On Finding the K-best Non-projective Dependency Trees

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

The connection between the maximum spanning tree in a directed graph and the best dependency tree of a sentence has been exploited by the NLP community. However, for many dependency parsing schemes, an important detail of this approach is that the spanning tree must have exactly one edge emanating from the root. While work has been done to efficiently solve this problem for finding the one-best dependency tree, no research has attempted to extend this solution to finding the K-best dependency trees. This is arguably a more important extension as a larger proportion of decoded trees will not be subject to the root constraint of dependency trees. Indeed, we show that the rate of root constraint violations increases by an average of 13 times when decoding with $K = 50$ as opposed to $K = 1$. In this paper, we provide a simplification of the K-best spanning tree algorithm of Camerini et al. (1980). Our simplification allows us to obtain a constant time speed-up over the original algorithm. Furthermore, we present a novel extension of the algorithm for decoding the K-best dependency trees of a graph which are subject to a root constraint.[1]

1 Introduction

Non-projective, graph-based dependency parsers are widespread in the NLP literature. (McDonald et al., 2005; Dozat and Manning, 2017; Qi et al., 2020). However, despite the prevalence of K-best dependency parsing for other parsing formalisms—often in the context of re-ranking (Collins and Koo, 2005; Sangati et al., 2009; Zhu et al., 2015; Do and Rehbein, 2020) and other areas of NLP (Shen et al., 2004; Huang and Chiang, 2005; Pauls and Klein, 2009; Zhang et al., 2009), we have only found three works that consider K-best non-projective

[1]Our implementation is available at https://github.com/rycolab/spanningtrees.

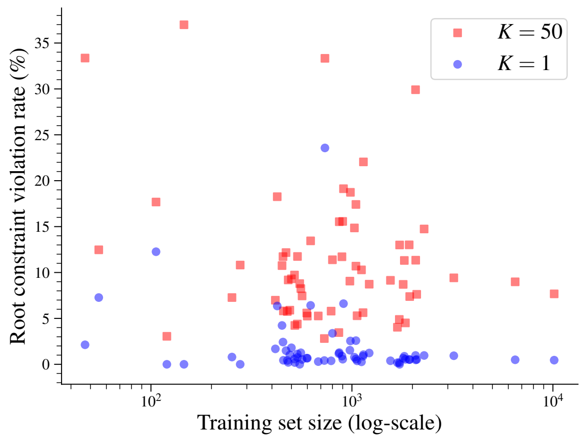

Figure 1: Violation rate of the root constraint when using regular K-best decoding (Camerini et al., 1980) on pre-trained models of Qi et al. (2020) for languages with varying training set sizes.

dependency parsing (Hall, 2007; Hall et al., 2007; Agić, 2012). All three papers utilize the K-best spanning tree algorithm of Camerini et al. (1980). Despite the general utility of K-best methods in NLP, we suspect that the relative lack of interest in K-best non-projective dependency parsing is due to the implementation complexity and nuances of Camerini et al. (1980)'s algorithm.[2]

We make a few changes to Camerini et al. (1980)'s algorithm, which result in both a simpler algorithm and simpler proof of correctness.[3] Firstly, both algorithms follow the key property that we can find the second-best tree of a graph by removing a single edge from the graph (Theorem 1); this property is used iteratively to enumerate the K-best trees in order. Our approach to finding the second-best tree (see §3) is faster because of it performs half as many of the expensive cycle-contraction operations (see §2). Overall, this change is responsible for our 1.39x speed-up

[2]In fact, an anonymous reviewer called it "one of the most 'feared' algorithms in dependency parsing."

[3]While our algorithm is by no means *simple*, an anonymous reviewer called it "a big step in that direction."

Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, pages 1324–1337
August 1–6, 2021. ©2021 Association for Computational Linguistics

(see §4). Secondly, their proof of correctness is based on reasoning about a complicated ordering on the edges in the K^{th} tree (Camerini et al., 1980, Section 4); our proof side-steps the complicated ordering by directly reasoning over the ancestry relations of the K^{th} tree. Consequently, our proofs of correctness are considerably simpler and shorter. Throughout the paper, we provide the statements of all lemmas and theorems in the main text, but defer all proofs to the appendix.

In addition to simplifying Camerini et al. (1980)'s algorithm, we offer a novel extension. For many dependency parsing schemes such as the Universal Dependency (UD) scheme (Nivre et al., 2018), there is a restriction on dependency trees to only have one edge emanate from the root.[4] Finding the maximally weighted spanning tree that obeys this constraint was considered by Gabow and Tarjan (1984) who extended the $\mathcal{O}(N^2)$ maximum spanning tree algorithm of Tarjan (1977); Camerini et al. (1979). However, no algorithm exists for K-best decoding of dependency trees subject to a root constraint. As such, we provide the first K-best algorithm that returns dependency trees that obey the root constraint.

To motivate the practical necessity of our extension, consider Fig. 1. Fig. 1 shows the percentage of trees that violate the root constraint when doing one-best and 50-best decoding for 63 languages from the UD treebank (Nivre et al., 2018) using the pre-trained model of Qi et al. (2020).[5,6] We find that decoding without the root constraint has a much more extreme effect when decoding the 50-best than the one-best. Specifically, we observe that on average, the number of violations of the root constraint increased by 13 times, with the worst increase being 44 times. The results thus suggest that finding K-best trees that obey the root constraint from a non-projective dependency parser requires a specialist algorithm. We provide a more detailed results table in App. A, including root constraint violation rates for $K=5$, $K=10$, and $K=20$. Furthermore, we note that the K-best algorithm may also be used for marginalization of latent variables (Correia et al., 2020) and for constructing parsers with global scoring functions (Lee et al., 2016).

[4]There are certain exceptions to this such as the Prague Treebank (Bejček et al., 2013).

[5]Zmigrod et al. (2020) conduct a similar experiment for only the one-best tree.

[6]We note that Qi et al. (2020) do apply the root constraint for one-best decoding, albeit with a sub-optimal algorithm.

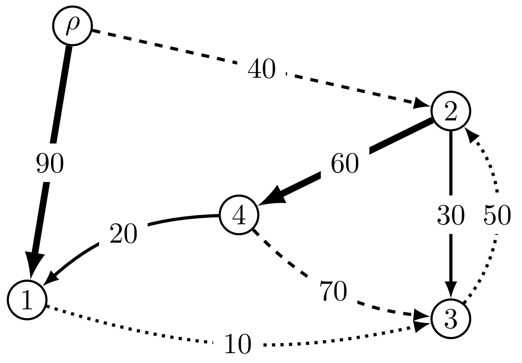

Figure 2: Example graph G (taken from Zmigrod et al. (2020)). Edges that are part of both the best tree $G^{(1)}$ and the best dependency tree $G^{[1]}$ are marked as thick solid edges. Edges only in $G^{(1)}$ are dashed and edges only in $G^{[1]}$ are dotted.

2 Finding the Best Tree

We consider the study of **rooted directed weighted graphs**, which we will abbreviate to simply **graphs**.[7] A graph is given by $G = (\rho, \mathcal{N}, \mathcal{E})$ where $\mathcal{N}$ is a set of $N+1$ nodes with a designated root node $\rho \in \mathcal{N}$ and $\mathcal{E}$ is a set of directed weighted edges. Each edge $e = (i \rightarrow j) \in \mathcal{E}$ has a weight $w(e) \in \mathbb{R}_+$. We assume that self-loops are not allowed in the graph (i.e., $(i \rightarrow i) \notin \mathcal{E}$). Additionally, we assume our graph is not a multi-graph, therefore, there can exist at most one edge from node i to node j.[8] When it is clear from context, we abuse notation and use $j \in G$ and $e \in G$ for $j \in \mathcal{N}$ and $e \in \mathcal{E}$ respectively. When discussing runtimes, we will assume a fully connected graph ($|\mathcal{E}| = N^2$).[9] An **arborescence** (henceforth called a **tree**) of G is a subgraph $d = (\rho, \mathcal{N}, \mathcal{E}')$ such that $\mathcal{E}' \subseteq \mathcal{E}$ and the following is true:

1. For all $j \in \mathcal{N} \smallsetminus \{\rho\}$, $|\{(\_ \rightarrow j) \in \mathcal{E}'\}| = 1$.
2. d does not contain any cycles.

Other definitions of trees can also include that there is at least one edge emanating from the root. However, this condition is immediately satisfied by the above two conditions. A **dependency tree**

[7]As we use the algorithm in Zmigrod et al. (2020) as our base algorithm, we borrow their notation wherever convenient.

[8]We make this assumption for simplicity, the algorithms presented here will also work with multi-graphs. This might be desirable for decoding labeled dependency trees. However, we note that in most graph-based parsers such as Qi et al. (2020) and Ma and Hovy (2017), dependency labels are extracted after the unlabeled tree has been decoded.

[9]We make this assumption as in the context of dependency parsing, we generate scores for each possible edge. Furthermore, (Tarjan, 1977) prove that the runtime of finding the best tree for dense graphs is $\mathcal{O}(N^2)$. This is $\mathcal{O}(|\mathcal{E}| \log N)$ in the non-dense case.

$d = (\rho, \mathcal{N}, \mathcal{E}')$ is a tree with the extra constraint

3. $|\{(\rho \rightarrow \_) \in \mathcal{N}'\}| = 1$

The set of all trees and dependency trees in a graph are given by $\mathcal{A}(G)$ and $\mathcal{D}(G)$ respectively. The weight of a tree is given by the sum of its edge weights[10]

$$w(d) = \sum_{e \in d} w(e) \tag{1}$$

This paper concerns finding the K highest-weighted (henceforce called ***K*-best**) tree or dependency tree, these are denoted by $G^{(K)}$ and $G^{[K]}$ respectively. Tarjan (1977); Camerini et al. (1979) provided the details for an $\mathcal{O}(N^2)$ algorithm for decoding the one-best tree. This algorithm was extended by Gabow and Tarjan (1984) to find the best dependency tree in $\mathcal{O}(N^2)$ time. We borrow the algorithm (and notation) of Zmigrod et al. (2020), who provide an exposition and proofs of these algorithms in the context of non-projective dependency parsing. The pseudocode for finding $G^{(1)}$ and $G^{[1]}$ is given in Fig. 3. We briefly describe the key components of the algorithm.[11]

The **greedy graph** of G is denoted by $\overrightarrow{G} = (\rho, \mathcal{N}, \mathcal{E}')$ where $\mathcal{E}'$ contains the highest weighted incoming edge to each non-root node. Therefore, if $\overrightarrow{G}$ has no cycles, then $\overrightarrow{G} = G^{(1)}$. A cycle C in $\overrightarrow{G}$ is called a **critical cycle**. If we encounter a critical cycle in the algorithm, we contract the graph by the critical cycle. A graph **contraction**, $G_{/C}$, by a cycle C replaces the nodes in C by a mega-node c such that the nodes of $G_{/C}$ are $\mathcal{N} \setminus C \cup \{c\}$. Furthermore, for each edge $e = (i \rightarrow j) \in G$:

1. If $i \notin C$ and $j \in C$, then $e' = (i \rightarrow c) \in G_{/C}$ such that $w(e') = w(e) + w\left(\overrightarrow{C_j}\right)$ where C_j is the subgraph of C rooted at j.
2. If $i \in C$ and $j \notin C$, then $e' = (c \rightarrow j) \in G_{/C}$ such that $w(e') = w(e)$.
3. If $i \notin C$ and $j \notin C$, then $e \in G_{/C}$.
4. If $i \in C$ and $j \in C$, then there is no edge related to $(i \rightarrow j)$ in $G_{/C}$.

There also exists a bookkeeping function π such that for all $e' \in G_{/C}$, $\pi(e') \in G$. This bookkeeping function returns the edge in the original graph that led to the creation of the edge in the contracted graph using one of the constructions above.

[10]For inference, the weight of a trees often decomposes multiplicatively rather than additively over the edges. One can take the exponent (or logarithm) of the original edge weights to make the weights distribute additively (or multiplicative).

[11]For a more complete and detailed description as well as a proof of correctness, please refer to the original manuscripts.

```
1:  def opt(G) :
2:    if $\overrightarrow{G}$ has a cycle $C$ :                ▷ Recursive case
3:      return opt$(G_{/C}) \looparrowright C$
4:    else                                          ▷ Base case
5:      if we require a dependency tree :
6:        return constrain$(G)$
7:      else
8:        return $\overrightarrow{G}$
9:  def constrain(G) :
10:   $\sigma \leftarrow$ set of $\rho$'s outgoing edges in $\overrightarrow{G}$
11:   if $|\sigma| = 1$ : return $\overrightarrow{G}$          ▷ Constraint satisfied
12:   $e \leftarrow \operatorname{argmax}_{e' \in \sigma} w\left(\overrightarrow{G \setminus\!\!\setminus e'}\right)$
13:   if $\overrightarrow{G \setminus\!\!\setminus e}$ has cycle $C$ :
14:     return constrain$(G_{/C}) \looparrowright C$
15:   else
16:     return constrain$(G \setminus\!\!\setminus e)$
```

Figure 3: Algorithms for finding $G^{(1)}$ and $G^{[1]}$. These are from Zmigrod et al. (2020).

Finding $G^{(1)}$ is then the task of finding a contracted graph G' such that $\overrightarrow{G'} = G'^{(1)}$. Once this is done, we can stitch back the cycles we contracted. If $G' = G_{/C}$, for any $d \in \mathcal{A}(G_{/C})$, $d \looparrowright C \in \mathcal{A}(G)$ is the tree made with edges $\pi(d)$ (π applied to each edge d) and $\overrightarrow{C_j}$ where C_j is the subgraph of the nodes in C rooted at node j and $\pi(e) = (i \rightarrow j)$ for $e = (i \rightarrow c) \in d$. The contraction weighting scheme means that $w(d) = w(d \looparrowright C)$ (Georgiadis, 2003). Therefore, $G^{(1)} = (G'^{(1)} \looparrowright C)^{(1)}$.

The strategy for finding $G^{[1]}$ is to find the contracted graph for $G^{(1)}$ and attempt to remove edges emanating from the root. This was first proposed by Gabow and Tarjan (1984). When we consider removing an edge emanating from the root, we are doing this in a possibly contracted graph, and so an edge $(\rho \rightarrow j)$ may exist multiple times in the graph. We denote $G \setminus\!\!\setminus e$ to be the graph G with all edges with the same end-points as e removed. Fig. 2 gives an example of a graph G, its best tree $G^{(1)}$, and its best dependency tree $G^{[1]}$.

The runtime complexity of finding $G^{(1)}$ or $G^{[1]}$ is $\mathcal{O}(N^2)$ for dense graphs by using efficient priority queues and sorting algorithms (Tarjan, 1977; Gabow and Tarjan, 1984). We assume this runtime

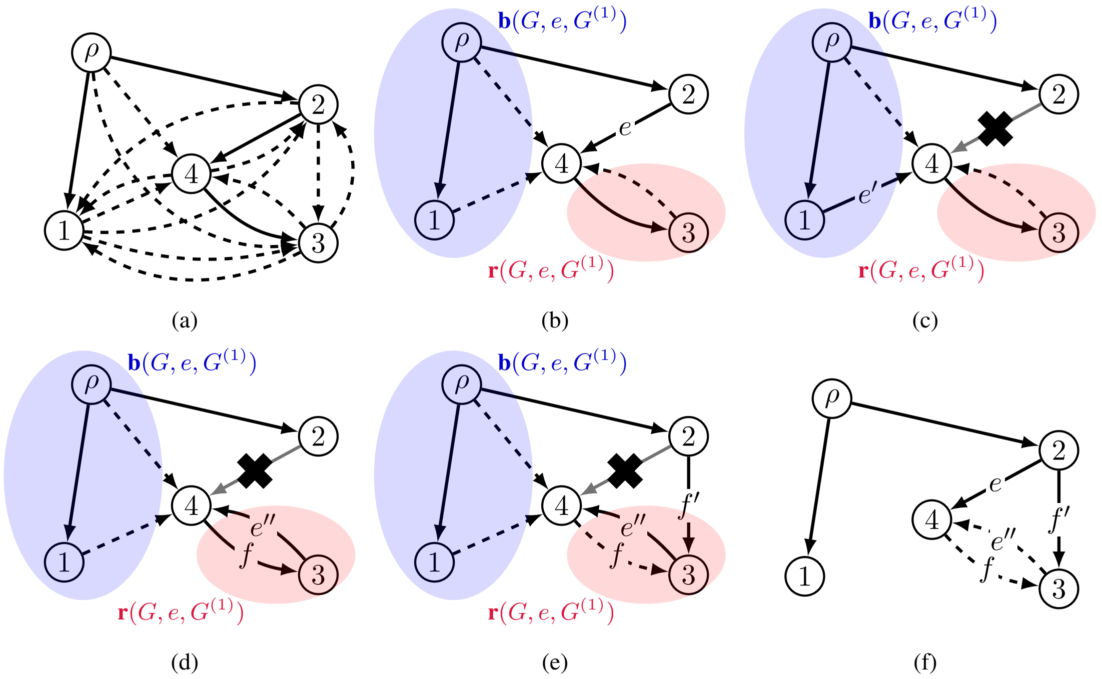

Figure 4: Worked example of Lemma 1. Consider a fully connected graph, G, of the example given in Fig. 2 as given in (a). Suppose that the solid edges in (a) represent $\overrightarrow{G}$. Therefore, $G^{(1)} = \overrightarrow{G}$. Next, suppose that we know that $e = (2 \rightarrow 4) \in G^{(1)}$ is not in $G^{(2)}$. Then one of the dashed edges in (b) must be in $G^{(2)}$ as ④ must have an incoming edge. The edges emanating from ⓟ and ① make up the set of blue edges, $\mathbf{b}(G, e, G^{(1)})$ while the edge emanating from ③ makes the set of red edges, $\mathbf{r}(G, e, G^{(1)})$. If $e' \in \mathbf{b}(G, e, G^{(1)})$ is in $G^{(2)}$ as in (c), then the solid lines in (c) make a tree and $G^{(2)}$ differs from $G^{(1)}$ by exactly one blue edge of e. Otherwise, we know that $e'' \in \mathbf{r}(G, e, G^{(1)})$ is in $G^{(2)}$ as in (d). However, the solid edges in (d) contain a cycle between ③ and ④ with edges e'' and f. We could break the cycle at ③ and include edge f' in our tree as in (e). However, while the solid edges in (e) make a valid tree, as $w(e) > w(e'')$ and $w(f) > w(f')$, the tree given by the solid lines of (f) will have a higher weight. This would mean that $e \in G^{(2)}$ which leads to a contradiction. Therefore, we must break the cycle at ④, which leads us to a tree as in (c). Consequently, $G^{(2)}$ will differ from $G^{(1)}$ by exactly one blue edge of e.

for the remainder of the paper.

3 Finding the Second Best Tree

In the following two sections, we provide a simplified reformulation of Camerini et al. (1980) to find the K-best trees. The simplifications additionally provide a constant time speed-up over Camerini et al. (1980)'s algorithm. We discuss the differences throughout our exposition.

The underlying concept behind finding the K-best tree, is that $G^{(K)}$ is the second best tree $G'^{(2)}$ of some subgraph $G' \subseteq G$. In order to explore the space of subgraphs, we introduce the concept of edge inclusion and exclusion graphs.

Definition 1 (Edge inclusion and exclusion)**.** For any graph G and edge $e \in G$, the **edge-inclusion graph** $G + e \subset G$ is the graph such that for any $d \in \mathcal{A}(G + e)$, $e \in d$. Similarly, the **edge-exclusion graph** $G - e \subset G$ is the graph such that for any $d \in \mathcal{A}(G - e)$, $e \notin d$.

When we discuss finding the K-best dependency trees in §5, we implicitly change the above definition to use $\mathcal{D}(G + e)$ and $\mathcal{D}(G - e)$ instead of $\mathcal{A}(G + e)$ and $\mathcal{A}(G - e)$ respectively.

In this section, we will specifically focus on finding $G^{(2)}$, we extend this to finding the $G^{(k)}$ in §4. Finding $G^{(2)}$ relies on the following fundamental theorem.

Theorem 1. *For any graph G and* $e \in G^{(1)}$

$$G^{(2)} = (G - e)^{(1)} \tag{6}$$

where

$$e = \underset{e' \in G^{(1)}}{\text{argmax}}\ w\Big((G - e')^{(1)}\Big) \tag{7}$$

Theorem 1 states that we can find $G^{(2)}$ by identifying an edge $e \in G^{(1)}$ such that $G^{(2)} = (G - e)^{(1)}$. We next show an efficient method for identifying this edge, as well as the weight of $G^{(2)}$ without actually having to find $G^{(2)}$.

Definition 2 (Blue and red edges)**.** For any graph

```
1:  def next($G$) :
2:      if $\overrightarrow{G}$ has a cycle $C$ :                    ▷ Recursive case
3:          $d, \langle w, e\rangle \leftarrow$ next$(G_{/C})$
4:          $d' \leftarrow d \looparrowright C$
5:          $e' \leftarrow \operatorname{argmin}_{e'' \in C \cap d'} \overline{w}_{G,d'}(e'')$
6:          $w' \leftarrow w(d') - \overline{w}_{G,d'}(e)$
7:          return $d', \max(\langle w, \pi(e)\rangle, \langle w', e'\rangle)$
8:      else                                                   ▷ Base case
9:          $e \leftarrow \operatorname{argmin}_{e' \in \overrightarrow{G}} \overline{w}_G(e')$
10:         $w \leftarrow w\left(\overrightarrow{G}\right) - \overline{w}_G(e)$
11:         return $\overrightarrow{G}, \langle w, e\rangle$
```

Figure 5: Algorithm for finding $G^{(1)}$, the best edge e to delete to find $G^{(2)}$, and $w(G^{(2)})$.

G, tree $d \in \mathcal{A}(G)$, and edge $e = (i \rightarrow j) \in d$, the set of **blue edges** $\mathbf{b}(G, e, d)$ and **red edges** $\mathbf{r}(G, e, d)$ are defined by[12]

$$\mathbf{b}(G, e, d) \overset{\text{def}}{=} \{e' = (i' \rightarrow j) \mid w(e') \leq w(e), \; d \setminus \{e\} \cup \{e'\} \in \mathcal{A}(G)\} \tag{2}$$

$$\mathbf{r}(G, e, d) \overset{\text{def}}{=} \{e' = (i' \rightarrow j) \mid e' \notin \mathbf{b}(G, e, d)\} \tag{3}$$

An example of blue and red edges are given in Fig. 4.

Lemma 1. *For any graph G, if $G^{(1)} = \overrightarrow{G}$, then for some $e \in G^{(1)}$ and $e' \in \mathbf{b}(G, e, G^{(1)})$*

$$G^{(2)} = G^{(1)} \setminus \{e\} \cup \{e'\} \tag{8}$$

Lemma 1 can be understood more clearly by following the worked example in Fig. 4. The moral of Lemma 1 is that in the base case where there are no critical cycles, we only need to examine the blue edges of the greedy graph to find the second best tree. Furthermore, our second best tree will only differ from our best tree by exactly one blue edge. Camerini et al. (1980) make use of the concepts of the blue and red edge sets, but rather than consider a base case as Lemma 1, they propose an ordering in which to visit the edges of the graph. This results in several properties about the possible orderings, requiring much more complicated proofs.

[12]We can also define $\mathbf{b}(G, e, d)$ as $(i' \rightarrow j) \in \mathbf{b}(G, e, d) \iff i'$ is an ancestor of j in d and $\mathbf{r}(G, e, d)$ as $(i' \rightarrow j) \in \mathbf{r}(G, e, d) \iff i'$ is a descendant of j in d. This equivalence exists as we can only swap an incoming edge to j in d without introducing a cycle if the new edge emanates from an ancestor of j. The exposition using ancestors and descendants is more similar to the exposition originally presented by Camerini et al. (1980).

Definition 3 (Swap cost)**.** For any graph G, tree $d \in \mathcal{A}(G)$, and edge $e \in d$, the **swap cost** denotes the minimum change to a tree weight to replace e by a single edge in d. It is given by

$$\overline{w}_{G,d}(e) = \min_{e' \in \mathbf{b}(G,e,d)} \left(w(e) - w(e')\right) \tag{4}$$

We will shorthand $\overline{w}_G(e)$ to mean $\overline{w}_{G,G^{(1)}}(e)$.

Corollary 1. *For any graph G, if $G^{(1)} = \overrightarrow{G}$, then $G^{(2)} = (G - e)^{(1)}$ where e is given by*

$$e = \operatorname*{argmin}_{e' \in G^{(1)}} \overline{w}_G(e') \tag{5}$$

Furthermore, $w(G^{(2)}) = w(G^{(1)}) - \overline{w}_G(e)$.

Corollary 1 provides us a procedure for finding the best edge to remove to find $G^{(2)}$ as well as its weight in the base case of G having no critical cycles. We next illustrate what must be done in the recursive case when a critical cycle exists.

Lemma 2. *For any G with a critical cycle C, either $G^{(2)} = (G_{/C})^{(2)} \looparrowright C$ (with $w(G^{(2)}) = w((G_{/C})^{(2)})$) or $G^{(2)} = (G - e)^{(1)}$ (with $w(G^{(2)}) = w(G^{(1)}) - \overline{w}_G(e)$) for some $e \in C \cap G^{(1)}$.*

Combining Corollary 1 and Lemma 2, we can directly modify opt to find the weight of $G^{(2)}$ and the edge we must remove to obtain it. We detail this algorithm as next in Fig. 5.

Theorem 2. *For any graph G, executing* next(G) *returns $G^{(1)}$ and $\langle w, e\rangle$ such that $G^{(2)} = (G - e)^{(1)}$ and $w(G^{(2)}) = w$.*

Runtime analysis. We know that without lines 5, 6, 9 and 10, next is identical to opt and so will run in $\mathcal{O}(N^2)$. We call $\overline{w}$ at most $N + 2$ times during a full call of next: N times from lines 5 and 9 combined, once from Line 6, and once from Line 10. To find $\overline{w}$, we first need to find the set of blue edges, which can be done in $\mathcal{O}(N)$ by computing the reachability graph. Then, we need another $\mathcal{O}(N)$ to find the minimising value. Therefore, next does $\mathcal{O}(N^2)$ extra work than opt and so retains the runtime of $\mathcal{O}(N^2)$. Camerini et al. (1980) require $G^{(1)}$ to be known ahead of time. This results in having to run the original algorithm in $\mathcal{O}(N^2)$ time and then having to do the same amount of work as next because they must still contract the graph. Therefore, next has a constant-time speed-up over its counterpart in Camerini et al. (1979).

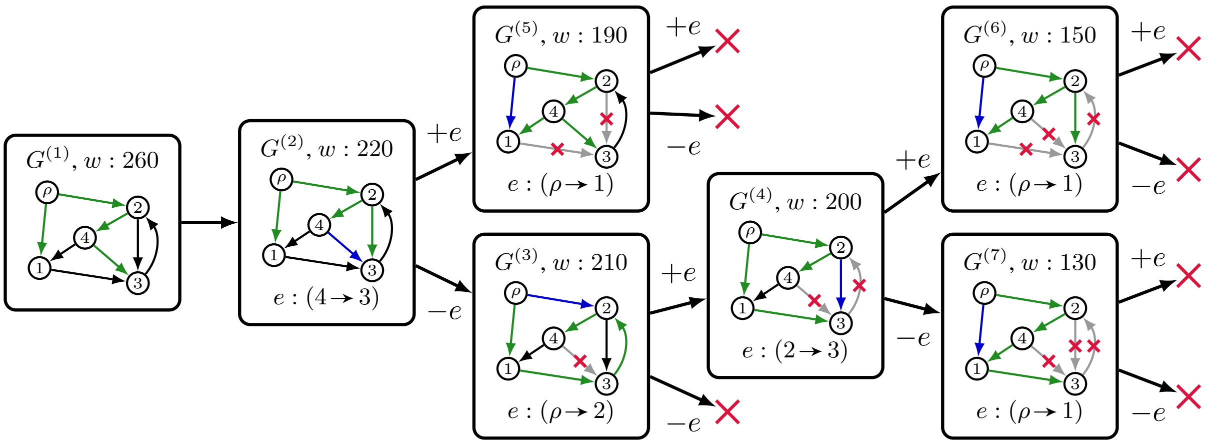

Figure 6: Example of running through `kbest` using the graph of Fig. 2. We start with $G^{(1)}$ that has a weight of 260 and consider the best edge to remove to find $G^{(2)}$. Using `next` we find that $G^{(2)} = (G - e)^{(1)}$ for $e = (4 \to 3)$. We then know that either $e \in G^{(3)}$ or $e \notin G^{(3)}$. We can push these two possibilities to the queue using two calls to `next`. We find that $G^{(3)}$ comes from the graph without e, and also removes the edge $e' = (\rho \to 2)$. We attempt to push two new elements to the queue, but we see that only by including e' in the graph can we find another tree. We repeat this process until we have found $G^{(K)}$ or the queue is empty.

```
1:  def kbest$(G, K)$ :
2:    $\langle G^{(1)}, \langle w, e\rangle\rangle \leftarrow$ next$(G)$
3:    yield $G^{(1)}$
4:    $Q \leftarrow$ priority_queue$([\langle w, e, G\rangle])$
5:    for $k = 2, \ldots, K$ :
6:      if $Q$.empty() : return
7:      $\langle w, e, G'\rangle \leftarrow Q$.pop()
8:      $\langle G^{(k)}, \langle w', e'\rangle\rangle \leftarrow$ next$(G' - e)$
9:      yield $G^{(k)}$
10:     $Q$.push$(\langle w', e', G' - e\rangle)$
11:     $\langle \cdot, \langle w'', e''\rangle\rangle \leftarrow$ next$(G' + e)$
12:     $Q$.push$(\langle w'', e'', G' + e\rangle)$
```

Figure 7: K-best tree enumeration algorithm.

4 Finding the K^{th} Best Tree

In the previous section, we found an efficient method for finding $G^{(2)}$. We now utilize this method to efficiently find the K-best trees.

Lemma 3. *For any graph G and $K > 1$, there exists a subgraph $G' \subseteq G$ and $1 \leq l < K$ such that $G^{(l)} = G'^{(1)}$ and $G^{(K)} = G'^{(2)}$.*

Lemma 3 suggests that we can find the K-best trees by only examining the second best trees of subgraphs of G. This idea is formalized as algorithm `kbest` in Fig. 7. A walk-through of the exploration space using `kbest` for our example graph in Fig. 2 is shown in Fig. 6.

Theorem 3. *For any graph G and $K > 0$, at any iteration $1 \leq k \leq K$, `kbest`(G, K) returns $G^{(k)}$.*

Runtime analysis. We call `next` once at the start of the algorithm, then every subsequent iteration we make two calls to `next`. As we have $K-1$ iterations , the runtime of `kbest` is $\mathcal{O}(KN^2)$. The first call to `next` in each iteration finds the K^{th} best tree as well as an edge to remove. Camerini et al. (1980) make one call to of `opt` and two calls to `next` which only finds the weight-edge pair of our algorithm. Therefore, `kbest` has a constant time speed-up on the original algorithm.[13]

	$K = 10$	$K = 20$	$K = 50$
Camerini et al.	6.95	14.04	35.11
`kbest`	4.89	10.10	25.63
Speed-up	1.42×	1.39×	1.37×

Table 1: Runtime experiment for parsing the K-best spanning trees in the English UD test set (Nivre et al., 2018). Times are given in 10^{-2} seconds for the average parse of the K-best spanning trees.

A short experiment. We empirically measure the constant time speed-up between `kbest` and the original algorithm of Camerini et al. (1980). We take the English UD test set (as used for Fig. 1) and find the 10, 20, and 50 best spanning trees using both algorithms.[14] We give the results of the experiment in Tab. 1.[15] We note that on average `kbest` leads to a 1.39 times speed-up. This is

[13]In practice, we maintain a set of edges to include and exclude to save space.

[14]Implementations for both versions can be found in our code release (see footnote 1)

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

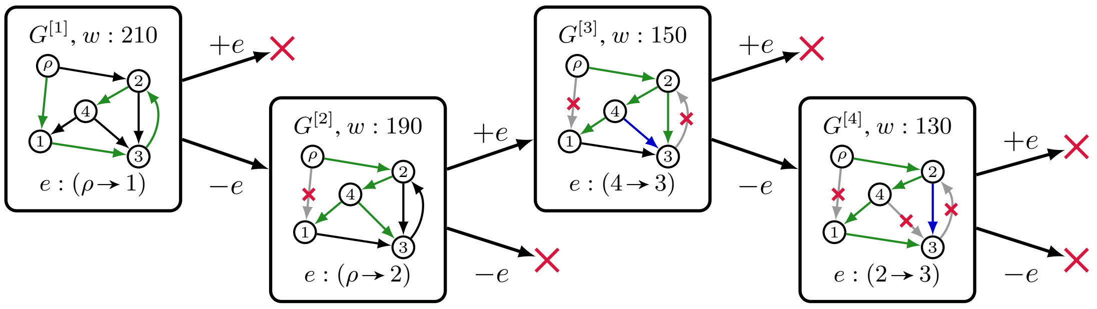

Figure 8: Example of running through `kbest_dep` using the graph of Fig. 2. We start with $G^{[1]}$ that has a weight of 210 and consider the best edge to remove to find $G^{(2)}$. We consider removing the best dependency tree with the same edge emanating from the root $e = (\rho \rightarrow 1)$ using `next`. However, no such dependency tree exists, and so we only need to push the graph $G - e$. When we next pop from the queue, we see that we have removed root edge e, and so must consider removing the new root edge $e' = (\rho \rightarrow e)$. In this case, no dependency tree exists without e and e', and so we only push to the queue the results of running `next`. We repeat this process until we have found $G^{[K]}$ or the queue is empty.

lower than we anticipated as we have to make half as many calls to `next` than the original algorithm. However, in the original `next` of Camerini et al. (1980), we do not require to stitch together the tree, which may explain the slightly smaller speed-up.

5 Finding the K^{th} Best *Dependency* Tree

In this section, we present a novel extension to the algorithm presented thus far, that allows us to efficiently find the K-best dependency trees. Recall that we consider dependency trees to be spanning trees *with a root constraint* such that only one edge may emanate from ρ. Naïvely, we can use `kbest` where we initialize the queue with $(G + e_\rho)^{(1)}$ for each $e_\rho = (\rho \rightarrow j) \in G$. However, this adds a $\mathcal{O}(N^3)$ component to our runtime as we have to call `opt` N times. Instead, our algorithm maintains the $\mathcal{O}(KN^2)$ runtime as the regular K-best algorithm. We begin by noting that we can find second best dependency tree, by finding either the best dependency tree with a different root edge or the second best tree with the same root edge.

Lemma 4. *For any graph G and edge $e_\rho = (\rho \rightarrow j) \in G^{[1]}$, $G^{[2]} = (G - e_\rho)^{[1]}$ or $G^{[2]} = (G + e_\rho)^{[2]}$.*

Lemma 5. *For any graph G and $K > 1$, if $e = (\rho \rightarrow j) \in G^{[K]}$, then either e is not in any of the $K-1$-best trees or there exists a subgraph $G' \subseteq G$ and $1 \leq l < K$ such that $G^{[l]} = G'^{[1]}$, $e \in G'^{[1]}$ and $G^{[K]} = G'^{[2]}$.*

Lemma 5 suggests that we can find the K-best dependency trees, by examining the second best dependency trees of subgraphs of G or finding the best dependency tree with a unique root edge. This idea is formalized as algorithm `kbest_dep` in Fig. 9. A walk-through of the exploration space using `kbest_dep` for our example graph in Fig. 2 is shown in Fig. 8.

```
1:  def kbest_dep($G, K$) :
2:     $G^{[1]} \leftarrow$ opt($G$)
3:     yield $G^{[1]}$
4:     $e_\rho \leftarrow$ outgoing edge from $\rho$ in $G^{[1]}$
5:     $\langle \cdot, \langle w, e \rangle \rangle \leftarrow$ next($G + e_\rho$)
6:     $d \leftarrow$ opt($G - e_\rho$)
7:     $Q \leftarrow$ priority_queue($[\langle w(d), e_\rho, G \rangle]$)
8:     $Q$.push($\langle w, e, G + e_\rho \rangle$)
9:     for $k = 2, \ldots, K$ :
10:       if $Q$.empty() : return
11:       $\langle w, e, G' \rangle \leftarrow Q$.pop()
12:       if $e$ does not emanate from $\rho$ :
13:          $G^{[k]}, \langle w', e' \rangle \leftarrow$ next($G' - e$)
14:          $Q$.push($\langle w', e', G' - e \rangle$)
15:          $\langle \cdot, \langle w'', e'' \rangle \rangle \leftarrow$ next($G' + e$)
16:          $Q$.push($\langle w'', e'', G' + e \rangle$)
17:       else
18:          $G^{[k]} \leftarrow$ opt($G'$)
19:          $e_\rho \leftarrow$ outgoing edge from $\rho$ in $G^{[k]}$
20:          $d \leftarrow$ opt($G' - e_\rho$)
21:          $Q$.push($\langle w(d), e_\rho, G' - e \rangle$)
22:          $\langle \cdot, \langle w', e' \rangle \rangle \leftarrow$ next($G' + e_\rho$)
23:          $Q$.push($\langle w', e', G + e_\rho \rangle$)
24:       yield $G^{(k)}$
```

Figure 9: K-best dependency tree enumeration algorithm.

Theorem 4. *For any graph G and $K \geq 1$, at iteration $1 \leq k \leq K$, `kbest_dep`(G, K) returns $G^{[k]}$.*

Runtime analysis. At the start of the algorithm, we call opt twice and next once. Then, at each iteration we either make two calls two next, or two calls to opt and one call to next. As both algorithms have a runtime of $\mathcal{O}(N^2)$, each iteration has a runtime of $\mathcal{O}(N^2)$. Therefore, running K iterations gives a runtime of $\mathcal{O}(KN^2)$.

6 Conclusion

In this paper, we provided a simplification to Camerini et al. (1980)'s $\mathcal{O}(KN^2)$ K-best spanning trees algorithm. Furthermore, we provided a novel extension to the algorithm that decodes the K-best dependency trees in $\mathcal{O}(KN^2)$. We motivated the need for this new algorithm as using regular K-best decoding yields up to 36% trees which violation the root constraint. This is a substantial (up to 44 times) increase in the violation rate from decoding the one-best tree, and thus such an algorithm is even more important than in the one-best case. We hope that this paper encourages future research in K-best dependency parsing.

Acknowledgments

We would like to thank the reviewers for their valuable feedback and suggestions to improve this 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.

Ethical Concerns

We do not foresee how the more efficient algorithms presented this work exacerbate any existing ethical concerns with NLP systems.

References

Željko Agić. 2012. K-best spanning tree dependency parsing with verb valency lexicon reranking. In *Proceedings of COLING*.

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.

Paolo M. Camerini, Luigi Fratta, and Francesco Maffioli. 1979. A note on finding optimum branchings. *Networks*, 9.

Paolo M. Camerini, Luigi Fratta, and Francesco Maffioli. 1980. The k best spanning arborescences of a network. *Networks*, 10.

Michael Collins and Terry Koo. 2005. Discriminative reranking for natural language parsing. *Computational Linguistics*, 31.

Gonçalo M. Correia, Vlad Niculae, Wilker Aziz, and André F. T. Martins. 2020. Efficient marginalization of discrete and structured latent variables via sparsity. In *Advances in Neural Information Processing Systems: Annual Conference on Neural Information Processing Systems*.

Bich-Ngoc Do and Ines Rehbein. 2020. Neural reranking for dependency parsing: An evaluation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Timothy Dozat and Christopher D. Manning. 2017. Deep biaffine attention for neural dependency parsing. In *Proceedings of the 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.

Leonidas Georgiadis. 2003. Arborescence optimization problems solvable by Edmonds' algorithm. *Theoretical Computer Science*, 301.

Keith Hall. 2007. K-best spanning tree parsing. In *Proceedings of the Annual Meeting of the Association of Computational Linguistics*.

Keith Hall, Jiří Havelka, and David A. Smith. 2007. Log-linear models of non-projective trees, k-best MST parsing and tree-ranking. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*.

Liang Huang and David Chiang. 2005. Better k-best parsing. In *Proceedings of the International Workshop on Parsing Technology*.

Kenton Lee, Mike Lewis, and Luke Zettlemoyer. 2016. Global neural CCG parsing with optimality guarantees. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Xuezhe Ma and Eduard Hovy. 2017. Neural probabilistic model for non-projective MST parsing. In *Proceedings of the International Joint Conference on Natural Language Processing*.

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*.

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 H`ông, 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 Nguy˜ên Thị, Huy`ên Nguy˜ên Thị Minh, Vitaly Nikolaev, Rattima Nitisaroj, Hanna Nurmi, Stina Ojala, Adédayọ 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.

Adam Pauls and Dan Klein. 2009. K-best A* parsing. In *Proceedings of the Joint Conference of the Annual Meeting of the ACL and the International Joint Conference on Natural Language Processing of the AFNLP*.

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 Association for Computational Linguistics: System Demonstrations*.

Federico Sangati, Willem Zuidema, and Rens Bod. 2009. A generative re-ranking model for dependency parsing. In *Proceedings of the International Conference on Parsing Technologies*.

Libin Shen, Anoop Sarkar, and Franz Josef Och. 2004. Discriminative reranking for machine translation. In *Proceedings of the Human Language Technology*

Conference of the North American Chapter of the Association for Computational Linguistics.

Robert Endre Tarjan. 1977. Finding optimum branchings. *Networks*, 7.

Hui Zhang, Min Zhang, Chew Lim Tan, and Haizhou Li. 2009. K-best combination of syntactic parsers. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Chenxi Zhu, Xipeng Qiu, Xinchi Chen, and Xuanjing Huang. 2015. A re-ranking model for dependency parser with recursive convolutional neural network. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing*, volume 1.

Ran Zmigrod, Tim Vieira, and Ryan Cotterell. 2020. Please mind the root: Decoding arborescences for dependency parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

A Supplementary Materials for Section 1 (Introduction)

Results Table for Fig. 1

Language	**|Train|**	**|Test|**	**Root Constraint Violation Rate** (%) $K = 1$	$K = 5$	$K = 10$	$K = 20$	$K = 50$
Czech	68495	10148	0.45	5.07	6.18	6.76	7.67
Russian	48814	6491	0.49	5.07	6.58	7.66	8.99
Estonian	24633	3214	0.93	5.59	7.02	8.24	9.42
Korean	23010	2287	0.96	6.68	9.51	11.91	14.74
Latin	16809	2101	0.52	5.17	5.57	6.25	7.62
Norwegian	15696	1939	0.52	4.26	5.20	6.22	7.38
Ancient Greek	15014	1047	0.57	4.74	7.00	8.38	10.69
French	14450	416	1.68	3.85	4.95	5.81	6.98
Spanish	14305	1721	0.17	2.25	3.25	3.96	4.89
Old French	13909	1927	0.52	6.81	9.41	11.38	13.01
German	13814	977	1.54	5.12	6.37	7.63	9.06
Polish	13774	1727	0.00	4.76	7.86	10.11	13.00
Hindi	13304	1684	0.18	1.34	2.19	2.98	4.04
Catalan	13123	1846	0.54	2.32	2.97	3.68	4.51
Italian	13121	482	0.21	4.02	5.66	7.25	9.19
English	12543	2077	0.48	9.12	10.73	11.12	11.34
Dutch	12264	596	0.67	3.39	4.18	4.82	5.59
Finnish	12217	1555	0.39	4.72	6.12	7.39	9.15
Classical Chinese	11004	2073	0.96	22.52	25.95	28.09	29.91
Latvian	10156	1823	0.88	7.05	8.77	9.95	11.31
Bulgarian	8907	1116	0.27	4.66	6.73	8.16	10.29
Slovak	8483	1061	0.38	4.81	5.34	5.29	5.29
Portuguese	8328	477	0.42	3.31	4.15	4.76	5.75
Romanian	8043	729	0.41	1.26	1.66	2.16	2.81
Japanese	7125	550	0.00	5.13	6.24	7.20	8.79
Croatian	6914	1136	0.88	2.90	3.71	4.44	5.62
Slovenian	6478	788	0.38	2.66	3.53	4.59	5.79
Arabic	6075	680	0.29	3.79	4.15	4.72	5.27
Ukrainian	5496	892	0.90	7.49	9.15	10.13	11.72
Basque	5396	1799	0.67	3.64	5.06	6.67	8.71
Hebrew	5241	491	1.02	2.81	4.01	5.04	5.90
Persian	4798	600	0.67	2.43	3.47	4.28	5.25
Indonesian	4477	557	1.26	4.06	5.48	6.65	8.25
Danish	4383	565	0.53	4.35	5.59	6.35	7.45
Swedish	4303	1219	1.23	4.63	6.08	7.09	8.73
Old Church Slavonic	4124	1141	1.05	14.32	17.64	19.88	22.05
Urdu	4043	535	1.12	2.47	3.08	3.60	4.39
Chinese	3997	500	1.80	4.80	5.90	7.68	9.31
Turkish	3664	983	2.54	12.47	15.53	17.09	18.73
Gothic	3387	1029	0.78	8.65	11.18	13.10	14.86
Serbian	3328	520	0.19	2.04	2.60	3.16	4.23
Galician	2272	861	1.16	2.07	2.36	2.88	3.46
North Sami	2257	865	1.27	7.49	10.15	12.43	15.54
Armenian	1975	278	0.00	7.34	8.42	9.64	10.81
Greek	1662	456	0.44	3.20	4.19	4.80	5.82
Uyghur	1656	900	0.56	7.18	9.64	12.24	15.57
Vietnamese	1400	800	3.38	6.78	8.25	9.56	11.39
Afrikaans	1315	425	6.35	13.65	14.73	16.12	18.26
Wolof	1188	470	1.49	6.89	8.32	9.91	12.17
Maltese	1123	518	0.58	5.17	6.70	8.12	9.73
Telugu	1051	146	0.00	27.81	32.81	36.16	36.99
Scottish Gaelic	1015	536	0.75	7.16	8.97	10.20	11.75
Hungarian	910	449	4.23	7.44	8.66	9.82	10.75
Irish	858	454	2.42	7.14	8.68	10.23	11.73
Tamil	400	120	0.00	1.17	1.50	1.83	3.05
Marathi	373	47	2.13	20.85	21.70	27.34	33.36
Belarusian	319	253	0.79	5.61	9.05	8.99	7.27
Lithuanian	153	55	7.27	9.82	10.36	10.82	12.47
Kazakh	31	1047	2.58	7.97	10.68	13.45	17.41
Upper Sorbian	23	623	6.42	9.34	10.72	11.78	13.45
Kurmanji	20	734	23.57	27.06	29.22	30.87	33.33
Buryat	19	908	6.61	10.37	13.00	15.48	19.13
Livvi	19	106	12.26	14.15	15.00	15.99	17.68

B Supplementary Materials for Section 3 (Finding the Second Best Tree)

Theorem 1. *For any graph G and $e \in G^{(1)}$*

$$G^{(2)} = (G - e)^{(1)} \tag{6}$$

where

$$e = \underset{e' \in G^{(1)}}{\operatorname{argmax}}\, w\Big((G - e')^{(1)}\Big) \tag{7}$$

Proof. There must be at least one edge $e \in G^{(1)}$ such that $e \notin G^{(2)}$. Therefore, there exists an $e \in G^{(1)}$ such that $G^{(2)} = (G - e)^{(1)}$. Now suppose by way of contradiction that e is not as given in (7). If we choose an e' that satisfies (7), then by definition $w\big((G - e')^{(1)}\big) > w\big((G - e)^{(1)}\big)$. As $(G - e')^{(1)} \neq G^{(1)}$, we arrive at a contradiction. ∎

Lemma 1. *For any graph G, if $G^{(1)} = \overrightarrow{G}$, then for some $e \in G^{(1)}$ and $e' \in \boldsymbol{b}(G, e, G^{(1)})$*

$$G^{(2)} = G^{(1)} \setminus \{e\} \cup \{e'\} \tag{8}$$

Proof. By Theorem 1, we have $G^{(2)} = (G - e)^{(1)}$ where $e = (i \rightarrow j)$ is chosen according to (7). Consider the graph $G - e$; we have that $\overrightarrow{G - e} = G^{(1)} \setminus \{e\} \cup \{e'\}$ where e' is the second best incoming edge to j in G by the definition of the greedy graph.

1. *Case $e' \in \boldsymbol{b}(G, e, G^{(1)})$:* Then $\overrightarrow{G - e}$ is a tree and $(G - e)^{(1)} = \overrightarrow{G - e}$.

2. *Case $e' \in \boldsymbol{r}(G, e, G^{(1)})$:* Then, $\overrightarrow{G - e}$ has a cycle C by construction. Since this is a greedy graph, cycle C is critical. In the expansion phase of the 1-best algorithm, we will break the cycle C.

   (a) *Case break C at j:* Then, $e' \notin (G - e)^{(1)}$ and we must choose an edge $e'' = (i' \rightarrow j)$ to be in $(G - e)^{(1)}$. We require that $e'' \in \mathbf{b}(G, e, G^{(1)})$ as we would otherwise re-introduce a cycle in the expansion phase, which is not possible. Therefore, $G^{(2)} = G^{(1)} \setminus \{e\} \cup \{e''\}$.

   (b) *Case break C at $j' \neq j$:* Then, there exists an edge $f = (i'' \rightarrow j') \in C$ (and in $G^{(1)}$) which is not in $G^{(2)}$. Instead, we choose $f' = (i' \rightarrow j')$ to be in $G^{(2)}$. Therefore, $G^{(2)} = G^{(1)} \setminus \{e, f\} \cup \{e', f'\}$. However, it is not possible for f' and e to form a cycle and so $d = G^{(1)} \setminus \{f\} \cup \{f'\} \in \mathcal{A}(G)$ and $w(d) > w\big(G^{(2)}\big)$. This is a contradiction as only $w\big(G^{(1)}\big) > w\big(G^{(2)}\big)$.

∎

Lemma 2. *For any G with a critical cycle C, either $G^{(2)} = (G_{/C})^{(2)} \looparrowright C$ (with $w\big(G^{(2)}\big) = w\big((G_{/C})^{(2)}\big)$) or $G^{(2)} = (G - e)^{(1)}$ (with $w\big(G^{(2)}\big) = w\big(G^{(1)}\big) - \overline{w}_G(e)$) for some $e \in C \cap G^{(1)}$.*

Proof. It must be that $G^{(2)} = (G_{/C})^{(2)} \looparrowright C$ or $G^{(2)} \neq (G_{/C})^{(2)} \looparrowright C$.

1. *Case $G^{(2)} = (G_{/C})^{(2)} \looparrowright C$:* Since the weight of a tree is preserved during expansion, we are done.

2. *Case $G^{(2)} \neq (G_{/C})^{(2)} \looparrowright C$:* Then, for all $e' \in (G_{/C})^{(1)}$, $\pi(e') \in G^{(2)}$. Therefore, if j is the entrance site of C in $(G_{/C})^{(1)}$, $G^{(2)} = \pi((G_{/C})^{(1)}) \cup C_j^{(2)}$. As $C_j^{(1)} = \overrightarrow{C_j}$, by Corollary 1, $C_j^{(2)} = (C_j - e)^{(1)}$ for $e \in C_j^{(1)}$ and $w\Big(C_j^{(2)}\Big) = w\Big(C_j^{(1)}\Big) - \overline{w}_{C_j}(e)$. Thus, $G^{(2)} = (G - e)^{(1)}$ where $e \in C \cap d$ and $w\big(G^{(2)}\big) = w\big(G^{(1)}\big) - \overline{w}_G(e)$.

∎

Theorem 2. *For any graph G, executing $\texttt{next}(G)$ returns $G^{(1)}$ and $\langle w, e\rangle$ such that $G^{(2)} = (G - e)^{(1)}$ and $w\big(G^{(2)}\big) = w$.*

Proof. `next`(G) returns $G^{(1)}$ by the correctness of `opt`. We prove that w, e satisfy the above conditions.

1. *Case* $G^{(1)} = \overrightarrow{G}$*:* Then, by Corollary 1 we can find the best edge to remove and the weight of $G^{(2)}$.
2. *Case* $G^{(1)} \neq \overrightarrow{G}$*:* Then, G has a critical cycle C. By Lemma 2, we can either recursively call `next`$(G_{/C})$ or examine the edges in $C \cap G^{(1)}$ to find the best edge to remove and the weight of $G^{(2)}$.

■

C Supplementary Materials for Section 4 (Finding the K^{th} Best Tree)

Lemma 3. *For any graph G and* $K > 1$*, there exists a subgraph* $G' \subseteq G$ *and* $1 \leq l < K$ *such that* $G^{(l)} = G'^{(1)}$ *and* $G^{(K)} = G'^{(2)}$*.*

Proof. There must exist some subgraph $G' \subseteq G$ such that $G^{(K)} = G'^{(2)}$. Suppose by way of contradiction that there does not exist an $l < K$ such that $G^{(l)} = G'^{(1)}$. However, since $w\big(G'^{(1)}\big) > w\big(G^{(K+1)}\big)$, $G'^{(1)}$ must be in the K-highest weighted trees. Therefore, there must exist an l such that $G^{(l)} = G'^{(1)}$ ■

Theorem 3. *For any graph G and* $K > 0$*, at any iteration* $1 \leq k \leq K$*,* `kbest`(G, K) *returns* $G^{(k)}$*.*

Proof. We prove this by induction on k.
Base Case: Then, $k = 1$ and $G^{(1)}$ is returned by Theorem 2.
Inductive Step: Assume that for all $l \leq k$, at iteration l, $G^{(l)}$ is returned. Now consider iteration $k + 1$, by Lemma 3, we know that $G^{(k+1)} = G'^{(2)}$ where $G'^{(1)} = G^{(l)}$ for some $l \leq k$. By the induction hypothesis, $G^{(l)}$ is returned at the l^{th} iteration, and by Theorem 2, we have pushed $G'^{(2)}$ onto the queue. Therefore, we will return $G^{(k+1)}$. ■

D Supplementary Materials for Section 5 (Finding the K^{th} Best *Dependency* Tree)

Lemma 4. *For any graph G and edge* $e_\rho = (\rho \rightarrow j) \in G^{[1]}$*,* $G^{[2]} = (G - e_\rho)^{[1]}$ *or* $G^{[2]} = (G + e_\rho)^{[2]}$*.*

Proof. If $e_\rho \notin G^{[2]}$, then clearly $G^{[2]} = (G - e_\rho)^{[1]}$. Otherwise, $e_\rho \in G^{[2]}$. As $e_\rho \in G^{[1]}$, $G^{[2]} = (G + e_\rho)^{[2]}$. ■

Lemma 5. *For any graph G and* $K > 1$*, if* $e = (\rho \rightarrow j) \in G^{[K]}$*, then either e is not in any of the* $K-1$*-best trees or there exists a subgraph* $G' \subseteq G$ *and* $1 \leq l < K$ *such that* $G^{[l]} = G'^{[1]}$*,* $e \in G'^{[1]}$ *and* $G^{[K]} = G'^{[2]}$*.*

Proof. It must be that either there exists an $1 \leq l < K$ such that $e \in G^{[l]}$ (Case 1) or no such l exists (Case 2).

1. Consider the graph $G + e$. Under our definition of edge-inclusion graphs for dependency trees, $\mathcal{A}(G + e) = \mathcal{D}(G + e)$. Then, by Lemma 3, there exists a l' and G' such that $(G^{[l']} = G'^{[1]}$ and $G^{[K]} = G'^{[2]}$.
2. Then, e is not in any of the $(K-1)$-best trees.

■

Theorem 4. *For any graph G and* $K \geq 1$*, at iteration* $1 \leq k \leq K$*,* `kbest_dep`(G, K) *returns* $G^{[k]}$*.*

Proof. We prove this by induction on k.
Base Case: Then, $k = 1$ and $G^{(1)}$ is returned by the correctness of `opt`.
Inductive Step: Assume that for all $l \leq k$, at iteration l, $G^{[l]}$ was returned. Now consider iteration $k + 1$, by Lemma 5, we know that either $G^{[k+1]}$ has a unique root edge to the k-best trees (Case 1) or $e = (\rho \rightarrow j)$ and there exists a G' and $l \leq k$ such that $G'^{(1)} = G^{(l)}$, $e \in G^{(l)}$, and $G^{[k+1]} = G'^{[2]}$ (Case 2).

1. There always exists a tree in the queue that has a unique root edge to all trees that came before it. Furthermore, it is the highest such tree by the correctness of `opt`.

2. By our induction hypothesis, $G^{[l]}$ is returned at the l^{th} iteration, and by Theorem 2, we have pushed $G' + e^{[2]}$ onto the queue. Therefore, we will return $G^{[k+1]}$.

■