A Root of a Problem: Optimizing Single-Root Dependency Parsing

Miloš Stanojević
School of Informatics
University of Edinburgh
m.stanojevic@ed.ac.uk

Shay B. Cohen
School of Informatics
University of Edinburgh
scohen@inf.ed.ac.uk

Abstract

We describe two approaches to single-root dependency parsing that yield significant speed ups in such parsing. One approach has been previously used in dependency parsers in practice, but remains undocumented in the parsing literature, and is considered a heuristic. We show that this approach actually finds the optimal dependency tree. The second approach relies on simple reweighting of the inference graph being input to the dependency parser and has an optimal running time. Here, we again show that this approach is fully correct and identifies the highest-scoring parse tree. Our experiments demonstrate a manyfold speed up compared to a previous graph-based state-of-the-art parser without any loss in accuracy or optimality.[1]

1 Introduction

Dependency parsing is one of the core steps in many Natural Language Processing pipelines. Given its wide and large-scale use, both in academic and commercial settings, even moderate improvements in the speed and accuracy of a dependency parser may significantly impact its utility. In this paper, we show how to improve the *speed* of graph-based dependency parsers (McDonald et al., 2005; Qi et al., 2020) without compromising at all on *accuracy*.

Graph-based dependency parsers work in two steps. The first step forms a complete weighted directed graph of words and a special *ROOT* token by computing the weights using a trained statistical model.[2] The second step then executes the main inference procedure: it identifies a directed spanning tree (often referred to as *arborescence*) in this graph, aiming to maximize its weight, and retaining *ROOT* as the root node of the arborescence.

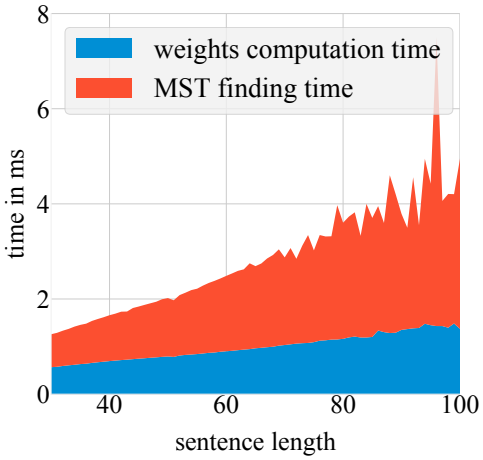


Figure 1: Time proportion used by the neural and MST component when parsing with Stanza on GPU.

While some of the previous work to optimize the speed of graph-based parsers focused on the first step (Anderson and Gómez-Rodríguez, 2020), we demonstrate in Figure 1 that most of the parsing time is actually spent on the spanning tree inference routine. As sentence length increases, the gap between the spanning tree inference time and time spent on constructing the weighted graph increases significantly.[3]

MST search is often done using the Chu-Liu-Edmonds (CLE) algorithm (Chu and Liu, 1965; Edmonds, 1967) that runs in $\mathcal{O}(n^3)$ where n is the sentence length. Tarjan (1977) presents a relatively complicated way of implementing the CLE algorithm in $\mathcal{O}(n^2)$. Tarjan's algorithm is often cited in NLP literature, but to the best of our knowledge has never been implemented for dependency parsing. This is due to the common

[1] Our code is available at https://github.com/stanojevic/Fast-MST-Algorithm.

[2] The weights of edges entering *ROOT* are $-\infty$.

[3] For this calculation, we used the Stanza parser (Qi et al., 2020), a widely-used dependency parser.

algorithm	appeared in	current implementation worst-case	claimed worst-case dense graph	average-case dense graph	claimed worst-case sparse graph
Gabow-Tarjan	Gabow and Tarjan (1984) Zmigrod et al. (2020)	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(m \log n)$
Naïve	mentioned in Zmigrod et al. (2020) and in Section 3	n/a	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(mn + n^2 \log n)$
Root Preselection	code of some parsers (undocumented) and thoroughly discussed in Section 3	$\mathcal{O}(n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}(mn + n^2 \log n)$
Reweighting	introduced in Section 4	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(m + n \log n)$

Table 1: Algorithms for single-root dependency parsing. The sentence length is denoted by n, and the number of the edges in the input graph by m.

belief that the original CLE often works well in practice (see footnote 2 in Zmigrod et al. 2020 or end of §4.2.2 in Kübler et al. 2009). We test this claim, and show that significant improvements can be made over CLE.

The (unconstrained) MST algorithm such as CLE produces a tree with one root node, namely the special token *ROOT*, but that root node may have multiple edges coming out of it. Yet, in some widely-used dependency treebanks, such as Universal Dependencies (Nivre et al., 2018), only one edge is permitted to come out of *ROOT*. We will refer to the task of finding an MST that contain only one outgoing edge out of *ROOT* as *single-root* or *constrained* MST parsing.

Zmigrod et al. (2020) provide an implementation of the non-trivial Gabow and Tarjan algorithm to compute a constrained MST with only one dependency edge coming out of *ROOT*. While both Gabow and Tarjan and Zmigrod et al. argue that this algorithm could be implemented in $\mathcal{O}(n^2)$, they do not describe or follow such an implementation. The only existing implementation of this algorithm runs in $\mathcal{O}(n^2 \log n)$, which is the best worst-case asymptotic running time tested in the literature for single-root dependency parsing.

In this paper, we provide two alternative approaches of computing the constrained MST by using an unconstrained MST algorithm as a subroutine. Both of these algorithms are very simple to implement and understand. We prove that the first one of them, *on average*, has the same asymptotic running time as the unconstrained algorithm used as a subroutine. The second algorithm has the same *worst-case* asymptotic runtime as the unconstrained algorithm which is optimal for complete graphs.

Worst-case complexity does not guarantee that an algorithm will be fast in practice (Roughgarden, 2019): the actual speed might be influenced by constant factors, memory access patterns and the difficulty of the typical input instances (Moret, 2002). This is why we test all our algorithms in a typical settings encountered in dependency parsing. Additionally, we propose a simple heuristic that recognizes if the input instance is "easy" and if so returns the correct solution even before running the full algorithm.

As a guide to this paper, the algorithms for single-root dependency parsing, both previously published ones and the ones presented in this paper, are shown in Table 1 together with their associated computational complexity. In the next section we will introduce the basic concepts from Gabow and Tarjan algorithm that Zmigrod et al. (2020) have put into practice for single-root dependency parsing. This is the only previously published work on single-root dependency parsing. Section 3 shows the *Root Preselection* algorithm for single-root MST parsing and proves its correctness and average runtime complexity. Section 4 shows even better *Reweighting* algorithm that performs well not only in average, but also in the worst-case. Section 5 introduces the *ArcMax* trick that improves practical speed of any MST parser by recognizing the "easy" cases mentioned above. Section 6 experimentally tests and verifies all of these findings.

2 The Gabow-Tarjan Algorithm

Gabow and Tarjan (1984) present an algorithm that solves a much more general combinatorial optimization problem than single-root MST parsing. Concretely, they abstract a family of optimization problems as an optimization of a *minimum weight base of a matroid*. We will not

describe here the full theory and workings of this algorithm but just present a few important points as related to the aspect of MST parsing. For a good introduction to the use of matroids for combinatorial optimization see Cormen et al. (2009, §16.4).

Many combinatorial optimization problems can be framed as a search for the minimum weight base of a matroid, a structure that consists of a set of "independent subsets" of a ground set, generalizing the notion of linear independence in vector spaces. Let us consider a minimum spanning tree problem over undirected graphs. This can be solved with a *graphic matroid*. In this matroid, the ground set contains all the edges of the graph, while independent sets contain all forests (sets of edges that do not form a cycle). A base of this matroid is a spanning tree. Finding a minimum weight base of graphic matroid is equivalent to finding a minimum spanning tree.[4]

Gabow and Tarjan extend the definition of the problem by introducing a coloring of the elements of matroid's ground set: every element can be marked as *green* or *red*. In the case of graphic matroid the coloring would be applied to the edges of the graph. Gabow and Tarjan described a matroid optimization method that finds a minimum weight base that contains exactly q red elements, given $q \in \mathbb{N}$. Let β_i stand for a set of all optimal bases with i red elements. Let *swap* (e, f) for base B stand for a pair of matroid's ground elements for which it holds that $B/\{e\}\cup\{f\}$ is also a base and that e is green and f is red. Swaps can be ranked from smallest to largest by $\mathit{weight}(f) - \mathit{weight}(e)$. Gabow and Tarjan prove the following theorem:

Theorem 1 *(Augmentation Theorem; Gabow and Tarjan 1984). Suppose B is a base in β_{i-1} and $\beta_i \neq \emptyset$. If (e, f) is a smallest swap for B, then $B/\{e\} \cup \{f\} \in \beta_i$.*

The *Augmentation Theorem* specifies the general approach of the Gabow and Tarjan algorithm: start by finding the optimal base for the smallest possible number of red elements (this number is matroid/task dependent) and then increase the number of red elements by incrementally finding the smallest swap that introduces more red elements. Stop when we have the desired number of red elements in the base.

While this general algorithm applies to undirected spanning trees (they form a matroid), it does not straightforwardly apply to directed spanning trees because they do not form a matroid. To accommodate for this Gabow and Tarjan extend their definition of a swap so that, instead of one, multiple swaps lead from one optimal base to another of a lower order.

So how does this relate to single-root dependency parsing? If we color all edges red, except for those that are connected to the artificial *ROOT* node which will be colored green, we can look for a directed MST with only one green edge (or equivalently with $n - 1$ red edges). This is a special case of the general Gabow and Tarjan (1984) algorithm. An adaptation of that algorithm to dependency parsing was presented by Zmigrod et al. (2020).

While it is stated by both Gabow and Tarjan (1984) and Zmigrod et al. (2020) that this algorithm can be implemented in $\mathcal{O}(n^2)$ for dense graphs by using data structures from Tarjan (1977), it is not trivial to see how to do that. Indeed, to the best of our knowledge, the only implementation of this algorithm for dependency parsing runs in $\mathcal{O}(n^2 \log\ n)$. Even implementing the original unconstrained Tarjan (1977) algorithm is non-trivial, and its presentations with this level of efficiency in the literature historically include errors. The correct efficient $\mathcal{O}(n^2)$ algorithm is distilled and described in our Appendix A, and in our experiments we contrast its implementation against the less efficient ones.

3 The Root Preselection Algorithm

There is a simple meta-algorithm algorithm for single-root (constrained) dependency parsing, when given access to an unconstrained solver as a subroutine. Imagine we want to find the best single-root dependency tree that contains an arc from *ROOT* to a *single particular word* in the sentence. We can accomplish this by disconnecting all other words from the *ROOT* and running the unconstrained MST parser (equivalently, give the relevant edges weight of $-\infty$). Now, we can repeat this process for all the words and compare the weights of the single-root dependency trees that are found for each

[4] We treat the problem of finding a minimum or a maximum spanning tree as equivalent.

word. The best tree in this comparison will be globally best single-root dependency tree. If the runtime complexity of the underlying unconstrained MST parser is $\mathcal{O}(T(n))$ for a sentence of length n, the asymptotic runtime of this meta-algorithm is $\mathcal{O}(nT(n))$. We refer to this algorithm as *Naïve* algorithm.

In practice, a simple heuristic is applied in several dependency parsers on top of *Naïve* algorithm (Parser-v3, Stanza, SuPar). The adapted algorithm with the heuristic first runs the usual unconstrained MST parsing. If the tree that is found contains only one word connected to the root, the algorithm returns it as the answer. Otherwise, the parser applies the *Naïve* algorithm but only over the words connected to the root in the unconstrained parse. Since this adapted algorithm preselects the nodes to which to apply the naïve algorithm we refer to it as the *Root Preselection Algorithm*.

We turn to explain that this undocumented heuristic is actually correct, and will always return the best single-rooted tree. We basically describe why the root edge in the constrained case has to be one of the root edges in the unconstrained spanning tree.

The reason for this stems from an extension of *Augmentation Theorem* for directed graphs by Gabow and Tarjan. This theorem establishes the connection between the optimal solution of $i-1$ red elements and an optimal solution of i red elements. It relates them with the optimal swap (in the extended version for directed graphs it is multiple swaps), where each swap removes a green element and replaces it with a red element. What this means in the context of dependency parsing is that an optimal solution with i edges connected to $ROOT$ contains all the edges connected to $ROOT$ from the optimal solution with $i-1$ edges connected to $ROOT$. This recurrence implies that the edge to $ROOT$ from the constrained single-root dependency parse is present in the unconstrained case, so it is valid for the algorithm above to concentrate only on finding the optimal edge in the set of root edges provided by the unconstrained algorithm.

The runtime of this algorithm depends on the number of words connected to $ROOT$ in the unconstrained MST. If there is only one edge to $ROOT$ in the unconstrained MST, the complexity is $\mathcal{O}(T(n))$. If there is more than one edge from $ROOT$, the complexity is $\mathcal{O}((r+1)T(n))$. We can write this complexity for any number r of edges connected to $ROOT$ as $\mathcal{O}\left((r+1-I_1(r))\,T(n)\right)$ where $I_1(\cdot)$ is an indicator function that returns 1 if the input is 1, otherwise it returns 0. Clearly, the worst case of this algorithm is the same as the worst case of the naïve algorithm because r can be as large as n, but it is interesting to see what is the average computational complexity for this algorithm.

To study the average time complexity of *Preselection* algorithm we need to compute expected runtime under some probability distribution of the number of edges connected to $ROOT$ in the unconstrained MST:

$$\begin{aligned}\mathbb{E}\left[\mathcal{O}\left(\mathit{preselect}(n)\right)\right] &= \mathbb{E}\left[r+1-I_1(r)\right]\quad T(n)\\ &= \left(\mathbb{E}\left[r\right]+1-P(r=1)\right)\quad T(n).\end{aligned} \tag{1}$$

This average complexity expresses the intuition that if the weights of the graph are more likely to produce unconstrained MST with small number of root edges, the algorithm will be faster. So what can we say about the probability over the number of root edges? In practice there are two extreme cases: graph weights in the initial stages of training and in the final stage after training. We will analyze them both in turn.

For the initial stage of training, when the parsing model is only initialized, it is reasonable to assume that the distribution over possible spanning trees is uniform. We can compute the probability of having r root edges by finding the ratio of the number of spanning trees rooted in $ROOT$ that contain r root edges and the total number of spanning trees rooted in $ROOT$. The total number of spanning trees is given by Cayle's formula $(n+1)^{n-1}$ (Cayley, 1889).[5] The number of spanning trees with root edges that go trough particular r nodes can be computed using Matrix-Tree Theorem (Tutte, 1984). To compute the number of spanning trees with any r root edges we need to correct the number by

[5]This is Cayle's formula with an offset of 1 because we have in fact $n+1$ nodes due to the artificial $ROOT$ node. Cayle's formula is originally defined for undirected spanning trees, but it applies equally to *rooted* directed spanning trees because there is one-to-one mapping between these two sets of trees.

multiplying with the number of r combinations. This gives us the following distribution over the number of root edges:

$$P(r;n) = \frac{\binom{n}{r} r n^{n-r-1}}{(n+1)^{n-1}}. \quad (2)$$

When we put Equation 2 into Equation 1, we get that the average case complexity under the uniform distribution of spanning trees is:

$$\mathbb{E}\left[\mathcal{O}\left(preselect(n)\right)\right] = \left(\frac{2n}{n+1} + 1 - \frac{n^{n-1}}{(n+1)^{n-1}}\right) T(n).$$

This expectation is monotonically increasing with n. We can compute the upper bound with:

$$\lim_{n\to\infty} \mathbb{E}\left[\mathcal{O}\left(preselect(n)\right)\right] = \left(3 - \frac{1}{e}\right) T(n) < 2.64\ T(n)$$

This shows that the *Preselection* algorithm for constrained MST parsing performs, on average, just as well as any unconstrained MST algorithm with only a small constant overhead. This is true under the assumption of uniform distribution over trees. The probability of the number of roots that need to be explored depends only mildly on the number of words: the larger n the larger is the probability of having multiple root edges, but for any n it converges to a small value. The number of having more then r number of edges drops rapidly for any n: $P(r > 4) < 0.02$, $P(r > 5) < 0.004$, $P(r > 6) < 0.0005$. In other words, it is very unlikely that this algorithm will need to explore more than a few of different root edges.

What about the distribution of root edges with unconstrained MST after training? For that case we can expect the distribution to be even more peaked over having only few root edges because the training data often has only few root edges (or only one in the case of Universal Dependencies). To test that we collected 10 sentences for each sentence length from the English portion of News Commentary v16 corpus. We ran the English bi-affine model of Stanza (Qi et al., 2020) and computed the average number of root edges for each sentence length. The plot with these counts is shown in Figure 2 as *trained weights* line. The plot also shows *random weights* which represents the uniform spanning tree distribution. To simulate this distribution we sample the weight of each edge of the graph from the uniform distribution. It is easy to see that in expectation all spanning trees will have the same weight.

Zmigrod et al. mention that the distribution of the number of root edges in a trained model depends on the amount of training data. The trained English model in this plot should represent the distribution with the smallest number of root edges since this language has the largest amount of training data. The random weights on this plot should be approximately a lower bound on the number of root edges of a model with small amount of training data.

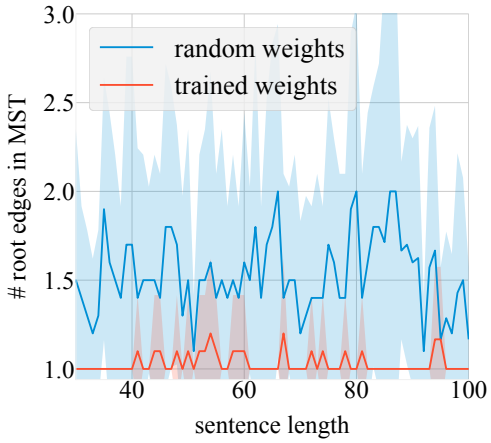


Figure 2: The number of root edges in unconstrained MST for two different types of graph weights.

This plot shows that in the weights produced by a trained English model, the number of unconstrained MSTs with multiple roots is small. This means that the *Root Preselection* algorithm will perform even better than in the random weights setting. This plot also confirms that the expected number of root edges for a randomly initialized weights is smaller than 2 for any sentence length. Clearly, the variance in the number of roots is much higher in the random weights than in the trained weights.

While the *Preselection* algorithm is used in practice by several implementations, to the best of our knowledge, the proof of its correctness and of its average-case complexity analysis that was presented in this section is new.

4 The Root Reweighting Algorithm

We turn to present a new algorithm for single-root dependency parsing that is as fast as the best unconstrained dependency parsing algorithm even in the worst case. It is based on a very simple observation that subtracting a constant value c from the weights of all edges coming out of $ROOT$:

- decreases the weight of any tree with k root edges by $k \cdot c$,
- does not change the ranking among the trees with the same number of root edges, and
- does potentially change the ranking among the trees with different number of root edges.

By choosing the right constant $c \in \mathbb{R}$ we can arrange all trees with more than one root edge to have lower weight than any tree with only one root edge. Let us denote by $w(\cdot)$ the function that provides the value of the weight of an edge in the original graph. Let n stand for the number of words. In a complete graph we have $n+1$ nodes due to the artificial $ROOT$ token. Any spanning tree in this graph will have n edges.

In the original graph, before the constant is subtracted, we know for certain that the score of any spanning tree is not smaller than $n \min_e w(e)$ and not bigger than $n \max_e w(e)$. After constant c is subtracted from all edges coming out of $ROOT$, all trees with k root edges will have their score decreased by $k \cdot c$. In this modified graph, any spanning tree with k root edges will have score that is upper bounded by $n \max_e w(e) - kc$ and lower bound of $n \min_e w(e) - kc$. We want the lowest scoring single-root tree to have a higher score than any k-root tree for $k \geq 2$. More formally, we want the following equation to hold:

$$n \min_e w(e) - c > n \max_e w(e) - kc \quad (3)$$

for all $2 \leq k \leq n$.

This implies that c should satisfy:

$$c > n(\max_e w(e) - \min_e w(e)) \quad (4)$$

A value of c that satisfies this constraint is:

$$c = 1 + n(\max_e w(e) - \min_e w(e)) \quad (5)$$

So by applying unconstrained MST over a graph with the following weight function we in fact obtain the best single-root solution:

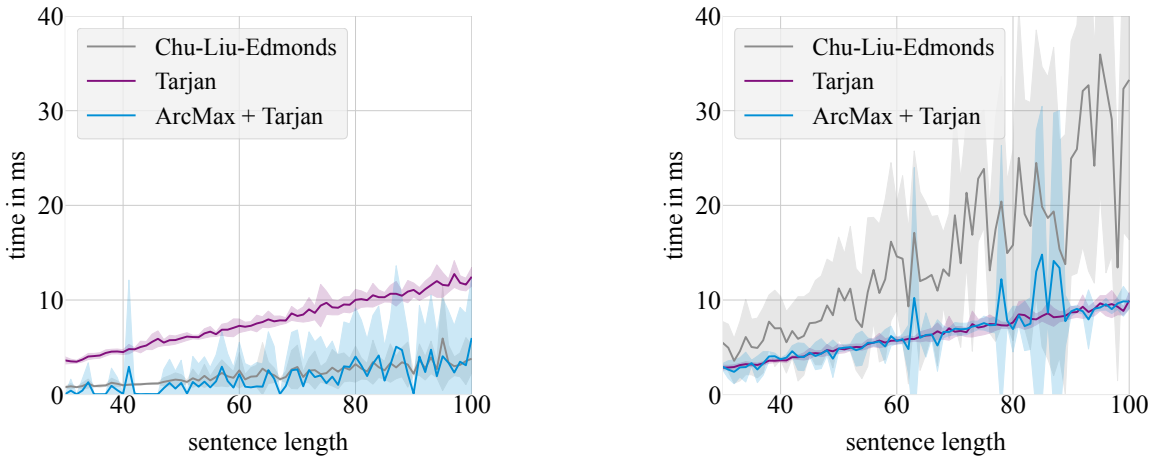
$$w'(e) = \begin{cases} w(e) - c & \text{if } src(e) = ROOT \\ w(e) & \text{otherwise} \end{cases} \quad (6)$$

There are multiple advantages of this algorithm. First, it is simple to understand and implement. Assuming an existing implementation of any unconstrained MST algorithm, this algorithm could be implemented very easily, without incurring further cost to the asymptotic complexity. A full implementation (in Python) is described in Appendix B.1. It is simpler to implement even in comparison to the *Root Preselection* algorithm described in Section 3.

The second advantage is that we could use any implementation of an unconstrained MST as a subroutine. As mentioned before, there is no precise description nor implementation of Gabow and Tarjan algorithm that runs in $\mathcal{O}(n^2)$. The fastest implementation of Gabow and Tarjan is by Zmigrod et al. that runs in $\mathcal{O}(n^2 \log n)$. The *Root Reweighting* algorithm can easily be implemented in $\mathcal{O}(n^2)$ by just using the unconstrained MST algorithm of Tarjan (1977) as a subroutine.

The third advantage is that, unlike the *Preselection* algorithm, the *Reweighting* algorithm always runs the unconstrained MST algorithm only once per sentence. This means that it will be asymptotically fast for any distribution of spanning trees.

Finally, in comparison to Zmigrod et al. (2020), the *Reweighting* algorithm provides for a great flexibility in choosing the underlying unconstrained MST algorithm that is used as a subroutine. In our experiments we use the MST algorithm of Tarjan for dense graphs that runs in $\mathcal{O}(n^2)$. If the graph were sparse, for example, due to pruning of unlikely or forbidden edges, we could use the unconstrained MST algorithm of Gabow et al. (1986) as a subroutine which runs in $\mathcal{O}(m + n \log n)$ where m is the number of edges in the input graph. In addition, if we want to perform single-root *projective* MST parsing, we could use the algorithm of Eisner (1996) as a subroutine. Our algorithm also applies to k-best parsing. Assuming any existing unconstrained k-best parsing algorithm (such



(a) Trained English input

(b) Random input

Figure 3: General MST algorithms performance.

as Camerini et al. 1980; Hall 2007; Zmigrod et al. 2021), the *Reweighting* algorithm can easily incorporate the constraint that all returned k-best trees have a single root edge by just changing the weights of the input graph before calling the unconstrained k-best algorithm.

In short, this simple algorithm has all the advantages of the previous single-root algorithms and none of their disadvantages.

5 The ArcMax Trick

The *Reweighting* algorithm from Section 4 that uses Tarjan's algorithm as a subroutine is the best possible algorithm we could hope for in the worst-case with respect to asymptotic complexity. No algorithm can be asymptotically faster than $\mathcal{O}(n^2)$ for complete graphs.

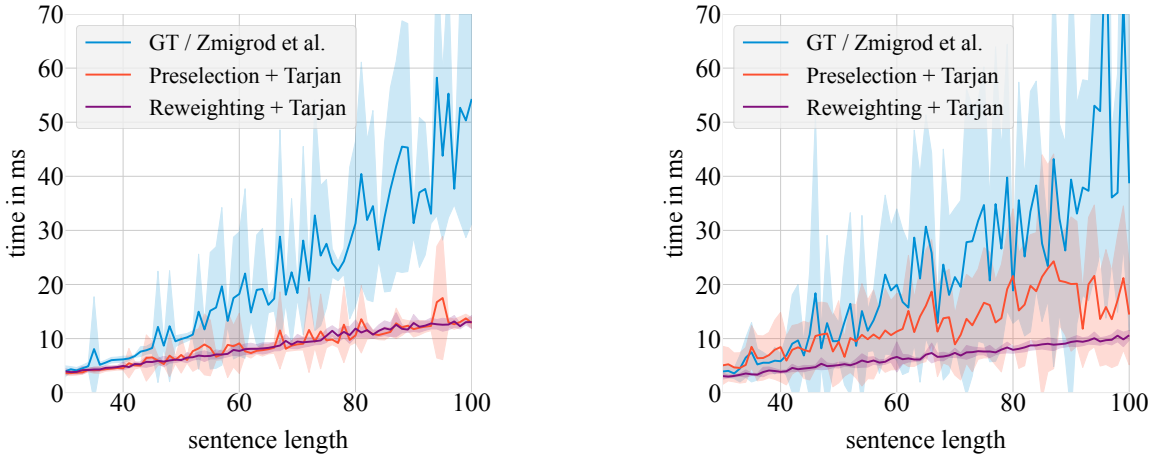
Tarjan's algorithm, works in two phases. The first one recursively contracts cycles that result by picking the best edge that enters each node. The second phase then reverses the recursion by expanding each contraction. To do all of this, the algorithm needs to keep track of all the contracted cycles and of modifications to the weights entering the cycles. All of these operations are asymptotically optimal, but they do incur some constant overhead. There are some input instances whose structure is such that we can avoid this overhead and avoid running the full Tarjan's algorithm altogether. Zhang et al. (2017) show that the neural models are often learned so accurately that just picking the input arc with the highest weight for each word often gives a valid tree.

If for each node we just pick the arc with the highest weight and check if these arcs form a tree we could avoid running the whole MST algorithm. We call this trick *ArcMax* trick. In principle it could be applied to any MST algorithm, but it would not give equal benefits to all of them. Zhang et al. apply it over the CLE algorithm but in that case it is redundant: CLE, as its first step, performs the same step as ArcMax. Zhang et al. do not report any speed improvements.

We show that a speedup can be achieved if this trick is used as a procedure before Tarjan's algorithm. Tarjan's algorithm requires that the graph is strongly connected. In order to achieve this we have to add edges that enter $ROOT$ and set their weight to $-\infty$. This means that Tarjan's algorithm will always find cycles to contract even if the problem is simple and could be solved by picking the maximum edges entering each word. To address this, we add the ArcMax trick to Tarjan's algorithm.

Checking whether ArcMax edges form a non-projective tree can be done in linear time: do depth-first search from the $ROOT$ node and in the end check if all words are visited. Checking for the projective tree can also be done in linear time by constructing a shift-reduce oracle (linear time), running it over the sentence (linear time) and checking whether in the end the only token left on the stack is $ROOT$ (constant time). The code for the checks of projective and non-projective trees is in Appendix B.2.

For the single-root constraint we need to ex-



(a) Trained English input

(b) Random input

Figure 4: Single-Root MST algorithms performance without ArcMax.

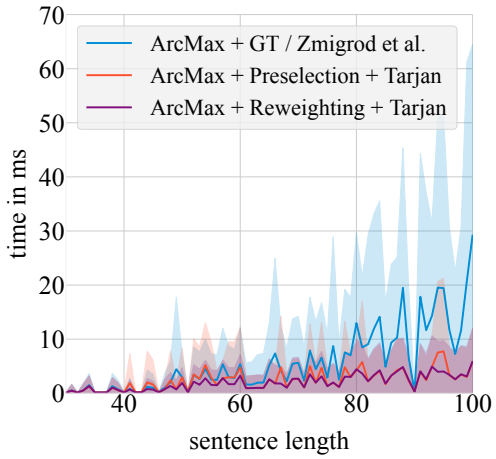


Figure 5: Single-Root MST algorithms performance with ArcMax in trained weights setting.

tend this trick to also verify that the extracted tree has only one edge coming out of *ROOT*. For the *Root Preselection* and *Reweighting* algorithms we have a choice of when to apply the *ArcMax* trick: before the single-root algorithm or inside it just before it calls the unconstrained MST algorithm that is used as a subroutine. For *Root Preselection*, in practice, there is no difference in performance. However, for *Reweighting* the choice is crucial. After reweighting is applied, the edges coming out of *ROOT* will not be the best edges that enter any word, therefore *ArcMax* will never be useful since it will not produce any complete tree. This is why the *ArcMax* trick can be applied to parsing with the *Reweighting* algorithm only if it is used before *Reweighting* is called.

6 Experiments

In this section we experimentally answer some questions about the performance (speed) of different variations of the algorithms we described. We test algorithms in two settings. The first setting has the graph weights from a trained English dependency paring model from the state-of-the-art Stanza parser (Qi et al., 2020). The parser is applied to sentences of different length selected from the News Commentary v16 corpus.[6] For each length we select exactly ten sentences. The second setting uses graphs with weights sampled from a uniform distribution. This setting should be similar to the initial stages of training of most models. The number of generated random graphs is the same as the number of sentences from the trained setting. We will refer to the first setting as *trained weights* and to the latter as *random weights*. We stress that we do not test for accuracy, but for speed. Accuracy of all the tested algorithms remains unchanged.

Which unconstrained algorithm is the fastest? Figure 3 shows the plots for two different settings for CLE, Tarjan and ArcMax+Tarjan. In the trained setting it is visible that worst-case complexity analysis is in fact not informative about the actual performance of the algorithm. CLE outperforms Tarjan's algorithm precisely because it can stop the algorithm if the problem is easy, as described in

[6] http://www.statmt.org/wmt21/translation-task.html#download

Section 5. In the random setting, Tarjan's algorithm works better than CLE. When we add ArcMax trick to Tarjan we get an algorithm that works best in both settings: it optimizes execution on the easy trained setting and it uses robustness of Tarjan's algorithm in the random setting without slowing it down.

Use ArcMax before or after single-root step? As mentioned in Section 5 there are two places where the ArcMax trick could be used. We argued that using it before the single-root step is preferable. The results in Figure 6 confirm that.

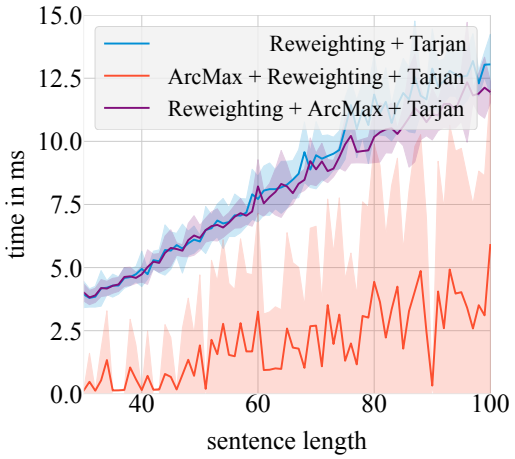


Figure 6: Comparison on the timing of using the ArcMax trick in the trained weights setting.

Which single-root algorithm is the fastest? First we compare the algorithms without using the ArcMax trick. Figure 4 shows this comparison. Both *Preselection* and *Reweighting* significantly outperform the algorithm of Zmigrod et al.. *Reweighting* outperforms Zmigrod et al. on average 2.6x on trained weights and 3.6x on random weights. The difference is most extreme on the longest sentences. The performance curve of Zmigrod et al. also seems much more volatile.

For *Preselection*, we can see that the average-case analysis from Section 3 is much more informative of the performance than the worst-case analysis. For *Preselection* vs. *Reweighting* we see that in the random setting the performance of *Reweighting* is much more stable with very low variance and that it consistently outperforms *Preselection*.

If we apply the ArcMax trick to all of these algorithms, they all get much faster but the relative speed between them stays the same. To see that, compare the results in Figures 4a and 5. We do not show the results on the random setting because they are equivalent to those without ArcMax in Figure 4b.

When using all of the techniques in our paper together, namely ArcMax+Reweighting+Tarjan, we get an algorithm that is on average 11x faster than the algorithm of Zmigrod et al. when applied to the output of a trained parser. A better implementation of Zmigrod et al. could possibly make this algorithm more competitive but it is unlikely that it would compensate for this large performance gap.

Is Reweighting algorithm always the fastest? While the Reweighting algorithm both theoretically and practically improves over the Preselection algorithm, it should be mentioned that the performance in practice depends on the implementation of Tarjan's algorithm as the underlying unconstrained MST algorithm. If instead of Tarjan we used CLE, the Preselection algorithm will work better on the trained input (see Figure 11 in Appendix). The main reason for that is that CLE algorithm, unlike Tarjan, has a computational complexity that varies, depending on the input, between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$. On trained input, in general, CLE tends to be closer to its best-case complexity because there are not many cycles to be contracted. However, the Reweighting algorithm changes the weights of a graph in such a way that there are *always* cycles that need to be contracted and thereby causes CLE to be closer to its worst-case complexity. This problem does not exist with Tarjan's algorithm that both in best and worst-case runs in $\mathcal{O}(n^2)$. Our recommendation is to use ArcMax+Reweighting+Tarjan as the fastest and most stable algorithm, but if some other unconstrained algorithm is used in place of Tarjan's, it should be tested if the Reweighting algorithm runs faster than Preselection.

7 Conclusion

We demonstrated how to obtain significant speed-ups in single-root dependency parsing. The two proposed algorithms are fast, flexible, easy to understand and simple to implement in comparison to previously published ones.

Acknowledgments

We thank the anonymous reviewers for their comments and useful feedback. We also thank the developers of the Stanza parser for enabling our work and analysis. This work was supported by ERC H2020 Advanced Fellowship GA 742137 SEMANTAX grant and Bloomberg. Miloš Stanojević is especially grateful to Xin Zhan and the staff of NHS Western General Hospital. During the last year they have helped significantly in making this paper possible, but the importance of their dedicated work goes far beyond that.

References

Mark Anderson and Carlos Gómez-Rodríguez. 2020. Distilling neural networks for greener and faster dependency parsing. In *Proceedings of the 16th International Conference on Parsing Technologies and the IWPT 2020 Shared Task on Parsing into Enhanced Universal Dependencies*, pages 2–13, Online. Association for Computational Linguistics.

P. M. Camerini, L. Fratta, and F. Maffioli. 1979. A note on finding optimum branchings. *Networks*, 9(4):309–312.

Paolo M. Camerini, Luigi Fratta, and Francesco Maffioli. 1980. Ranking arborescences in o(km log n) time. *European Journal of Operational Research*, 4(4):235–242. Combinational Optimization.

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

Y. Chu and T. Liu. 1965. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396—-1400.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition*, 3rd edition. The MIT Press.

J. Edmonds. 1967. Optimum branchings. *J. Res. Nat. Bur. Standards*, pages 233—-240.

Jason M. Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *COLING 1996 Volume 1: The 16th International Conference on Computational Linguistics*.

Harold N Gabow, Zvi Galil, Thomas Spencer, and Robert E Tarjan. 1986. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122.

Harold N Gabow and Robert E Tarjan. 1984. Efficient algorithms for a family of matroid intersection problems. *Journal of Algorithms*, 5(1):80–131.

Keith Hall. 2007. K-best spanning tree parsing. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 392–399, Prague, Czech Republic. Association for Computational Linguistics.

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

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajic. 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.

B. M. E. Moret. 2002. Towards a discipline of experimental algorithmics. In M. H. Goldwasser, D. S. Johnson, and C. C. McGeoch, editors, *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, pages 197–214.

Joakim Nivre, Mitchell Abrams, Željko Agić, Lars Ahrenberg, Lene Antonsen, Katya Aplonova, Maria Jesus Aranzabe, Gashaw Arutie, Masayuki Asahara, Luma Ateyah, Mohammed Attia, Aitziber Atutxa, Liesbeth Augustinus, Elena Badmaeva, Miguel Ballesteros, Esha Banerjee, Sebastian Bank, Verginica Barbu Mititelu, Victoria Basmov, John Bauer, Sandra Bellato, Kepa Bengoetxea, Yevgeni Berzak, Irshad Ahmad Bhat, Riyaz Ahmad Bhat, Erica Biagetti, Eckhard Bick, Rogier Blokland, Victoria Bobicev, Carl Börstell, Cristina Bosco, Gosse Bouma, Sam Bowman, Adriane Boyd, Aljoscha Burchardt, Marie Candito, Bernard Caron, Gauthier Caron, Gülşen Cebiroğlu Eryiğit, Flavio Massimiliano Cecchini, Giuseppe G. A. Celano, Slavomír Čéplö, Savas Cetin, Fabricio Chalub, Jinho Choi, Yongseok Cho, Jayeol Chun, Silvie Cinková, Aurélie Collomb, Çağrı Çöltekin, Miriam Connor, Marine Courtin, Elizabeth Davidson, Marie-Catherine de Marneffe, Valeria de Paiva, Arantza Diaz de Ilarraza, Carly Dickerson, Peter Dirix, Kaja Dobrovoljc, Timothy Dozat, Kira Droganova, Puneet Dwivedi, Marhaba Eli, Ali Elkahky, Binyam Ephrem, Tomaž Erjavec, Aline Etienne, Richárd Farkas, Hector Fernandez Alcalde, Jennifer Foster, Cláudia Freitas, Katarína Gajdošová, Daniel Galbraith, Marcos Garcia, Moa Gärdenfors, Sebastian Garza, Kim Gerdes, Filip Ginter, Iakes Goenaga, Koldo Gojenola, Memduh Gökırmak, Yoav Goldberg, Xavier Gómez Guinovart, Berta Gonzáles Saavedra, Matias Grioni, Normunds Grūzītis, Bruno Guillaume, Céline Guillot-Barbance, Nizar

Habash, Jan Hajič, Jan Hajič jr., Linh Hà Mỹ, Na-Rae Han, Kim Harris, Dag Haug, Barbora Hladká, Jaroslava Hlaváčová, Florinel Hociung, Petter Hohle, Jena Hwang, Radu Ion, Elena Irimia, Ọlájídé Ishola, Tomáš Jelínek, Anders Johannsen, Fredrik Jørgensen, Hüner Kaşıkara, Sylvain Kahane, Hiroshi Kanayama, Jenna Kanerva, Boris Katz, Tolga Kayadelen, Jessica Kenney, Václava Kettnerová, Jesse Kirchner, Kamil Kopacewicz, Natalia Kotsyba, Simon Krek, Sookyoung Kwak, Veronika Laippala, Lorenzo Lambertino, Lucia Lam, Tatiana Lando, Septina Dian Larasati, Alexei Lavrentiev, John Lee, Phuong Lê Hồng, Alessandro Lenci, Saran Lertpradit, Herman Leung, Cheuk Ying Li, Josie Li, Keying Li, KyungTae Lim, Nikola Ljubešić, Olga Loginova, Olga Lyashevskaya, Teresa Lynn, Vivien Macketanz, Aibek Makazhanov, Michael Mandl, Christopher Manning, Ruli Manurung, Cătălina Mărănduc, David Mareček, Katrin Marheinecke, Héctor Martínez Alonso, André Martins, Jan Mašek, Yuji Matsumoto, Ryan McDonald, Gustavo Mendonça, Niko Miekka, Margarita Misirpashayeva, Anna Missilä, Cătălin Mititelu, Yusuke Miyao, Simonetta Montemagni, Amir More, Laura Moreno Romero, Keiko Sophie Mori, Shinsuke Mori, Bjartur Mortensen, Bohdan Moskalevskyi, Kadri Muischnek, Yugo Murawaki, Kaili Müürisep, Pinkey Nainwani, Juan Ignacio Navarro Horñiacek, Anna Nedoluzhko, Gunta Nešpore-Bērzkalne, Luong Nguyễn Thị, Huyền Nguyễn Thị Minh, Vitaly Nikolaev, Rattima Nitisaroj, Hanna Nurmi, Stina Ojala, Adédayọ̀ Olúòkun, Mai Omura, Petya Osenova, Robert Östling, Lilja Øvrelid, Niko Partanen, Elena Pascual, Marco Passarotti, Agnieszka Patejuk, Guilherme Paulino-Passos, Siyao Peng, Cenel-Augusto Perez, Guy Perrier, Slav Petrov, Jussi Piitulainen, Emily Pitler, Barbara Plank, Thierry Poibeau, Martin Popel, Lauma Pretkalniņa, Sophie Prévost, Prokopis Prokopidis, Adam Przepiórkowski, Tiina Puolakainen, Sampo Pyysalo, Andriela Rääbis, Alexandre Rademaker, Loganathan Ramasamy, Taraka Rama, Carlos Ramisch, Vinit Ravishankar, Livy Real, Siva Reddy, Georg Rehm, Michael Rießler, Larissa Rinaldi, Laura Rituma, Luisa Rocha, Mykhailo Romanenko, Rudolf Rosa, Davide Rovati, Valentin Roșca, Olga Rudina, Jack Rueter, Shoval Sadde, Benoît Sagot, Shadi Saleh, Tanja Samardžić, Stephanie Samson, Manuela Sanguinetti, Baiba Saulīte, Yanin Sawanakunanon, Nathan Schneider, Sebastian Schuster, Djamé Seddah, Wolfgang Seeker, Mojgan Seraji, Mo Shen, Atsuko Shimada, Muh Shohibussirri, Dmitry Sichinava, Natalia Silveira, Maria Simi, Radu Simionescu, Katalin Simkó, Mária Šimková, Kiril Simov, Aaron Smith, Isabela Soares-Bastos, Carolyn Spadine, Antonio Stella, Milan Straka, Jana Strnadová, Alane Suhr, Umut Sulubacak, Zsolt Szántó, Dima Taji, Yuta Takahashi, Takaaki Tanaka, Isabelle Tellier, Trond Trosterud, Anna Trukhina, Reut Tsarfaty, Francis Tyers, Sumire Uematsu, Zdeňka Urešová, Larraitz Uria, Hans Uszkoreit, Sowmya Vajjala, Daniel van Niekerk, Gertjan van Noord, Viktor Varga, Eric Villemonte de la Clergerie, Veronika Vincze, Lars Wallin, Jing Xian Wang, Jonathan North Washington, Seyi Williams, Mats Wirén, Tsegay Woldemariam, Tak-sum Wong, Chunxiao Yan, Marat M. Yavrumyan, Zhuoran Yu, Zdeněk Žabokrtský, Amir Zeldes, Daniel Zeman, Manying Zhang, and Hanzhi Zhu. 2018. Universal dependencies 2.3. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.

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*.

Tim Roughgarden. 2019. Beyond worst-case analysis. *Communications of the ACM*, 62(3):88–96.

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.

Xingxing Zhang, Jianpeng Cheng, and Mirella Lapata. 2017. Dependency parsing as head selection. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 665–676, Valencia, Spain. Association for Computational Linguistics.

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

Ran Zmigrod, Tim Vieira, and Ryan Cotterell. 2021. 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.

Uri Zwick. 2013. Lecture notes on "Analysis of Algorithms": Directed Minimum Spanning Trees (More complete but still unfinished).

A Tarjan's Unconstrained MST Algorithm

The original Chu-Liu-Edmonds algorithm (CLE) runs in $\mathcal{O}(n^3)$. Tarjan (1977) improves this by using advanced data structures. Tarjan proposes two variations of the algorithm. The first one runs in $\mathcal{O}(m \log n)$ where m is the number of edges and n is the number of nodes. In the case of dense input graphs, such as those in dependency parsing, where the number of edges is n^2 the complexity of this algorithm is $\mathcal{O}(n^2 \log n)$. Tarjan proposed a second version of the algorithm that in the case of dense graphs has complexity of $\mathcal{O}(n^2)$. These two versions of the algorithm differ only in the type of a priority queue that they use.

The description of the algorithm in the original paper is not very accessible and contains a small error. Camerini et al. (1979) fixes this error and introduces some simplifications. Zwick (2013) provides a very accessible introduction to this algorithm, but unfortunately also with some errors and it does not cover the optimization for dense graphs. Our presentation here is a synthesis of these previous presentations. We assume that the reader is familiar with the standard CLE algorithm, Union-Find (disjoint sets) and meldable heaps (such as Fibonacci Heaps). For an introduction of CLE, see Kübler et al. (2009, §4.3.3). For an introduction to Union-Find and Fibonacci Heaps see Cormen et al. (2009, §19 and §21)

Just like the CLE algorithm, Tarjan's algorithm works in two phases. The first phase performs all the detection and contractions of cycles. Phase two expands those contractions to recover the optimal spanning tree. The algorithm for Phase I is shown in Algorithm 1. This is the first version of the algorithm that runs in $\mathcal{O}(n^2 \log n)$ on dense graphs. We explain later how to modify it to get $\mathcal{O}(n^2)$ runtime.

The algorithm uses the following data structures:

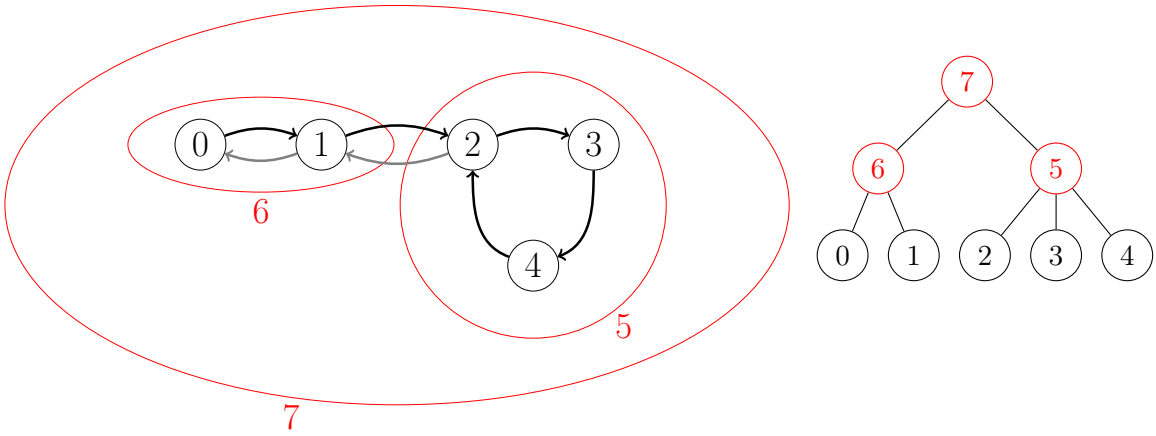
- $P[i]$ is a priority queue that contain all edges that enter (super-)node i
- $in[i]$ stores the best edge that enters the (super-)node i,
- $prev[i]$ stores the (super-)node that precedes (super-)node i on the path that is currently being formed,
- $parent[i]$ stores the super-node (cycle) in which (super-)node i takes part,
- $children[i]$ stores all the (super-)nodes that are part of the cycle represented by super-node i (inverse of *parent*).

One of the main insights of Tarjan is that when we do contraction of cycles, we do not need to explicitly change the edges that enter and leave the cycle. Instead, we keep the edges as they are but keep a separate disjoint-set data structure that will tell us for any edge to which cycle its source and target belong. This disjoint-set is represented by *parent* array. To make disjoint set operations efficient two heuristics are often applied in combination: union-by-rank and path-compression. Union-by-rank complicates implementation slightly and is not very important because even without it Tarjan's algorithm has the same runtime since disjoint-set is not a bottleneck. Path-compression is sufficient to get a fast runtime, but path compression destroys the tree (it maintains only the information of which node the root of the tree is). Since Phase II of Tarjan's algorithm needs the whole tree we should keep a separate array that works like *parent*, but unlike *parent* it is used only for the destructive find operation of the disjoint-sets. We do not put this in the pseudo-code since it would complicate the presentation.

Tarjan's algorithm requires that the graph is strongly connected. We can easily ensure this in $\mathcal{O}(n)$ time by adding edges with weight $-\infty$ between every node i and $i+1$ in both directions (assuming any arbitrary ordering between nodes).

The algorithm starts at an arbitrary node a. It takes the highest scoring edge entering a (line 9) and finds the cycle (super-node) to which the source of the edge belongs. There are three cases to be explored for this edge.

1. this is a self-loop, i.e. both source and target belong to an already collapsed cycle, in that case just move to the next best edge of the current node,
2. this is an extension of the path, in that case we move to the source of the path,
3. this edge closes a cycle, in that case we collapse the cycle into a super-node.



(a) Graph (b) Cycle tree

Figure 7: Example run of Tarjan's algorithm.

When we collapse the cycle in *case 3*, we meld the priority queues with the edges of all the nodes that participate in the cycle. This is why *case 1* is possible: after collapsing done by *case 3* we do not remove the edges that are within the elements that are inside the cycle. This is the key point that differentiates the two versions of Tarjan's algorithm.

The first version can use any implementation of a priority queue that has an efficient meld operation, for example Fibonacci Heaps can do it in constant time. With that heap implementation the algorithm needs to do m extract_max operations and n meld operations that gives complexity $\mathcal{O}(m \log n)$.

The second version of the algorithm which is optimized for dense graphs has a very different implementation of a priority queue. In this version of the algorithm, a queue is only a simple array of length n (number of nodes) where each element is a weight of some source edge entering the current node or a *NaN* value if that edge is already extracted. Extracting the maximum in this representation is done by a linear scan trough the array. Melding is an interesting operation here because it is a lossy operation. Imagine that we need to meld two queues of this type named a and b. If both queues have entering edges from some node i, the melded queue needs to store only the highest scoring one (we care only about the best edge that enters the cycle from some outside node). So $c[i] = \max(a[i], b[i])$. If either $a[i]$ or $b[i]$ is *NaN* then $c[i]$ will be *NaN* too. This will remove self-loops and therefore eliminate the need for *case 1* of the first version of the algorithm. A Python implementation of this priority queue is shown in Figure 12.

As a note, we should mention that some papers mention that Radix sort is needed for implementing this efficient queue. This stems from Tarjan's original paper that mentions Radix sort for initialization of the queue. However, Radix sort is not needed in the complete graph. The reason why Tarjan proposes Radix sort is to avoid worst-case complexity when graph is not sparse but not fully complete either. If we want to stretch this analogy, we can see our implementation as one that uses Counting sort instead of Radix sort.

Since this version of a queue has a slower extract_max and slower meld, what is its purpose? The main advantage of this type of queue is that it removes the self-loop edges that appear with contraction so the *case 1* described above will never appear. That means that this version of the algorithm has only n extract_max operations and n meld operations which gives total runtime of $\mathcal{O}(n^2)$.

The Algorithm 2 presents the second phase Tarjan's algorithm that decomposes the cycle tree constructed by the first phase. For a more detailed description of this phase, see Zwick (2013).

As an illustration of the first phase of Tarjan's algorithm consider the graph in Figure 7a. Imagine that the original graph contained only black edges. In order to apply Tarjan's algorithm we first needed to add gray edges with cost $-\infty$ (we did not add similar edges for

$3 \to 2$ and $4 \to 3$ to keep things simple). This graph is now strongly connected. We can start parsing from any node in the graph. Let us assume we started from node 2. We take the best non-visited edge entering the current node (extract_max($P[a]$)). That gives us the edge from node 4. We go to node 4 and repeat the same process that leads us to node 3 and then 2. We have formed a cycle by building the path backwards. This cycle is contracted forming a super-node 5. A note of this is taken in Figure 7b that represents non-compressed version of the disjoint-set structure. We continue choosing the best edge from node 5 and get to node 1 and then 0. When we take the best edge entering node 0 we form a cycle. Notice that this edge has weight $-\infty$ but that is still the best edge that enters 0. Finally we form the cycle that covers the whole graph. Notice that for this phase of the algorithm it does not matter which node is our designated root node. The choice of the root plays part only in the second phase.

Algorithm 1 Tarjan Phase I – Collapsing

for $i \in V$ **do** ▷ Initialization
  $P[i] \leftarrow$ priority_queue($\{(j,i) \in G\}$)
  $in[i] \leftarrow null$
  $prev[i] \leftarrow null$
  $parent[i] \leftarrow null$
  $children[i] \leftarrow null$
$a \leftarrow$ arbitrary vertex
while $P[a] \neq \emptyset$ **do**
  $(u, v) \leftarrow$ extract_max($P[a]$)
  $b \leftarrow$ find(u)
  if $a = b$ **then**
    continue ▷ This is a self-loop
  else
    $in[a] \leftarrow (u, v)$
    $prev[a] \leftarrow b$
    if $in[u] = null$ **then**
      $a \leftarrow b$ ▷ Path extension
    else
      $c \leftarrow$ new_vertex() ▷ New cycle
      $i \leftarrow a$
      do ▷ Collect nodes in the cycle
        insert($children[c]$, find(i))
        $i \leftarrow prev[i]$
      while $i \neq a$
      for $i \in children[c]$ **do**
        $parent[i] \leftarrow c$
        add_const($P[i], -w(in[i])$)
        $P[c] \leftarrow$ meld($P[c], P[i]$)
      $a \leftarrow c$

Algorithm 2 Tarjan Phase II – Expanding

$R \leftarrow \emptyset$
procedure DISMANTLE(u)
  while $parent[u] \neq null$ **do**
    for $v \in children[parent[u]] \backslash \{u\}$ **do**
      $parent[v] \leftarrow null$
      if $children[v] \neq null$ **then**
        insert(R, v)
DISMANTLE(r)
while $R \neq \emptyset$ **do**
  $c \leftarrow$ extract(R)
  $(u, v) \leftarrow in[c]$
  $in[v] \leftarrow (u, v)$
  DISMANTLE(v)
return $\{in[u] \mid u \in V \backslash \{r\}\}$

B Python Implementation

B.1 *Reweighting* Implementation

The implementation of the *Reweighting* algorithm is shown in Figure 8. As input it accepts two arguments. One of them is a function that does unconstrained MST search. This can be an implementation of Chu-Liu-Edmonds or Tarjan's algorithm.

The `scores` parameter is a *NumPy* square matrix (`np.array`) with shape $(n+1, n+1)$. Every entry `scores[i, j]` represents the weight of arc that leaves node j and enters node i (i.e. $j \to i$). Node 0 is *ROOT* node by convention. All edges entering *ROOT* (i.e. `scores[0, :]`) are in most implementation set to $-\infty$ to force MST the solution to have *ROOT* as root. Also all self-loops (diagonal entries) are set to $-\infty$.

While it is in general fine to use $-\infty$ to signify disconnected edges, it would make Reweighting Equation 6 not behave correctly and make every spanning tree have weight ∞. That is why with the first line we replace all infinite values with a *NaN* value. The other lines just apply the Equation 6 before calling the unconstrained MST function.

B.2 *ArcMax* Implementation

Figure 9 shows the implementation of all functions needed for the *ArcMax* optimization. The *arcmax* function takes three arguments. *scores* and *mst_func* are the same as in the previous case. The *one_root* argument is a Boolean flag defining whether we want to perform single-root edge parsing or not.

This function has three main parts. The part that computes the highest scoring edge entering every node (`scores.argmax`), the part that checks whether the subgraph is a tree, and an optional third part that will be executed if a subgraph is not a valid tree. The `scores.argmax` part runs in $\mathcal{O}(n^2)$ but in practice it is extremely fast because it does a very simple operation that is implemented in *C* under the hood. Checking whether the sub-graph is a tree `is_tree` is done quickly in linear time. The full MST parsing is performed in $\mathcal{O}(n^2)$ (or $\mathcal{O}(n^3)$ if we use CLE) only if the previous fast checks fail.

Function `fast_single_root_mst` shows how to combine *ArcMax* and *Reweighting*, assuming that there is an existing implementation of some unconstrained MST parsing algorithm such as Tarjan's.

For the projective case we would need to replace function `is_tree` with the function `is_projective_tree` from Figure 10 and to replace `tarjan` with `eisner`. The algorithm for checking of whether the tree is projective in Figure 10 runs in linear time because it visits every arc in the sub-graphs only once.

```python
def reweighting(scores, mst_func):
    scores2 = np.where(np.isinf(scores), np.nan, scores)
    n = scores.shape[0]-1  # number of words
    scores[:, 0] -= 1 + n*(np.nanmax(scores2)-np.nanmin(scores2))
    return mst_func(scores)
```

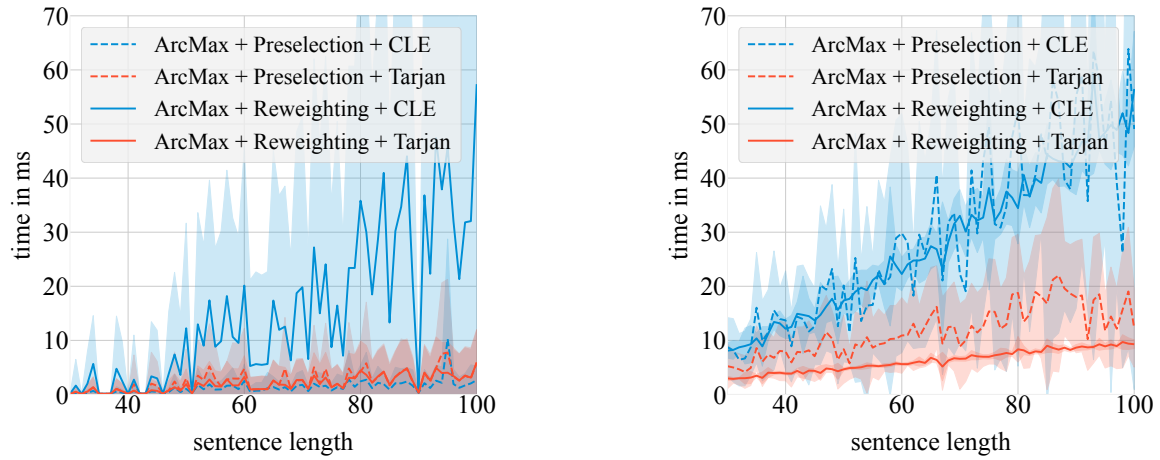
Figure 8: Python implementation of *Reweighting* algorithm

```python
def is_tree(proposal):
    # proposal[i] is a parent of node i
    n = proposal.shape[0]  # number of words + 1 for ROOT
    # convert child-parent pointers to parent-child
    children = [[] for _ in range(n)]
    for i in range(1, n):
        children[proposal[i]].append(i)
    # do depth-first search iteratively
    is_visited = np.zeros(n, dtype=bool)
    stack = [0]
    while len(stack) != 0:
        i = stack.pop()
        is_visited[i] = True
        stack.extend(children[i])
    return is_visited.all()  # true if all nodes were visited

def arcmax(scores, one_root, mst_func):
    proposal = scores.argmax(axis=1)  # find best arc for each node
    root_count = sum(proposal[1:] == 0)
    if is_tree(proposal) and (root_count == 1 or not one_root):
        return proposal
    else:
        return mst_func(scores)

def fast_unconstrained_mst(scores):
    return arcmax(scores, False, tarjan)

def fast_single_root_mst(scores):
    return arcmax(scores, True, lambda x: reweighting(x, tarjan))
```

Figure 9: Python implementation of *ArcMax* optimization

```python
def is_projective_tree(proposal):
    n = proposal.shape[0]
    deps_count = np.zeros(n, dtype=int)
    for i in range(1, n):
        deps_count[proposal[i]] += 1
    stack = [0]
    for i in range(1, n):
        stack.append(i)
        while len(stack) > 1:
            right = stack.pop()
            left = stack.pop()
            if proposal[left] == right:
                # exists left arc
                stack.append(right)
                deps_count[right] -= 1
            elif proposal[right] == left and deps_count[right] == 0:
                # exists right arc
                stack.append(left)
                deps_count[left] -= 1
            else:
                # no attachments possible
                # restore stack and move to next word
                stack.append(left)
                stack.append(right)
                break
    return stack == [0]
```

Figure 10: Python implementation of a linear-time check for whether a sub-graph is a projective tree.



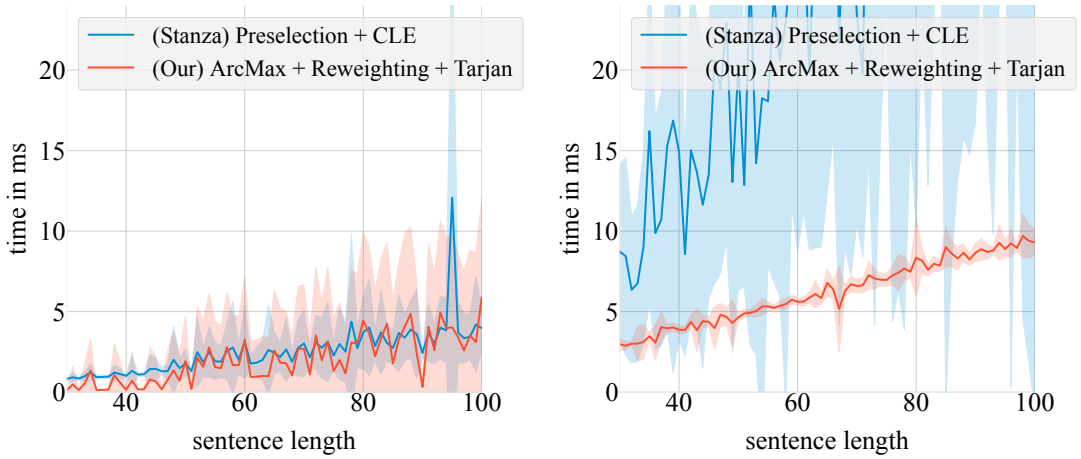
(a) Trained English input

(b) Random input

Figure 11: Comparison of combinations of Meta-Algorithm (Preselection, Reweighting) and MST algorithm (Tarjan, CLE).

```python
class EdgePriorityQueue:

    def __init__(self, node_id: int, edge_weights: np.ndarray):
        self.target = np.full(edge_weights.shape, node_id)
        self.weights = edge_weights
        self.weights[node_id] = np.nan

    def __len__(self) -> int:
        return np.count_nonzero(~np.isnan(self.weights))

    def extract_max(self) -> (int, int, float):
        i = np.nanargmax(self.weights)
        if np.isnan(self.weights[i]):  # nanargmax bug with -inf
            i = np.argmax(np.isinf(self.weights))
        w = self.weights[i]
        self.weights[i] = np.nan
        return i, self.target[i], w

    def meld_inplace(self, other) -> None:
        to_replace = (self.weights < other.weights)
        self.target[to_replace] = other.target[to_replace]
        self.weights[to_replace] = other.weights[to_replace]
        self.weights[np.isnan(other.weights)] = np.nan

    def add_const(self, const: float) -> None:
        self.weights[~np.isinf(self.weights)] += const
```

Figure 12: Python implementation of the priority queue needed for the dense graph version of Tarjan's algorithm.



(a) Trained English input

(b) Random input

Figure 13: Comparison against Stanza's implementation of single-root dependency parsing.