# Generating Random Spanning Trees More Quickly than the Cover Time

David Bruce Wilson*
dbwilson@mit.edu
Department of Mathematics,
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

## 1. Introduction

It is widely known how to generate random spanning trees of an undirected graph. Start at any vertex and do a simple random walk on the graph. Each time a vertex is first encountered, mark the edge from which it was discovered. When all the vertices are discovered, the marked edges form a random spanning tree. This algorithm is easy to code up, has small running time constants, and has a nice proof that it generates trees with the right probabilities.

This paper gives a new algorithm for generating random spanning trees. It too is simple, easy to code up, and has nice proofs. The new algorithm also has the following advantages:

- On graphs for which the old algorithm works, the new algorithm is never slower (by more than a factor of two), and is usually much faster.
- It also works for directed graphs.
- It yields the current fastest algorithm for sampling from the stationary probability distribution of a Markov chain whose transition probabilities are unknown.

Additionally, the proofs double as proofs for an identity and an inequality.

We are given a weighted directed graph $G$ on $n$ vertices with edge weights that are nonnegative real numbers. The weight of a spanning tree, with edges directed towards its root, is the product of the weights of its edges. Let $\Upsilon_r(G)$ be the probability distribution on spanning trees rooted at vertex $r$ such that the probability of a tree is proportional to its weight, and let $\Upsilon(G)$ be the probability distribution on all spanning trees, with probabilities proportional to the weights of the trees. The goal is to sample a random spanning tree (according to $\Upsilon(G)$), or a random spanning tree with fixed root $r$ (according to $\Upsilon_r(G)$).

There is a tradeoff between generality and simplicity, so we give more than one variation of the same meta-algorithm. The first algorithm is `RandomTreeWithRoot`, which given a vertex $r$, returns a random spanning tree rooted at $r$. Sampling a random spanning tree (with unrestricted root) of an undirected or Eulerian graph is easily reduced to one call to `RandomTreeWithRoot`. For more general graphs, the procedure `RandomTree` can be used.

These are random-walk based algorithms, so we start by defining a Markov chain from $G$. $G$ is *stochastic* if for each vertex the weighted outdegree, i.e. the sum of weights of edges leaving the vertex, is 1. If $G$ is stochastic, then it already is a Markov chain. Otherwise, we can define two stochastic graphs $\bar{G}$ and $\tilde{G}$ based on $G$. To get $\bar{G}$, for each vertex we normalize the weights of its outdirected edges so that its weighted outdegree is 1. `RandomTreeWithRoot` uses $\bar{G}$ since $\Upsilon_r(\bar{G}) = \Upsilon_r(G)$. To get $\tilde{G}$, first add self-loops until the weighted outdegree of each vertex is the same, and then normalize the weights. `RandomTree` uses $\tilde{G}$ since $\Upsilon(\tilde{G}) = \Upsilon(G)$. Both procedures use a subroutine `RandomSuccessor`$(u)$ that returns a random successor vertex using the appropriate Markov chain. Markov chain parameters written with overbars refer to $\bar{G}$, and parameters written with tildes refer to $\tilde{G}$.

`RandomTreeWithRoot` (see Figure 1) maintains a "current tree", which initially consists of just the root. While there remain vertices not in the tree,

* Supported in part by NSA grant MDA904-92-H-3060 and NSF grant DMS 9206374.

```
RandomTreeWithRoot($r$)
    for $i \leftarrow 1$ to $n$
        $InTree[i] \leftarrow$ false
    $Next[r] \leftarrow$ nil
    $InTree[r] \leftarrow$ true
    for $i \leftarrow 1$ to $n$
        $u \leftarrow i$
        while not $InTree[u]$
            $Next[u] \leftarrow$ RandomSuccessor($u$)
            $u \leftarrow Next[u]$
        $u \leftarrow i$
        while not $InTree[u]$
            $InTree[u] \leftarrow$ true
            $u \leftarrow Next[u]$
    return $Next$
```

Figure 1: Algorithm for obtaining random spanning tree with prescribed root $r$.

the algorithm does a random walk from one such vertex, erasing cycles as they are created, until the walk encounters the current tree. Then the cycle-erased trajectory gets added to the current tree. It has been known for some time that the path from a vertex to the root of a random spanning tree is a loop-erased random walk (see e.g. [23] and [7]), but this is the first time that anyone has used this fact to make a provably efficient algorithm. See [20] for background on loop-erased random walks.

**Theorem 1** `RandomTreeWithRoot`($r$) *returns a random spanning tree rooted at $r$.*

The proofs of this and subsequent theorems in the introduction are in §3.

Suppose that what we want is a random spanning tree without prescribing the root. If we may easily pick a random vertex from the steady state distribution $\tilde{\pi}$ of the random walk on $\tilde{G}$, then pick a $\tilde{\pi}$-random root, and call `RandomTreeWithRoot` with this root. The result is a random spanning tree. Aldous calls this theorem "the most often rediscovered result in probability theory" [3]; [6] includes a nice proof.

For undirected graphs and Eulerian graphs, $\tilde{\pi}$ is just the uniform distribution on vertices. In the case of undirected graphs, since any vertex $r$ may be used to generate a free spanning tree, it turns out to be more efficient to pick $r$ to be a random endpoint of a random edge, sample from $\Upsilon_r$, and then pick a uniformly random vertex to be the root.

**Theorem 2** *The number of times that* `RandomTreeWithRoot`($r$) *calls* `RandomSuccessor` *is given by the mean commute time between $r$ and a $\bar{\pi}$-random vertex. (The running time is linear in the number of calls to* `RandomSuccessor`*.)*

With $E_i T_j$ denoting the expected number of steps for a random walk started at $i$ to reach $j$, the mean commute time between $i$ and $j$ is $E_i T_j + E_j T_i$, and is always dominated by twice the cover time.

The *mean hitting time* is the expected time it takes to go from a $\pi$-random vertex to another $\pi$-random vertex:

$$\tau = \sum_{i,j} \pi(i)\pi(j) E_i T_j .$$

(A nice fact, which the proofs will not need, is that for each start vertex $i$, $\tau = \sum_j \pi(j) E_i T_j$ [3].) If $G$ is stochastic, and we have a $\pi$-random vertex $r$ as root, `RandomTreeWithRoot` makes an average of $2\tau$ calls to `RandomSuccessor`. For undirected graphs, a random endpoint of a random edge is $\bar{\pi}$-random, so the variation described above runs in $2\bar{\tau}$ time. In these cases it is perhaps surprising that the running time should be so small, since the expected time for just the first vertex to reach the root is $\tau$. The expected additional work needed to connect all the remaining vertices to the root is also $\tau$.

For general graphs `RandomTree` (see Figure 2) may be used to sample a random spanning tree within $O(\tilde{\tau})$ time. Note that since the root of a random tree is distributed according to $\tilde{\pi}$, the second algorithm automatically yields a random sampling procedure: return the root of a random spanning tree.

The second algorithm is essentially the same as the first algorithm, except that it adjoins an extra vertex labeled "death" to the graph. The probability of a normal vertex moving to death is $\varepsilon$, and the other transition probabilities are scaled down by $1-\varepsilon$. Since the death node is a sink, it makes a natural root from which to grow a spanning tree. The death node is then deleted from the spanning tree, resulting in a rooted forest in the original graph. If the forest has one tree, then it is a random tree. Otherwise $\varepsilon$ is decreased and another try is made.

**Theorem 3** *If* `Attempt` *returns a spanning tree, then it is a random spanning tree.* `RandomTree` *calls* `RandomSuccessor` *on average $\leq 22\tau$ times, so the expected running time of* `RandomTree` *is $O(\tau)$.*

## 2. Relation to Previous Work

There is a long history of algorithms to generate random spanning trees from a graph, and recently

```
RandomTree
   $\varepsilon \leftarrow 1$
   repeat
      $\varepsilon \leftarrow \varepsilon/2$
      $tree \leftarrow$ Attempt($\varepsilon$)
   until $tree \neq$ Failure
   return $tree$

Attempt($\varepsilon$)
   for $i \leftarrow 1$ to $n$
      $InTree[i] \leftarrow$ false
   $num\_roots \leftarrow 0$
   for $i \leftarrow 1$ to $n$
      $u \leftarrow i$
      while not $InTree[u]$
         if Chance($\varepsilon$) then
            $Next[u] \leftarrow$ nil
            $InTree[u] \leftarrow$ true
            $num\_roots \leftarrow num\_roots + 1$
            if $num\_roots = 2$ then
               return Failure
         else
            $Next[u] \leftarrow$ RandomSuccessor($u$)
            $u \leftarrow Next[u]$
      $u \leftarrow i$
      while not $InTree[u]$
         $InTree[u] \leftarrow$ true
         $u \leftarrow Next[u]$
   return $Next$
```

Figure 2: Algorithm for obtaining random spanning trees. `Chance`($\varepsilon$) returns `true` with probability $\varepsilon$.

there has been much work on self-verifying algorithms for sampling from the stationary distribution of a Markov chain that don't require knowledge of the mixing time of the Markov chain. In this section we compare the results of this paper to previous work.

## 2.1. Random tree generation

The first algorithms for generating random spanning trees were based on the Matrix Tree Theorem, which allows one to compute the number of spanning trees by evaluating a determinant (see [5, ch. 2, thm. 8]). Guénoche [16] and Kulkarni [19] gave one such algorithm that runs in $O(n^3m)$ time[1], where $n$ is the number of vertices and $m$ is the number of edges. This algorithm was optimized for the generation of many random spanning trees to make it more suitable for Monte Carlo studies [9]. Colbourn, Day, and Nel [8] reduced the time spent computing determinants to get an $O(n^3)$ algorithm for random spanning trees. Colbourn, Myrvold, and Neufeld [10] simplified this algorithm, and showed how to sample random arborescences in the time required to multiply $n \times n$ matrices, currently $O(n^{2.376})$ [11].

[1]Guénoche used $m \leq n^2$ and stated the running time as $O(n^5)$.

A number of other algorithms are based on random walks on the graph. For some graphs the best determinant algorithm will be faster, and for others the random-walk algorithms will be faster, but Broder argues that for most graphs, the random-walk algorithms will be faster [6]. Broder [6] and Aldous [2] found the random-walk algorithm (described in the introduction) for randomly generating spanning trees after discussing the Matrix Tree Theorem with Diaconis. The algorithm works for undirected graphs and runs within the cover time of the random walk. The cover time $T_c$ of a simple undirected graph is bounded by $O(n^3)$, and is often as small as $O(n \log n)$; see [6] and references contained therein. Broder also described an algorithm for the *approximate* random generation of arborescences from a directed graph. When the out-degree is regular, the running time is $O(T_c)$, but for general simple directed graphs, Broder's algorithm takes $O(nT_c)$ time. Kandel, Matias, Unger, and Winkler [18] extended the Aldous-Broder algorithm to sample arborescences of a directed Eulerian graph (i.e., one in which in-degree equals out-degree at each node) within the cover time. Wilson and Propp [26] gave an algorithm for sampling random arborescences from a *general* directed graph within 18 cover times. Most of this running time is spent just picking the root, which must be distributed according to the stationary distribution of the random walk.

All of these random-walk algorithms run within the cover time of the graph $\bar{G}$ — the expected time it takes for the random walk to reach all the vertices. `RandomTree` runs within the mean hitting time of $\tilde{G}$ — the expected time it takes for the random walk to reach a single vertex distributed according to the steady state distribution. For the previous random-walk algorithms (except Wilson-Propp), it is a simple matter to randomly select the root according the steady state distribution $\tilde{\pi}$ of the graphs for which the algorithms work, so `RandomTreeWithRoot` may be applied to these graphs. For undirected graphs the time bound is the mean hitting time of $\bar{G}$, for Eulerian graphs the time bound is the *maximum*

| Tree Algorithm | Expected running time |
|---|---|
| Guénoche [16] / Kulkarni [19] | $O(n^3 m)$ |
| Colbourn, Day, Nel [8] | $O(n^3)$ |
| Colbourn, Myrvold, Neufeld [10] | $M(n) = O(n^{2.376})$ |
| Aldous [2] / Broder [6] | $O(\bar{T}_c)$ (undirected) |
| Kandel, Matias, Unger, Winkler [18] | $O(\bar{T}_c)$ (Eulerian) |
| Wilson, Propp [26] | $O(\bar{T}_c)$ (any graph) |
| This paper | $O(\bar{\tau})$ (undirected) |
| This paper | $O(\bar{h})$ (Eulerian) |
| This paper | $O(\tilde{\tau})$ (any graph) |

$n$ = number of vertices

$m$ = number of edges

$M(n)$ = time to multiply two $n \times n$ matrices

$\pi$ = stationary probability distribution

$E_i T_j$ = expected time to reach $j$ starting from $i$

$\tau$ = mean hitting time = $\sum_{i,j} \pi(i)\pi(j) E_i T_j$

$h$ = maximum hitting time = $\max_{i,j} E_i T_j$

$E_i C$ = expected time to visit all states starting from $i$

$T_c$ = cover time = $\max_i E_i C$

Table 1: Summary of algorithms for generating random spanning trees of a graph.

*hitting time* $\bar{h}$ of $\bar{G}$, i.e. the maximum over all pairs of states $i$ and $j$ of $E_i T_j$.

The mean and maximum hitting times are always less than the cover time. Broder described a simple directed graph on $n$ vertices which has an exponentially large cover time [6]. It is noteworthy that the mean hitting time of Broder's graph is linear in $n$. Even for undirected graphs these times can be substantially smaller than the cover time. For instance, the graph consisting of two paths of size $n/3$ adjoined to a clique of size $n/3$ will have a cover time of $\Theta(n^3)$ but a mean hitting time of $\Theta(n^2)$. Broder notes that most random graphs have a cover time of $\Theta(n \log n)$; since most random graphs are expanders and have a mixing time of $O(1)$, their maximum hitting time will be $\Theta(n)$. Thus these times will usually be much smaller than the cover time, and in some cases the difference can be quite striking.

## 2.2. Self-verifying sampling algorithms

Random sampling of combinatorial objects has numerous applications in computer science and statistics. Usually there is a finite space $\mathcal{X}$ of objects, and a probability distribution $\pi$ on $\mathcal{X}$, and we wish to sample object $x \in \mathcal{X}$ with probability $\pi(x)$. One effective method for random sampling is to construct an ergodic Markov chain whose steady state distribution is $\pi$. Then one may start the Markov chain in some arbitrary state, run the chain for a long time, and output the final state as the sample. The final state will be sampled from a probability distribution that can be made arbitrarily close to $\pi$, if the chain is run for long enough. Despite much work at determining how long is "long enough" [12] [17] [14] [21] [25] [13], this remains an arduous task.

Asmussen, Glynn, and Thorisson [4] addressed the problem of not knowing when to stop running the chain. They give a general procedure, which given $n$ and a Markov chain on $n$ states, simulates the Markov chain for a while, stops after finite time,

| Exact Sampling Algorithm | Expected running time |
|---|---|
| Asmussen, Glynn, Thorisson [4] | finite time |
| Aldous [1] ($\varepsilon$ bias in sample) | $81\tau/\varepsilon^2$ |
| Lovász, Winkler [22] | $O(hT'_{\text{mix}} n \log n)$ |
| Propp, Wilson [24] (requires monotonicity) | $O(T_{\text{mix}} \log l)$ |
| Wilson, Propp [26] | $15T_c$ or $O(T'_{\text{mix}} n \log n)$ |
| Fill [15] | (not yet analyzed) |
| This paper | $22\tau$ $(\tau \leq h \leq T_c)$ |

$n$ = number of states

$l$ = length of longest chain (monotone Markov chains only). Usually $\log l = O(\log \log n)$.

$\tau$ = mean hitting time

$h$ = maximum hitting time

$T_c$ = cover time

$T_{\text{mix}}$ = mixing time threshold; time for Markov chain to "get within $1/e$ of random"

$T'_{\text{mix}}$ = optimal expected stationary stopping time

Table 2: Summary of exact sampling algorithms. See [3] for background on the Markov chain parameters.

and then outputs a random state distributed *exactly* according to $\pi$. However, they did not gives bounds on the running time, and described the procedure as more of a possibility result than an algorithm to run. Aldous [1] described an efficient procedure for sampling within $O(\tau/\varepsilon^2)$ time from an unknown Markov chain that can be simulated, but with some bias $\varepsilon$ in the samples. Lovász and Winkler [22] found the first provably polynomial time procedure for obtaining unbiased samples by observing an unknown Markov chain. If $T'_{\text{mix}}$ is the optimal expected time of any randomized stopping rule that leaves the Markov chain in the stationary distribution, then the Lovász-Winkler algorithm runs in $O(hT'_{\text{mix}} n \log n) \leq O(h^2 n \log n)$ time. (The parameter $T'_{\text{mix}}$ is a measure of how long the Markov chain takes to randomize, and is defined as $\tau_1^{(2)}$ in [3].) Wilson and Propp [26] described another sampling procedure which runs within the cover time of the random walk, using a technique called "coupling-from-the-past". They used this same technique to obtain unbiased samples from Markov chains with huge state spaces (e.g. $2^{35,000,000}$ states) provided that the Markov chain has a certain structure called monotonicity [24]. Recently Fill [15] has found another exact sampling method which may be applied to either moderate-sized general Markov chains or huge Markov chains with special structure. His method requires the ability to simulate the reverse Markov chain. Because the algorithm is new, the running time has not yet been determined.

The sampling application is more pleasant than the tree application, since by definition the input is already stochastic. Returning the root of the output of `RandomTree` takes $O(\tau)$ time, which compares favorably with all the previous algorithms, except the ones that rely on the Markov chain having special structure. A preliminary analysis of Fill's general Markov chain procedure suggests that the algorithm given here is faster [15]. Aldous's $O(\tau/\varepsilon^2)$ approximate sampling algorithm suggested that an $O(\tau)$ exact sampling algorithm should be possible, but several changes in the algorithm were necessary, and the analysis is completely different.

## 3. Proofs of Theorems

We will describe a randomized process that results in the random generation of a tree with a given root $r$. The procedure `RandomTreeWithRoot` simulates this process. Then we reduce the problem of finding a random tree to that of finding a random tree with a given root, and analyze the running time.

Associate with vertex $r$ an empty stack, and as-

```
while $G$ has a cycle
    Pop any such cycle off the stacks
return tree left on stacks
```

Figure 3: Cycle popping procedure.

sociate with each vertex $u \neq r$ an infinite stack of random vertices $S_u = S_{u,1}, S_{u,2}, S_{u,3}, \ldots$ such that $\Pr[S_{u,i} = v] = \Pr[$`RandomSuccessor`$(u) = v]$, and such that all the items in all the stacks are mutually independent.

The process will pop items off the stacks. To pop an item off $u$'s current stack $S_{u,i}, S_{u,i+1}, S_{u,i+2}, \ldots$, replace it with $S_{u,i+1}, S_{u,i+2}, \ldots$.

The tops of the stacks define a directed graph $G$, which contains edge $(u, v)$ if and only if $u$'s stack is nonempty and its top (first) item is $v$. If there is a directed cycle in $G$, then by "popping the cycle" we mean that we pop the top item of the stack of each vertex in the cycle. The process is summarized in Figure 3. If this process terminates, the result will be a directed spanning tree with root $r$. We will see later that this process terminates with probability 1 iff there exists a spanning tree with root $r$ and nonzero weight. But first let us consider what effect the choices of which cycle to pop might have.

For convenience, suppose there are an infinite number of colors, and that stack entry $S_{u,i}$ has color $i$. Then the directed graph $G$ defined by the stacks is vertex-colored, and the cycles that get popped are colored. A cycle may be popped many times, but a colored cycle can only be popped once. If eventually there are no more cycles, the result is a colored tree.

**Theorem 4** *The choices of which cycle to pop next are irrelevant: For a given set of stacks, either 1) the algorithm never terminates for any set of choices, or 2) the algorithm returns some fixed colored tree independent of the set of choices.*

**Proof:** Consider a colored cycle $C$ that can be popped, i.e. there is a sequence of colored cycles $C_1, C_2, C_3, \ldots, C_k = C$ that may be popped one after the other until $C$ is popped. But suppose that the first colored cycle that the algorithm pops is not $C_1$, but instead $\tilde{C}$. Is it still possible for $C$ to get popped? If $\tilde{C}$ shares no vertices with $C_1, \ldots, C_k$, then the answer is clearly yes. Otherwise, let $C_i$ be the first of these cycles that shares a vertex with $\tilde{C}$. If $C_i$ and $\tilde{C}$ are not equal as cycles, then they share some vertex $w$ which has different successor vertices in the two cycles. But since none of $C_1, \ldots, C_{i-1}$ contain $w$, $w$ has the same color in $C_i$ and $\tilde{C}$, so it must have the same successor vertex in the two cycles. Since $\tilde{C}$ and $C_i$ are equal as cycles, and $\tilde{C}$ shares no vertices with $C_1, \ldots, C_{i-1}$, $\tilde{C}$ and $C_i$ are equal as colored cycles. Hence we may pop colored cycles $\tilde{C} = C_i, C_1, C_2, \ldots, C_{i-1}, C_{i+1}, \ldots, C_k = C$.

If there are infinitely many colored cycles which can be popped, then there always will be infinitely many colored cycles which can be popped, and the algorithm never terminates. If there are a finite number of cycles which can be popped, then every one of them is eventually popped. The number of these cycles containing vertex $u$ determines $u$'s color in the resulting tree. □

To summarize, the stacks uniquely define a tree together with a partially ordered set of cycles layered on top of it. The algorithm peels off these cycles to find the tree.

An implementation of the cycle popping algorithm might start at some vertex, and do a walk on the stacks so that the next vertex is given by the top of the current vertex's stack. Whenever a vertex is re-encountered, then a cycle has been found, and it may be popped. If the current tree (initially just the root $r$) is encountered, then if we redo the walk from the start vertex with the updated stacks, no vertex encountered is part of a cycle. These vertices may then be added to the current tree, and the implemenation may then start again at another vertex. `RandomTreeWithRoot` is just this implementation. `RandomSuccessor`$(u)$ reads the top of $u$'s stack and deletes this item; in case this item wasn't supposed to be popped, it gets stored in the *Next* array. The *InTree* array gives the vertices of the current tree.

If there is a tree with root $r$ and nonzero weight, then a random walk started at any vertex eventually reachs $r$ with probability 1. Thus the algorithm halts with probability 1 if such a tree exists.

**Proof of Theorem 1:** Consider the probability that the stacks define a tree $T$ and a set $\mathcal{C}$ of colored cycles. By the i.i.d. nature of the stack entries, this probability factors into a term depending on $\mathcal{C}$ alone and a term depending on $T$ alone — the product of the cycle weights, and the weight of $T$. Even if we condition on a particular set of cycles being popped, the resulting tree is distributed according to $\Upsilon_r$. □

The proof of Theorem 1 also shows that if we sum the weights of sets of colored cycles that exclude vertex $r$, the result is the reciprocal of the weighted sum of trees rooted at $r$.

**Proof of Theorem 2:** What is the expected number of times that `RandomSuccessor`$(u)$ is called? Since the order of cycle popping is irrelevant, we may assume that the first trajectory starts at $u$.

It is a standard result (see [3]) that the expected number of times the random walk started at $u$ returns to $u$ before reaching $r$ is given by $\pi(u)[E_uT_r + E_rT_u]$, where the "return" at time 0 is included, and $E_iT_j$ is the expected number of steps to reach $j$ starting from $i$. Thus the number of calls to `RandomSuccessor` is

$$\sum_u \pi(u)(E_uT_r + E_rT_u)$$

□

If the root $r$ is $\pi$-random, then the expected number of calls to `RandomSuccessor` is

$$\sum_{u,r} \pi(u)\pi(r)(E_uT_r + E_rT_u) = 2\tau.$$

Since the number of calls to `RandomSuccessor` is at least $n-1$, we get for free

$$\tau \geq \frac{n-1}{2}.$$

This inequality isn't hard to obtain by other methods, but this way we get a nice combinatorial interpretation of the numerator.

**Proof of Theorem 3:** Consider the original graph modified to include a "death node", where every original node has an $\varepsilon$ chance of moving to the death node, which is a sink. Sample a random spanning tree rooted at the death node. If the death node has one child, then the subtree is a random spanning tree of the original graph. The `Attempt` procedure aborts if the death node will have multiple children, and otherwise it returns a random spanning tree.

The expected number of steps before the second death is $2/\varepsilon$, which upper bounds the expected running time of `Attempt`. Suppose that we start at a $\pi$-random location and do the random walk until death. The node at which the death occurs is a $\pi$-random node, and it is the death node's first child. Suppose that at this point all future deaths are suppressed. The expected number of additional calls to `RandomSuccessor` before the tree is built is at most $2\tau$. This bound has two consequences. First, the expected number of steps that a call to `Attempt` will take is bounded above by $1/\varepsilon + 2\tau$. More importantly, the expected number of suppressed deaths is bounded by $2\tau\varepsilon$. Thus the probability that a second death will occur is bounded by $2\tau\varepsilon$. But the probability of aborting is independent of which vertex `Attempt` starts at. Hence the probability of aborting is at most $\min(1, 2\tau\varepsilon)$.

The expected amount of work done before $1/\varepsilon$ becomes bigger than $\tau$ is bounded by $O(\tau)$. Afterwards the probability that a call to `Attempt` results in `Failure` decays exponentially. The probability that `Attempt` gets called at least $i$ additional times is $2^{-\Omega(i^2)}$, while the expected amount of work done the $i$th time is $\tau 2^{O(i)}$. Thus the total amount of work done is $O(\tau)$. (□)

If constant factors do not concern the reader, the proof is done. Otherwise read on.

Let $\varepsilon_j$ be the value of $\varepsilon$ the $j$th time `RandomTree` calls `Attempt` procedure. The expected number of times $T$ that `RandomTree` calls `RandomSuccessor` is bounded by

$$T \leq \sum_{i=1}^{\infty} \min(2/\varepsilon_i, 1/\varepsilon_i + 2\tau) \prod_{j=1}^{i-1} \min(1, 2\tau\varepsilon_j).$$

where $\varepsilon_j = s^{-j}$. Let $k$ be the smallest $j$ with $2\tau\varepsilon_j \leq 1$, and let $\kappa = 2\tau\varepsilon_k$; $1/s < \kappa \leq 1$. Breaking the sum apart and using $\varepsilon_j = \kappa/(2\tau)s^{k-j}$ we get

$$\begin{aligned}
T &\leq \sum_{i=1}^{k-1} \frac{4\tau}{\kappa} s^{i-k} + \sum_{i=k}^{\infty} \left(\frac{2\tau}{\kappa} s^{i-k} + 2\tau\right) \prod_{j=k}^{i-1} \kappa/s^{j-k} \\
\frac{T}{2\tau} &\leq \sum_{i=1}^{k-1} \frac{2}{\kappa}\frac{1}{s^i} + \sum_{i=0}^{\infty} \left(\frac{1}{\kappa} s^i + 1\right) \prod_{j=0}^{i-1} \kappa/s^j \\
&< \frac{2}{\kappa}\frac{1}{s-1} + \sum_{i=0}^{\infty} \left(\frac{s^i}{\kappa} + 1\right) s^i s^{-i(i-1)/2} \kappa^i \\
&= \frac{2/\kappa}{s-1} + \sum_{i=0}^{\infty} s^{-i(i-3)/2} \kappa^{i-1} + \sum_{i=0}^{\infty} s^{-i(i-1)/2} \kappa^i
\end{aligned}$$

Now since this expression is concave up in $\kappa$, we may evaluate it at $\kappa = 1$ and $\kappa = 1/s$, and take the maximum as an upper bound. It should not be surprising that evaluating at these two values of $\kappa$ yields the same answer. With $s = 2$ we get a bound of $T < 21.85\tau$. □

Suppose that the initial $\varepsilon$ is chosen to be $1/s$ raised to a random power between 0 and 1. Then $\kappa$ is $1/s$ raised to a random power between 0 and 1, and we have

$$E[\kappa^i] = \begin{cases} \frac{1-1/s^i}{i \ln s} & i \neq 0 \\ 1 & i = 0 \end{cases}.$$

Then when $s = 2.3$ we get $T < 21\tau$.

## Note

In the course of finishing this writeup I found a nicer `RandomTree` algorithm, but had insufficient time to properly analyze its performance. Look for it in the journal version.

## Acknowledgements

The author thanks Jim Propp for useful discussions and comments. Thanks also to Henry Cohn and Jim Propp for their suggestions on the manuscript.

## References

[1] David Aldous. On simulating a Markov chain stationary distribution when transition probabilities are unknown, 1994. Preprint.

[2] David J. Aldous. A random walk construction of uniform spanning trees and uniform labelled trees. *SIAM Journal of Discrete Mathematics*, 3(4):450–465, 1990.

[3] David J. Aldous and James A. Fill. *Reversible Markov Chains and Random Walks on Graphs*. Book in preparation, 1995.

[4] Søren Asmussen, Peter W. Glynn, and Hermann Thorisson. Stationary detection in the initial transient problem. *ACM Transactions on Modeling and Computer Simulation*, 2(2):130–157, 1992.

[5] Béla Bollobás. *Graph Theory: An Introductory Course*. Springer-Verlag, 1979. Graduate texts in mathematics, #63.

[6] Andrei Broder. Generating random spanning trees. In *Foundations of Computer Science*, pages 442–447, 1989.

[7] Robert Burton and Robin Pemantle. Local characteristics, entropy and limit theorems for spanning trees and domino tilings via transfer-impedances. *The Annals of Probability*, 21(3):1329–1371, 1993.

[8] Charles J. Colbourn, Robert P. J. Day, and Louis D. Nel. Unranking and ranking spanning trees of a graph. *Journal of Algorithms*, 10:271–286, 1989.

[9] Charles J. Colbourn, Bradley M. Debroni, and Wendy J. Myrvold. Estimating the coefficients of the reliability polynomial. *Congressus Numerantium*, 62:217–223, 1988.

[10] Charles J. Colbourn, Wendy J. Myrvold, and Eugene Neufeld. Two algorithms for unranking arborescences. *Journal of Algorithms*, 1995. To appear.

[11] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.

[12] Persi Diaconis. *Group Representations in Probability and Statistics*. Institute of Mathematical Statistics, 1988.

[13] Persi Diaconis and Laurent Saloff-Coste. What do we know about the Metropolis algorithm? In *Symposium on the Theory of Computing*, pages 112–129, 1995.

[14] Persi Diaconis and Daniel Stroock. Geometric bounds for eigenvalues of Markov chains. *The Annals of Applied Probability*, 1(1):36–61, 1991.

[15] James A. Fill, 1995. Personal communication.

[16] A. Guénoche. Random spanning tree. *Journal of Algorithms*, 4:214–220, 1983. In French.

[17] Mark Jerrum and Alistair Sinclair. Approximating the permanent. *SIAM Journal on Computing*, 18(6):1149–1178, 1989.

[18] D. Kandel, Y. Matias, R. Unger, and P. Winkler. Shuffling biological sequences, 1995. Preprint.

[19] V. G. Kulkarni. Generating random combinatorial objects. *Journal of Algorithms*, 11(2):185–207, 1990.

[20] Gregory F. Lawler. *Intersections of Random Walks*. Birkhäuser, 1991.

[21] László Lovász and Miklós Simonovits. On the randomized complexity of volume and diameter. In *Foundations of Computer Science*, pages 482–491, 1992.

[22] László Lovász and Peter Winkler. Exact mixing in an unknown Markov chain. *Electronic Journal of Combinatorics*, 2, 1995. Paper #R15.

[23] Robin Pemantle. Choosing a spanning tree for the integer lattice uniformly. *The Annals of Probability*, 19(4):1559–1574, 1991.

[24] James G. Propp and David B. Wilson. Exact sampling with coupled Markov chains and applications to statistical mechanics. *Random Structures and Algorithms*, 1996. To appear.

[25] Alistair Sinclair. *Algorithms for Random Generation and Counting: A Markov Chain Approach*. Birkhäuser, 1993.

[26] David B. Wilson and James G. Propp. How to get an exact sample from a generic Markov chain and sample a random spanning tree from a directed graph, both within the cover time. In *Symposium on Discrete Algorithms*, 1996.