Generating Trees, Forests and Outerplanar graphs, uniformly at random

Asish Mukhopadhyay and Karandeep Dhillon

December 9, 2024

Abstract

We review algorithms that generate labeled free trees and forests, uniformly at random. These are based on setting up recursive counting formulas.

1 Introduction

In the next section we discuss two algorithms for generating labeled free trees, uniformly at random. The first algorithm exploits a one-to-one correspondence between Prufer sequences and labeled free trees. The second algorithm is based on a recursive formula for counting labeled trees.

2 Generating trees

Let $S = \{1, 2, \ldots, n\}$. A Prufer sequence of length $n-2$ is: $\langle a_1, a_2, \ldots, a_{n-2} \rangle$, where each $a_i \in S$. Given a labeled, free tree on n vertices with labels in S, a Prufer sequence is constructed thus:

1. Set $pf = \langle\rangle$.
2. Select a leaf u with the lowest label and add its parent v to pf.
3. Delete the edge $\{u, v\}$ from the tree.
4. If the tree has more than 2 nodes, go to step 2, else stop.

Using the algorithm described above, for the tree in Fig. 1, we obtain the following Prufer sequence, : $\langle 6, 6, 7, 7, 7 \rangle$

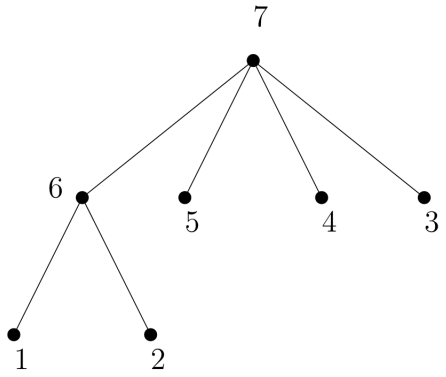

Figure 1: *A tree on 7 nodes*

We observe that only the internal nodes of the tree appear in the Prufer sequence, and a node of degree d_i appears with frequency $d_i - 1$.

We now explain the construction of a tree on n nodes from a given Prüfer sequence of length $n - 2$ over S. Consider the Prufer sequence to be a queue, and mark its last element. Each time we dequeue an element from the front of the queue, we add an edge to the tree by joining this element and the smallest element of S that is not in the queue. We also enqueue this smallest element. We stop when we have finally dequeued the marked element. A last edge is added to the tree by joining the remaining two unused elements of S (note that by now the queue has $n - 2$ distinct elements of S).

For the Prüfer sequence $\langle 6, 6, 7, 7, 7\rangle$, let the initial queue be $Q = \langle 6, 6, 7, 7, 7\rangle$. The dynamically growing tree and the corresponding queue are shown in Fig. 2.

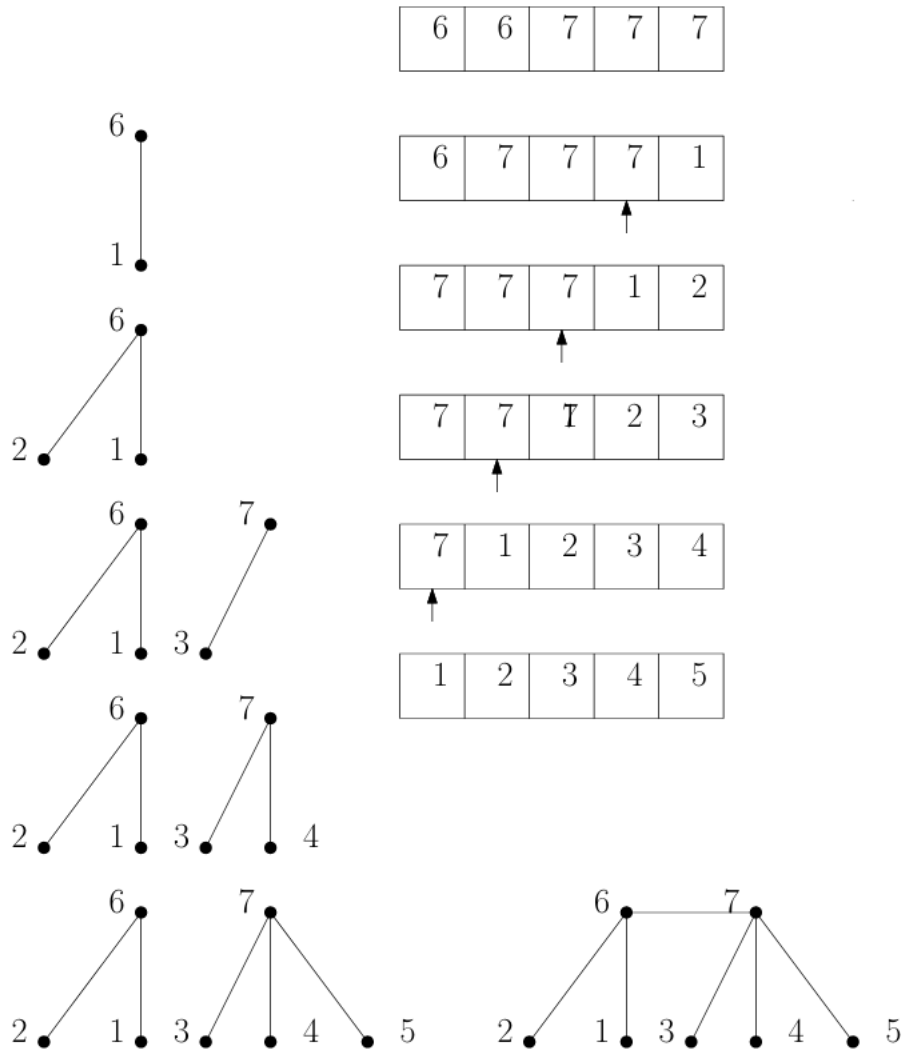

Figure 2: *Prufer sequence to tree*

This one-to-one correspondence between Prufer sequences and labeled trees on n nodes provides another proof of Cayley's theorem that the number of such trees is n^{n-2}.

3 Klingsberg's algorithm

The above algorithm is missing some details. We need to spell out how we will implement this step: "add an edge to the tree by joining this element and the smallest element of S that is not in the queue".

We maintain three sequences: a label sequence, LS, the prufer sequence, PS, and a frequency sequence, FS. The last sequence maintains the frequency with which a label appears in the Prufer sequence. In addition, we maintain a min-heap data structure that maintains the labels whose frequencies are 0, using the ordering of the labels by value for comparisons needed for min-heap operations.

The steps in the algorithm for creating the tree are as follows:

Algorithm 1 Construct a labeled tree from a Prufer Sequence

1: **Input:** LS, PS and FS
2: **Output:** Labeled Tree
3: **procedure** LABELED TREE(PS, LS, FS)
4: Intialize the frequency list, FS, by setting all values to 0
5: Go through the Prufer sequence, PS, and update the frquency of occurrence of each label.
6: Insert into a min-heap each label that has frequency value 0.
7: Remove the first element from the Prufer sequence, reduce its frequency by 1 in the frequency list
8: Remove the minimum label from the min-heap and reheapify.
9: Join the label removed from the Prufer sequence to the label deleted from the min-heap with an edge.
10: If the frequency of the element removed from the prufer sequence, PS, is reduced to 0, insert it into the min-heap.
11: Go to Step 4, unless there are only two elements left in the Prufer sequence both of which have frequency 0; join these with an edge and stop.
12: **end procedure**

Example:

We simulate the algorithm on an example.

Let LS = [1, 2, 3, 4, 5, 6, 7, 8], PS = [4, 5, 6, 6, 7, 7] and therefore FL = [0, 0, 0, 1, 1, 2, 2, 0], where LS is short for label sequence, PS is short for Prufer sequence and FL is short for frequency list.

We extract labels from the prufer sequence from left to right, marking the corresponding position with u to indicate that it has been used. From the label sequence we remove the smallest label that has frequncy 0 and join the two with an edge. Thus 4 is joined to 1. In the label sequence we mark 1 as u and reduce the frequency of 4 by 1 in the freqiuency list. If its frequency is 0m we add it to the min-heap for possible use in the next round.

The three lists for the next round are:

LS = [u, 2, 3, 4, 5, 6, 7, 8]
PS = [u, 5, 6, 6, 7, 7]
FL = [u, 0, 0, 0, 1, 2, 2, 0]

In the next round we introduce an edge between 5 and 2, and update the lists to:

LS = [u, u, 3, 4, 5, 6, 7, 8]
PS = [u, u, 6, 6, 7, 7]
FL = [u, u, 0, 0, 0, 2, 2, 0]

In the next round we introduce an edge between 6 and 3, and update the lists to:

LS = [u, u, u, 4, 5, 6, 7, 8]
PS = [u, u, u, 6, 7, 7]

FL = [u, u, 0, 0, 0, 1, 2, 0]

In the next round we introduce an edge between 6 and 4, and update the lists to:

LS = [u, u, u, u, 5, 6, 7, 8]
PS = [u, u, u, u, 7, 7]
FL = [u, u, u, u, 0, 0, 2, 0]

In the next round we introduce an edge between 7 and 5, and update the lists to:

LS = [u, u, u, u, u, 6, 7, 8]
PS = [u, u, u, u, u, 7]
FL = [u, u, u, u, u, 0, 1, 0]

In the next round we introduce an edge between 7 and 6, and update the lists to:

LS = [u, u, u, u, u, u, 7, 8]
PS = [u, u, u, u, u, u]
FL = [u, u, u, u, u, u, 0, 0]

Finally, we introduce an edge between 7 and 8 both of whose frequencies are now 0.

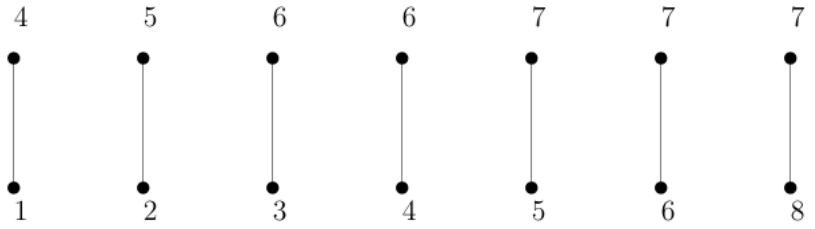

Figure 3: *Sequence of edges added*

By merging common vertices of the individual edges we get the tree shown in the following figure:

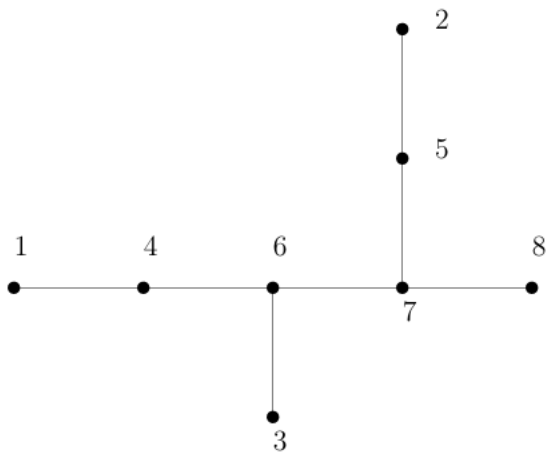

Figure 4: *Tree obtained by merging common vertices*

Klingsberg's algorithm (see Nijenhaus-Wilf text) avoids the use of a min-heap by a clever use of two pointers. We implemented the above algorithm with Kilingsberg's modification. With this change, the time-complexity of the algorithm can be shown to be linear.

4 Random tree generation, based on recursive counting

Let $t_d(n)$ be the number of labeled trees in which the vertex with label 1 has degree d. Then

$$\begin{aligned} t(n) &= \Sigma_{d=1}^{n-1} t_d(n), n \geq 2 \\ t(1) &= 1 \end{aligned} \tag{1}$$

is a count of all labeled trees on $S = \{1, 2, \ldots, n\}$.

Consider a grouping of the subtrees of 1, into the subtree that contains the vertex 2; the rest of the subtrees constitute another group. The following formula is a count of $t_d(n)$, for $d \geq 2$, $n \geq 2$.

$$t_d(n) = \Sigma_{i=1}^{n-d} \binom{n-2}{i-1} t_{d-1}(n-i) t(i) * i \tag{2}$$

The RHS of the last formula has this interpretation: We have to choose $i-1$ vertices from $n-2$ vertices to make up the complement of i vertices in the tree that contains 2. The number of trees with i vertices is $t(i)$. This reduces the degree of the vertex 1 to $d-1$ and the number of trees on $d-i$ vertices, with 1 having degree $d-1$ is $t_{d-1}(n-i)$. We have an additional factor i as 1 can be joined to any of the i vertices, including 2.

When $d = 1$ and $n \geq 2$, we have:

$$t_1(n) = t(n-1) * (n-1) \tag{3}$$

We have an additional factor $n-1$ on the RHS because 1 can be joined to any one of the $n-1$ vertices.

We note that $t_1(1)$ and $t_2(2)$ are both undefined.

This gives us the following random tree generation algorithm.

Algorithm 2 Construct a random labeled tree on S

Input: $S = \{1, 2, \ldots, n\}$ ▷ $n \geq 2$
Output: Random Tree on S
procedure LABELEDTREE(n, S) ▷ returns random tree on S
    if $n \leq 0$ **then**
        return $\emptyset$
    else
        Choose degree d of vertex $minLabel(S)$ with probability $t_d(n)/t(n)$. ▷ The tables $t_n(d)$ and $t(n)$ are global
        return LabeledTree(n, d, S)
    end if
end procedure
procedure LABELEDTREE(n, d, S) ▷ returns random tree on S, where $minLabel(S)$ has degree d
    if $d = 1$ **then**
        $Sdash \leftarrow S - minLabel(S)$
        T = LabeledTree($n - 1, Sdash$)
        return $T \cup \{minLabel(S), i\}$ ▷ i is any vertex in the set $S - minLabel(S)$
    else ▷ $d \geq 2$
        Choose the size i of the split subtree with probability $\binom{n-2}{i-1} t_{d-1}(n-i)t(i) * i/t_d(n)$
        Choose random subset $\{w_1, w_2, \ldots, w_i\} \subseteq S - minLabel(S)$ of size i
        $Sdash = \{1, 2, \ldots, i\}$
        T_1 = LabeledTree($i, Sdash$); relabel vertex j in T_1 to w_j
        $Sdash = S - \{w_1, w_2, \ldots, w_i\}$
        T_2 = Generate($n - i, d - 1, Sdash$)
        return $T_1 \cup T_2 \cup \{minlabel(S), w_1\}$
    end if
end procedure

4.1 Computation of $t(n)$ and $t_d(n)$

These tables are needed for computing probabilities and have to be set-up before the random-tree algorithm can be used.

Equations (1), (2) and (3) are used for this. As we can see, these are tightly coupled. We store the values of $t(n)$ in a one-dimensional array and those of $t_d(n)$ in a two dimensional array, indexed by the values of n and d.

We calculated the entries of these two tables for $n = 5$ and $d = 5$, shown below.

n	1	2	3	4	5
$t(n)$	1	1	3	16	125

$n\backslash d$	1	2	3	4	5
1	$\times$				
2	1	$\times$			
3	2	1	$\times$		
4	9	6	1	$\times$	
5	64	48	12	1	$\times$

The following algorithm was used to fill up the tables.

The $t_d(n)$ entries are filled in row-wise from top to bottom and left to right in each row; the $t(n)$ entries are filled in from left right, using Eqn. (1). An entry $t(n)$ is determined using Eqn. (2) after all the entries in a row of the $t_d(n)$ table are determined. The first entry in a row of the $t_d(n)$ table is determined using Eqn. (3).

4.2 Sampling from a given discrete probability distribution

Define a probability distribution on the set $\{x_1, x_2, \ldots, x_n\}$ as follows. Let X be a discrete random variable. For $i = 1, 2, \ldots, n$, let

$$P(X = x_i) = p_i$$

To sample from this discrete distribution, we take the help of a pseudo-random generator to generate a value x uniformly at random in the interval [0,1]. We now determine an i such that:

$$p_1 + p_2 + \cdots + p_{i-1} < x < p_1 + p_2 + \ldots + p_i$$

and thus choose x_i with probability p_i.

For a given n, we compute the probabilities $t_d(n)/t(n)$, for $d = 1, \ldots, n-1$, where $t(n)$ is obtained from the array t and the values $t_d(n)$ from the table of values for $t_d(n)$ from the row of entries coprresponding to this n. Now, choose a d according to the above method.

Next, given values of n and d, we choose an i by computing the probabilities $p_i = \binom{n-2}{i-1} t_{d-1}(n-i)t(i) * i/t_d(n)$, for $i = 1, 2, \ldots, n-d$ and selecting an i according to the above scheme.

I have written code to implement the algorirhm described.

5 Generation of a random forest

An algorithm similar to generating a random labeled tree can be devised for generating a random labeled forest, using suitable counting formulas. Let $f(n)$ be a count of the number of forests on the label set S. Then,

$$f(n) = \Sigma_{c=1}^{n} f_c(n), \tag{4}$$

where $f_c(n)$ is a count of the number of forests with c trees.

When $c = 1$, we have $f_1(n) = t(n)$. For $c \geq 2$, we have the following recursive counting formula. Considering a partition of the c trees into a tree with i vertices that includes vertex 1, with the remaining $n - i$ vertices distributed among the remaining $c - 1$ trees.

$$f_c(n) = \Sigma_{i=1}^{n} \binom{n-1}{i-1} f_1(i) f_{c-1}(n-i) \tag{5}$$

The factor $\binom{n-1}{i-1}$ accounts for the number of ways of choosing $i - 1$ vertices out of $n - 1$ to account for the complement of i vertices that make up the tree containing the vertex 1.

Interpreting both Eqn.(4) and Eqn.(5) as a sum of probabilitites by dividing the right-hand side by the left-hand side in these equations, we have the following algorithm for generating a random forest.

Algorithm 3 Construct a random forest on S

Input: $S = \{1, 2, \ldots, n\}$ ▷ $n \geq 2$
Output: Random Labeled Forest on S
procedure RANDOMLABELEDFOREST(n, S) ▷ returns random labeled forest on S
    if $n \leq 0$ **then**
        return $\emptyset$
    else
        Choose the number of trees c with probability $f_c(n)/f(n)$. ▷ $c \geq 1$
        return RandomLabeledForest(n, c, S)
    end if
end procedure
procedure RANDOMLABELEDFOREST(n, c, S) ▷ returns random labeled forest on S, with c trees
    if $c = 1$ **then**
        T = RandomLabeledTree(n, S) ▷ Invoke random tree generation algorithm
        return T
    else
        Choose the size i of the tree containing the vertex minLabel(S) with probability $\binom{n-1}{i-1} f_{c-1}(n-i) f_1(i) / f_c(n)$
        w_1 = minLabel(S)
        Choose random subset $\{w_2, \ldots, w_i\} \subseteq S - \{w_1\}$ of size $i - 1$
        $S_w = \{w_1, w_2, \ldots, w_i\}$
        $Sdash = S - S_w$
        T = RandomLabeledTree(i, S_w); ▷ Invoke random tree generation algorithm
        F = RandomLabeledForest($n - i, c - 1, Sdash$)
        return $T \cup F$
    end if
end procedure

Now, we can proceed to build tables for $f(n)$ and $f_c(n)$ exactly as in the case of trees in order to determine the relevant probabilities in the forest generation algorithm above (We have written code that does this).

6 Time Complexity Evaluation for Random Forest Generation

The random forest generation algorithm's time complexity is analyzed as follows:

Base Case: If $n \leq 0$, the function terminates in $O(1)$.

Recursive Case: The algorithm selects c (number of trees) with probability $f_c(n)/f(n)$, which is assumed to take $O(1)$ time if probabilities are precomputed or efficiently calculated. For each c, the algorithm:

- Invokes the labeled tree generation algorithm for the tree rooted at the minimum label.
- Recursively generates the forest on the remaining subsets of S, reducing the size by i.

Key Steps:

- **Probability Calculation:** Selecting the size c and subsets involves $O(1)$ operations if probabilities are efficiently handled.
- **Subset Selection:** For $c > 1$, subsets S_w of size $i - 1$ are selected in $O(n)$ operations for n elements.
- **Tree Generation:** Tree generation is determined by Algorithm 2. The random labeled tree algorithm involves recursive calls where the set S is split into smaller subsets. Additionally, generating a labeled tree involves computing probabilities and using precomputed tables for probabilities like $t_d(n)/t(n)$.

Table Computation Complexity: Each table entry requires $O(n \log n)$ bits to store due to the size of n^{n-2}. Computing an entry involves multiplying two n-bit numbers, which has a bit complexity of $O(n \log n \log \log n)$. For $O(n)$ such multiplications per entry, the bit complexity becomes $O(n^2 \log n \log \log n)$. Since the table has $O(n^2)$ entries, the total complexity for computing the table is:

$$O(n^4 \log n \log \log n)$$

Recursive Forest Generation: The recurrence relation for the forest generation is given by:

$$T_f(n) = \sum_{c=1}^{n} P(c) \cdot [T_t(n) + T_f(n-i)]$$

where:

- $T_t(n)$ is the time complexity of labeled tree generation, $O(n^4 \log n \log \log n)$.
- $T_f(n-i)$ represents recursive forest generation for subsets of size $n - i$.

At each recursion level, the dominant cost comes from $T_t(n) = O(n^4 \log n \log \log n)$, and the recursive calls for forest generation add overhead, but tree generation dominates.

Final Time Complexity: Combining all contributions, the dominant term for the random labeled forest generation algorithm is determined by the table computations and tree generation. Hence, the overall worst-case time complexity is:

$$O(n^4 \log n \log \log n)$$

References

[1] A. Nijenhuis and H.S. Wilf. *Combinatorial Algorithms for Computers and Calculators*. Academic Press, New York, 1978.

[2] Alexey S. Rodionov and Hyunseung Choo. On generating random network structures: Trees. In *Computational Science - ICCS 2003, International Conference, Melbourne, Australia and St. Petersburg, Russia, June 2-4, 2003. Proceedings, Part II*, pages 879–887, 2003.

[3] Alexey S. Rodionov and Hyunseung Choo. On generating random network structures: Connected graphs. In *Information Networking, Networking Technologies for Broadband and Mobile Networks, International Conference ICOIN 2004, Busan, Korea, February 18-20, 2004, Revised Selected Papers*, pages 483–491, 2004.

[4] G. Tinhofer. *Generating Graphs Uniformly at Random*, pages 235–255. Springer Vienna, Vienna, 1990.

[5] Herbert S. Wilf. The uniform selection of free trees. *J. Algorithms*, 2(2):204–207, 1981.

As described in [2], random tree generation is important for analyzing network structures.
According to [3], connected graphs can be generated using recursive methods.
The work in [4] explores the uniform generation of graphs, focusing on various graph classes.
As noted in [1], combinatorial algorithms play a crucial role in generating and analyzing structures like labeled trees.
In [5], the uniform selection of free trees is discussed, providing foundational insights into tree generation methods.