The Advantages of Forward Thinking in Generating Rooted and Free Trees

Extended Abstract (SODA 99)

GANG LI AND FRANK RUSKEY *

Department of Computer Science

University of Victoria

Victoria, B.C. V8W 3P6, Canada

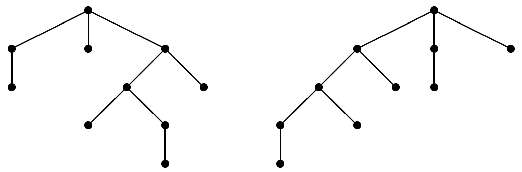
Abstract

We present a new approach to the exhaustive generation of rooted and free trees of n nodes. Our algorithms have the following features: (1) They are optimal in the sense of using linear space and having running times that are proportional to the number of trees produced. (2) They use a natural representation of the trees, the parent array. (3) They can generate the trees in many different orders, including reverse lexicographic order. (4) They are recursive, not iterative. (5) They can be easily modified to generate restricted classes of trees and still preserve properties (1)-(4). The possible restrictions are: (a) in the case of rooted trees, upper bounds on the number of children of a node and lower and upper bounds on the height of the tree; and (b) in the case of free trees, upper bounds on the degree of a node and lower and upper bounds on the diameter of the tree.

1 Introduction

Rooted trees occur throughout computer science, where, for example, they are used in data structures for disjoint sets, and in mathematics, where they are studied in conjunction with bracketing systems, and even in biology, where they are used for the evolutionary classification of species. Free trees (connected acyclic graphs) are a staple of graph theory and graph algorithms. Several algorithms have been developed for generating (i.e., listing) all rooted trees, all free trees, and their variants. The earliest algorithm seems to be that of Scions [10]. He represented a rooted tree by "level sequences", obtained by recording the levels (distance to the root) of each node in a

*Research supported in part by the Natural Sciences and Engineering Research Council of Canada under Grant A3379. e-mail: `fruskey@csr.uvic.ca`



0121156685 0123432181

Figure 1: Two isomorphic rooted trees. The one on the right is canonic.

012345	012321	012345	012321	0123316
012344	012315	012344	012312	0123216
012343	012311	012343	012311	0123156
012342	012222	012342	012222	0122221
012341	012221	012341	012221	0122216
012333	012215	012333	012212	0122211
012332	012211	012332	012211	0122155
012331	012141	012331	012121	0122151
012325	012111	012323	012111	0121416
012322	011111	012322	011111	0121411
				0111111
(a)		(b)		(c)

Figure 2: (a) The 20 parent arrays of canonic trees with $n = 6$ listed in relex order (read down); (b) The corresponding level sequences; (c) parent arrays of canonic free trees for $n = 7$.

preorder (depth-first-search) traversal. In order to identify a unique "canonic" tree, the rooted trees are (recursively) drawn with subtrees of greater height to the left (see Figure 1). The level sequences of canonic trees were then generated in lexicographic order.

The most well-known algorithm for generating rooted trees is due to Beyer and Hedetniemi [1] (also described in the book of Wilf [14]). The algorithm also uses level sequences, but generates them iteratively in relex (reverse lexicographic) order. Moreover, they were the first to prove that an algorithm for generating rooted trees did so in constant time, amortized over all trees. We call this the CAT property (for Constant Amortized Time). The output of the Beyer-Hedetniemi algorithm is shown in Figure 2.

Another method of generating rooted trees was developed by Kozina [4] but it seems to have no advantages over the other two mentioned above, and has a running time of $O(nr_n)$, where r_n is the number of rooted trees on n nodes.

Generating free trees is considerably more complicated. A *center* in a tree is a node that minimizes the maximum distance to any other node in the tree. It is

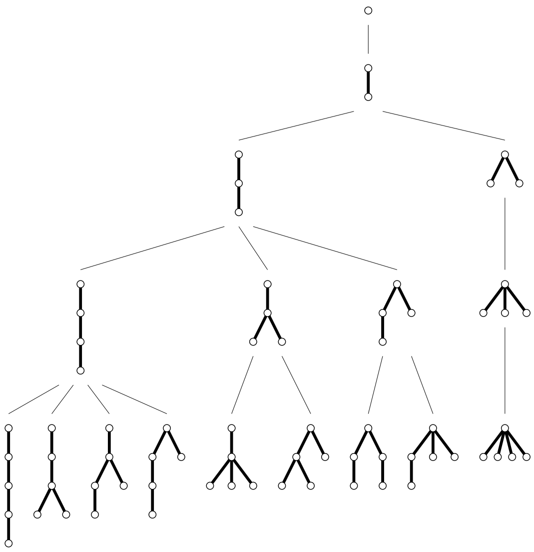


Figure 3: The tree of rooted trees (up to 5 nodes) $\mathcal{T}_5$.

well-known that every tree has either one center (is *unicentral*) or has two centers (is *bicentral*), and that the two centers of a bicentral tree must be adjacent. The main idea most algorithms for generating free trees is to root the free tree at one of its centers and then generate the resulting restricted class of rooted trees.

Scions [10] discussed the problem, and Read [9] developed and algorithm which is impractical since it had to store all trees of $n-1$ nodes to generate those with n nodes. Kozina [4] developed an algorithm that uses linear space and which generated bicentral and unicentral trees separately, with running time $O(n^2 r_n)$. Another algorithm with the same running time as Kozina's was developed by Liu [7]. Tinhofer and Schreck [11] gave a new coding of free trees and an $O(nr_n)$ generation algorithm.

A definite breakthrough came when Wright, Richmond, Odzlyko, and McKay [15] adapted the Beyer-Hedetniemi algorithm to generate free trees, for the first time by a CAT algorithm.

Our contributions are outlined in the remainder of this section. First of all, our algorithms are recursive. Generally speaking, recursive algorithms are simpler, easier to prove correct, and more flexible. Algorithms for generating combinatorial objects are often used as the basis of backtracking algorithms and recursive formulations are prefered, because of the ease of eliminating non-productive subtrees.

Secondly, our algorithms are optimal in the sense that they use linear space and are CAT. There are existing iterative algorithms with these properties as mentioned above.

Thridly, our algorithms can be easily adapted to generate various restricted types of trees for which there were no previous algorithms. For rooted trees, we can allow handle upper bounds, k, on the number of children of nodes in the tree. The $k = 2$ case is particularly important and Sloane's sequence database refers to the number of such trees as the Wedderburn-Etherington numbers [2], [13]. Algorithms for $k = 2$ have been developed previously by Pallo [8] (who observed experimentally that his algorithm was CAT) and by Kubicka and Kubicki [6] (who proved that their algorithm is CAT). The case of $k > 2$ had not been previous treated. For rooted trees we also can handle via a CAT algorithm upper and/or lower bounds on the height of the tree. For free trees, we can handle via a CAT algorithm upper and/or lower bounds on the diameter of the tree, and/or an upper bound, u, on the degree of nodes. For $u = 3$ the tree are called Cayley 3-trees, and for $u = 4$ they are called hydrocarbons or quartic trees (i.e., in Sloane's database). Generating these restricted classes of free trees had not been previously treated in the literature.

Finally, our algorithms use the parent array to represent trees. This representation is somewhat more natural than the level sequence representation. If the application calls for a parent array then no separate conversion step is necessary.

The algorithms developed in this paper have been implemented and their output is available to users in an interactive manner over the world-wide-web. The "combinatorial object server" (COS) at `http://sue.csc.uvic.ca/~cos` in the tree section, contains options for generating rooted and free trees and their restricted versions. Programs for generating rooted and free trees will soon be be available there as well.

2 Generating Rooted Trees

For computer scientists, the most well-known way of representing rooted trees is by parent arrays. Let T be a rooted tree whose nodes are labeled $1, 2, \ldots, n$. For each non-root node i, the array entry $par[i]$ is the parent of i. If i is the root, then $par[i] = 0$. Of course, a given rooted tree could have many different par representations depending upon how the tree was labeled. In order to achieve a unique representative for each tree, we label the tree in a depth-first-search manner, and among all such labelings select the one that is lexicographically maximum. Such a tree is said to be *canonic*. The parent array of a canonic tree is also said to be canonic.

Given a canonic tree T on n nodes, removing the nth node results in a canonic tree T' with $n-1$ nodes. This implies an infinite ordered tree structure $\mathcal{T}$ on the set of all rooted trees, with the single node tree at the root of $\mathcal{T}$, and T the child of T'. The ordering of subtrees is reverse lexicographic order of the corresponding parent arrays. Let $\mathcal{T}_n$ be that subtree of $\mathcal{T}$ containing rooted trees with at most n nodes. Figure 3 shows $\mathcal{T}_5$.

Our strategy is to develop a recursive algorithm whose computation tree is $\mathcal{T}_n$ and which uses only a constant amount of computation at each node of the computation tree. We will then observe that the number of nodes in $\mathcal{T}_n$ is proportional to the number of rooted trees with n nodes, which will prove that the algorithm is CAT.

Now suppose that we have a tree T with $n-1$ nodes and want to extend to a tree with n nodes. Because of the preorder labeling, node n is the child of some ancestor of node $n-1$ (a node is considered to be an ancestor of itself). What about

```
procedure Gen( p, s, cL : integer);
begin
{R1}  if (p > n) (* or ((par[p-1] = 1) and (par[p-2] = 1)) *) then PrintIt
{R2}  else begin
{R3}    if cL = 0 then par[p] := p-1 else   {<---- first tree}
{R4}    if par[p-cL] < s
{R5}       then par[p] := par[s]            {<---- copying root}
{R6}       else par[p] := cL + par[p-cL];   {<---- copying non-root}
{R7}    Gen( p+1, s, cL );
{R8}    while par[p] > 1 do begin
{R9}       s := par[p];
{R10}      par[p] := par[s];
{R11}      Gen( p+1, s, p-s );
{R12}   end;
{R13} end;
end {of Gen};
```

Figure 4: Recursive CAT algorithm for generating rooted trees.

canonicity? Observe that if $\langle par[1..n-1], par[n]\rangle$ is canonic and $par[n] \neq 1$, then $\langle par[1..n-1], par[par[n]]\rangle$ is also canonic. Thus, given $par[1..n-1]$, we wish to know the lowest ancestor x of $n-1$ for which $par[1..n-1], par[x]$ is canonic.

Our algorithm maintains two parameters s and cL. Parameter s is the root of a subtree that is being repeatedly copied as we descend along a leftmost path (except for *the* leftmost path) in $\mathcal{T}$. Parameter cL is the number of nodes in the subtree $T(s)$ rooted at s. Pascal code is shown in Figure 4 as the procedure `Gen(p,s,cL)`. The procedure produces all parent arrays with prefix $par[1..p-1]$. The initial call is `Gen(1,0,0)`; no initialization is necessary.

In order to prove that the algorithm is correct we need to more precisely define the critical node s. We will give the definition but omit the proof for space reasons.

DEFINITION **2.1** *Let T be a tree with $n-1$ nodes and let q be the highest (i.e., having smallest value) ancestor of $n-1$ such that $T(q)$ is a prefix of $T(c)$, where c is the rightmost proper sibling of q. Then the* critical node *s is the leftmost sibling of c such that $T(s)$ is isomorphic to $T(c)$.*

The key idea is that the lexicographically largest infinite tree, agreeing with T in the first $n-1$ positions of its parent array, is formed by deleting $T(q)$ and replacing it with an infinite sequence of subtrees, all isomorphic to $T(s)$ and all with the same root as s. Of course truncating this infinite tree at any node $N \geq n$ gives the lexicographically largest extension of N nodes. This is the reason for the updates at lines {`R5`} and {`R6`}. Either we are starting a new copy of $T(s)$ (line {`R5`}) or are making a new isomorphic copy with offset cL (line {`R6`}).

We now argue that the algorithm is CAT. Observe that every iteration of the while loop results in a recursive call, and thus that the total amount of computation is proportional to the number of recursive calls. Let r_n denote the number of rooted trees with n nodes. The number of recursive calls is clearly $r_1 + r_2 + \cdots + r_n$ since the only dependence on n is the output test at line {R1}.

Let $T(x) = \sum_{i \geq 0} r_n x^n$ be the ordinary generating function of the r_n numbers. It's radius of convergence is $\rho \approx 0.3383219$. Kubicka [5] shows that asymptotically

$$\frac{1}{r_n} \sum_{i=0}^{n} r_n \sim \frac{1}{1-\rho} \approx 1.5113.$$

This constant ratio implies that the algorithm is CAT.

The preceding proof relied ultimately on an asymptotic expression for r_n. By slightly modifying the algorithm, we can prove by completely elementary means that the algorithm is CAT. In the computation tree nodes with one child only occur if the corresponding rooted trees have two successive nodes whose parent is the root; i.e., if $par[p-1] = par[p-2] = 1$. Removing the comment delimiters `(*` and `*)` at line {R1} eliminates nodes with only one child. The computation tree now has more leaves than internal nodes and so clearly the underlying algorithm is CAT. Another change is needed to maintain the data structure, namely the assignment `par[p] := 1;` after each recursive call.

2.1 Rooted trees with height restrictions

To generate all trees of height at most h change the test `cL = 0` at line {R3} to

```
(cL = 0) and (p <= h+1)
```

To generate all trees of height exactly h, initialize `par[1..h]` to be $0, 1, \ldots, h-1$ and then call `Gen(h+1,h,1)`. To generate all trees of height at least h, initialize `par[1..h+1]` to be $0, 1, \ldots, h$ and then call `Gen(h+2,0,0)`. In all three cases the resulting algorithms can be proven to be CAT.

2.2 Rooted trees with parenthood restrictions

In this section we consider rooted tree where each node has at most k children. Our strategy is the same as in the rooted tree case; that is, we develop a recursive algorithm whose computation tree is that subtree of $\mathcal{T}_n$, call it $\mathcal{B}_{n,k}$, containing only rooted trees with at most k nodes. Figure 5 shows $\mathcal{B}_{6,2}$. An important difference from $\mathcal{T}_n$ is that in $\mathcal{B}_{n,k}$ there can be rooted trees at levels less than n with no children. The three such nodes in $\mathcal{B}_{6,2}$ are boxed with dotted lines in Figure 5.

A naïve approach is to introduce an array `chi[1..n]` whose ith entry is the number of children of node i. Replace each of the recursive calls `Gen(p,s,cL)` at lines {R7} and {R11} by the three lines

```
chi[par[p]] := chi[par[p]] + 1;
if chi[par[p]] <= k then Gen(p,s,cL);
chi[par[p]] := chi[par[p]] - 1;
```

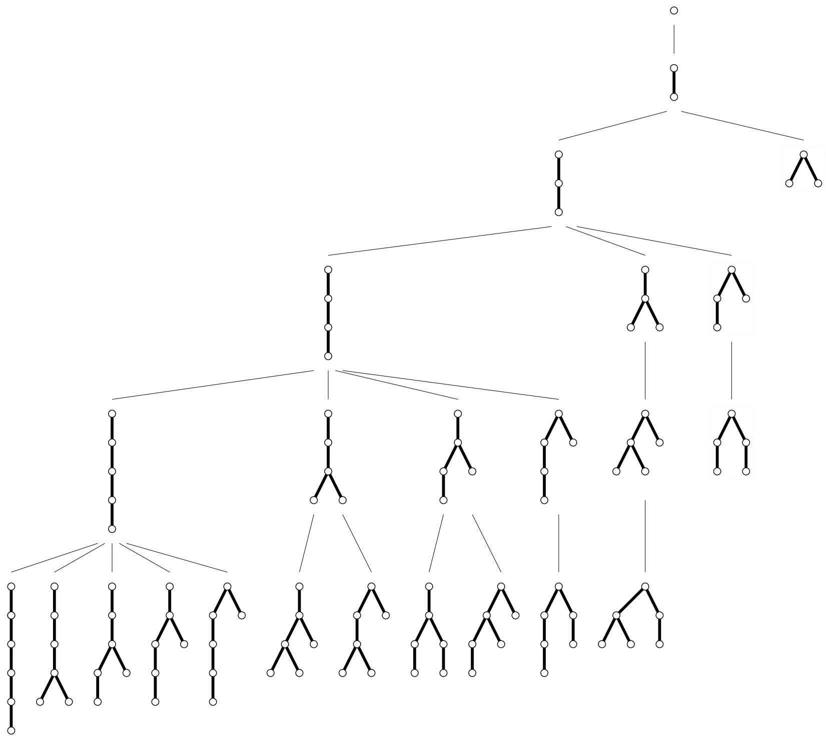


Figure 5: The tree of rooted "binary" trees (up to 6 nodes).

The result is an algorithm whose running times are CAT for realistic values of n (i.e., on average there is at most 3 iterations of the loop for $n \leq 25$), but the amortized cost is in fact growing very slowly.

We may obtain an algorithm that is provably CAT by maintaining an array `jump[1..n]`, where `jump[i]` is the closest ancestor of node i with less than k children, and an array `rchild[1..n]`, where `rchild[i]` is the righmost child of node i. Array `jump` is the parent array of a certain "subtree" of T.

3 Generating Free Trees

With each free tree T we associate a unique canonic rooted tree T_r with root r. The root r is chosen as the unique center if T is unicentral, and is chosen to be the center that lexicographically maximizes the canonic level sequence of T_r if T is bicentral. The parent array of T_r is the representation of T. In the tree T_r, there are two disjoint rooted canonic subtrees T_L and T_R, obtained by removing the edge of T_r from the root to the leftmost subtree. If T_r has height $h+1$, then T_L has height h, and T_R has height $h+1$ if T is unicentral, and has height h if T is bicentral. The parent array of T_r has the form

$$\langle p_1, \underbrace{p_2, \ldots, p_{sL}}_{T_L}, \underbrace{p_{sL+1}, \ldots, p_n}_{T_R} \rangle.$$

Our initial strategy is to adapt the rooted tree procedure Gen to separately generate T_L and T_R by separate procedures, call them GenLT and GenRT.

To generate T_L we have to be able to generate trees with varying numbers of nodes. Another complication is that we will have to bound the height by some function of n. After examing on a case-by-case basis the various possibilities, we can write a procedure GenLT(p, s, cL, h, sL), where the first three parameters are inherited from Gen and h is the desired height and sL can be used to determine the number of nodes in T_L. The required modifications require about 5 additional lines be added to Gen. The call to PrintIt is replaced with a call to GenRT, which generates T_R; further, a call to GenRT is added at as the final statement, which takes care of the different size T_L's.

To generate T_R the number of nodes is fixed, and the height is fixed at one of two values depending on whether the tree is unicentral or bicentral. For fixed T_L, in relex order, the unicentral T_R's precede the bicentral T_R's. The procedure GenRT can be written with the same parameter list as GenLT.

The code for GenLT and GenRT have much in common and we have combined them in GenFree of Figure 6. This code illustrates one the one hand that there are many cases to be dealt with, and on the other hand that there is sufficient simplicity to supply a complete coding in one page. The initial call is GenFree$(3, 0, 0, \lfloor N/2 \rfloor, 0, \lfloor (N+3)/2 \rfloor, 0, 0)$, where $par[0] = 0$ and $par[1] = 1$, will generate all free trees with N nodes. Parameter f indicates whether T_L or T_R is being generated. Parameter g indicates whether T_R is unicentral or bicentral.

Since only constant computation is done at each node of the computation tree, we need only to compute or accurately estimate the number of nodes in this tree. This is not nearly as straightforward as the rooted case since (a) the nodes of the computation tree do not necessarily correspond to canonic free tree representative, and (b) some nodes of the tree can lead to dead-ends. However, we were able to prove that the size of the computation tree is proportional to the number of free trees, thereby proving that the algorithm is CAT. In the proof, we had to make use of asymptotic expresssions for r_n and f_n, and one expression from [15], estimating the number of frees tree representatives of given height.

Using the same general kinds of ideas used to produce restricted classes of rooted trees, we were able to give for the first time CAT algorithms for generating free trees with diameter bounded from above or below or both, and/or degrees of vertices bounded from above.

4 Final Remarks

We compared the running times of our algorithm and the WROM algorithm and found that ours was always faster (e.g., to generate the 14,828,074 free trees on 23 node trees, WROM took 261 sec and ours took 186 sec). These comparisons were done on naïve Pascal implementations of both algorithms and further tuning and

```
procedure expand( p, h, n, N : integer );
begin
  if N-p >= h then Gen( p+1, 2, p-1, h, n, N, 1, 0 );
  if ((p-1)*2 >= N) or ((p-h-1=1) and (par[p] > 2)) or
     ((p-h-1>=2) and ((par[h+2]>2) or (par[h+3]>2))) then
     Gen( p+1, 2, p-2, h, n, N, 1, 1 );
end;

procedure Gen( p, s, cL, h, l, n, f, g : integer );
begin
  if (p > n) then begin
     if (f = 0) then expand( p-1, h, n, N ) else PrintIt;
  end else begin
     if (cL = 0) and (p <= (N div 2)+2) then par[p] := p-1 else
     if par[p-cL] < s then par[p] := par[p-cL]
     else begin
        par[p] := cL + par[p-cL];
        if (g = 1) then
           if ( (l-1)*2 < n) and ( (p-cL+1<l) and (par[p-cL+1]=2)
                 and (p-cL+2<=l) and (par[p-cL+2]=2) )  {case 1}
              or ((p-cL+1=l) and (par[p-cL+1]=2))       {case 2}
              or (p-cL+1>l) ) then begin                {case 3}
                 s := par[p];  cL := p-s;
                 par[p] := par[par[p]];
              end else if (par[p-cL]=2) then par[p] := 1;
     end;
     Gen( p+1, s, cL, h, l, n, f, g );
     while (par[p] > 2) and ((f=0) or (p>l+h-g)) do begin
        if (s=0) then h := p-2;
        s := par[p];  par[p] := par[s];
        if (f=0) {generating T_L?}
           then Gen( p+1, s, p-s, h, 0, N-h+1, f, g )
           else Gen( p+1, s, p-s, h, l,   n,   f, g );
     end {while};
     if (f=0) then expand( p-1, h, p-1, N );
  end;
end {of Gen};
```

Figure 6: Pascal implementation of the free tree generation algorithm.

testing should be undertaken.

References

[1] T. Beyer and S.M. Hedetniemi, "Constant Time Generation of Rooted Trees", *SIAM J. Computing*, 9 (1980) 706-712.

[2] Etherington, "Non-associative powers and a funtional equation", *Mathematical Gazette*, 21 (1937) 36-39.

[3] D.E. Knuth, *Fundamental Agorithms, The Art of Computer Programming*, 1968, Addison-Wesley.

[4] A. V. Kozina, "Coding and generation of nonisomorphic trees", *Cybernetics (Kibernetica)*, vol. 15 (5), 1975 (1979), pg. 645-651 (38-43).

[5] E. Kubicka, "An Efficient Method of Examining All Trees", Combinatorics, Probability and Computing, 5 (1996) 403-413.

[6] E. Kubicka and G. Kubicki, "Constant Time Algorithm for Generating Binary Rooted Trees", Congressus Numerantium, 90 (1992) 57-64.

[7] J. Liu, "Lexicographic generation of rooted trees and trees," Kexue Tongbao, 28 (1983) 448-451.

[8] J. Pallo, "Lexicographic generation of binary unordered trees", *Pattern Recognition Letters*, 10 (1989) 217-221.

[9] R. Read, "How to grow trees," in *Combinatorial Structures and their Applications*, Gordon and Breach, New York, 1970.

[10] H.I. Scions, "Placing Trees in Lexicographic Order", *Machine Intelligence*, 3 (1969) 43-60.

[11] G. Tinhofer and H. Schreck, "Linear time tree codes," Computing, 33 (1984) 211-225.

[12] V. Vajnovszki, "Constant time generation of binary unordered trees," Bulletin of the European Association for Theoretical Computer Science, 57 (1995) 221-229.

[13] Wedderburn, "The functional equation $g(x^2) = 2\alpha x + g^2(x)$", Annals of Mathematics, 24 (1922) 121-140.

[14] H.S. Wilf, *Combinatorial Algorithms: An Update*, SIAM, CBMS 55, 1989.

[15] R.A. Wright, B. Richmond, A. Odlyzko, and B.D. McKay, "Constant Time Generation of Free Trees", SIAM J. Computing, 15 (1986) 540-548.