How to Get a Perfectly Random Sample from a Generic Markov Chain and Generate a Random Spanning Tree of a Directed Graph *†

James Gary Propp‡ David Bruce Wilson§

Abstract

A general problem in computational probability theory is that of generating a random sample from the state space of a Markov chain in accordance with the steady-state probability law of the chain. Another problem is that of generating a random spanning tree of a graph or spanning arborescence of a directed graph in accordance with the uniform distribution, or more generally in accordance with a distribution given by weights on the edges of the graph or digraph. This paper gives algorithms for both of these problems, improving on earlier results and exploiting the duality between the two problems. Each of the new algorithms hinges on the recently-introduced technique of coupling from the past or on the linked notions of loop-erased random walk and "cycle-popping".

1. Introduction

Random sampling of combinatorial objects has found numerous applications in computer science and statistics. In such situations there is a (usually finite) space $\mathcal{X}$ of objects and a probability distribution π on $\mathcal{X}$, and we seek a random algorithm whose output is an element of $\mathcal{X}$ so that object $x \in \mathcal{X}$ is returned as our sample with probability $\pi(x)$ (if more than one sample is desired, the algorithm is repeated). One effective method for random sampling is to construct a Markov chain whose steady-state distribution is π. 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 "Monte Carlo method"). The final state will be a sample from a probability distribution that can be made arbitrarily close to π, if the chain is run for long enough. Despite much work at determining how long is "long enough" [19] [37] [21] [44] [56] [20], this remains a difficult matter, requiring delicate analysis in each individual case.

Now suppose that we had a general algorithm that, when given a Markov chain, would determine on its own how long to run the chain, and then would output a sample distributed

*This research was supported in part by NSA grant MDA904-92-H-3060, NSF grant DMS 9206374, and an ONR-NDSEG fellowship. The research that led to this article was done in part while the second author was visiting Sandia National Laboratories.

†This paper appeared in preliminary form in *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms* and *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing.*

‡MIT Math Department, propp@math.mit.edu

§Currently at DIMACS, dbwilson@alum.mit.edu

exactly according to the stationary distribution π. Then not only could we eliminate the initialization bias from our samples (obtaining what we call "exact" or "unbiased" random samples), but we could also get these samples in finite time without first having to analyze the convergence properties of the Markov chain.

This might seem to be too much to ask for, but a few years ago Asmussen, Glynn, and Thorisson showed that such an approach is possible, provided the algorithm is told how many states the Markov chain has [6]. Lovász and Winkler subsequently showed that the goal was not only theoretically possible but computationally feasible [45]. Here (and in a companion paper [55]) we give the best known algorithms (subsection 1.2 of this paper gives a more detailed comparison). These algorithms are simple and efficient, and are quite well suited to applications. The impatient reader may turn to subsection 1.1 to see what such an algorithm can look like.

Before we proceed, we need to clarify what exactly it means to say that an algorithm is "given" a Markov chain. There are two distinct senses of the word "given" that we will find useful. In the *passive setting* the algorithm is given a `NextState()` procedure that allows it to observe the state of a Markov chain as the chain evolves through time. In the *active setting* the algorithm is given a `RandomSuccessor()` procedure, which is similar to `NextState()` but first sets the state of the Markov chain to some particular value (namely the argument of the call to `RandomSuccessor()`) before letting it run one step and returning the random successor state. In both settings we refer to the problem of generating a random state of the Markov chain (in accordance with the steady-state distribution of the chain) as the *Random State* problem.

In section 2 we review the philosophy behind the *coupling from the past* (CFTP) protocol, which is a general exact sampling procedure that uses three subroutines for which there is a good deal of flexibility in implementation. We first introduced CFTP in [55], and focused on a version known as monotone-CFTP. The algorithm in subsection 1.1 is a more general (but less efficient) version of CFTP that we call voter-CFTP because of its connection with the voter model studied in the theory of interacting particle systems. Section 4 explains these connections, and bounds the running time of voter-CFTP as well as another version of CFTP which we call coalescing-CFTP.

For the passive setting, in section 3 we give a CFTP-based algorithm for the Random State problem that runs within a fixed multiple of the cover time and hence is within a constant factor of optimal. As for the active setting, it is clear that any passive-case algorithm will work in the active case, but the *cycle-popping* algorithm discussed in sections 6 and 7 does even better in general, returning an answer within the mean hitting time of the Markov chain rather than the cover time (up to constant factors). In actual practice, cycle-popping is implemented via a variant of simple random walk called Loop-Erased Random Walk (LERW), rather than the "random stacks" that give the algorithm its name; however, random stacks provide a helpful way of analyzing the behavior of the algorithm, and we think that the stacks picture is one that other probabilists may find useful as well.

One particular sort of random sampling problem that has gotten special attention over the past few years is the problem of generating a random spanning tree of a finite graph, where each spanning tree is to have equal probability of being generated. More generally, one can have a finite graph or digraph G in which each (directed or undirected) edge comes equipped with a positive real number called its *weight*, where one wishes to generate a random spanning tree or in-directed spanning arborescence such that the probability with which any particular tree/arborescence is generated is proportional to the product of the weights of its constituent edges. (Hereafter, we shall often call in-directed arborescences "trees" where there is no danger of confusion, and we shall call the problem of generating random spanning trees or arborescences the *Random Tree* problem.) Without going into details, we mention that applications of random spanning trees include

- the generation of random Eulerian tours for
  - de Bruijn sequences in psychological tests [26]
  - evaluating statistical significance of DNA patterns [5] [29][1] [8] [38]
- Monte Carlo evaluation of coefficients of the network reliability polynomial [16] [52] [14]
- generation of random tilings of regions [57] [13]

Furthermore, such trees have recreational uses; the maze shown in Figure 1 was obtained by generating a random spanning tree on a graph with 901 vertices.

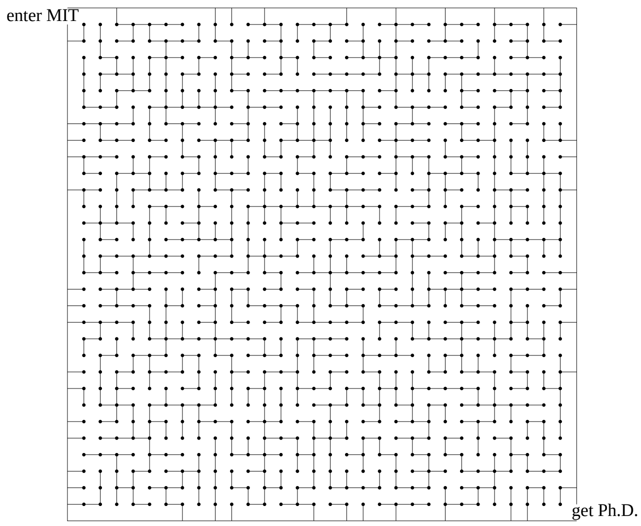

Figure 1: A spanning tree of the 30×30 grid with a 901^{st} outer vertex adjoined (the boundary), chosen randomly from all such spanning trees.

We present in this paper some new algorithms for the Random Tree problem: one based on CFTP (tree-CFTP; see section 5) and several others based on cycle-popping (see sections 6 and 7). (These algorithms work by reducing the Random Tree problem to the related *Random Tree With Root* problem, i.e. the problem of generating a random tree with specified root.) Known algorithms for generating random trees may be placed into two categories: those based on computing determinants and those based on random walks. Direct comparison of the two categories is difficult, as the relative efficiency of the two algorithms depends on rather different features of the input. However, we can show that in the class of known

[1][29] does not use random spanning trees, and generates biased Eulerian tours

algorithms based on random walk, ours are both the most general and the most efficient. Subsection 1.3 compares these tree algorithms at greater length.

The Random Tree problem might seem to be merely a special case of the more general Random State problem; however, it is a special case of central importance, and in a certain sense there is a dual relationship between the problem of obtaining a random state from the steady-state distribution of a Markov chain and the problem of generating a random spanning tree of a weighted digraph. On the one hand, one can create a Markov chain whose states are the spanning trees of G and (despite the lack of a partial ordering on the state space that our earlier article [55] relied upon) one can apply CFTP in an efficient way to get a random spanning tree of G, as described in section 5. More specifically, the tree sampling algorithm first calls the general Markov chain sampling algorithm to pick the root of the tree, and then does some additional coupling from the past to pick the rest of the tree. On the other hand, the cycle-popping algorithm described in section 6 is not really a procedure for generating a random state of a Markov chain but is in fact a procedure for generating a random *spanning tree* of a graph associated with the Markov chain; to get a random sample, one simply asks the algorithm to reveal the root of its randomly-generated tree. Thus, in the passive setting, the Random Tree problem is best solved by way of the Random State problem, while in the active setting, the Random State problem is best solved by way of the Random Tree problem.

1.1. A Simple Exact Sampling Algorithm

To illustrate the simplicity of the sampling algorithms in this paper, we give one of them here, called voter-CFTP. We have a Markov chain with states numbered 1 through n, and a randomized procedure `RandomSuccessor()` that, when given a state of the Markov chain, returns a random successor state with appropriate probabilities. We define a two-dimensional array M with the rules

$$M_{0,i} = i \quad (i = 1 \ldots n)$$

and

$$M_{t,i} = M_{t+1,\texttt{RandomSuccessor}(i)} \quad (t < 0,\ i = 1 \ldots n),$$

as illustrated in Figure 2. If the Markov chain is irreducible and aperiodic, then with probability 1, for some T all the entries of the column $M_{T,\cdot}$ will have the same value k. In section 2 we show that this value k is a perfectly random (i.e., unbiased) sample from the steady-state distribution of the Markov chain.

(Note that it is only necessary to generate columns of the unbounded array M until we obtain a constant column. Also note that once one has generated the t-column all earlier-constructed columns can be discarded, so a genuinely two-dimensional array, although conceptually helpful, is not really needed. Although this procedure works, variants that are only slightly more complicated are much more efficient; see section 6.)

1.2. History of Exact Sampling with Markov Chains

Just a few years ago Asmussen, Glynn, and Thorisson [6] gave the first algorithm for sampling from the stationary distribution π of a Markov chain without knowledge of the mixing time of the Markov chain. They gave a general procedure that, given n and a Markov chain on

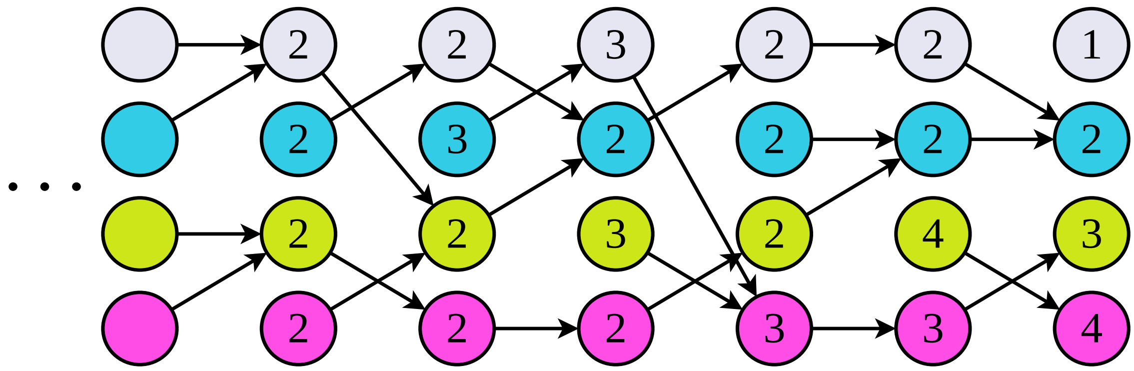

Figure 2: Illustration of voter-CFTP. The values in the first (rightmost) column are initialized with the numbers $1, \ldots, n$, and the values in each of the other columns are taken from the column to its right using the `RandomSuccessor()` procedure.

n states, simulates the Markov chain for a while, stops after finite time, and then outputs a random state distributed exactly according to π. However, their procedure is complicated, no bounds on the running time were given, and the authors described it as more of an existence proof than an algorithm to run. Aldous [1] then devised an efficient procedure for sampling, but with some bias ε in the samples. Let τ denote the *mean hitting time* of the Markov chain, i.e. the expected number of Markov chain steps to travel from one state i distributed according to π to another state j distributed according to π but independent of i. Aldous's procedure runs within time $O(\tau/\varepsilon^2)$. Intrigued by these results, Lovász and Winkler [45] found the first exact sampling algorithm that provably runs in time polynomial in certain natural parameters associated with the Markov chain. Let $E_i T_j$ denote the expected number of steps for the Markov chain to reach j starting from i. The *maximum hitting time* h is the maximum over all pairs of states i and j of $E_i T_j$. A *(randomized) stopping time* is a rule that decides when to stop running a Markov chain, based on the moves that it has made so far (and some additional random variables). Let T'_{mix} be the optimal expected time of any randomized stopping time that leaves a particular Markov chain in the stationary distribution. The parameter T'_{mix} is one measure of how long the Markov chain takes to randomize, and is denoted by T_{mix} in [45] and by $\tau_1^{(2)}$ in [3]. The Lovász-Winkler algorithm is a randomized stopping rule (i.e. it works by simply observing the Markov chain) and runs in time $O(hT'_{\mathrm{mix}} n \log n) \le O(h^2 n \log n)$.

Meanwhile, in order to generate random domino tilings of regions, we devised the "monotone coupling from the past" algorithm [55]. In contrast with the previous algorithms, this one relies on a special property called monotonicity to deal with Markov chains with huge state spaces (e.g. $2^{35{,}000{,}000}$ states). A surprisingly large number of Markov chains have a monotone structure. Figure 3 shows a random sample from the Ising model of statistical mechanics (see e.g. [7] [9]) generated by applying monotone coupling from the past to the random cluster model, in conjunction with the clever technique due to Fortuin and Kasteleyn [30]. Further applications of CFTP to huge state spaces appear (or will appear) in [55] [25] [46] [40] [34] [35] [47] [51] [39] [48] [50].

In this paper we optimize the coupling from the past technique as applied to general Markov chains. The result is an algorithm that runs within a constant multiple of the

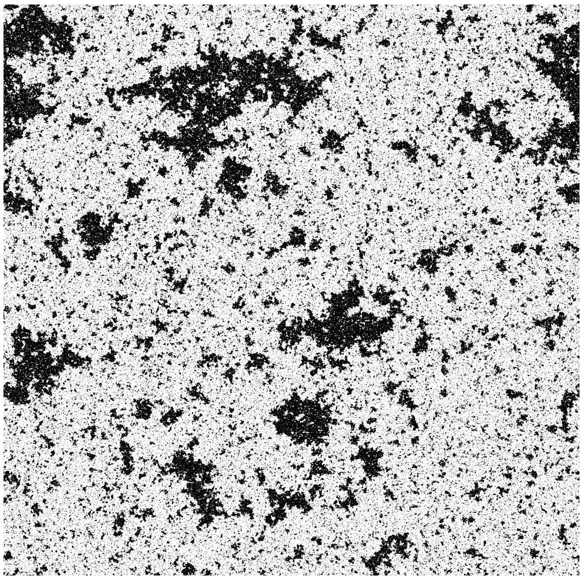

Figure 3: A perfectly equilibrated Ising state at the critical temperature on a 2100×2100 toroidal grid. This sample was generated using the method of coupling from the past. No good rigorous bounds on the mixing time are known, yet this sample was obtained in a reasonable amount of time. Details are in [55].

cover time of the Markov chain, defined as the expected time required by the Markov chain to visit all states, starting from the worst possible start state. The running time of this algorithm is much smaller than the best previously known bound on the running time of such an algorithm, as well as being smaller than the previous algorithms' *actual* running time. In section 5 we also show how to apply coupling from the past to efficiently generate random spanning trees of a graph, in spite of the fact that there is no known way to use monotone-CFTP to solve this problem.

It has been noted (e.g. in [45]) that any exact sampling algorithm that works for any Markov chain must necessarily visit all the states. For if it did not then the algorithm could not output the unvisited state (for all that the algorithm knows, this state might be almost transient), nor could it output any other state (for all that the algorithm knows, the unvisited state is almost a sink). Therefore, of the passive sampling algorithms, the cover time algorithm is within a constant factor of optimal[2]. But in the active setting, where the algorithm is also allowed to reset the state of the Markov chain, Aldous's $O(\tau/\varepsilon^2)$ approximate sampling algorithm suggested that an $O(\tau)$-time active exact sampling algorithm should be possible. After making several changes in Aldous's algorithm and completely redoing the analysis, we obtained not only an $O(\tau)$ exact sampling algorithm for the Random State problem, but also a Random Tree algorithm faster than the Broder/Aldous algorithm discussed in subsection 1.3. The running time compares favorably with all the previous algorithms, except the ones that apply only when the Markov chain has special properties.

[2]Sometimes the cover time starting from a particular state is much less than the cover time starting from the worst state. Conceivably an algorithm could do as well as the best-case cover time rather than the worst-case cover time, if it were fortunate enough to be started in the right state.

Random State algorithm	Expected running time	Space
Asmussen, Glynn, Thorisson [6]	finite	
Aldous [1] *	$< 81\tau/\varepsilon^2$	$\Theta(n)$
Lovász, Winkler [45] †	$O(hT'_{\text{mix}} n \log n)$	$\Theta(n)$
voter-CFTP, section 4	$O(T_{\text{mix}} n^2)$	$\Theta(n)$
coalescing-CFTP, section 4	$O(T_{\text{mix}} n \log n)$	$\Theta(n)$
cover-CFTP, section 3 †	$\leq 15T_c$	$\Theta(n)$
Fill [27] § ‖	under investigation	
cycle-popping / LERW, sections 6–7 ‖	$< 21\tau$	$\Theta(n)$
Propp, Wilson [55] ‡	$\Theta(T_{\text{couple}}) \leq O(T_{\text{mix}} \log l)$	$\Theta(S)$
Fill [28] ‡ § ‖	$\Theta(T_{\text{sep}})$	$\Theta(T_{\text{sep}} S)$ or $\Theta(T_{\text{sep}} + S)$ ¶ **
Fill [28] ‡ § ‖	$\Theta(T_{\text{sep}} \log T_{\text{sep}})$	$\Theta((\log T_{\text{sep}}) S)$¶

* ε bias in sample (others are bias-free).

† May simply passively observe the Markov chain.

‡ Requires monotone implementation of the Markov chain.

§ Requires the ability to simulate the time-reversal of the Markov chain.

¶ Expected space requirements (others are deterministic).

‖ "Interruptible"; see section 8.

** The second bound applies to attractive spin systems.

Table 1: Summary of Random State algorithms. Refer to Table 2 for definitions of the various Markov chain parameters. Note that all algorithms except that of Aldous are exact, i.e. free of bias. In all the cases for which the expected running time is given, the probability of the actual running time exceeding its expected value by more than a constant factor will decay exponentially. All algorithms are for the active case unless otherwise specified. For the general-purpose algorithms that work for any Markov chain (algorithms in the upper box), time and space requirements are given for the "word model of computation", for which the space required to store a single Markov chain state, and the time needed to copy a state, are both treated as $O(1)$. For the algorithms that work with huge state spaces with special structure (lower box), this idealization is not used, and the space required to store a single state is denoted by S (which is typically $\Theta(\log n)$). In all cases the space required to write down an integer as large as the running time, and the space required to read/generate a source of random or pseudo-random bits, are both treated as $O(1)$ quantities, negligible compared to the other space requirements.

n = number of states

S = space required to write down a Markov chain state

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

π = stationary probability distribution

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

τ = mean hitting time = $\sum_{i,j} \pi(i)\pi(j) E_i T_j \quad (\tau \leq h \leq T_{\mathrm{c}})$

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 = $\max_i E_i C$

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

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

T_{sep} = time for Markov chain to
"get within $1/e$ of random" in separation distance ($T_{\mathrm{sep}} \geq T_{\mathrm{mix}}$)

T_{couple} = coupling time; maximum expected time for two states
to coalesce in coupled Markov chain

Table 2: Definitions of Markov chain parameters used in Table 1. See [3] for further background on these parameters.

Jim Fill [27] [28] 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. The running time of the first of these algorithms is still under investigation.

Tables 1 and 2 summarize this history. Since the original submission of this article for publication, there have been many new developments on perfectly random sampling using Markov chains with huge state spaces. For information on this progress, the reader is referred to an annotated bibliography of perfectly random sampling with Markov chains [58], available on the World Wide Web, which will continue to be updated as new articles are written. (The 1998 version of this document will appear in the conference proceedings *Microsurveys in Discrete Probability*, edited by D. Aldous and J. Propp, to be published by the American Mathematical Society.)

1.3. History of Random Spanning Tree Generation

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 $\Upsilon(G)$ be the probability distribution on all spanning trees, where the probability of a tree is proportional to its weight, and let $\Upsilon_r(G)$ be the probability distribution on spanning trees rooted at vertex r, where the probability of each such tree is proportional to its weight. The goal is to generate a random spanning tree (according to $\Upsilon(G)$), or a random spanning tree with fixed root r (according to $\Upsilon_r(G)$).

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 [10, ch. 2, thm. 8]). Guénoche [33] and Kulkarni [41] gave one such algorithm that runs in time $O(n^3m)$, where n is the number of vertices and m is the number of edges (Guénoche used $m \leq n^2$ and stated the running time as $O(n^5)$). Colbourn, Debroni, and Myrvold [16] optimized this algorithm for the generation of many random spanning trees to make it more suitable for Monte Carlo studies. Colbourn, Day, and Nel [15] reduced the time spent computing determinants to get an $O(n^3)$ algorithm for random spanning trees. Colbourn, Myrvold, and Neufeld [17] simplified this algorithm and showed how to generate random arborescences in the time required to multiply $n \times n$ matrices (for which the best known upper bound has been $O(n^{2.376})$ for ten years [18]). The CDM economy-of-scale technique can be applied to the these subsequent determinant algorithms as well, though the speed-up is not likely to be as significant since these newer algorithms are already fairly efficient.

A number of other algorithms use random walks, that is, Markov chains based on the weighted directed graph G. For some graphs the best determinant algorithm will be faster, but Broder [12] argues that for most graphs the random-walk algorithms will be faster. Say the weighted directed graph G is *stochastic* if for each vertex the weighted out-degree, i.e. the sum of weights of edges leaving the vertex, is 1. If G is stochastic, then in a sense it already is a Markov chain — the state space is the set of vertices, and the probability of a transition from i to j is given by the weight of the edge from i to j. Otherwise, we can define two stochastic graphs $\overline{G}$ and $\widetilde{G}$ based on G. To get $\overline{G}$, for each vertex we normalize the weights of its out-directed edges so that its weighted out-degree is 1. To get $\widetilde{G}$, first add self-loops until the weighted out-degree of every vertex is the same, and then normalize the weights. Markov chain parameters written with over-bars refer to $\overline{G}$, and parameters written with tildes refer to $\widetilde{G}$. Running time bounds given in terms of $\overline{G}$ will be better than similar bounds given in terms of $\widetilde{G}$.

Broder [12] and Aldous [2] independently found the first random-walk algorithm 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 $\overline{T_c}$ of the random walk. The cover time $\overline{T_c}$ of a simple undirected graph is less than $2mn < n^3$ [4], and is often as small as $O(n \log n)$ [12]. Broder also described an algorithm for the *approximate* random generation of arborescences from a directed graph in time $O(\widetilde{T}_c)$. It works well when all vertices have the same out-degree, since then $\widetilde{T}_c = \overline{T_c}$. For general simple directed graphs we have $\widetilde{T}_c \leq n\overline{T_c}$, but for weighted graphs $\widetilde{T}_c$ could easily be much larger than $n\overline{T_c}$. Kandel, Matias, Unger, and Winkler [38] extended the Broder/Aldous algorithm to generate random arborescences of a directed Eulerian graph (i.e., one in which in-degree equals out-degree at each node) within the cover time $\overline{T_c}$.

This led us to wonder whether an equally efficient procedure for generating random arborescences could be found in the case of general weighted directed graphs. The answer is "yes", to within constant factors; section 5 shows how to use coupling from the past to generate random arborescences from a *general* directed graph within 18 cover times of $\overline{G}$. Most of this running time is spent just picking the root, which must be distributed according to the stationary distribution of the random walk on $\widetilde{G}$. (A sample from the stationary distribution of $\widetilde{G}$ may be obtained within the cover time of $\overline{G}$ using a continuous-time simulation, as described briefly in the paragraph that follows the proof of Lemma 12 in section 5.)

All of these random-walk algorithms run within the *cover time* of the graph $\overline{G}$ — the expected time it takes for the random walk to reach all the vertices. Section 6 gives another class of tree algorithms that are generally better than the previous random walk algorithms (except possibly the one given in section 5). These algorithms run within various versions of the *hitting time* of the graph — loosely speaking, the expected time it takes to get from one vertex to another under random walk. More precisely, the time bounds are $O(\overline{\tau})$ for undirected graphs, $O(\min(\overline{h}, \tilde{\tau}))$ for Eulerian graphs, and $O(\tilde{\tau})$ for general graphs. We do not know whether a $O(\overline{\tau})$ algorithm exists for general graphs.

The mean and maximum hitting times are always less than the cover time, and in some cases the difference can be quite striking. Broder [12] described a simple directed graph on n vertices for which the mean hitting time is linear in n while the cover time is exponential in n. Even for undirected graphs these hitting times can be substantially smaller than the cover time. For instance, the graph consisting of two paths of length $n/3$ adjoined to a clique on $n/3$ vertices will have a cover time of $\Theta(n^3)$ but a mean hitting time of $\Theta(n^2)$.[‡]

In addition to actually sampling random spanning trees, several of the random walk algorithms have been used to analyze the structure of random spanning trees on very large or infinite graphs [2] [53] [49] (see also [13]). For purposes of studying random spanning trees of infinite graphs, LERW is preferable if the graph is transient, whereas the Broder/Aldous

[‡]When only one such path is adjoined to the clique, the worst case cover time is still $\Theta(n^3)$, but the cover time starting from a good vertex is $\Theta(n^2)$.

Random Tree algorithm	Expected running time	Space
Guénoche [33] / Kulkarni [41] ‖	$\Theta_A(n^3 m)$	$\Theta_A(n^2)$
Colbourn, Day, Nel [15] ‖	$\Theta_A(n^3)$	$\Theta_A(n^2)$
Colbourn, Myrvold, Neufeld [17] ‡ ‖	$\Theta_A(M(n)) = O_A(n^{2.376})$	$\Theta_A(n^2)$
Phillips [54] ‖	$\Theta_A(n^3)$ (undirected)	$\Theta_A(n^2)$
matroid basis sampling [22] [24] *	polynomial (undirected)	$\Theta(n)$
Broder [12] / Aldous [2] †	$O(\overline{T_c})$ (undirected)	$\Theta(n)$
Broder [12] *	$\Theta(\widetilde{T_c})$ (any graph)	
Kandel, Matias, Unger, Winkler [38]	$O(\overline{T_c})$ (Eulerian)	$\Theta(n)$
tree-CFTP, section 5 ‡	$\Theta(\overline{T_c})$ (any graph)	$\Theta(n)$
cycle-popping / LERW, sections 6–7 ‡ ‖	$\Theta(\overline{\tau})$ (undirected)	$\Theta(n)$
cycle-popping / LERW, sections 6–7 ‡ ‖	$O(\min(\overline{h}, \tilde{\tau}))$ (Eulerian)	$\Theta(n)$
cycle-popping / LERW, sections 6–7 ‡ † ‖	$O(\tilde{\tau})$ (any graph)	$\Theta(n)$

‡ An optimal algorithm; there are graphs for which this algorithm is faster than the other algorithms by more than any constant factor.

† Has been used to analyze infinite random spanning trees (see e.g. [49]).

* ε bias in sample (others are bias-free).

‖ "Interruptible"; see section 8.

$n = (\overline{n} = \widetilde{n} =)$ number of vertices of graph

$m =$ number of edges of graph with nonzero weight

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

Table 3: Summary of Random Tree algorithms. The top three are based on linear algebra, the bottom six on random walks, and the middle one combines both techniques. The A subscript appears in the running time and space requirements of the algebraic algorithms since the operations being counted are arithmetic. In the absence of information on the numerical stability of these algorithms, it may be advisable to use exact arithmetic. Of the random walk algorithms, the top three are in the passive setting and the bottom three are in the active setting. All Markov chain parameters (which are defined in Table 2) refer *not* to the Markov chain on the space of trees, but to the random walk on the graph. Quantities with over-bars and tildes refer to $\overline{G}$ (no self-loops added) and $\widetilde{G}$ (self-loops added to make the graph out-degree regular), respectively.

algorithm is preferable if the graph is undirected and recurrent.

In the case of undirected graphs, there are two additional algorithms for sampling random spanning trees. One of these algorithms simulates the Broder/Aldous algorithm, but uses linear algebra to determine which vertex and edge will next be added to the tree [54]. For weighted graphs this approach can be faster than the Broder/Aldous algorithm, but it is not currently competitive with the faster determinant algorithms. The other random tree algorithm repeatedly adds a random edge and then deletes a random edge from the resulting cycle. This approach is a special case of a more general Markov chain algorithm for sampling maximal bases of a matroid. This Markov chain is known to randomize in polynomial time [22] [24], but when it is specialized to sampling trees, the algorithm fails to be competitive with the other tree algorithms.

Table 3 summarizes this history.

2. Coupling From the Past

The main idea used in the first set of algorithms in this paper is the "coupling from the past" protocol, recently introduced by Propp and Wilson [55] and anticipated in the non-algorithmic work of Borovkov and Foss on stochastically recursive sequences (see e.g. [11]). The idea behind coupling from the past is simple: Suppose that a Markov chain with (finite) state space $\mathcal{X}$ has been running for all time. Then the state at time 0 is distributed exactly according to the stationary distribution π of the Markov chain, assuming the chain is irreducible and aperiodic. Suppose that the Markov chain makes use of random coin-tosses that we can observe. If we can figure out the state at time 0 by looking at the outcomes of a finite number of these tosses in the recent past, then the result is an unbiased sample.

In general, the three ingredients for making CFTP workable are a *procedure for randomly generating maps* (from $\mathcal{X}$ to itself), a *method of composing random maps*, and a *test to determine if a composition of random maps is collapsing* (i.e. if the composition sends every state to one particular state). Not only must these three sub-procedures be efficient, but the random-map generator `RandomMap()` must preserve the target-distribution (see below) and must have the property that collapsing occurs after composition of relatively few randomly-chosen maps.

In applications where $\mathcal{X}$ has a specific combinatorial structure, one can often exploit this structure in one's design of the `RandomMap()` procedure (as we will do in section 5). In the case we consider now, where $\mathcal{X}$ is just a set with n elements, there is no such structure, and a map from $\mathcal{X}$ to itself (of which there are n^n) is simply given by a list of length n. We will give three specializations of CFTP that do the job: the voter-CFTP algorithm given in subsection 1.1 and analyzed in section 4, the (only slightly different) coalescing-CFTP algorithm also discussed in section 4, and the cover-time algorithm discussed in section 3. In all three cases, `RandomMap()` has the following two properties:

1. The distribution π is preserved:

$$\sum_i \pi(i) \Pr[f(i) = j] = \pi(j) \quad \text{for all } j$$

   where f is the result of a call to `RandomMap()`.

2. When $f_{-1}, f_{-2}, \ldots$ are independent values of `RandomMap()`, there is a nonzero chance that the composition $f_{-1} \circ f_{-2} \circ f_{-3} \circ \cdots$ converges to a constant-valued function (i.e., with probability greater than zero there exists a constant function F and a number N such that the composition $f_{-1} \circ f_{-2} \circ \cdots \circ f_{-n}$ is F for all $n \geq N$).

In the active case, one natural idea for implementing the `RandomMap()` procedure is, for i ranging from 1 to n, to apply `RandomSuccessor()` to the state i, obtaining $f(i)$. This gives the simple algorithm of 1.1. It is easy to see that the procedure satisfies condition 1 above, but if the Markov chain has periodicity problems, condition 2 will not be satisfied. Therefore, for the present discussion we prefer to use the following fully general procedure: For i ranging from 1 to n, with probability 1/2 set $f(i) = i$, and with probability 1/2 use `RandomSuccessor()` to set $f(i)$. As before, this satisfies condition 1. Moreover, if there is some state j which is reached with positive probability starting from any state i, then the composition of n independent outputs of `RandomMap()` has a positive chance of being a constant map, and therefore condition 2 above is also satisfied. If there is no such state j, one can check that there are multiple steady state distributions, so this `RandomMap()` procedure works precisely when the Markov chain defined by `RandomSuccessor()` has a unique steady state distribution π.

In the passive case, an analogous idea for implementing the `RandomMap()` procedure is, for i ranging from 1 to n, to let the Markov chain run (i.e., to repeatedly call `NextState()`) until the Markov chain is in state i, and to set $f(i)$ to be the state of the Markov chain after one further application of `NextState()`. In the passive setting we need the somewhat stronger hypothesis of irreducibility, i.e. that every pair of vertices is connected by paths of positive probability going in both directions, so as to ensure that the cover time is finite. (Otherwise this `RandomMap()` procedure might take forever to run.) As with the active case, probability 1/2 self-loops can be adjoined to eliminate periodicity problems and thereby ensure that condition 2 is satisfied.

We quote from [55] the pseudocode for coupling from the past (Figure 4) and the theorem proving that it works. This validates the claim we made for the simple algorithm described in subsection 1.1. Section 3 will describe a better implementation of the `RandomMap()` procedure, which in conjunction with the top-level protocol described below will yield unbiased samples in time proportional to the cover time.

`CoupleFromThePast`:
  $t \leftarrow 0$
  $F_t^0 \leftarrow$ the identity map
  `while` $F_t^0(\cdot)$ is not collapsing
    $t \leftarrow t-1$
    $f_t \leftarrow$ `RandomMap()`
    $F_t^0 \leftarrow F_{t+1}^0 \circ f_t$
  `return` the value to which $F_t^0(\cdot)$ collapses $\mathcal{X}$

Figure 4: Pseudocode for coupling from the past.

For convenience we have let $F_{t_1}^{t_2}(\cdot)$ denote the composition of random maps that maps a value at time t_1 to a value at time $t_2 \geq t_1$:

$$F_{t_1}^{t_2} = f_{t_2-1} \circ f_{t_2-2} \circ \cdots \circ f_{t_1+1} \circ f_{t_1}.$$

When coupling from the past is applied to Markov chains with huge state spaces that have special structure, the outputs of `RandomMap()` and the compositions of these functions are not represented explicitly, but for the application to general Markov chains, we may assume that these functions are represented as arrays of size n where the ith entry is $f_t(i)$ or $F_t^0(i)$. We remark that the procedure above can be run with $O(n)$ memory.

Theorem 1 *If* `RandomMap()` *satisfies conditions 1 and 2, then with probability 1 this procedure returns a value, and this value is distributed according to the stationary distribution of the Markov chain.*

Proof: Since `RandomMap()` satisfies condition 2, there must be an L such that the composition of L random maps has a positive chance of being constant. Each of the maps $F_{-L}^0(\cdot), F_{-2L}^{-L}(\cdot), \ldots$ has some positive probability $\varepsilon > 0$ of being collapsing, and since these events are independent, it will happen with probability 1 that one of these maps is collapsing, in which case F_{-M}^0 is constant for some M, and the algorithm terminates.

Let X be π-random, and let $Y_t = F_t^0(X)$. For each fixed t, Y_t is π-random. If `CoupleFromThePast` stops, then for some M F_{-M}^0 is collapsing, and for all $t < -M$, $Y_t = Y_{-M}$. The limit $Y_{-\infty} = \lim_{t\to-\infty} Y_t$ exists. But the algorithm returns precisely this limit $Y_{-\infty}$, which is π-random. □

We remark that running the chain forward in time until coupling is achieved (i.e., finding the smallest $t > 0$ for which $f_{t-1} \circ \cdots \circ f_1 \circ f_0$ is constant and then using this constant value as the output of the protocol) will usually yield a biased sample. This is related to the fact that, unlike the infinite composition $f_{-1} \circ f_{-2} \circ f_{-3} \circ \ldots$ which typically is well-defined with probability 1, the infinite composition $\cdots \circ f_2 \circ f_1 \circ f_0$ typically is well-defined with probability 0.

For an exhaustive bibliography on coupling from the past and other recent work on perfectly random sampling with Markov chains, see `http://dimacs.rutgers.edu/~dbwilson/exact`.

3. The Cover-Time Algorithm

Here we give the new `RandomMap()` procedure which makes the CFTP protocol run in time proportional to the cover time. This procedure outputs a random map from $\mathcal{X}$ to $\mathcal{X}$ satisfying conditions 1 and 2 after observing the Markov chain for some number of steps.

Note that with f denoting the result of a call to `RandomMap()`, it is not required that the random variables $f(i_1)$ and $f(i_2)$ be independent. So we will use the Markov chain to make a suitable random map, yet contrive to make the map have small image. Then the algorithm will terminate quickly. The `RandomMap()` procedure consists of two phases. The first phase estimates the cover time from state 1. The second phase starts in state 1, and actually constructs the map.

Initialization phase (Expected time $\leq T_c + T_c + T_c = 3T_c$)

- Wait until we visit state 1.
- Observe the chain until all states are visited, and let C be the number of steps required.
- Wait until we next visit state 1.

Construct Map phase (Expected time $\leq T_c + \frac{1}{2}(4T_c) = 3T_c$)

- Randomly set an alarm clock that goes off every $4C$ steps.
- When we visit state i for the first time in this phase, commit to setting $f(i)$ to be the state when the alarm clock next goes off.

Pseudocode for `RandomMap()` is given in Figure 5.

The following two lemmas show that this implementation of `RandomMap()` satisfies the two necessary conditions.

Lemma 2 *If X is a random state distributed according to π, and $f =$* `RandomMap()`*, then $f(X)$ is a random state distributed according to π.*

Proof: Let c be the value of *clock* when state X was encountered for the first time in the Construct Map phase. Since *clock* was randomly initialized, c itself is a random variable uniform between 1 and $4C$. The value $f(X)$ was obtained by starting the Markov chain in the random state X and then observing the chain after some number $(4C - c)$ of steps, where the distribution of the number of steps is independent of X. Hence $f(X)$ is also distributed according to π. $\square$

Lemma 3 *The probability that the output of* `RandomMap()` *is a constant map is at least* $3/8$.

Proof: Let C' be the first time at which we have visited all the states during the Construct Map phase (counting the start of the Construct Map phase as time 0) and let A denote the first time at which the alarm clock goes off. If $C' \leq A$, then the output of `RandomMap()` will be a constant map.

Since C and C' are independent identically distributed random variables,

$$\Pr[C' \leq C] \geq 1/2.$$

On the other hand,

$$\Pr[A \geq C] = 3/4.$$

Since A and C' are independent conditioned on C,

$$\Pr[C' \leq C \leq A] = \Pr[C' \leq C] \cdot \Pr[C \leq A] \geq 3/8.$$

$\square$

Theorem 4 *Using the above* `RandomMap()` *procedure with the CFTP protocol yields an unbiased sample from the steady-state distribution. On average the Markov chain is observed for $\leq 15T_c$ steps. The expected computational time is also $O(T_c)$, the memory is $O(n)$, and the expected number of external random bits used is $O(\log T_c)$.*

Proof: Since the odds are $\geq 3/8$ that a given output of `RandomMap()` is constant, the expected number of calls to `RandomMap()` before `RandomMap()` returns a constant map is $\leq 8/3$. Each call to `RandomMap()` observes the chain an average of $\leq 6T_c$ steps. Before starting we may arbitrarily label the current state to be state 1 and thereby reduce the expected number of steps from $16T_c$ to $15T_c$. $\Box$

The algorithm is likely to be better than the above analysis suggests, and if the Markov chain may be reset, even faster performance is possible. Also note that in general some external random bits will be required, as the Markov chain might be deterministic or almost deterministic. When the Markov chain is not deterministic, Lovász and Winkler [45] discuss how to make do without external random bits by observing the random transitions.

We remark that $T_c = O(h \log n)$. To see this, consider random walk from the vertex i that maximizes the expected time until all states have been visited. Let U_t^j be the indicator random variable for state j not having been visited by time t and let $U_t = \max_j(U_t^j)$, so that U_t is the indicator variable for there being some state that remains unvisited by time t. By Markov's inequality $\mathrm{Exp}[U^j_{\lceil 3h \rceil}] \leq 1/3$, and by submultiplicativity $\mathrm{Exp}[U^j_{m\lceil 3h \rceil}] \leq 1/3^m$. Put $H = \lceil 3h \rceil$ and $L = \lceil \log_3 n \rceil$ for convenience. We have $T_c = \sum_t \mathrm{Exp}[U_t]$, where the first HL terms in the sum (indeed all of them) are at most 1 and each subsequent term satisfies $\mathrm{Exp}[U_t] \leq \sum_j \mathrm{Exp}[U_t^j] \leq n/3^m$ when $t \geq mH$. Hence we can bound the entire sum to get $T_c \leq HL + H\left(1 + \frac{1}{3} + \frac{1}{9} + \cdots\right) = H(L + 3/2)$.

4. Running Time for Voter-CFTP and Coalescing-CFTP

The CFTP procedure given in Figure 4 is closely related to two stochastic processes, known as *the voter model* and *the coalescing random walk model*. Both of these models are based on Markov chains, in either discrete or continuous time. In this section we bound the time it take for these two processes to stabilize, and then give a variant of the CFTP procedure (for the active setting) and bound its running time.

Given a (discrete-time) Markov chain on n states, in which state i goes to state j with probability $p_{i,j}$, one defines a "coalescing random walk" by placing a pebble on each state and decreeing that the pebbles must move according to the Markov chain statistics, with one further proviso: pebbles must move independently unless they collide, at which point they must stick together. For ease of exposition we treat only Markov chains in discrete time, though there are no difficulties in generalizing our results to continuous time by taking limits of suitable discrete-time chains. The results in this section hold true vacuously when the mixing time of the Markov chain is infinite, but are informative when the mixing time is finite. In the case where a finite Markov chain is irreducible but periodic, so that its mixing time is infinite, the usual trick of adjoining self-loops gives an aperiodic Markov chain with finite mixing time.

The original Markov chain also determines a (generalized finite discrete-time) voter model, in which each of n voters starts by wanting to be be the leader. At each time step each of the voters picks a random neighbor (voter i picks neighbor j with probability $p_{i,j}$), asks for whom that neighbor will vote, and at the next time step changes his choice to be that candidate. (These updates are done in parallel: a voter at time n switches to the vote that a randomly chosen neighbor held at time $n-1$.) The voter model has received

```
RandomMap():
    Initialization phase:
        Wait until we visit state 1.
        Observe the chain until all states are visited;
        let C be the number of steps required.
        Wait until we next visit state 1.
    Construct Map phase:
        clock ← random number between 1 and 4C
        for i ← 1 to n
            status[i] ← UNSEEN
        num_assigned ← 0
        status[1] ← SEEN
        stack[0] ← 1
        ptr ← 1
        while num_assigned < n
            while clock < 4C
                clock ← clock + 1
                s ← NextState()
                if status[s] = UNSEEN then
                    status[s] ← SEEN
                    stack[ptr] ← s
                    ptr ← ptr + 1
            clock ← 0
            num_assigned ← num_assigned + ptr
            while ptr > 0
                ptr ← ptr − 1
                Map[stack[ptr]] ← s
        return Map
```

Figure 5: Pseudocode for `RandomMap()`. The Construct Map phase starts out looking at state 1. The variable $status[i]$ indicates whether or not state i has been seen yet. When i is first seen, $status[i]$ gets set to `SEEN` and i is put on the stack. When the alarm clock next goes off, for each state i in the stack, $Map[i]$ gets set to be the state at that time.

much attention in the continuous-time case, usually on a grid where $p_{i,j}$ is nonzero only if i and j are neighbors on the grid. See [36], [32], and [43] for background on the voter model.

These two models are dual to one another, in the sense that each can be obtained from the other by simply reversing the direction of time (see [3]). Specifically, suppose that for each t between t_1 and t_2, for each state we choose in advance a successor state according to the rules of the Markov chain. We can run the coalescing random walk model from time t_1 to t_2 with each pebble following the choices made in advance. Or we can run the voter model from time t_2 to time t_1 using the choices made in advance. The pebble that started at state i ends up at state j if and only if voter i ends up planning to vote for voter j.

Note that a simulation of the voter model is equivalent to running the CFTP procedure in Figure 4. Since the CFTP algorithm returns a random state distributed according to the steady-state distribution π of the Markov chain, or else runs forever, it follows that if all the voters ever agree on a common candidate, their choice will be distributed according to the steady-state distribution of the Markov chain. This result holds for both discrete- and continuous-time versions of the voter model.

In this section we analyze the running time of this process. We show

Theorem 5 *The expected time before all n voters agree on a common candidate is less than $49T_{mix}n$ steps, where T_{mix} is the mixing time of the associated Markov chain.*

(T_{mix} is defined just below the statement of Theorem 6.)

Since every `RandomMap()` step of voter-CFTP, or equivalently every time-step in a simulation of the voter model, takes $\Theta(n)$ computer time, the preceding theorem implies that one can obtain random samples within $\Theta(T_{\mathrm{mix}}n^2)$ time. However, we give a variation of the algorithm that runs in $\Theta(T_{\mathrm{mix}}n \log n)$ time. We can do this by taking advantage of the connection with the coalescing random walk model. As time moves forward, the pebbles tend to get glued together into pebble-piles, and less computational work is needed to update their positions. The total number of pile-moves that need to be made (where the decision to leave a pile in the same place counts as a move) is just the number of piles that exist at each step, summed over all steps in the time-interval under consideration. Of course simply waiting until all the pebbles get glued together will in general result in a biased sample; one needs to use the CFTP approach to get rid of the bias. We give the actual variation on the CFTP algorithm that does the job after proving the following theorem about the coalescing random walk model.

Theorem 6 *The expected time before all the pebbles coalesce is less than $49T_{mix}n$ steps. The expected number of pebble-pile moves until coalescence is less than $49T_{mix}n\ln n$.*

Theorem 5 is immediate from Theorem 6.

The mixing time T_{mix} is a measure of the time it takes for a Markov chain to become approximately randomly distributed, and is formally defined using the variation distance between probability distributions. Let μ and ρ be two probability distributions on a space $\mathcal{X}$; then the variation distance between them, $\|\mu-\rho\|$, is given by

$$\|\mu-\rho\| \equiv \max_{A\subseteq\mathcal{X}} |\mu(A)-\rho(A)| = \frac{1}{2}\sum_x |\mu(x)-\rho(x)|.$$

Let ρ_x^{*k} be the probability distribution of the state of the Markov chain when started in state x and run for k steps. Define

$$\overline{d}(k) = \max_{x,y} \|\rho_x^{*k} - \rho_y^{*k}\|.$$

Then the variation distance threshold time T_{mix} is the smallest k for which $\overline{d}(k) \leq 1/e$. It is not hard to show that $\overline{d}(k)$ is submultiplicative (in the sense that $\overline{d}(k+\ell) \leq \overline{d}(k)\overline{d}(\ell)$) and that for any starting state x, $\|\rho_x^{*k} - \pi\| \leq \overline{d}(k)$.

To prove Theorem 6 we make use of the following lemma.

Lemma 7 *Let ρ_x denote the probability distribution given by $\rho_x(y) = p_{x,y}$. Suppose that for each x the variation distance of ρ_x from π is at most $\varepsilon < 1/8$. Then the expected time before all the pebbles are in the same pile is $\leq (4/[1-\varepsilon^{1/3}]^6)n$, and the expected number of pile-moves, that is, the expected sum of the number of pebble-piles at each time step until they are all in the same pile, is $\leq (4/[1-\varepsilon^{1/3}]^6)(n \ln n)$. If $\varepsilon < 1/2$, the bounds remain $O(n)$ and $O(n \log n)$ respectively.*

Proof: Suppose that there are $m > 1$ piles at the start. Let $0 < \beta < 1$ and $1/2 < \alpha < 1$. Say that x avoids y if $\rho_x(y) < \beta\pi(y)$. Call a state *lonely* if it is avoided by at least $(1-\alpha)m$ states that are initially occupied by piles, and *popular* otherwise. We have

$$\begin{aligned}
(1-\beta)(1-\alpha)m \sum_{y \text{ lonely}} \pi(y) &\leq (1-\beta) \sum_{\substack{x,y \\ \text{pile at } x \\ x \text{ avoids } y}} \pi(y) \\
&\leq \sum_{\substack{x,y \\ \text{pile at } x \\ x \text{ avoids } y}} \pi(y) - \rho_x(y) \\
&\leq \sum_{\substack{x \\ \text{pile at } x}} \|\pi - \rho_x\| \\
&\leq m\varepsilon.
\end{aligned}$$

Let

$$\gamma = \sum_y \overline{\pi}(y) \geq 1 - \frac{1}{1-\alpha}\,\frac{\varepsilon}{1-\beta},$$

where $\overline{\pi}(y) = \pi(y)$ if y is popular, and 0 if y is lonely. Since $\varepsilon < 1/2$, we may choose α and β in the specified intervals so that $\gamma > 0$.

Let A_y be the number of pile-mergers that happen at state y after one step. A_y is one less than the number of piles that get mapped to y, unless no piles at all get mapped to y, in which case A_y is zero. Thus the expected value of A_y is the sum of the probabilities that each individual pile gets mapped to y, minus 1, plus the probability that no pile gets mapped to y.

Suppose y is popular. There are at least $a = \lceil \alpha m \rceil$ piles that have at least a $\beta\pi(y) = \beta\overline{\pi}(y)$ chance of being mapped to y. If we suppose that there are exactly a piles that get mapped to y with probability $\beta\overline{\pi}(y)$, and no other piles can get mapped to y, we can only underestimate A_y. Thus we find

$$\mathrm{Exp}[A_y] \geq a\beta\overline{\pi}(y) - 1 + (1-\beta\overline{\pi}(y))^a,$$

which is also true even if y is lonely.

Since $a \geq 1$, the above expression for $\mathrm{Exp}[A_y]$ is convex in $\overline{\pi}(y)$, and we get

$$\begin{aligned}\sum_y \mathrm{Exp}[A_y] &\geq n\left(a\beta\gamma/n - 1 + (1-\beta\gamma/n)^a\right)\\ &\geq a\beta\gamma - n + n\left[1 - (\beta\gamma/n)\binom{a}{1} + (\beta\gamma/n)^2\binom{a}{2} - (\beta\gamma/n)^3\binom{a}{3}\right]\\ &\geq \frac{(\beta\gamma)^2 a(a-1)}{2n}\left(1-\frac{a-2}{3n}\right)\end{aligned}$$

as $\beta\gamma \leq 1$. Since $\alpha > 1/2$ and $m \geq 2$, $a = \lceil \alpha m \rceil \geq 2$. If $a = 2$, then we have

$$(a-1)\left(1-\frac{a-2}{3n}\right) = a/2.$$

If $a \geq 3$, we use $n \geq m \geq a$ to get

$$(a-1)\left(1-\frac{a-2}{3n}\right) \geq (a-1)\left(1-\frac{a-2}{3a}\right) = \frac{2}{3}a\left(1-\frac{1}{a^2}\right) \geq \frac{2}{3}a\left(1-\frac{1}{9}\right) > a/2.$$

Therefore, when we let m' denote the number of piles at the next step, whenever $m > 1$ we have

$$\mathrm{Exp}[m-m'] = \mathrm{Exp}\left[\sum_y A_y\right] \geq \delta m^2,$$

where $\delta = (\beta\gamma\alpha)^2/4n$.

When there are m piles initially, let $T(m)$ be the worst-case (over all initial placements of the piles) expected number of time steps before all the piles coalesce, and let $W(m)$ be the worst-case expected sum of the number of piles at each time step before coalescence (a measure of the total amount of work the algorithm must do). Using the above bound on the expected rate of decrease in the number of piles, we show that

$$T(m) \leq (1-1/m)/\delta$$

and

$$W(m) \leq (\ln m)/\delta,$$

from which the lemma follows after choosing α and β appropriately ($\alpha = \beta = 1 - \varepsilon^{1/3}$ if $\varepsilon < 1/8$).

We will prove both estimates using a technical sub-lemma (Claim 8). We introduce the integer-valued stochastic process $m_0, m_1, m_2, \ldots$, where m_t is the number of heaps at time t (we take m_0 to be a constant). Putting $\Delta_m = \delta m^2$, we have $\mathrm{Exp}[m_{t+1} \mid m_t = m] \leq m - \Delta_m$ for all $m > 1$, so that $\mathrm{Prob}[m_{t+1} \leq m-1 \mid m_t = m] \geq \Delta_m/m$ for all times t. Indeed, this remains true if we condition on some event measurable with respect to $m_0, m_1, \ldots, m_{t-1}$. From this it easily follows that m_t converges to 1 almost surely (indeed in finite expected time). Therefore it makes sense to define $F_t = f(m_t) + f(m_{t+1}) + \cdots$ where $f(\cdot)$ is any function satisfying $f(1) = 0$. In particular, if $f(m) = 1$ for all $m > 1$, F_t is the number of time steps remaining before coalescence occurs, while if $f(m) = m$ for all $m > 1$, F_t is the amount of work that remains to be done before coalescence occurs. We wish to prove $\mathrm{Exp}[F_0] \le B(m_0)$, where the bound $B(m)$ is $(1 - \frac{1}{m})/\delta$ in the T-case and $(\ln m)/\delta$ in the W-case. We will do this by showing, more generally, that $\mathrm{Exp}[F_t \mid m_t = m] \le B(m)$ for all t.

Claim 8 *Suppose m_t is a sequence of integer-valued random variables satisfying $1 \le m_{t+1} \le m_t$, and $\mathrm{Exp}[m_{t+1} \mid E] \le m - \Delta_m$ where E is some $(m_0, m_1, \ldots, m_t)$-measurable event lying inside the event $m_t = m$ and where $\Delta_m > 0$ when $m > 1$. Suppose F_t is a sequence of random variables satisfying $F_t = 0$ when $m_t = 1$ and $F_t = f(m_t) + F_{t+1}$ when $m_t > 1$. If $B(x)$ is a function such that $B(1) \ge 0$, and $B'(x) \ge 0$ and $B''(x) \le 0$ when $x \ge 1$, and $B'(m) \ge f(m)/\Delta_m$ for $m > 1$, then $\mathrm{Exp}[F_t \mid m_t = m] \le B(m)$.*

Proof of claim: Let $F(m) = \sup_t \sup_E \mathrm{Exp}[F_t \mid E]$ where E ranges over all sub-events of $m_t = m$ that are measurable with respect to $m_0, m_1, \ldots, m_t$; this supremum is finite since $\mathrm{Exp}[F_t \mid E]$ is bounded above by $\max(f(m), f(m-1), \ldots, f(1))$ times the expected time until $m_t = 1$. We will show that $F(m) \le B(m)$. The claim is true when $m = 1$, as $\mathrm{Exp}[F_t \mid m_t = 1] = 0 \le B(1)$. Suppose $m > 1$, and assume $F(k) \le B(k)$ for all $k < m$. We have

$$\mathrm{Exp}[F_{t+1}] = \sum_{k=1}^{m} \mathrm{Prob}[m_{t+1} = k]\ \mathrm{Exp}[F_{t+1} \mid m_{t+1} = k]$$

(where all probabilities and expectations are implicitly conditional upon the event $m_t = m$ and the values $m_0, \ldots, m_{t-1}$). For all $1 \le k \le m-1$, the induction hypothesis yields

$$\mathrm{Exp}[F_{t+1} \mid m_{t+1} = k] \le F(k) \le B(k),$$

so, putting $p = \mathrm{Prob}[m_{t+1} < m \mid m_t = m]$, we have

$$\begin{aligned}\mathrm{Exp}[F_{t+1}] \;&\le\; \sum_{k=1}^{m-1} \mathrm{Prob}[m_{t+1} = k] B(k) \\ &\quad + (1-p)\mathrm{Exp}[F_{t+1} \mid m_{t+1} = m].\end{aligned}$$

On the other hand

$$\mathrm{Exp}[B(m_{t+1})] = \sum_{k=1}^{m-1} \mathrm{Prob}[m_{t+1} = k] B(k) + (1-p) B(m).$$

Comparing these last two formulas, we obtain

$$\begin{aligned}\mathrm{Exp}[F_{t+1}] \;&\le\; \mathrm{Exp}[B(m_{t+1})] \\ &\quad + (1-p)\left(\mathrm{Exp}[F_{t+1} \mid m_{t+1} = m] - B(m)\right).\end{aligned}$$

But our assumptions on $B'(x)$ and $B''(x)$ yield

$$\begin{aligned}\mathrm{Exp}[B(m_{t+1})] \;&\le\; B(\mathrm{Exp}[m_{t+1}]) \\ &\le\; B(m - \Delta_m) \\ &\le\; B(m) - \Delta_m B'(m) \\ &\le\; B(m) - f(m),\end{aligned}$$

and

$$\mathrm{Exp}[F_{t+1} \mid m_{t+1} = m] \leq F(m),$$

so

$$\begin{aligned}\mathrm{Exp}[F_{t+1}] &\leq (B(m) - f(m)) + (1-p)(F(m) - B(m)) \\ &= pB(m) - f(m) + (1-p)F(m)\end{aligned}$$

and

$$\begin{aligned}\mathrm{Exp}[F_t] &= f(m) + \mathrm{Exp}[F_{t+1}] \\ &\leq pB(m) + (1-p)F(m).\end{aligned}$$

Taking the sup over all t and all values for $m_0, \ldots, m_{t-1}$, we get

$$F(m) \leq pB(m) + (1-p)F(m),$$

from which it follows that $F(m) \leq B(m)$. ◇

We use the above claim with $F = T$ and $F = W$ to prove the desired upper bounds. The functions T and W satisfy the necessary recursive definitions, and the alleged upper bounds on their expected values take on the right value when $m = 1$ and satisfy the necessary derivative requirements (with $\Delta_m = \delta m^2$). □

Proof of Theorem 6: Following an approach used by Lovász and Winkler [45], we work with a Markov chain derived from the original Markov chain. We let one step in the derived Markov chain be T steps in the original chain. If two pebble-piles coalesce in the derived chain, they must have coalesced in the original chain, so we focus on the derived chain. We can choose for instance $T = 9T_{\text{mix}}$, so that by submultiplicativity the ε in the hypothesis on variation distance in Lemma 7 is bounded by e^{-9}. Then $9 \times \left(4/[1-\varepsilon^{1/3}]^6\right) < 49$, and we conclude from the lemma that the expected number of time steps in the original chain until coalescence is less than $49T_{\text{mix}}n$, and the expected number of pebble moves is less than $49T_{\text{mix}}n \ln n$. □

The pebbles model can be used to give us a variation on coupling from the past suitable for general Markov chains in the active setting. Rather than "pulling the values back" as voter CFTP does, which requires $\Theta(n)$ work per time step for $O(T_{\text{mix}}n)$ time steps, we start at some time in the past and run the simulation forwards in time. As the pebble piles coalesce, less work is required in later time steps. But note that until this coalescence has largely happened and there are few remaining pebble piles, the amount of work required per time step could still be quite large. The usual strategy of starting at successively larger times in the past (e.g. at times -2^i, $i = 0, 1, 2, 3, \ldots$) will not be optimal, as the most time-consuming parts of the pebble-pile simulations get repeated $O(\log(T_{\text{mix}}n))$ times, and we would get a time bound of $O(T_{\text{mix}}n \log n \log(T_{\text{mix}}n))$. Instead we use the Markov chain to define a random map procedure, and then compose these random maps in the fashion prescribed in the coupling from the past algorithm to get a random sample. To construct a random map, run the coalescing random walk process and count the number of steps before all the pebbles are in the same pile. Then restart the coalescing random walk process and run it for precisely that many steps. The places where the pebbles end up define the random map, which is collapsing with probability 1/2. The expected number of Markov simulation steps to create the map is at most $2 \times 49 T_{\mathrm{mix}} n \ln n + 49 T_{\mathrm{mix}} n$, and the expected number of maps that we need to make is at most 2. The algorithm makes on average at most $196 T_{\mathrm{mix}} n(\ln n + 1/2)$ calls to `RandomSuccessor()`. We leave it as an exercise to check that the computational overhead can be kept similarly small while using only $\Theta(n)$ space.

5. Random Spanning Trees via Coupling From the Past

We now turn from the study of the Random State problem to the study of the Random Tree and Random Tree With Root problems. Given a directed graph G with vertex set $\{1, \ldots, n\}$ and with a stochastic weighting on its directed edges (i.e., weights of outgoing edges at each vertex sum to 1), let $\mathcal{X}$ be the set of directed spanning trees of G with edges directed towards the root. Recall that we have put a probability measure Υ on $\mathcal{X}$ by letting the probability of each directed tree be proportional to its weight (defined as the product of the weights of its constituent directed edges). Also, for each vertex r we have defined Υ_r as the distribution on directed spanning trees rooted at r in which probability is proportional to weight. Our goal is to give efficient algorithms for sampling from the distributions Υ and Υ_r.

We create a Markov chain on the state space $\mathcal{X}$ as follows. Take the root of the current tree and pick a random successor according to the underlying Markov chain on G; make this vertex the new root, add an arc from the old root to the new one, and delete the out-going edge from the new root. We call this Markov chain M. See Figure 6. It is easy to check that the distribution Υ is stationary under M. Indeed, given any directed tree T with root r, its predecessors in the chain M are those trees T_s obtained from T by adding an edge $r \to s$ and deleting an edge $x \to r$; this x is uniquely determined by T and s, and is the root of T_s. The weight of T_s times the weight of $x \to r$ equals the weight of T times the weight of $r \to s$, and summing this equation over s (with x varying accordingly) we verify that M preserves Υ.

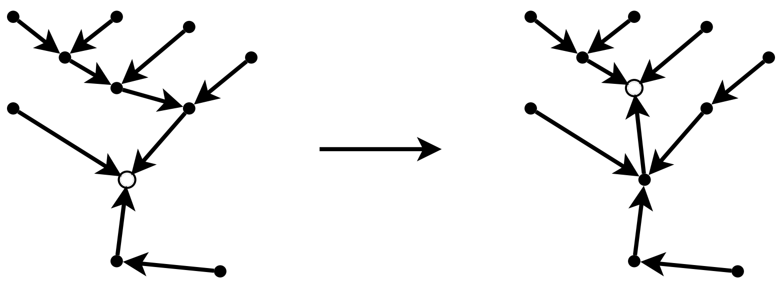

Figure 6: Example move of the Markov chain on arborescences. The root, shown in white, moves according to the underlying Markov chain.

Broder [12] and Aldous [2] used the Markov chain M to generate random spanning trees of connected undirected graphs. They exploited the fact that the simple random walk on an undirected graph is a reversible Markov chain to make an algorithm that outputs a random tree in finite time. Kandel et al. also used this Markov chain in their generation of random spanning arborescences of connected Eulerian directed graphs. We will apply the method of coupling from the past to the tree Markov chain to sample from the set of spanning arborescences of a general strongly-connected directed graph. All of these algorithms are passive-simulation algorithms, and Theorem 11 and Corollary 12 state running-time estimates for our algorithm in the passive case; in the active case, the constant factors are smaller.

Fix a vertex r, and consider the following Markov chain M_r on the set of spanning trees rooted at r: Given a spanning tree rooted at r, perform the random walk as above until the root returns to r. Call the path the walk takes from r to r an excursion. The resulting spanning tree is the next state of the Markov chain M_r. The following two easy lemmas show that the steady-state distribution of this Markov chain is the desired distribution on the set of spanning trees rooted at r.

Lemma 9 *M_r is irreducible and aperiodic if the underlying graph is strongly connected.*

Proof: Since the graph is strongly connected, there is a trajectory $r = u_0, u_1, \ldots, u_l = r$ that visits every vertex. Consider a tree T rooted at r. For i from $l-1$ down to 1, perform an excursion (or succession of excursions) that goes to u_i via the trajectory, then returns to r via the path in T. For each vertex v other than r, the last time v was reached, the succeeding vertex is its parent in the tree. Regardless of the initial starting tree, after some number of excursions the final tree is T. Thus M_r can be neither reducible nor periodic. □

Henceforth in this section, we assume that G is strongly connected, though the theorems are vacuously true even if it is not.

Lemma 10 *M_r preserves Υ_r.*

Proof: We already know that M preserves Υ. Pick a random tree, and do the walk N steps. Each tree in the walk is random. If the ith tree is rooted at r, then its conditional distribution is governed by Υ_r. Consider the subsequence of trees rooted at r. Each tree occurs in the subsequence with the proper frequency, and each tree (other than first) is obtained from the previous tree by running M_r one step. □

We can use the Markov chain M_r as the basis for a `RandomMap()` procedure that satisfies the requirements of the CFTP procedure: (1) `RandomMap()` preserves Υ_r, (2) maps returned by `RandomMap()` ("tree maps") may be easily composed, and (3) it is easy to determine whether or not a composition of tree maps is collapsing.

We will represent a spanning tree by an array T of size n, with $T[i]$ giving the label of the parent of vertex i (or `nil` if i is the root r). After one step in the Markov chain M_r (that is, after one or more steps in the Markov chain M corresponding to a single excursion in the underlying Markov chain from r to r), T will be replaced by some other tree T' rooted at r. We will see that the map that sends T to T' can be represented by an array U of size n, each entry of which is either `nil` or the label of a vertex. Note that every time the Markov chain M advances one step, it makes the current root-vertex v_i the child of its successor v_{i+1} in the excursion and makes v_{i+1} the new root. To summarize the effect of these moves, let $U[i]$ be `nil` if i does not show up in the excursion or if $i = r$, and otherwise let $U[i]$ be the vertex occurring after i in the excursion the last time that i is visited. Then $T'[i] = U[i]$ unless $U[i] = $ `nil`, in which case $T'[i] = T[i]$ (which may be `nil`).

Composing tree maps given by these arrays U is straightforward. If U_1 and U_2 represent tree maps, where U_1 is applied before U_2, the composed map is obtained by setting $(U_2 \circ U_1)[i]$ to $U_2[i]$ unless $U_2[i]$ is `nil`, in which case we set $(U_2 \circ U_1)[i]$ to $U_1[i]$. Thus we can compose tree maps efficiently. We can also test if a map is collapsing: it is collapsing if and only if the only `nil` entry is at the root. Hence all the requirements for CFTP are satisfied:

Theorem 11 *The procedure shown in Figure 7 returns a random arborescence with root r. The expected number of times we observe the Markov chain is $\leq 3T_c$, and the memory used is $O(n)$.*

Proof: The procedure repeatedly generates random excursions which define random tree maps as described above, composes them (prepending new maps to the current map as prescribed by CFTP), and returns the resulting tree. The expected runtime is at most three cover times: we wait until we see the root, then visit all the vertices, and then return to the root. $O(n)$ memory is used to store the directed tree and remember which vertices were discovered during the present excursion. □

Note that the $3T_c$ time bound is rather pessimistic. If many arborescences with the given root are desired, then the average time per directed tree is one cover-and-return time.

This handles the Random Tree With Root problem in the passive case. As for the Random Tree problem, we have:

Corollary 12 *We can generate random arborescences within 18 cover times.*

Proof: The root of a random arborescence is distributed according to π, the stationary distribution of the underlying Markov chain. So we may pick a random root using the unbiased state–sampler treated in section 2, and then pick a random tree given that root. We need only observe the Markov chain for 15+3 cover times on average. The computational effort is also $O(T_c)$, and the memory is $O(n)$. □

Note that result is within a constant factor of being optimal, since in the passive case it is impossible to randomly sample a spanning tree in less than the cover time.

Now suppose that we are given a weighted directed graph (not a Markov chain) and that we wish to generate a random directed spanning tree, such that the probability of any particular tree is proportional to the product of the edge weights. If all the weighted out-degrees are the same, then we may normalize them to be 1, and directly apply the above algorithm on the associated Markov chain. Even if the out-degrees are different, we may still generate a random arborescence with prescribed root. The only potential difficulty lies in picking the root of a random arborescence. But this problem is readily resolved by considering the continuous-time Markov chain associated with the graph. It is straightforward to generalize the unbiased sampler, specifically the `RandomMap()` procedure, to work in continuous time. The waiting time for a transition is an exponential random variable. It is only necessary for the simulation to keep track of the running total of these numbers (the elapsed simulation-time). The runtime is $O(\overline{T_c})$, where $\overline{T_c}$ is the cover time of the graph — the expected number of transitions before the graph is covered.

We pause here to consider how tree-CFTP relates to the Broder/Aldous algorithm, and more generally to review the role played by time-reversal. The random walk on a weighted

```
RandomTreeWithRootViaCFTP(r):
  wait until we visit r (actively or passively)
  for i ← 1 to n
    status[i] ← UNSEEN
  Tree[r] ← nil
  status[r] ← DEFINED
  num_assigned ← 1
  while num_assigned < n
    ptr ← 0
    s ← r
    repeat
      t ← NextState()
      if status[s] = UNSEEN then
        status[s] ← SEEN
        stack[ptr] ← s
        ptr ← ptr + 1
      if status[s] = SEEN then
        Tree[s] ← t
      s ← t
    until s = r
    num_assigned ← num_assigned + ptr
    while ptr > 0
      status[stack[ptr]] ← DEFINED
      ptr ← ptr − 1
  return Tree
```

Figure 7: Pseudocode for generating a random arborescence with prescribed root via coupling from the past. The parent of node i is stored in $Tree[i]$. If node i has not been seen, then $status[i]$ = `UNSEEN`. The nodes seen for the first time in the present excursion from the root are stored on a stack. If i is such a node, then $status[i]$ = `SEEN` and $Tree[i]$ may be changed if i is seen again in the present excursion. When the root is encountered the present excursion ends; $status[i]$ is set equal to `DEFINED`, making $Tree[i]$ immutable for the nodes i seen so far and thereby effectively prepending the excursion just finished to the preceding excursions.

digraph G gives rise to a Markov chain M on the set of trees, and it does so in a coalescent way: assuming strong connectivity holds, the present state of the chain M is independent of the far past of the walk on G. This makes it possible for one to apply coupling from the past, where, as usual, one goes into the past by taking larger and larger steps until one has gone back far enough to ensure convergence by time 0. However, in the case where the walk on G is reversible, one can simulate M into the past more directly using the time-reversed walk on G. The partial tree determined by n steps of the time-reversed walk corresponds to the set of edges in M that are forced by the behavior of the forward-time walk in G from time $-n$ to time 0. For instance, if the walk on G is unbiased random walk on an undirected graph, one obtains the Broder/Aldous algorithm, whereas if the walk on G is unbiased random walk on an Eulerian directed graph, one obtains the Kandel, Matias, Unger, and Winkler algorithm.

In general, there is no simple way to simulate the time-reversal of the random walk on G one step at a time. However, if the walk in G is at vertex r at time 0, then one can simulate a time-reversed history in chunks, where the chunks are separated by visits to r. These chunks are exactly what we have referred to as excursions, and tree maps are the way we summarize what has happened during an excursion in terms of its effect on our growing knowledge of the state of M at time 0.

6. Cycle-Popping Algorithms

In this section and the following section we describe a process called cycle-popping that is quite effective at returning random spanning trees of a directed weighted graph in the active-simulation setting. It scores over the algorithm described in section 5 in two respects: it does not depend on the assumption that the digraph is strongly connected, and it is often faster. We present two versions of cycle-popping: `RandomTreeWithRoot()`, a simple algorithm for returning a random tree with specified root, and `RandomTree()`, a slightly more complicated algorithm for returning a random tree with root not specified. In some cases where we want a random tree with unspecified root, it is possible to directly randomly select the root in accordance with the correct distribution on the root of a random tree, thereby reducing the Random Tree problem to the simpler Random Tree With Root problem; this is the case for undirected graphs and Eulerian digraphs. For general digraphs, one should use `RandomTree()`.

A different way of reducing the Random Tree problem to the Random Tree With Root problem is to adjoin a new vertex v^* to the graph (together with arcs of weight $\delta > 0$ from each vertex of the original graph to v^*) and to generate a random spanning tree of the new graph rooted at v^*. If one deletes v^* from this tree, the result is a forest on the original vertices. It is not hard to show that *in the case* where the forest that is generated has just one component (i.e., is a tree), the conditional distribution governing this tree is in fact $\Upsilon(G)$ (the desired distribution on the spanning trees of the graph G). Moreover, as δ goes to 0, the chance that the forest is a tree goes to 1; unfortunately, the running-time of the algorithm goes to infinity. What `RandomTree()` does is run `RandomTreeWithRoot()` repeatedly on the augmented graph with successively smaller values of δ until the forest on G that is produced is a tree, or (putting it differently) until v^* has only one child in the tree generated on the augmented graph.

The actual definition of cycle-popping will be deferred to section 7, since an understanding

```
RandomTreeWithRoot(r):
   for i ← 1 to n
      InTree[i] ← false
   Tree[r] ← nil
   InTree[r] ← true
   for i ← 1 to n
      u ← i
      while not InTree[u]
         Tree[u] ← RandomSuccessor(u)
         u ← Tree[u]
      u ← i
      while not InTree[u]
         InTree[u] ← true
         u ← Tree[u]
   return Tree
```

Figure 8: Algorithm for obtaining a random spanning tree with prescribed root r via cycle-popping.

of cycle-popping is not necessary for a description of the Cycle-Popping / Loop-Erased-Random-Walk algorithms. Here we merely present the algorithms; in the next section we will describe the random-stacks picture and explain why cycle-popping works.

Recall from subsection 1.3 that for a given weighted graph G we can define two different Markov chains, $\overline{G}$ and $\widetilde{G}$, where in $\widetilde{G}$ we first adjoin self-loops to make all vertices have the same out-degree, whereas in $\overline{G}$ we simply re-scale the weights so that every vertex has out-degree 1. `RandomTreeWithRoot()` uses $\overline{G}$ since $\Upsilon_r(\overline{G}) = \Upsilon_r(G)$. `RandomTree()` uses $\widetilde{G}$ since $\Upsilon(\widetilde{G}) = \Upsilon(G)$. Both procedures use a subroutine `RandomSuccessor`(u) that returns a random successor vertex using the appropriate Markov chain. As before, Markov chain parameters written with over-bars refer to $\overline{G}$, and parameters written with tildes refer to $\widetilde{G}$. Our only hypothesis is that there must exist at least one directed spanning tree of G with positive weight, with weight defined as in section 5. `RandomTreeWithRoot()` (see Figure 8) maintains a "current tree", which initially consists of just the root. While there remain vertices not in the tree, 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. [53] and [13]), but this is the first time that anyone has used this fact to make a provably efficient algorithm. See [42] for background on loop-erased random walk.

Theorem 13 *With probability 1,* `RandomTreeWithRoot`(r) *returns a random spanning tree rooted at r.*

The proofs of this and subsequent theorems in this section are in section 7.

Suppose that what we want is a random spanning tree without prescribed root. It is a pleasant fact that the root of a random spanning tree is distributed according to $\tilde{\pi}$, the steady-state distribution for the random walk process on the graph $\widetilde{G}$. (Aldous has called this "the most often rediscovered result in probability theory" [3]; the article [12] includes a nice proof.) Hence, in the case where we have an efficient way of choosing a vertex from the distribution $\tilde{\pi}$, we can reduce the Random Tree problem to the Random Tree With Root problem. This is indeed the case for undirected graphs and Eulerian digraphs; that is why there are three versions of the LERW Random Tree algorithm featured in Table 3 (one for each of the two special cases, via `RandomTreeWithRoot()`, plus the general-purpose procedure that makes use of the somewhat less efficient `RandomTree()`, described below).

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

Theorem 14 *The expected number of times that* `RandomTreeWithRoot(`r`)` *calls the procedure* `RandomSuccessor()` *is given by the mean commute time between r and a $\overline{\pi}$-random vertex. (The running time is linear in the number of calls to* `RandomSuccessor()`*.)*

With E_iT_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 $E_iT_j + E_jT_i$, and is always dominated by twice the cover time.

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

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

(A nice fact, which the proofs will not need, is that for each start vertex i, $\tau = \sum_j \pi(j)E_iT_j$ [3].) If G is stochastic, and we have a π-random vertex r as root, `RandomTreeWithRoot()` makes an average of $2\tau = 2\overline{\tau}$ calls to `RandomSuccessor()`. For undirected graphs, a random endpoint of a random edge is $\overline{\pi}$-random, so the variation described above runs in $2\overline{\tau}$ 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 $\overline{\tau}$. The expected additional work needed to connect all the remaining vertices to the root is also $\overline{\tau}$.

For general graphs, `RandomTree()` (see Figure 9) may be used to generate a random spanning tree within $O(\tilde{\tau})$ time. As was mentioned at the start of this section, `RandomTree()` effectively does `RandomTreeWithRoot()` on a larger graph to sample from spanning forests on G, rejecting those spanning forests that have more than one component while at the same time adjusting the weight-parameter so that the chance of failure will be lower the next time around. To implement this, the probability of a transition from a normal vertex to the added vertex v^* is set equal to ε, and the other transition probabilities are scaled down by $1-\varepsilon$. We think of transitions to v^* as random extinctions, and call v^* itself "death" or "the death node". 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 ε is decreased and another try is made.

Theorem 15 *If* `Attempt()` *returns a spanning tree, then it is a random spanning tree. Furthermore,* `RandomTree()` *calls* `RandomSuccessor()` *on average* $< 22\tau$ *times, so the expected running time of* `RandomTree()` *is* $O(\tau)$.

Note that since the root of a random tree is distributed according to $\tilde{\pi}$, `RandomTree()` automatically yields an (active-setting) procedure for randomly sampling from the state space of a generic Markov chain: return the root of a random spanning tree. We remark further that a small modification to `RandomTree()` reduces the expected number of calls to `RandomSuccessor()` to less than 21τ (see Section 7).

```
RandomTree():
   ε ← 1
   repeat
      ε ← ε/2
      tree ← Attempt(ε)
   until tree ≠ Failure
   return tree

Attempt(ε):
   for i ← 1 to n
      InTree[i] ← false
   num_roots ← 0
   for i ← 1 to n
      u ← i
      while not InTree[u]
         if Chance(ε) then
            Tree[u] ← nil
            InTree[u] ← true
            num_roots ← num_roots + 1
            if num_roots > 1 then
               return Failure
         else
            Tree[u] ← RandomSuccessor(u)
            u ← Tree[u]
      u ← i
      while not InTree[u]
         InTree[u] ← true
         u ← Tree[u]
   return Tree
```

Figure 9: Algorithm for obtaining a random spanning tree. `Chance(`ε`)` returns `true` with probability ε.

7. Analysis of Cycle-Popping Algorithms

The *stack model* of random walk gives us a way of enriching the state space of a Markov chain so that its transitions become deterministic rather than random. Specifically, we associate with each state (or vertex) u of the chain an infinite stack $S_u = [S_{u,1}, S_{u,2}, S_{u,3}, \ldots]$ whose elements are states of the Markov chain, such that

$$\Pr[S_{u,i} = v] = \Pr[\texttt{RandomSuccessor}(u) = v]$$

for all u, i, and such that all the $S_{u,i}$'s are jointly independent of one another. To *pop* an item off the stack $S_u = [S_{u,i}, S_{u,i+1}, S_{u,i+2}, \ldots]$, replace the stack with $[S_{u,i+1}, S_{u,i+2}, S_{u,i+3}, \ldots]$.

To simulate one step of the Markov chain, starting from the vertex u, one can simply look at the top item in the stack at u (vertex v say), pop that item from the stack, and move to vertex v. By repeating this process, one could simulate the entire history of the Markov chain deterministically. However, for our purposes (generation of a random tree with root r in the active setting), the connection between the passage of time and the popping of stacks is less direct. Also, we need to adopt a variant of the stacks picture in which the vertex r is assigned an empty stack.

In this section we will describe a way to use these "stacks of random transitions" so as to generate a random tree with a given root r. We will show that the procedure `RandomTreeWithRoot()` introduced earlier simulates this process. Then we will reduce the problem of finding a random unrooted tree to that of finding a random tree with a given root, and analyze the running time of the resulting algorithm.

At any moment, the tops of the stacks define a directed graph G that contains edge (u, v) if and only if the stack at u is nonempty and its top (first) item is v. We call G the "visible graph" determined by the stacks. If there is a directed cycle in G, then by "popping" that cycle we mean popping the top item of the stack of each vertex in the cycle. The process is summarized in Figure 10.

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

Figure 10: Cycle-popping procedure.

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 if and only if 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 16 *The choices of which cycle to pop next are irrelevant: For a given configuration of the 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.*

Remark: In the terminology of 1-player combinatorial games, cycle-popping is *strongly convergent*. Other terms that have been applied to this phenomenon are *confluence*, the *Jordan-Dedekind property*, the *diamond lemma*, and *Church-Rosser systems*. See [23] for a bibliography of these term and for other examples of the phenomenon of strong convergence.

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 $\widetilde{C}$. Is it still possible for C to get popped? If $\widetilde{C}$ 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 $\widetilde{C}$. If C_i and $\widetilde{C}$ are not equal as cycles, then they share some vertex w that 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 $\widetilde{C}$, so it must have the same successor vertex in the two cycles. Since $\widetilde{C}$ and C_i are equal as cycles, and $\widetilde{C}$ shares no vertices with $C_1, \ldots, C_{i-1}$, $\widetilde{C}$ and C_i are equal as colored cycles. Hence we may pop colored cycles $\widetilde{C} = C_i, C_1, C_2, \ldots, C_{i-1}, C_{i+1}, \ldots, C_k = C$.

If there are infinitely many colored cycles that can be popped, then there always will be infinitely many colored cycles that can be popped, and the algorithm never terminates. If there are a finite number of cycles that 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. At any stage, the "current tree" is the set of vertices v that are known not to participate in any cycles by virtue of the fact that the algorithm has found a non-self-intersecting path (via top items in the stacks) from v to r.

An implementation of the cycle-popping algorithm might start at some vertex v and do a walk on the stacks so that the next vertex is always given by the top of the stack of the current vertex. Whenever a vertex is re-encountered, a cycle has been found and it may be popped. If the current tree (initially consisting of just the root r) is encountered, then if we were to redo the walk from v with the updated stacks, none of the vertices encountered would be part of a cycle in the (new) visible graph. In fact, no matter which other vertices are popped later on, none of these vertices can ever again be part of a cycle in the visible graph. These vertices may therefore be added to the current tree, and the implementation may then start again at another vertex v'.

`RandomTreeWithRoot()` is just this implementation. `RandomSuccessor(`u`)` reads the top of u's stack and deletes this item while saving it in the $Tree$ array. The $InTree$ array gives the vertices of the current tree. (Technically, in order to be a true implementation of cycle-popping, `RandomTreeWithRoot()` would have to push back onto the stacks those items $S_{u,i}$ which were eventually found to belong to the loop-erased path from v to the current tree; however, if we are not interested in the stacks per se but only in the tree they determine, it suffices that this information be present in the $Tree$ array.)

If the weights on the edges are such that there is a tree with root r and nonzero weight, then a random walk started at any vertex eventually reaches r with probability 1. Thus the algorithm halts with probability 1 if such a tree exists.

Proof of Theorem 13: We have just seen that the procedure terminates with probability 1 when the set of trees from which we are trying to sample is non-empty. There are two ways to see that its output is governed by the proper distribution.

First, define the weight of a cycle to be the product of the weights of its constituent directed edges. 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, and the fact that $\mathcal{C}$ is compatible with T in exactly one way (i.e., under exactly one coloring of the directed edges of T), 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 colored cycles being popped, the resulting tree is distributed according to Υ_r. $\square$

Second proof of Theorem 13 (sketch): Alternatively, we can appeal to the fact that loop-erased random walk that starts at vertex v and ends when it hits vertex r creates a path governed by the same distribution as the unique directed path from v to r in a random directed tree rooted at r, with each tree being assigned probability proportional to the product of the transition probabilities of its arcs. Pemantle [53] proves this fact for undirected graphs. It is not hard to generalize the proof for strongly connected directed graphs. A simple continuity argument, in which arc weights are reduced very slightly and extra arcs of very small weight are introduced so as to yield strong connectivity, suffices to prove the claim for general directed graphs. This fact about LERW guarantees that the first stage of the `RandomTreeWithRoot(`r`)`, started from a vertex v, generates a path from v to r with the appropriate distribution.

To see that the same is true for later stages, imagine collapsing all the vertices on this path to a single vertex, obtaining a new graph G' and a designated vertex r'. Each spanning tree of G' corresponds to a spanning tree of G that contains the chosen path from v to r, and the weight of the latter is equal to the weight of the former times the product of the edge-weights along the path. Hence the distribution on spanning trees of G' rooted at r' coincides with the conditional distribution on spanning trees of G rooted at r in which the chosen path from v to r occurs. Doing loop-erased random walk in G until one first encounters this path is tantamount to doing loop-erased random walk in G'. Thus the second stage of the procedure (in which one starts from a new, unvisited vertex) generates a path that is governed by the appropriate conditional distribution. The same argument applies at subsequent stages. $\square$

However, rather than view the LERW / tree-path connection as the basis of a second proof of Theorem 13, we prefer to view the cycle-popping proof as a more pleasant, combinatorial proof of the LERW / tree-path connection.

(The first proof of Theorem 13 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 14: What is the expected number of times that `RandomSuccessor(`u`)` is called? Since the order in which cycles are popped 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 visits u before reaching r (including the visit at time 0) is given by $\pi(u)[E_u T_r + E_r T_u]$, where $E_i T_j$ is the expected number of steps to reach j starting from i. Thus the expected number of calls to `RandomSuccessor()` is

$$\sum_u \pi(u)(E_u T_r + E_r T_u)$$

$\square$

If the root r is π-random, then the expected number of calls to `RandomSuccessor()` is

$$\sum_{u,r} \pi(u)\pi(r)(E_u T_r + E_r T_u) = 2\tau.$$

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

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

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

Proof of Theorem 15: Recall that for `RandomTree`, the original graph has been modified to include a "death node", where the death node is a sink and where the transition probability to the death node from any other node is ε. Generate 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 it finds that 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 π-random location and do the random walk until death. The node at which the death occurs is a π-random node, and it is the death node's first child. Suppose that at this point all future deaths were suppressed. By the remark following the proof of Theorem 14, the expected number of additional calls to `RandomSuccessor()` before the tree is built would be at most 2τ. This bound has two consequences. First, the expected number of steps that a call to `Attempt()` will take is bounded above by $1/\varepsilon + 2\tau$ (the time until the first death plus the time to construct the rest of the tree). More importantly, the expected number of suppressed deaths is $2\tau\varepsilon$. 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, since the probability of a second death depends only on whether the death node has more than one child in the random tree that is generated, and since the distribution on spanning trees generated by the loop-erased random walk is independent of the starting vertex (Theorem 13). 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 τ is bounded by $O(\tau)$ (since the partial sum of a geometric series whose ratio stays away from 1 is bounded by a constant times its largest term). After $1/\varepsilon$ exceeds τ 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 ith time is $\tau 2^{O(i)}$. Thus the total amount of work done is $O(\tau)$. $(\square)$

If constant factors do not concern the reader, the proof is done (as indicated by the parenthesized end-of-proof symbol). The constant factor of 22 is derived below for the more diligent reader.

Let ε_j be the value of ε the jth time `RandomTree()` calls the `Attempt()` procedure. The probability that `Attempt()` aborts for the first $i-1$ attempts is at most $\prod_{j=1}^{i-1}\min(1, 2\tau\varepsilon_j)$, and given that `Attempt()` is being run for the ith time, the expected number of calls to `RandomSuccessor()` is at most $\min(2/\varepsilon_i, 1/\varepsilon_i + 2\tau)$. Hence the expected number of times T that `RandomTree()` calls `RandomSuccessor()` is bounded by

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

Suppose $\varepsilon_j = s^{-j}$ (the algorithm uses $s = 2$, though we will see below that we would get a slight improvement in our bound if we were to use a different value). Let k be the smallest j with $2\tau\varepsilon_j \le 1$, and let $\kappa = 2\tau\varepsilon_k$; $1/s < \kappa \le 1$. Breaking the sum apart and using $\varepsilon_j = \frac{\kappa}{2\tau} s^{k-j}$ we get

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

Now since this expression is concave in κ, 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 κ yields the same answer. With $s = 2$ we get a bound of $T < 21.85\tau$. □

Suppose that the initial ε is chosen to be $1/s$ raised to a power between 0 and 1 chosen uniformly at random. Then κ is $1/s$ raised to a random power between 0 and 1, and the expected value of κ^i is given by

$$\operatorname{Exp}[\kappa^i] = \begin{cases} \frac{1-1/s^i}{i \ln s} & \text{if } i \ne 0, \\ 1 & \text{if } i = 0. \end{cases}$$

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

8. Caveats and Comments

The algorithms described above will deliver samples that are free of initialization bias, provided that the algorithms are properly used. Many of the relevant strictures apply to all algorithms for random sampling, not just ours, but since we have gone so far as to call our sampling procedures "perfect" we think it only fair that we alert the reader as to what sort of uses will "void the warranty".

Most randomized algorithms have the property that the output they produce is correlated in some fashion with the running time, and most of our algorithms are no exception. In such situations one must exercise care lest the correlation lead to bias. For instance, suppose one decides to run an "exact sampling" algorithm as many times as one can within one hour, with the proviso that at the end of the hour, one will let the current run finish. Suppose further that one then intends to average some quantity $f(\cdot)$ over all samples x_i that are generated, using the average as an estimate of the mean value $\overline{f}$ of the quantity $f(x)$ as x ranges over $\mathcal{X}$ under the distribution π (so that $\overline{f} = \sum_x \pi(x)f(x)$). It is easy to devise examples in which the average of the $f(x_i)$'s, as given by the one-hour sampling protocol, fails to be an unbiased estimator of the true mean $\overline{f}$.

In this case, one paradoxical-sounding way to remove the bias is to do *less* computation; specifically, in generating an "hour's worth" of samples, one should commit ahead of time to stopping after exactly one hour, regardless of how close the computer is to generating one more sample, except in the case where the computer is still working on its *first* sample at the end of the hour, in which case one should let it run to completion. The curious fact that averages computed under this scheme are unbiased estimators of the true mean was first noticed by Glynn and Heidelberger [31]; we will not digress to give the proof here, but we mention this phenomenon as an example of the subtleties that can arise in the study of bias.

Another possible remedy is to devise algorithms whose output and running time are uncorrelated. The cycle-popping/LERW sampling algorithm has a certain interruptibility property (a notion introduced by Jim Fill [27] [28]) that allows its runs to be terminated without introducing bias. Interruptibility follows from the last sentence of the first proof of Theorem 13; even if one conditions on a particular number of Markov state transitions being generated before the algorithm terminates, the resulting state is still governed by the desired distribution. It follows that even if one has a limit on the number of Markov simulation steps that can be performed, this limitation will not introduce bias.

More realistically, any deadline would be in terms of time rather than Markov chain simulation steps. But whereas the Glynn-Heidelberger result is completely robust, and holds whether the resource limitation is time, Markov chain steps, or any other measure or combination of measures, the notion of interruptibility is fragile. The algorithm is interruptible when the limiting resource is Markov chain steps, but if, for instance, `RandomSuccessor(`u`)` takes a different amount of time to return depending on whether u has degree 2 or degree 3, then the algorithm is *not* interruptible with respect to a deadline specified in terms of time.

(We note here that when a computer's source of randomness is a stream of random bits, there can be no interruptible sampling algorithm, regardless of whether the limiting resource is random bits or computer time, if the desired distribution assigns an irrational probability to some event.)

A more straightforward way to circumvent the problem of bias is to do preliminary runs whose outputs are thrown away, purely for the purpose of seeing how long a typical run of the algorithm takes, and then to commit to doing some fixed number of runs, in accordance with one's computational resources.

Note, however, that neither this scheme nor the Glynn-Heidelberger scheme is guaranteed to return an answer in bounded time. So if one's goal is to generate a single unbiased sample, and if one has some finite bound on how long one is willing to wait, one cannot count on getting that sample with 100% certainty. This has led some to initially question whether our algorithm, which returns unbiased samples with very high probability but might return no samples at all, is really a clear win over more traditional methods, which return very slightly biased samples all the time.

One way to respond to this concern is to ask, How can one know in any particular Monte Carlo experiment that a sample generated after some particular number of steps is only slightly biased? To know this, one would have to know the mixing time, and to know the mixing time, one would need either to do complicated analyses of the Markov chain at hand or else to resort to schemes that are designed to diagnose the occurrence of mixing. But coupling from the past is just as fast as general-purpose schemes for diagnosing mixing (indeed, it arose from the study of one such scheme), so if one is taking the time to use such procedures to derive confidence in the smallness of the bias, one might as well use CFTP to push the bias down to zero.

In most applications it is preferable to obtain an answer in less than one's absolute limit of patience, and CFTP can help here. Furthermore, in most applications one wants to obtain many samples, not just one. Here one can use the Glynn-Heidelberger scheme; the probability that there is no answer by the deadline (an upper bound on the deadline-induced bias) quickly becomes negligible. Putting it differently: the point of CFTP is not that it drives the bias down from 10^{-100} to 0 (a meaningless assertion at best, in view of the likelihood that a cosmic ray will strike the computer and cause it to make a mistake); rather, the point is that in situations where CFTP is workable, it achieves a *negligible* bias in about as little time as it takes ordinary Monte Carlo to make non-negligible *progress* towards eradication of bias (in the sense of variation distance). Finally, it must never be forgotten that so-called random number generators are merely deterministic procedures that, in a variety of settings, seem to behave as if they were random. All of these effects contribute, and none should be neglected in the design and analysis of a computer experiment. Our goal is not to remove all error, but to contain and where possible abolish the initialization bias that inheres in many Monte Carlo algorithms.

Open Problems

The main open question is, how well can one solve the Random State and Random Tree problems in the active setting? In particular, is it possible to solve the Random State problem more quickly than the mean hitting time τ? Aldous gives a lower bound [1], but it is not clear how these bounds compare. There are quite a few maximal elements in the partial order of algorithms for the Random Tree problem (Table 3); an algorithm that runs in time $O(\overline{\tau})$ for general graphs (rather than time $O(\widetilde{\tau})$) would reduce this number to two. Perhaps further progress could be made by combining random walk and algebraic techniques as done in [54].

Another issue that could be addressed by future work is the question of how well one can do (in both the active and passive cases) when one wants to obtain many independent samples. One can always simply iterate the algorithms described in this paper many times, but it seems plausible that there could be economies of scale when many samples are required (see e.g. [16] and the paragraph following the proof of Theorem 11), particularly in the case of the cycle-popping algorithm, which (in the context of the Random State problem) gives so much more than is asked for.

Finally, we note that the mathematical principles that makes cycle-popping work may sometimes apply outside the realm of random arborescences. Consider, for instance, the problem of generating a random orientation of an undirected graph in which no vertex is a sink. A natural procedure for generating such an orientation is to generate an unconstrained orientation of the graph and to re-randomize the orientations of all edges that participate in sinks, iterating until no sinks remain. Adopting a suitable stack-picture, one can check that the order in which sinks are popped is irrelevant, so that all the steps in the proof of Theorem 13 go through in this new setting; the procedure generates an unbiased sink-free orientation. Perhaps cycle-popping will turn out to be just one member of a new class of efficient algorithms for a variety of combinatorial applications.

Acknowledgements

We thank Henry Cohn for offering comments on the manuscript, David Aldous for providing us with an advance copy of his book with Jim Fill on Markov chains [3], Persi Diaconis for giving helpful comments, and David Aldous, László Lovász, and Peter Winkler for sending us copies of their papers [1] [45].

References

[1] David Aldous. On simulating a Markov chain stationary distribution when transition probabilities are unknown. In David Aldous, Persi Diaconis, Joel Spencer, and J. Michael Steele, editors, *Discrete Probability and Algorithms*, volume 72 of *IMA Volumes in Mathematics and its Applications*, pages 1–9. Springer-Verlag, 1995.

[2] David J. Aldous. A random walk construction of uniform spanning trees and uniform labelled trees. *SIAM Journal on 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, `http://www.stat.berkeley.edu/~aldous/book.html`.

[4] Romas Aleliunas, Richard M. Karp, Richard J. Lipton, László Lovász, and Charles Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *20th Annual Symposium on Foundations of Computer Science*, pages 218–223, 1979.

[5] S. F. Altschul and B. W. Erickson. Significance of nucleotide sequence alignments: A method for random sequence permutation that preserves dinucleotide and codon usage. *Molecular Biology and Evolution*, 2:526–538, 1985.

[6] 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.

[7] Rodney J. Baxter. *Exactly Solved Models in Statistical Mechanics*. Academic Press, 1982.

[8] Laurel Beckett and Persi Diaconis. Spectral analysis for discrete longitudinal data. *Advances in Mathematics*, 103(1):107–128, 1994.

[9] J. J. Binney, N. J. Dowrick, A. J. Fisher, and M. E. J. Newman. *The Theory of Critical Phenomena: An Introduction to the Renormalization Group*. Oxford University Press, 1992.

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

[11] A. A. Borovkov and S. G. Foss. Stochastically recursive sequences and their generalizations. *Siberian Advances in Mathematics*, 2(1):16–81, 1992. Translated from PRIM.

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

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

[14] Charles J. Colbourn. *The Combinatorics of Network Reliability*. Oxford University Press, 1987.

[15] 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.

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

[17] Charles J. Colbourn, Wendy J. Myrvold, and Eugene Neufeld. Two algorithms for unranking arborescences. *Journal of Algorithms*, 20:268–281, 1996.

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

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

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

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

[22] Martin Dyer and Alan Frieze. Random walks, totally unimodular matrices, and a randomised dual simplex algorithm. *Mathematical Programming*, 64:1–16, 1994.

[23] Kimmo Eriksson. Strong convergence and the polygon property of 1-player games. *Discrete Mathematics*, 153:105–122, 1996.

[24] Tomás Feder and Milena Mihail. Balanced matroids. In *Proceedings of the Twenty Fourth Annual ACM Symposium on the Theory of Computing*, pages 26–38, 1992.

[25] Stefan Felsner and Lorenz Wernisch. Markov chains for linear extensions, the two-dimensional case. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 239–247, 1997.

[26] W. Fernandez de la Vega and A. Guénoche. Construction de mots circulaires aléatoires uniformément distribués. *Mathématiques et Sciences Humaines*, 58:25–29, 1977.

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

[28] James Allen Fill. An interruptible algorithm for perfect sampling via Markov chains. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 688–695, 1997.

[29] Walter M. Fitch. Random sequences. *Journal of Molecular Biology*, 163:171–176, 1983.

[30] C. M. Fortuin and P. W. Kasteleyn. On the random cluster model. I. Introduction and relation to other models. *Physica*, 57(4):536–564, 1972.

[31] Peter W. Glynn and Philip Heidelberger. Bias properties of budget constrained simulations. *Operations Research*, 38(5):801–814, 1990.

[32] David Griffeath. *Additive and Cancellative Interacting Particle Systems*. Springer-Verlag, 1979. Lecture Notes in Mathematics, #724.

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

[34] O. Häggström, M. N. M. van Lieshout, and J. Møller. Characterisation results and Markov chain Monte Carlo algorithms including exact simulation for some spatial point processes. Technical Report R-96-2040, Aalborg University, 1996.

[35] Olle Häggström and Karin Nelander. Exact sampling from anti-monotone systems, 1997. Preprint.

[36] Richard Holley and Thomas Liggett. Ergodic theorems for weakly interacting systems and the voter model. *Annals of Probability*, 3:643–663, 1975.

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

[38] D. Kandel, Y. Matias, R. Unger, and P. Winkler. Shuffling biological sequences. *Discrete Applied Mathematics*, 71:171–185, 1996.

[39] W. S. Kendall and J. Møller. Perfect Metropolis-Hastings simulation of locally stable spatial point processes. In preparation.

[40] Wilfrid S. Kendall. Perfect simulation for the area-interaction point process. In *Proceedings of the Symposium on Probability Towards the Year 2000*, 1998. To appear in C.C. Heyde and L. Accardi, editors, *Probability Perspective, World Scientific Press.*

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

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

[43] Thomas Liggett. *Interacting Particle Systems*. Springer-Verlag, 1985.

[44] László Lovász and Miklós Simonovits. On the randomized complexity of volume and diameter. In *33rd Annual Symposium on Foundations of Computer Science*, pages 482–491, 1992.

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

[46] Michael Luby, Dana Randall, and Alistair Sinclair. Markov chain algorithms for planar lattice structures (extended abstract). In *36th Annual Symposium on Foundations of Computer Science*, pages 150–159, 1995.

[47] Michael Luby and Eric Vigoda. Approximately counting up to four (extended abstract). In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 682–687, 1997.

[48] Robert B. Lund and David B. Wilson. Exact sampling algorithms for storage systems and networks. In preparation.

[49] Russell Lyons and Yuval Peres. *Probability on Trees*. Book in preparation, `http://php.indiana.edu/~rdlyons/prbtree/prbtree.html`.

[50] Jesper Møller. A note on perfect simulation of conditionally specified models, 1997. Preprint.

[51] D. J. Murdoch and P. J. Green. Exact sampling from a continuous state space, 1997. Preprint.

[52] Louis D. Nel and Charles J. Colbourn. Combining Monte Carlo estimates and bounds for network reliability. *Networks*, 20:277–298, 1990.

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

[54] Cynthia A. Phillips, 1991. Personal communication.

[55] James G. Propp and David B. Wilson. Exact sampling with coupled Markov chains and applications to statistical mechanics. *Random Structures and Algorithms*, 9:223–252, 1996.

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

[57] H. N. V. Temperley. In *Combinatorics: Proceedings of the British Combinatorial Conference 1973*, pages 202–204, 1974. London Mathematical Society Lecture Notes Series #13.

[58] David B. Wilson. Annotated bibliography of perfectly random sampling with Markov chains. `http://dimacs.rutgers.edu/~dbwilson/exact`.