Efficient Sampling of Dependency Structures

Ran Zmigrod Tim Vieira Ryan Cotterell
University of Cambridge Johns Hopkins University ETH Zürich
rz279@cam.ac.uk tim.f.vieira@gmail.com ryan.cotterell@inf.ethz.ch

Abstract

Probabilistic distributions over spanning trees in directed graphs are a fundamental model of dependency structure in natural language processing, syntactic dependency trees. In NLP, dependency trees often have an additional root constraint: only one edge may emanate from the root. However, no sampling algorithm has been presented in the literature to account for this additional constraint. In this paper, we adapt two spanning tree sampling algorithms to faithfully sample dependency trees from a graph subject to the root constraint. Wilson (1996)’s sampling algorithm has a running time of $O(H)$ where $H$ is the mean hitting time of the graph. Colbourn et al. (1996)’s sampling algorithm has a running time of $O(N^3)$, which is often greater than the mean hitting time of a directed graph. Additionally, we build upon Colbourn’s algorithm and present a novel extension that can sample $K$ trees without replacement in $O(KN^3 + K^2N)$ time. To the best of our knowledge, no algorithm has been given for sampling spanning trees without replacement from a directed graph.\(^1\)

1 Introduction

Spanning trees in directed graphs\(^2\) are fundamental combinatorial structures in natural language processing where they are used to represent dependency structures—especially syntactic dependency structure (Kübler et al., 2009). Additionally, probabilistic models over spanning trees are common in the NLP literature with applications primarily in non-projective dependency parsing (Pei et al., 2015; Wang and Chang, 2016; Dozat and Manning, 2017; Ma and Hovy, 2017), but also in recovering phylogenetic structures (Andrews et al., 2012), and event extraction (McClosky et al., 2011).

Given the prevalence of such probabilistic models, efficient dependency tree sampling algorithms deserve study. Indeed, some work has been done in transition-based dependency parsing (Keith et al., 2018) as well as graph-based dependency parsing (Nakagawa, 2007; Mareček and Žabokrtský, 2011). Sampling has also been utilized in an abundance of NLP tasks, such as text generation (Clark et al., 2018; Fedus et al., 2018), co-reference resolution (Singh et al., 2012), and language modeling (Mnih and Hinton, 2007; Logan IV et al., 2020).

The theoretical computer science literature has several efficient algorithms for sampling directed spanning trees. These algorithms come in two flavors. First, random walks through Markov chains have been used to sample spanning trees from both undirected (Broder, 1989; Aldous, 1990) and directed graphs (Wilson, 1996). The algorithm of Wilson (1996) is linear in the mean hitting time of the graph and is currently the fastest sampling algorithm for directed spanning trees. It has been used in dependency parsing inference by Zhang et al. (2014a,b). Second, several algorithms have leveraged the matrix–tree theorem (MTT; Kirchhoff, 1847; Tutte, 1984). The MTT has been frequently used to perform inference on non-projective graph-based dependency parsers (Koo et al., 2007; McDonald and Satta, 2007; Smith and Smith, 2007; Zmigrod et al., 2021). This theorem was first used for sampling by Guénoche (1983) who gave an $O(N^5)$ algorithm which was then improved by Kulkarni (1990) and Colbourn et al. (1996). Colbourn et al. (1996) give an $O(N^3)$ algorithm to sample spanning trees from an unweighted directed graph. We generalize their algorithm to the weighted case.

While directed spanning tree sampling algorithms exist, an important constraint of many dependency tree schemes, such as the Universal De-
We consider distributions over spanning trees in
Wilson (1996), that a
dependency tree may only have one edge emanating from the designated root symbol. Algorithms exist for enforcing this constraint in decoding
(Gabow and Tarjan, 1984; Zmigrod et al., 2020; Stanojević and Cohen, 2021) and inference (Koo et al., 2007; Zmigrod et al., 2021). However, to the best of our knowledge, no sampling algorithm exists which enforces the root constraint.

In this paper, we adapt the algorithms of Wilson (1996) and Colbourn et al. (1996) to efficiently sample directed spanning trees subject to a root constraint while maintaining the runtime of the original algorithms. Additionally, we provide a further extension to Colbourn et al. (1996)'s algorithm that allows us to sample trees without replacement. Sampling without replacement (SWOR) algorithms are useful when distributions are skewed, which is often the case in a trained system. To the best of our knowledge, no SWOR algorithm has been presented in the literature for directed spanning trees, though Shi et al. (2020) provides a general framework that enables a SWOR algorithm to be adapted for particular kinds of sampling algorithms.

2 Distributions over Trees

We consider distributions over spanning trees in rooted directed weighted graphs (graphs for short). A graph is denoted by $G = (\rho, N, \mathcal{E})$ where $N$ is a set of $N + 1$ nodes including a designated root node $\rho$ and $\mathcal{E}$ is a set of ordered pairs between two nodes $(i \rightarrow j)$. For each edge $(i \rightarrow j) \in \mathcal{E}$, we associate a non-negative weight $w_{i \rightarrow j} \in \mathbb{R}_{\geq 0}$. Note that $w_{i \rightarrow j} = 0$ when $(i \rightarrow j) \notin \mathcal{E}$.

A directed spanning tree, denoted by $t$, is a collection of edges in a graph $G$ such that each node $j \in N \setminus \{\rho\}$ has exactly one incoming edge and $t$ contains no cycles. Moreover, we specifically examine distributions over dependency trees, which are spanning trees with an additional constraint that exactly one edge must emanate from the root $\rho$. This additional constraint is common amongst most dependency tree annotation schemes, e.g., Universal Dependencies. When the type of tree (spanning or dependency) is clear from context, we will simply use trees. The set of all spanning trees of a graph $G$ is given by $T(G)$ and the set of all dependency trees of a graph $G$ is given by $D(G)$. Note that $D(G) \subseteq T(G)$. When the graph $G$ is clear from context, we will refer to these sets $T$ and $D$ respectively.

The weight of a tree $t$ is the product of its edge weights:

$$w(t) \overset{\text{def}}{=} \prod_{(i \rightarrow j) \in t} w_{i \rightarrow j}$$

The probability of a spanning tree is then given by

$$p(t) \overset{\text{def}}{=} \frac{w(t)}{Z} \quad \text{where} \quad Z \overset{\text{def}}{=} \sum_{t \in T} w(t)$$

The normalizing constant in the case of dependency trees sums over $D$ instead of $T$.

3 Random Walk Sampling

In this section, we present the spanning tree sampling algorithm of Wilson (1996) and adapt it to sample dependency trees. The algorithm is based on a random walk through the nodes of the graph until a tree is formed. In order to do this, we require a graph $G$ to be stochastic. A stochastic graph is such that the weights of all incoming edges to a node sum to one. That is, for all non-root nodes $j \in N \setminus \{\rho\}$ we have that

$$\sum_{i \in N} w_{i \rightarrow j} = 1$$

Wilson (1996) shows that any graph may be converted into a stochastic graph, by adjusting the edge weights to be locally normalized along all incoming edges. Therefore, the edge $(i \rightarrow j)$ has a weight $w'_{i \rightarrow j}$ defined by

$$w'_{i \rightarrow j} \overset{\text{def}}{=} \frac{w_{i \rightarrow j}}{\sum_{\mathcal{N}' \in \mathcal{N}} w_{i \rightarrow j'}}$$

We can also define this new weight over a tree $w'(t) \overset{\text{def}}{=} \prod_{(i \rightarrow j) \in t} w'_{i \rightarrow j}$.

Lemma 1. For any tree $t \in T$,

$$w'(t) \propto w(t)$$

Note that Colbourn et al. (1996) presents two algorithms for sampling directed spanning trees. In this work, we focus on their first algorithm, which runs in $O(N^3)$. While the second algorithm is based on a reduction to fast matrix multiplication, which is typically impractical. We did not extend this algorithm because it is not amendable to sampling without replacement and it is generally slower than the algorithm of Wilson (1996).
Figure 1: Cycle erasure in a random walk of a graph. The associated graph of the above walk has six nodes (including the root node $\rho$) and we start the walk at $\rho$ with the tree only containing the root node. The random walk includes a cycle with nodes $2$, $3$, and $4$, and this cycle is erased when we create the path from $\rho$ back up to $\rho$. Note that the arrows here mark the path upwards rather than the edges in the tree, the tree edges are reversed (e.g., $\odot \to \odot$).

Proof.

\[
\begin{align*}
\frac{w(t)}{w'_{ij}(t)} &= \prod_{(i \to j) \in t} \frac{w'_{ij}}{w_{ij}} \\
&= \prod_{(i \to j) \in t} \frac{w_{ij}}{\sum_{i' \in N} w_{i'j}} \\
&= \left( \prod_{(i \to j) \in t} \frac{1}{\sum_{i' \in N} w_{i'j}} \right) \left( \prod_{(i \to j) \in t} w_{ij} \right) \\
&= \left( \prod_{(i \to j) \in t} \frac{1}{\sum_{i' \in N} w_{i'j}} \right) w(t)
\end{align*}
\]

$\propto w(t)$

Note that the left-hand term in (6c) is a constant since every non-root node has in incoming edge in the tree, and so the constant is equal to $\prod_{i \in N \setminus \{\rho\}} \left( \sum_{i' \in N} w_{i'j} \right)^{-1}$.

A stochastic graph then defines a Markov chain that we can perform a random walk on. We begin with a tree that is just composed of the root $\rho$. Then while there exists a node connect to the tree, we start a random walk where we add each edge used to the tree, until we encounter a node that is connected to the tree. When we have connected to the tree, we can proceed to start a new random walk from a node not in the tree.

Of course, during a random walk, we may go through a cycle. Whenever we walk through a cycle, we simply forget the cycle as part of the walk. That is, if we find ourselves visiting node $i$ in our walk for a second time, we erase the cyclic path formed at $i$ and continue our walk. This can be seen visually in Fig. 1. This type of walk is known as a loop-erased random walk (Lawler, 1979). Given

\[
\begin{align*}
\text{def} \ wilson(G): \\
\quad \triangleright \text{Sample a spanning tree from a graph } G; \text{ requires } O(H) \text{ time, } O(N^2) \text{ space.} \\
\quad t \leftarrow 0 \\
\quad \text{visited} \leftarrow \{\rho\} \\
\quad \triangleright \text{The following for and while loops take } O(H) \text{ to execute. More specifically, line 9 is called} \\
\quad \text{ } \text{ } O(H) \text{ times.} \\
\quad \text{for } i \in N \setminus \{\rho\}: \\
\quad \quad u \leftarrow i \\
\quad \quad \text{while } u \notin \text{visited}: \\
\quad \quad \quad \text{Sample } v \in N \text{ with weight } w'_{uv} \text{.} \\
\quad \quad \quad t_u \leftarrow v \\
\quad \quad \quad u \leftarrow v \\
\quad \quad u \leftarrow i \\
\quad \quad \text{while } u \notin \text{visited}: \\
\quad \quad \quad \text{visited.add}(u) \\
\quad \quad \quad u \leftarrow v \text{ such that } t_u = v \\
\quad \text{return } t
\end{align*}
\]

Figure 2: Wilson (1996)’s algorithm to sample spanning trees from a graph.

we sample a sequence of edges $S$ in a random walk, we can split the We can split the edges into those corresponding to the loop-erased random walk, $\tilde{S}$, and those that do not, $\bar{S}$.

Lemma 2. For any sequence of edges $S$ sampled from a random walk, we have

\[
p(S) = p(\tilde{S}) p(\bar{S})
\]

Proof.

\[
p(S) = \prod_{(i \to j) \in S} w'_{ij} \quad \text{(8a)}
\]

\[
= \left( \prod_{(i \to j) \in \tilde{S}} w'_{ij} \right) \left( \prod_{(i \to j) \in \bar{S}} w'_{ij} \right)
\]
Furthermore, Wilson will contain a directed spanning tree. Let $S_{\pi}$ where Theorem 1.

Proof. It is clear that if $\text{wilson}(G)$ terminates, $t$ will contain a directed spanning tree. Let $S$ be the set of edges sampled in line 9 of $\text{wilson}$. As we sample $t$ using several independent random walks, by Lemma 2, we sample the edges in $t$ independently of the edges in $S \setminus t$. In particular, we sample $t$ with probability

$$p(t) \propto \prod_{(i \rightarrow j) \in t} w_{i \rightarrow j}$$

By Lemma 1, $t$ is a sample from our desired distribution over trees.

Wilson (1996, Theorem 3) proves that line 9 of $\text{wilson}$ executes $O(H)$ times before the program terminates with tree $t$.

3.1 Root Constraint Modification

Wilson (1996)’s algorithm does not ensure that only one edge emanates from the root. However, a simple modification to the algorithm allows us to do this. The original algorithm samples a spanning tree rooted at $\rho$ for a graph $G$. Suppose we know we want the edge $(\rho \rightarrow j)$ to be the single edge emanating from the root. Then we can run Wilson’s algorithm on the graph $G'$, which is $G$ with node $\rho$ removed and node $j$ defined as the new root. By adding $(\rho \rightarrow j)$ to the newly sampled tree, we clearly have a dependency tree in $G$. We can sample the root edge $(\rho \rightarrow j)$ from the root weights $\rho$ to sample an unbiased dependency tree from the distribution. The pseudocode for this algorithm is given as $\text{wilson}_rc$ in Fig. 3

**Theorem 2.** For any graph $G$, $\text{wilson}_rc(G)$ samples a dependency tree $t \in \mathcal{T}$ with probability

$$p(t) \propto \prod_{(i \rightarrow j) \in t} w_{i \rightarrow j}$$

Furthermore, $\text{wilson}_rc$ runs in $O(H)$ in.

Proof. We note that executing lines 6 to line 18 is equivalent to running $\text{wilson}$ on the graph $G'$ that is rooted at $j$ (and does not have $\rho$). By Theorem 1, this results in a tree $t' \in \mathcal{T}(G')$ and was sampled with probability proportional to $\prod_{(i \rightarrow j) \in t'} w_{i \rightarrow j}$. We sample the edge emanating from the root $(\rho \rightarrow j)$ with probability proportional to $w_{\rho \rightarrow j}$ and so $t = t' \cup \{\rho \rightarrow j\}$ is a dependency tree rooted at $\rho$ for a graph $G$. Suppose we know we want the edge $(\rho \rightarrow j)$ to be the single edge emanating from the root. Then we can run Wilson’s algorithm on the graph $G'$, which is $G$ with node $\rho$ removed and node $j$ defined as the new root. By adding $(\rho \rightarrow j)$ to the newly sampled tree, we clearly have a dependency tree in $G$. We can sample the root edge $(\rho \rightarrow j)$ from the root weights $\rho$ to sample an unbiased dependency tree from the distribution. The pseudocode for this algorithm is given as $\text{wilson}_rc$ in Fig. 3.

**Theorem 2.** For any graph $G$, $\text{wilson}_rc(G)$ samples a dependency tree $t \in \mathcal{T}$ with probability

$$p(t) \propto \prod_{(i \rightarrow j) \in t} w_{i \rightarrow j}$$

Furthermore, $\text{wilson}_rc$ runs in $O(H)$ in.

Proof. We note that executing lines 6 to line 18 is equivalent to running $\text{wilson}$ on the graph $G'$ that is rooted at $j$ (and does not have $\rho$). By Theorem 1, this results in a tree $t' \in \mathcal{T}(G')$ and was sampled with probability proportional to $\prod_{(i \rightarrow j) \in t'} w_{i \rightarrow j}$. We sample the edge emanating from the root $(\rho \rightarrow j)$ with probability proportional to $w_{\rho \rightarrow j}$ and so $t = t' \cup \{\rho \rightarrow j\}$ is a dependency tree rooted at $\rho$ for a graph $G$. Suppose we know we want the edge $(\rho \rightarrow j)$ to be the single edge emanating from the root. Then we can run Wilson’s algorithm on the graph $G'$, which is $G$ with node $\rho$ removed and node $j$ defined as the new root. By adding $(\rho \rightarrow j)$ to the newly sampled tree, we clearly have a dependency tree in $G$. We can sample the root edge $(\rho \rightarrow j)$ from the root weights $\rho$ to sample an unbiased dependency tree from the distribution. The pseudocode for this algorithm is given as $\text{wilson}_rc$ in Fig. 3.

Figure 3: Modification of Wilson (1996)’s algorithm to sample dependency trees from a graph. The lines that differ to $\text{wilson}$ are highlighted.
We present the algorithm using the above Laplacian
matrix without considering edges emanating from the root.
They then arbitrarily replace the first row of the Laplacian
matrix with the root edge weights. One can see that the first
row is chosen for convenience by examining the proof of
Proposition 1 in Koo et al. (2007). Indeed, the desired Laplacian
can be obtained by replacing any row by the root edge weights.

The Laplacian matrix for spanning trees is given by

\[ L_{ij} = \begin{cases} \sum_{i' \in N \setminus \{\rho, j\}} w_{i' \to j} & \text{if } i = j \\ -w_i \to j & \text{otherwise} \end{cases} \] (13)

We present the algorithm using the above Laplacian
matrix to sample dependency trees rather than spanning
trees. However, one can easily modify this algorithm to sample spanning trees.\(^6\) The premise of the algorithm is that we iteratively sample an
incoming edge to a non-root node of the graph until we
have a tree. Without loss of generality, we can enumerate the edges of any sampled tree \(t\) as \(e_1\) to \(e_N\). Therefore, at time step \(n\) of our sampling
algorithm, we will have a subset of our tree

\[ t_{<n} = [e_1, \ldots, e_{n-1}] \] (14)

Note that \(t_{<1} = \emptyset\) and \(t_{<N+1} = t\). We can then express the probability of a tree as

\[ p(t) = \prod_{n=1}^{N} p(e_n \mid t_{<n}) \] (15)

We sample the first edge \(e_1\) with probability \(p(e_1)\). We can find its marginal probability by taking
the derivative of the log partition function \(\log Z\) in
Theorem 3 which Koo et al. (2007) show to be\(^7\)

\[ p(i \to j) = \begin{cases} w_{\rho \to j}/B_{jj} & \text{if } i = \rho \\ w_i \to j (\delta_{j \neq 1} B_{jj} - \delta_{i \neq 1} B_{ij}) & \text{otherwise} \end{cases} \] (16)

where \(B = L^{-T}\) (the transpose of the inverse of \(L\)) and \(\delta_x = 1 \iff x\) is true, otherwise, \(\delta_x = 0\). Therefore, after computing \(B\) once, we can compute each \(p(i \to j)\) in \(O(1)\) time. Finding \(B\)
requires us to take a matrix inverse, and so runs in
\(O(N^3)\) time.\(^8\)

Each subsequent edge that we sample must be conditioned by all previously sampled edges \((t_{<n})\). At the \(n^{th}\) step, we have sampled \(t_{<n}\), and so our final sampled tree \(t\) will be such that
\(t_{<n} \subseteq t\). Therefore, sampling \(e_n\) from \(G\) is equivalent to sampling \(e_n\) from the subgraph \(G \supseteq t_{<n}\),
which is defined as the largest graph such that
\(t \in \mathcal{T}(G \supseteq t_{<n}) \Rightarrow t_{<n} \subseteq t\). Consequently, if \((i \to j) \in t_{<n}\), then \(G \supseteq t_{<n}\) does not contain any
other incoming edges to node \(j\) other than \((i \to j)\).\(^9\)

A correct ancestral sampling algorithm will sample an edge \(e\) for each non-root node from the graph
using (16), and then update graph to be \(G \supseteq e\) and
repeat. This algorithm will have to recompute \(B\)
\(O(N)\) times and so will have a runtime of \(O(N^4)\). We
show a graphical example of the algorithm in
Fig. 4. We give pseudocode for this as colbourn in
Fig. 5. The function sample_edge samples from
the distribution defined in (16) and the function
condition updates \(B\) to contain the transpose of
the Laplacian inverse of the conditioned graph. We

\(^6\)Koo et al. (2007)’s adaptation constructs the Laplacian
matrix without considering edges emanating from the root.

\(^8\)This runtime is also true by automatic differentiation
(Griewank and Walther, 2008) as finding \(Z\) takes \(O(N^3)\) time.

\(^9\)When sampling dependency trees, one would think that
if \((\rho \to j) \in t_{<n}\), we would need to remove all outgoing edges
from the root. However, by the construction of our Laplacian,
\(Z\) only accounts for dependency trees and so the marginals
already enforce this restriction. Therefore, we only need to
remove all other incoming edges to \(j\).

\(^7\)For spanning trees, the marginal probability can be similarly
derived as \(p(i \to j) = w_{i \to j} (B_{jj} - \delta_{i \neq 1} B_{ij})\) where \(B\) is
the transpose of the inverse of the Laplacian matrix in
footnote 6.
describe an efficient procedure for this conditioning step in the following section.

4.1 Efficiently Computing Marginals

Colbourn et al. (1996) show that we can update the marginals in $O(N^2)$ rather than $O(N^3)$ by using rank-one updates on $L$. Namely, Colbourn et al. (1996)'s adds an outer-product $Luv^\top$ to $L$ for some column vectors $u, v \in \mathbb{R}^N$ for each conditioning operation. We extend this to the weighted Laplacian for dependency trees.

**Lemma 3.** For any graph $G$ with Laplacian $L$ and any edge $e = (i \rightarrow j) \in E$, the $j$th column of the Laplacian $L^e$ of $G \supset e$ is given by

$$L^e_{[i,:]} = \begin{cases} u_{p \rightarrow j} \mathbf{1}_j & \text{if } i = \rho \\ u_{i \rightarrow j}(\delta_{j \neq 1} \mathbf{1}_j - \delta_{i \neq 1} \mathbf{1}_i) & \text{otherwise} \end{cases}$$

(17)

where $\mathbf{1}_j$ is the one-hot vector such that the $j$th element is 1. Furthermore, the $k$th column of $L^e$, where $k \neq j$, is equivalent to the $k$th column of $L$.

**Proof.** Consider the column $L^e_{[i,:]}$.

- **Case $k = j$:** Then the only incoming edge to $j$ in $G \supset e$ is $(i \rightarrow j)$.

- **Case $i = \rho$:** Then element $L^e_{1,j} = u_{p \rightarrow j}$ by Koo et al. (2007)'s construction. As there are no other incoming edges to $kj$, the remainder of the column is filled with zeros ans so $L^e_{[i,:]} = u_{p \rightarrow j} \mathbf{1}_i$.

- **Case $i \neq \rho$:** Then $L^e_{i,j} = -u_{i \rightarrow j}$ as long as $i \neq 0$. Since $i \neq \rho$, we also have that $L^e_{ij} = w_{i \rightarrow j}$ as long as $j \neq 0$. Therefore, we can represent the $j$th column by $w_{i \rightarrow j}(\delta_{j \neq 1} \mathbf{1}_j - \delta_{i \neq 1} \mathbf{1}_i)$.

- **Case $k \neq j$:** Then all incoming edges to node $k$ are still in $G \supset e$ and so $L^e_{[i,k]} = L_{[i,k]}$.

Lemma 3 shows that conditioning by an edge $(i \rightarrow j)$ is equivalent to a column replacement for $L$. A column replacement is equivalent to a rank-one update where $v = \mathbf{1}_j$ and $u = w_{i \rightarrow j}(1_j - \mathbf{1}_i) - L_{[i,j]}$ and $L_{[j,i]}$ is the $j$th column of $L$. Performing such a rank-one update speeds up the conditioning of $L$ from $O(N^2)$ to $O(N)$. More importantly, it lets us update $B$ in $O(N^2)$ using the Sherman–Morrison formula (Sherman and Morrison, 1950), which states that for any matrix $A \in \mathbb{R}^{N \times N}$ and column vectors $u, v \in \mathbb{R}^{N^10}$

$$(A + uv^\top)^{-1} = A^{-1} - T$$

(18)

where

$$T = \frac{A^{-1}u v^\top A^{-1}}{1 + v^\top A^{-1}u}$$

(19)

Recalling that $B$ requires the inverse transpose, and our choice of $v = \mathbf{1}_j$, we can simplify the expression for $T^\top$ to be

$$T^\top = \left( \frac{L^{-1}u_{[j,:]}^\top L^{-1}}{1 + u_{[j,:]}^\top L^{-1}u} \right)^\top$$

(20a)

$$= \frac{L^{-1}u_{[j,:]}^\top L^{-1}}{1 + u_{[j,:]}^\top L^{-1}u}$$

(20b)

$$= \frac{L_{[j,:]}^\top u_{[j,:]}^\top L^{-1}}{1 + u_{[j,:]}^\top L^{-1}u}$$

(20c)

$$= \frac{B_{[j,:]}^\top u_{[j,:]}^\top B_{[j,:]}^\top}{1 + u_{[j,:]}^\top B_{[j,:]}^\top}$$

(20d)

Therefore, we can update $B$ in $O(N^2)$. We give pseudocode for this efficient update as condition in Fig. 5.
Global Variables: $L$, $B$

```python
def colbourn(G):
    ▶ Sample a dependency tree from a graph $G$; requires $O(N^2) \text{ time, } O(N^2) \text{ space.}$
    $L \leftarrow \text{Laplacian}(G)$
    $B \leftarrow L^{-T}$  ▶ $O(N^3)$
    $t \leftarrow []$
    for $j \in \mathcal{N} \setminus \{\rho\}$:
        $e \leftarrow \text{sample_edge}(j)$
        $t$.append($e$)
        condition($e$)
    return $t$
def Laplacian(G):
    ▶ Construct the Laplacian of Koo et al. (2007) for dependency trees as in (13); requires $O(N^2) \text{ time, } O(N^2) \text{ space.}$
    $L \leftarrow 0$
    for $j \in \mathcal{N} \setminus \{\rho\}$:
        for $i \in \mathcal{N} \setminus \{\rho, j\}$:
            $L_{ij} = -w_{i \rightarrow j}$
            $L_{ji} = w_{i \rightarrow j}$
        $L_{ij} \leftarrow w_{j \rightarrow i}$
    ▶ For spanning trees, we can construct the Laplacian in footnote 6 by replacing line 18 with $L_{ij} \leftarrow w_{i \rightarrow j}$.
    return $L$
def sample_edge(j):
    ▶ Sample an incoming edge to $j$ using global variable $B$ as in (16); requires $O(N) \text{ time, } O(N) \text{ space.}$
    $m \leftarrow 0$
    $m_{p \rightarrow j} \leftarrow w_{p \rightarrow j}B_{1j}$
    for $i \in \mathcal{N} \setminus \{\rho, j\}$:
        $m_i \leftarrow w_{i \rightarrow j}(\delta_{j \neq 1}B_{ij} - \delta_{i \neq 1}B_{ij})$
    ▶ For spanning trees, we can construct the marginals in footnote 7 by replacing line 24 with $m_{p \rightarrow j} \leftarrow w_{p \rightarrow j}B_{ij}$ and line 26 with $m_i \leftarrow w_{i \rightarrow j}(B_{ij} - B_{i1})$.
    return sample from $m$
def condition(e):
    ▶ Condition the Laplacian and the transpose of its inverse to always include $e$ in any tree; requires $O(N^2) \text{ time, } O(N^2) \text{ space.}$
    Let $e = (i \rightarrow j)$
    if $i = \rho$
        $u \leftarrow w_{\rho \rightarrow j}1_j - L_{[i];j}$
    else:
        $u \leftarrow w_{i \rightarrow j}(\delta_{j \neq 1}1_j - \delta_{i \neq 1}1_i) - L_{[i];j}$
    $L_{[i];j} += u$
    $B = (B_{[j];i}u^T B) / (1 + u^T B_{[j];i})$
```

Figure 5: Algorithm for sampling dependency trees using the method of Colbourn et al. (1996). We describe the changes required to sample spanning trees in the comments.

**Theorem 4.** For any graph $G$, colbourn($G$) samples a dependency tree with probability

$$p(t) = \frac{1}{Z} \prod_{(i \rightarrow j) \in t} w_{i \rightarrow j}$$

Furthermore, colbourn runs in $O(N^3)$ time.

**Proof.** The probability of a tree $p(t)$ can equivalently be written as the product of the conditional edge marginals as in (15). To prove correctness, we prove by induction that for all $n \leq N$, at the $n^{th}$ call to sample_edge, $B$ contains the transpose of the Laplacian inverse of $G \supseteq t_{<n}$ and an edge $e$ with probability $p(e \mid t_{<n})$.

**Base case:** Then $n = 1$ and $t_{<1} = \emptyset$. $B$ contains the transpose of the Laplacian inverse of $G$ as expected and so sample_edge will sample an edge $e$ with probability $p(e)$ as expected.

**Inductive step:** Then $t_{<n} = [e_1, \ldots, e_{n-1}]$. At the $(n - 1)^{th}$ call to sample_edge, $B$ contains the transpose of the Laplacian inverse of $G \supseteq t_{<n-1}$. We then call condition, which by Lemma 3 and the Sherman–Morrison formula updates $B$ to be the transpose of the Laplacian inverse of $G \supseteq t_{<n}$. Therefore, at the $n^{th}$ call to sample_edge, $B$ contains the correct values and so sample_edge will sample an edge $e$ with probability $p(e \mid t_{<n})$.

Therefore, colbourn samples a dependency tree $t$ with the correct probability. We have $N$ iterations of the main loop, each call to sample_edge takes $O(N)$ times and each call to condition takes $O(N^2)$ time. These runtimes are easily observed from the pseudocode. Therefore, colbourn has a runtime of $O(N^3)$.

### 4.2 Runtime Experiment

We conduct a brief runtime experiment for colbourn and wilson_rc (see §3) whose runtimes are $O(N^3)$ and $O(H)$ respectively. We artificially generate random complete graphs of increasing size and measure the average sample time of each algorithm.\footnote{The experiment was conducted using an Intel(R) Core(TM) i7-7500U processor with 16GB RAM.} The results of the experiment are shown in Fig. 6. We note that despite colbourn being slower, the best-fit line for colbourn’s runtime has a slope of 1.42, suggesting it is much faster in practice than its complexity bound $O(N^3)$.

\footnote{We would expect the slope to be $\approx 3$ to match the complexity bound.}
Figure 6: Runtime experiment for sampling using wilson_rc and colbourn. For each graph size, we randomly generated 100 graphs and took 20 samples from each graph. The best fit lines for colbourn and wilson_rc have slopes of 1.42 and 1.14 respectively.

5 Sampling Without Replacement

In this section, we present a novel extension to colbourn that can sample dependency trees without replacement. SWOR algorithms are useful when we must sample multiple trees from the same graph. Specifically, when the distribution of trees over the graph is skewed so that a normal sampling algorithm frequently samples the same trees. This is often the case when the edge weights have been learned using a neural model (Dozat and Manning, 2017; Ma and Hovy, 2017). The SWOR algorithm we present follows the scheme of Shi et al. (2020).

In order to use colbourn to sample without replacement, we need an expression of the edge marginals conditioned on the set of previously sampled trees. If $D$ is the set of previously sampled trees, then we need to compute the following marginal probability efficiently

$$p(i \rightarrow j \mid D) = \frac{1}{Z_D} \sum_{t \in D \setminus D_0} w(t)$$

where $D_{ij}$ is the set of all dependency trees that contain edge $(i \rightarrow j)$ and

$$Z_D = \sum_{t \in D \setminus D_0} w(t) = Z - \sum_{t \in D} w(t)$$

**Lemma 4.** For any graph $G$, set of trees $D$, and edge $(i \rightarrow j) \in E$,

$$p(i \rightarrow j \mid D) = \frac{1}{Z_D} \left[ Z \cdot p(i \rightarrow j) - \sum_{t \in D_{ij}} w(t) \right]$$

where $D_{ij} \subseteq D$ is the set of trees in $D$ that contain the edge $(i \rightarrow j)$.

**Proof.**

$$p(i \rightarrow j \mid D) = \frac{1}{Z_D} \sum_{t \in D_{ij} \setminus D} w(t)$$

$$= \frac{1}{Z_D} \left[ \sum_{t \in D_{ij}} w(t) - \sum_{t \in D} w(t) \right]$$

$$= \frac{1}{Z_D} \left[ Z \cdot p(i \rightarrow j) - \sum_{t \in D_{ij}} w(t) \right]$$

Lemma 4 then gives a new formula to use for sampling edges by re-weighting the probability of an edge.\(^\text{13}\) We can then compute the marginal distribution for an incoming edge to a node in $\mathcal{O}(N + K)$ time. Note that (24) makes explicit use of $Z$ which is not needed for the original marginals in (16). Consequently, as we sample an edge from the tree, we must condition $Z$ as well as $B$. Fortunately, this can be done in $\mathcal{O}(N)$ using the matrix determinant Lemma, which states that for any matrix $A \in \mathbb{R}^{N \times N}$ and column vectors $u, v \in \mathbb{R}^N$

$$|A + uv^\top| = |A|(1 + v^\top A^{-1} u)$$

Furthermore, at each conditioning step, we must also update $D$ (and $Z_D$), to only include the sampled trees containing the new sampled edge. These can both be achieved in $\mathcal{O}(K)$ time where $K$ is the number of trees that we sample. The pseudocode for the sampling and conditioning steps are given as sample_edge' and condition' in Fig. 7. The sampling algorithm itself, swor is similar to colbourn in Fig. 5. However, it samples $K$ dependency trees rather than a single dependency tree and stores additional variables in order to cache frequently used values such as the original $Z$, $L$, and $B$ values.

**Theorem 5.** For any graph $G$ and $K > 0$, swor($G, K$) samples $K$ dependency trees without replacement, where $t$ is sampled with probability

$$p(t \mid D) = \frac{\delta_{t,D}}{Z_D} \prod_{(i \rightarrow j) \in t} w_{i \rightarrow j}$$

\(^{13}\)Re-weighting has been recently used by Stanojević and Cohen (2021) to speed-up algorithms for single-root tree decoding algorithms.
Global Variables: $L$, $B$, $D$, $Z$, and $Z_D$

1. def swor($G$, $K$):
2.   $\triangleright$ Sample $K$ dependency trees without replacement from a graph $G$; requires $O(KN^3 + K^2N)$ time, $O(N^3 + KN)$ space.
3.   $L' \leftarrow \text{Laplacian}(G)$
4.   $Z' \leftarrow |L|; B' \leftarrow L'^{-T}$ \triangleright $O(N^3)$
5.   $D' \leftarrow []; Z_D' \leftarrow Z'$
6.   for $k \in \{1, \ldots, K\}$:
7.     $D \leftarrow D' ; Z \leftarrow Z'$
8.     $Z_D \leftarrow Z_D'$
9.     $L \leftarrow L'; B \leftarrow B'; t \leftarrow []$
10.    for $j \in N \setminus \{\rho\}$ :
11.       $e \leftarrow \text{sample_edge}'(j)$
12.       $t.append(e)$
13.       condition'(e)
14.       $D'.append(t); Z_D' = w(t)$
15.    return $D'$
16. def sample_edge'(j):
17.   $\triangleright$ Sample an incoming edge to $j$ using global variables $B$, $Z$, and $Z_D$ as in (24); requires $O(N)$ time, $O(N)$ space.
18.   $m \leftarrow 0$
19.   $m_p \leftarrow Z \sum_{i \in T_p} w(t)$
20.   for $i \in N \setminus \{\rho, j\}$:
21.   \hspace{1em} $m_i \leftarrow Z w_i \sum_{i \neq j} (\delta_{j \neq 1} \delta_{j \neq i} - \delta_{i \neq 1} \delta_{i \neq j})$
22.   \hspace{1em} $m_i \leftarrow \sum_{i \in T_p} w_i$
23.   return sample from $\frac{1}{Z_D} m$
24. def condition'(e):
25.   $\triangleright$ Condition the Laplacian, the transpose of the Laplacian inverse, the partition partition function, and the set of previously sampled trees to always include $e$ in any tree; requires $O(N^3 + K)$ time, $O(N^2)$ space.
26.   Let $e = (i \rightarrow j)$
27.   if $i = \rho$
28.       $u \leftarrow w_p j I^T \rightarrow L[i:j]$  \hspace{1em} 5
29.   else:
30.       $u \leftarrow w_i \sum_{j \neq i \neq 1} \left(\delta_{j \neq 1} I^T j - \delta_{i \neq 1} I^T i\right) - L[i:j]$
31.       $Z \leftarrow Z + u I^T B_{[i,j]}$
32.       $B \leftarrow (B_{[i,j]} I^T B) / (1 + u I^T B_{[i,j]})$
33.       $D \leftarrow D_{[i,j]}$
34.       $Z_D \leftarrow Z - \sum_{i \in D} w(t)$

Figure 7: Algorithm for sampling dependency trees without replacement.  

where $D$ is the set of trees sampled prior to $t$.  Furthermore, swor runs in $O(KN^3 + K^2N)$ time.

Proof. To prove correctness, we prove by induction that the $K$th sampled tree is sampled with the probability in (27).

Base case: Then $K = 1$ and so $D' = \emptyset$ and $Z_D' = Z'$. Therefore, swor executes colbourn and samples tree $t$ with probability $p(t)$ as expected.

Inductive step: Assume that $D'$ contains $K - 1$ trees, which were each sampled with the correct probability. Then $Z_D' = Z' - \sum_{i \in D} w(t)$ and so by Lemma 4, sample_edge’ will sample the first edge of the new tree with the correct probability. We can then prove that all edges of the new tree are sampled with the correct probability using an inductive proof analogous to Theorem 4.

We require $O(N^3)$ to find $Z'$ and $B'$. Then for each of the $K$ sampled trees, we have $N$ iterations of the main loop. Each call to sample_edge’ takes $O(N + K)$ times and each call to condition’ takes $O(N^2 + K)$ time. These runtimes are easily observed from the pseudocode. Therefore, swor has a runtime of $O(KN^3 + K^2N)$.  

6 Conclusion

We presented two efficient approaches to sample spanning trees subject to a root constraint, which were based on prior algorithms by Wilson (1996) and Colbourn et al. (1996). While Wilson (1996)’s $O(H)$ algorithm was more rapid, Colbourn et al. (1996)’s $O(N^3)$ algorithm is extendable to a novel sampling without replacement algorithm. To the best of our knowledge, not much work has been done in graph-based dependency parsing to sample dependency trees, and none has used sampling without replacement. We hope that this paper serves as a tutorial for how this can be done and encourages the use of sampling in future work.

Acknowledgments

We would like to thank all the reviewers for their invaluable feedback and time spent engaging with our work. The first author is supported by the University of Cambridge School of Technology Vice-Chancellor’s Scholarship as well as by the University of Cambridge Department of Computer Science and Technology’s EPSRC.

References

David J. Aldous. 1990. The random walk construction of uniform spanning trees and uniform labelled trees. Society for Industrial and Applied Mathematics Journal of Discrete Mathematics, 3(4):450–465.
Nicholas Andrews, Jason Eisner, and Mark Dredze. 2012. Name phylogeny: A generative model of string variation. In Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, pages 344–355, Jeju Island, Korea. Association for Computational Linguistics.

Eduard Bejček, Eva Hajíčová, Jan Hajíč, Pavlína Jinová, Václava Kettnerová, Veronika Kolářová, Marie Mikulová, Jiří Mirovský, Anna Nedoluzhko, Jarmila Panevová, Lucie Poláková, Magda Ševčíková, Jan Štěpánek, and Šárka Zikánová. 2013. Prague dependency treebank 3.0.

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

Elizabeth Clark, Yangfeng Ji, and Noah A. Smith. 2018. Neural text generation in stories using entity representations as context. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers), pages 2250–2260, New Orleans, Louisiana. Association for Computational Linguistics.

Charles J. Colbourn, Wendy J. Myrvold, and Eugene Neufeld. 1996. Two algorithms for unranking arborescences. Journal of Algorithms, 20(2):268–281.

Timothy Dozat and Christopher D. Manning. 2017. Deep biaffine attention for neural dependency parsing. In Proceedings of the International Conference on Learning Representations.

William Fedus, Ian J. Goodfellow, and Andrew M. Dai. 2018. MaskGAN: Better text generation via filling in the _______. In 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings.

Harold N. Gabow and Robert Endre Tarjan. 1984. Efficient algorithms for a family of matroid intersection problems. Journal of Algorithms, 5(1).

Andreas Griewank and Andrea Walther. 2008. Evaluating Derivatives—Principles and Techniques of Algorithmic Differentiation, 2 edition. Society for Industrial and Applied Mathematics.

A. Guénoche. 1983. Random spanning tree. Journal of Algorithms, 4(3):214–220.

Katherine Keith, Su Lin Blodgett, and Brendan O’Connor. 2018. Monte Carlo syntax marginals for exploring and using dependency parses. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers), pages 917–928, New Orleans, Louisiana. Association for Computational Linguistics.

Gustav Kirchhoff. 1847. Über die auflösung der gleichungen, auf welche man bei der untersuchung der linearen vertheilung galvanischer ströme geführt wird. Annalen der Physik, 148(12).

Terry Koo, Amir Globerson, Xavier Carreras, and Michael Collins. 2007. Structured prediction models via the matrix-tree theorem. In Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning.

Sandra Kübler, Ryan T. McDonald, and Joakim Nivre. 2009. Dependency Parsing. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers.

Vidyadhar G. Kulkarni. 1990. Generating random combinatorial objects. Journal of Algorithms, 11(2):185–207.

Gregory F. Lawler. 1979. A Self-Avoiding Random Walk. Ph.D. thesis, Princeton University.

Robert L. Logan IV, Matt Gardner, and Sameer Singh. 2020. On importance sampling-based evaluation of latent language models. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 2171–2176, Online. Association for Computational Linguistics.

Xuezhe Ma and Eduard Hovy. 2017. Neural probabilistic model for non-projective MST parsing. In Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 1: Long Papers), Taipei, Taiwan. Asian Federation of Natural Language Processing.

David Mareček and Zdeněk Žabokrtský. 2011. Gibbs sampling with treeness constraint in unsupervised dependency parsing. In Proceedings of Workshop on Robust Unsupervised and Semisupervised Methods in Natural Language Processing, pages 1–8, Hisar, Bulgaria. Association for Computational Linguistics.

David McClosky, Mihai Surdeanu, and Christopher Manning. 2011. Event extraction as dependency parsing for BioNLP 2011. In Proceedings of BioNLP Shared Task 2011 Workshop, pages 41–45, Portland, Oregon, USA. Association for Computational Linguistics.

Ryan McDonald and Giorgio Satta. 2007. On the complexity of non-projective data-driven dependency parsing. In Proceedings of the Tenth International Conference on Parsing Technologies, Prague, Czech Republic. Association for Computational Linguistics.

Andriy Mnih and Geoffrey E. Hinton. 2007. Three new graphical models for statistical language modelling. In Machine Learning, Proceedings of the Twenty-Fourth International Conference (ICML 2007), Corvallis, Oregon, USA, June 20-24, 2007, pages 641–648.
Kensen Shi, David Bieber, and Charles Sutton. 2020. Incremental sampling without replacement for sequence models. In Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event, pages 8785–8795.

Sameer Singh, Michael Wick, and Andrew McCallum. 2012. Monte Carlo MCMC: Efficient inference by approximate sampling. In Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, pages 1104–1113, Jeju Island, Korea. Association for Computational Linguistics.

David A. Smith and Noah A. Smith. 2007. Probabilistic models of nonprojective dependency trees. In Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL), Prague, Czech Republic. Association for Computational Linguistics.

Miloš Stanojević and Shay B. Cohen. 2021. A root of a problem: Optimizing single-root dependency parsing. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), Online. Association for Computational Linguistics.

W. T. Tutte. 1984. Graph Theory. Addison-Wesley Publishing Company.

Wenhui Wang and Baobao Chang. 2016. Graph-based dependency parsing with bidirectional LSTM. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 2306–2315, Berlin, Germany. Association for Computational Linguistics.

Stanley G. Williamson. 1985. Combinatorics for Computer Science. Computer Science Press.

David Bruce Wilson. 1996. Generating random spanning trees more quickly than the cover time. In Proceedings of the Twenty-Eighth Annual Association for Computing Machinery Symposium on the Theory of Computing, pages 296–303.

Yuan Zhang, Tao Lei, Regina Barzilay, and Tommi Jaakkola. 2014a. Greed is good if randomized: New inference for dependency parsing. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 1013–1024, Doha, Qatar. Association for Computational Linguistics.

Yuan Zhang, Tao Lei, Regina Barzilay, Tommi Jaakkola, and Amir Globerson. 2014b. Steps to excellence: Simple inference with refined scoring of dependency trees. In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 197–207, Baltimore, Maryland. Association for Computational Linguistics.

Ran Zmigrod, Tim Vieira, and Ryan Cotterell. 2020. Please mind the root: Decoding arborescences for dependency parsing. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 4809–4819, Online. Association for Computational Linguistics.

Ran Zmigrod, Tim Vieira, and Ryan Cotterell. 2021. Efficient computation of expectations under spanning tree distributions. Transactions of the Association for Computational Linguistics, 9:675–690.