Unbiased and Efficient Sampling of Dependency Trees

Miloš Stanojević
DeepMind
stanojevic@deepmind.com

Abstract

Most computational models of dependency syntax consist of distributions over spanning trees. However, the majority of dependency treebanks require that every valid dependency tree has a single edge coming out of the ROOT node, a constraint that is not part of the definition of spanning trees. For this reason all standard inference algorithms for spanning trees are sub-optimal for inference over dependency trees.

Zmigrod et al. (2021b) proposed algorithms for sampling with and without replacement from the dependency tree distribution that incorporate the single-root constraint. In this paper we show that their fastest algorithm for sampling with replacement, Wilson-RC, is in fact producing biased samples and we provide two alternatives that are unbiased. Additionally, we propose two algorithms (one incremental, one parallel) that reduce the asymptotic runtime of algorithm for sampling k trees without replacement to $\mathcal{O}(kn^3)$. These algorithms are both asymptotically and practically more efficient.

1 Introduction

Dependency trees are one of the core structures for representing syntactic relations between words (Kübler et al., 2009; Osborne, 2019). There are multiple ways that these trees can be formally defined which has computational, linguistic and modeling consequences. Until recently, the most common formalisation of dependency structures used in computational linguistics was a *spanning tree* formalisation. A spanning tree is a directed graph that contains $n+1$ nodes (n words and one artificial ROOT token), n directed edges (each word has one incoming edge except for the ROOT node that has only outgoing edges) and there are no cycles. This simple formalisation allows for the usage of efficient algorithms for finding optimal spanning trees (Tarjan, 1977; McDonald et al., 2005).

Unfortunately, spanning trees are too loose of a formalisation to represent the constraints linguists would expect from dependency trees. One of the main linguistic constraints is that there is only one word that represents the main head of the sentence. This is a standard in Universal Dependencies treebanks (Nivre et al., 2020) where the constraint is incorporated by stating that every valid dependency tree should have exactly one edge coming out of the artificial ROOT node (UD, 2022).[1] In the rest of the article we will use the term *spanning tree* to refer to the unconstrained trees and term *dependency tree* to refer to the trees that have a single-root constraint as in Universal Dependencies.

The distinction between spanning and dependency trees at first may appear unimportant because all dependency trees are a subset of the set of spanning trees and maybe a strong neural model would learn distributions that ignore spanning trees which are not valid dependency trees. However, Zmigrod et al. (2020) have shown that that is not the case—state-of-the-art trained parsers often predict an invalid dependency tree as the mode and the situation is even worse with k-best trees (Zmigrod et al., 2021c). This motivates the creation of specialized algorithms that do inference *only* over valid dependency structures.

The most important algorithms needed for inference are: (1) finding the most probable tree, (2) finding marginals over edges, and (3) generating tree samples. Finding the most probable dependency tree is a solved problem since the algorithm by Stanojević and Cohen (2021) is optimal for that task. Marginals can be computed efficiently using the adaptation of Matrix-Tree Theorem to single-root dependency trees by Koo et al. (2007).

In this paper we focus on the last of these problems—generation of samples of dependency trees. The sampling of dependency trees is often used as a key component of unsupervised grammar induction (Mareček and Žabokrtský, 2011), semi-

[1] There are few treebanks that make exemption to this rule, e.g. Prague Dependency Treebank (Bejček et al., 2013).

Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, pages 1691 - 1706
December 7-11, 2022 ©2022 Association for Computational Linguistics

supervised training of parsers (Corro and Titov, 2019), or for enabling the approximate decoding of higher-order models (Zhang et al., 2014).

The only previous work that deals specifically with sampling from distributions of constrained dependency trees is by Zmigrod et al. (2021b). They adapt algorithms for sampling spanning trees to include the single-root constraint and provide algorithms for sampling with and without replacement.

Our main contributions are in: (i) showing that the most efficient sampling algorithm by Zmigrod et al. is in fact producing biased samples, (ii) proposing two new algorithms that address this issue, and (iii) providing two more algorithms (one incremental and one parallel) that improve the worst-case complexity of sampling without replacement (SWOR). All these algorithms are shown to be both asymptotically and practically efficient.

2 Distributions over Dependency Trees

Here we define more formally the distributions over dependency and spanning trees. We will consider only unlabelled trees, but extending this to the labelled case is trivial.

Dependency graph G of a sentence with n words is an unlabelled directed weighted rooted graph that consists of:

$n+1$ **nodes** where each word of the sentence is a node, and in addition to those n nodes there is a special ROOT node,

n^2 **edges** out of which $n(n-1)$ are directed edges between every pair of word nodes (excluding self-loops) and n edges that go from the special ROOT node to each of the word nodes,

n^2 **weights**, sometimes also called potentials, are non-negative real numbers associated with each of the edges using a function $\phi(\cdot)$.[2]

We can represent any graph of this type with an adjacency matrix W of shape $(n+1, n+1)$ where all elements are non-negative weights with a constraint that diagonal entries are 0 (self-loops are not allowed) and first column entries are 0 (no edges can enter the ROOT node which is at position 0 by convention).

Spanning tree is any subgraph of G for which it holds that (i) every word node has exactly one incoming edge and (ii) there are no cycles.

Dependency tree is spanning tree that in addition to the above mentioned properties also contains the constraint that there can be only one edge coming out of the special ROOT node.

We will refer to the set of all spanning trees as T and set of all dependency trees as D. The following definitions are presented for dependency trees but their analogs for spanning trees are trivial to define by just replacing every occurrence of D with T.

Weight of a dependency tree t is a product of the weights of all its edges:

$$\phi(t) = \prod_{e \in t} \phi(e)$$

Probability of a dependency tree t is a normalized weight of a tree t with respect to all other dependency trees:

$$p(t) = \frac{\phi(t)}{\sum_{t' \in D} \phi(t')} = \frac{\phi(t)}{Z_D}$$

Here Z_D stands for the partition function of dependency trees that must be positive i.e. $Z_D > 0$.

Marginal probability of an edge e in the distribution over dependency trees is a sum of probabilities of all dependency trees D_e that contain edge e:

$$p(e) = \sum_{t \in D_e} p(t) = \frac{\sum_{t \in D_e} \phi(t)}{\sum_{t' \in D} \phi(t')}$$

Sampler of dependency trees is **unbiased** *iff* it independently draws random dependency trees with the probability proportional to their weight, i.e. probability of sampling any tree t is $p(t)$.

3 Previous Work on Sampling Spanning Trees

Here we present the main two algorithms for sampling from the distributions of unconstrained spanning trees. Both of these algorithms have different advantages. The first one, by Wilson (1996), is oftentimes faster in practice and we will use it to design a fast sampler of dependency trees in Section 4. The second algorithm is a type of ancestral sampling by Colbourn et al. (1996) and has a convenient form for extension to sampling without replacement, that will be used in Section 5.

3.1 WILSON Sampler

Wilson's algorithm for sampling unconstrained directed spanning trees is in practice the fastest algorithm for that task. It runs in $\mathcal{O}(h)$ expected computational complexity where h is the *mean hitting*

[2] The type of models in which a weight can be associated only with a single edge are called *first-order models*. For a comment on higher-order models see the limitations section.

Algorithm 1 WILSON's sampling algorithm.

```
 1: $t \leftarrow \emptyset$                ▷ Sampled tree edges where
                                    ▷ $t[i] = j$ stands for edge $j \rightarrow i$
 2: visited $\leftarrow$ {ROOT}
 3: for $i \in [1 \dots n]$ do              ▷ Loop over words
 4:     $u \leftarrow i$
 5:     while $u \notin$ visited do
 6:         Sample node $v$ with weight $\phi(v \rightarrow u)$
 7:         $t[u] \leftarrow v$
 8:         $u \leftarrow v$
 9:     $u \leftarrow i$
10:     while $u \notin$ visited do
11:         visited.add($u$)
12:         $u \leftarrow t[u]$
13: return $t$
```

time. Mean hitting time depends on the weights, but it can often be as small as $\mathcal{O}(n)$ (Wilson, 1996).

We use Wilson's algorithm as a black box, so understanding the details of its workings is not important, but we provide it in Algorithm 1 for completeness. Each iteration of the for loop starts from some word (i.e. non-ROOT node) that is not visited and walks a random chain until it reaches some node that was visited in the previous iterations of the for loop. This chain will be attached to the tree that is being sampled. The random walk may contain cycles which are implicitly deleted by overriding parent pointers in lines 7 and 8. The second while loop just marks the nodes in the random walk as visited. In the end of this function all the word nodes will be visited, have one incoming arc, and there will be no cycles.

3.2 COLBOURN Sampler

Colbourn et al. (1996) constructed a sampling algorithm that is very different in nature to the Wilson in that it is an ancestral sampling method that has a fixed runtime complexity that depends only on the size of the graph, but not on the weights. If we have a sentence with words $w_1 w_2 \dots w_n$ we can represent any dependency tree over it by having a sequence of integers $t_1 t_2 \dots t_n$ where the value of t_i defines an edge $w_{t_i} \rightarrow w_i$. Colbourn's algorithm samples each of these integers one by one. After sampling each integer (i.e. edge) we need to constrain the weights of the graph so that the probability of sampling future edges is conditioned on the previously sampled edges. The underspecified version of this algorithm is shown in Algorithm 2. A naïve implementation of this algorithm could be done in $\mathcal{O}(n^4)$ because we can recompute marginals n times and recomputing marginals each time from scratch takes $\mathcal{O}(n^3)$. This cubic complexity bottleneck comes from the matrix-inverse that is used by Matrix-Tree Theorem to compute marginals (Koo et al., 2007). The main contribution of Colbourn et al. is a clever way of reducing the complexity of this bottleneck down to $\mathcal{O}(n^2)$ by keeping around the running version of the matrix-inverse and then updating it for each new changed column using Sherman-Morrison formula for rank-one update of matrix-inverse (Sherman and Morrison, 1950). The technical details of this formula are not important for this presentation, but for completeness can be found in Appendix B. With this trick the total worst-case complexity is $\mathcal{O}(n^3)$.

Algorithm 2 COLBOURN's sampling algorithm.

```
$t \leftarrow \emptyset$                ▷ Sampled tree edges where
                                ▷ $t[i] = j$ stands for edge $j \rightarrow i$
$M \leftarrow$ marginals($W$)
for $i \in [1 \dots n]$ do              ▷ Loop over words
    Sample node $v$ with weight $M(v \rightarrow u)$
    $t[i] \leftarrow v$
    $M \leftarrow$ constrain($M, v, i$)
return $t$
```

Zmigrod et al. (2021b) showed that even though Colbourn's algorithm was originally designed for unconstrained spanning trees it can easily be extended to sampling dependency trees by just using the version of Matrix-Tree Theorem by Koo et al. (2007). Appendix A contains details on MTT.

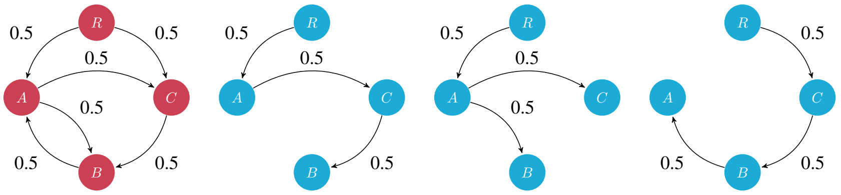
4 Extensions of Wilson's Algorithm to Valid Dependency Trees

In this section we will describe novel contributions with regards to sampling dependency trees with replacement. In §4.1 we show that the previous fastest sampling algorithm is in fact biased. In §4.2 and §4.3 we present two new unbiased algorithms.

4.1 WILSONRC Extension by Zmigrod et al. is Biased

Zmigrod et al. (2021b) propose a very simple extension of the Wilson's algorithm to the single-root dependency trees that works in following steps:

1. sample an edge e from the set of all edges that are coming out of ROOT by using their original weights $\phi(e)$,



(a) Example graph (b) Tree 1 (c) Tree 2 (d) Tree 3

Figure 1: Example graph and its single-root trees where Zmigrod et al. algorithm fails to provide unbiased samples.

2. construct graph G' by deleting all edges coming out of ROOT and set the target of the edge e to be the new root,
3. apply original Wilson's algorithm to graph G'.

This extension is very simple, but unfortunately, as we show here, it is biased.[3] We demonstrate that that Zmigrod et al. (2021b) algorithm is biased with a simple example graph in Figure 1a. For a formal proof see Appendix E.

This graph has only three constrained dependency trees that are shown in Figures 1b-1d. They all have same weights so any unbiased sampling algorithm should return them with the same probability. However, the step 1 of Zmigrod et al. will half of the time sample ROOT edge $R \rightarrow A$ for the first two trees and half of the time the ROOT edge $R \rightarrow C$ for the last tree and therefore it is biased to sample the last tree over the first two. In the following sections we propose unbiased alternatives.

4.2 Improvement 1: WilsonMarginal

Intuitively, by looking at the example in Figure 1 the first step that samples the single root edge should reflect our intuition that edge $R \rightarrow A$ should be sampled with two times higher probability than edge $R \rightarrow C$ because it has two times more of the (weighted) trees. This can be accomplished by sampling from the *marginal* of each root edge. Marginal for the edges $R \rightarrow A$ and $R \rightarrow C$ in this graph will be $\frac{2}{3}$ and $\frac{1}{3}$ respectively.

WILSONMARGINAL (shown in Algorithm 3) does exactly that. Step 1 is computing marginals which can be done using Koo et al. (2007) version of the Matrix-Tree Theorem in $\mathcal{O}(n^3)$. Steps 2 and 3 run in $\mathcal{O}(n)$ since that is the maximal number of edges coming out of ROOT. Step 4 is the standard Wilson's algorithm that runs in $\mathcal{O}(h)$. Therefore getting a single sample with this algorithm can be done in $\mathcal{O}(n^3 + h)$. If we want to draw multiple samples from the same graph we can reuse the computation of the marginals. Drawing k samples from the same graph can be done in $\mathcal{O}(n^3 + kh)$.

Algorithm 3 WILSONMARGINAL

```
1: $M \leftarrow$ MARGINALS$(G)$
2: $e \leftarrow$ SAMPLE(ROOT edges by $M$)
3: $G' \leftarrow G \setminus \{$all ROOT edges$\} \cup \{e\}$
4: return WILSON$(G')$
```

Theorem 1. WILSONMARGINAL *is an unbiased sampling algorithm i.e. it samples each dependency tree $t \in D$ with probability:*

$$p(t) = \frac{\phi(t)}{\sum_{t' \in D} \phi(t')}$$

Proof. The root edge of the sampled tree t will be sampled with probability $\frac{\sum_{t' \in D_e} \phi(t')}{\sum_{t' \in D} \phi(t')}$ where D_e is the subset of all dependency trees that contain edge e. The rest of the tree will be sampled from graph G' with probability $\frac{\phi(t)}{\sum_{t' \in D_e} \phi(t')}$. The probability of sampling tree t using WILSONMARGINAL is $\frac{\sum_{t' \in D_e} \phi(t')}{\sum_{t' \in D} \phi(t')} \frac{\phi(t)}{\sum_{t' \in D_e} \phi(t')} = \frac{\phi(t)}{\sum_{t' \in D} \phi(t')} = p(t)$ ■

4.3 Improvement 2: WilsonReject

The complexity of WILSONMARGINAL is worse than the original unconstrained Wilson's algorithm because of the cubic term. This term comes from the need to invert the Laplacian matrix during the computation of the marginals. This operation is optimized in many software packages (Abdelfattah et al., 2017) and in practice it does not take a long time to compute.Still, it would be good to avoid computing matrix-inverse unless necessary because matrix-inverse can be numerical unstable if the matrix is ill-conditioned (Croz and Higham, 1992; Higham, 2002). Here we present an alternative

[3] Since the publication of our paper Zmigrod et al. (2021b) have modified their paper to acknowledge that WILSONRC is biased. Zmigrod (2022) PhD thesis is an excellent work containing many important algorithms for non-projective dependency distributions. It also includes re-implementation and additional benchmarks of our unbiased algorithms.

based on rejection sampling that on average works well for most graphs that are of interest.

Rejection sampling is used in cases when it is difficult to sample from the distribution of interest $p(t)$ but it is easy to sample from some related proposal distribution $q(t)$. The proposal distribution $q(t)$ needs to satisfy $cq(t) \geq \tilde{p}(t)$ for some constant $c > 0$ where $\tilde{p}(t)$ is the unnormalized target distribution. In other words, $cq(t)$ needs to form an upper envelope of $\tilde{p}(t)$. After sample is retrieved from $q(t)$ it is accepted as a sample of $p(t)$ with probability $\frac{\tilde{p}(t)}{cq(t)}$. See Murphy (2012, §23.3) for a good introduction to rejection sampling.

Barring the envelope condition, we have a complete freedom to choose the proposal distribution $q(\cdot)$ and constant c. Any choice that satisfies the envelope condition gives an unbiased sampler.

Here the target distribution from which we want to sample is the distribution over dependency trees $p(t)$. A convenient distribution to choose as a proposal $q(t)$ is the distribution over spanning trees in the same weighted graph. All valid dependency trees t will be present in the support of both $p(\cdot)$ and $q(\cdot)$ and their *unnormalized* weight will be the same since they came from the same weighted graph. If a tree is a spanning tree t' that has more than one root edge then it will be in the support of $q(\cdot)$ but not of $p(\cdot)$ so their unnormalized score for target distribution is $\tilde{p}(t') = 0$. More formally:

$$\tilde{p}(t) = \begin{cases} \tilde{q}(t), & \text{if } t \in D \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

In this setup the unnormalized $\tilde{q}(t)$ forms an envelope over unnormalized $\tilde{p}(t)$ because $\forall t. \tilde{q}(t) \geq \tilde{p}(t)$. Still, this is not sufficient to apply rejection sampling because we need an envelope with the *normalized* distribution $q(t)$. To get that we multiply the normalized proposal with constant $c = Z_T$ which is the partition function over the set of unconstrained spanning trees.

Theorem 2. WILSONREJECT *is an unbiased sampling procedure for dependency tree distributions.*

Proof. Given that we already have an unbiased way for generating proposal samples from $q(\cdot)$, i.e. the original Wilson algorithm that generates spanning trees, we only need to prove that the chosen proposal $q(\cdot)$ and constant c satisfy the envelope condition for rejection sampling $cq(t) \geq \tilde{p}(t)$:

$$cq(t) = Z_T \frac{\tilde{q}(t)}{Z_T} = \tilde{q}(t) \geq \tilde{p}(t)$$

Algorithm 4 WILSONREJECT

```
do $t \leftarrow$ WILSON($G$)
until $t$ has only one ROOT edge
return $t$
```

The last step follows from Equation 1. ■

When the rejection sampling algorithm gets a sample from $q(\cdot)$ it accepts it with probability:

$$\frac{\tilde{p}(t)}{cq(t)} = \frac{\tilde{p}(t)}{Z_T \frac{\tilde{q}(t)}{Z_T}} = \begin{cases} 1, & \text{if } t \in D \\ 0, & \text{otherwise} \end{cases}$$

This gives us a very simple implementation of the WILSONREJECT sampling algorithm presented in Algorithm 4. This algorithm samples spanning trees with the original Wilson's algorithm until it gets a valid dependency tree.

4.3.1 Expected number of runs for WilsonReject

The efficiency of WILSONREJECT depends on how many samples we need to draw from $q(t)$ until we get a valid dependency tree. We can model the process of getting a successful sample as a geometric distribution in which the probability of success is $\sum_{t \in D} q(t)$. The expected number of draws before success in a geometric distribution is $\frac{1}{\sum_{t \in D} q(t)} = \frac{Z_T}{\sum_{t \in D} \tilde{q}(t)} = \frac{Z_T}{Z_D}$, i.e. the expected number of attempts until success is the ratio of partitions for spanning and dependency trees.

To determine this number we need to know something about actual distribution over trees. In dependency parsing we usually care about two scenarios. The first one is the random weights setting that usually happens in the beginning of training of dependency parsers. The second one is the setting of already trained parsers. We answer the first one formally and second one empirically.

Random weights setting Here edge weights are sampled independently from the same distribution with mean μ. We are interested in finding expected number of runs under the distribution of graph weights $\mathbb{E}\left[\frac{Z_T}{Z_D}\right]$. Let w_{avg}^T and w_{avg}^D be the average weights of spanning and dependency trees in a given weighted graph. We can write the following:

$$\mathbb{E}\left[\frac{Z_T}{Z_D}\right] = \mathbb{E}\left[\frac{|T| w_{avg}^T}{|D| w_{avg}^D}\right] = \frac{|T|}{|D|} \mathbb{E}\left[\frac{w_{avg}^T}{w_{avg}^D}\right] \tag{2}$$

The number of spanning trees $|T|$ is given by Cayley's formula:[4]

$$|T| = C_n = (n+1)^{n-1} \tag{3}$$

The number of dependency trees $|D|$ is nC_{n-1} because we can pick any of the n words to be the root of the smaller sub-graph of $n-1$ words.

$$|D| = nC_{n-1} = n^{n-1} \tag{4}$$

Equations 2, 3 and 4 together give us an upper bound on the expected number of tries before WILSONREJECT finds a successful sample:

$$\mathbb{E}\left[\frac{Z_T}{Z_D}\right] = \frac{(n+1)^{n-1}}{n^{n-1}}\,\mathbb{E}\left[\frac{w^T_{avg}}{w^D_{avg}}\right] < e\,\mathbb{E}\left[\frac{w^T_{avg}}{w^D_{avg}}\right]$$

The expectation of averages is difficult to solve analytically, as usual with ratio distributions, because the weights of spanning and dependency trees in a given weighted graph are not independent. If we assume that average weights of spanning and dependency trees in a given weighted graph are approximately the same, which is a reasonable assumption since their expected weights across all possible weightings are the same, the solution to inequality above will be e.

To verify how reasonable is this assumption we have ran a large number of simulations on different graph sizes and weight distributions over edges. We found that $w^D_{avg} \approx w^T_{avg}$ to be very accurate in all settings we have tested. Appendix D contains the results of the simulations.

All this taken together means that on average WILSONREJECT will need to sample approximately less than three times before it finds a successful sample in a randomly weighted graph.

Trained weights setting In this setting we would expect that the model that was trained only on single-root trees to automatically put most of the probability mass on single-root trees, i.e. $Z_D \lessapprox Z_T$. To test this we used the model of Stanza parser (Qi et al., 2020) trained for English. We take English as the most extreme example since it has more training data than other languages and Zmigrod et al. (2020) show that that makes the model more likely to put the mode of the distribution on a valid dependency tree. We would expect that models for languages with less training data would be somewhere between the random weights setting and English trained weights setting. We applied the parser to the English portion of the News Commentary v16 corpus and computed Z_D and Z_T exactly using the algorithm of Koo et al. (2007). On almost all sentences the Stanza model puts more than 95% of probability mass on the single-root trees. This means that the average number of tries needed by WILSONREJECT to get a successful sample on a trained graph is ≈ 1.06.

The average behavior of WILSONREJECT on both random weights and trained weights setting is as good as the unconstrained original Wilson's sampler except for a small multiplicative constant.

5 Sampling Without Replacement (SWOR)

Sampling without replacement (SWOR) is a sampling procedure where after some instance is sampled it cannot be sampled again. That is useful in cases of low-entropy distributions where standard sampling is likely to provide many repetitions of the same set of samples and be inefficient in estimating the target value. Samples are not independently drawn and therefore require specialized estimators (Vieira, 2017; Kool et al., 2020b).

Zmigrod et al. (2021b) propose a SWOR algorithm that is based on modification of Colbourn's ancestral sampling algorithm. They follow the general pattern for constructing SWOR algorithms by Shi et al. (2020) where after one tree is sampled, its probability mass is removed from the distribution so it cannot be sampled again. Zmigrod et al. maintain an unstructured list of sampled trees that is always queried whenever a new sample is drawn. This makes the algorithm have quadratic complexity in terms of the number of drawn samples. Concretely, for a sentence of length n the SWOR algorithm of Zmigrod et al. draws k samples in $\mathcal{O}(kn^3 + k^2n)$. This is inefficient if we want to draw a large number of samples.

We provide two solutions that reduce complexity to $\mathcal{O}(kn^3)$ which is linearly dependent on the number of samples k. The main idea for both algorithms is to represent distributions over dependency trees as sequential auto-regressive models. After that is done we can apply any algorithm for sampling without replacement from sequential models.

[4] This is Cayley's formula with an offset of 1 to account for the artificial ROOT node. Cayley's formula was originally defined for undirected spanning trees, but it applies equally to *rooted* directed spanning trees.

5.1 Dependency Tree Distributions in Sequential Auto-Regressive Form

Colbourn's algorithm from Section 3.2 can be interpreted as an auto-regressive form of dependency tree distributions. Here we restate presentation from Algorithm 2 as a state machine in which:

state is a sequence of edges sampled until this point, together with additional information needed for efficient computation of marginals such as an inverse of the graph's Laplacian,

transition from any state to another state adds one more edge that enters the upcoming word, i.e. the upcoming word's head is determined.

The initial state contains an empty set of edges and has transitions that generate the incoming edge for the first word in the sentence. The probabilities of these initial transitions are computed using a method of computing marginals with Matrix-Tree Theorem in $\mathcal{O}(n^3)$ (Koo et al., 2007) as in line 2 of Algorithm 2. This computation is done only once because it is needed only for the initial state. When the state machine transitions from one state to another it selects one of the dependency arcs and then constrains the graph (and its marginals) so that any upcoming selection of incoming edges for future words will have to condition on the newly selected edge. This can be done efficiently using Sherman-Morrison formula in $\mathcal{O}(n^2)$ (Sherman and Morrison, 1950). The detailed formulation of the state machine, pseudocode and usage of Sherman-Morrison formula is presented in Appendix B. This state-machine needs to take n transitions to go from initial state to the final state where each transition costs $\mathcal{O}(n^2)$. In total, unfolding one complete transition sequence takes $\mathcal{O}(n^3)$ which is expected since this is just a reformulation of Colbourn's algorithm.

Now that we have Colbourn's algorithm as an auto-regressive model we can use any SWOR algorithm for sequential models. Two popular options are Trie algorithm by Shi et al. (2020) and Stochastic Beam Search (SBS) by Kool et al. (2019).

Shi et al.'s Trie algorithm for sampling sequences uses an efficient trie data-structure to remove previously sampled sequences from the support of the probability distribution so that the sequence that will be sampled next will not be from the set of already sampled sequences.

Kool et al.'s Stochastic Beam Search takes a different approach that is based on the generalization of Gumbel trick. Gumbel (1954) has shown that sampling from a categorical distribution can be reformulated as finding an *argmax* from logits with special noise added to them. Vieira (2014) has shown that if *argmax* is replaced by *top-k* what we get is k samples without replacement for categorical distribution. Kool et al. (2019, 2020a) generalized this to structured sequential distributions by using beam search and carefully controlling the injected noise.

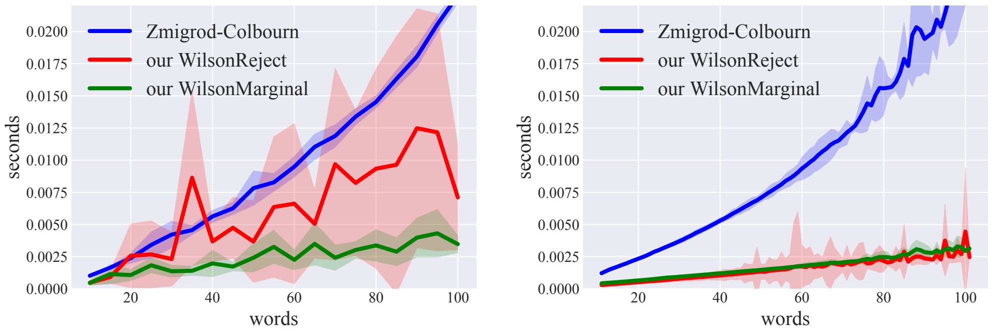
The worst-case complexity of getting k SWOR samples from each of these algorithms is $\mathcal{O}(knt)$ where t is the complexity of taking a single transition. With our representation of Colbourn's algorithm each transition takes $\mathcal{O}(n^2)$ which means the total worst-case complexity is $\mathcal{O}(kn^3)$.

We refer the reader to the papers of Shi et al. (2020) and Kool et al. (2019) for the details of Trie SWOR sampling and SBS since the details are not of particular importance for our approach, in principle any ancestral SWOR could be applied in our case, and we cannot do justice to those works by presenting them briefly. We focus instead on how our approach differs from previously proposed one by Zmigrod et al. (2021b). Given that both our Trie-SWOR and the SWOR of Zmigrod et al. are based on extending Shi et al. (2020) with Colbourn's algorithm it may seem strange that they get a worse worst-case complexity of $\mathcal{O}(kn^3 + k^2n)$. That is because Zmigrod et al. maintain previous samples in an unstructured list of size $\mathcal{O}(k)$ so removing the probability of already sampled sequences requires an additional pass over this list for each new sample. We instead follow Shi et al. (2020) in using a trie data structure and do not need to make any additional passes.

Having both Trie and SBS SWOR options is not redundant as each of these approaches have pros and cons. The advantage of Trie algorithm is that it draws samples incrementally one by one so we can dynamically decide when we have a sufficient number of samples. The advantage of Stochastic Beam Search is that the samples are drawn in parallel which is more efficient on hardware like GPUs.

6 Experiments

We test the empirical performance of the algorithms presented here. We implemented all of them in Python and NumPy in order to comparable to previous work on Zmigrod et al. (2021b) for SWOR sampling. We do not compare performance against their adaptation of Wilson's algorithm for single-



(a) Random weights

(b) Trained weights

Figure 2: Speed for sampling with replacement.

root trees since that adaptation is not correct, as we have shown in Section 4.1.

Sampling with Replacement experiments were conducted in two settings: random weights and trained weights. For random weights we used graphs with weights sampled from the uniform distribution. To get the trained weights we applied Stanza's English model (Qi et al., 2020) on English sentences of the News Commentary corpus.

For each graph we sample 100 trees with replacement. We compared COLBOURN with single-root constraint from Zmigrod et al. (2021b) against our WILSONMARGINAL and WILSONREJECT from Section 4. The results are shown in Figure 2.

Both of our algorithms significantly improve upon the version of Colbourn's algorithm presented in Zmigrod et al. (2021b). Among the two of our algorithms, WILSONMARGINAL performs better in random weights setting, while in trained weights setting their performance is almost identical. This is expected result from the analysis in Section 4.3.1.

In trained weights there are almost no rejected samples so the difference between WILSONREJECT and WILSONMARGINAL comes down to WILSONMARGINAL having matrix-inversion operation which does not seem to cause any slowdown.

Sampling without Replacement tests were conducted in a similar way except that we vary the number of SWOR samples instead of the sentence length. We use sentence length of 14 words because that is the longest sequence that we could run Zmigrod et al. algorithm without encountering issues with numerical instability. In Appendix C we show results with running our algorithm on the longer sentences. We test only on random weights since the runtime of these algorithms does not depend on the weights value. As visible from Figure 3 both of our algorithms outperform the algorithm of Zmigrod et al. (2021b). Of our two algorithms, the Trie algorithm is slightly slower due to the constant factors needed to maintain the trie data structure.

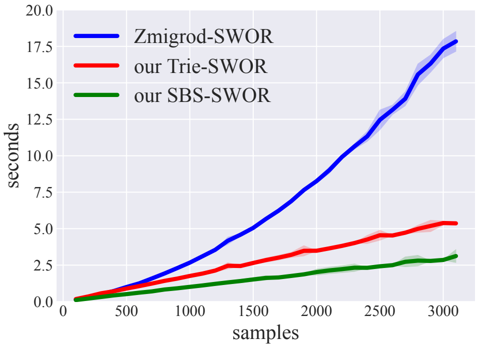


Figure 3: Speed for SWOR sampling with 14 words.

We also reimplemented a version of SBS-SWOR in JAX so that we can see the benefits of having a parallel algorithm executed with parallel hardware and got up to ×5 speedup on GPU over CPU with sentences of length 100 (Figure 4). See Appendix C for tests on other lengths. The main takeaway is that SBS-SWOR allows for a vectorized implementation that can exploit modern hardware accelerators, unlike other SWOR algorithms.

7 Conclusion

We have presented multiple contributions to the techniques for sampling dependency trees that (1) correct errors in the previously published algorithms, (2) propose new algorithms for sampling with replacement, (3) propose new algorithms for sampling without replacement and (4) show that these algorithms have both asymptotically and practically good performance. We hope the proposed algorithms will lead to advancements in all problems where sampling is crucial such as unsuper-

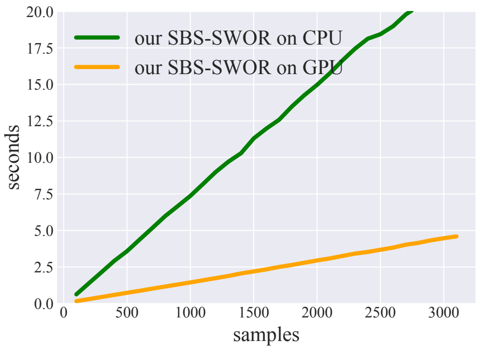


Figure 4: SBS-SWOR in JAX with 100 words.

vised grammar induction (Corro and Titov, 2019), syntax marginalization in syntactic language models (Sartran et al., 2022) and approximate decoding of higher-order models (Zhang et al., 2014).

Limitations

A limitation of our approach is that we have a multitude of different algorithms with their pros and cons instead of one unified algorithm that would have the best properties of all of them. For instance, Wilson-based algorithms are fast in practice, but they provide only expected runtime and no worst-case complexity. Colbourn-based algorithms have a predictable runtime, but are not as fast in practice. Ideally we would have an algorithm that has the practical speed of WILSONMARGINAL, numerical stability of WILSONREJECT, predictable runtime and flexibility of COLBOURN.

One could try to make Colbourn algorithm closer to Wilson's in performance by having a better implementation – in their paper Colbourn et al. (1996) show how sampling can be reduced to matrix multiplication and therefore use sub-cubic algorithms for implementation such as the ones by Strassen (1969), Coppersmith and Winograd (1990) or Fawzi et al. (2022). However, in most cases these sub-cubic algorithms would not reach the speed of Wilson-based algorithms. It would appear that either we have to choose random walk paradigm of Wilson's algorithm or matrix multiplication approach of Colbourn. There are some recent works that show that it may be possible to have a separate paradigm that can with high probability have runtime asymptotically faster than matrix multiplication (Durfee et al., 2017).

Another limitation, in comparison to projective dependency parsing and CFG parsing, is that we have a completely separate algorithms for sampling (presented in this paper) and finding an *argmax* (presented by Stanojević and Cohen (2021)) of a non-projective dependency tree distribution. In projective and CFG parsers it is possible to express both algorithms as a single inside algorithm that just uses different semi-rings for *sampling* and *argmax* (Goodman, 1999; Aziz, 2015; Rush, 2020). It is possible to simulate this to some extent by injecting some noise to weights and then finding *argmax* as done by Corro and Titov (2019), but that provides biased samples.

In this work we also do not deal directly with sampling from higher-order models (McDonald and Pereira, 2006; Koo and Collins, 2010). However, the algorithms presented here can be useful for approximate decoding of higher order models. In general, finding the best tree from a higher-order model is NP-complete (McDonald and Satta, 2007), but as Zhang et al. (2014) show it is possible to decode approximately by sampling from the first-order model and improving the sample using the higher-order model. Zhang et al. have used the original Wilson's algorithm for sampling spanning trees from uniform distribution which means that on average $\frac{2}{e} \approx 73\%$ of the sampled trees would be invalid dependency trees (see Section 4.3.1). It would be easy to apply any of our sampling algorithms in their setting and get valid dependency trees as a better starting point for decoding a higher-order model.

Acknowledgements

I am grateful to Shay Cohen, Clara Meister, Gábor Melis, Laura Rimell and Chris Dyer for the comments on the earlier versions of this work.

References

Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2017. Factorization and Inversion of a Million Matrices using GPUs: Challenges and Countermeasures. *Procedia Computer Science*, 108:606–615. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

Wilker Aziz. 2015. Grasp: Randomised Semiring Parsing. *Prague Bulletin of Mathematical Linguistics*, 104:51–62.

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.

Arthur Cayley. 1889. A theorem on trees. *Quarterly Journal of Mathematics*, 23:376–378.

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

Don Coppersmith and Shmuel Winograd. 1990. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280. Computational algebraic complexity editorial.

Caio Corro and Ivan Titov. 2019. Differentiable Perturb-and-Parse: Semi-Supervised Parsing with a Structured Variational Autoencoder. In *International Conference on Learning Representations*.

Jeremy J Du Croz and Nicholas J Higham. 1992. Stability of methods for matrix inversion. *IMA Journal of Numerical Analysis*, 12(1):1–19.

David Durfee, Rasmus Kyng, John Peebles, Anup B. Rao, and Sushant Sachdeva. 2017. Sampling random spanning trees faster than matrix multiplication. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2017, page 730–742, New York, NY, USA. Association for Computing Machinery.

Jason Eisner. 2016. Inside-outside and forward-backward algorithms are just backprop (tutorial paper). In *Proceedings of the Workshop on Structured Prediction for NLP*, pages 1–17, Austin, TX. Association for Computational Linguistics.

Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. 2022. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53.

Joshua Goodman. 1999. Semiring Parsing. *Computational Linguistics*, 25(4):573–606.

Emil Julius Gumbel. 1954. Statistical theory of extreme values and some practical applications: a series of lectures.

Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms*, second edition, chapter 14. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

Terry Koo and Michael Collins. 2010. Efficient Third-Order Dependency Parsers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1–11, Uppsala, Sweden. Association for Computational Linguistics.

Terry Koo, Amir Globerson, Xavier Carreras, and Michael Collins. 2007. Structured Prediction Models via the Matrix-Tree Theorem. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 141–150, Prague, Czech Republic. Association for Computational Linguistics.

Wouter Kool, Herke Van Hoof, and Max Welling. 2019. Stochastic beams and where to find them: The Gumbel-top-k trick for sampling sequences without replacement. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3499–3508. PMLR.

Wouter Kool, Herke van Hoof, and Max Welling. 2020a. Ancestral Gumbel-Top-k Sampling for Sampling Without Replacement. *Journal of Machine Learning Research*, 21:1–36.

Wouter Kool, Herke van Hoof, and Max Welling. 2020b. Estimating gradients for discrete random variables by sampling without replacement. In *International Conference on Learning Representations*.

Sandra Kübler, Ryan McDonald, and Joakim Nivre. 2009. Dependency Parsing. *Synthesis lectures on human language technologies*, 1(1):1–127.

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.

Ryan McDonald and Fernando Pereira. 2006. Online Learning of Approximate Dependency Parsing Algorithms. In *11th Conference of the European Chapter of the Association for Computational Linguistics*, pages 81–88, Trento, Italy. Association for Computational Linguistics.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005. Non-Projective Dependency Parsing using Spanning Tree Algorithms. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 523–530, Vancouver, British Columbia, Canada. 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*, pages 121–132, Prague, Czech Republic. Association for Computational Linguistics.

Kevin P. Murphy. 2012. *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning Series. The MIT Press.

Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Jan Hajič, Christopher D. Manning, Sampo Pyysalo, Sebastian Schuster, Francis Tyers, and Daniel Zeman. 2020. Universal Dependencies v2: An evergrowing multilingual treebank collection. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 4034–4043, Marseille, France. European Language Resources Association.

Timothy Osborne. 2019. *A Dependency Grammar of English: An introduction and beyond*. John Benjamins.

Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. 2020. Stanza: A Python natural language processing toolkit for many human languages. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*.

Alexander Rush. 2020. Torch-Struct: Deep Structured Prediction Library. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 335–342, Online. Association for Computational Linguistics.

Laurent Sartran, Samuel Barrett, Adhiguna Kuncoro, Miloš Stanojević, Phil Blunsom, and Chris Dyer. 2022. Transformer Grammars: Augmenting Transformer Language Models with Syntactic Inductive Biases at Scale.

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 *International Conference on Machine Learning*, pages 8785–8795. PMLR.

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)*, pages 132–140, 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 2021 Conference on Empirical Methods in Natural Language Processing*, pages 10540–10557, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Volker Strassen. 1969. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356.

R. E. Tarjan. 1977. Finding optimum branchings. *Networks*, 7(1):25–35.

W. T. Tutte. 1984. *Graph Theory*, volume 21 of *Encyclopedia of Mathematics and Its Applications*. Addison-Wesley, Menlo Park, CA.

UD. 2022. Root relation in universal dependencies. `https://universaldependencies.org/u/dep/root.html`. Accessed: 2022-05-24.

Tim Vieira. 2014. Gumbel-max trick and weighted reservoir sampling, 2014. Blog post accessed on 13 Jan 2022. https://timvieira.github.io/blog/post/2014/08/01/gumbel-max-trick-and-weighted-reservoir-sampling.

Tim Vieira. 2017. Estimating means in a finite universe. Blog post accessed on 16 Oct 2022. https://timvieira.github.io/blog/post/2017/07/03/estimating-means-in-a-finite-universe/.

Martin J. Wainwright and Michael I. Jordan. 2008. Graphical Models, Exponential Families, and Variational Inference. *Foundations and Trends® in Machine Learning*, 1(1–2):1–305.

David Bruce Wilson. 1996. Generating Random Spanning Trees More Quickly than the Cover Time. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 296–303, New York, NY, USA. Association for Computing Machinery.

Yuan Zhang, Tao Lei, Regina Barzilay, and Tommi Jaakkola. 2014. 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.

Ran Zmigrod. 2022. *A hitchhiker's guide to efficient non-projective dependency parsing*. Ph.D. thesis, University of Cambridge.

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. 2021a. Efficient Computation of Expectations under Spanning Tree Distributions. 9:675–690.

Ran Zmigrod, Tim Vieira, and Ryan Cotterell. 2021b. Efficient Sampling of Dependency Structures. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 10558–10569, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Ran Zmigrod, Tim Vieira, and Ryan Cotterell. 2021c. On finding the k-best non-projective dependency trees. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1324–1337, Online. Association for Computational Linguistics.

A Matrix-Tree Theorem

Matrix-Tree Theorem (MTT) is an important theorem from graph theory that establishes the connection between spanning trees of a graph and its Laplacian matrix. It allows for an efficient computations of the number of spanning trees present in a graph. It was originally devised for counting undirected spanning trees, but it was later extended to weighted directed spanning trees by Tutte (1984). In 2007 three different papers simultaneously presented application of MTT to dependency parsing (McDonald and Satta, 2007; Smith and Smith, 2007; Koo et al., 2007). Out of these Koo et al. (2007) is of particular interest here because it extends MTT to have a single-root constraint that all valid dependency trees should have.

As usual, we can represent any weighted directed graph using a weighted adjacency matrix W of shape $(n+1, n+1)$ where $W[i, j]$ defines the weight of an edge $i \rightarrow j$. All weights must be non-negative real numbers. Node 0 is reserved for the artificial ROOT token while all other nodes are regular words that appear in the sentence. Self-loops are not allowed so all diagonal entries have weight 0. There cannot be any edges entering ROOT so all entries in the 0th column are also 0.

Degree matrix D is a diagonal matrix that on the ith element of the diagonal contains the weighted in-degree of the ith node. The in-degree is simply the sum of weights of all the edges that enter ith node.

Laplacian matrix L is defined by equation:

$$L = D - W$$

Matrix-Tree Theorem in the form presented by Tutte (1984) states that any co-factor of a graph's Laplacian matrix will be the partition function over the distribution of all spanning trees in the graph. For instance, computing the partition function can be done in the following way:

$$Z = |L[1\!:, 1\!:]|$$

Here we have used Python notation for slicing for convenience. Determinant of this sub-matrix of a Laplacian gives us a partition function. Sometimes in the literature this sub-matrix of a Laplacian is referred to as the Laplacian. The only non-trivial computation here is computation of a determinant that can be done as fast as matrix-multiplication but in most practical cases is implemented in $\mathcal{O}(n^3)$. For convenience let us name the function that computes this sub-matrix $\hat{L}$ i.e.:

$$\hat{L}(W) = (\text{degree}(W) - W)[1\!:, 1\!:] \tag{5}$$

Koo et al. (2007) make an adjustment to this sub-matrix in order to incorporate the single-root constraint. To do that it is necessary to separately treat the edges that come out of the ROOT node. These edges' weights are located in $W[0, 1\!:]$. The modification by Koo et al. is (1) to not use ROOT edges in the construction of the degree matrix, and (2) to replace first row of the Laplacian sub-matrix with those edges. Algorithm 5 shows a JAX NumPy implementation of the modified $\hat{L}$.

Algorithm 5 Computation of Laplacian sub-matrix $\hat{L}(W)$ for single-rooted dependency trees.

```
def L_hat(W):
  Wp = W[1:, 1:]
  I = jnp.eye(Wp.shape[0])
  D = jnp.sum(Wp, axis=0) * I
  L = D - Wp
  return L.at[0].set(Wp[0])
```

Note that each line takes $\mathcal{O}(n^2)$ to compute.

Another useful statistic is computation of marginals that can be done by extending MTT further. For finding marginals it often comes handy to use the identity which states that the gradient of a log-partition is equivalent to marginals (Wainwright and Jordan, 2008; Eisner, 2016):

$$M(W) = \frac{\partial \log |\hat{L}(W)|}{\partial W}$$

The simplest way to implement this is by using automatic differentiation tools (Zmigrod et al., 2021a), but it can also be done by an explicit computation of a derivative of a determinant which involves inverting Laplacian sub-matrix as presented by Koo et al. (2007). For the motivation of that derivation see the original paper. In Algorithm 6 we present only the few lines of JAX NumPy code that are needed for implementing this computation. The code could be written in a single function but it will become clear in Appendix B why it is beneficial to have it as two separate functions.

Each line takes $\mathcal{O}(n^2)$ to execute except for the line that computes matrix inversion which takes $\mathcal{O}(n^3)$.

Algorithm 6 Computation of marginals for single-rooted dependency trees.

```
def marginals(W):
  L = L_hat(W)
  B = jnp.linalg.inv(L).T
  return _for_a_given_B(W, B)

def _for_a_given_B(W, B):
  M = jnp.zeros(W.shape)
  X = jnp.diag(B)\
        .at[0].set(0)\
        .reshape(1, -1)
  Y = B.at[0].set(0)
  M = M.at[1:, 1:].set(
          W[1:, 1:] * (X - Y)
      ).at[0 , 1:].set(
          W[0, 1:] * jnp.diag(B)
      )
  return M
```

Algorithm 7 COLBOURN's sampling algorithm.

```
1: $t \leftarrow \emptyset$          ▷ Sampled tree edges where
                                    ▷ $t[i] = j$ stands for edge $j \to i$
2: $M \leftarrow$ marginals$(W)$
3: for $i \in [1 \dots n]$ do       ▷ Loop over words
4:     Sample node $v$ with weight $M(v \to u)$
5:     $t[i] \leftarrow v$
6:     $M \leftarrow$ constrain$(M, v, i)$
7: return $t$
```

B Colbourn Algorithm

This section continues on the discussion from Section 3.2 and Appendix A. We are repeating the general specification of the Colbourn algorithm in Algorithm 7. The Colbourn algorithm samples the edges of a dependency/spanning tree one by one ordered by their end points. So if we have have n words it will first find an edge entering word w_1, then edge entering word w_2 etc. In the initial state of the algorithm, line 2, the algorithm computes the total probability that is concentrated for each edge that could be chosen for the first word. This is done by simply computing the marginals for the whole graph as in Algorithm 6. Given the marginals it is easy to sample the first edge in line 4. After that edge is sampled and stored we need to adjust the marginals so that they condition on the selected edge. A naïve way of doing that would be to edit the weight matrix W so that all other edges entering the word have weight 0 and then rerun Algorithm 6 to find the new marginals. If we repeat this process n times we will select one entering edge for each word and form a full dependency tree.

A problem with the naïve implementation is that its complexity would be $\mathcal{O}(n^4)$. The main bottleneck is matrix inversion that needs to be done in the computation of marginals after each new edge is added. Colbourn et al. noticed that it is possible reuse the previously computed matrix-inversion of matrix W to compute the matrix-inversion of the new matrix W' which differs from W by only one column. The method for doing that is Sherman-Morrison formula (Sherman and Morrison, 1950) that makes the computation of marginals alone $\mathcal{O}(n^2)$ and the complexity of the whole Colbourn algorithm down to $\mathcal{O}(n^3)$. Here we present the version that updates B which is not only matrix-inversion but also a transposed matrix. In the ith iteration of the loop the algorithm will sample edge $j \to i$ as the entering edge for the word w_i. We can update the matrices with the following equations:

$$W'[a,b] = \begin{cases} 0, & \text{if } a \neq j \wedge b = i \\ W[a,b], & \text{otherwise} \end{cases} \tag{6}$$

$$u = \hat{L}(W')[:,i] - \hat{L}(W)[:,i] \tag{7}$$

$$B' = \frac{B[:,j](u^T B)}{1 + u^T B[:,j]} \tag{8}$$

Eq 6 will compute the new weight matrix W' by changing only one column of matrix W. Eq 7 computes the difference of that column between the Laplacians of old and new weight matrix. Eq 8 applies Sherman-Morrison formula to find the updated B matrix.

The main idea of our SWOR sampling algorithms is to use Colbourn's sampling algorithm as a basis for treating distributions over dependency trees as an sequential auto-regressive model and then apply existing sampling algorithms for sequential models to dependency trees. This reformulation is presented in Algorithm 8. This is the same Colbourn algorithm but expressed as a state machine. The state contains the current position, current weight matrix, current Laplacian sub-matrix, and its inverse-transpose. Computation of the initial state takes $\mathcal{O}(n^3)$ due to the matrix inversion. Finding probabilities for transitioning to a next state is $\mathcal{O}(n^2)$ because we already have matrix-inverse as a part of the state. Transitioning to the next state also takes $\mathcal{O}(n^2)$ because we apply Sherman-Morrison formula to update the inverse-transpose for the selected edge j.

Algorithm 8 Dependency tree model as a sequence generation model.

```
def initial_state(W):
  i = 1
  L = L_hat(W)
  B = jnp.linalg.inv(L).T
  return i, W, L, B

def transition_probs(state)
  i, W, L, B = state
  return _for_a_given_B(W, B)[:,i]

def transit_state(state, j):
  Wp, Bp = ...  apply Eq (6)-(8)
  return i+1, Wp, L_hat(Wp), Bp
```

C Additional SWOR Results

In the main text we have shown the results against the only previous SWOR sampler for dependency trees by Zmigrod et al. (2021b) for sentences with 14 words. However, we could not run their algorithm for longer sentences so here we show only the results of our SWOR algorithms for different sentence length. In Figure 5 are results of running the NumPy implementation of Trie-SWOR and SBS-SWOR and SBS-SWOR consistently outperforms Trie-SWOR due to the lower constant factors.

The real benefit of SBS-SWOR comes with parallel hardware. To show that we needed to reimplement the algorithm in some other toolkit that supports GPU execution. We used JAX as it has interface similar to NumPy. It is difficult to implement Trie-SWOR with JAX because JAX needs to know all shapes ahead of time, which is not possible to do with the dynamically growing data structure such as trie. We have implemented only SBS-SWOR and applied just-in-time compilation to it. The algorithm becomes substantially faster even on CPU, most likely because the compilation removes the Python overhead. Even such an optimized version that runs fast on CPU gets much faster if executed on GPU as visible from Figure 6. All experiments were ran on Intel Xeon® W-2135 CPU and NVIDIA® Quadro P1000 GPU.

D Ratio of average tree weights

In Section 4.3.1 we have shown that with the first-order approximation of the expectation of ratio of dependency and spanning tree average weights, the expected number of trials of WilsonReject smaller than e. Here we give empirical support for it based on large number of simulations with different distributions and graph sizes.

To verify the approximation we have sampled $10,000$ weighted graphs for different combinations of weight distributions and sentence lengths and estimated ratio $\frac{w^T_{avg}}{w^D_{avg}}$. The results are shown in Figure 7. Since all the weights need to be non-negative we used the truncated version of Normal distribution, exponential distribution and uniform distribution. As can be seen from the results no matter what distribution we sample weights from and no matter what the sentence length is, the average value of the ratio $\frac{w^T_{avg}}{w^D_{avg}}$ is ≈ 1 which justifies the approximation made in Section 4.3.1.

If we look only at the plots for the Truncated Normal distribution they look almost identical independently of the particular parameters that are used as mean and variance. Exponential seems to have a different shape of the distribution of ratio for small graph sizes, but the mean value of the ratio is still 1.

Interestingly, for almost all sampled graphs the ratios of averages are smaller than 2. If we make a weaker assumption that $\mathbb{E}\left[\frac{w^T_{avg}}{w^D_{avg}}\right] < 2$ the main statement of Section 4.3.1 would still hold: WilsonReject is on average as fast as Wilson up to the small multiplicative constant.

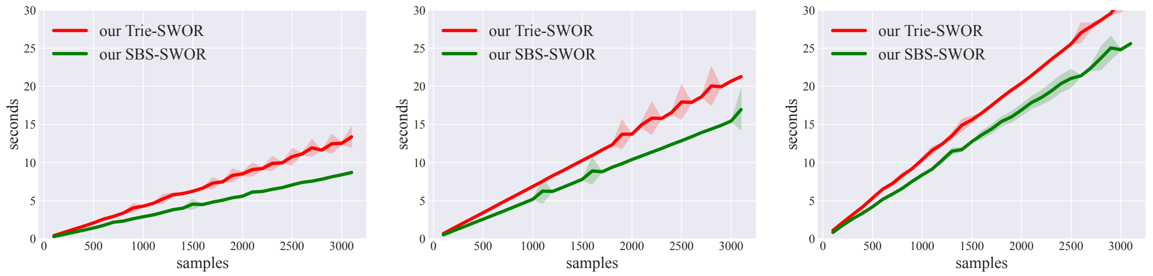
E Formal Proof that WilsonRC is Biased

In Section 4.1 we have showed that WilsonRC algorithm is biased through an example. Here we will show that formally and answer the question under what conditions can WilsonRC be unbiased. As a refresher, WilsonRC works as follows:

1. sample one edge e emanating from ROOT using their weights w_e,
2. remove all ROOT outgoing edges except e,
3. run regular Wilson's algorithm.

The probability of sampling any dependency tree t is given by the product of probability of sampling the root edge e of t in step 1 and sampling t in step 3 from all the trees that remain after filtering in step 2. If all the edges emanating from ROOT are given by set R, we can express the probability of sampling t with WilsorRC as:

$$\frac{\phi(e)}{\sum_{e'\in R}\phi(e')}\frac{\phi(t)}{\sum_{t'\in D_e}\phi(t')} \tag{9}$$

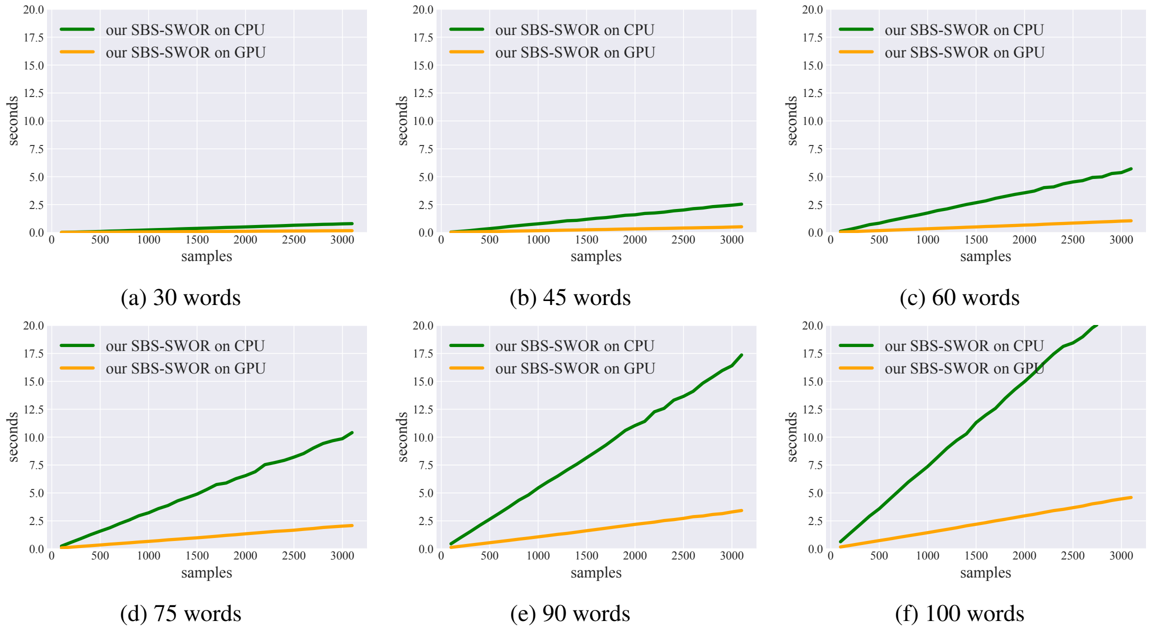


(a) 30 words

(b) 45 words

(c) 60 words

Figure 5: Comparison of NumPy implementation of Trie-SWOR and SBS-SWOR algorithms over different sentence lengths.



(a) 30 words

(b) 45 words

(c) 60 words

(d) 75 words

(e) 90 words

(f) 100 words

Figure 6: Comparison of JAX implementation of SBS-SWOR on CPU vs GPU for different sentence lengths.

The sampling algorithm is unbiased if only if the probability of sampling any tree t is given by $p(t) = \frac{\phi(t)}{Z}$ where Z is a partition function. For WILSONRC to be unbiased all parts of Equation 9 except for $\phi(t)$ need to add up to Z:

$$\begin{aligned} \frac{\phi(e)}{\sum_{e' \in R} \phi(e')} \frac{1}{\sum_{t' \in D_e} \phi(t')} &= \frac{1}{Z} \\ \iff \frac{\phi(e)}{\sum_{t' \in D_e} \phi(t')} &= \frac{\sum_{e' \in R} \phi(e')}{Z} \end{aligned} \tag{10}$$

Since Equation 10 should hold for **all** root edges, and since the righthand side of the equation is constant for a given weighted graph, it follows that all root edges have the same value for the lefthand side of Equation 10. If we pick two arbitrary root edges e_1 and e_2 we get:

$$\begin{aligned} \frac{\phi(e_1)}{\sum_{t' \in D_{e_1}} \phi(t')} &= \frac{\phi(e_2)}{\sum_{t' \in D_{e_2}} \phi(t')} \\ \iff \frac{\phi(e_1)}{\phi(e_2)} &= \frac{\sum_{t' \in D_{e_1}} \phi(t')}{\sum_{t' \in D_{e_2}} \phi(t')} \\ \iff \frac{\phi(e_1)}{\phi(e_2)} &= \frac{p(e_1)}{p(e_2)} \end{aligned}$$

In other words, WILSONRC is unbiased if and only if the weights on the root edges happen to be exactly proportional to their marginal value which is an extremely unlikely scenario to happen with randomly sampled weight graph. For any other setting of weights WILSONRC returns biased samples. WILSONMARGINAL solves this problem by sampling the root edge directly from the marginal probability of the root edges, instead of using edge weights directly.

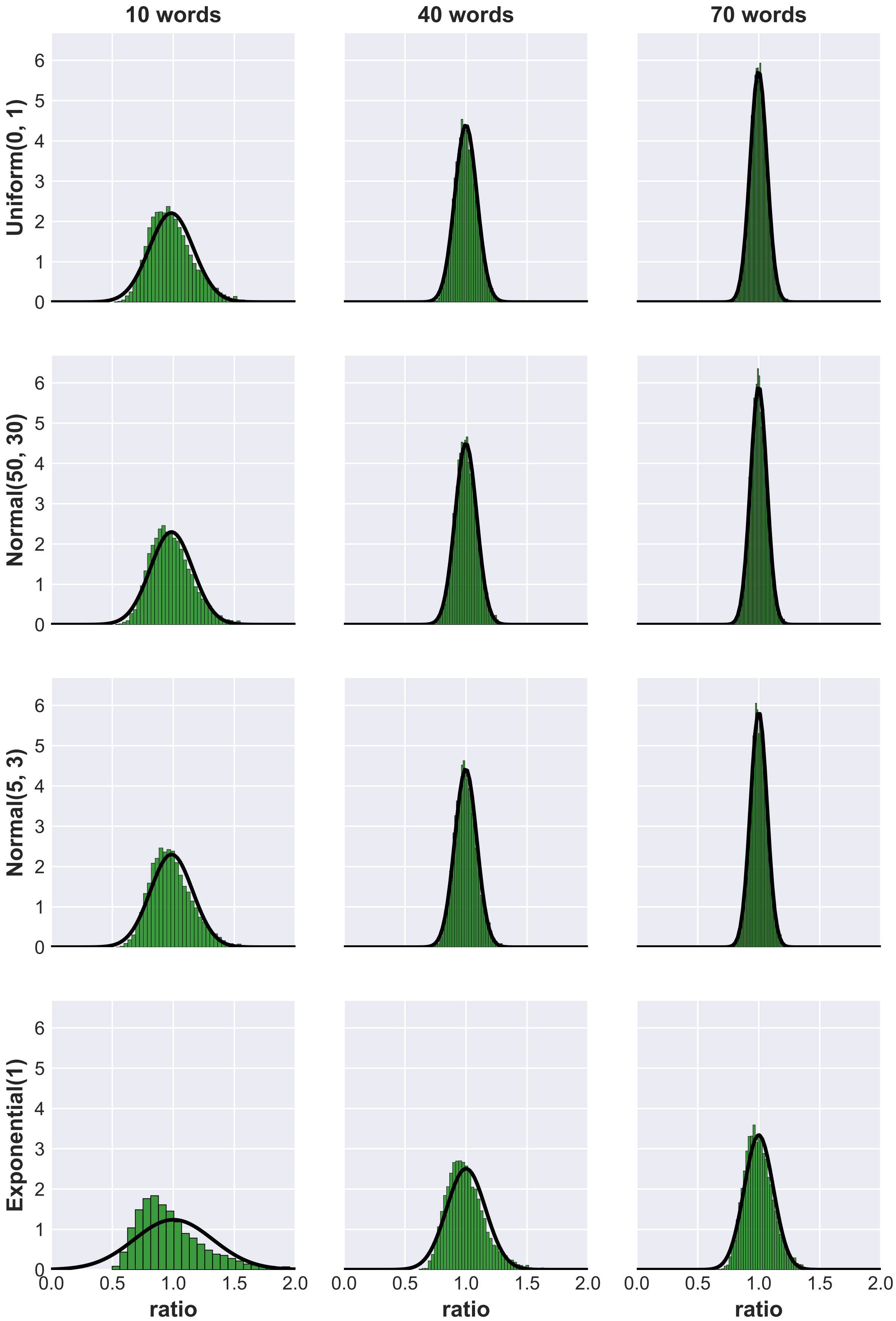


Figure 7: Distribution of ratio $\frac{w^T_{avg}}{w^D_{avg}}$ over graphs whose weights are sampled from different distributions.