Effective Greedy Inference for Graph-based Non-Projective Dependency Parsing

Ilan Tchernowitz **Liron Yedidsion** **Roi Reichart**
Faculty of Industrial Engineering and Management, Technion, IIT
{ilantc@campus|lirony@ie|roiri@ie}.technion.ac.il

Abstract

Exact inference in high-order graph-based non-projective dependency parsing is intractable. Hence, sophisticated approximation techniques based on algorithms such as belief propagation and dual decomposition have been employed. In contrast, we propose a simple greedy search approximation for this problem which is very intuitive and easy to implement. We implement the algorithm within the second-order TurboParser and experiment with the datasets of the CoNLL 2006 and 2007 shared task on multilingual dependency parsing. Our algorithm improves the run time of the parser by a factor of 1.43 while losing 1% in UAS on average across languages. Moreover, an ensemble method exploiting the joint power of the parsers, achieves an average UAS 0.27% higher than the TurboParser.

1 Introduction

Dependency parsing is instrumental in NLP applications, with recent examples in information extraction (Wu and Weld, 2010), word embeddings (Levy and Goldberg, 2014), and opinion mining (Almeida et al., 2015). The two main approaches for this task are graph based (McDonald et al., 2005) and transition based (Nivre et al., 2007).

The graph based approach aims to optimize a global objective function. While exact polynomial inference algorithms exist for projective parsing (Eisner, 1996; McDonald et al., 2005; Carreras, 2007; Koo and Collins, 2010, inter alia), high order non-projective parsing is NP-hard (McDonald and Pereira, 2006). The current remedy for this comes in the form of advanced optimization techniques such as dual decomposition (Martins et al., 2013), LP relaxations (Riedel et al., 2012), belief propagation (Smith and Eisner, 2008; Gormley et al., 2015) and sampling (Zhang et al., 2014b; Zhang et al., 2014a).

The transition based approach (Zhang and Nivre, 2011; Bohnet and Nivre, 2012; Honnibal et al., 2013; Choi and McCallum, 2013a, inter alia), and the easy first approach (Goldberg and Elhadad, 2010) which extends it by training non-directional parsers that consider structural information from both sides of their decision points, lack a global objective function. Yet, their sequential greedy solvers are fast and accurate in practice.

We propose a greedy search algorithm for high-order, non-projective graph-based dependency parsing. Our algorithm is a simple iterative graph-based method that does not rely on advanced optimization techniques. Moreover, we factorize the graph-based objective into a sum of terms and show that our basic greedy algorithm relaxes the global objective by sequentially optimizing these terms instead of globally optimizing their sum.

Unlike previous greedy approaches to dependency parsing, transition based and non-directional, our algorithm does not require a specialized feature set or a training method that specializes in local decisions. In contrast, it supports global parameter training based on the comparison between an induced tree and the gold tree. Hence, it can be integrated into any graph-based parser.

We first present a basic greedy algorithm that relaxes the global graph-based objective (Section 3). However, as this simple algorithm does not provide a

realistic estimation of the impact of an arc selection on uncompleted high-order structures in the partial parse forest, it is not competitive with state of the art approximations. We hence present an advanced version of our algorithm with an improved arc score formulation and show that this simple algorithm provides high quality solutions to the graph-based inference problem (Section 4).

Particularly, we implement the algorithm within the TurboParser (Martins et al., 2013) and experiment (Sections 8 and 9) with the datasets of the CoNLL 2006-2007 shared tasks on multilingual dependency parsing (Buchholz and Marsi, 2006; Nilsson et al., 2007). On average across languages our parser achieves UAS scores of 87.78% and 89.25% for first and second order parsing respectively, compared to respective UAS of 87.98% and 90.26% achieved by the original TurboParser.

We further implement (Section 6) an ensemble method that integrates information from the output tree of the original TurboParser and the arc weights learned by our variant of the parser into our search algorithm to generate a new tree. This yields an improvement: average UAS of 88.03% and 90.53% for first and second parsing, respectively.

Despite being greedy, the theoretical runtime complexity of our advanced algorithm is not better than the best previously proposed approximations for our problem ($O(n^{k+1})$, for n word sentences and k order parsing, Section 5). In experiments, our algorithms improve the runtime of the TurboParser by a factor of up to 2.41.

The main contribution of this paper is hence in providing a simple, intuitive and easy to implement solution for a long standing problem that has been addressed in past with advanced optimization techniques. Besides the intellectual contribution, we believe this will make high-order graph-based dependency parsing accessible to a much broader research and engineering community as it substantially relaxes the coding and algorithmic proficiency required for the implementation and understanding of parsing algorithms.

2 Problem Formulation

We start with a brief definition of the high order graph-based dependency parsing problem. Given an n word input sentence, an input graph $G = (V, E)$ is defined. The set of vertices is $V = \{0, ..., n\}$, with the $\{1, \ldots, n\}$ vertices representing the words of the sentence, in their order of appearance, and the 0 vertex is a specialized *root* vertex. The set of arcs is $E = \{(u, v) : u \in \{0, ..., n\}, v \in \{1, ..., n\}, u \neq v\}$, that is, the root vertex has no incoming arcs.

We further define a part of order k to be a subset of E of size k, and denote the set of all parts with *parts*. For the special case of $k = 1$ a part is an arc. Different works employed different $parts$ sets (e.g. (Martins et al., 2013; McDonald et al., 2005; Koo and Collins, 2010)). Generally, most *parts* sets consist of arcs connecting vertices either vertically (e.g. $\{(u, v), (v, z)\}$ for $k = 2$) or horizontally (e.g. $\{(u, v), (u, z)\}$, for $k = 2$). In this paper we focus on the parts employed by (Martins et al., 2013), a state-of-the-art parser, but our algorithms are generally applicable for any *parts* set consistent with this general definition.[1]

In graph-based dependency parsing, each part p is given a score $W_p \in \mathcal{R}$. A Dependency Tree (*DT*) T is a subset of arcs for which the following conditions hold: (1) Every vertex, except for the root, has an incoming arc: $\forall v \in V \setminus \{0\} : \exists u \in V s.t. (u, v) \in T$; (2) No vertex has multiple incoming arcs: $\forall (u, u', v) \in V, (u, v) \in T \rightarrow (u', v) \notin T$; and (3) There are no cycles in T. The score of a DT T is finally defined by:

$$score(T) = \sum_{part \subseteq T} W_{part}$$

The inference problem in this model is to find the highest scoring DT in the input weighted graph.

3 Basic Greedy Inference

We start with a basic greedy algorithm (Algorithm 1), analyze the approximation it provides for the graph-based objective and its inherent limitations.

[1] More generally, a part is defined by two arc subsets, A and B, such that a part p belongs to a tree T if $\forall e \in A : e \in T$ and $\forall e \in B : e \notin T$. In this paper we assume $B = \phi$. Hence, we cannot experiment with the third order TurboParser as in all its third order parts $B \neq \phi$. Also, when we integrate our algorithms into the second order TurboParser we omit the nextSibling part for which $B \neq \phi$. For the original TurboParser to which we compare our results, we do not omit this part as it improves the parser's performance.

Algorithm 1 maintains a *partial tree* data structure, T^i, to which it iteratively adds arcs from the input graph G, one in each iteration, until a dependency tree T^n is completed. For this end, in every iteration, i, a value, v_e^i, composed of $loss_e^i$ and $gain_e^i$ terms, is computed for every arc $e \in E$ and the arc with the lowest v_e^i value is added to T^{i-1} to create the extended partial tree T^i.

Due to the aforementioned conditions on the induced dependency tree, every arc that is added to T^{i-1} yields a set of *lostArcs* and *lostParts* that cannot be added to the partial tree in subsequent iterations. The loss value is defined to be:

$$loss_e^i := \sum_{part \in lostParts} W_{part}$$

That is, every part that contains one or more arcs that violate the dependency tree conditions for a tree that extends the partial tree $T^{i-1} \cup \{e\}$ is considered a *lost part* as it will not be included in any tree extending $T^{i-1} \cup \{e\}$. The loss value sums the weights of these parts.

Likewise, the gain value is the sum of the weights of all the parts that are added to T^{i-1} when we add e to it. Denote this set of parts with $P_e := \{part : part \subseteq T^{i-1} \cup \{e\}, part \notin T^{i-1}\}$, then:

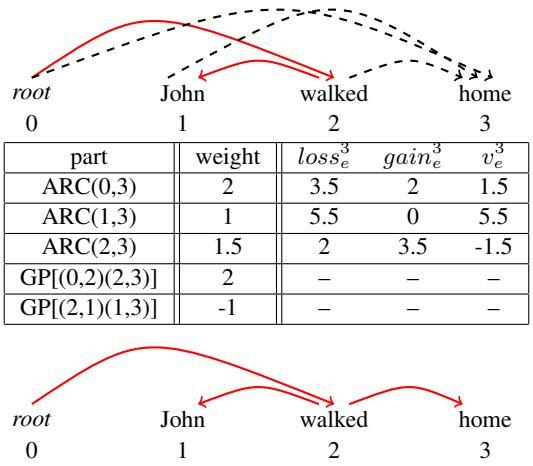
$$gain_e^i := \sum_{part \in P_e} W_{part}$$

Finally, v_e^i is given by:

$$v_e^i = loss_e^i - gain_e^i$$

After the arc with the minimal value v_e^i is added to T^{i-1}, the arcs that violate the structural constraints on dependency trees are removed from G.

An example of an update iteration of the algorithm (lines 3-16) is given in Figure 1. In this example we consider two types of parts: first-order, arc, parts (ARC) and second-order grandparent parts (GP), consisting of arc pairs, $\{(g,u),(u,v)\}$. The upper graph shows the partial tree T^2 (solid arcs) as well as the rest of the graph G (dashed arcs). The parts included in T^2 are ARC(0,2), ARC(2,1) and GP[(0,2),(2,1)]. The table contains the weights of the parts and the values computed for the arcs during the third iteration. The arc that is chosen is (2,3), as it has the minimal v_e^3 value. Thus, in the bottom graph that corresponds to T^3 all other incoming arcs to vertex 3 are removed. (in this instance there are no cycle forming arcs).



part	weight	$loss_e^3$	$gain_e^3$	v_e^3
ARC(0,3)	2	3.5	2	1.5
ARC(1,3)	1	5.5	0	5.5
ARC(2,3)	1.5	2	3.5	-1.5
GP[(0,2)(2,3)]	2	–	–	–
GP[(2,1)(1,3)]	-1	–	–	–

Figure 1: An example of an iteration of Algorithm 1 (lines 3-16)). See description in text.

Analysis We now turn to an analysis of the relaxation that Algorithm 1 provides for the global graph-based objective. Recall that our objective in iteration i is: $v_{e^i}^i = min\{v_e^i\}$. For the inferred tree T^n it holds that:

$$\sum_{e^i \in T^n} v_{e^i}^i - \sum_{part \subseteq G} W_{part} =$$

$$\sum_{part \notin T^n} W_{part} - \sum_{part \subseteq T^n} W_{part} - \sum_{part \subseteq G} W_{part} =$$

$$-2 \times \sum_{part \subseteq T^n} W_{part} + \sum_{part \notin T^n} W_{part} - \sum_{part \notin T^n} W_{part} =$$

$$-2 \times \sum_{part \subseteq T^n} W_{part}$$

The first equation holds since $\sum_{e^i \in T^n} v_{e^i}^i$ is the sum of all lost parts (parts that are not in T^n) minus all the gained parts (parts in T^n). Each of these parts was counted exactly once: when the part was added to the partial tree or when one of its arcs was removed from G. The second equation splits the term of $\sum_{part \subseteq G} W_{part}$ to two sums, one over parts in T^n and the other over the rest. Since $\sum_{part \subseteq G} W_{part}$ and 2 are constants, we get:

$$\arg\min_{T^n}(-\sum_{part \subseteq T^n} W_{part}) = \arg\min_{T^n} \sum_{e^i \in T^n} v_{e^i}^i$$

From this argument it follows that our inference algorithm performs sequential greedy optimization over the presented factorization of the graph-based

objective instead of optimizing the sum of terms, and hence the objective, globally.

The main limitation of Algorithm 1 is that it does not take into account high order parts contribution until the part is actually added to T. For example, in Figure 1, when the arc $(2, 1)$ is added, the part GP[(2,1),(1,3)] is getting closer to completion. Yet, this is not taken into account when considering whether $(2, 1)$ should be added to the tree or not. Including this information in the gain and loss values of an arc can improve the accuracy of the algorithm, especially in high-order parsing.

Algorithm 1 Basic Greedy Inference

1: $T^0 = \{\}$
2: **for** $i \in 1..n$ **do**
3: **for** $e = (u, v) \in E$ **do**
4: $P_e := \{part \in parts : part \subseteq T^{i-1} \cup \{e\}, part \nsubseteq T^{i-1}\}$
5: $gain_e^i := \sum_{part \in P_e} W_{part}$
6: $incomingSet := \{(u', v) \in E : u' \neq u\}$
7: $cycleSet := \{(u', v') \in E : T^{i-1} \cup \{e\} \cup (u', v')\ contains\ a\ cycle\}$
8: $lostArcs = (incomingSet \cup cycleSet)$
9: $lostParts = \{part : \exists e \in lostArcs \cap part\}$
10: $loss_e^i := \sum_{part \in lostParts} W_{part}$
11: $v_e^i := loss_e^i - gain_e^i$
12: **end for**
13: $e^i = (u^i, v^i) = \arg\min_{e'}\{v_{e'}^i\}$
14: $T^i = T^{i-1} \cup \{e^i\}$
15: remove from G all incoming arcs to v^i
16: remove from G all cycle forming arcs w.r.t T^i
17: **end for**

4 Greedy Inference with Partial Part Predictions

In order for the algorithm to account for information about partial high order parts, we estimate the probability that such parts would be eventually included in T^n. Our way to do this (Algorithm 2) is by estimating these probabilities for arcs and from these derive parts probabilities.

Particularly, for the set of incoming arcs of a vertex v, $E_v = \{e = (u, v) : e \in E\}$, a probability measure p_e is computed according to:

$$p_{e=(u,v)} = \frac{exp^{\alpha \times W_e}}{\sum_{e'=(u',v)} exp^{\alpha \times W_{e'}}}$$

Where α is a hyper parameter of the model. For $\alpha = 0$ we get a uniform distribution over all possible heads of a vertex v, and for large α values arcs with larger weights get higher probabilities.

Algorithm 2 Greedy Inference with Partial Part Predictions

1: $T^0 = \{\}$
2: **for** $i \in 1..n$ **do**
3: **for** $e = (u, v) \in E$ **do**
4: $P_e := \{part \in parts : e \in part\}$
5: $gain_e^i := \sum_{part \in P_e} W_{part} \times p_{part}|(T^{i-1} \cup \{e\})$
6: $incomingSet := \{(u', v) \in E : u' \neq u\}$
7: $cycleSet := \{(u', v') \in E : T^{i-1} \cup \{e\} \cup (u', v')\ contains\ a\ cycle\}$
8: $lostArcs = (incomingSet \cup cycleSet)$
9: $lostParts = \{part : \exists e \in lostArcs \cap part\}$
10: $loss_e^i := \sum_{part \in lostParts} W_{part} \times p_{part}|T^{i-1}$
11: $v_e^i := \beta \times loss_e^i - (1 - \beta) \times gain_e^i$
12: **end for**
13: $e^i = (u^i, v^i) = \arg\min_{e'}\{v_{e'}^i\}$
14: $T^i = T^{i-1} \cup \{e^i\}$
15: remove from G all incoming arcs to v^i
16: remove from G all cycle forming arcs w.r.t T^i
17: **end for**

The intuition behind this measure is that arcs mostly compete with other arcs that have the same target vertex and hence their weight should be normalized accordingly. Using this measure, we define the arc-factored probability of a part to be:

$$p_{part} = \prod_{e \in part} p_e$$

And the residual probability of a part given an existing partial tree T:

$$p_{part}|T = \frac{p_{part}}{\prod_{e \in (part \bigcap T)} p_e}$$

These probability measures are used in both the gain and the loss computations (lines 5 and 10 in Algorithm 2) as follows:

$$gain_e^i := \sum_{part: e \in part} W_{part} \times p_{part}|(T^{i-1} \cup \{e\})$$

$$loss_e^i := \sum_{part \in lostParts} W_{part} \times p_{part}|T^{i-1}$$

Finally, as adding an arc to the dependency subtree results in an exclusion of several arcs, the number of lost parts is also likely to be much higher than the number of gained parts. In order to compensate for this effect, we introduce a balancing

hyper-parameter, $\beta \in [0,1]$, and change the computation of v_e^i (line 10 in Algorithm 2) to be: $v_e^i := \beta \times loss_e^i - (1-\beta) \times gain_e^i$.

5 Runtime Complexity Analysis

In this section we provide a sketch of the runtime complexity analysis of the algorithm. Full details are in appendix A. In what follows, we denote the maximal indegree of a vertex with n_{in}.

Algorithm 1 Algorithm 1 consists of two nested loops (lines 2-3) and hence lines 4-11 are repeated $O(n \times |E|) = O(n \times n \times n_{in})$ times. At each repetition, loss (lines 6-10) and gain (lines 4-5) values are computed. Afterwards the graph's data structures are updated (lines 13-16). We define data structures (DSs) that keep our computations efficient. With these DSs the total runtime of lines 4-11 is $O(n_{in} + min\{n_{in}^{k-1}, n_{in}^2\})$. The DSs are initialized in $O(|parts| \times k)$ time and their total update time is $O(k \times |parts|) = O(n_{in}^{k+1})$. Thus algorithm 1 runs in $O(|parts| \times k + n^2 \times n_{in} \times (n_{in} + min\{n_{in}^{k-1}, n_{in}^2\}))$ time.

Algorithm 2 Algorithm 2 is similar in structure to Algorithm 1. The enhanced loss and gain computations take $O(n_{in} + min\{n_{in}^{k-1}, n_{in}^2\})$ time. The initialization of the DSs takes $O(|parts| \times k)$ time and their update time is $O(n_{in}^k \times k^2)$. The total runtime of Algorithm 2 is $O(n_{in}^{k+1} \times k + n \times (n \times n_{in} \times (n_{in} + min\{n_{in}^2, n_{in}^{k-1}\}) + n_{in}^k \times k^2))$. For unpruned graphs and $k \geq 2$ this is equivalent to $O(n^{k+1})$, the theoretical runtime of the TurboParser's dual decomposition inference algorithm.

6 Error Propagation

Unlike modern approximation algorithms for our problem, our algorithm is greedy and deterministic. That is, in each iteration it selects an arc to be included in its final dependency tree and this decision cannot be changed in subsequent iterations. Hence, our algorithm is likely to suffer from error-propagation. We propose two solutions to this problem described within Algorithm 2.

Beam search In each iteration (lines 3-16) the algorithm outputs its $|B|$ best solutions to be subsequently considered in the next iteration. That is, lines 4-10 are performed $|B|$ times for each edge $e \in E$, one for each of the $|B|$ partial solutions in the beam, $b^j \in B$. For each such solution, we denote its weight, as calculated by the previous iteration of the algorithm with $beamVal_{b^j}$. When evaluating v_e^i for an arc e with respect to b^j (line 11), we set $v_e^{i,j} = beamVal_{b^j} + \beta \times loss_e^i - (1-\beta) \times gain_e^i$.

Post-search improvements After Algorithm 2 is executed, we perform s iterations of local greedy arc swaps. That is, for every vertex v, s.t. $(u,v) \in T^n$, we try to switch the arc (u,v) with the arc (u',v) as follows. Let T_v^n be the sub tree that is rooted at v, we distinguish between two cases:
(1) If $u' \notin T_v^n$ then $T^n = T^n \setminus \{(u,v)\} \cup \{(u',v)\}$.
(2) If $u' \in T_v^n$ then let w be the first vertex on the path from v to u' (if $(v,u') \in T$ then $w = u'$): $T^n = T^n \setminus \{(u,v),(v,w)\} \cup \{(u',v),(u,w)\}$.

After inspecting all possible substitutions, we choose the one that yields the best increase in the tree score (if such a substitution exists) and perform the substitution.

7 Parser Combination

In our experiments (see below), we implemented our algorithms within the TurboParser so that each of them, in turn, serves as its inference algorithm. In development data experiments with Algorithm 2 we found that for first order parsing, both our algorithm and the TurboParser predict on average over all languages around 1% of the gold arcs that are not included in the output of the other algorithm. For second order parsing, the corresponding numbers are 1.75% (for gold arcs in the output of our algorithm but not of the original TurboParser) and 4.3% (for the other way around). This suggests that an ensemble method may improve upon both parsers.

We hence introduce a variation of Algorithm 2 that accepts a dependency tree T_o as an input, and biases its output towards that tree. As different parsers usually generate weights on different scales, we do not directly integrate part weights. Instead, we change the weight of each part $part \subseteq T_o$ of order j, to be $W_{part} = W_{part} + \gamma_j$, where γ_j is an hyperparameter reflecting our belief in the prediction of the other parser on parts of order j. The change is applied only at test time, thus integrating two pre-trained parsers.

8 Experimental Setup

We implemented our algorithms within the TurboParser (Martins et al., 2013)[2]. That is, every other aspect of the parser – feature set, pruning algorithm, cost-augmented MIRA training (Crammer et al., 2006) etc., is kept fixed but our algorithms replace the inference algorithms: Chu-Liu-Edmonds ((Edmonds, 1967), first order) and dual-decomposition (higher order). We implemented two variants, for algorithm 1 and 2 respectively, and compare their results to those of the original TurboParser.

We experiment with the datasets of the CoNLL 2006 and 2007 shared task on multilingual dependency parsing (Buchholz and Marsi, 2006; Nilsson et al., 2007), for a total of 17 languages. When a language is represented in both sets, we used the 2006 version. We followed the standard train/test split of these datasets and, for the 8 languages with a training set of at least 10000 sentences, we randomly sampled 1000 sentences from the training set to serve as a development set. For these languages, we first trained the parser on the training set and then used the development set for hyperparameter tuning ($|B|$, s, α, β, and $\gamma_1, \ldots, \gamma_k$ for k order parsing).[3][4]

We employ four evaluation measures, where every measure is computed per language, and we report the average across all languages: (1) Unlabeled Attachment Score (UAS); (2) Undirected UAS (UUAS) - for error analysis purposes; (3) Shared arcs (SARC) - the percentage of arcs shared by the predictions of each of our algorithms and of the original TurboParser; and (4) Tokens per second (TPS) - for ensemble models this measure includes the TurboParser's inference time.[5] We also report a *gold(x,y) = (a,b)* measure: where a is the percentage of gold standard arcs included in trees produced by algorithm x but not by y, and b is the corresponding number for y and x. We consider two setups.

[2]https://github.com/andre-martins/TurboParser

[3]$|B| = 3$, $s = 5$, $\alpha \in [0, 2.5]$, $\beta \in [0.2, 0.5]$, $\gamma_1 \in [0.5, 1.5]$, $\gamma_2 \in [0.2, 0.3]$. Our first order part weights are in $[-9, 4]$, and second order part weights in $[-3, 13]$.

[4]The original TurboParser is trained on the training set of each language and tested on its test set, without any further division of the training data to training and development sets.

[5]Run times where computed on an Intel(R) Xeon(R) CPU E5-2697 v3@2.60GHz machine with 20GB RAM memory.

Fully Supervised Training In this setup we only consider the 8 languages with a development set. For each language, the parser is trained on the training set and then the hyperparameters are tuned. First we set the beam size ($|B|$) and number of improvement iterations (s) to 0, and tune the other hyperparameters on the language-specific development set. Then, we tune $|B|$ and s, using the optimal parameters of the first step, on the English dev. set.

Minimally Supervised Training Here we consider all 17 languages. For each language we randomly sampled 20 training sets of 500 sentences from the original training set, trained a parser on each set and tested on the original test set. Results for each language were calculated as the average over these 20 folds. The hyper parameters for all languages were tuned once on the English development set to the values that yielded the best average results across the 20 training samples.

9 Results

Fully Supervised Training Average results for this setup are presented in table 1 (top). Unsurprisingly, UAS for second order parsing with basic greedy inference (Algorithm 1, BGI) is very low, as this model does not take information about partial high order parts into account in its edge scores. We hence do not report more results for this algorithm.

The table further reflects the accuracy/runtime tradeoff provided by Algorithm 2 (basic greedy inference with partial part predictions, BGI-PP): a UAS degradation of 0.34% and 2.58% for first and second order parsing respectively, with a runtime improvement by factors of 1.01 and 2.4, respectively. Employing beam search and post search improvements (BGI-PP+i+b) to compensate for error propagation improves UAS but harms the runtime gain: for example, the UAS gap in second order parsing is 1.01% while the speedup factor is 1.43.

As discussed in footnote 1 and Section 11, our algorithm does not support the third-order parts of the TurboParser. However, the average UAS of the third-order TurboParser is 90.62% (only 0.36% above second order TurboParser) and its TPS is 72.12 (almost 5 times slower).

The accuracy gaps according to UAS and undirected UAS are similar, indicating that the source

Fully supervised		Individual Models				Ensemble Models			
		UAS	TPS	SARC	U-UAS	UAS	TPS	SARC	U-UAS
TurboParser	order1	87.98	5621.30	–	88.82	–	–	–	–
	order2	90.26	356.63	–	90.98	–	–	–	–
BGI	order1	83.78	5981.91	90.87	90.87	–	–	–	–
	order2	27.54	715.41	27.76	27.77	–	–	–	–
BGI-PP	order1	87.64	5680.60	97.15	88.53	88.03	2876.03	99.59	88.84
	order2	87.68	858.25	92.66	88.73	90.50	249.40	99.54	91.20
BGI-PP + i	order1	87.76	4648.4	98.10	88.64	87.96	2557.00	99.47	88.80
	order2	88.98	639.97	94.40	89.81	90.50	297.10	99.43	91.19
BGI-PP + i + b	order1	87.78	3253.80	98.29	88.73	87.91	2053.00	99.07	88.82
	order2	89.25	511.47	94.79	90.02	90.53	212.40	99.40	91.21

(a) The fully supervised setup.

Minimally supervised		Individual Models				Ensemble Models			
		UAS	TPS	SARC	U-UAS	UAS	TPS	SARC	U-UAS
TurboParser	order1	78.99	13097.00	–	80.38	–	–	–	
	order2	80.52	830.05	–	81.84	–	–	–	
BGI-PP	order1	78.76	13848.00	85.36	80.15	79.14	6499.00	87.36	80.50
	order2	78.80	3089.40	84.59	80.27	80.60	636.30	95.57	81.88
BGI-PP + i	order1	78.87	11673.00	85.54	80.25	79.24	6516.00	87.55	80.59
	order2	79.36	2414.00	84.81	80.76	80.67	621.50	95.41	82.16
BGI-PP + i + b	order1	78.91	4212.50	85.58	80.29	79.29	4349.00	87.61	80.62
	order2	79.45	1372.70	84.89	80.84	80.69	518.10	95.44	81.96

(b) The minimally supervised setup.

Table 1: Results for the fully supervised (top table) and minimally supervised (bottom table) setups. The left column section of each table is for individual models while the right column section is for ensemble models (Section 7). BGI-PP is the basic greedy inference algorithm with partial part predictions, +i indicates post-search improvements and +b indicates beam search (Section 6). The Tokens per Second (TPS) measure for the ensemble models reports the additional inference time over the TurboParser inference. All scores are averaged across individual languages.

of differences between the parsers is not arc directionality. The percentage of arcs shared between the parsers increases with model complexity but is still as low as 94.79% for BGI-PP+i+b in second order parsing. In this setup, gold(BGI-PP+i+b, TurboParser) = (1.6%,2.6%) which supports the development data pattern reported in Section 6 and further justifies an ensemble approach.

The right column section of the table indeed shows consistent improvements of the ensemble models over the TurboParser for second order parsing: the ensemble models achieve UAS of 90.5-90.53% compared to 90.26% of the TurboParser. Naturally, running the TurboParser alone is faster by a factor of 1.67. Like for the individual inference algorithms, the undirected UAS measure indicates that the gain does not come from arc directionality improvements. The ensemble methods share almost all of their arcs with the TurboParser, but in cases of disagreement ensembles tend to be more accurate.

Table 2 complements our results, providing UAS values for each of the 8 languages participating in this setup. The UAS difference between BGI+PP+i+b and the TurboParser are (+0.24)-(-0.71) in first order parsing and (+0.18)-(-2.46) in second order parsing. In the latter case, combining these two models (BGI+PP+i+b+e) yields improvements over the TurboParser in 6 out of 8 languages.

Minimally Supervised Training Results for this setup are in table 1 (bottom). While result patterns are very similar to the fully supervised case, two observations are worth mentioning. First, the percentage of arcs shared by our algorithms and the original parser is much lower than in the fully supervised case. This is true also for shared gold arcs: gold(BGI-PP+b+i,TurboParser) = (4.86%,5.92%) for second order parsing. This suggests that more sophisticated ensemble techniques may be useful in this setup.

Second, ensemble modeling improves UAS over the TurboParser also for first order parsing, leading to a gain of 0.3% in UAS for the BGI+i+b ensemble (79.29% vs. 78.99%). As the percentage of shared arcs between the ensemble models and the TurboParser is particularly low in first order parsing, as well as the shared gold arcs

language	First Order				Second Order			
	TurboParser	BGI-PP	BGI-PP +i+b	BGI-PP +i+b+e	TurboParser	BGI-PP	BGI-PP +i+b	BGI-PP +i+b+e
swedish	**87.12**	86.35	86.93	**87.12**	88.65	86.14	87.85	**89.29**
bulgarian	**90.66**	90.22	90.42	**90.66**	92.43	89.73	91.50	**92.58**
chinese	**84.88**	83.89	84.17	84.17	86.53	81.33	85.18	**86.59**
czech	**83.53**	83.46	83.44	83.44	86.35	84.91	86.26	**87.50**
dutch	88.48	**88.56**	88.43	88.43	91.30	89.64	90.49	**91.34**
japanese	93.03	93.18	**93.27**	**93.27**	93.83	93.78	**94.01**	**94.01**
catalan	**88.94**	88.50	88.67	88.93	**92.25**	89.3	90.46	92.24
english	**87.18**	86.94	86.84	**87.18**	**90.70**	86.52	88.24	90.66

Table 2: Per language UAS for the fully supervised setup. Model names are as in Table 1, 'e' stands for ensemble. Best results for each language and parsing model order are highlighted in bold.

(gold(BGI+i+b,TurboParser) = (4.98%,5.5%)), improving the ensemble techniques is a promising future research direction.

10 Related Work

Our work brings together ideas that have been considered in past, although in different forms.

Greedy Inference Goldberg and Elhadad (2010) introduced an easy-first, greedy, approach to dependency parsing. Their algorithm adds at each iteration the best candidate arc, in contrast to the left to right ordering of standard transition based parsers. This work is extended at (Tratz and Hovy, 2011; Goldberg and Nivre, ; Goldberg and Nivre, 2013).

The easy-first parser consists of a feature set and a specialized variant of the structured perceptron training algorithm, both dedicated to greedy inference. In contrast, we show that a variant of the TurboParser that employs Algorithm 2 for inference and is trained with its standard global training algorithm, performs very similarly to the same parser that employs dual decomposition inference.

Error Propagation in Deterministic Parsing Since deterministic algorithms are standard in transition-based parsing, the error-propagation problem has been dealt with in that context. Various methods were employed, with beam search being a prominent idea (Sagae and Lavie, 2006; Titov and Henderson, 2007; Zhang and Clark, 2008; Huang et al., 2009; Zhang and Nivre, 2011; Bohnet and Nivre, 2012; Choi and McCallum, 2013b, inter alia).

Post Search Improvements Several previous works employed post-search improvements techniques. Like in our case, these techniques improve the tree induced by an initial, possibly more principled, search technique through local, greedy steps.

McDonald and Pereira (2006) proposed to approximate high-order graph-based non-projective parsing, by arc-swap iterations over a previously induced projective tree. Levi et al. (2016) proposed a post-search improvements method, different than ours, to compensate for errors of their graph-based, undirected inference algorithm. Finally, Zhang et al. (2014a) demonstrated that multiple random initialization followed by local improvements with respect to a high-order parsing objective result in excellent parsing performance. Their algorithm, however, shouldbhhb employ hundreds of random initializations in order to provide state-of-the-art results.

Ensemble Approaches Finally, several previous works combined dependency parsers. These include Nivre and McDonald (2008) who used the output of one parser to provide features for another, Zhang and Clark (2008) that proposed a beam-search based parser that combines two parsers into a single system for training and inference, and Martins et al. (2008) that employed stacked learning, in which a second predictor is trained to improve the performance of the first. Our work complements these works by integrating information from a pre-trained TurboParser in our algorithm at test time only.

11 Discussion

We presented a greedy inference approach for graph-based, high-order, non-projective dependency parsing. Our experiments with 17 languages show that our simple and easy to implement algorithm is a decent alternative for dual-decomposition inference.

A major limitation of our algorithm is in-

cluding information from parts that require a given set of arcs *not* to be included in the dependency tree (footnote 1). For example, the $nextSibling((1,2),(1,5))$ part of the TurboParser would fire *iff* the tree includes the arcs $(1,2)$ and $(1,5)$ but not the arcs $(1,3)$ and $(1,4)$.

In order to account for such parts, we should decide how to compute their probabilities and, additionally, at which point they are considered part of the tree. We explored several approaches, but failed to improve our results. Hence, we did not experiment with the third-order TurboParser as all of its third-order parts contain "non-included" arcs. This is left for future work.

A Runtime Complexity Analysis

Here we analyze the complexity of our algorithms, denoting the maximal indegree of a vertex with n_{in}.

Algorithm 1 Algorithm 1 consists of two nested loops (lines 2-k3) and hence lines 4-11 are repeated $O(n \times |E|) = O(n \times n \times n_{in})$ times. At each repetition, loss (lines 6-10) and gain (lines 4-5) values are computed. Afterwards the graph's data structures are updated (lines 13-16).

For every arc that we examine (line 3), there are $O(n_{in})$ *lost* arcs, as there are $O(n_{in})$ incoming arcs (set 1) and $O(n_{in})$ cycles to break (set 2). Since every lost arc translates to a set of *lost parts*, we can avoid repeating computations by storing the partial loss of every arc in a data structure (DS): $e \rightarrow \sum_{part:e \in part} w_{part}$. Now, instead of summing all the lost parts, (every edge participates in $O(n_{in}^{k-1})$ parts,[6] thus there are $O(n_{in}^{k})$ lost parts per added arc), we can sum only $O(n_{in})$ partial loss values. However, since some lost parts may contain an arc from set 1 and an arc from set 2, we need to subtract the values that were summed twice, this can be done in $O(min\{n_{in}^{k-1}, n_{in}^{2}\})$ time by holding a second DS: $e_1 \times e_2 \rightarrow \sum_{part:e_1 \in part \wedge e_2 \in part} w_{part}$.[7]

In order to efficiently compute the gain values, we hold a mapping from arcs to the sum of weights of parts that can be completed in the current iteration by adding the arc to the tree. With this DS, gain values can be computed in constant time. In total, the runtime of lines 4-11 is $O(n_{in} + min\{n_{in}^{k-1}, n_{in}^{2}\})$.

The DSs are initialized in $O(|parts| \times k)$ time. Since every part is deleted at most once, and gets updated (its arcs are added to the tree) at most k times, the total DS update time is $O(k \times |parts|) = O(n_{in}^{k+1})$. Thus algorithm 1 runs in $O(|parts| \times k + n^2 \times n_{in} \times (n_{in} + min\{n_{in}^{k-1}, n_{in}^{2}\}))$ time.

Algorithm 2 Algorithm 2 is similar in structure to Algorithm 1 but the loss and gain computations are more complex. To facilitate efficiency, we hold two DSs: (a) a mapping from arcs to the sum of *lost parts* values, which are now $w_{part} \times P_{part}$ for $part \in parts$; and (b) a mapping from arc pairs to the sum of part values for parts that contain both arcs. The loss and gain values can be computed, as above, in $O(n_{in} + min\{n_{in}^{k-1}, n_{in}^{2}\})$ time.

The initialization of the DSs takes $O(|parts| \times k)$ time. In the i-th iteration we add $e = (u, v)$ to T^i, and remove the $lostArcs$ from E. Every lost arc participates in $O(n_{in}^{k-1})$ parts, and we need to update $O(k)$ entries for each lost part in DS(a) (as the value of the other arcs of that part should no longer account for that part's weight) and $O(k^2)$ entries in DS (b). Thus, the total update time of the DSs is $O(n_{in}^{k} \times k^2)$ and the total runtime of Algorithm 2 is $O(n_{in}^{k+1} \times k + n \times (n \times n_{in} \times (n_{in} + min\{n_{in}^{2}, n_{in}^{k-1}\}) + n_{in}^{k} \times k^2))$. For unpruned graphs and $k \geq 2$ this is equivalent to $O(n^{k+1})$, the theoretical runtime of the TurboParser's dual decomposition inference algorithm.

Acknowledgments

The third author was partly supported by a research grant from the Microsoft/Technion research center for electronic commerce: Context Sensitive Sentence Understanding for Natural Language Processing.

References

Mariana SC Almeida, Cláudia Pinto, Helena Figueira, Pedro Mendes, and André FT Martins. 2015. Aligning opinions: Cross-lingual opinion mining with dependencies. In *ACL*.

Bernd Bohnet and Joakim Nivre. 2012. A transition-based system for joint part-of-speech tagging and labeled non-projective dependency parsing. In *Pro-*

[6] Assuming that a part is a connected component.

[7] For first order parsing this is not needed; for second order parsing it is done in $O(n_{in})$ time.

ceedings of EMNLP-CoNLL. Association for Computational Linguistics.

Sabine Buchholz and Erwin Marsi. 2006. Conll-x shared task on multilingual dependency parsing. In *CoNLL*.

Xavier Carreras. 2007. Experiments with a higher-order projective dependency parser. In *EMNLP-CoNLL*.

Jinho D Choi and Andrew McCallum. 2013a. Transition-based dependency parsing with selectional branching. In *ACL*.

Jinho D Choi and Andrew McCallum. 2013b. Transition-based dependency parsing with selectional branching. In *ACL*.

Koby Crammer, Ofer Dekel, Joseph Keshet, Shai Shalev-Shwartz, and Yoram Singer. 2006. Online passive-aggressive algorithms. *The Journal of Machine Learning Research*, 7:551–585.

J. Edmonds. 1967. Optimum branchings. *Journal of Research of the National Bureau of Standards*, 71B:233–240.

Jason Eisner. 1996. Efficient normal-form parsing for combinatory categorial grammar. In *ACL*.

Yoav Goldberg and Michael Elhadad. 2010. An efficient algorithm for easy-first non-directional dependency parsing. In *NAACL-HLT*.

Yoav Goldberg and Joakim Nivre. A dynamic oracle for arc-eager dependency parsing. In *COLING*.

Yoav Goldberg and Joakim Nivre. 2013. Training deterministic parsers with non-deterministic oracles. *Transactions of the Association for Computational Linguistics*, 1(Oct):403–414.

Matthew Gormley, Mark Dredze, and Jason Eisner. 2015. Approximation-aware dependency parsing by belief propagation. *Transactions of the Association for Computational Linguistics*, 3:489–501.

Matthew Honnibal, Yoav Goldberg, and Mark Johnson. 2013. A non-monotonic arc-eager transition system for dependency parsing. In *CoNLL*.

Liang Huang, Wenbin Jiang, and Qun Liu. 2009. Bilingually-constrained (monolingual) shift-reduce parsing. In *EMNLP*.

Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *ACL*.

Effi Levi, Roi Reichart, and Ari Rappoport. 2016. Edge-linear first-order dependency parsing with undirected minimum spanning tree inference. In *ACL*.

Omer Levy and Yoav Goldberg. 2014. Neural word embedding as implicit matrix factorization. In *NIPS*.

André FT Martins, Dipanjan Das, Noah A Smith, and Eric P Xing. 2008. Stacking dependency parsers. In *EMNLP*.

A. Martins, M. Almeida, and N. A. Smith. 2013. Turning on the turbo: Fast third-order non-projective turbo parsers. In *ACL*.

Ryan McDonald and Fernando Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *EACL*.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajic. 2005. Non-projective dependency parsing using spanning tree algorithms. In *HLT-EMNLP*.

Jens Nilsson, Sebastian Riedel, and Deniz Yuret. 2007. The conll 2007 shared task on dependency parsing. In *Proceedings of the CoNLL shared task session of EMNLP-CoNLL*.

Joakim Nivre and Ryan McDonald. 2008. Integrating graph-based and transition-based dependency parsers. In *ACL-08: HLT*, June.

Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gülsen Eryigit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. 2007. Maltparser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(02):95–135.

Sebastian Riedel, David Smith, and Andrew McCallum. 2012. Parse, price and cut – delayed column and row generation for graph based parsers. In *EMNLP-CoNLL*.

Kenji Sagae and Alon Lavie. 2006. A best-first probabilistic shift-reduce parser. In *Proc. of the COLING/ACL on Main conference poster sessions*.

David Smith and Jason Eisner. 2008. Dependency parsing by belief propagation. In *EMNLP*.

Ivan Titov and James Henderson. 2007. Fast and robust multilingual dependency parsing with a generative latent variable model. In *EMNLP-CoNLL*.

Stephen Tratz and Eduard Hovy. 2011. A fast, accurate, non-projective, semantically-enriched parser. In *EMNLP*.

Fei Wu and Daniel S Weld. 2010. Open information extraction using wikipedia. In *ACL*.

Yue Zhang and Stephen Clark. 2008. A tale of two parsers: Investigating and combining graph-based and transition-based dependency parsing using beam-search. In *EMNLP*.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *ACL*.

Yuan Zhang, Tao Lei, Regina Barzilay, and Tommi Jaakkola. 2014a. Greed is good if randomized: New inference for dependency parsing. In *EMNLP*.

Yuan Zhang, Tao Lei, Regina Barzilay, Tommi Jaakkola, and Amir Globerson. 2014b. Steps to excellence: Simple inference with refined scoring of dependency trees. In *ACL*.