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 $\mathcal{O}(H)$ where H is the mean hitting time of the graph. Colbourn et al. (1996)'s sampling algorithm has a running time of $\mathcal{O}(N^3)$, 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 $\mathcal{O}(KN^3 + K^2N)$ 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 phylogenic 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 $\mathcal{O}(N^5)$ algorithm which was then improved by Kulkarni (1990) and Colbourn et al. (1996). Colbourn et al. (1996) give an $\mathcal{O}(N^3)$ 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-

[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.

Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pages 10558–10569
November 7–11, 2021. ©2021 Association for Computational Linguistics

pendency (UD) scheme (Nivre et al., 2018), is that a dependency tree may only have one edge emanating from the designated root symbol. Algorithms exists 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)[3] 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 $G = (\rho, \mathcal{N}, \mathcal{E})$ where $\mathcal{N}$ is a set of $N+1$ nodes including a designated root node ρ and $\mathcal{E}$ is a set of ordered pairs between two nodes $(i \to j)$. We note that the non-root nodes $\mathcal{N} \setminus \{\rho\}$ can be enumerated as $[1, \ldots, N]$. For each edge $(i \to j) \in \mathcal{E}$, we associate a non-negative weight $w_{i \to j} \in \mathbb{R}_{\geq 0}$. Note that $w_{i \to j} = 0$ when $(i \to j) \notin \mathcal{E}$.

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 ρ. 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:

$$w(t) \stackrel{\text{def}}{=} \prod_{(i \to j) \in t} w_{i \to j} \tag{1}$$

The **probability** of a spanning tree is then given by

$$p(t) \stackrel{\text{def}}{=} \frac{w(t)}{\mathrm{Z}} \quad \text{where} \quad \mathrm{Z} \stackrel{\text{def}}{=} \sum_{t \in \mathcal{T}} w(t) \tag{2}$$

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

$$\sum_{i \in \mathcal{N}} w_{i \to j} = 1 \tag{3}$$

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 $(i \to j)$ has a weight $w'_{i \to j}$ defined by

$$w'_{i \to j} \stackrel{\text{def}}{=} \frac{w_{i \to j}}{\sum_{i' \in \mathcal{N}} w_{i' \to j}} \tag{4}$$

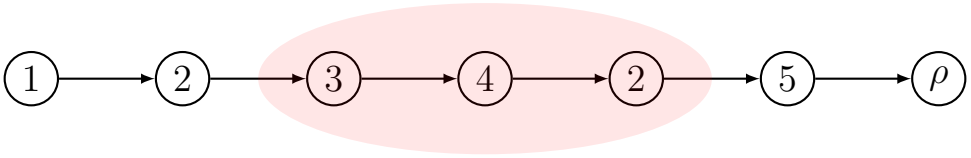
We can also define this new weight over a tree $w'(t) \stackrel{\text{def}}{=} \prod_{(i \to j) \in t} w'_{i \to j}$.

Lemma 1. *For any tree $t \in \mathcal{T}$,*

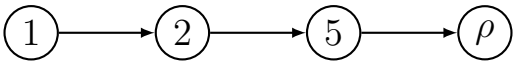
$$w'(t) \propto w(t) \tag{5}$$

[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 $\mathcal{O}(N^3)$. 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).

[4] However, we note that there are exceptions that do not require the root constraint, such as the Prague Treebank (Bejček et al., 2013).



(a) Random walk path from node ① to ⓡ.



(b) Tree path from node ① to ⓡ.

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 ⓡ) and we start the walk at ① 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 ① back up to ⓡ. Note that the arrows here mark the path upwards rather than the edges in the tree, the tree edges are reversed (e.g, ⓡ → ⑤).

Proof.

$$w'(t) = \prod_{(i \to j) \in t} w'_{i \to j} \tag{6a}$$

$$= \prod_{(i \to j) \in t} \frac{w_{i \to j}}{\sum_{i' \in \mathcal{N}} w_{i' \to j}} \tag{6b}$$

$$= \left(\prod_{(i \to j) \in t} \frac{1}{\sum_{i' \in \mathcal{N}} w_{i' \to j}} \right) \left(\prod_{(i \to j) \in t} w_{i \to j} \right)$$

$$= \underbrace{\left(\prod_{(i \to j) \in t} \frac{1}{\sum_{i' \in \mathcal{N}} w_{i' \to j}} \right)}_{\text{constant}} w(t) \tag{6c}$$

$$\propto w(t) \tag{6d}$$

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

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

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

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

Lemma 2. *For any sequence of edges $\mathcal{S}$ sampled from a random walk, we have*

$$p(\mathcal{S}) = p(\widehat{\mathcal{S}})\, p(\overline{\mathcal{S}}) \tag{7}$$

Proof.

$$p(\mathcal{S}) = \prod_{(i \to j) \in \mathcal{S}} w'_{i \to j} \tag{8a}$$

$$= \left(\prod_{(i \to j) \in \widehat{\mathcal{S}}} w'_{i \to j} \right) \left(\prod_{(i \to j) \in \overline{\mathcal{S}}} w'_{i \to j} \right)$$

$$= p(\widehat{\mathcal{S}})\, p(\overline{\mathcal{S}}) \tag{8b}$$

■

Pseudocode for Wilson (1996)'s algorithm is given in Fig. 2. Examining the pseudocode, one can see it is possible to infinitely encounter cycles. Fortunately, Wilson (1996) proves that this is not the case and that the algorithm has a probabilistic bound of $\mathcal{O}(H)$ where H is the mean hitting time of the graph. The **mean hitting time** of a Markov chain is defined as

$$H \stackrel{\text{def}}{=} \sum_{i,j} \pi_i \pi_j h(i,j) \tag{9}$$

where π is the stationary distribution of the Markov chain, and $h(i,j)$ is the expected number of steps to reach node j starting at node i. Wilson (1996) and Broder (1989) demonstrate that the mean hitting time for directed graphs is usually small (sometimes as low as linear in N). We will compare the empirical runtime of the algorithm against an $\mathcal{O}(N^3)$ algorithm in §4.2.

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

$$p(t) \propto \prod_{(i \to j) \in t} w_{i \to j} \tag{10}$$

Furthermore, `wilson` *runs in $\mathcal{O}(H)$ time.*

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

$$p(t) = \prod_{(i \to j) \in t} w'_{i \to j} \tag{11}$$

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

Wilson (1996, Theorem 3) proves that line 9 of `wilson` executes $\mathcal{O}(H)$ 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 ρ for a graph G. Suppose we know we want the edge $(\rho \to j)$ 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 ρ removed and node j defined as the new root. By adding $(\rho \to j)$ to the newly sampled tree, we clearly have a dependency tree in G. We can sample the root edge $(\rho \to j)$ from the root weights $\boldsymbol{\rho}$ to sample an unbiased dependency tree from the distribution. The pseudocode for this algorithm is given as `wilson_rc` in Fig. 3

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

Figure 3: Modification of Wilson (1996)'s algorithm to sample dependency trees from a graph. The lines that differ to `wilson` are highlighted.

Theorem 2. *For any graph G,* `wilson_rc`(G) *samples a dependency tree $t \in \mathcal{D}$ with probability*

$$p(t) \propto \prod_{(i \to j) \in t} w_{i \to j} \tag{12}$$

Furthermore, `wilson_rc` *runs in $\mathcal{O}(H)$ in.*

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 ρ). By Theorem 1, this results in a tree $t' \in \mathcal{T}(G')$ and was sampled with probability proportional to $\prod_{(i \to j) \in t'} w_{i \to j}$. We sample the edge emanating from the root $(\rho \to j)$ with probability proportional to $w_{\rho \to j}$ and so $t = t' \cup \{\rho \to j\}$ is a dependency

tree sampled with probability

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

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

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 (Kirchhoff, 1847; Tutte, 1984), allows us to compute Z in $\mathcal{O}(N^3)$ by taking the determinant of the Laplacian matrix, $\mathbf{L} \in \mathbb{R}^{N \times N}$. We use Koo et al. (2007)'s adaptation of the MTT to dependency trees.[5]

Theorem 3 (Proposition 1, Koo et al. (2007))**.** *For any graph G, the normalization constant Z over the distribution of dependency trees $\mathcal{D}$ is given by $\mathrm{Z} = |\mathbf{L}|$ where*

$$\mathrm{L}_{ij} = \begin{cases} w_{\rho \to j} & \textbf{if } i = 1 \\ \sum_{i' \in \mathcal{N} \setminus \{\rho, i\}} w_{i' \to j} & \textbf{if } i = j \\ -w_{i \to j} & \textbf{otherwise} \end{cases} \tag{13}$$

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

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

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

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

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 $\log \mathrm{Z}$ in Theorem 3 which Koo et al. (2007) show to be[7]

$$p(i \to j) \tag{16}$$
$$= \begin{cases} w_{\rho \to j} \mathrm{B}_{1j} & \textbf{if } i = \rho \\ w_{i \to j} \left(\delta_{j \neq 1} \mathrm{B}_{jj} - \delta_{i \neq 1} \mathrm{B}_{ij}\right) & \textbf{otherwise} \end{cases}$$

where $\mathbf{B} = \mathbf{L}^{-\top}$ (the transpose of the inverse of $\mathbf{L}$) and $\delta_x = 1 \iff x$ is true, otherwise, $\delta_x = 0$. Therefore, after computing $\mathbf{B}$ once, we can compute each $p(i \to j)$ in $\mathcal{O}(1)$ time. Finding $\mathbf{B}$ requires us to take a matrix inverse, and so runs in $\mathcal{O}(N^3)$ 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 $G \succ t_{<n}$, which is defined as the largest graph such that $t \in \mathcal{T}(G \succ t_{<n}) \implies t_{<n} \subseteq t$. Consequently, if $(i \to j) \in t_{<n}$, then $G \succ t_{<n}$ 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 $G \succ e$ and repeat. This algorithm will have to recompute $\mathbf{B}$ $\mathcal{O}(N)$ times and so will have a runtime of $\mathcal{O}(N^4)$. 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 $\mathbf{B}$ to contain the transpose of the Laplacian inverse of the conditioned graph. We

[5] 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.

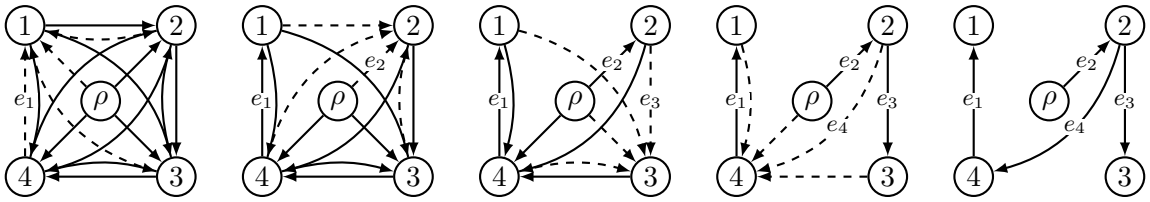
[6] The Laplacian matrix for spanning trees is given by

$$\mathrm{L}_{ij} = \begin{cases} \sum_{i' \in \mathcal{N} \setminus \{i\}} w_{i' \to j} & \textbf{if } i = j \\ -w_{i \to j} & \textbf{otherwise} \end{cases}$$

[7] For spanning trees, the marginal probability can be similarly derived as $p(i \to j) = w_{i \to j} \left(\mathrm{B}_{jj} - \delta_{i \neq \rho} \mathrm{B}_{ij}\right)$ where $\mathbf{B}$ is the transpose of the inverse of the Laplacian matrix in footnote 6.

[8] This runtime is also true by automatic differentiation (Griewank and Walther, 2008) as finding Z takes $\mathcal{O}(N^3)$ 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.



(a) G (b) $G \succ t_{<2}$ (c) $G \succ t_{<3}$ (d) $G \succ t_{<4}$ (e) $G \succ t$

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 ①, 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 ③ 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 $\mathcal{T}(G)$. Note $\mathcal{T}(G \succ t) = \{t\}$.

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

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

$$\mathrm{L}^{\mathrm{e}}{}_{[:,j]} = \begin{cases} w_{\rho \to j} \overrightarrow{\mathbf{1}}_1 & \textbf{if } i = \rho \\ w_{i \to j}(\delta_{j \neq 1} \overrightarrow{\mathbf{1}}_j - \delta_{i \neq 1} \overrightarrow{\mathbf{1}}_i) & \textbf{otherwise} \end{cases} \tag{17}$$

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

Proof. Consider the column $\mathrm{L}^{\mathrm{e}}{}_{[:,k]}$.

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

Case $i = \rho$: Then element $\mathrm{L}^{\mathrm{e}}{}_{1j} = w_{\rho \to j}$ 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 $\mathrm{L}^{\mathrm{e}}{}_{[:,j]} = w_{\rho \to j} \overrightarrow{\mathbf{1}}_1$.

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

Case $k \neq j$: Then all incoming edges to node k are still in $G \succ e$ and so $\mathrm{L}^{\mathrm{e}}{}_{[:,k]} = \mathrm{L}_{[:,k]}$.

■

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

$$(\mathbf{A} + \mathbf{u}\mathbf{v}^\top)^{-1} = \mathbf{A}^{-1} - \mathbf{T} \tag{18}$$

where

$$\mathbf{T} = \frac{\mathbf{A}^{-1}\mathbf{u}\,\mathbf{v}^\top\mathbf{A}^{-1}}{1 + \mathbf{v}^\top\mathbf{A}^{-1}\mathbf{u}} \tag{19}$$

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

$$\mathbf{T}^\top = \left(\frac{\mathbf{L}^{-1}\mathbf{u}\,\overrightarrow{\mathbf{1}}_j^\top\mathbf{L}^{-1}}{1 + \overrightarrow{\mathbf{1}}_j^\top\mathbf{L}^{-1}\mathbf{u}}\right)^\top \tag{20a}$$

$$= \left(\frac{\mathbf{L}^{-1}\mathbf{u}\,\mathbf{L}^{-1}_{[j,:]}}{1 + \mathbf{L}^{-1}_{[j,:]}\mathbf{u}}\right)^\top \tag{20b}$$

$$= \frac{\mathbf{L}^{-\top}_{[j,:]}\mathbf{u}^\top\mathbf{L}^{-\top}}{1 + \mathbf{u}^\top\mathbf{L}^{-\top}_{[j,:]}} \tag{20c}$$

$$= \frac{\mathrm{B}_{[j,:]}\mathbf{u}^\top\mathbf{B}}{1 + \mathbf{u}^\top\mathrm{B}_{[j,:]}} \tag{20d}$$

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

[10] The Sherman–Morrison formula can be computed in $\mathcal{O}(N^2)$ due to the associativity of matrix multiplication.

Global Variables: $\mathbf{L}$, $\mathbf{B}$

```
1.  def colbourn(G):
2.      ▷ Sample a dependency tree from a graph G;
        requires O(N^3) time, O(N^2) space.
3.      L ← Laplacian(G)
4.      B ← L^{-⊤}                                   ▷ O(N^3)
5.      t ← []
6.      for j ∈ N ∖ {ρ} :
7.          e ← sample_edge(j)
8.          t.append(e)
9.          condition(e)
10.     return t
11. def Laplacian(G):
12.     ▷ Construct the Laplacian of Koo et al. (2007)
        for dependency trees as in (13); requires
        O(N^2) time, O(N^2) space.
13.     L ← 0
14.     for j ∈ N ∖ {ρ} :
15.         for i ∈ N ∖ {ρ, j} :
16.             L_ij ← −w_{i→j}
17.             L_jj += w_{i→j}
18.         L_1j ← w_{ρ→j}
19.         ▷ For spanning trees, we can construct the Lapla-
            cian in footnote 6 by replacing line 18 with
            L_jj += w_{ρ→j}.
20.     return L
21. def sample_edge(j):
22.     ▷ Sample an incoming edge to j using global
        variable B as in (16); requires O(N) time,
        O(N) space.
23.     m ← 0
24.     m_ρ += w_{ρ→j} B_1j
25.     for i ∈ N ∖ {ρ, j} :
26.         m_i += w_{i→j} (δ_{j≠1} B_jj − δ_{i≠1} B_ij)
27.     ▷ For spanning trees, we can construct the
        marginals in footnote 7 by replacing line 24
        with m_ρ += w_{ρ→j} B_jj and line 26 with
        m_i += w_{i→j} (B_jj − B_ij).
28.     return sample from m
29. def condition(e):
30.     ▷ Condition the Laplacian and the transpose of
        its inverse to always include e in any tree; re-
        quires O(N^2) time, O(N^2) space.
31.     Let e = (i → j)
32.     if i = ρ :
33.         u ← w_{ρ→j} 1⃗_1 − L_{[:,j]}
34.     else:
35.         u ← w_{i→j} (δ_{j≠1} 1⃗_j − δ_{i≠1} 1⃗_i) − L_{[:,j]}
36.     L_{[:,j]} += u
37.     B -= (B_{[j,:]} u^⊤ B) / (1 + u^⊤ B_{[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.

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

$$p(t) = \frac{1}{\mathrm{Z}} \prod_{(i \to j) \in t} w_{i \to j} \tag{21}$$

Furthermore, `colbourn` *runs in $\mathcal{O}(N^3)$ 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`, $\mathbf{B}$ 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 $t_{<1} = \varnothing$. $\mathbf{B}$ 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 $(n-1)^{\text{th}}$ call to `sample_edge`, $\mathbf{B}$ 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 $\mathbf{B}$ to be the transpose of the Laplacian inverse of $G \succ t_{<n}$. Therefore, at the n^{th} call to `sample_edge`, $\mathbf{B}$ 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 $\mathcal{O}(N)$ times and each call to `condition` takes $\mathcal{O}(N^2)$ time. These runtimes are easily observed from the pseudocode. Therefore, `colbourn` has a runtime of $\mathcal{O}(N^3)$. ■

4.2 Runtime Experiment

We conduct a brief runtime experiment for `colbourn` and `wilson_rc` (see §3) whose runtimes are $\mathcal{O}(N^3)$ and $\mathcal{O}(H)$ respectively. We artificially generate random complete graphs of increasing size and measure the average sample time of each algorithm.[11] 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 $\mathcal{O}(N^3)$.[12]

[11] The experiment was conducted using an Intel(R) Core(TM) i7-7500U processor with 16GB RAM.

[12] We would expect the slope to be ≈ 3 to match the complexity bound.

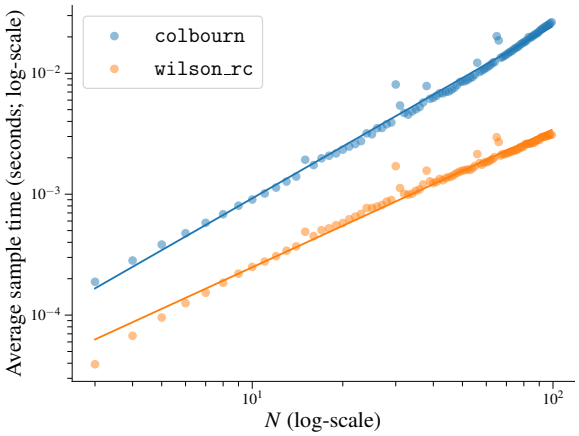


Figure 6: Runtime experiment for sampling using `wilson_rc` and `colbourn`. For each graph size, we randomly generated 100 graphs and took 20 samples from each graph. The best fit lines for `colbourn` and `wilson_rc` have slopes of 1.42 and 1.14 respectively.

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

$$p(i \to j \mid D) = \frac{1}{\mathrm{Z}_D} \sum_{t \in \mathcal{D}_{ij} \setminus D} w(t) \tag{22}$$

where $\mathcal{D}_{ij}$ is the set of all dependency trees that contain edge $(i \to j)$ and

$$\mathrm{Z}_D \overset{\text{def}}{=} \sum_{t \in \mathcal{D} \setminus D} w(t) = \mathrm{Z} - \sum_{t \in D} w(t) \tag{23}$$

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

$$p(i \to j \mid D) = \frac{1}{\mathrm{Z}_D} \left[\mathrm{Z} \cdot p(i \to j) - \sum_{t \in D_{ij}} w(t) \right] \tag{24}$$

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

Proof.

$$\begin{aligned}
& p(i \to j \mid D) && \text{(25a)} \\
&= \frac{1}{\mathrm{Z}_D} \sum_{t \in \mathcal{D}_{ij} \setminus D} w(t) && \text{(25b)} \\
&= \frac{1}{\mathrm{Z}_D} \left[\sum_{t \in \mathcal{D}_{ij}} w(t) - \sum_{t \in D_{ij}} w(t) \right] && \text{(25c)} \\
&= \frac{1}{\mathrm{Z}_D} \left[\mathrm{Z}\, p(i \to j) - \sum_{t \in D_{ij}} w(t) \right] && \text{(25d)}
\end{aligned}$$

■

Lemma 4 then gives a new formula to use for sampling edges by re-weighting the probability of an edge.[13] We can then compute the marginal distribution for an incoming edges 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 $\mathbf{B}$. Fortunately, this can be done in $\mathcal{O}(N)$ using the matrix determinant Lemma, which states that for any matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ and column vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^N$

$$|\mathbf{A} + \mathbf{u}\mathbf{v}^\top| = |\mathbf{A}|(1 + \mathbf{v}^\top \mathbf{A}^{-1} \mathbf{u}) \tag{26}$$

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 its self, `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, $\mathbf{L}$, and $\mathbf{B}$ 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*

$$p(t \mid D) = \frac{\delta_{t \notin D}}{\mathrm{Z}_D} \prod_{(i \to j) \in t} w_{i \to j} \tag{27}$$

[13] Re-weighting has been recently used by Stanojević and Cohen (2021) to speed-up algorithms for single-root tree decoding algorithms.

Global Variables: $\mathbf{L}$, $\mathbf{B}$, D, Z, and Z_D

def swor(G, K):
  ▷ *Sample K dependency trees without replacement from a graph G; requires $\mathcal{O}(KN^3 + K^2N)$ time, $\mathcal{O}(N^2 + KN)$ space.*
  $\mathbf{L}' \leftarrow$ Laplacian(G)
  $\mathrm{Z}' \leftarrow |\mathbf{L}'|$; $\mathbf{B}' \leftarrow \mathbf{L}'^{-\top}$ ▷ $\mathcal{O}(N^3)$
  $D' \leftarrow [\,]$; $\mathrm{Z}'_D \leftarrow \mathrm{Z}'$
  for $k \in \{1, \ldots, K\}$:
    $D \leftarrow D'$; $\mathrm{Z} \leftarrow \mathrm{Z}'$
    $\mathrm{Z}_D \leftarrow \mathrm{Z}'_D$
    $\mathbf{L} \leftarrow \mathbf{L}'$; $\mathbf{B} \leftarrow \mathbf{B}'$; $t \leftarrow [\,]$
    for $j \in \mathcal{N} \setminus \{\rho\}$:
      $e \leftarrow$ sample_edge$'(j)$
      t.append(e)
      condition$'(e)$
    D'.append(t); Z'_D -= $w(t)$
  return D'
def sample_edge$'(j)$:
  ▷ *Sample an incoming edge to j using global variables $\mathbf{B}$, Z, and Z_D as in (24); requires $\mathcal{O}(N)$ time, $\mathcal{O}(N)$ space.*
  $\mathbf{m} \leftarrow \mathbf{0}$
  m_ρ += $\mathrm{Z}\, w_{\rho \to j} \mathrm{B}_{1j} - \sum_{t \in T_{\rho j}} w(t)$
  for $i \in \mathcal{N} \setminus \{\rho, j\}$: ▷ $\mathcal{O}(N)$
    m_i += $\mathrm{Z}\, w_{i \to j} (\delta_{j \neq 1} \mathrm{B}_{jj} - \delta_{i \neq 1} \mathrm{B}_{ij})$
    m_i -= $\sum_{t \in D_{ij}} w(t)$
  return sample from $\frac{1}{\mathrm{Z}_D}\mathbf{m}$
def condition$'(e)$:
  ▷ *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 $\mathcal{O}(N^2 + K)$ time, $\mathcal{O}(N^2)$ space.*
  Let $e = (i \to j)$
  if $i = \rho$:
    $\mathbf{u} \leftarrow w_{\rho \to j} \overrightarrow{\mathbf{1}_1} - \mathrm{L}_{[:,j]}$
  else:
    $\mathbf{u} \leftarrow w_{i \to j} (\delta_{j \neq 1} \overrightarrow{\mathbf{1}_j} - \delta_{i \neq 1} \overrightarrow{\mathbf{1}_i}) - \mathrm{L}_{[:,j]}$
  $\mathrm{L}_{[:,j]}$ += $\mathbf{u}$
  Z *= $1 + \mathbf{u}^\top \mathrm{B}_{[j,:]}$
  $\mathbf{B}$ -= $(\mathrm{B}_{[j,:]} \mathbf{u}^\top \mathbf{B})$ / $(1 + \mathbf{u}^\top \mathrm{B}_{[j,:]})$
  $D \leftarrow D_{ij}$
  $\mathrm{Z}_D \leftarrow \mathrm{Z} - \sum_{t \in D} w(t)$

Figure 7: Algorithm for sampling dependency trees without replacement.

where D is the set of trees sampled prior to t. Furthermore, swor *runs in $\mathcal{O}(KN^3 + K^2N)$ 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 $D' = \varnothing$ and $\mathrm{Z}'_D = \mathrm{Z}'$. 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 $\mathrm{Z}'_D = \mathrm{Z}' - \sum_{t \in D'} w(t)$ 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 $\mathcal{O}(N^3)$ to find Z' and $\mathbf{B}'$. Then for each of the K sampled trees, we have N iterations of the main loop. Each call to sample_edge$'$ takes $\mathcal{O}(N+K)$ times and each call to condition$'$ takes $\mathcal{O}(N^2 + K)$ time. These runtimes are easily observed from the pseudocode. Therefore, swor has a runtime of $\mathcal{O}(KN^3 + K^2N)$. ■

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 $\mathcal{O}(H)$ algorithm was more rapid, Colbourn et al. (1996)'s $\mathcal{O}(N^3)$ 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 Hajičová, Jan Hajič, Pavlína Jínová, Václava Kettnerová, Veronika Kolářová, Marie Mikulová, Jiří Mírovský, 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*.

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, Hissar, 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.

Tetsuji Nakagawa. 2007. Multilingual dependency parsing using global features. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 952–956, Prague, Czech Republic. Association for Computational Linguistics.

Joakim Nivre, Mitchell Abrams, Željko Agić, Lars Ahrenberg, Lene Antonsen, Katya Aplonova, Maria Jesus Aranzabe, Gashaw Arutie, Masayuki Asahara, Luma Ateyah, Mohammed Attia, Aitziber Atutxa, Liesbeth Augustinus, Elena Badmaeva, Miguel Ballesteros, Esha Banerjee, Sebastian Bank, Verginica Barbu Mititelu, Victoria Basmov, John Bauer, Sandra Bellato, Kepa Bengoetxea, Yevgeni Berzak, Irshad Ahmad Bhat, Riyaz Ahmad Bhat, Erica Biagetti, Eckhard Bick, Rogier Blokland, Victoria Bobicev, Carl Börstell, Cristina Bosco, Gosse Bouma, Sam Bowman, Adriane Boyd, Aljoscha Burchardt, Marie Candito, Bernard Caron, Gauthier Caron, Gülşen Cebiroğlu Eryiğit, Flavio Massimiliano Cecchini, Giuseppe G. A. Celano, Slavomír Čéplö, Savas Cetin, Fabricio Chalub, Jinho Choi, Yongseok Cho, Jayeol Chun, Silvie Cinková, Aurélie Collomb, Çağrı Çöltekin, Miriam Connor, Marine Courtin, Elizabeth Davidson, Marie-Catherine de Marneffe, Valeria de Paiva, Arantza Diaz de Ilarraza, Carly Dickerson, Peter Dirix, Kaja Dobrovoljc, Timothy Dozat, Kira Droganova, Puneet Dwivedi, Marhaba Eli, Ali Elkahky, Binyam Ephrem, Tomaž Erjavec, Aline Etienne, Richárd Farkas, Hector Fernandez Alcalde, Jennifer Foster, Cláudia Freitas, Katarína Gajdošová, Daniel Galbraith, Marcos Garcia, Moa Gärdenfors, Sebastian Garza, Kim Gerdes, Filip Ginter, Iakes Goenaga, Koldo Gojenola, Memduh Gökırmak, Yoav Goldberg, Xavier Gómez Guinovart, Berta Gonzáles Saavedra, Matias Grioni, Normunds Grūzītis, Bruno Guillaume, Céline Guillot-Barbance, Nizar Habash, Jan Hajič, Jan Hajič jr., Linh Hà Mỹ, Na-Rae Han, Kim Harris, Dag Haug, Barbora Hladká, Jaroslava Hlaváčová, Florinel Hociung, Petter Hohle, Jena Hwang, Radu Ion, Elena Irimia, Ọlájídé Ishola, Tomáš Jelínek, Anders Johannsen, Fredrik Jørgensen, Hüner Kaşıkara, Sylvain Kahane, Hiroshi Kanayama, Jenna Kanerva, Boris Katz, Tolga Kayadelen, Jessica Kenney, Václava Kettnerová, Jesse Kirchner, Kamil Kopacewicz, Natalia Kotsyba, Simon Krek, Sookyoung Kwak, Veronika Laippala, Lorenzo Lambertino, Lucia Lam, Tatiana Lando, Septina Dian Larasati, Alexei Lavrentiev, John Lee, Phuong Lê Hồng, Alessandro Lenci, Saran Lertpradit, Herman Leung, Cheuk Ying Li, Josie Li, Keying Li, KyungTae Lim, Nikola Ljubešić, Olga Loginova, Olga Lyashevskaya, Teresa Lynn, Vivien Macketanz, Aibek Makazhanov, Michael Mandl, Christopher Manning, Ruli Manurung, Cătălina Mărănduc, David Mareček, Katrin Marheinecke, Héctor Martínez Alonso, André Martins, Jan Mašek, Yuji Matsumoto, Ryan McDonald, Gustavo Mendonça, Niko Miekka, Margarita Misirpashayeva, Anna Missilä, Cătălin Mititelu, Yusuke Miyao, Simonetta Montemagni, Amir More, Laura Moreno Romero, Keiko Sophie Mori, Shinsuke Mori, Bjartur Mortensen, Bohdan Moskalevskyi, Kadri Muischnek, Yugo Murawaki, Kaili Müürisep, Pinkey Nainwani, Juan Ignacio Navarro Horñiacek, Anna Nedoluzhko, Gunta Nešpore-Bērzkalne, Luong Nguyễn Thị, Huyền Nguyễn Thị Minh, Vitaly Nikolaev, Rattima Nitisaroj, Hanna Nurmi, Stina Ojala, Adédayọ̀ Olúòkun, Mai Omura, Petya Osenova, Robert Östling, Lilja Øvrelid, Niko Partanen, Elena Pascual, Marco Passarotti, Agnieszka Patejuk, Guilherme Paulino-Passos, Siyao Peng, Cenel-Augusto Perez, Guy Perrier, Slav Petrov, Jussi Piitulainen, Emily Pitler, Barbara Plank, Thierry Poibeau, Martin Popel, Lauma Pretkalniņa, Sophie Prévost, Prokopis Prokopidis, Adam Przepiórkowski, Tiina Puolakainen, Sampo Pyysalo, Andriela Rääbis, Alexandre Rademaker, Loganathan Ramasamy, Taraka Rama, Carlos Ramisch, Vinit Ravishankar, Livy Real, Siva Reddy, Georg Rehm, Michael Rießler, Larissa Rinaldi, Laura Rituma, Luisa Rocha, Mykhailo Romanenko, Rudolf Rosa, Davide Rovati, Valentin Roșca, Olga Rudina, Jack Rueter, Shoval Sadde, Benoît Sagot, Shadi Saleh, Tanja Samardžić, Stephanie Samson, Manuela Sanguinetti, Baiba Saulīte, Yanin Sawanakunanon, Nathan Schneider, Sebastian Schuster, Djamé Seddah, Wolfgang Seeker, Mojgan Seraji, Mo Shen, Atsuko Shimada, Muh Shohibussirri, Dmitry Sichinava, Natalia Silveira, Maria Simi, Radu Simionescu, Katalin Simkó, Mária Šimková, Kiril Simov, Aaron Smith, Isabela Soares-Bastos, Carolyn Spadine, Antonio Stella, Milan Straka, Jana Strnadová, Alane Suhr, Umut Sulubacak, Zsolt Szántó, Dima Taji, Yuta Takahashi, Takaaki Tanaka, Isabelle Tellier, Trond Trosterud, Anna Trukhina, Reut Tsarfaty, Francis Tyers, Sumire Uematsu, Zdeňka Urešová, Larraitz Uria, Hans Uszkoreit, Sowmya Vajjala, Daniel van Niekerk, Gertjan van Noord, Viktor Varga, Eric Villemonte de la Clergerie, Veronika Vincze, Lars Wallin, Jing Xian Wang, Jonathan North Washington, Seyi Williams, Mats Wirén, Tsegay Woldemariam, Tak-sum Wong, Chunxiao Yan, Marat M. Yavrumyan, Zhuoran Yu, Zdeněk Žabokrtský, Amir Zeldes, Daniel Zeman, Manying Zhang, and Hanzhi Zhu. 2018. Universal dependencies 2.3. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

Wenzhe Pei, Tao Ge, and Baobao Chang. 2015. An effective neural network model for graph-based dependency parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 313–322, Beijing, China. Association for Computational Linguistics.

Jack Sherman and Winifred J. Morrison. 1950. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *The Annals of Mathematical Statistics*, 21(1):124–127.

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.