Random Walks and Random Spanning Trees

Vijay Kothari

March 4, 2013

Abstract

We explore algorithms for generating random spanning trees. We first study an algorithm that was developed independently by David Alduous [1] and Andrei Broder [2]. The algorithm uses a a simple random walk in which edges that correspond to the first visit to vertices are added to the spanning tree. Analysis was inspired by Andrei Broder's paper. Additionally, we study an algorithm by David Wilson [5] that uses loop-erased random walks and employs a clever cycle-popping proof.

1 Introduction

Before discussing the problem, we review some terminology. Let $G = (V, E)$ be a finite, connected, undirected graph. Let $n = |V|$ be the number of vertices, $m = |E|$ be the number of edges, and, for each $i \in V$, define d_i to be the degree of i. From G, define a Markov chain M with transition matrix P where $P_{i,j} = \dfrac{1}{d_i}$ if $(i, j) \in E$ and $P_{i,j} = 0$ otherwise. A subgraph $T \subseteq G$ is a tree if and only if it is connected and it contains no cycles. Spanning trees compose a particularly important class of trees. A spanning tree $T \subseteq G$ is a tree that contains all vertices in V.

Define a directed graph G_M, associated with M, in the usual way. A subgraph $T \subseteq G_M$ is said to be a directed spanning tree of G_M rooted at $i \in V$ if and only if, for each vertex $j \neq i$, there exists a unique path from j to i. In this paper, we occasionally abuse notation and refer to trees simply by their edge sets.

The problem we discuss in this paper is that of generating a spanning tree T uniformly at random from amongst all spanning trees of G.

2 Andrei Broder Algorithm

In 1989 and 1990 respectively, Andrei Broder and David Alduous independently arrived at Algorithm 1.

Algorithm 1

1: Choose a starting vertex s arbitrarily. Set $T_V \leftarrow \{s\}$ and $T_E \leftarrow \emptyset$.
2: Do a simple random walk starting at s. Whenever we cross an edge $e = \{u, v\}$ with $v \notin V$, add v to T_V and add e to T_E.
3: Stop the random walk when $T_V = V$. Output $T = (T_V, T_E)$ as our spanning tree.

Note that the random walk stops once we've covered all vertices. It immediately follows that the expected running time of this algorithm is within a constant factor of the expected cover time, which is $O(n^3)$ in the worst case, but often $O(n \log n)$.

Before proving the correctness of the algorithm, we introduce some concepts. Let $M = X_0, X_1, \ldots$ be the Markov chain with associated directed graph $G_M = (V, E)$. Let π be the stationary distribution of G_M. Recall that $\pi_i = \frac{1}{d_i}$. Define the weight of edge $(i, j) \in E$ as $w_{i,j} = P_{i,j}$. We can then define the weight of a directed spanning tree $T = (T_V, T_E)$ as $w(T) = \sum_{e \in T_E} w_e$. Let $\mathcal{T}_i(G_M)$ refer to the set of all directed spanning trees of G_M that are rooted at i. Let $\mathcal{T}(G_M)$ refer to the set of all directed spanning trees of G_M.

Consider a time step t in the Markov chain. Define I to be the set of states visited at or before time step t. That is, $I = \bigcup_{0 \le i \le t} \{X_i\}$. For each $i \in I$, let $l(i, t)$ be the last time that state i was visited before time $t+1$. Define the backward tree at time t as $B_t = \{(X_{l(i,t)}, X_{l(i,t)+1}) |\ i \in I \setminus \{X_t\}\}$. Note that B_t is a tree rooted at X_t. In fact, for $t \ge C$, where C is the cover time, B_t is a spanning tree of G_M.

Thus the random walk X_t generates the backward tree chain $\{B_t\}$.

Theorem 2.1. *The backward tree chain has stationary distribution σ. Moreover, for any tree T, $\sigma(T) = \dfrac{w(T)}{\sum_{T' \in \mathcal{T}(G_M)} w(T')}$.*

Proof. First note that every backward tree B_t for $t \le C$ corresponds to a transient non-spanning tree. This follows from the fact that for all $t \ge C$, B_t is a spanning tree. On the

other hand, the set of all rooted directed spanning trees of G_M compose a recurrent class. We prove this below.

Let L be the set comprising all leaves in G_M. Let T be a directed spanning tree of G_M rooted at vertex i. For each leaf $l \in L$, let $p_{l,r}$ be the unique path from l to r in T. Let $p_{r,l}$ be the reverse path. Note that by construction of G_M we know that $p_{r,l}$ must exist. Let $T' = B_t$ for some $t \geq C$ be a directed spanning tree rooted at some r' that appears in the backward the chain. We show that we can transition from T' to T by choosing the correct edges to travel. If $r' \neq r$ then travel from r' to r using the unique path in T. Then, for each leaf $l \in L$, take the path $p_{r,l}$ followed by the path $p_{l,r}$. At the end of this process, each leaf $l \in L$ will have a unique path, namely $p_{l,r}$, from l to r. It directly follows that after completing the process, the backward tree will just be T. Since this argument can be made for all directed spanning trees $T, T' \in \mathcal{T}(G_M)$, we conclude that the set of all spanning trees is indeed a recurrent class.

From the discussion above, it follows that the backward tree chain has a stationary distribution. Call it σ. We relate π to σ as follows:

$$\pi_i = \lim_{N\to\infty} \frac{1}{N} \sum_{0\leq t\leq N} \mathbf{Pr}(X_t = i) = \lim_{N\to\infty} \frac{1}{N} \sum_{0\leq t\leq N} \mathbf{Pr}(B_t \text{ is rooted at } i) = \sum_{T\in\mathcal{T}_i(G_M)} \sigma(T)$$

Now, consider a directed spanning tree $T^{(i)}$ rooted at i that appears in the backward tree chain. A tree T' may only precede $T^{(i)}$ in the backward tree chain under the following conditions. There must be a vertex $j \neq i$ such that (i, j) is an edge in T'. Also, there must be a vertex $l(j)$ that is the root of T' and consequently the last vertex prior to i on the path from j to i in $T^{(i)}$. Then, $T' = T^{(i)} + (i, j) - (l(j), i)$. The stationary equations tell us:

$$\sigma(T^{(i)}) = \sum_{j\in V|(i,j)\in E} \sigma(T^{(i)} + (i, j) - (l(j), i))P_{l(j),i}$$

The summation in this equation iterates over trees that may precede $T^{(i)}$ in the backward tree chain, and the equation itself tells us how their stationary probabilities relate to $\sigma(T^{(i)})$. But the weight function w also satisfies this equality:

$$\sum_{j\in V|(i,j)\in E} w(T^{(i)} + (i, j) - (l(j), i))P_{l(j),i} = \sum_{j\in V|(i,j)\in E} \frac{w(T^{(i)})P_{i,j}}{P_{l(j),i}} P_{l(j),i} = w(T^{(i)})$$

Therefore σ is proportional to w. That is, $\sigma(T) = c \cdot w(T)$ for some constant of proportionality c. We now determine the value of c. We have:

$$1 = \sum_{i \in V} \pi_i = \sum_{T \in \mathcal{T}(G_M)} \sigma(T) = \sum_{T \in \mathcal{T}(G_M)} c \cdot w(T)$$

It's easy to see that $c = \dfrac{1}{\sum_{T \in \mathcal{T}(G_M)} w(T)}$. Thus $\sigma(T) = \dfrac{w(T)}{\sum_{T' \in \mathcal{T}(G_M)} w(T')}$ as desired. □

The forward tree chain is the complement to the backward tree chain. For each vertex $i \in I$, let $f(i,t)$ be the first time i was visited before time $t+1$. Define the forward tree at time t as $F_t = \{(X_{f(i,t)}, X_{f(i,t)-1}) | i \in I - X_t\}$. Note that the construction of F_C is similar to the spanning tree constructed by our algorithm. The difference is that F_C contains directed edges pointing towards the root X_0 whereas in the algorithm undirected edges are added.

Theorem 2.2. *If we start the markov chain M from the stationary distribution π then, for any $T \in \mathcal{T}$, we have* $\mathbf{Pr}(F_C = T) = \dfrac{w(T)}{\sum_{T' \in \mathcal{T}(G_M)} w(T')}$.

Proof. Let k be some positive integer. Then, since we start the Markov chain M from the stationary distribution and M is reversible, we have the following:

$$\begin{aligned}\mathbf{Pr}(X_0 = x_o, X_1 = x_1, \ldots X_k = x_k) \quad &= \frac{\pi_{x_0}}{d_{x_0}} \prod_{0<i<k} \frac{1}{d_i} = \frac{\pi_{x_k}}{d_{x_k}} \prod_{0<i<k} \frac{1}{d_i} \\ &= \mathbf{Pr}(X_0 = x_k, X_1 = x_{k-1}, \ldots X_k = x_0)\end{aligned}$$

So, $\mathbf{Pr}(B_k = T) = \mathbf{Pr}(F_k = T)$. Using some concepts from Theorem 2.1, we have:

$$\sigma(T) \quad = \lim_{N \to \infty} \sum_{0 \le t \le N} \mathbf{Pr}(B_t = T) = \lim_{N \to \infty} \sum_{0 \le t \le N} \mathbf{Pr}(F_t = T) = \mathbf{Pr}(F_t = T)$$

Applying Theorem 2.1 we prove the theorem: $\mathbf{Pr}(F_t = T) = \sigma(T) = \dfrac{w(T)}{\sum_{T' \in \mathcal{T}(G_M)} w(T')}$. □

Theorem 2.3. *Algorithm 1 generates a random spanning tree.*

Proof. In Algorithm 1 we start the random walk from an arbitrarily chosen vertex s. Therefore, we cannot directly apply Theorem 2.2. Nevertheless, all the relevant (non-zero) probabilities scale by the same factor. That is, if we start from an arbitrarily chosen vertex s instead of from π, we have $\mathbf{Pr}(F_C = T) = 0$ for $T \notin \mathcal{T}_s$ and $\mathbf{Pr}(F_C = T) = \dfrac{w(T)}{\sum_{T' \in \mathcal{T}_s(G_M)} w(T')}$ for $T \in \mathcal{T}_s$. Note that the weight $w(T) = \prod_{v \neq s} \dfrac{1}{d_v}$ is the same for all directed spanning trees $T \in \mathcal{T}_s$.

Thus far, we've shown that we're equally likely to choose each of the directed spanning trees in $\mathcal{T}_s$ as F_C. But each tree $T \in \mathcal{T}_s$ rooted at s in G_M corresponds to exactly one undirected tree that may be obtained by simply ignoring edge directions. Similarly, each undirected tree in G corresponds to exactly one directed tree $T \in \mathcal{T}_s$, which can be obtained by orienting the edges in the direction of s. Thus Algorithm 1 does indeed produce a random spanning tree since, as we've shown earlier, $\mathbf{Pr}(F_C = T)$ is the same for all $T \in \mathcal{T}_s$. □

The provided proof is very similar to the one provided by Andrei Broder [2].

3 David Wilson Algorithm

In 1996, David Wilson [5] constructed a different algorithm, Algorithm 2, to generate a random spanning tree. A root is given as input and serves as the initial tree. Then another vertex not yet in the tree is chosen. From that vertex, a random path is created to the current tree with loops erased as they are created. This process of creating a path between a vertex not yet in the tree to the tree is repeated until all vertices are part of the tree. The algorithm itself ensures that there are no loops in the tree and that the tree is indeed a spanning tree since it has no loops and it contains all n vertices.

The function *randomSuccessor* is used to pick a new vertex; *randomSuccessor*(u) chooses the next vertex $v \in N(u)$ uniformly at random. Note that the running time of Algorithm 2 is linear in the number of times *randomSuccessor* is called.

To assist in examining Algorithm 2 we consider an equivalent cycle-popping procedure. But before we get to that, we must understand what a stack is. One may think of a stack of cards or a stack of coins. A stack in the context of this paper can be thought of in much the same way. A stack is simply a data structure that has two basic operations, push and pop.

Algorithm 2 (Input: root $r \in V$)

1: $\forall i \neq r, inTree(i) \leftarrow false$
2: $next(r) \leftarrow nil$
3: $inTree(r) \leftarrow true$.
4: **for** $i \leftarrow 1$ to n **do**
5: $u \leftarrow i$
6: **while** $\neg inTree(u)$ **do**
7: $next(u) \leftarrow randomSuccessor(u)$
8: $u \leftarrow next(u)$
9: **end while**
10: $u \leftarrow i$
11: **while** $\neg inTree(u)$ **do**
12: $inTree(u) \leftarrow true$
13: $u \leftarrow next(u)$
14: **end while**
15: **end for**
16: Output *next*.

Pushing an element onto the stack means placing the element at the top of the stack. Popping an element from the stack means removing the top element from the stack.

We associate a stack S_u with each vertex $u \in V$. The root vertex r has stack $S_r = \emptyset$. Every vertex $u \neq r$ has an infinite stack associated with it. The infinite stack is constructed by choosing a neighbor of u uniformly at random from all neighbors, pushing it onto the stack, and repeating indefinitely. That is, if $S_{u,i}$ denotes the i^{th} element in the stack, we have $\mathbf{Pr}[S_{u,i} = v] = \mathbf{Pr}[randomSuccessor(u) = v]$ for all $u \neq r$.

In the algorithm, G_S denotes the graph induced by the stacks. In particular, if we let $top(u)$ denote the vertex at the top of stack S_u then $G_S = (V, E_S)$ where $E_S = \{(u, top(u)) | u \neq r\}$. Though G_s always contains exactly $n - 1$ edges, it needn't be a tree; indeed, it may contain cycles. If this is the case, we choose a cycle C and pop the stacks corresponding to vertices in C. This is what we mean by popping a cycle. If we ever reach a stage where G_s has no cycles, we pop it off the stack.

We color the stack elements with an infinite set of colors. For a given vertex u, stack element $S_{u,i}$ has color i. These colors produce a coloring on the cycles and the graph G_S.

Theorem 3.1. *The choices of which cycles to pop have no influence on the final set of stacks generated at output. That is, for a given set of stacks, Algorithm 3 either (a) never terminates*

Algorithm 3 (Input: root $r \in V$, stacks S_u associated with each vertex $u \in V$)

1: **while** G_S has a cycle **do**
2: Choose a cycle C in G_S at random and pop it
3: **end while**
4: Output G_S

for each set of choices or (b) outputs the same colored tree for each set of choices.

Proof. Let C be a colored cycle that is popped in one execution of the algorithm. Let $C_1, C_2, \ldots C_k$ be the sequence of colored cycles popped before C is popped. Now, suppose that the first cycle we pop is C', not C_1. Will C still be popped? If C' shares no vertices with $C_1, C_2, \ldots, C_k$, then the answer is clearly yes since either $C = C'$ or C shares no vertices with C'. Else, C shares a vertex with at least one cycle in $\{C_i | 1 \leq i \leq k\}$. Let C_l be the first such cycle. But then C' has the same colors as C_l. So, popping $C_l, C_1, C_2, \ldots, C_{l-1}, C_{l+1}, \ldots, C_k$ will still allow for C being popped. And it will be. This idea can easily be extended for other arrangements as well.

From the discussion above, it's evident that we'll pop the same cycles. Therefore, if the algorithm does not terminate in one execution due to an infinite number of colored cycles then it will not terminate in any execution. And if we produce a colored tree T in one execution of the algorithm, we'll produce the same colored tree T in all executions of the algorithm. □

Theorem 3.2. *Algorithm 2 returns a random spanning tree.*

Proof. Consider the probability that the stacks induce a particular tree T and set of cycles $\mathcal{C}$. The probability of producing tree T can be factored into two terms, $w(T)$ which depends solely on T, and $w(\mathcal{C})$, a term that is independent of T. Thus, for all trees $T \in T_r$, $\mathbf{Pr}(T) = c \cdot w(T)$ for a fixed constant c. Hence we're equally likely to produce any of the spanning trees and the theorem follows. □

Define $E_i\tau_j$ to be the expected time to go from vertex i to j. Define $\kappa(i,j) = E_i\tau_j + E_j\tau_i$ to be the commute time from i to j. Define $\tau = \sum_{u \in V} \pi(u)\pi(r)\kappa(u,r)$ be the mean hitting time. We now determine the running time.

Theorem 3.3. *If we run Algorithm 2 as is, then it will call randomSuccesor* $\sum_{u \in V} \pi(u)\kappa(u,r)$ *times in expectation. If we instead choose the starting vertex s according to the distribution* π *the expected number of calls to randomSuccesor is* 2τ *and the expected running time is* $O(\tau)$.

Proof. Consider a vertex u. The probability that a random walk starting at u visits v before returning to u is $\frac{1}{\pi(u)\kappa(u,r)}$. This is shown in [4]. So, $randomSuccessor(u)$ will be called $\pi(u)\kappa(u,r)$ times in expectation. Summing over all vertex $u \in V$, we find that Algorithm 2 calls $randomSuccessor$ $\sum_{u \in V} \pi(u)\kappa(u,r)$ times. This proves the first statement of the theorem. The second statement follows by the same logic. □

4 Conclusion

We analyzed two algorithms for generating random spanning trees. However, each algorithm can be extended to tackle other problems which haven't been discussed here. Readers who are so inclined may refer to the appropriate references. Additionally, an interesting applicational problem of biological sequence shuffling is studied in Kandel et al [3].

References

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

[2] Andrei Broder. Generating random spanning trees. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 442–447. IEEE, 1989.

[3] D Kandel, Yossi Matias, Ron Unger, and Peter Winkler. Shuffling biological sequences. *Discrete Applied Mathematics*, 71(1):171–185, 1996.

[4] László Lovász. Random walks on graphs: A survey. *Combinatorics, Paul erdos is eighty*, 2(1):1–46, 1993.

[5] David Bruce Wilson. Generating random spanning trees more quickly than the cover time. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 296–303. ACM, 1996.