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 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).

\(^1\)Our implementation of these algorithms is publicly available at https://github.com/rycolab/treesample.

\(^2\)Directed spanning trees are known as arborescences in the graph theory literature (Williamson, 1985). However, we will simply refer to them as spanning trees.

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; Kirchoff, 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 Dependency (UD) scheme (Nivre et al., 2018), is that a dependency tree may only have one edge em-
We consider distributions over spanning trees in
Wilson (1996). 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)\(^3\) to efficiently sample directed spanning trees subject to a root constraint while maintaining the runtime of the original algorithms. While our modification to Colbourn et al. (1996)’s algorithm produces biased samples. 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 root ed directed weighted graphs (graphs for short). A graph is denoted by \(G = (\rho, \mathcal{N}, \mathcal{E})\) where \(\mathcal{N}\) 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 \((i \rightarrow j)\). We note that the non-root nodes \(\mathcal{N} \setminus \{\rho\}\) can be enumerated as \([1, \ldots, N]\). 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 \(j \in \mathcal{N} \setminus \{\rho\}\) 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.\(^4\) 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 \(\mathcal{T}(G)\) and the set of all dependency trees of a graph \(G\) is given by \(\mathcal{D}(G)\). Note that \(\mathcal{D}(G) \subseteq \mathcal{T}(G)\). When the graph \(G\) is clear from context, we will refer to these sets \(\mathcal{T}\) and \(\mathcal{D}\) respectively.

The weight of a tree \(t\) is the product of its edge weights:

\[
w(t) \equiv \prod_{(i \rightarrow j) \in t} w_{i \rightarrow j} \tag{1}\n\]

The probability of a spanning tree is then given by

\[
p(t) \defeq \frac{w(t)}{Z} \quad \text{where} \quad Z \defeq \sum_{t \in \mathcal{T}} w(t) \tag{2}\n\]

The normalizing constant in the case of dependency trees sums over \(\mathcal{D}\) instead of \(\mathcal{T}\).

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 \(j \in \mathcal{N} \setminus \{\rho\}\) we have that

\[
\sum_{i \in \mathcal{N}} w_{i \rightarrow j} = 1 \tag{3}\n\]

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

\[
w'_{i \rightarrow j} \defeq \frac{w_{i \rightarrow j}}{\sum_{j' \in \mathcal{N}} w'_{i \rightarrow j'}} \tag{4}\n\]

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

Lemma 1. For any tree \(t \in \mathcal{T}\),

\[
w'(t) \propto w(t) \tag{5}\n\]

\(^3\)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 amenable to sampling without replacement and it is generally slower than the algorithm of Wilson (1996).

\(^4\)However, we note that there are exceptions that do not require the root constraint, such as the Prague Treebank (Bejček et al., 2013).
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 ①) and we start the walk at ③ with the tree only containing the root node. The random walk includes a cycle with nodes ②, ③, and ④, this cycle is erased when we create the path from ③ back up to ①.

Note that the arrows here mark the path upwards rather than the edges in the tree, the tree edges are reversed (e.g., ② → ③).

Proof.

\[ w(t) = \prod_{(i \to j) \in t} w'_{i \to j} \]

\[ = \prod_{(i \to j) \in t} \left( \sum_{i' \in \mathcal{N}'} w'_{i' \to j} \right)^{-1} \]

\[ \prod_{i \in \mathcal{N} \setminus \{\rho\}} \left( \sum_{i' \in \mathcal{N}'} w'_{i' \to j} \right)^{-1} \]

\[ \propto w(t) \] (6d)

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 \( \prod_{i \in \mathcal{N} \setminus \{\rho\}} \left( \sum_{i' \in \mathcal{N}'} w'_{i' \to j} \right)^{-1} \).

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 ρ. 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

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

```
1. def wilson(G):
2.     ▷ Sample a spanning tree from a graph G; requires \( \mathcal{O}(H) \) time, \( \mathcal{O}(N^2) \) space.
3.     t ← 0
4.     visited ← \{ρ\}
5.     ▷ The following for and while loops take \( \mathcal{O}(H) \) time to execute. More specifically, line 9 is called \( \mathcal{O}(H) \) times.
6.     for i \( \in \mathcal{N} \setminus \{ρ\} \):
7.         u ← i
8.         while u \( \notin \) visited:
9.             Sample \( v \in \mathcal{N} \) with weight \( w'_{v \to u} \)
10.            \( t_u \leftarrow v \)
11.            u ← v
12.         u ← i
13.         while u \( \notin \) visited:
14.             visited.add(u)
15.             u ← v such that \( t_u = v \)
16.     return t
```

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, \( \hat{S} \), and those that do not, \( \overline{S} \).

**Lemma 2.** For any sequence of edges \( S \) sampled from a random walk, we have

\[ p(S) = p(\hat{S}) p(\overline{S}) \] (7)

Proof.

\[ p(S) = \prod_{(i \to j) \in S} w'_{i \to j} \]

\[ = \left( \prod_{(i \to j) \in \hat{S}} w'_{i \to j} \right) \left( \prod_{(i \to j) \in \overline{S}} w'_{i \to j} \right) \] (8a)
Furthermore, 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 \( \text{wilson}_\text{rc} \) in Fig. 3.

**Theorem 1.** For any graph \( G \), \( \text{wilson}(G) \) samples a directed spanning tree \( t \in T \) with probability

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

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

**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 \( S \). In particular, we sample \( t \) with probability

\[
p(t) = \prod_{(i \rightarrow j) \in t} w'_{i \rightarrow j}
\]

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 \( \text{wilson}_\text{rc} \) in Fig. 3.

**Theorem 2.** For any graph \( G \), \( \text{wilson}_\text{rc}(G) \) samples a dependency tree from a biased distribution. Furthermore, \( \text{wilson}_\text{rc} \) runs in \( O(H) \) time.

**Proof.** We note that executing lines line 6 to line 18 is equivalent to running Wilson on the graph \( G' \) that is rooted at \( j \) (and does not have \( \rho \)). By Theorem 1, this results in a tree \( t' \in T(G') \). As we

---

5Note that this theorem has been corrected following Stanjević (2022), who showed that \( \text{wilson}_\text{rc} \) returns a biased sample by counter-example. We have included their counter-example into our proof. They additionally provide an alternative solution to extend Wilson to use rejection sampling in order to sample a dependency tree. They show that on average Wilson would be run three times to find a successful dependency tree sample, thus maintaining the \( O(H) \) runtime. We refer the reader to Stanjević (2022, Section 4) for further details.
then sample an edge emanating from the root, we sample a valid dependency tree \( t \). We show that \( t \) is sampled from a biased distribution through an example. Consider a graph with two edges emanating from the root, both with weight \( \frac{1}{2} \). Suppose that there exist two dependency trees using one of the edges, and only one dependency tree using the other edge. Then we should sample the root edges with their marginal probabilities \( \frac{2}{7} \) and \( \frac{1}{7} \) respectively. However, we sample the root edges by their edge weight \( \frac{1}{2} \). Therefore, \( t \) is sampled from a bias distribution.

Furthermore, as line 3 of \texttt{wilson_rc} takes \( O(N) \) which is less than \( O(H) \), by Theorem 1, \texttt{wilson_rc} has a runtime of \( O(H) \).

### 4 Ancestral Sampling

In this section, we present an extension to the ancestral sampling algorithm of Colbourn et al. (1996) to the weighted graph case. This algorithm relies on the efficient computation of \( Z \) using the MTT (Kirchoff, 1847; Tutte, 1984), allows us to compute \( Z \) in \( O(N^3) \) by taking the determinant of the Laplacian matrix, \( L \in \mathbb{R}^{N \times N} \). We use Koo et al. (2007)'s adaptation of the MTT to dependency trees.\(^6\)

**Theorem 3** (Proposition 1, Koo et al. (2007)). For any graph \( G \), the normalization constant \( Z \) over the distribution of dependency trees \( D \) is given by

\[
Z = |L| \quad \text{where} \\
L_{ij} = \begin{cases} 
  w_{i \rightarrow j} & \text{if } i = 1 \\
  \sum_{i \in N \setminus \{\rho, i\}} w_{i \rightarrow j} & \text{if } i = j \\
  -w_{i \rightarrow j} & \text{otherwise}
\end{cases} \quad (12)
\]

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.\(^7\) 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

\[
t_{<n} = [e_1, \ldots, e_{n-1}] \quad (13)
\]

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

\[
p(t) = \prod_{n=1}^{N} p(e_n \mid t_{<n}) \quad (14)
\]

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\(^8\)

\[
p(i \rightarrow j) = \begin{cases} 
  w_{i \rightarrow j} B_{ij} & \text{if } i = \rho \\
  w_{i \rightarrow j} (\delta_{j \neq 1} B_{jj} - \delta_{i \neq 1} B_{ij}) & \text{otherwise}
\end{cases} \quad (15)
\]

where \( B = L^{-1} \) (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 \( p(i \rightarrow j) \) in \( O(1) \) time. Finding \( B \) requires us to take a matrix inverse, and so runs in \( O(N^3) \) time.\(^9\)

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 \( G \supset t_{<n} \), which is defined as the largest graph such that \( t \in T(G \supset t_{<n}) \implies t_{<n} \subseteq t \). Consequently, if \( (i \rightarrow j) \in t_{<n} \), then \( G \supset t_{<n} \) does not contain any other incoming edges to node \( j \) other than \( (i \rightarrow j) \).\(^{10}\)

A correct ancestral sampling algorithm will sample an edge \( e \) for each non-root node from the graph

\(^6\)Koo et al. (2007)'s adaptation constructs the 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.

\(^7\)The Laplacian matrix for spanning trees is given by

\[
L_{ij} = \begin{cases} 
  \sum_{i \in N \setminus \{i\}} w_{i \rightarrow j} & \text{if } i = j \\
  -w_{i \rightarrow j} & \text{otherwise}
\end{cases}
\]

\(^8\)For spanning trees, the marginal probability can be similarly derived as

\[
p(i \rightarrow j) = w_{i \rightarrow j} (B_{jj} - \delta_{i \neq 1} B_{ij})
\]

where \( B \) is the transpose of the inverse of the Laplacian matrix in footnote 7.

\(^9\)This runtime is also true by automatic differentiation (Griewank and Walther, 2008) as finding \( Z \) takes \( O(N^3) \) time.

\(^{10}\)When sampling dependency trees, one would think that if \( (p \rightarrow j) \in t_{<n} \), 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 \( j \).
using (15), and then update graph to be \( G \succ e \) 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 colbournO in Fig. 5. The function sample_edge samples from the distribution defined in (15) and the function condition updates \( B \) to contain the transpose of the Laplacian inverse of the conditioned graph. We 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 compute 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 \( u v^\top \) 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 \( e = (i \rightarrow j) \in \mathcal{E} \), the \( j \)-th column of the Laplacian \( L^e \) of \( G \succ e \) is given by

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

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

**Proof.** Consider the column \( L^e_{[\cdot,k]} \).

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

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

**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 \( u = w_{i \rightarrow j} (\mathbf{1}_j - \mathbf{1}_i) \) and \( L_{[\cdot,j]} \) and \( L_{[\cdot,j]} \) 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 \( u, v \in \mathbb{R}^N \)

\[
(A + uv^\top)^{-1} = A^{-1} - \frac{T}{1 + v^\top A^{-1} u}
\]  

(17)

where

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

(18)

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

\[
T^\top = \left( \frac{L^{-1} u \mathbf{1}_j^\top L^{-1} u}{1 + u^\top L^{-1} u} \right)^\top
\]

(19a)

\[
= \left( \frac{L^{-1} u L_{[\cdot,j]}^\top}{1 + u^\top L^{-1} u} \right) u^\top L^{-1} u
\]

(19b)

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

(19c)

The Sherman–Morrison formula can be computed in \( O(N^2) \) due to the associativity of matrix multiplication.

---

![Graph example](image-url)

Figure 4: Consider sampling a tree from the fully connected graph \( G \) given in (a). We do this by sampling an incoming edge to each non-root node. We first sample an incoming edge to \( \rho \), the possible edges are dashed in (a). Suppose we sample \( e_1 \) with probability \( p(e_1) \), then we have \( t_{<2} = \{e_1\} \). If we include \( e_1 \) in our graph as in (b), and repeat the process, we will sample edge \( e_2 \) with probability \( p(e_2 \mid t_{<2}) \). We now have \( t_{<3} = \{e_1, e_2\} \), and we can sample an incoming edge \( e_3 \) to \( \rho \) with probability \( p(e_3 \mid t_{<3}) \) as in (c). We can similarly find \( t_{<4} \) in (d). Finally, in (e), we have \( t_{<5} = t = \{e_1, e_2, e_3, e_4\} \), which is a tree in \( T(G) \). Note \( T(G \succ t) = \{t\} \).
Global Variables: \( L, B \)

1. \( \text{def} \ colbourn(G): \)
2. \( \triangleright \) Sample a dependency tree from a graph \( G \); requires \( O(N^3) \) time, \( O(N^2) \) space.
3. \( L \leftarrow \text{Laplacian}(G) \)
4. \( B \leftarrow L^{-T} \quad \triangleright O(N^3) \)
5. \( t \leftarrow [] \)
6. for \( j \in N \setminus \{ \rho \} : \)
7. \( e \leftarrow \text{sample_edge}(j) \)
8. \( t.\text{append}(e) \)
9. \( \text{condition}(e) \)
10. return \( t \)
11. \( \text{def} \ \text{Laplacian}(G): \)
12. \( \triangleright \) Construct the Laplacian of Koo et al. (2007) for dependency trees as in (12): requires \( O(N^2) \) time, \( O(N^2) \) space.
13. \( L \leftarrow 0 \)
14. for \( j \in N \setminus \{ \rho \} : \)
15. for \( i \in N \setminus \{ \rho, j \} : \)
16. \( L_{ij} \leftarrow -w_{i \rightarrow j} \)
17. \( L_{jj} \leftarrow w_{i \rightarrow j} + w_{\rho \rightarrow j} \)
18. \( L_{ij} \leftarrow w_{\rho \rightarrow j} \)
19. \( \triangleright \) For spanning trees, we can construct the Laplacian in footnote 7 by replacing line 18 with \( L_{ij} \leftarrow w_{i \rightarrow j} \).
20. return \( L \)
21. \( \text{def} \ \text{sample_edge}(j): \)
22. \( \triangleright \) Sample an incoming edge to \( j \) using global variable \( B \) as in (15); requires \( O(N) \) time, \( O(N) \) space.
23. \( m \leftarrow 0 \)
24. \( m_p +\ = w_{\rho \rightarrow j} B_{1j} \)
25. for \( i \in N \setminus \{ \rho, j \} : \)
26. \( m_i +\ = w_{i \rightarrow j} (\delta_{j \neq 1} B_{ij} - \delta_{i \neq 1} B_{ij}) \)
27. \( \triangleright \) For spanning trees, we can construct the marginals in footnote 8 by replacing line 24 with \( m_p +\ = w_{\rho \rightarrow j} B_{ij} \) and line 26 with \( m_i +\ = w_{i \rightarrow j} (B_{ij} - B_{ij}) \).
28. return \( \text{sample from } m \)
29. \( \text{def} \ \text{condition}(e): \)
30. \( \triangleright \) Condition the Laplacian and the transpose of its inverse to always include \( e \) in any tree; requires \( O(N^2) \) time, \( O(N^2) \) space.
31. Let \( e = (i \rightarrow j) \)
32. if \( i = \rho : \)
33. \( u \leftarrow w_{\rho \rightarrow j} \delta_{i=1} - L[i,j] \)
34. else:
35. \( u \leftarrow w_{i \rightarrow j} (\delta_{j \neq 1} \delta_{\rho=1} - \delta_{i \neq 1} \delta_{\rho=1}) - L[i,j] \)
36. \( L[i,j] +\ = u \)
37. \( B \leftarrow (B[i,j] u^T B) / (1 + u^T B[i,j]) \)

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.

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

**Theorem 4.** For any graph \( G \), \( \text{colbourn}(G) \) samples a dependency tree with probability

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

Furthermore, \( \text{colbourn} \) 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 (14). To prove correctness, we prove by induction that for all \( n \leq N \), at the \( n^{th} \) call to \( \text{sample_edge} \), \( B \) contains the transpose of the Laplacian inverse of \( G \succ t_{<n} \) 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 \( \text{sample_edge} \) will sample an edge \( e \) with probability \( p(e \mid t_{<n}) \) as expected.

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

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

**4.2 Runtime Experiment**

We conduct a brief runtime experiment for \( \text{colbourn} \) and \( \text{wilson_rc} \) (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.}

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

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

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

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

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

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

(23)

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

Proof.

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

(24a)

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

(24c)

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

(24d)

$\blacksquare$

Lemma 4 then gives a new formula to use for sampling edges by re-weighting the probability of an edge.\footnote{Re-weighting has been recently used by Stanojević and Cohen (2021) to speed-up algorithms for single-root tree decoding algorithms.} We can then compute the marginal distribution for an incoming edges to a node in $O(N + K)$ time. Note that (23) makes explicit use of $Z$ which is not needed for the original marginals in (15). Consequently, as we sample an edge from the tree, we must condition $Z$ as well as $B$. Fortunately, this can be done in $O(N)$ using the matrix determinant Lemma, which states that for any matrix $A \in \mathbb{R}^{N \times N}$ and column vectors $u, v \in \mathbb{R}^N$

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

(25)

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 $O(K)$ 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 $K > 0$, $\text{swor}(G, K)$ samples $K$ dependency trees without replacement, where $t$ is sampled with probability

$$p(t \mid D) = \frac{\delta_{\log D}}{Z_D} \prod_{(i \rightarrow j) \in t} w_{i \rightarrow j} \quad (26)$$

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

Proof. To prove correctness, we prove by induction that the $K$th sampled tree is sampled with the probability in (26).

Base case: Then $K = 1$ and so $D' = \emptyset$ and $Z_D = Z'$. Therefore, $\text{swor}$ 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 $Z_D' = Z' - \sum_{t \in D'} w(t)$ and so by Lemma 4, $\text{sample_edge}'$ 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 $\text{sample_edge}'$ takes $O(N + K)$ times and each call to $\text{condition}'$ takes $O(N^2 + K)$ time. These runtimes are easily observed from the pseudocode. Therefore, $\text{swor}$ has a runtime of $O(KN^3 + K^2 N)$.

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


Ryan McDonald and Giorgio Satta. 2007. On the complexity of non-projective data-driven dependency


Wenzhe Pei, Tao Ge, and Baobao Chang. 2015. An effective neural network model for graph-based de-


