Generating rooted trees of \( m \) nodes uniformly at random

Kenneth Matheis and Spyros S. Magliveras

CCIS, Department of Mathematical Sciences, Florida Atlantic University,
777 Glades Road, Boca Raton, FL 33431
kmatheis@fau.edu, spyros@fau.edu

Abstract. A rooted tree is an ordinary tree with an equivalence condition: two trees are the same if and only if one can be transformed into the other by reordering subtrees. In this paper, we construct a bijection and use it to generate rooted trees (or forests) of any specified nodecount \( m \) uniformly at random. As an application, we see that in [6] Raddum and Semaev propose a technique to solve systems of polynomial equations over \( \mathbb{F}_2 \) as occurring in algebraic attacks on block ciphers. This approach is known as MRHS. In [3] Geiselmann, Matheis, and Steinwandt propose an ASIC hardware design to implement MRHS, and they show that the use of ASICS seems to enable significant performance gains over a software implementation of MRHS. What hasn’t been asserted is the total time complexity of their platform, though individual components’ runtimes are provided. If one supposes that deletions in MRHS occur as rooted trees generated uniformly at random, then one application of the proposed algorithm would be to contribute to such a time complexity; experiments are generated to provide statistical averages of key quantities.

Keywords. rooted tree, rooted forest, uniform random generation, genetic programming, MRHS, PET SNAKE

Introduction

We view a rooted tree as an equivalence class of ordinary trees, where two trees are equivalent if one can be transformed into the other by re-ordering subtrees [7]. Similarly, we view a rooted forest as an equivalence class of forests, where two rooted forests are equivalent if one can be transformed into the other by re-ordering the rooted trees. Alternately, we may consider a rooted forest as nothing more than the subtrees of a rooted tree (of one node more) whose root is hidden.

The idea of a rooted tree has been around since 1875 [2] when countings for smaller nodecounts have been computed. Since then there have been a few proposals constructing bijections between all rooted trees and \( \mathbb{N} \).

For each \( m \in \mathbb{N} \), we define \( T_m \) to be the set of rooted trees of \( m \) nodes, and \( F_m \) to be the set of rooted forests of \( m \) nodes. Our contribution is an implicit construction of a bijection between \( F_m \) and \( \mathbb{Z}/|F_m| \). Such a bijection can then be used to generate a rooted forest of \( m \) nodes uniformly at random.

As an immediate application to cryptography, we note that some statistics generated from these trees can be used to help calculate a time estimate for PET SNAKE. Now, In
[6] Raddum and Semaev proposed a technique known as MRHS (Multiple Right Hand Sides) to handle polynomial systems of equations over $\mathbb{F}_2$. This algorithm is particularly well-suited for describing systems of equations for an algebraic key recovery attack against common block ciphers such as AES or DES, but a complete time estimate of it was not forthcoming. Later, the hardware platform PET SNAKE [3] was designed to implement MRHS attacks in hardware, but PET SNAKE’s time estimate is also hard to calculate. The statistics mentioned can help contribute to such a time estimate.

**Related Work**

It has been established that we can map rooted trees to natural numbers [5] and that there is a rooted tree for every natural number [4]. If one relaxes the equivalence condition and merely examines arbitrary trees, then a uniform random generation algorithm is known [1] by modeling them using a context-free grammar for use in a genetic algorithm. However, it is not clear that it is possible to create rooted trees using a context-free grammar, so we do not use this algorithm. We instead develop a different algorithm which, as it happens, shares some features with the one in [1].

Further, much information about rooted trees is available in [8, sequence A000081], and some of those facts will be used in this paper.

**Structure of the Paper**

We first discuss the construction of the bijection between $F_m$ and $\mathbb{Z}_{|F_m|}$. Once this is established, we review the relevant details about MRHS and show how $F_m$ is related to the processing of PET SNAKE (notably the deletion count therein). Finally, we generate some statistics based on $F_m$ for $m \leq 1000$ and relate those to time estimate processing for PET SNAKE.

**1. Generating Rooted Forests Uniformly at Random**

We begin with some notation: we define the natural numbers $\mathbb{N}$ to be $\{1, 2, 3, \ldots \}$, the whole numbers $\mathbb{W}$ to be $\mathbb{N} \cup \{0\}$, and for each $n \in \mathbb{N}$, we define $\text{seg}_n$ to be $\{1, 2, \ldots, n\}$.

In order to generate a rooted forest of $m$ nodes uniformly at random, we first construct some data tables dynamically (so that no unneeded space is allocated), and then we perform many lookups on those tables.

We view a rooted forest of $m$ nodes as being constructed by a collection of $r$ rooted trees $a_1, a_2, \ldots, a_r$, for some $r \in \text{seg}_m$, with respective nonincreasing node counts $c_1, c_2, \ldots, c_r$ such that $\sum c_i = m$.

We then construct sequences of counts $b_i$ such that $b_{11}, b_{12}, \ldots, b_{1s_1}$ are the $s_1$ counts starting with $c_1$ that are equal to $c_1$ and $b_{21}, b_{22}, \ldots, b_{2s_2}$ are the $s_2$ counts starting with $c_1+s_1$ that are equal to $c_1+s_1$, and so on, and we suppose there are $d$ such sequences. This breaks up the counts into subsequences of equal-valued terms. For example, if the counts $c$ were 9, 8, 8, 8, 7, 7, 7, 6, 4, 3, 3, 3, 3, 2, 1, 1, 1, then $b_1$ has one term (namely 9), $b_2$ has three terms (all of which are 8), $b_3$ has two terms (both of which are 7), $b_4$ has one term (namely 6), and so on, ending with $b_8$ having three terms (all of which are 1) and $d = 8$. 
Since we envision the trees $T_k$ (for any $k \in \mathbb{N}$) as being ordered, for each $i \in \text{seg}_m$ we must count the number of ordered arrangements of $s_i$ trees in $T_{b_i}$. Call this number $B_i$. We then calculate the number of rooted forests with this count sequence as $\prod B_i$.

In order to correlate a number in $\mathbb{Z}|F_m|$ to a forest in $F_m$, we must have a way to obtain the number of forests of subtrees with any nonincreasing count sequence. As one might imagine, this is done recursively using the building blocks described below.

### 1.1. Setup

The setup phase of the algorithm consists of building three tables. First, for each $i \in \text{seg}_m$, $|T_i|$ is calculated using the recurrence formula

$$|T_i| = \frac{1}{i-1} \sum_{k=1}^{i-1} \left( \sum_{d \mid k} d \cdot |T_d| \right) |T_{i-k}|$$

with $|T_1| = 1$ [8]. This takes $O(i^2)$ time for each $i$, totalling a time of $O(m^3)$.

Then an $m \times m$ table $R$ called the runtable is created. Its purpose is to store forest counts in the following way: for any two $i, j \in \text{seg}_m$, $R_{ij}$ is the number of sequences $u : \text{seg}_j \to \text{seg}_i$ such that $u_1 \leq u_2 \leq \cdots \leq u_j$; in other words, it is the number of non-decreasing $j$-length sequences of $i$-node rooted trees. (Note that we have not mentioned how to order the trees $T_i$, but certainly one exists. Indeed, a side effect of this process is to construct the bijections $f_m : \mathbb{Z}|F_m| \overset{1-1}{\longrightarrow} F_m$, which in turn constructs the bijections $t_m : \mathbb{Z}|T_m| \overset{1-1}{\longrightarrow} T_m$, so for two trees $p$ and $q$, $p \leq q$ if and only if the index that constructs $p$ is less than or equal to the index that constructs $q$. Since $R$ simply concerns itself with the number of sequences $u$, and not the individual sequences themselves, we run into no difficulty constructing $R$.)

To calculate these values, we take advantage of the following theorem:

**Theorem 1**

$$(\forall n \in \mathbb{N})(\forall k \in \mathbb{N}) \left[ \sum_{i=1}^{n} \binom{i+k}{k+1} = \binom{n+k+1}{k+2} \right]$$

To prove this, simply use induction on “$n$”. Five lines are all that are necessary. ■

We now make an observation about sequences of finite length. (To be clear, we make no claim about the originality of Theorems 1 and 2, but absent appropriate references, proofs are provided to justify their correctness.)

**Theorem 2** For each $i, j \in \mathbb{N}$, let $S_{ij}$ be the number of nondecreasing sequences $u : \text{seg}_j \to \text{seg}_i$. Then

$$(\forall j \in \mathbb{N})(\forall i \in \mathbb{N}) \left[ S_{ij} = \binom{i+j-1}{j} \right].$$
To prove this, we proceed by induction on \( j \).

\( \varphi(1) \): Let \( i \in \mathbb{N} \). Then \( S_{i1} \) is the number of nondecreasing sequences \( u : \text{seg}_1 \to \text{seg}_i \). But there are only \( i \) such things, one for each choice of \( u_1 \). Hence, \( S_{i1} = i = {i \choose 1} = (i+1-1) \). Thus, \( \varphi(1) \).

Let \( k \in \mathbb{N} \) and assume \( \varphi(k) \): (\( \forall i \in \mathbb{N} \)) \[ S_{ik} = {i+k-1 \choose k} \].

\( \varphi(k+1) \): Let \( i \in \mathbb{N} \). Consider \( S_{i(k+1)} \). These are all the nondecreasing sequences \( u : \text{seg}_{k+1} \to \text{seg}_i \). Now let us consider the possibilities for \( u_1 \). If \( u_1 = 1 \), then the remaining terms comprise a \( k \)-length nondecreasing sequence to \( \text{seg}_i \). But the number of such sequences is just \( S_{ik} \). Now, if \( u_1 = 2 \), the remaining terms comprise a \( k \)-length sequence to \( \text{seg}_i(1) \). But the number of such sequences is the same as the number of \( k \)-length sequences to \( \text{seg}_i(1) \) (just subtract 1 from each term), which is \( S_{(i−1)k} \). Similarly, for each \( v \in \text{seg}_i \), if \( u_1 = v \), then the remaining terms comprise a \( k \)-length sequence to \( \text{seg}_i(−v−1) \), whose count is the same as the count of \( k \)-length sequences to \( \text{seg}_i(−v−1) \) (by subtracting \( v−1 \) from each term), which is \( S_{(i−v+1)k} \). Hence,

\[
S_{i(k+1)} = S_{ik} + S_{(i−1)k} + \cdots + S_{1k}
\]

\[
= \sum_{v=1}^{i} S_{vk} \]

\[
= \sum_{v=1}^{i} {v+k−1 \choose k} \quad \text{by Ind. Hyp.}
\]

\[
= {i+k \choose k+1} \quad \text{by Theorem 1}
\]

Thus, \( \varphi(k+1) \). The rest follows by the Principle of Mathematical Induction and routine steps.

Since, for each \( j \in \text{seg}_m \), \( R_{ij} \) is the number of nondecreasing sequences from \( \text{seg}_j \) to \( T_i \), this is the same as the number of nondecreasing sequences from \( \text{seg}_j \) to \( \text{seg}_{|T_i|} \), which is \( \left( \left( |T_i| - 1 \right) + j - 1 \right) \) by Theorem 2. Hence, we build the \( R \) table by populating it with this binomial coefficient for each \( i \in \text{seg}_m \) and \( j \in \text{seg}_m \) such that \( j \leq \left\lfloor \frac{m}{2} \right\rfloor \); \( j \) is restricted in this way since, for any choice of tree size \( i \) in an \( m \)-node forest, you can only have at most \( \left\lfloor \frac{m}{2} \right\rfloor \) such trees. As a side effect, we see that \( |T_i| \) is stored in \( R_{1i} \) by this process.

As a point of interest, note that we had to use this binomial simplification when populating the \( R \) table. Otherwise, since \( |T_i| \) is asymptotically \( 0.4399 \cdot 2.9558^{i} \cdot i^{-3/2} \) [8, sequence A000081], asking the computer to perform the sum listed in the proof of Theorem 2 would become infeasible very quickly.

We construct two more tables, the two-dimensional \textit{partable} denoted \( P \), and the three-dimensional \textit{lentable} denoted \( L \). For each \( i, j \in \text{seg}_m \), \( P_{ij} \) is the number of rooted forests of \( i \) nodes whose first tree has \( j \) nodes. It could be that the first few trees have \( j \) nodes, so we keep track of this using the lentable: \( L_{ijk} \) is the number of rooted forests of \( i \) nodes whose first \( k \) trees have \( j \) nodes. To calculate \( L_{ijk} \), we use
$$L_{ijk} = \begin{cases} R_{jk} & \text{if } i - jk = 0 \\ \sum_{q=1}^{j-1} P_{(i-jk)q} & \text{otherwise} \end{cases}$$

Then,

$$P_{ij} = \sum_{k=1}^{\lfloor \frac{i}{j} \rfloor} L_{ijk}.$$ 

Finally, we recognize that $|F_m| = |T_{m+1}|$ gives us no intuitive breakdown of all the counts, but

$$|F_m| = \sum_{j=1}^{m} P_{mj}$$

does. Note that, though we concern ourselves with how a given number of nodes $m$ breaks down into each partition of $m$, this setup prevents us from having to loop through each partition of $m$, which also would be infeasible very quickly.

We remark that the storage for $R$ is $O(m^2)$ but is significantly less than $m^2$ since, for each $i \in \text{seg}_m$, we only populate $R_{ij}$ when $j \leq \lfloor \frac{m}{i} \rfloor$. Further, for similar reasons the storage for $L$ is $O(m^3)$, but is significantly less than $m^3$.

### 1.2. Teardown

In order to generate a forest in $F_m$ uniformly at random, we first generate a number $r$ in $\mathbb{Z}_{|F_m|}$ (called an index) uniformly at random. Then, we go through the process of whittling down $r$ by successively discovering which count sequence to use for that forest, and which indices to use for each tree of that forest. (Such data collectively is called a decomposition of the index $r$.) After the decomposition is constructed, we recur on each tree size of the decomposition, noting that if the $i$th tree has $c_i$ nodes, it can be viewed as a forest (of its subtrees) of $c_i - 1$ nodes, the root itself being one node. The recursion terminates when we are faced with generating a forest of one node with index zero, at which point we return a leaf.

#### 1.2.1. Composing Decompositions

For any forest of $n$ nodes whose first tree can have as many as $h$ nodes (called the head size), composing a decomposition is itself a recursive process which relies on three algorithms which we call PTCORDS, LENREM, and RUNCOORDS. The process produces two vectors, sizes and idxs.

PTCOORDS identifies which column of $P_n$ that $r$ is in (say it’s the $j$th) and reduces the index and provides a new head. LENREM identifies which tower of $L_{nj}$ that the reduced index is in (say it’s the $k$th) and produces a re-reduced index, remainder index, and a remaining node count for subsequent trees. RUNCOORDS converts the re-reduced
index into a sequence of indices for each of the $k$ trees. We then recur on the remaining node count, the new head minus 1, and the remainder index, and we append a sequence of $k$ entries of $j$ to the front of the result’s sizes, and also the sequence of indices to the front of the result’s idxs.

This process is started with a call to \textsc{decomp}(m, m, r).

Algorithm 1 \textsc{decomp}

\begin{algorithm}
\begin{algorithmic}
\Require A nodecount $n$, a head $h$, an index $r \in \mathbb{Z}_{|F_n|}$.
\State set sizes and idxs to be empty lists
\If{$n \leq 0$ or $h < 1$}
\State return (sizes, idxs)
\ElseIf{$h > n$}
\State $h \leftarrow n$
\EndIf
\State $(r', h') \leftarrow \text{ptcoords}(n, h, r)$
\State $(k, n', r'', x) \leftarrow \text{lenrem}(n, h', r')$
\State \text{frontidxs} $\leftarrow$ \text{runcoords}(h', r'', k)
\State set \text{frontsizes} to be a list of $k$ copies of $h'$
\State $(\text{backsizes}, \text{backidxs}) \leftarrow \text{decomp}(n', h' - 1, x)$
\State return (append(frontsizes, backsizes), append(frontidxs, backidxs))
\end{algorithmic}
\end{algorithm}

Algorithm 2 \textsc{ptcoords}

\begin{algorithm}
\begin{algorithmic}
\Require A nodecount $n$, a head $h$, an index $r \in \mathbb{Z}_{|F_n|}$.
\State $r' \leftarrow r$, $h' \leftarrow h$
\While{$P_{nh'} \leq r'$}
\State $r' \leftarrow r' - P_{nh'}$
\State $h' \leftarrow h' - 1$
\EndWhile
\State return $(r', h')$
\end{algorithmic}
\end{algorithm}

\textsc{lenrem} sends two of its outputs to \textsc{runcoords}, which uses a binary search to determine what the indices should be for each of the $k$ trees of nodesize $h'$, based on the re-reduced index $r''$. This approach is needed since $r''$ is an index into one of the $(|T_{h'}|+k-1)$ nondecreasing sequences from $\text{seg}_k$ to $\text{seg}|T_{h'}|$, but this quantity is a sum (as per Theorem 1), so we have to figure out where $r''$ is in that sum without examining $|T_{h'}| - 2$ individual binomial coefficients, as $|T_{h'}|$ can get very large. (Indeed, this is the part that is significantly different than the uniform random generation algorithm in [1].)

We remark in passing that, after the Setup phase, building a rooted forest of $m$ nodes corresponding to an index $r$ takes slightly more than $O(m)$ time but definitely within $O(m^2)$ time.
2. Application to MRHS and PET SNAKE

Now that we have a reliable method to generate rooted forests uniformly at random, one application would be to compute relevant statistics from them to help predict PET SNAKE’s run time. We recall the relevant facts about MRHS and PET SNAKE. MRHS operates on a collection of pairs of matrices called symbols, and one phase of its processing is called the Agreement Phase, where each symbol must be agreed to each other symbol. Sometimes the act of agreeing a pair of symbols induces a deletion in one (or both) symbols; other times, nothing changes. If a deletion occurs, then the process starts over: each symbol must be re-agreed to each other symbol. This continues until no (more) deletions are detected, at which point the symbols are said to be pairwise agreed. Hence, for a body of $n$ symbols, at least $\binom{n}{2}$ agreements must be performed. In software, each agreement must be performed one at a time. PET SNAKE is a hardware design employing lots of processors, and it uses them to perform half of the $\binom{n}{2}$ agreements simultaneously. If no deletion is detected, it then performs half of the remaining agreements simultaneously. And so on.

Since some deletions cannot occur until other deletions occur first, we choose to model the deletions as a collection of rooted trees. In each tree, each node symbolizes a deletion after two symbols are agreed, and each child of a node symbolizes deletions that can now occur as a result of the parent node’s deletions taking place. In the beginning of an agreement phase, it is certainly possible that many deletions do not depend on each other, so these deletions are the roots of the trees in this collection.

We observe that the order of subtrees of a given node is irrelevant; it does not matter which subtree is the first subtree, which is the second, and so on; hence the choice of a rooted forest is appropriate. We notice that at any stage, PET SNAKE will perform half of the agreements necessary simultaneously, so at any point, about half of the deletions that can be performed will be performed on average.

Now, if a deletion gets performed, then that deletion’s children will then be available to be deleted. Examining the consequences for the model, we see that only the roots of the trees in the forest are available for deletion, so when such a deletion is performed, the corresponding root must be eliminated. This, however, means that that root’s children are now roots in the forest. This operation of deleting a root and promoting its children we call a lift.

**Algorithm 3 LENREM**

**Require:** A nodecount $n$, a new head $h'$, a reduced index $r'$.

1. $k \leftarrow \lfloor n/h' \rfloor$
2. while $L_{n h'} k \leq r'$ do
3. $r' \leftarrow r' - L_{n h'} k$
4. $k \leftarrow k - 1$
5. end while
6. $n' \leftarrow n - (k \cdot h')$
7. $c \leftarrow L_{n h'} k / R_{h'} k$
8. $r'' \leftarrow \lfloor r'/c \rfloor$
9. $x \leftarrow r'' \mod c$
10. return $(k, n', r'', x)$
Algorithm 4 \textsc{runcoords}

\textbf{Require:} A new head $h'$, a re-reduced index $r''$, a length $k$.

1: set $idxs$ to be an empty list
2: $top \leftarrow R_{h'}$, $prev \leftarrow 0$
3: for $t \in \{k, k - 1, \ldots, 2\}$ by $-1$ do
4: \hspace{1em} $i \leftarrow top + 1$, $j \leftarrow 1$, $mid \leftarrow \lfloor (i + j)/2 \rfloor$
5: \hspace{1em} $total \leftarrow (top + t - 1)/t$, $pen \leftarrow total - 1 - r''$
6: \hspace{1em} $found \leftarrow false$
7: \hspace{2em} while not $found$ do
8: \hspace{3em} $c_1 \leftarrow (mid + t - 1)/t > pen$, $c_2 \leftarrow pen \geq (mid + t - 2)/t$
9: \hspace{3em} $found \leftarrow (c_1$ and $c_2)$
10: \hspace{2em} if not $found$ then
11: \hspace{3em} if $c_1$ then
12: \hspace{4em} $i \leftarrow mid$
13: \hspace{3em} else
14: \hspace{4em} $j \leftarrow mid$
15: \hspace{3em} end if
16: \hspace{3em} $mid \leftarrow \lfloor (i + j)/2 \rfloor$
17: \hspace{2em} end if
18: \hspace{1em} $prev \leftarrow top - mid + prev$
19: \hspace{1em} insert $prev$ onto the back of the list $idxs$
20: $top \leftarrow mid$, $r'' \leftarrow r'' - (total - (mid + t - 1)/t))$
21: end for
22: insert $r'' + prev$ onto the back of the list $idxs$
23: return $idxs$

Hence, about half of the roots are lifted in a given stage. (Such an action we will refer to as a parallel lift.) The agreement phase is not complete until all the nodes in the forest are eliminated. To get a handle on time estimates, it is pertinent to ask how many roots exist at a given time, and how many times to we expect to perform parallel lifts until the forest is eliminated. Since we do not have theoretical answers to these questions, we assume that the $m$ deletions in an agreement phase occur as a forest of $m$ nodes chosen uniformly at random. With this assumption, we design an experiment as follows: for various $m \leq 1000$, we perform the Experimental Procedure (see Figure 1) several times (say, $s$ times). Throughout each procedure run, we count the number of roots that the forest has (once before each parallel lift) so as to calculate the average when the forest is eliminated, and we also count the number of times we have to parallel lift.

Once the number of parallel lifts and the average number of roots are calculated, we do it again for the same forest. This is repeated $s$ times. Once these $s$ procedures are complete, we choose another rooted forest of $m$ nodes uniformly at random and perform the procedure again. We construct $t$ such forests (each giving rise to $s$ procedures), and a global average of number of parallel lifts required and number of roots appearing at any point are calculated.

This procedure was performed for $s = t = 1000$ and $m \in \{50, 100, 150, \ldots, 1000\}$ and the results are summarized in Table 1.
• Construct a rooted forest of \( m \) nodes uniformly at random.
• While it is nonempty,
  * take note of the number of roots of the forest,
  * uniformly at random choose half of the roots, and
  * lift them from the forest.
• Calculate the average number of roots the forest had.

Figure 1. Experimental Procedure

Table 1. Experimental Procedure Results (\( n = t = 1000 \))

<table>
<thead>
<tr>
<th>( m )</th>
<th>Avg parallel lifts</th>
<th>Avg roots</th>
<th>( m )</th>
<th>Avg parallel lifts</th>
<th>Avg roots</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>25.4741</td>
<td>3.9869</td>
<td>550</td>
<td>100.283</td>
<td>11.4762</td>
</tr>
<tr>
<td>100</td>
<td>38.7107</td>
<td>5.3268</td>
<td>600</td>
<td>105.187</td>
<td>11.9226</td>
</tr>
<tr>
<td>150</td>
<td>49.2455</td>
<td>6.330</td>
<td>650</td>
<td>109.466</td>
<td>12.4351</td>
</tr>
<tr>
<td>200</td>
<td>56.9224</td>
<td>7.3193</td>
<td>700</td>
<td>112.619</td>
<td>13.01</td>
</tr>
<tr>
<td>250</td>
<td>65.1864</td>
<td>7.9930</td>
<td>750</td>
<td>119.717</td>
<td>13.1435</td>
</tr>
<tr>
<td>300</td>
<td>71.7676</td>
<td>8.7246</td>
<td>800</td>
<td>123.128</td>
<td>13.6412</td>
</tr>
<tr>
<td>350</td>
<td>78.5635</td>
<td>9.3096</td>
<td>850</td>
<td>125.295</td>
<td>14.2402</td>
</tr>
<tr>
<td>400</td>
<td>83.6236</td>
<td>10.0201</td>
<td>900</td>
<td>129.423</td>
<td>14.5746</td>
</tr>
<tr>
<td>450</td>
<td>89.7707</td>
<td>10.4778</td>
<td>950</td>
<td>133.577</td>
<td>14.9439</td>
</tr>
<tr>
<td>500</td>
<td>94.8623</td>
<td>11.0328</td>
<td>1000</td>
<td>135.625</td>
<td>15.4812</td>
</tr>
</tbody>
</table>

If we multiply the average roots by the average parallel lifts and plot this result for all twenty pairs, we discover that the plot forms a near-straight line of slope approximately \( \frac{40}{17} \). This isn’t too surprising, since in each parallel lift we eliminate about half the roots, and the roots multiplied by the parallel lifts (if we eliminated every root per lift) should give us the total number of nodes in the forest. Further, if we multiply the number of roots by itself and plot this, we get a near-straight line of slope approximately \( \frac{9}{38} \). From these two observations, we propose the following:

**Proposition 1** A rooted forest of \( m \) nodes chosen uniformly at random will have, on average, \( \sqrt{\frac{2}{3\pi}m} \approx 0.4866\sqrt{m} \) roots on average as its corresponding set of deletions get deleted through an agreement phase.

Further, the number of parallel lifts required to eliminate such a forest is on average approximately \( \frac{40}{17}m/0.4866\sqrt{m} \approx 4.3264\sqrt{m} \).

These estimates can be used in conjunction with estimates of how many deletions to expect per agreement phase to help predict the runtime of PET SNAKE.

As a point of interest, if we choose not to lift half of the roots, but instead all of them, we can use a similar procedure to determine the average depth and the average number of nodes per depth for these trees.
3. Conclusion

We have provided a way to implicitly construct bijections between $T_m$ and $Z_{|T_m|}$, and between $F_m$ and $Z_{|F_m|}$ with reasonable time and space consumption, for any $m \in \mathbb{N}$, and we hope that this proves useful in many environments. One such environment is in the realm of cryptography, where we aid in the construction of a time estimate for a hardware platform implementing an algebraic attack on block ciphers. Another might be in genetic programming to create initial trees corresponding to non-context-free grammars.

References