ALGORITHMS FOR SAMPLING SPANNING TREES UNIFORMLY AT RANDOM

A Degree Thesis Submitted to the Faculty of Mathematics and Statistics of the Universitat Politècnica de Catalunya by

Lucia Costantini

In Partial Fulfillment of the Requirements for the Master Degree in Advanced Mathematics and Mathematical Engineering

Supervised by:

Dr. Guillem Perarnau
Contents

1 Introduction 5

2 Preliminaries 7
   2.1 Spanning Trees ........................................... 7
   2.2 Matrix Tree Theorem .................................... 11
   2.3 Markov Chains ............................................. 13
   2.4 Closed classes ............................................ 15
   2.5 Random walks on graphs ................................. 17

3 Exact sampling of spanning trees through exact counting 19
   3.1 Multigraphs ................................................. 19
   3.2 Reduction from sampling to counting ....................... 20
   3.3 Time complexity ........................................... 21

4 Aldous-Broder Algorithm 22
   4.1 Backward and forward tree chains ....................... 23
   4.2 Correctness ............................................... 27
   4.3 Time complexity ........................................... 27

5 Wilson’s algorithm 30
   5.1 Loop-erased random walks ................................ 30
   5.2 Stacks ........................................................ 31
   5.3 Colouring .................................................... 32
   5.4 Correctness .................................................. 32
   5.5 Time complexity ............................................ 34

6 Partial Rejection Sampling Algorithm. 36
6.1 Lovász local lemma ................................. 36
6.2 Construction ........................................... 39
6.3 Correctness ............................................ 41
6.4 Time complexity ....................................... 44

7 Conclusion ............................................. 52
Acknowledgements

First of all, I would like to offer my special thanks to Dr. Perarnau for the valuable help he provided with this research project. Despite the obstacles that this unusual spring has brought upon us, his willingness to give his time and his constructive suggestions have been vital in the completion of this thesis.

Secondly, I would like to express my gratitude to Dr. Serra, who, throughout a semester of lectures on Graph Theory, has managed to engage me to the point of inspiring me to explore the subject further.
Chapter 1

Introduction

Spanning trees play a fundamental role in a variety of contexts. Finding a spanning tree of a graph means finding an object which reaches every point in the original, possibly enormous graph, with the minimum amount of edges possible. Indeed, the key to the relevance of spanning trees in many fields lies in the fact that they are capable of somehow capturing important characteristics of a large, intricate graph in the most rudimentary way possible. It is not surprising then that spanning trees are largely utilised in network analysis [3] and design [18], statistical physics and mechanics [10], random maze construction [12], graph sparsification [7], graph expanders [6], and many other areas. Throughout this paper, we are going to mainly be looking at different ways that these spanning trees can be sampled uniformly at random.

The first chapter is going to focus on the background information we are going to need: we discuss some important notions and theorems from graph theory and related to Markov chains and random walks. We introduce some important definition and, through the Matrix Tree Theorem, provide the formula for counting the number of spanning trees of any given graph. We also explain how to define a Markov chain on a graph and the way we can use the probabilistic results in the construction of algorithms for uniform spanning trees.

In the second chapter, we talk about the relation between exact counting and exact sampling of spanning trees. We give the necessary intuition to recursively quantify the number of spanning trees of a given graph in terms of that of two other
graphs, which we obtain using deletion and contraction of edges. This will give us a
way to express the probability of a given spanning tree occurring using the number
of spanning trees of said graphs. Then we can use a count tree in order to show
how the counting and sampling of these trees relate to each other and reduce the
problem of sampling uniformly at random to merely calculating the eigenvalues of
the Laplacian matrix of a graph. This method is a straight-forward application of the
Matrix Tree Theorem, though it is not very efficient, as it takes \( O(m \cdot n^3) = O(n^5) \)
time.

In the third chapter, we are going to introduce the Aldous-Broder algorithm. We begin by introducing the notions of forward and backward tree and observing
that the tree output by the algorithm is indeed a forward tree. We use the Markov
chain tree theorem for proof of correctness of the algorithm to show the distribution
of the outputs is indeed uniform. Since the time it takes for it to run is the same
as the cover time, we deduce that we can sample a uniformly distributed spanning
tree with the Aldous-Broder algorithm in \( O(mn) = O(n^3) \) time.

The fourth chapter will look at Wilson’s algorithm, which uses the ideas of
popping cycles and loop-erased random walks in order to construct a uniform span-
ning tree. We will make use of the notion of stacks to describe the visible graph at
any stage in the algorithm and we will explain how to remove the cycles as they arise
using random walks. The procedure will prove to be independent of any arbitrary
choices of ordering we may make. The time complexity of Wilson’s algorithm is
going to be given by the mean hitting time of the graph, which is only as big as
the cover time in the worst possible case. So, again, Wilson’s algorithm will take at
most \( O(n^3) \) to run.

Lastly, in the fifth chapter, we are going to use an algorithmic version of the
Lovász local lemma to give a new interpretation of Wilson’s algorithm. The cycles
of a dependency graph will be characterised as the “bad” events which we want
to avoid. The famous combinatorial lemma by Lovász will show that the complete
avoidance of these events is indeed possible. Then we will construct the algorithm
for sampling trees under the important condition that any dependent events be
disjoint, so that when a bad event occurs, we can safely resample the corresponding
variables.
Chapter 2

Preliminaries

2.1 Spanning Trees

A tree can be simply defined as a connected graph with no cycles. Then, given a connected graph \( G = (V, E) \), a spanning tree \( T \) is a cycle-free subgraph of \( G \) which covers its entire vertex set. Then any spanning tree \( T = (V_T, E_T) \) of \( G \) has vertex set \( V_T = V \) and edge set \( E_T \subseteq E \) such that \( E_T = |V| - 1 \).

Any tree can be defined as two sub-trees connected by a single edge. In this way, we can define spanning trees recursively, where the base case scenario is joining a single vertex with the empty tree by an edge.

A natural question arises: how many spanning trees are there for a generic graph and how can we choose one uniformly at random? In order to answer this, let us first define a few matrices that can be constructed from a graph \( G \).

**Definition 2.1.1.** The *adjacency matrix* \( A \) of an undirected graph \( G = (V, E) \) is a square symmetric binary matrix with entries defined as follows:

\[
A_{ij} = \begin{cases} 
1, & \text{if } (i, j) \in E \\
0, & \text{otherwise}
\end{cases}
\]  

(2.1)

For directed graphs, this translates into a matrix with entry \( A_{ij} \) corresponding to the number of directed edges going from \( i \) to \( j \).
Definition 2.1.2. The incidence matrix of an undirected graph \( G \) with \( n \) vertices and \( m \) edges is the \( n \times m \) matrix \( Q \) with entries

\[
Q_{ij} = \begin{cases} 
1, & \text{if vertex } i \text{ is an endpoint of edge } j \\
0, & \text{otherwise.}
\end{cases}
\] (2.2)

The entries can be slightly modified in the case of directed graphs. Let there be a directed edge in the graph going from \( x \) to \( y \); then we define \( x \) to be the tail of the edge, and \( y \) to be its head. Then for a directed graph, the incidence matrix is

\[
Q_{ij} = \begin{cases} 
-1, & \text{if vertex } i \text{ is the tail of } j \\
1, & \text{if } i \text{ is the head of } j \\
0, & \text{otherwise.}
\end{cases}
\] (2.3)

Definition 2.1.3. Let the degree of a vertex \( i \in V \) for an undirected graph be defined by

\[
d(i) = |\{j \in V \mid (i,j) \in E\}|.
\] (2.4)

In other words, \( d(i) \) gives the number of neighbours of vertex \( i \).

For a directed graph we have the out-degree defined by

\[
d(i) = |\{j \in V \mid [i,j]\}|,
\] (2.5)

where \([i,j]\) is the edge with tail \( i \) and head \( j \).

Definition 2.1.4. The Laplacian matrix \( L \) of a graph \( G \) is \( L = D - A \), where \( D \) is the Degree matrix, that is the diagonal matrix whose entries are the degrees of the vertices corresponding to each column/row, and \( A \) is defined as above. In other words its entries are computed as follows:

\[
L_{ij} = \begin{cases} 
d(i), & \text{if } i = j \\
-1, & \text{if } i \neq j \text{ and } (i,j) \text{ form a (un)directed edge} \\
0, & \text{otherwise.}
\end{cases}
\] (2.6)

Equivalently, the Laplacian matrix can be written as \( L = QQ^T \), where \( Q \) is the incidence matrix defined above.
Note that, by construction, any row or column of the matrix sums up to 0. Indeed, in every row or column \( i \) we have \( d(i) \) in exactly one (the \( i = j \)) position, and \(-1\) in \( d(i) \) position, once for every neighbour of \( i \). It follows that all the nonzero entries cancel each other out.

Same as with any matrix, one can compute the eigenvectors of the Laplacian matrix, that is the vectors \( v_i \) which satisfy

\[
L \cdot v_i = \lambda_i \cdot v_i, 
\]

where the \( \lambda_i \)'s are some ordered scalars such that \( \lambda_1 \geq \lambda_2 \geq ... \geq \lambda_n = 0 \). We say that the \( \lambda_i \) are the eigenvalues of the matrix.

Before stating the theorem that allows us to count the spanning trees of a given graph, let us first state and prove an important formula we are going to need in order to calculate the determinant of a product of matrices.

Let \( S \subset \{1, 2, ..., m\} \) and \( T \subset \{1, 2, ..., n\} \). From this point forward, given an \( m \times n \) matrix \( A \), we are going to denote by \( A[S|T] \) the sub-matrix of \( A \) consisting of the rows that correspond to the elements of \( S \) and the columns that correspond to those of \( T \).

**Lemma 2.1.1.** *(Cauchy-Binet formula)* Let \( m \leq n \) and \( A \) and \( B \) be matrices of size \( m \times n \) and \( n \times m \), respectively. Then

\[
\det(AB) = \sum_T \det(A[\{1, 2, ..., m\} | T]) \det(B[T | \{1, 2, ..., m\}]),
\]

where the sum runs over all subsets \( T \subset \{1, ..., n\} \) such that \(|T| = m\). In the case where \( m = n \), this simply translates to

\[
\det(AB) = \det(A)\det(B).
\]

**Proof.** For simplicity, let us use the following notation

\[
f(A, B) = \det(AB)
\]

\[
(2.10)
\]
and

$$g(A, B) = \sum_T \det(A_T)\det(B_T),$$  

(2.11)

where $A_T = A[\{1, \ldots, m\} | T]$ and $B_T = [T | \{1, \ldots, m\}]$.

Think of $A$ and $B$ as $n$-tuples in $\mathbb{R}^n$. Then we can write equivalently

$$f(A, B) = f(A_1, \ldots, A_n, B_1, \ldots, B_n)$$  

(2.12)

and

$$g(A, B) = g(A_1, \ldots, A_n, B_1, \ldots, B_n).$$  

(2.13)

Our goal is to show that the two functions $f$ and $g$ change in the same way when we modify $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$, one vector at a time.

First we analyse what happens when we use multiply $A_i$ or $B_i$ by a real scalar $a$. By properties of the determinant and the dot multiplication, we have:

- If $A_i$ is replaced by $a \cdot A_i$ then
  1. $f(A, B) = a \cdot f(A, B)$;
  2. $g(A, B) = a \cdot g(A, B)$

- If $B_i$ is replaced by $a \cdot B_i$ then
  1. $f(A, B) = a \cdot f(A, B)$;
  2. $g(A, B) = a \cdot g(A, B)$.

So $f$ and $g$ behave the same with respect to scalar multiplication.

Now let us see what happens to them when we turn one of the vectors into a sum.

- Let $A_i = A'_i + A''_i$. Then
  1. $f(A, B) = f(A'_i, B) + f(A''_i, B)$;
  2. $g(A, B) = g(A'_i, B) + g(A''_i, B)$,

where we denoted by $A'$ and $A''$ the lists obtained by changing $A_i$ into $A'_i$ and into $A''_i$, respectively.

- Let $B_i = B'_i + B''_i$. Then
1. \[ f(A, B) = f(A, B') + f(A, B''); \]
2. \[ g(A, B) = g(A, B') + g(A, B''), \]

where \( B'_i \) and \( B''_i \) are defined analogously to \( A'_i \) and \( A''_i \).

Therefore \( f \) and \( g \) change in the same way even with respect to addition.

Suppose that \( A_i = A_j \) for some indices \( i \) and \( j \). Then the determinant of \( A_T \) vanishes for all \( T \), i.e. \( \det(A_T) = 0 \), and so does \( \det(AB) \) since \( AB \) has a repeated row. The same conclusions hold when \( B_i = B_j \) for some \( i \) and \( j \). In these cases, the desired result would hold trivially.

Then, without loss of generality, we can assume that there are no two identical vectors in \( A \) or in \( B \). Matrices that have this property are made up of \( n \) 1’s, while all other entries are 0. Then all rows of matrix \( A \) are linearly independent, and so are all columns of \( B \).

This implies that there exist unique sets \( T_A \) and \( T_B \) of \( h \) elements such that \( \det(A_{T_A}) = \det(B_{T_B}) = 1 \), and that for all other sets \( T \), \( \det(A_T) = \det(B_T) = 0 \).

If \( T_A = T_B \), then \( g(A, B) = 1 \) and \( AB \) is the identity matrix, so \( f(A, B) = 1 \); if \( T_A \neq T_B \), then \( g(A, B) = 0 \) and \( AB \) has at most \( n-1 \) nonzero entries, so \( f(A, B) = 0 \).

Then, in either case, \( f(A, B) = g(A, B) \).

\[ \square \]

2.2 Matrix Tree Theorem

The following theorem, by Kirchoff, is an essential tool in algebraic graph theory, as it provides a way to count the number of spanning trees of any connected graph [17].

**Theorem 2.2.1.** (Matrix Tree Theorem) Let \( \lambda_n = 0 \) and \( \lambda_1 \cdot \lambda_2...\lambda_{n-1} \) are the nonzero eigenvalues of the Laplacian matrix, where \( \lambda_i > \lambda_{i+1} \) for all \( i \).

The number of spanning trees \( \tau(G) \) of an (un)directed graph \( G = (V, E) \) with \( |V| = n \) is

\[
\tau(G) = \frac{1}{n!} \lambda_1 \lambda_2...\lambda_{n-1} = \det(L_0),
\]

where \( L_0 \) is a principal minor of size \( n - 1 \).

**Proof.** Let \( G = (V, E) \) be a simple directed graph on \( n \) vertices and \( m \) edges. In the alternative case where \( G \) is undirected, the result can be proved analogously.
Since the Laplacian matrix has the property that the entries of each row or column adds to 0, we can turn any minor into a different minor by adding, interchanging or modifying the sign of the rows and columns. Consequently, no matter which row and corresponding column we remove from $L$, the determinant of $L_0$ will not vary.

Therefore, without loss of generality, we can consider the case where $L_0$ is obtained by deleting row $n$ and column $n$. We want to show that its determinant counts the number of spanning trees of $G$.

Since $L = QQ^T$, we also have that $L_0 = \tilde{Q}\tilde{Q}^T$, where $\tilde{Q}$ is the $(n-1) \times m$ matrix obtained by removing the $n$-th row from $Q$. By the Cauchy-Binet formula (2.8) for the determinant of a product of matrices, we have

$$
\det(L_0) = \det(\tilde{Q}\tilde{Q}^T) = \sum_T \det(\tilde{Q}[\{1,2,...,n-1\} \mid T]) \det(\tilde{Q}[T \mid \{1,2,...,n-1\}]) = \sum_T \det(\tilde{Q}[\{1,2,...,n-1\} \mid T])^2,
$$

(2.15)

where the summation runs over all subsets $T \subset \{1,...,m\}$ such that $|T| = n - 1$. Note that this is equivalent to summing over all subgraphs on $n - 1$ edges.

Let $H$ be the subgraph of $G$ whose $n - 1$ edges are represented by $T$. To prove that $\det(L_0)$ indeed counts the number of spanning trees, it suffices to show that $\det(\tilde{Q}[\{1,2,...,n-1\} \mid T]) = \pm 1$, whenever $T$ induces a spanning tree, and $\det(\tilde{Q}[\{1,2,...,n-1\} \mid T]) = 0$, otherwise.

Suppose $H$ is a subgraph of $G$ on which is not a spanning tree. Since $H$ has $n$ vertices and $n - 1$ edges, then it must be disconnected. Let us consider a component $H'$ of $H$, which does not contain vertex $n$.

By relabeling the vertices and edges of $G$, we can write $\tilde{Q}[\{1,2,...,n-1\} \mid T] = \tilde{Q}[\{1,2,...,n-1\} \mid T] = \begin{bmatrix} \tilde{Q}_1 & 0 \\ 0 & \tilde{Q}_2 \end{bmatrix}$,

(2.16)

where $\tilde{Q}_1$ is the incidence matrix of $H'$, so the edges and vertices of $H'$ all appear in the first quadrant of the matrix. Now since $\tilde{Q}_1$ has exactly two non-zero
entries per column, namely +1 and −1, all its rows when added are going to give det(\tilde{Q}_1) = 0, hence det(\tilde{Q}) = det(\tilde{Q}_1) \cdot det(\tilde{Q}_2) = 0.

Now let \( H \) be a subgraph of \( G \) on with \( n - 1 \) vertices which is a tree and \( T = \{t_1, ..., t_{n-1}\} \) be a subset of \([m]\). Since \( H \) is a tree, there are at least two vertices in \( H \) with degree exactly 1.

Denote by \( v_n \) the \( n \)-th vertex, whose row we previously removed. Then all vertices \( u_i \neq v_n \) can be relabeled them as follows: consider \( u_1 \) such that \( d(u_1) = 1 \). Then without loss of generality we can assign \( t_1 \) to \( u_1 \), and remove \( u_1 \) from \( H \). In the resulting graph \( Y \setminus \{u_1\} \), select \( u_2 \) such that \( d(u_2) = 1 \) in \( Y \), and let \( t_2 \) be its incident edge. We keep going until, by the end of this process we have all \( n - 1 \) vertices \( u_i \) different from \( v_n \) are assigned to the \( n - 1 \) edges in \( T \). Another way of picturing this is that we are ”trimming” one leaf a time, until we are left with no tree, where a leaf is any vertex with degree 1.

The matrix \( P \) obtained by relabeling the vertices corresponds to a permutation of \( Q[\{1, 2, ..., n-1\} \mid T] \), so the determinants of the two matrices must be the same. By construction, \( u_i \) is mapped to \( t_i \), for all \( i \in \{1, 2, ..., n-1\} \), thus all entries of the diagonal are either 1 or −1 and it is lower triangular. We conclude that

\[
\det(Q[\{1, 2, ..., n-1\} \mid T]) = \det(P) = \pm1. \tag{2.17}
\]

\[ \square \]

**Corollary 2.2.1.** Counting the spanning trees of a graph \( G \) can be done in \( O(n^3) \) time.

**Proof.** Since diagonalising an \( n \times n \)-matrix by Gaussian elimination can be done in \( O(n^3) \) time, and multiplying the diagonal entries only takes constant time, the time complexity of counting the spanning trees of a graph is \( O(n^3) \). \[ \square \]

### 2.3 Markov Chains

Markov chains are going to be an important tool for developing the algorithms we need in order to generate spanning trees at random, so let us discuss a little bit about them first [14].
Definition 2.3.1. A sequence of random variables \((X_t)_{t \geq 0}\) with state space \(S\) is a discrete-time Markov chain with transition matrix \(P\) if for all \(s, s_j \in S\), for all \(t \geq 1\), and for all events \(H_{t-1} = \bigcap_{r=0}^{t-1}\{X_r = s_r\}\), we have the so-called Markov property:

\[
Pr\{X_{t+1} = s_j | H_{t-1} \cap \{X_t = s_i\}\} = Pr\{X_{t+1} = s_j | X_t = s_i\} = P_{ij}. \tag{2.18}
\]

In (2.18), the variable \(t\) keeps track of the repetitions of this random process, while \(H_{t-1}\) can be regarded as the history or the sequence of states that occurred before a particular stage.

We say that the Markov chain is finite if its state space is finite-dimensional.

In other words, the Markov property requires that the conditional probability of going from one state, say \(x\), to another, say \(y\), to remain invariant under the different possible sequences of states that precede \(x\). This implies that in a Markov chain, the future events only depend on the present, and never on the past.

Let us expand a bit more on the transition matrix \(P\) in the definition. This is a \(|S| \times |S|\) matrix such that entry \(p_{ij}\) gives the probability that, given that we are starting at state \(s_i\), the next state is going to be \(s_j\). Note all these transition probabilities are therefore fixed, and \(P\) is stochastic. Indeed, all its entries are non-negative and all its rows add up to 1, since every row \(r_i\) gives the probability distribution conditional on \(s_i\) being the current state.

For any time \(t \geq 1\), we can store the information about the distribution in a vector of the form

\[
\mu_t = (Pr\{X_t = s_1|X_0 = s_i\}, Pr\{X_t = s_2|X_0 = s_i\}, ..., Pr\{X_t = s_k|X_0 = s_i\}), \tag{2.19}
\]

where \(k = |S|\). Notice how we do not need to include the states that the random variables between \(X_0\) and \(X_t\) took on, as this does not affect the probability.

It is easy to see that for \(t = 0\), the row vector \(\mu_0\) is the indicator vector of the initial state. This observations yields the following recursive formula: for all \(t \geq 1\),

\[
\mu_t = \mu_{t-1}P, \tag{2.20}
\]

14
which in turn implies that we can compute any vector \( \mu_t \) using the transition matrix and the initial distribution \( \mu_0 \) in this way:

\[
\mu_t = \mu_0 P^t,
\]

(2.21)

for any \( t \geq 0 \).

**Definition 2.3.2.** We say that a Markov chain \( (X_t)_{t \geq 0} \) is **irreducible** if for any two states \( x, y \in S \) we can find a \( t \) such that \( P^t(x, y) > 0 \). In other words, the Markov chain has only one closed class, and we can reach any state from any other through finitely many steps.

**Definition 2.3.3.** For a Markov chain \( (X_t)_{t \geq 0} \), we define the **hitting time** for a state \( x \) to be

\[
h(x) := \min\{t \geq 0 : X_t = x\},
\]

(2.22)

that is the first time that we encounter state \( x \) starting at initial state \( X_0 \).

In the case where \( X_0 = x \) is the initial state and therefore the hitting time is trivial, we might be interested in knowing the **first return time** instead:

\[
h^+(x) := \min\{t > 0 : X_t = x\}.
\]

(2.23)

**Definition 2.3.4.** We define the **cover time** of the Markov chain to be the time it requires for all possible states to be visited, so

\[
C := \max_{x \in S} h(x).
\]

(2.24)

By definition, all states of any irreducible chain the cover time is finite.

### 2.4 Closed classes

Given a graph \( G = (V, E) \), a **walk** of length \( r \) is a sequence of \( r + 1 \) vertices \( v_0, v_1, ..., v_r \) such that the edge \( v_{i-1}, v_i \) is in \( G \) for all \( i \) with \( 1 \leq i \leq r \).

Then we can define the relation \( \sim \) between two vertices: for two vertices \( u, v \in V \), we write that \( u \sim v \) if and only if there exists a walk from \( u \) to \( v \) and a walk from \( v \) to \( u \). This is an equivalence relation, for if we have \( u, v, w \in V \) then
• $u \sim u$, since there exists a walk of length zero from vertex $u$ to itself; so the relation is reflexive;

• $u \sim v$ implies that $v \sim u$; so the relation is symmetric;

• $u \sim v$ and $v \sim w$ implies that $u \sim w$; so it is also transitive.

This equivalence relation forms equivalence classes on the vertex set $V$, which we call strongly connected components. A strongly connected component is called a closed class when there are no outgoing edges. The vertices in a closed class are said to be recurrent, while the others are transient [13].

Then we can rephrase the definition of irreducible for a Markov chain by just saying it only has one closed class.

**Lemma 2.4.1.** For every graph $G = (V, E)$ and starting at any vertex in $V$, we can construct a walk that terminates in a closed class. In particular, any graph has at least one closed class.

**Proof.** Select any vertex $v$ in $G$, and let $C_1$ be the corresponding strongly connected component. If $C_1$ is a closed class, then we are done, because there exists a walk of length zero from $v$ to itself.

Then suppose that $C_1$ is not a closed class. It follows that there is at least one outgoing edge connecting a vertex $u_1 \in C_1$ to another vertex $u_2$ in another strongly connected component, say $C_2$. Since $v$ and $u_1$ are both in $C_1$, there is a walk going from $v$ to $u_1$. Then by transitivity, since $(u_1, u_2) \in E$ we have a walk going from $v$ to $u_2 \in C_2$. If $C_2$ is a closed class, then we have a walk starting at $v$ and terminating in a closed class, so we are done.

Again, suppose this is not the case. Then there is a walk connecting $u_2$ to another vertex $u_3 \in C_3$. We can concatenate the walks again to form one that takes $v$ to $C_3$.

Continuing with this process, we construct a sequence of strongly connected components $C_1, C_2, \ldots$ such that for all $i, C_i \neq C_{i+1}$ and for another index $j \geq i$ there exists a walk going from $v_i \in C_i$ which ends up in $C_j$. Then, since the number of components is finite, we have two options: either we end up in a component which we have already seen, or we eventually terminate in a closed class.
Let us analyse the first case, where we have a sequence \( C_1, C_2, \ldots, C_n \) such that \( C_n = C_i \) for some \( i < n - 1 \). Note that there exists a walk from vertex \( v_i \) in \( C_i \) to vertex \( v_{i+1} \) in \( C_{i+1} \), and one from vertex \( v_{i+1} \) in \( C_{i+1} \) to \( v_n \) in \( C_n \), because \( n > i + 1 \). Since \( C_n = C_i \), we have another walk from \( v_n \) to \( v_i \), so by concatenation we get one also going from \( v_{i+1} \) to \( v_i \). Since the walk exists in both directions between \( v_i \) and \( v_{i+1} \), this implies that \( v_i \sim v_{i+1} \). By definition of strongly connected components, the two vertices should then belong in the same one, giving \( C_i = C_{i+1} \). This contradicts our assumption, so we can rule out the possibility that the sequence will return to a previously visited component. Hence we conclude that the sequence terminates in a closed class, so any graph \( G \) has at least one closed class.

\[ \square \]

### 2.5 Random walks on graphs

Let us explain exactly how we are going to apply the notions we discussed to the graphs whose spanning trees we want to find [8].

Let \( G = (V, E) \) be a connected, finite graph of \( n \) vertices and \( m \) edges. A simple random walk on \( G \) can be described to be a trajectory along the vertices of \( G \), where from any vertex \( X_t = v_i \) at time \( t \), the next position \( X_{t+1} = v_j \) is chosen uniformly at random from the set of neighbours of \( v_i \). More generally, for a weighted graph, we have a function \( w : E \to (0, \infty) \) which assigns a number to every edge in \( E \). Then the successor \( X_{t+1} = v_j \) of \( v_i \) is chosen with probability proportional to the weight of the edge connecting \( v_i \) to \( v_j \).

From any Markov chain we can construct a random walk in the following way. Let \( (X_t)_{t \geq 0} \) be a Markov chain on \( V \) with transition matrix \( P \). Let \( \pi : V \to [0, 1] \) be the stationary distribution with respect to which the chain is reversible. Then for each pair of vertices \( u, v \in V \), we associate the weight

\[
w(u, v) = \pi(u)p_{uv}.
\]  

(2.25)

It follows that the weight function is symmetric, i.e. it satisfies the detailed balance equations \( w(u, v) = w(v, u) \) for all pairs of vertices \( u, v \in V \). Therefore we can express the transition probabilities in terms of the weights by

\[
p_{uv} = \frac{w(u, v)}{W(u)},
\]  

(2.26)

17
where $W(u)$ is the weighted outdegree of $u$ defined by $W(u) := \sum_{v \in V} w(u, v)$ for all vertices $u \in V$.

It follows that the entries of the transition matrix of a Markov chain associated to $G$ are

$$
(p_{uv}) = \begin{cases} 
\frac{1}{d(u)}, & \text{if } (u, v) \in E \\
0, & \text{otherwise},
\end{cases}
$$

(2.27)

when $G$ is unweighted, so when $w(u, v) = \frac{1}{m}$ for all pairs $(u, v) \in E$. If instead $G$ is weighted, these entries are

$$
(p_{uv}) = \begin{cases} 
\frac{w(u, v)}{W(u)}, & \text{if } (u, v) \in E \\
0, & \text{otherwise}.
\end{cases}
$$

(2.28)

We are mainly going to be focusing on unweighted graphs, so that the spanning trees that we sample follow the uniform distribution. The proofs can be adapted by substituting the transition probabilities defined in (2.28) to show that, in the weighted case, the distribution depends on the weight of a given tree.
Chapter 3

Exact sampling of spanning trees through exact counting

In this section, we are going to show that the problem of sampling uniform spanning trees of a given graph \( G = (V, E) \) can be reduced to that of merely counting the spanning trees of \( G \), which as we have seen can be done in polynomial time in \(|V|\).

3.1 Multigraphs

**Definition 3.1.1.** A *multigraph* is a graph in which we allow multiple edges between vertices, so edges which share both endpoints.

**Definition 3.1.2.** For a graph \( G = (V, E) \), we call *edge deletion* the operation that simply removes a given edge \( e = (u, v) \) from the graph. This leaves the vertex set \( V \) unchanged, while the edge set \( E \) is replaced with \( E \setminus \{e\} \).

**Definition 3.1.3.** The operation of *edge contraction* consists in removing \( e = (u, v) \) and simultaneously joining the two incident vertices \( u \) and \( v \) into a new vertex \( w \). We keep all the other edges of the original graph, including ones that are repeated. The resulting graph is a multigraph and has one less element in both the vertex set and the edge set.

Let \( G_1 = (V, E \setminus \{e\}) \) be the graph obtained by deleting \( e = (u, v) \) from \( G \), and \( G_2 = (\tilde{V}, \tilde{E}) \) be the graph obtained by contracting \( e = (u, v) \) in \( G \).
Consider the latter graph $G_2$. The contraction of $e$ may cause the graph to turn into a multigraph, in the case where $u$ and $v$ share one or more neighbours.

For example, say $u, v,$ and another vertex $w$ have edges connecting all of them in a cycle. then merging $u$ and $v$ into new vertex $x$ will cause $w$ and $x$ to be connected by two distinct edges.

More generally, in the case where $G$ was a multigraph to begin with, the number of edges between $w$ and $x$ in $G_2$ will be the sum of the number of edges between $w$ and $u$ and those between $w$ and $v$. Note that the Matrix tree theorem can be adapted to hold for multigraphs as well: one only needs to modify the Laplacian matrix $L$ of the graph, by taking entries $l_{ij} = -k$, where $k$ is the number of edges connecting vertex $i$ to vertex $j$ in the graph. Moreover, the degree of a given vertex takes into accounts all loops.

### 3.2 Reduction from sampling to counting

The key realisation in order to understand the link between exact counting and exact sampling lies in the fact that the set of spanning trees of the original graph $G$ can be partitioned into two disjoint subsets: the spanning trees that contain edge $e$, and those which do not.

This can be done by observing that the number of spanning trees of $G$ not containing $e$ in their edge set, is equivalent to the number of spanning trees in $G_1$, say $\tau(G_1)$. On the other hand, the number of spanning trees of $G$ which do contain $e$ can be viewed as the number of spanning trees of $G_2$, say $\tau(G_2)$, where two spanning trees are considered distinct if they have a different edge connecting $w$ with $x$.

It follows that we can express the total number of spanning trees in $G$ as a sum in this way:

$$\tau(G) = \tau(G_1) + \tau(G_2).$$  \hspace{1cm} (3.1)

Now let $T$ be a uniformly random spanning tree in $G$. One can calculate the probability of a particular edge $e$ being in the edge set of $T$ by using the above observation. We have that
\[ Pr(e \in T) = \frac{\tau(G_2)}{\tau(G_1) + \tau(G_2)}, \quad (3.2) \]

where we have the number of spanning trees containing \( e \) on the numerator, and the total number of spanning trees in the denominator.

So we can use this to construct \( T \) recursively by considering one edge at a time using a count tree. We start with the root of the tree \( r \) and label it with the whole graph \( G \). Then \( r \) has two children, say \( x \) and \( y \), which we label by \( G_1 \) and \( G_2 \), defined as above. We assign 0 and 1, to the edges connecting \( r \) to \( x \) and \( y \), respectively. At the next step, both children of \( r \) are going to have children of their own. The children of \( x \) are going to be labeled by the graph obtained by edge deletion from \( G_1 \) and the graph obtained by edge contraction of \( G_1 \), while the children of \( y \) are going to be labeled analogously using \( G_2 \). As before, the edges connecting \( x \) and \( y \) to their children are assigned 0 or 1, depending on whether we deleted or contracted the edge. This process goes on until we are left with no edges. The count tree is constructed so that its leaves correspond to the spanning trees of the \( G \).

### 3.3 Time complexity

Since counting the spanning trees of any multigraph can be done in \( O(n^3) \) time, as we have seen in Corollary 2.2.1, we can use this count tree to sample spanning trees uniformly at random in \( O(n^5) \) time.

Indeed, by (3.1), each time we decide whether a particular edge is present or not we count the spanning trees of graph \( G_1 \) and \( G_2 \), which we know can be done in \( O(n^3) \) time by calculating the determinants of the corresponding Laplacian matrices. Since there is a total of \( m = |E| \) edges, the total time complexity is \( O(n^5) \).
Chapter 4

Aldous-Broder Algorithm

Given a connected finite graph $G = (V, E)$, the Aldous-Broder algorithm outputs a uniformly distributed random spanning tree of $G$.

**Algorithm 1: Aldous-Broder**

**Input**: $G = (V, E)$ finite, connected

**Output**: Spanning tree $T$ of $G$

**Initialisation**: Choose initial state $X_0$ and run simple random walk with state space $V$ and transition probability as in (2.27);

**while not all vertices have been hit do**

**if vertex hit at time t has not yet been visited then**

- add the edge leading to said vertex to $T$;

**else**

- do not record the edge

The idea is to arbitrarily pick a vertex at which to start and run the simple random walk on $G$. As we move along the vertices, we add to the set of edges $T$. We only record those edges that terminate in a vertex which has not been visited before, as to avoid the formation of cycles [11].

Since the graph $G$ is connected, from any given vertex we can reach any other through a finite number of edges, so the Markov chain is irreducible and the cover time is finite. This means that the algorithm will terminate with probability 1.
The edges which we record during the algorithm can be written as the set

\[ T \subseteq E = \{X_{h(v) - 1}, X_{h(v)} \mid v \in V \setminus \{X_0\}\}, \quad (4.1) \]

where \( h(v) \) is the hitting time of \( v \) as defined in the previous chapter. Note that the set \( T \) has the following properties:

- \( T \) has no cycles: we only add the edges that bring us to vertices not yet visited;
- \( T \) is connected: by construction, since we apply the Markov chain repeatedly updating the initial state (or vertex) with the endpoint of the last edge considered;
- \( T \) visits every vertex in \( V \): we run the algorithm until we hit all vertices.

Hence the edges in \( T \) form a spanning tree of \( G \) by definition. Now we have left to show that the spanning trees produced by the algorithm are uniformly distributed, or, in the case on weighted tree, distributed proportionally according to their weights.

### 4.1 Backward and forward tree chains

Let \( t \) represent the time stages of a Markov chain. Then

\[ I_t = \bigcup_{0 \leq j \leq t} \{X_j\} \quad (4.2) \]

is the set of states or, in the case of a Markov chain on a graph, vertices visited by the chain in all steps up to and including stage \( t \).

We denote by \( l(i, t) \) the largest index in \([0, ..., t]\) such that at which vertex \( i \in I \) is hit. Then we can define a backward tree rooted at \( X_t \) by

\[ B_t = \{(X_{l(i,t)}, X_{l(i,t)+1}) \mid i \in I \setminus \{X_t\}\}. \quad (4.3) \]

If \( t \) exceeds the cover time then the set of edges in \( B_t \) form a spanning tree of \( G \). The random walk \( X_t \) induces a Markov chain \((B_t)_{t \geq 0}\), the backward tree chain.

Similarly, we can define a forward tree by letting \( f(i, t) \) be the first index in \([0, ..., t]\) to hit vertex \( i \), and setting

\[ F_t = \{(X_{f(i,t)}, X_{f(i,t)-1}) \mid i \in I \setminus \{X_t\}\}. \quad (4.4) \]
Observe that, unlike the backward tree chain which keeps on changing, the forward tree chain stays the same after $t$ exceeds the cover time, i.e. for all steps $t \geq C$, we have that $F_t = F_C$. Then Aldous-Broder algorithm described above outputs the spanning tree $F_C$.

**Lemma 4.1.1.** The distributions of backward tree $B_T$ and the forward tree $F_T$ are the same.

**Proof.** One can construct a backward tree from $X_0, X_1, ..., X_t$ by reversing the chain $X_t, X_{t-1}, ..., X_0$ and then computing the forward tree of the reversed chain. Since the distribution of a walk and its reverse are the same, we conclude that $B_T$ and $F_T$ also have equal distribution.

**Lemma 4.1.2.** The set of spanning trees $T$ of the graph $G$ is the unique closed class of $B_T$.

**Proof.** Let $G = (V, E)$ be a graph with $|V| = n$, and let $T = (V_T, E_T)$ be a non-spanning tree, then $|V_T| \leq n - 1$.

Suppose that $Pr(B_{t+1} = T' \mid B_t = T) > 0$, for some spanning tree $T' = (V'_T, E'_T)$, and some time step $t$. Since the steps in a backward tree chain never make the tree smaller, we know that $|V'_T| \geq |V_T|$.

Let $z \in V \setminus V_T$ be a vertex not in $T$. By irreducibility, we can go from $X_t$ to $z$ in a finite number of steps, so there exists some spanning tree $T'$ with $|V'_T| > |V_T|$ and

$$Pr(B_S = T' \text{ for some } s > t \mid B_t = T) > 0,$$  \hfill (4.5)

equivalently

$$Pr(B_S = T \text{ for some } s > t \mid B_t = T) < 1,$$  \hfill (4.6)

so the probability to remain in the set of non-spanning trees is $< 1$, and the class is not closed. Hence a non-spanning tree in $B_T$ is transient.

Now we want to show that for any pair of directed spanning trees of $G$, say $(T, T')$, we have a path of $s$ spanning trees starting at one and terminating at the other

$$T' = T_0, T_1, ..., T_s = T,$$  \hfill (4.7)
and that for all $i \in [1, s]$,

$$P r(B_{t+1} = T_i \mid B_t = T_{t-1}) > 0. \quad (4.8)$$

In order to show this, consider the set of leaves of $T$, denoted by $L$, and let $r$ and $r'$ be the roots of $T$ and $T'$, respectively. For any leaf $l \in L$, $P_{l,r}$ is the unique path travelling from $l$ to the root $r$ in $T$. Similarly, we can define the reverse path $P_{r,l}$.

Notice that, with an appropriate choice of edges to travel, one can obtain $T$ from $T'$. Indeed, suppose the two roots are distinct, $r \neq r'$. Then we can start the path by going from $r'$ to $r$. Then for every leaf $l$, we take $P_{r,l}$, followed by $P_{l,r}$. The path terminates in $r$, so the tree is rooted at $r$. Once we have completed this process, the backward tree will be $T$.

Since this holds for any pair of directed spanning trees, we conclude that the set which contains all of them $\mathcal{T}$ is a closed class. □

**Corollary 4.1.1.** $B_t$ has a unique stationary distribution supported on spanning trees.

**Proof.** This follows immediately from Lemma 4.1.2, as an irreducible chain has a positive stationary distribution if and only if all states are in the same closed class, so all the states are recurrent and $B_t$ has a unique stationary distribution. □

**Theorem 4.1.1.** *(Markov Chain-Tree)* The stationary distribution of $B_t$ is proportional to the weight of the tree, for every $T \in \mathcal{T}$,

$$\pi(T) = \frac{w(T)}{\sum_{T' \in \mathcal{T}} w(T)} \quad (4.9)$$

**Proof.** For all vertices $i \in V$,

$$\kappa(i) := \lim_{N \to \infty} \frac{1}{N} \sum_{t=1}^{N} P r(X_t = i) = \lim_{N \to \infty} \frac{1}{N} P r(B_t \text{ is rooted at } i) = \sum_{T \in \mathcal{T}_i} \pi(T), \quad (4.10)$$

where $\mathcal{T}_i$ is the set of spanning trees of $G$ rooted at $i$. Let $T^{(i)} \in \mathcal{T}_i$, and suppose that $B_{t+1} = T^{(i)}$. If $T'$ precedes $T^{(i)}$ in the backward chain, then

- there exists a vertex $j$ such that $i \neq j$ and $(i, j) \in T'$;
• there exists another vertex $k$ such that $k$ is the root of $T'$ and the vertex preceding $i$ in the path from $j$ to $i$ in $T^{(i)}$.

Observe that at any time step $t$ of the backward chain, the tree $B_t$ is rooted at $k$. Then we can define $T'$ from $T^{(i)}$ by writing

$$T' = T^{(i)} - (k, i) + (i, j).$$

(4.11)

Since, given $i$ and $j$, $k$ is fixed and we can compute the stationary distribution of $T^{(i)}$ by summing over all its choices:

$$\pi(T^{(i)}) = \sum_{T' \in T} \pi(T') Pr(B_{t+1} = T^{(i)} \mid B_t = T') = \sum_{j \in V, (i, j) \in E} \pi(T^{(i)} - (k, i) + (i, j)) Pr(X_{t+1} = i \mid X_t = k).$$

(4.12)

Since $T$ has root $k$, its weight is

$$w(T') = \prod_{x \in V, x \neq k} \frac{1}{d(x)}.$$

(4.13)

Moreover, by definition of the transition probabilities,

$$Pr(X_{t+1} = i \mid X_t = k) = p_{ik} = \frac{1}{d(k)}.$$

(4.14)

Then we have that

$$\sum_{j \in V, (i, j) \in E} \prod_{x \in V} \frac{1}{d(x)} = d(i) \prod_{x \in V} \frac{1}{d(x)} = \prod_{x \in V, x \neq i} \frac{1}{d(x)} = w(T^{(i)}).$$

(4.15)

Combining (4.12) and (4.15), we see that, for some constant $c$,

$$\pi(T) = c \cdot w(T).$$

(4.16)

Since $\pi(T)$ is a probability distribution, $\sum_{T \in \mathcal{T}} \pi(T) = 1$, so the constant is

$$c = \frac{1}{\sum_{T \in \mathcal{T}} w(T)},$$

(4.17)

hence

$$\sum_{T \in \mathcal{T}} \pi(T) = \sum_{T \in \mathcal{T}} \frac{w(T)}{\sum_{T' \in \mathcal{T}} w(T')} = 1,$$

(4.18)

which gives

$$\pi(T) = \frac{w(T)}{\sum_{T' \in \mathcal{T}} w(T')},$$

(4.19)

as required.
Corollary 4.1.2. The stationary distribution of $F_t$ is also proportional to the weight.

Proof. This is a trivial consequence of Theorem 4.1.1 and Lemma 4.1.1. □

4.2 Correctness

Theorem 4.2.1. The Aldous-Broder algorithm outputs a uniformly distributed spanning tree.

Proof. Since the algorithm chooses the initial state $s$ arbitrarily, we cannot directly apply Corollary 4.1.2 here. Nonetheless, this can be easily fixed.

Suppose we start from $s$ instead of a $\pi$-random vertex and let $T_s$ be the set of all spanning trees starting at $s$. Since the weight

$$w(T) = \prod_{e \in E_T} w(e) = \prod_{v \in V_T, v \neq s} \frac{1}{d(v)} \quad (4.20)$$

is the same for all directed trees in $T \in T_s$, then

$$Pr(F_C = T) = \begin{cases} \frac{w(T)}{\sum_{T' \in T} w(T')} = \frac{1}{|T_s|}, & \text{if } T \in T_s \\ 0, & \text{otherwise.} \end{cases} \quad (4.21)$$

Moreover, to any directed tree in $T_s$ corresponds exactly one undirected spanning tree, as we can simply disregard the orientation of the edges and the root. Similarly, to any undirected spanning tree corresponds a directed tree from $T_s$, which we can obtain by rooting the tree at $s$ and directing all the edges towards the root. This shows there is a bijection between $T_s$ and the set of all undirected spanning trees of a graph, so for any undirected spanning tree $T$, we have

$$Pr(F_C = T) = \frac{1}{|T|}, \quad (4.22)$$

so all spanning trees are equally likely. □

4.3 Time complexity

Let us denote by $\eta_{i,j}$ the expected number of transitions needed to reach vertex $j$ from vertex $i$. When $i = j$, we say that $\eta_{i,i}$ is the mean recurrence time. Moreover, we define the mean commute time to be the sum $\eta_{i,j} + \eta_{j,i}$ [2].
Lemma 4.3.1. For a random walk on a graph on \( n \) vertices and \( m \) edges, the mean recurrence time is \( \eta_{i,i} = \frac{2m}{d(i)} \) for all \( i \).

Proof. Let \( \pi(i) \) be the stationary probability of vertex \( i \). Then the vector of probability
\[
\pi = (\pi(1), \pi(2), ..., \pi(n)) \tag{4.23}
\]
is such that \( \pi P = \pi \) and \( \sum_{i=1}^{n} \pi(i) = 1 \).

By substitution, since the entries of \( P \) are \( p_{ij} = \frac{1}{d(i)} \), we have that \( \pi(i) = \frac{d(i)}{2m} \).

Notice that on average a chain visits \( i \) once every \( \eta_{i,i} \) time, so the mean recurrence time of a state is the reciprocal of the stationary probability of said state \([19]\), that is
\[
\eta_{i,i} = \frac{1}{\pi(i)} = \frac{2m}{d(i)}, \tag{4.24}
\]

\( \square \)

Lemma 4.3.2. Let \( G = (V,E) \) be a graph on \( m \) edges. If \( (i,j) \in E \), then mean commute time of \( i \) and \( j \) is
\[
\eta_{i,j} + \eta_{j,i} \leq 2m. \tag{4.25}
\]

Proof. We are now looking at the expected number of transitions in a round trip, from \( i \) to \( j \) and then back to \( i \). All transitions are such that they happen with the same long-run frequency, that is to say that for a very long random walk on the graph, we expect that every edge is traversed in each direction every \( 2|E| = m \) steps. Then it follows that if we start on vertex \( i \), and \( j \) is adjacent to \( i \), then we expect to pass through edge \((j,i)\) within \( 2m \) time steps. Therefore,
\[
\eta_{i,j} + \eta_{j,i} \leq 2m. \tag{4.26}
\]

\( \square \)

More generally, for a pair of vertices which are not necessarily adjacent, we have the following bound.

Lemma 4.3.3. Let \( G = (V,E) \) be a graph. For any two vertices \( i, j \in V \) such that \( i \neq j \),
\[
\eta_{i,j} + \eta_{j,i} \leq 2m \cdot \Delta(i,j), \tag{4.27}
\]
where \( \Delta(i,j) \) is the distance between vertices \( i \) and \( j \).
Proof. We prove this by induction on $\Delta(i,j)$. Let $\Delta(i,j) = 1$ for the base case. Then (4.27) holds by Lemma 4.3.2. Now suppose the result holds for all $(i,j)$ with $\Delta(i,j) \leq r$, and consider the case where $\Delta(i,j) = r + 1$.

There exists another vertex $k \in V$ such that $k$ is adjacent to $j$, i.e. $(j,k) \in E$, and $\Delta(i,k) = r$. Then

$$\eta_{i,j} + \eta_{j,i} \leq \eta_{i,k} + \eta_{k,i} + \eta_{j,k} \leq 2m \cdot r + 2m = 2m(r + 1),$$

(4.28)

where we used the inductive hypothesis and Lemma 4.3.2 again. \hfill \Box

**Theorem 4.3.1.** Let $G = (V, E)$ be a graph on $n$ vertices and $m$ edges. The cover time satisfies

$$C \leq 2m(n - 1).$$

(4.29)

**Proof.** Let $T = (V_T, E_T)$ be a spanning tree of $G$. Then $T$ has exactly $n - 1$ edges and there exists a walk $i = i_0, i_1, ..., i_{2n-2} = i$ which travels through all the edges of $T$ exactly once in each direction.

The cover time is clearly less than the time it takes the Markov chain to visit all vertices in the walk we have constructed. Therefore

$$C \leq \eta_{i_0,i_1} + \eta_{i_1,i_2} + ... + \eta_{i_{2n-3},i_{2n-2}} = \sum_{(i,j) \in E_T} \eta_{i,j} + \eta_{j,i} = 2m(n - 1),$$

(4.30)

by Lemma 4.3.3.

Since the Aldous-Broder algorithm runs within the cover time of the given graph, its time complexity is $O(mn)$ or, equivalently $O(n^3)$ [5].

However, the time is only as bad as $O(n^3)$ in the worst possible case. It can be shown that the cover time is as small as $O(n \log n)$ whenever the second largest eigenvalue of the transition matrix $P$ is bounded away from 1. As it turns out, this happens for almost all graphs [4]. \hfill \Box
Chapter 5

Wilson’s algorithm

In this section we present an algorithm due to Wilson which, for a directed graph \( G = (V, E) \), produces a rooted spanning tree \((T, r)\) uniformly at random. Note that, since for any given tree we can pick a root arbitrarily, sampling a rooted tree is no different to sampling an unrooted one, so this algorithm can be used for undirected graphs as well. We only fix a root for simplicity, since the algorithm generates an oriented tree.

5.1 Loop-erased random walks

An important concept we need in order to understand Wilson’s algorithm is that of loop erased random walks. Let \( X_0 = x, X_1, X_2, ... \) be a simple random walk on the graph \( G \); a loop erased random walk from vertex \( x \in V \) to \( A \subset V \) can be constructed in the following way: we first consider the path

\[
\gamma = (X_0, X_1, ..., X_{T_A}), \tag{5.1}
\]

where \( T_A \) is the stopping time defined by

\[
T_A = \min\{n \geq 0 \mid X_n \in A\}. \tag{5.2}
\]

Walking along \( \gamma \), every time we visit a vertex that we have already seen, we erase the cycle, or 'loop', which we just gave rise to. Then all loops are erased chronologically, in the order in which they appear. When all loops are erased at the end of this process, we are left with a self-avoiding path from \( x \) to \( A \), that is with a path that does not intersect itself at any point. This path is the loop-erased path \( LE(x, A) \).
Algorithm 2: Wilson

Input: $G = (V, E)$ finite, connected

Output: Spanning tree $T$ of $G$

Initialisation: $T(0) = \{r\}$ and choose an ordering $\{v_1, v_2, ..., v_n\}$ of the remaining vertices. Assume that at any stage $i$, $T(i)$ is known;

while $T(i) \neq V$ do
  take the first vertex $v_j$ not in $T(i)$ and start a random walk at $v_j$;
  if the random walk hits some vertex in $T(i)$ then
    let $[v_j, T(i)]$ be the walk from $v_j$ to $T(i)$ and set
    $T(i+1) = T(i) \cup LE([v_j, T(i)])$
  else
    keep going

5.2 Stacks

Before proving the correctness of Wilson’s algorithm, we introduce the notion of stacks, which helps with the visualisation of the construction of the tree in this specific algorithm. We define the stacks to be the random variables

$$(S_{x,i} \mid x \in V \setminus \{r\}, i \in \mathbb{N}),$$

which are all independent from one another and have probability

$$Pr(S_{x,i} = y) = p_{xy}. \quad (5.4)$$

so the stack points at a random neighbour of $x$, for all vertices $x$.

The idea is to construct a simple random walk using these stacks. We choose as the initial state $X_0$ a vertex $v_1 \neq v_0$, and use the corresponding stack $S_{v_1,1}$ to find a random neighbour of $v_1$, say $w$. We draw a directed edge from $v_1$ to $w$, set $X_1 = w$, and discard the value $S_{v_1,1}$, so that now the top element of the stack is $S_{v_1,2}$. We use $S_{w,1}$ to select another neighbour, which is going to become $X_2$, pop the stack which pointed us to that neighbour, and continue until we hit the already constructed tree.

At any time in the walk, the top items of the stacks form a directed graph, which we refer to as the visible graph. If no cycles arise in the process, then by the
end we will have a spanning tree. If instead we hit a vertex which we have already visited, we know a cycle is formed, so we select all the edges of the visible graph at this time and we replace $S_{x,i}$ with $S_{x,i+1}$ for every $x$ in the cycle. This is what we call ‘popping a cycle’. We keep popping all cycles until they are all gone and we are left with a spanning tree of $G$.

5.3 Colouring

To keep track of what level of the stacks each directed edge comes from, we are going to assign a colour to all of them. So to an edge which is identified by vertex $x$ and stack $S_{x,i}$, we are going to give colour $i$, for all $x \in V$. We call coloured cycles those which have all edges of the same colour. In the first step of this popping cycles algorithm, all edges have colour 1, so any cycles which may appear will be coloured. However, as we progress, we will get cycles whose edges come from different levels and which therefore are not monochromatic.

With this idea of loop erasure described by using stacks and then popping the cycles, we can prove the theorem of correctness of Wilson’s algorithm [15].

5.4 Correctness

**Theorem 5.4.1.** Wilson’s algorithm terminates with probability 1, returning a spanning tree of $G$ uniformly at random, or if the edges are weighted with probability proportional to its weight as in (2.25).

**Proof.** Building on our previous characterisation of Wilson’s algorithm using the notion of popping cycles, we want to show that the resulting spanning tree is invariant under the order in which the cycles are popped. Let $C_1, ..., C_n$ be a sequence of coloured cycles, which can be popped in this order to get a tree, and let $D_1, D_2, ..., D_m$ be another sequence of cycles that can be popped.

By induction we are going to show that the new sequence cannot possibly be made up by different cycles from the first, even if they are ordered differently. If $n = 0$, the result is trivial, as there are no cycles to be popped and both sequences are empty. For $m \geq 1$, let us assume that the statement is true when the length of
the first sequence is less than \( n \). Take the first cycle \( D_i \) which shares a vertex with \( C_1 \), and consider \( x \in D_i \cap C_1 \). Since \( D_i \) is the first cycle in the second sequence that intersects \( C_1 \), \( x \) is not contained in \( D_1, \ldots, D_{i-1} \). Hence the edge in \( D_i \) starting at \( x \) also has colour 1 and it ends at the same vertex \( y \) as in \( C_1 \). A similar reasoning applies to \( y \), and all subsequent vertices. Therefore \( D_i \) and \( C_1 \) represent the same cycle. Then popping \( D_1, D_2, \ldots, D_m \) gives rise to the same tree as \( C_1, D_1, \ldots, D_{i-1}, D_{i+1}, \ldots \). Once \( D_i = C_1 \) is popped, we can use the induction hypothesis and conclude that the two sequences uncover the same spanning tree.

Then since our algorithm can be seen as a method of popping the cycles in a particular order, it will result into the same tree distribution as any other method.

Let us show that for an unweighted tree, the distribution of the rooted trees is uniform. First, define the descendant \( D(x, T) \), that is the nearest vertex to \( x \) in the unique path from \( x \) to the root of the tree \( r \). Now fix a spanning tree \( T = (V_T, E_T) \) for \( G \). Then the probability of seeing that given tree \( T \) on top of the stacks is

\[
Pr(T) = \prod_{x \neq r} Pr(S_{x, 1} = D(x, T)) = \prod_{x \neq r} \frac{1}{d(x)} = d(r) \prod_{x \in V_T} \frac{1}{d(x)},
\]

(5.5)

where we used the transition probability as defined in (2.27), and independence of the random variables \( S_{x,i} \). Observe that the final term does not at all depend on the tree we chose, and call this quantity \( p_G \).

We can also calculate the probability of a particular labelled cycle appearing. Let \( v_0 = r \) be the root of the spanning tree, and

\[
C = (v_0, i_0), (v_1, i_1), \ldots, (v_k, i_k),
\]

(5.6)

with \( S_{v_j, i_j} = v_{j+1} \) for all \( j \). The probability of this cycle \( C \) to be formed is the product of probabilities of all edge of \( C \) occurring:

\[
Pr(C) = Pr(v_0, v_1) \cdot Pr(v_1, v_2) \cdot \ldots \cdot Pr(v_k, v_0) = \prod_{j=0}^{k-1} Pr(v_j, v_{j+1}) = \prod_{j=0}^{k-1} \frac{1}{d(v_j)}.
\]

(5.7)

As we showed above, the resulting tree of the algorithm is independent of the order in which the cycles are popped, so the probability of popping a set of cycles
\[ C = C_1, C_2, \ldots, C_n \] and uncovering vertex \( T \) is given by

\[
Pr(T \mid C_i) = \prod_{i=1}^{n} Pr(C_i) \cdot Pr(T) = Pr(C) \cdot p_G.
\] (5.8)

Since the probability is the same for all spanning trees, then they are uniformly distributed and all equally likely to appear.

In the case where we consider a weighted graph, the proof can be constructed analogously by substituting the transition probability from (2.27) with that of (2.28). Then the probability of a tree appearing, given that the cycles \( C_i \) have been popped is proportional to the weight of the tree \( T = (V_T, E_T) \), i.e.

\[
\Psi(T) = \prod_{(u,v) \in E_T} p_{uv} = \frac{\prod_{(u,v) \in E_T} c(u,v)}{\prod_{x \in V, x \neq \text{root}(T)} \pi(x)}.
\] (5.9)

\[
5.5 \text{ Time complexity}
\]

Let \( \eta_{i,j} \) be the expected number of steps it takes to go from vertex \( i \) to vertex \( j \) as before. Since \( \eta_{i,j} \leq C \) for all pairs \( i, j \), the mean commute time can always be bounded above by twice the cover time.

Let \( \pi \) be the stationary distribution. We define the mean hitting time \( \zeta \) as the expected time it takes to go from a \( \pi \)-random vertex to another \( \pi \)-random vertex

\[
\zeta = \sum_{i,j} \pi(i)\pi(j)\eta_{i,j}.
\] (5.10)

We want to know how many times we expect to have to use that stacks to select a random neighbour. Since we have shown that the ordering of the cycle-popping is irrelevant to the resulting tree, suppose we start at vertex \( u \). The expected number of times that the random walk returns to \( u \) before reaching the root \( r \) is given by

\[
\pi(u)(\eta_{u,r} + \eta_{r,u}),
\] (5.11)

where the number of times includes time \( t = 0 \) [1]. So the number of times we expect to use the stacks in Wilson’s algorithm is the sum over all \( u \in V \):

\[
\sum_{u \in V} \pi(u)(\eta_{u,r} + \eta_{r,u}) = 2\zeta
\] (5.12)
Moreover, the time it takes to create the loop-erasure of a path and connect the vertices to make the tree is $\zeta$, therefore the whole algorithm runs in $O(\zeta)$ time.

Note that

$$\zeta = \sum_{i,j} \pi(i)\pi(j)\eta_{i,j} \leq \max_{i,j}(\eta_{i,j} + \eta_{j,i}) \leq 2C,$$

so in most instances, Wilson’s algorithm returns a uniform spanning tree more quickly than the previous two algorithms.

The worst case scenario for the time complexity of Wilson’s algorithm is the one where we have a barbell graph: this is a graph which consists of two cliques of size $\frac{n}{3}$, connected by a path of length $\frac{n}{3}$. In this case the mean hitting time will coincide with the cover time and it will take us $O(n^3)$ time to run the algorithm.
Chapter 6

Partial Rejection Sampling
Algorithm.

The task of uniform sampling trees of a graph can be interpreted as that of avoidance of a finite sequence of “bad” events, when we consider said events to be cycles in the graph we generate [9].

6.1 Lovász local lemma

First, we are interested in seeing whether the complete avoidance of all these events is even possible.

For instances where the sequence of bad events $A_1, A_2, ..., A_n$ is such that

$$\sum_{i=1}^{n} P_r(A_i) < 1$$

or those where all events in the sequence are independent, then of course this can be done with probability strictly larger than 0.

However, in most cases the sum of probabilities of events in the sequence is going to exceed 1, and the events may depend on one another, so we want to know whether all bad events can still be avoided in these cases.

**Definition 6.1.1.** The *dependency graph* is a graph which represents the dependency relations between some given events. In other words, if we assign the event
A_i to vertex i for all i, then we have that A_i is mutually independent of the set of events \{A_j \mid (i, j) \notin E, i \neq j\}.

**Theorem 6.1.1.** (Lovász local lemma.) Suppose we have a sequence A_1, A_2, ..., A_m of “bad” events and let D = (V, E) be their dependency directed graph. If there exists a real vector (x_1, x_2, ..., x_m) ∈ [0, 1]^m such that for all i ∈ [s]

\[
Pr(A_i) \leq x_i \prod_{(i,j) \in E} (1 - x_j),
\]

then we have

\[
Pr\left( \bigcap_{i=1}^{m} \tilde{A}_i \right) \geq \prod_{i=1}^{m} (1 - x_i) > 0. \tag{6.3}
\]

**Proof.** In order to simplify the notation of the proof a little, for a subset S ∈ [m], let

\[
\tilde{P}_S := Pr\left( \bigcap_{i \in S} \tilde{A}_i \right), \tag{6.4}
\]

and take \(\tilde{P}_\emptyset := 1\).

We are going to show by induction that, for all S ∈ [m], and all k ∈ S,

\[
\tilde{P}_S \geq (1 - x_k) \cdot \tilde{P}_{S \setminus \{k\}} > 0. \tag{6.5}
\]

For the base case, this holds trivially, since we have

\[
\frac{\tilde{P}_{\{k\}}}{\tilde{P}_\emptyset} = Pr(\tilde{A}_k) \geq 1 - x_k \prod_{(a,j) \in E} (1 - x_j) \geq 1 - x_k. \tag{6.6}
\]

Now assume that (6.5) holds for all subsets \(S' \in [m]\), with |S| ≤ r, and let S ∈ [m] be of size r + 1. We define the *neighbourhood* of k ∈ S

\[
N(k) := \{l \in V : (k, l) \in E\}, \tag{6.7}
\]

and its *closure*

\[
N^+(k) := \{k\} \cup N(k). \tag{6.8}
\]
Then for \( k \in S \) fixed, we have
\[
\bar{P}_S = \Pr \left( \bigcap_{i \in S} \bar{A}_i \right) = \Pr \left( \bigcap_{i \in S \setminus \{k\}} \bar{A}_i \right) - \Pr \left( A_k \cap \bigcap_{i \in S \setminus \{k\}} \bar{A}_i \right) \\
\geq \Pr \left( \bigcap_{i \in S \setminus \{k\}} \bar{A}_i \right) - \Pr \left( A_k \cap \bigcap_{i \in S \setminus N^+(k)} \bar{A}_i \right) \\
= \bar{P}_{S \setminus \{k\}} - \Pr(A_k) \bar{P}_{S \setminus N^+(a)} ,
\]
where the last equality is due to mutual independence between \( A_k \) and the set of events with indices which are distinct from \( k \) nor they are in its neighbourhood, i.e. \( k, \{ A_i | i \notin N^+(k) \} \).

Dividing through by \( \bar{P}_{S \setminus \{k\}} \) in (6.9), we get
\[
\frac{\bar{P}_S}{\bar{P}_{S \setminus \{k\}}} \geq 1 - \Pr(A_k) \cdot \frac{\bar{P}_{S \setminus N^+(k)}}{\bar{P}_{S \setminus \{k\}}} . \tag{6.10}
\]
Since \( S \setminus \{k\} \) has size \( r \), then by inductive hypothesis, \( \bar{P}_{S \setminus \{k\}} > 0 \). Now consider intersection \( N(k) \cap S = \{ b_1, b_2, ..., b_d \} \), where \( d \geq 0 \). We can write
\[
\frac{\bar{P}_{S \setminus N^+(k)}}{\bar{P}_{S \setminus \{k\}}} = \frac{\bar{P}_{S \setminus \{k,b_1\}}}{\bar{P}_{S \setminus \{k\}}} \cdot \frac{\bar{P}_{S \setminus \{k,b_1,b_2\}}}{\bar{P}_{S \setminus \{k,b_1\}}} \cdot \cdots \cdot \frac{\bar{P}_{S \setminus \{k,b_1,b_2,...,b_d\}}}{\bar{P}_{S \setminus \{k,b_1,b_2,...,b_{d-1}\}}} . \tag{6.11}
\]
Now, by inductive hypothesis we know that all terms on the right hand side are strictly positive and are bounded by \( \frac{1}{1-x_b} \), so
\[
\frac{\bar{P}_{S \setminus N^+(k)}}{\bar{P}_{S \setminus \{k\}}} \leq \frac{1}{1-x_{b_1}} \cdot \frac{1}{1-x_{b_2}} \cdot \cdots \cdot \frac{1}{1-x_{b_d}} . \tag{6.12}
\]
By hypothesis of the lemma, \( \Pr(A_i) \leq x_i \prod_{b \in N^+(k)} (1 - x_b) \), hence, substituting this in our previous inequality (6.10),
\[
\frac{\bar{P}_S}{\bar{P}_{S \setminus \{k\}}} \geq (1 - x_k) \prod_{b \in N(k)} (1 - x_b) \prod_{c \in N(k) \cap S} \frac{1}{1-x_c} \geq 1 - x_k > 0 . \tag{6.13}
\]
Now we have proved that
\[
\frac{\bar{P}_S}{\bar{P}_{S \setminus \{k\}}} \geq 1 - x_k \tag{6.14}
\]
for all \( S \in [m] \) and \( k \in S \), we can easily see that this implies the Lovász local lemma holds, since
\[
\Pr \left( \bigcap_{i=1}^m \bar{A}_i \right) = \bar{P}_{[m]} \geq (1 - x_m) \bar{P}_{[m-1]} = (1 - x_m)(1 - x_{m-1}) \bar{P}_{[m-2]} \geq ... \tag{6.15}
\]
\[\geq \prod_{i=1}^m (1 - x_i) > 0, \]
as desired. \( \square \)
6.2 Construction

Let us first look at the intuition behind how one can construct a partial rejection algorithm for a generic case where we want to avoid some given events, drawing samples from a product distribution $\mu(\cdot)$ of all random variables.

Indeed, in this chapter, we will only be considering product spaces, that is cases where we have mutually independent random variables $X_1, X_2, ..., X_n$ and the events $A_1, A_2, ..., A_m$ depend on a subset of them, namely $\text{var}(A_i)$.

Since the avoidance of bad events is attainable under the condition of Theorem 6.1.1, we could think about using this to generate scenarios free of the undesired events in this way:

- Initialise the variables randomly using the respective distributions.
- If no bad events occur, then we are done. If one or multiple bad events $A_i$ occur, arbitrarily pick one of the bad events and resample all $\text{var}(A_i)$, where $\text{var}(A_i)$ is the index set of all random variables that the corresponding bad event depends on.
- Output the new assignment.
- Repeat until the output includes no bad events.

The issue with this procedure is that in general the outcome will not be uniformly distributed, as it will be inevitably biased if there are elements which belong to $\text{var}(A_i)$ for more than one $i$.

In order to achieve the conditioned product distribution we want, we need to add a crucial condition.

**Condition 6.2.1.** We require the intersection of any two dependent bad events to be empty. That is, if $(i, j)$ is an edge in the dependency graph, then we have $Pr(A_i \cap A_j) = 0$. We define cases where this condition is satisfied as *extremal*.

If this condition is satisfied, then the occurring of bad events forms an independent set on the dependency graph and therefore we can resample in a parallel fashion. This observation leads to the following improved algorithm:
Algorithm 3: General Partial Rejection Sampling

Input : Random variables $X_1, X_2, ..., X_n$ with respective distributions;

Output: Assignment of variables which avoid all bad events $A_i$;

Initialisation: Draw independent samples of all variables $X_1, X_2, ..., X_n$ from their respective distributions:

while there is at least one event $A_i$ occurring: do

find the independent set $I$ of occurring $A_i$'s on dependency graph and independently resample all the variables in $\bigcup_{i \in I} \text{var}(A_i)$;

We will see later that this revisited version of the original algorithm gives us the distribution needed: this is because, as soon as we require any two bad events to be disjoint, we automatically get that $\text{var}(A_i) \cap \text{var}(A_j)$ is empty for any $i, j \in I$, where $I$ is defined as above. So we can resample safely the variables of one event, without interfering with any other in the process.

This can be adapted to the specific case in which we are given a graph with root $r$ and we want to sample uniformly at random a spanning tree rooted at $r$. We let the state space be the vertex set $V$ of the graph $G$, and we take the random variables $X_1, X_2, ..., X_n$ as a choice of neighbour for any vertex in $V$. In this case, we consider the appearance of cycles to be the events we want to avoid, and define two cycles to be dependent with one another if they share one or more vertices.

Call $A_C$ the event that cycle $C$ is formed. Then $\text{var}(A_C) = V_C$, where $C = (V_C, E_C)$. Let us show that the cycles satisfy the condition of extremal events. Suppose there are two distinct cycles $C$ and $C'$ present, and let $v \in V_C \cap V_C'$ be a shared vertex. Then we can start from $v$ and follow an arrow $X_v$ from $v$ to $v'$. Since both cycles are present, it must be the case that $v' \in V_C \cap V_C'$. We can keep following arrows until at some point we get back to $v$. This implies that $C = C'$, a contradiction, so $Pr(A_C \cap A_{C'}) = 0$ unless $C = C'$ and the condition is satisfied.

For every vertex $v$ other than the root, let us assign a random variable, which we can think of as an arrow pointing to one of the neighbours. Then the following algorithm gives us a way to sample spanning trees uniformly.
Algorithm 4: Partial Rejection Sampling for Spanning Trees

Input : $G = (V, E)$ finite, connected;
Output: Uniformly chosen spanning tree $T$ of $G$;

Initialisation: Set $T = \emptyset$ and choose $r$ uniformly at random. For every vertex $v \neq r$ choose a neighbour $u$ randomly and add $[v, u]$ to $T$:

while there is at least one cycle in $T$; do

remove from $T$ all edges of all cycles and for the vertices whose edges are removed, randomly choose a neighbour again and add the corresponding edge to $T$;

end while

6.3 Correctness

The procedure of the partial rejection algorithm can be described using a resampling table. Suppose for each of the bad event, we have a processor looking at the random variables associated with that particular event in order to decide whether the event has occurred or not. If the event has occurred, we want to resample the corresponding variables. The way we do this is by assigning an infinite stack of random values $\{X_{i,1}, X_{i,2}, \ldots\}$ to each variable $X_i$. Observe how this idea is equivalent to the cycle-popping procedure we used for Wilson’s algorithm.

The resampling table is going to have $i$ rows, one for every random variable $X_i$ and an infinite amount of columns, which we are going to move across (to the right) when needing to select a different random value. Let $t$ represent the round of the algorithm, and suppose that at time $t$, the random variable $X_i$ takes on value $X_{i,j_{i,t}}$. Then the set

$$\sigma_t = \{X_{i,j_{i,t}} \mid 1 \leq i \leq n\}$$ (6.16)

contains the information about the current assignments and therefore determines which events happen. By Condition 6.2.1, the set of events $I_t$ happening at any round forms an independent set of $G$, so one can resample the variables associated to the events by doing

$$j_{i,t+1} = \begin{cases} j_{i,t} + 1, & \text{if there is } l \text{ such that } i \in \text{var}(A_l) \\ j_{i,t}, & \text{otherwise} \end{cases}$$ (6.17)

From this, we see that any event occurring in round $t+1$ must have at least
one variable in common with an event from $I_l$. In other words, $I_{l+1} \subset N^+(I_l)$, where $N^+$ denotes the closure of the set.

**Definition 6.3.1.** We say that a list $S = S_1, S_2, \ldots, S_l$ of independent sets in $G$ is an *independent set sequence* if, for all $i \in [1, l - 1]$, $S_i \neq \emptyset$ and $S_{i+1} \subset N^+(S_i)$.

**Definition 6.3.2.** Let $l \geq 1$. We define the log of running the algorithm on the resampling table up to round $l$ to be the sequence $I_1, I_2, \ldots, I_l$ of independent sets created in the process.

Then the sequence given by the log gives an independent set sequence, as defined in Definition 6.3.1.

**Definition 6.3.3.** We call a assignment $\sigma$ *valid* whenever none of the bad events $A_i$ happen where $i \notin N^+(S_l)$.

Take $T$ to be the round at which the algorithm terminates, and let $\sigma(t) = \sigma(T)$ for all $t \geq T$.

**Lemma 6.3.1.** Suppose condition 6.2.1 holds and let log $S = S_1, S_2, \ldots, S_l$ of length $l \geq 1$. Condition on seeing the events in log $S$, $\sigma_{l+1}$ is a random sample from

$$
\mu\left( \cdot \middle| \bigcap_{i \in [m] \setminus N^+(S_l)} \bar{A}_i \right),
$$

the product distribution conditioned on the avoidance of the bad events whose indices are not in the closure of $S_l$.

**Proof.** $S_l$ is the set of events occurring at round $l$, so $\sigma_{l+1}$ is valid.

We can shorten the resampling table of the algorithm to a table

$$
M = \{X_{i,k} \mid 1 \leq i \leq n, 1 \leq k \leq j_{i,l+1}\},
$$

since we are only interested in the first $l - 1$ columns. Let us now define another table

$$
M' = \{X'_{i,k} \mid 1 \leq i \leq n, 1 \leq k \leq j_{i,l+1}\}
$$

where the random values $X'_{i,j}$ only change in the final round $l + 1$ and exclusively to another valid assignment, so we have $X_{i,k} = X'_{i,k}$ for all $k \leq j_{i,l}$.

We claim that $M$ and $M'$ generate the same log $S$. Suppose this is not the case and let $t_0$ be the first round where the resampling is different. Without loss
of generality, we can take $A$ to be the event occurring in $S_{t_0}$ for table $M$ but not table $M'$. For the two runs to be different, we need there to be some nonempty set of variables $Y \subset \text{var}(A)$ with corresponding values $(X_{i,j,t+1})$. Since the resampling does not change before round $t_0$, in $M'$, $Y$ is assigned to $(X'_{i,j,t+1})$ at time $t_0$. Then we have one of two possible cases:

- $Y = \text{var}(A)$: since $A$ holds in the run generated by $M'$, then $A \in N^+(S_t)$. It follows that an event occurs in the run generated by $M$ such that it intersects $A$. But then the algorithm would need access to columns beyond the final round of the table in order to replace the variables which the two events share. So this is a contradiction.

- $Y \neq \text{var}(A)$: then there is a nonempty set $Z = \text{var}(A) \setminus Y$. Any variable in this set is not attained in the final round, so it must be resampled in the $M$ run. Let us take $X_j$ to be the first variable to be resampled at or after round $t_0$. Since $A$ cannot occur, there must be a distinct event $A' \neq A$ which causes $X_j$ to be resampled by the algorithm. Then $\text{var}(A) \cap \text{var}(A') \neq \emptyset$, so we can take a variable $X_k$, where $k \in \text{var}(A) \cap \text{var}(A')$, which due to $A'$ occurring, is resampled at or after round $t_0$ in the $M$ run. Then for any such $k$, $X_k \in Z$. Since $A'$ is by construction the first resampling event involving $Z$ at or after stage $t_0$, we know that $X_k$ has not been resampled until $A'$ occurs. This shows that we can find a assignment to variables contained in the intersection $\text{var}(A) \cap \text{var}(A')$ allowing both $A$ and $A'$ to happen, which can be then extended to a full assignment. This is a contradiction to condition 6.2.1, since $A$ and $A'$ share the random variable $X_j$ and their intersection is nonempty.

Since both possible scenarios lead to contradictions, we conclude that the claim is true, that is that the algorithms running on $M'$ and $M$ generate the same log $S$.

This implies that every possible table conditioned on the log $S$ such that $\sigma_{t+1}$ is a valid assignment has one-to-one correspondence to another table where $\sigma_{t+1}$ is another valid assignment. So for any valid assignments, say $\sigma, \sigma'$, there is a bijection between the resampling tables that induce them. Moreover, we have that the ratio between the probability of the two tables is equivalent to that of the probabilities of $\sigma$ and $\sigma'$ under the product distribution of all random variables $\mu(\cdot)$. This shows

43
in turn that any two valid assignments are proportional to their own probability of occurring in $\mu(\cdot)$, and therefore $\sigma_{t+1}$ has the desired distribution. \hfill \Box

**Theorem 6.3.1.** When condition 6.2.1 holds and the algorithm terminates, the output is

$$\mu\left( \cdot \ \big| \ \bigcap_{i \in [m]} \tilde{A}_i \right), \tag{6.21}$$

the product distribution conditioned on avoiding all bad events.

**Proof.** Let $S = S_1, S_2, ..., S_t$ be the log of a successful run. Then $S_t = \emptyset$. By the previous Lemma 6.3.1, conditioned on $S$, the resulting assignment is

$$\mu\left( \cdot \ \big| \ \bigcap_{i \in [m] \setminus N^+(S_t)} \tilde{A}_i \right) = \mu\left( \cdot \ \big| \ \bigcap_{i \in [m]} \tilde{A}_i \right). \tag{6.22}$$

This holds for any possible log, so the result follows. \hfill \Box

### 6.4 Time complexity

Let $p_i :=Pr(A_i)$ for all $i \in [1, m]$, and let $\mathcal{I}$ denote the set of independent sets of the graph $G$. We define the *weighted independent polynomial* $q_I$ by

$$q_I(p) := \sum_{J \in \mathcal{I}, I \subseteq J} (-1)^{|J|-|I|} \prod_{i \in J} p_i, \tag{6.23}$$

for $p = (p_1, p_2, ..., p_m)$.

In order to simplify the notation, let $A(S) = \bigcap_{i \in S} A_i$ be the conjunction of all events indexed by $S$. Let $Pr_\mu$ denote the probability space with respect to the product distribution $\mu$. For set $I$ in the dependency graph, we can write the probability that all events indexed by $I$ happen as

$$p_I = Pr_\mu(A(I)) = \begin{cases} \prod_{i \in I} p_i, & \text{if } I \text{ is independent} \\ 0, & \text{otherwise}, \end{cases} \tag{6.24}$$

where the first case follows from the fact that any two sets in an independent set are independent, and the second is due to Condition 6.2.1. Note that this probability does not exclude the scenarios where other events aside from those in $I$ also happen.
Let us compute the probability that only the events in $I$ happen. The *inclusion-exclusion principle* states that for a finite set of events $A_1, A_2, \ldots, A_m$, their union can be calculated using the formula

$$\left| \bigcup_{i \in [m]} A_i \right| = \sum_{J \subseteq [m], J \neq \emptyset} (-1)^{|J|+1} \left| \bigcap_{j \in J} A_j \right|. \quad (6.25)$$

This comes from the fact that when the intersection are non-trivial between the events, the repeated inclusion of the elements lying in those intersections will need to be compensated.

Denote the whole space of events by $\Omega$. Then using (6.25) and the De Morgan’s laws, we get

$$\left| \bigcap_{i \in [m]} A_i \right| = \left| \Omega - \sum_{J \subseteq [m], J \neq \emptyset} (-1)^{|J|+1} \left| \bigcap_{j \in J} A_j \right| \right|. \quad (6.26)$$

By applying (6.26) to both the intersection of events and the intersection of their negations, we have

$$Pr_{\mu} \left( \bigcap_{i \in I} A_i \cap \bigcap_{i \not\in I} \bar{A}_i \right) = \sum_{J \superset I} (-1)^{|J|} p_J = \sum_{J \in \mathcal{I}, I \subset J} (-1)^{|J|-|I|} \prod_{i \in J} p_i, \quad (6.27)$$

since the cases where $J$ is not independent are 0 and do not contribute to the summation. Note that this is exactly the definition of $q_I$, so

$$Pr_{\mu} \left( \bigcap_{i \in I} A_i \cap \bigcap_{i \not\in I} \bar{A}_i \right) = q_I. \quad (6.28)$$

Since for any two distinct sets $I$, the events $\left( \bigcap_{i \in I} A_i \cap \bigcap_{i \not\in I} \bar{A}_i \right)$ have trivial intersection, we have that

$$\sum_{I \in \mathcal{I}} q_I = 1. \quad (6.29)$$

Additionally, $A(I)$ is the union of the events $\left( \bigcap_{i \in I} A_i \cap \bigcap_{i \not\in I} \bar{A}_i \right)$ over all sets $J \superset I$, which implies

$$p_I = Pr_{\mu}(A(I)) = \sum_{J \in \mathcal{I}, J \superset I} q_J. \quad (6.30)$$

**Definition 6.4.1.** For an independent set $I \in \mathcal{I}$, we define

- the *boundary* $\delta(I) := \mathcal{N}^+(I) \setminus I$, the set of events that depend on $I$ but are not in $I$;
• the exterior $I^e := [m] \setminus N^+(I)$, the set of events which are independent of all events in $I$;
• the complement $I^c := [m] \setminus I$, the set of events which are not in $I$.

Then the complement of $I$ can also be written in as the union of the other two, i.e. $I^c = \delta(I) \cup I^e$.

**Lemma 6.4.1.** If Condition 6.2.1 holds, and $\mathcal{S} = (S_1, S_2, \ldots, S_l)$ is an independent set sequence, then

$$Pr(\log \text{ is } \mathcal{S} \text{ up to round } l) = q_{S_l} \prod_{i=1}^{l-1} p_{S_i}$$  \hspace{1cm} (6.31)

in the algorithm.

**Proof.** We can assume without loss of generality that $q_{S_l} > 0$, for if it was equivalent to 0, then the sequence would not occur.

Let $A(S)$ be as defined above, and $B(S) = \bigcap_{i \in \mathcal{S}} \bar{A}_i$. By Definition 6.4.1 and by De Morgan’s laws, we have that

$$B(I^c) = B(\delta(I)) \cap B(I^e).$$  \hspace{1cm} (6.32)

Then we can rephrase the probability $q_I$ that exactly the events in $I$ and no other occur using $B(\cdot)$:

$$q_I = Pr_\mu(A(I) \cap B(I^e)).$$  \hspace{1cm} (6.33)

Again by Definition 6.4.1, we have

$$Pr_\mu(B(I^e) \mid A(I)) = Pr_\mu(B(I^e)).$$  \hspace{1cm} (6.34)

Moreover, by Condition 6.2.1,

$$A(I) \cap B(\delta(I)) = A(I).$$  \hspace{1cm} (6.35)

This implies then that for every set $I \in \mathcal{I}$,

$$q_I = Pr_\mu(A(I) \cap B(I^e)) = Pr_\mu(A(I) \cap B(\delta(I)) \cap B(I^e)) =$$

$$= Pr_\mu(A(I) \cap B(I^e)) = Pr_\mu(A(I)) Pr_\mu(B(I^e)),$$  \hspace{1cm} (6.36)

where we used (6.35) for the penultimate equality, and (6.34) for the last.

46
We show the result by induction on the rounds $l$. For the base case, $l = 1$ and the lemma holds trivially. Now suppose that $l > 1$. By Definition 6.3.1 of independent set sequence, we know that $S_l \subset N^+(S_{l-1})$, and

$$B(S^c_l) \cap B(S^c_{l-1}) = B(S^c_l),$$

(6.37)

because we do not resample any of the random variables associated with the events $A_i$ when $i \in S_{l-1}$.

Using Lemma 6.3.1, we have that, conditioned on $S_{l-1}$, the distribution of the random sample $\sigma_l$ at round $l$ is the product distribution conditioned on none of the events outside of $N^+(S_{l-1})$ happening.

Let $Pr_{PRS}$ denote the probability space with respect to the partial rejection sampling algorithm. Then the probability of having $S_l$ at round $l$ is

$$Pr_{PRS}(A(S_l) \cap B(S^c_l)) \text{ holds at } l \mid \text{ prior log is } S_1, S_2, \ldots, S_{l-1})$$

$$= Pr_{\mu}(A(S_l) \cap B(S^c_l) \mid B(S^c_{l-1}))$$

$$= \frac{Pr_{\mu}(A(S_l) \cap B(S^c_l) \cap B(S^c_{l-1}))}{Pr_{\mu}(B(S^c_{l-1}))}$$

$$= \frac{Pr_{\mu}(A(S_l) \cap B(S^c_l))}{Pr_{\mu}(B(S^c_{l-1}))}$$

(6.38)

$$= q_{S_l} s_l \frac{q_{S_l}}{Pr_{\mu}(B(S^c_{l-1}))},$$

where the penultimate equation holds by (6.37) and the last by (6.33).

Now we can apply the inductive hypothesis and see that

$$Pr_{PRS}(\text{log is } S \text{ up to } l) = \frac{q_{S_l}}{Pr_{\mu}(B(S^c_{l-1}))} \cdot q_{S_{l-1}} \prod_{t=1}^{l-2} p_{S_t} = q_{S_l} \prod_{t=1}^{l-2} p_{S_t} = q_{S_l} \prod_{t=1}^{l-1} p_{S_t},$$

(6.39)

where the last equality is due to (6.36).

**Corollary 6.4.1.** Let $S = S_1, S_2, \ldots, S_l$ be an independent set sequence and $I$ be an independent set of the dependency graph. If Condition 6.2.1 holds and $q_0 > 0$, then

$$\sum_{S:S_1=I} p_{S} q_0 = q_I.$$

(6.40)
Proof. By Lemma 6.3.1 it follows that the distribution of $\sigma_l$ conditioned on $S$ at round $l$ is $\mu(\cdot \mid B(S_{l-1}))$. Then the probability of getting the assignment that we want is
\[
\mu(B([m]) \mid B(S_{l-1})) = \frac{\mu(B([m]))}{\mu(B(S_{l-1}))} \geq \mu(B[m]) = q_\emptyset. \tag{6.41}
\]

This goes to show that the probability of the algorithm terminating at round $t$ is bounded above by $(1 - q_\emptyset)^t$, which tends to 0, as $t$ tends to infinity. This implies that whenever $q_\emptyset > 0$ the algorithm terminates with probability 1.

Let $S$ be an independent set sequence with final independent set $S_l = \emptyset$. Then, by Lemma 6.4.1,
\[
p_S q_\emptyset = Pr(\log is S up to round $l$), \tag{6.42}
\]
and $\sum_{S: S_l = \emptyset} p_S q_\emptyset$ is the sum of probabilities of all halting logs that have their first independent set equal to $I$. This is just the probability of having exactly $I$ as the first independent set, which is $q_I$ by definition.

Lemma 6.4.2. If Condition 6.2.1 holds and $q_\emptyset > 0$, then for all $i \in [m]$ and for all $z \in [0, 1]$,
\[
q_\emptyset(p_1, p_2, ..., p_iz, ..., p_m) > 0. \tag{6.43}
\]

Proof. Since, by hypothesis $q_\emptyset > 0$, we want to show that
\[
q_\emptyset(p_1, p_2, ..., p_iz, ..., p_m) > q_\emptyset. \tag{6.44}
\]

Recall that by definition we have
\[
q_\emptyset(q_\emptyset(p)) = \sum_{I \in \mathcal{I}} (-1)^{|I|} \prod_{i \in I} p_i, \tag{6.45}
\]
so, splitting the sum into sets that do and do not contain $i$, we get
\[
q_\emptyset(p_1, p_2, ..., p_iz, ..., p_m) = \sum_{i \in \mathcal{I}, i \notin I} (-1)^{|I|} \prod_{j \in I} p_j + z \sum_{i \in \mathcal{I}, i \in I} (-1)^{|I|} \prod_{j \in I} p_j. \tag{6.46}
\]
Since $q_i(p)$, the probability of event $A_i$ and no other occurring, is given by
\[
q_i(p) = \sum_{I \in \mathcal{I}, i \notin I} (-1)^{|I|-1} \prod_{j \in I} p_j, \tag{6.47}
\]
it follows that the second summation term (6.46) is
\[
z \sum_{I \in \mathcal{I}, i \in I} (-1)^{|I|} \prod_{j \in I} p_j = z(-q_i(p)) = -zq_i(p) \geq 0, \tag{6.48}
\]

48
since \( z \) is non-negative, and \( q_i(p) \) is positive. Then
\[
q_\emptyset(p_1, p_2, \ldots, p iz, \ldots, p_m) \geq q_\emptyset > 0,
\]
by hypothesis.

\[\textbf{Theorem 6.4.1.}\] Let \( R_i \) be the number of resamplings of event \( A_i \) and \( R = \sum_{i \in [m]} R_i \) be the total number of resamplings for all the bad events. If Condition 6.2.1 holds, and \( q_\emptyset > 0 \), then the expected number of resamplings is \( \mathbb{E}(R) = \sum_{i \in [m]} \frac{q_i}{q_\emptyset} \).

\textit{Proof.} We first show that \( \mathbb{E}(R_i) = q_\emptyset \left( \frac{1}{q_\emptyset(p_1, p_2, \ldots, p iz, \ldots, p_m)} \right)' \bigg|_{z=1}. \)

Since \( p_S q_\emptyset \) gives a probability distribution, we should have that the sum of all such probabilities adds up to 1. Then, using this observation and Corollary 6.4.1,
\[
\sum_S p_S q_\emptyset = \sum_{i \in I} \sum_{S : S_1 = I} p_S q_\emptyset = \sum_{i \in I} q_i = 1. \quad (6.50)
\]

We can rearrange the equations to get that
\[
\sum_S p_S = \frac{1}{q_\emptyset}. \quad (6.51)
\]

Let \( R_i(S) \) be the total number of resamplings of \( A_i \) in \( S \), i.e. the number of times event \( A_i \) occurs in the independent set sequence \( S \). By Lemma 6.4.2 and 6.50, we have
\[
\sum_S p_S z^{R_i(S)} = \frac{1}{q_\emptyset(p_1, p_2, \ldots, p iz, \ldots, p_m)}.
\]

The derivative with respect to \( z \) gives
\[
\sum_S R_i(S) p_S z^{R_i(S)-1} = \left( \frac{1}{q_\emptyset(p_1, p_2, \ldots, p iz, \ldots, p_m)} \right)', \quad (6.53)
\]
which evaluated at \( z = 1 \) is
\[
\sum_S R_i(S) p_S = \left( \frac{1}{q_\emptyset(p_1, p_2, \ldots, p iz, \ldots, p_m)} \right)' \bigg|_{z=1} \quad (6.54)
\]
Since \( \mathbb{E}(R_i) = \sum_S Pr_{P_S}(\log is S) R_i(S) \) by definition,
\[
\mathbb{E}(R_i) = \sum_S p_S q_\emptyset R_i(S) = q_\emptyset \left( \frac{1}{q_\emptyset(p_1, p_2, \ldots, p iz, \ldots, p_m)} \right)' \bigg|_{z=1} \quad (6.55)
\]

Now we claim that \( q_\emptyset \left( \frac{1}{q_\emptyset(p_1, p_2, \ldots, p iz, \ldots, p_m)} \right)' \bigg|_{z=1} = \frac{q_i}{q_\emptyset}. \)
Take the derivative of (6.46) with respect to \( z \). Then

\[
q_i'(p_1, p_2, ..., p_i z, ..., p_m) = \sum_{I \in I, i \in I} (-1)^{|I|} \prod_{j \in I} p_j = -q_i. \tag{6.56}
\]

Then, by the derivative rule for inverse functions,

\[
\left( \frac{1}{q_0(p_1, p_2, ..., p_i z, ..., p_m)} \right)' = \frac{q_i'(p_1, p_2, ..., p_i z, ..., p_m)}{q_0(p_1, p_2, ..., p_i z, ..., p_m)^2} = \frac{q_i}{q_0(p_1, p_2, ..., p_i z, ..., p_m)^2}. \tag{6.57}
\]

Setting \( z = 1 \), the claim holds. Since \( \mathbb{E}(R) = \sum_{i \in [m]} \mathbb{E}(R_i) \), by linearity of expectation the theorem follows. \( \square \)

The results discussed thus far in this section apply to a generic scenario of partial rejection sampling under extremal instances. Let us narrow things down to the case of sampling rooted spanning trees of a graph \( G \). The expected number \( \mathbb{E}(R_i) \) of resamplings of event \( A_i \) is now the expected number of times that a cycle \( C_i \) arises in the dependency graph, so the number of times we expect to have to pop cycle \( C_i \). Denote by \( Z_0 \) the number of assignments of arrows to the vertices of \( G \) which result in a directed tree with root \( r \), and by \( Z_1 \) those that yield a subgraph with one cycle, or a unicyclic subgraph. Then the

\[
\mathbb{E}(R) = \frac{Z_1}{Z_0}. \tag{6.58}
\]

The following theorem gives a bound on the ratio between these two quantities.

**Theorem 6.4.2.** Let \( G = (V, E) \) be a graph with \( |V| = n \), \( |E| = m \). Then \( \frac{Z_1}{Z_0} \leq mn \).

**Proof.** Consider an assignment that yields a unicyclic subgraph. Then this assignment can be partitioned into two components, one of which is a directed tree with root \( r \) and the other is a directed cycle. By removing the second component we would get a graph with edge set size one smaller than the vertex set, i.e. a spanning tree. Therefore, in the unicyclic component we have some subtrees rooted on the cycle.

By connectivity of \( G \), any pair of vertices in \( G \) has a path which connects it. In particular, there exists an edge in \( G \) between the two components, say \( \{v_0, v_1\} \), with \( v_0 \) in the tree component and \( v_1 \) in the unicyclic component. We can extend
this edge to a path \(v_0, v_1, ..., v_l\) which follows the arrows until we reach a vertex \(v_l\) that lies on the cycle. Then we have arrows \(v_0 \rightarrow v_1, v_1 \rightarrow v_2, ..., v_{l-1} \rightarrow v_l\). Now let \(v_l \rightarrow v_{l+1}\) and reassign the arrows by \(v_l \rightarrow v_{l-1}, v_{l-1} \rightarrow v_{l-2}, ..., v_1 \rightarrow v_0\). The resulting subgraph is a directed tree rooted at \(r\).

Now that we have seen how we can obtain a directed rooted tree from a unicyclic graph, we want to look at how many unicyclic subgraphs can be associated with a given directed tree. The edge \(\{v_l, v_{l+1}\}\) in the procedure is undirected. However, \(v_l\) is the closer vertex to \(r\), so there is no ambiguity. The other vertices \(v_{l-1}, v_l, ..., v_0\) can be easily recovered if we have edge \(v_l, v_{l+1}\), since the path from \(v_l\) to \(v_0\) is unique. Then the unicyclic subgraph can be recovered by just reassigning the arrows to the vertices in this way: \(v_1 \rightarrow v_2, v_2 \rightarrow v_3, ..., v_l \rightarrow v_{l+1}\). It follows that in order to reverse the procedure, all we need is to know one edge, \(v_l, v_{l+1}\), and one vertex, \(v_0\). Since we have \(m\) edges and \(n\) vertices in \(G\), for a given directed tree rooted at \(r\) we have at most \(mn\) unicyclic graphs associated with it.

Then by Theorem 6.4.2 and (6.58), the expected number of popped cycles in the partial rejection sampling algorithm is

\[
\mathbb{E}(R) \leq mn.
\]  

(6.59)

Then the time complexity of the algorithm is \(O(mn) = O(n^3)\), as expected since it is equivalent to Wilson’s algorithm.
Chapter 7

Conclusion

Throughout this thesis we have seen how the task of sampling spanning trees of a graph uniformly at random can be tackled using different approaches and exploiting different results, both of algebraic and probabilistic nature.

Here is a table of comparison for the time complexity of the algorithms which we have discussed, along with the corresponding references.

<table>
<thead>
<tr>
<th>Name of algorithm</th>
<th>Main technique</th>
<th>Time complexity</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact sampling (1847)</td>
<td>Reduction to exact counting</td>
<td>$O(n^3 \cdot m) = O(n^5)$</td>
<td>[16] [20]</td>
</tr>
<tr>
<td>Partial rejection sampling (2017)</td>
<td>Lovász local lemma and extremal cases</td>
<td>$O(\zeta)$</td>
<td>[9]</td>
</tr>
</tbody>
</table>

Table 7.1: Algorithms for sampling uniform spanning trees of a graph
Starting with a perhaps more obvious approach, we first introduced a classical method in the sampling of uniform spanning trees that follows intuitively from Kirchhoff’s Matrix Tree Theorem. Later on, we analysed some more efficient procedures, building on well known results and concepts which allow us to optimise the running time and which apply to more generic contexts as well.

As mentioned earlier, spanning trees can be extremely helpful in various fields. A particularly interesting application in the direction of which this research could be extended is that of the use of spanning trees in order to expand a graph. Expanding a graph can be a crucial aspect in the subject of network design: by doing so, we create enough alternative and disjoint paths in order to ensure that a network will most likely recover from random failures. Indeed, the path diversity of the graph will reduce the probability of a congestion happening. In [6], it is shown that the union of two uniformly distributed spanning trees of a graph approximates an expansion of the graph to within a factor of $O(\log n)$.

Another reason why we might want to deepen our understanding of spanning trees of a graph is for graph sparsification purposes. We say a graph is dense when the number of edges is close to the maximum possible number of edges it can contain. If the graph is not dense, we say it is sparse. Then a sparsifier $G'$ of a graph $G$ is a sparse subgraph which retains some properties of the original graph. We can use this idea to approximate a dense graph $G$ with a sparsifier, which is easier to work with and encloses the characteristics of $G$ that we care about. This substitution enables us to decrease the time complexity of the algorithms that we may want to implement, which often depends on the number of edges. Since spanning trees contain the minimum possible number of edges, they are a great example of a sparsifier of a graph. Instead of sampling the edges of a graph independently, we can pick a uniformly random spanning tree, which preserves the connectivity of the original graph [7].
Bibliography


