a binary tree generator such that any tree in B(N) has the same probability 1/C(N) of being generated as any other tree in B(N). One simply generates a random integer i in $\{1, 2, \ldots, C(N)\}$ and then constructs the binary tree $R^{-1}[i]$. Solomon and Finkel have shown that Knott's construction of $R^{-1}[i]$ can be done in O(N log N) time so that Knott's random binary tree generator runs in O(N log N) time, [12]. This is mathematically an elegant approach to constructing an unbiased random binary tree generator, but for large values of N it is computationally unwieldy. For example, if N = 5000, C(N) will have over 2000 digits in decimal notation. Therefore, both the random generation of i and the construction of the tree $R^{-1}[i]$ are greatly hampered.

In this paper we present a new unbiased random binary tree generator which runs in O(N) time, requires very little storage, and in addition does not require multiple precision arithmetic. Our binary tree generator constructs a random tree directly by randomly generating an inversion table, which is one of several types of sequential representations for binary trees found in the literature, [1 - 14].

If T is a binary tree having N nodes, then the inversion table representing T is a unique sequence $\{x(i)\}$ of length N consisting of non-negative integers such that x(1) = 0 and having the property that if $j < N$, then x(j+1) cannot exceed x(j) + 1. The two properties, that x(1) = 0 and that if $j < N$, then x(j+1) cannot exceed x(j) + 1, are actually characteristic of inversion tables,

* Partially supported by the National Science Foundation, Grant Number CCR-8804250

[3]. Given the value of N, our generator probabilistically selects the successive elements x(2), x(3), ... , x(N) of an inversion table in such a way that every inversion table of length N has the same probability of being generated. The generated tree is represented by an inversion table, but we have developed O(N) time algorithms for transforming an inversion table for a binary tree T into other standard representations for T, [5], so the method is quite flexible.

We will now describe an infinite tree AB, depicted in Figure 1, which will be used to motivate the idea underlying the binary tree generator as well as to graphically illustrate the distinction between our generator and a generator built on an inverse rank function. Each node of AB represents a unique binary tree. The root is at level 0 and represents the empty binary tree. The root has one child, at level one, which represents the single binary tree having one node. The two nodes at level two represent the two trees in B(2), etc. The edges of AB are labeled by non-negative integers. If a node n is such that the edge from the parent of n to n is labeled i, then n has i + 2 children; the i + 2 edges from n to the children of n, ordered from left to right, are labeled 0, 1, ... , i + 1. Suppose N = 4 and we wish to randomly generate a binary tree having four nodes, that is, an element of B(4). Letting each node at level 4 in AB represent an element of B(4), we could randomly generate an integer in the set 1, 2, ... , 14 , say i, and then select the i-th node from the left in the fourth level of AB. If i = 5, then the node at the end of the path starting from the root whose edges are labeled 0, 0, 1, 2 has been selected. An inverse rank function R^{-1} will produce a binary tree $R^{-1}[5]$ in some notation. But, the sequence 0,0,1,2 is an inversion table which represents a tree in B(4), so one choice for R is to construct 0,0,1,2 . Our generator works in the opposite direction by starting at the root and generating a path, or inversion table, which terminates at level 4. This is done as follows. If x(1),x(2),x(3),x(4) represents the final result, we must have x(1) = 0. From Figure 1 it is clear that we must randomly choose to have x(2) = 0 with probability 5/14 and to have x(2) = 1 with probability 9/14. If the assignment 0 is made to x(2), then x(3) must be either 0 or 1, and x(3) = 0 with probability 2/5 and x(3) = 1 with probability 3/5. If x(2) = 1, then x(3) must be 0, 1 or 2 with probabilities 2/9, 3/9 or 4/9 respectively. If x(3) = i, then the possible assignments of 0, 1, ... , i + 1 to x(4) are made with equal probabilities.

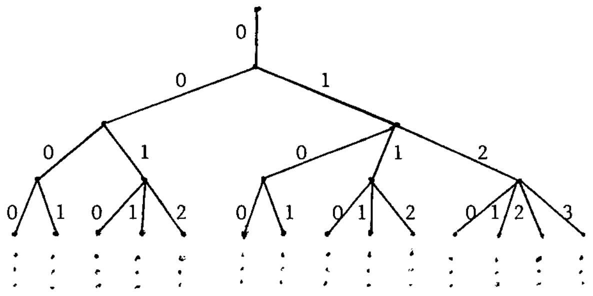

FIGURE 1.

In Section 2 we will give a brief discussion of inversion tables as preparation for Section 3. Our definition of inversion table given in Section 2 is not standard but is easily seen to be equivalent to the usual definition given by Knott in [3] and is more revealing for our purpose. The formal description of our unbiased binary tree generator is given in Section 3 and in Section 4 we will conclude with some remarks on further directions to be explored.

2. INVERSION TABLES.

The inversion table representation for binary trees is usually defined in terms of another representation called a tree permutation [3]. The term "inversion table" makes sense within the context of tree permutations and we stay with the established terminology. It is not difficult to show that our definition does indeed correctly define the concept of an inversion table.

2.1 DEFINITION. Given a binary tree T having N nodes, let the root of T be labeled with 0. Inductively we define a labeling for every node of T as follows: if a node z is a left child of its parent, then label z with 1 plus the value of the label of its parent; if z is a right child of its parent, label z with the same label as its parent. After having labeled the nodes of T, do a preorder traversal of T, writing down the labels as each node is visited, to form a sequence called the inversion table representing T.

Given an inversion table, it is easy to construct the binary tree that it represents. Each entry in the inversion table is a label on a node of the tree being constructed with the initial element 0 labeling the root. Starting from the root we do a preorder construction as follows: if the second element is 1, go left and create a left child of the root and label this new node with a 1; if the second element is 0, go right, creating a right child of the root and label this new node with a 0. In general, suppose the i-th element of the inversion table is k where $i < N$, and that i nodes of the tree have been constructed so far. The label of the i-th node of the partial tree so far constructed is k. Let m be the (i + 1)-th element of the inversion table. Three cases may occur, namely, $k < m$, $k = m$ and $k > m$. If $k < m$, then go left creating a left child of the i-th node and label this new node m. If $k = m$, go right creating a right child of the i-th node and label this new node m. Finally, if $k > m$, then back track in the partial tree already constructed until we reach the first node, say z, whose label is m; there will always be such a node; from z, go right creating a new node which is the right child of z and label this new node m.

We illustrate the construction below. The binary tree of Figure 2 is the tree associated with the inversion table $\{0,1,2,3,1,0,0,1\}$.

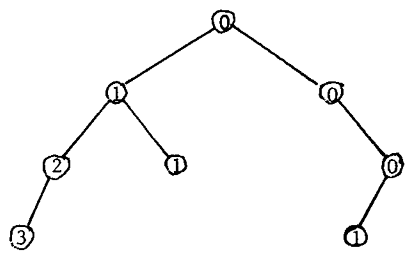

FIGURE 2.

Note that a preorder traversal of the tree of Figure 2 yields the original inversion table $\{0,1,2,3,1,0,0,1\}$.

3. AN UNBIASED RANDOM BINARY TREE GENERATOR.

Suppose we are generating an inversion table of length N and suppose we have already determined the values of the first j terms, $x(1) = 0$, $x(2)$, ... , $x(j)$, where $j < N$. Suppose also that $x(j) = i$. We must assign a value to $x(j+1)$ from the set $\{0, 1, \dots , i+1\}$. We need to know the probability that should be assigned to each integer k in the set $\{0, 1, \dots , i+1\}$. It is more convenient to use a cumulative probability distribution function F(k); that is, we let F(k) denote the probability that should be given to the event that $x(j+1)$ is assigned a value from the set $\{0, 1, \dots , k\}$. We could compute the value of F(k), tediously, in the following way. First construct a tree T of height N where T consists of the first N + 1 levels of the tree AB described in Section 1. Starting at the root of T, follow the path $x(1), x(2), \dots , x(j)$ down to the node z at level j in T. Let S denote the subtree of T which is rooted at the node z. Let A denote the set of all the leaves of S which lie on a path p from z such that the first edge in p has a label from the set $\{0, 1, \dots , k\}$. Let B be the set of all leaves in S. Then,

$$F(k) = |A| / |B|.$$

Our fundamental result is a combinatorial formula for F(k), which is given in terms of N, i and j, so we express F(k) as F(N,i,j,k) :

$$(3.1) \quad \dots \quad F(N,i,j,k) = \frac{(k+1)\cdot(N-j+i+2)!\cdot(2N-2j+k)!}{(i+2)\cdot(N-j+k+1)!\cdot(2N-2j+i+1)!} .$$

The proof of correctness of Equation 3.1 is too lengthy to include here, but can be found in [6].

To illustrate the use of Equation 3.1, let $N = 5$, $j = 3$ and $i = 1$. Then $x(3) = 1$. Note that $x(2)$ may be either 0 or 1. The value of $x(4)$ must be either 0, 1 or 2. The possibilities are listed below :

(3.2) ...

$\{0,x(2),1,0,0\}$	$\{0,x(2),1,1,1\}$	$\{0,x(2),1,2,1\}$
$\{0,x(2),1,0,1\}$	$\{0,x(2),1,1,2\}$	$\{0,x(2),1,2,2\}$
$\{0,x(2),1,1,0\}$	$\{0,x(2),1,2,0\}$	$\{0,x(2),1,2,3\}$.

Letting F(5,1,3,k) be denoted by F(k), we see from the nine inversion tables above that we must have F(0) = 2/9, F(1) = 5/9 and F(2) = 1, that is, the occurrence of x(4) = 0 should have a probability of 2/9, of x(4) = 1 a probability of 3/9 and of x(4) = 2 a probability of 4/9. The values of F(0), F(1) and F(2) may be computed directly from Equation 3.1.

We will now show how Equation 3.1 can be used in practice to give an O(N) time unbiased random binary tree generator.

Suppose N, i and j are given where $N > 1$, $i < j < N$ and x(1) = 0, x(2), ... , x(j) = i have already been computed. We must randomly determine the value of x(j+1) from the set $M = \{0,1, .. ,i+1\}$. Let x be a randomly generated number in the interval [0,1). We do a sequential search on M starting at i + 1 and working down :

```
procedure assign*;
begin
   m := i + 1;
   while x < F(N,i,j,m-1) do
     m := m - 1;
   x(j + 1) := m;
end;
```

Note that F(N,i,j,-1) = 0 so if x = 0, then the while loop is finished when m = 0, in which case x(j + 1) is correctly assigned the value zero.

Procedure assign* could be written as a function so that we may notationally express the assignment to x(j + 1) as x(j + 1) := assign*(j). We also assume that the random number x is generated within the code for the function assign*. Then the random binary tree generator is simply a succession of calls to assign*. In addition, we also are hypothetically assuming that the random number generator is perfect.

Computationally the procedure assign* is wasteful. We will now develop a refined version of assign*, called assign, and show that our tree generator runs in O(N) time when using assign.

Let P(N,i,j,k) denote the probability that x(j + 1) will be assigned the value k where k is in $M = \{0, 1, ... , i+1\}$. Then

$$P(N,i,j,k) = F(N,i,j,k) - F(N,i,j,k-1)$$

or, from Equation 3.1,

$$(3.3) \; ... \; P(N,i,j,k) = \frac{(k+2)\cdot(N-j)\cdot(N-j+i+2)!\cdot(2N-2j+k-1)!}{(i+2)\cdot(N-j+k+1)!\cdot(2N-2j+i+1)!} .$$

Define Q(N,j,k) = P(N,i,j,k-1)/P(N,i,j,k). From Equation 3.2 we get

$$(3.4) \; ... \; Q(N,j,k) = \frac{(k+1)\cdot(N-j+k+1)}{(k+2)\cdot(2N-2j+k-1)} ,$$

which is curious because of the disappearance of the i. From Equation 3.3 we get the probability that x(j + 1) will be assigned the value i + 1, namely

$$(3.5) \; ... \; P(N,i,j,i+1) = \frac{(i+3)\cdot(N-j)}{(i+2)\cdot(2N-2j+i+1)} ,$$

which is easy to evaluate since the factorials are gone. Also we have

$$P(N,i,j,k-1) = Q(N,j,k) \cdot P(N,i,j,k) ,$$

so that computing the values P(N,i,j,k) for decreasing values of k starting at i + 1 can be done iteratively and quickly. We now reformulate the procedure assign*. We will abbreviate P(N,i,j,k) to P(j,k) and Q(N,j,k) to Q(j,k).

```
function assign(j);
begin
   k := i + 1;  /* recall that x(j) = i  */
   P := P(j,k);
   sum := P;
   x := random;
   while (1 - x)    sum do
    begin
      Q := Q(j,k);     /* The first time thru
      P := Q * P;       * the loop, the new
                        * value of P is P(j,i)
      sum := sum + P;   * the second time is
      k := k - 1;       * P(j,i-1), etc.  */
    end;
   assign := k;  /* x(j + 1) := k */
end;  /*  function assign  */

procedure generate_tree;
begin
   x(1) := 0;
   for j := 1 to N - 1 do
      x(j + 1) := assign(j);
end;   /*  procedure generate_tree  */
```

In order to analyse the procedure generate_-tree, let count(j) denote one plus the number of times the while loop is executed in the function call assign(j); count(j) is the number of times the variable P is computed in a call to assign(j). Then let total(N) denote the sum of the numbers count(j) for j equal to 1, 2, ... , N - 1.

3.6 THEOREM. total(N) = 2N - 2 - x(N) for all $N > 1$.

PROOF. The proof is by induction on N. If N = 2, a simple computation shows that the statement of the theorem holds.

Assume the statement of the theorem holds for all values less than N + 1. An analysis of the function assign shows that

count(j) = x(j) - x(j + 1) + 2.

For example, if x(j + 1) = x(j) + 1, then count(j) = 1; if x(j + 1) = x(j), then count(j) = 2; if x(j + 1) = x(j) - 1, then count(j) = 3, etc. We have

(3.7) ... total(N + 1) = total(N) + count(N).

By the statement of the theorem, we must show that total(N+1) = 2(N+1) - 2 - x(N+1) = 2N - x(N+1). But this is true since the summands in the right hand side of Equation 3.7 compute to the following values :

total(N) = 2N - 2 - x(N), by the inductive hypothesis;

count(N) = x(N) - x(N + 1) + 2;

So total(N) + count(N) = 2N - x(N + 1), completing the proof.

3.8 COROLLARY. The unbiased random binary tree generator has time complexity O(N).

4. CONCLUDING REMARKS.

Definition 2.1 easily generalizes to yield the notion of a K-inversion table where K is a positive integer: a sequence {x(1), x(2), ... , x(N)} of non-negative integers is a K-inversion table if x(1) = 0 and for any positive integer $j < N$, the quantity x(j + 1) - x(j) is less than K. An inversion table is a 2-inversion table and a K-inversion table uniquely represents a K-ary tree. For each $K > 2$ an analogue of the tree AB exists, each node of which uniquely represents a K-ary tree and such that the set of all K-ary trees having N nodes is represented by the set of all nodes at level N in the tree. The techniques of this paper generalize in a straightforward way to yield an O(N) time unbiased random K-ary tree generator which works by randomly generating K-inversion tables. The key lies in finding an appropriate generalization of Equation 3.1; this combinatorial work is contained in [6].

The derivation of the following approximation formula is too long to include here. In practice Equation 4.2 has proven to be reasonably good, losing accuracy only when the random number x is close to 0, usually about 0.005 or less but higher when j is near N.

4.1 APPROXIMATION FORMULA. For N, i and j fixed, make the assignment x(j + 1) = k where

(4.2) .. k = i + 1 - int(ln(x)/ln(1 - P(N,i,j,i)),

where x is a random number in (0,1), P is given by Equation 3.5, and int is the integer part function. If x = 0, we let x(j + 1) = 0.

We have made scores of trial runs with values of N ranging from 50 to 5000. Here are some samples from a trial run with N = 1000; the results of this run were typical of all runs which we made :

With j = 74, i = 28 and x = 0.021889 we had x(75) = 24 by both Equation 4.2 and the function assign. With j = 382, i = 51 and x = 0.002486 we had x(383) = 42 by assign and x(383) = 44 by Equation 4.2. With j = 81, i = 23 and x = 0.000156 we had x(82) = 0 by assign and x(82) = 12 by Equation 4.2, an unacceptable error. On the run from which this data was taken, there were only 13 cases out of 999 in which the values for x(j + 1) computed by Equation 4.2 and by assign were in disagreement and seven of these disagreed by only one.

With a good implementation of the natural logarithmic function ln, Equation 4.2 used in conjunction with assign, should lead to improved performance. For example, for large values of i and small values of x, assign will do a lot of looping; if x is in the range of applicability of Equation 4.2, the approximation formula should be called.

Our interest in Equation 4.2 is not only computational, but also analytic; it clearly shows for fixed N, i and j how the assignments x(j + 1) depend upon the random number x. The problems of precisely understanding the range of applicability of Equation 4.2 and of finding better approximation techniques remain.

Finally, it would be interesting to construct unbiased random K-ary tree generators which would run in O(log N) time on parallel computers.

REFERENCES

1. Beyer, Terry and Hedetniemi, Sandra Mitchell, Constant time generation of rooted trees, SIAM Journal on Computing, 9 (1980), 706-712.

2. Er, M. C., A note on generating well-formed parenthesis strings lexicographically, The Computer Journal, 26 (1983), 205-207.

3. Knott, Gary D., A numbering system for binary trees, Communications of the ACM, 20 (1977), 113-115.

4. Knuth, D. E., The Art of Computer Programming: Vol. 1 : Fundamental Algorithms, Addison-Wesley, Reading, MA., 1968.

5. Martin, H. W., Linear time recognition and transformation algorithms for binary tree representations, in preparation.

6. Martin, H. W., Kiltinen, J. O. and McNeill, R. B., The representation and enumeration of ordered K-ary trees, submitted for publication.

7. Pallo, J. M., Enumerating, ranking and unranking binary trees, The Computer Journal, 29 (1986), 171-175.

8. Proskurowski, Andrzej, On the generation of binary trees, Journal of the ACM, 27 (1980), 1-2.

9. Rotem, Doron and Varol, Y. L., Generation of binary trees from ballot sequences, Journal of the ACM, 25 (1978), 396-404.

10. Ruskey, F. and Hu, T. C., Generating binary trees lexicographically, SIAM Journal on Computing, 6 (1978), 745-758.

11. Smith, Harry F., Data Structures: Form and Function, Harcourt Brace Jovanovich, Orlando, Florida, 1987.

12. Solomon, Marvin and Finkel, Raphael A., A note on enumerating binary trees, Journal of the ACM, 27 (1980), 3-5.

13. Trojanowski, Anthony E., Ranking and listing algorithms for k-ary trees, SIAM Journal on Computing, 7 (1978), 492-509.

14. Zaks, S., Lexicographic generation of ordered trees, Theoretical Computer Science, 10 (1980), 63-82.