Transition-based Dependency Parsing with Selectional Branching

Jinho D. Choi
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA, 01003, USA
jdchoi@cs.umass.edu

Andrew McCallum
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA, 01003, USA
mccallum@cs.umass.edu

Abstract

We present a novel approach, called selectional branching, which uses confidence estimates to decide when to employ a beam, providing the accuracy of beam search at speeds close to a greedy transition-based dependency parsing approach. Selectional branching is guaranteed to perform a fewer number of transitions than beam search yet performs as accurately. We also present a new transition-based dependency parsing algorithm that gives a complexity of $O(n)$ for projective parsing and an expected linear time speed for non-projective parsing. With the standard setup, our parser shows an unlabeled attachment score of 92.96% and a parsing speed of 9 milliseconds per sentence, which is faster and more accurate than the current state-of-the-art transition-based parser that uses beam search.

1 Introduction

Transition-based dependency parsing has gained considerable interest because it runs fast and performs accurately. Transition-based parsing gives complexities as low as $O(n)$ and $O(n^2)$ for projective and non-projective parsing, respectively (Nivre, 2008).[1] The complexity is lower for projective parsing because a parser can deterministically skip tokens violating projectivity, while this property is not assumed for non-projective parsing. Nonetheless, it is possible to perform non-projective parsing in expected linear time because the amount of non-projective dependencies is notably smaller (Nivre and Nilsson, 2005) so a parser can assume projectivity for most cases while recognizing ones for which projectivity should not be assumed (Nivre, 2009; Choi and Palmer, 2011).

[1] We refer parsing approaches that produce only projective dependency trees as projective parsing and both projective and non-projective dependency trees as non-projective parsing.

Greedy transition-based dependency parsing has been widely deployed because of its speed (Cer et al., 2010); however, state-of-the-art accuracies have been achieved by globally optimized parsers using beam search (Zhang and Clark, 2008; Huang and Sagae, 2010; Zhang and Nivre, 2011; Bohnet and Nivre, 2012). These approaches generate multiple transition sequences given a sentence, and pick one with the highest confidence. Coupled with dynamic programming, transition-based dependency parsing with beam search can be done very efficiently and gives significant improvement to parsing accuracy.

One downside of beam search is that it always uses a fixed size of beam even when a smaller size of beam is sufficient for good results. In our experiments, a greedy parser performs as accurately as a parser that uses beam search for about 64% of time. Thus, it is preferred if the beam size is not fixed but proportional to the number of low confidence predictions that a greedy parser makes, in which case, fewer transition sequences need to be explored to produce the same or similar parse output.

We first present a new transition-based parsing algorithm that gives a complexity of $O(n)$ for projective parsing and an expected linear time speed for non-projective parsing. We then introduce selectional branching that uses confidence estimates to decide when to employ a beam. With our new approach, we achieve a higher parsing accuracy than the current state-of-the-art transition-based parser that uses beam search and a much faster speed.

2 Transition-based dependency parsing

We introduce a transition-based dependency parsing algorithm that is a hybrid between Nivre's arc-eager and list-based algorithms (Nivre, 2003; Nivre, 2008). Nivre's arc-eager is a projective parsing algorithm showing a complexity of $O(n)$. Nivre's list-based algorithm is a non-projective parsing algorithm showing a complexity of $O(n^2)$. Table 1 shows transitions in our algorithm. The top 4 and

Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, pages 1052–1062,
Sofia, Bulgaria, August 4-9 2013.

Transition	Current state	⇒	Resulting state
LEFT$_l$-REDUCE	($[\sigma\|i]$, δ, $[j\|\beta]$, A)	⇒	(σ, δ, $[j\|\beta]$, $A \cup \{i \overset{l}{\leftarrow} j\}$)
RIGHT$_l$-SHIFT	($[\sigma\|i]$, δ, $[j\|\beta]$, A)	⇒	($[\sigma\|i\|\delta\|j]$, [ ], β, $A \cup \{i \overset{l}{\rightarrow} j\}$)
NO-SHIFT	($[\sigma\|i]$, δ, $[j\|\beta]$, A)	⇒	($[\sigma\|i\|\delta\|j]$, [ ], β, A)
NO-REDUCE	($[\sigma\|i]$, δ, $[j\|\beta]$, A)	⇒	(σ, δ, $[j\|\beta]$, A)
LEFT$_l$-PASS	($[\sigma\|i]$, δ, $[j\|\beta]$, A)	⇒	(σ, $[i\|\delta]$, $[j\|\beta]$, $A \cup \{i \overset{l}{\leftarrow} j\}$)
RIGHT$_l$-PASS	($[\sigma\|i]$, δ, $[j\|\beta]$, A)	⇒	(σ, $[i\|\delta]$, $[j\|\beta]$, $A \cup \{i \overset{l}{\rightarrow} j\}$)
NO-PASS	($[\sigma\|i]$, δ, $[j\|\beta]$, A)	⇒	(σ, $[i\|\delta]$, $[j\|\beta]$, A)

Table 1: Transitions in our dependency parsing algorithm.

Transition	Preconditions
LEFT$_l$-$*$	$[i \neq 0] \wedge \neg[\exists k.\ (i \leftarrow k) \in A] \wedge \neg[(i \rightarrow^* j) \in A]$
RIGHT$_l$-$*$	$\neg[\exists k.\ (k \rightarrow j) \in A] \wedge \neg[(i\ {}^*\!\leftarrow j) \in A]$
$*$-SHIFT	$\neg[\exists k \in \sigma.\ (k \neq i) \wedge ((k \leftarrow j) \vee (k \rightarrow j))]$
$*$-REDUCE	$[\exists h.\ (h \rightarrow i) \in A] \wedge \neg[\exists k \in \beta.\ (i \rightarrow k)]$

Table 2: Preconditions of the transitions in Table 1 ($*$ is a wildcard representing any transition).

the bottom 3 transitions are inherited from Nivre's arc-eager and list-based algorithms, respectively.[2]

Each parsing state is represented as a tuple (σ, δ, β, A), where σ is a stack containing processed tokens, δ is a deque containing tokens popped out of σ but will be pushed back into σ in later parsing states to handle non-projectivity, and β is a buffer containing unprocessed tokens. A is a set of labeled arcs. (i, j) represent indices of their corresponding tokens (w_i, w_j), l is a dependency label, and the 0 identifier corresponds to w_0, introduced as the root of a tree. The initial state is ([0], [ ], [$1, \ldots, n$], $\emptyset$), and the final state is (σ, δ, [ ], A). At any parsing state, a decision is made by comparing the top of σ, w_i, and the first element of β, w_j. This decision is consulted by gold-standard trees during training and a classifier during decoding.

LEFT$_l$-$*$ and RIGHT$_l$-$*$ are performed when w_j is the head of w_i with a dependency label l, and vice versa. After LEFT$_l$-$*$ or RIGHT$_l$-$*$, an arc is added to A. NO-$*$ is performed when no dependency is found for w_i and w_j. $*$-SHIFT is performed when no dependency is found for w_j and any token in σ other than w_i. After $*$-SHIFT, all tokens in δ as well as w_j are pushed into σ. $*$-REDUCE is performed when w_i already has the head, and w_i is not the head of any token in β. After $*$-REDUCE, w_i is popped out of σ. $*$-PASS is performed when neither $*$-SHIFT nor $*$-REDUCE can be performed. After $*$-PASS, w_i is moved to the front of δ so it can be compared to other tokens in β later. Each transition needs to satisfy certain preconditions to ensure the properties of a well-formed dependency graph (Nivre, 2008); they are described in Table 2. $(i \leftarrow j)$ and $(i\ {}^*\!\leftarrow j)$ indicate that w_j is the head and an ancestor of w_i with any label, respectively.

When a parser is trained on only projective trees, our algorithm learns only the top 4 transitions and produces only projective trees during decoding. In this case, it performs at most $2n - 1$ transitions per sentence so the complexity is $O(n)$. When a parser is trained on a mixture of projective and non-projective trees, our algorithm learns all transitions and produces both kinds of trees during decoding. In this case, it performs at most $\frac{n(n+1)}{2}$ transitions so the complexity is $O(n^2)$. However, because of the presence of $*$-SHIFT and $*$-REDUCE, our algorithm is capable of skipping or removing tokens during non-projective parsing, which allows it to show a linear time parsing speed in practice.

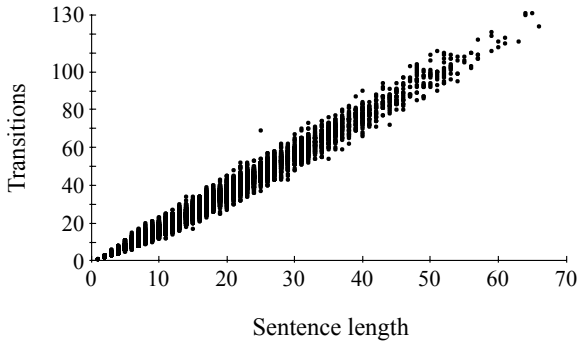
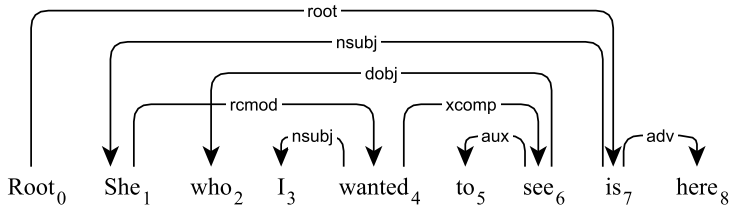


Figure 1: The # of transitions performed during training with respect to sentence lengths for Dutch.

[2]The parsing complexity of a transition-based dependency parsing algorithm is determined by the number of transitions performed with respect to the number of tokens in a sentence, say n (Kübler et al., 2009).



	Transition	σ	δ	β	A
0	Initialization	[0]	[ ]	[1|β]	$\emptyset$
1	NO-SHIFT	[σ|1]	[ ]	[2|β]	
2	NO-SHIFT	[σ|2]	[ ]	[3|β]	
3	NO-SHIFT	[σ|3]	[ ]	[4|β]	
4	LEFT-REDUCE	[σ|2]	[ ]	[4|β]	$A \cup \{3 \leftarrow_{\text{NSUBJ}} 4\}$
5	NO-PASS	[σ|1]	[2]	[4|β]	
6	RIGHT-SHIFT	[σ|4]	[ ]	[5|β]	$A \cup \{1 \rightarrow_{\text{RCMOD}} 4\}$
7	NO-SHIFT	[σ|5]	[ ]	[6|β]	
8	LEFT-REDUCE	[σ|4]	[ ]	[6|β]	$A \cup \{5 \leftarrow_{\text{AUX}} 6\}$
9	RIGHT-PASS	[σ|2]	[4]	[6|β]	$A \cup \{4 \rightarrow_{\text{XCOMP}} 6\}$
10	LEFT-REDUCE	[σ|1]	[4]	[6|β]	$A \cup \{2 \leftarrow_{\text{DOBJ}} 6\}$
11	NO-SHIFT	[σ|6]	[ ]	[7|β]	
12	NO-REDUCE	[σ|4]	[ ]	[7|β]	
13	NO-REDUCE	[σ|1]	[ ]	[7|β]	
14	LEFT-REDUCE	[0]	[ ]	[7|β]	$A \cup \{1 \leftarrow_{\text{NSUBJ}} 7\}$
15	RIGHT-SHIFT	[σ|7]	[ ]	[8]	$A \cup \{0 \rightarrow_{\text{ROOT}} 7\}$
16	RIGHT-SHIFT	[σ|8]	[ ]	[ ]	$A \cup \{7 \rightarrow_{\text{ADV}} 8\}$

Table 3: A transition sequence generated by our parsing algorithm using gold-standard decisions.

Figure 1 shows the total number of transitions performed during training with respect to sentence lengths for Dutch. Among all languages distributed by the CoNLL-X shared task (Buchholz and Marsi, 2006), Dutch consists of the highest number of non-projective dependencies (5.4% in arcs, 36.4% in trees). Even with such a high number of non-projective dependencies, our parsing algorithm still shows a linear growth in transitions.

Table 3 shows a transition sequence generated by our parsing algorithm using gold-standard decisions. After w_3 and w_4 are compared, w_3 is popped out of σ (state 4) so it is not compared to any other token in β (states 9 and 13). After w_2 and w_4 are compared, w_2 is moved to δ (state 5) so it can be compared to other tokens in β (state 10). After w_4 and w_6 are compared, RIGHT-PASS is performed (state 9) because there is a dependency between w_6 and w_2 in σ (state 10). After w_6 and w_7 are compared, w_6 is popped out of σ (state 12) because it is not needed for later parsing states.

3 Selectional branching

3.1 Motivation

For transition-based parsing, state-of-the-art accuracies have been achieved by parsers optimized on multiple transition sequences using beam search, which can be done very efficiently when it is coupled with dynamic programming (Zhang and Clark, 2008; Huang and Sagae, 2010; Zhang and Nivre, 2011; Huang et al., 2012; Bohnet and Nivre, 2012). Despite all the benefits, there is one downside of this approach; it generates a fixed number of transition sequences no matter how confident the one-best sequence is.[3] If every prediction leading to the one-best sequence is confident, it may not be necessary to explore more sequences to get the best output. Thus, it is preferred if the beam size is not fixed but proportional to the number of low confidence predictions made for the one-best sequence.

The selectional branching method presented here performs at most $d \cdot t - e$ transitions, where t is the maximum number of transitions performed to generate a transition sequence, $d = \min(b, |\lambda|+1)$, b is the beam size, $|\lambda|$ is the number of low confidence predictions made for the one-best sequence, and $e = \frac{d(d-1)}{2}$. Compared to beam search that always performs $b \cdot t$ transitions, selectional branching is guaranteed to perform fewer transitions given the same beam size because $d \leq b$ and $e > 0$ except for $d = 1$, in which case, no branching happens. With selectional branching, our parser shows slightly

[3]The 'one-best sequence' is a transition sequence generated by taking only the best prediction at each parsing state.

higher parsing accuracy than the current state-of-the-art transition-based parser using beam search, and performs about 3 times faster.

3.2 Branching strategy

Figure 2 shows an overview of our branching strategy. s_{ij} represents a parsing state, where i is the index of the current transition sequence and j is the index of the current parsing state (e.g., s_{12} represents the 2nd parsing state in the 1st transition sequence). p_{kj} represents the k'th best prediction (in our case, it is a predicted transition) given s_{1j} (e.g., p_{21} is the 2nd-best prediction given s_{11}).

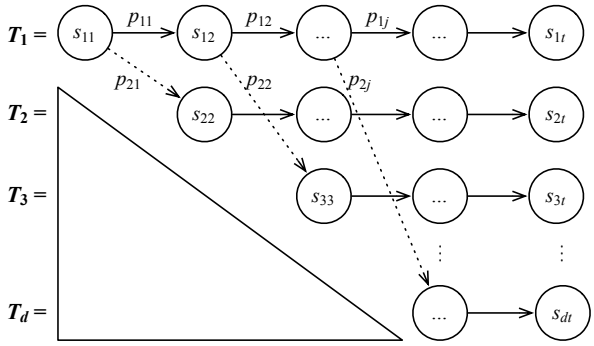


Figure 2: An overview of our branching strategy. Each sequence $T_{i>1}$ branches from T_1.

Initially, the one-best sequence $T_1 = [s_{11}, ..., s_{1t}]$ is generated by a greedy parser. While generating T_1, the parser adds tuples $(s_{1j}, p_{2j}), ..., (s_{1j}, p_{kj})$ to a list λ for each low confidence prediction p_{1j} given s_{1j}.[4] Then, new transition sequences are generated by using the b highest scoring predictions in λ, where b is the beam size. If $|\lambda| < b$, all predictions in λ are used. The same greedy parser is used to generate these new sequences although it now starts with s_{1j} instead of an initial parsing state, applies p_{kj} to s_{1j}, and performs further transitions. Once all transition sequences are generated, a parse tree is built from a sequence with the highest score.

For our experiments, we set $k = 2$, which gave noticeably more accurate results than $k = 1$. We also experimented with $k > 2$, which did not show significant improvement over $k = 2$. Note that assigning a greater k may increase $|\lambda|$ but not the total number of transition sequences generated, which is restricted by the beam size, b. Since each sequence $T_{i>1}$ branches from T_1, selectional branching performs fewer transitions than beam search: at least $\frac{d(d-1)}{2}$ transitions are inherited from T_1, where $d = \min(b, |\lambda| + 1)$; thus, it performs that many transitions less than beam search (see the left lower triangle in Figure 2). Furthermore, selectional branching generates a d number of sequences, where d is proportional to the number of low confidence predictions made by T_1. To sum up, selectional branching generates the same or fewer transition sequences than beam search and each sequence $T_{i>1}$ performs fewer transitions than T_1; thus, it performs faster than beam search in general given the same beam size.

[4] λ is initially empty, which is hidden in Figure 2.

3.3 Finding low confidence predictions

For each parsing state s_{ij}, a prediction is made by generating a feature vector $x_{ij} \in \mathcal{X}$, feeding it into a classifier C^1 that uses a feature map $\Phi(x, y)$ and a weight vector $\mathbf{w}$ to measure a score for each label $y \in \mathcal{Y}$, and choosing a label with the highest score. When there is a tie between labels with the highest score, the first one is chosen. This can be expressed as a logistic regression:

$$C^1(x) = \arg\max_{y \in \mathcal{Y}} \{f(x, y)\}$$

$$f(x, y) = \frac{\exp(\mathbf{w} \cdot \Phi(x, y))}{\sum_{y' \in \mathcal{Y}} \exp(\mathbf{w} \cdot \Phi(x, y'))}$$

To find low confidence predictions, we use the margins (score differences) between the best prediction and the other predictions. If all margins are greater than a threshold, the best prediction is considered highly confident; otherwise, it is not. Given this analogy, the k-best predictions can be found as follows ($m \geq 0$ is a margin threshold):

$$C^k(x, m) = \mathcal{K} \arg\max_{y \in \mathcal{Y}} \{f(x, y)\}$$
$$\text{s.t.} \quad f(x, C^1(x)) - f(x, y) \leq m$$

'$\mathcal{K} \arg\max$' returns a set of k' labels whose margins to $C^1(x)$ are smaller than any other label's margin to $C^1(x)$ and also $\leq m$, where $k' \leq k$. When $m = 0$, it returns a set of the highest scoring labels only, including $C^1(x)$. When $m = 1$, it returns a set of all labels. Given this, a prediction is considered not confident if $|C^k(x, m)| > 1$.

3.4 Finding the best transition sequence

Let P_i be a list of all predictions that lead to generate a transition sequence T_i. The predictions in P_i are either inherited from T_1 or made specifically for T_i. In Figure 2, P_3 consists of p_{11} as its first prediction, p_{22} as its second prediction, and

further predictions made specifically for T_3. The score of each prediction is measured by $f(x, y)$ in Section 3.3. Then, the score of T_i is measured by averaging scores of all predictions in P_i.

$$\text{score}(T_i) = \frac{\sum_{p \in P_i} \text{score}(p)}{|P_i|}$$

Unlike Zhang and Clark (2008), we take the average instead of the sum of all prediction scores. This is because our algorithm does not guarantee the same number of transitions for every sequence, so the sum of all scores would weigh more on sequences with more transitions. We experimented with both the sum and the average, and taking the average led to slightly higher parsing accuracy.

3.5 Bootstrapping transition sequences

During training, a training instance is generated for each parsing state s_{ij} by taking a feature vector x_{ij} and its true label y_{ij}. To generate multiple transition sequences during training, the bootstrapping technique of Choi and Palmer (2011) is used, which is described in Algorithm 1.[5]

Algorithm 1 Bootstrapping

Input: D_t: training set, D_d: development set.
Output: A model M.

1: $r \leftarrow 0$
2: $I \leftarrow$ getTrainingInstances(D_t)
3: $M_0 \leftarrow$ buildModel(I)
4: $S_0 \leftarrow$ getScore(D_d, M_0)
5: **while** $(r = 0)$ or $(S_{r-1} < S_r)$ **do**
6: $r \leftarrow r + 1$
7: $I \leftarrow$ getTrainingInstances(D_t, M_{r-1})
8: $M_r \leftarrow$ buildModel(I)
9: $S_r \leftarrow$ getScore(D_d, M_r)
10: **return** M_{r-1}

First, an initial model M_0 is trained on all data by taking the one-best sequences, and its score is measured by testing on a development set (lines 2-4). Then, the next model M_r is trained on all data but this time, M_{r-1} is used to generate multiple transition sequences (line 7-8). Among all transition sequences generated by M_{r-1}, training instances from only T_1 and T_g are used to train M_r, where T_1 is the one-best sequence and T_g is a sequence giving the most accurate parse output compared to the gold-standard tree. The score of M_r is measured (line 9), and repeat the procedure if $S_{r-1} < S_r$; otherwise, return the previous model M_{r-1}.

[5] Alternatively, the dynamic oracle approach of Goldberg and Nivre (2012) can be used to generate multiple transition sequences, which is expected to show similar results.

3.6 Adaptive subgradient algorithm

To build each model during bootstrapping, we use a stochastic adaptive subgradient algorithm called ADAGRAD that uses per-coordinate learning rates to exploit rarely seen features while remaining scalable (Duchi et al., 2011).This is suitable for NLP tasks where rarely seen features often play an important role and training data consists of a large number of instances with high dimensional features. Algorithm 2 shows our adaptation of ADAGRAD with logistic regression for multi-class classification. Note that when used with logistic regression, ADAGRAD takes a regular gradient instead of a subgradient method for updating weights. For our experiments, ADAGRAD slightly outperformed learning algorithms such as average perceptron (Collins, 2002) or Liblinear SVM (Hsieh et al., 2008).

Algorithm 2 ADAGRAD + logistic regression

Input: $D = \{(x_i, y_i)\}_{i=1}^n$ s.t. $x_i \in \mathcal{X}, y_i \in \mathcal{Y}$
$\Phi(x, y) \in \mathbb{R}^d$ s.t. $d =$ *dimension*$(\mathcal{X}) \times |\mathcal{Y}|$
T: iterations, α: learning rate, ρ: ridge
Output: A weight vector $\mathbf{w} \in \mathbb{R}^d$.

1: $\mathbf{w} \leftarrow 0$, where $\mathbf{w} \in \mathbb{R}^d$
2: $\mathbf{G} \leftarrow 0$, where $\mathbf{G} \in \mathbb{R}^d$
3: **for** $t \leftarrow 1 \ldots T$ **do**
4: **for** $i \leftarrow 1 \ldots n$ **do**
5: $\mathbf{Q}_{\forall y \in \mathcal{Y}} \leftarrow I(y_i, y) - f(x_i, y)$, s.t. $\mathbf{Q} \in \mathbb{R}^{|\mathcal{Y}|}$
6: $\partial \leftarrow \sum_{y \in \mathcal{Y}} (\Phi(x_i, y) \cdot \mathbf{Q}_y)$
7: $\mathbf{G} \leftarrow \mathbf{G} + \partial \circ \partial$
8: **for** $j \leftarrow 1 \ldots d$ **do**
9: $\mathbf{w}_j \leftarrow \mathbf{w}_j + \alpha \cdot \frac{1}{\rho + \sqrt{\mathbf{G}_j}} \cdot \partial_j$

$$I(y, y') = \begin{cases} 1 & y = y' \\ 0 & \text{otherwise} \end{cases}$$

The algorithm takes three hyper-parameters; T is the number of iterations, α is the learning rate, and ρ is the ridge ($T > 0, \alpha > 0, \rho \geq 0$). $\mathbf{G}$ is our running estimate of a diagonal covariance matrix for the gradients (per-coordinate learning rates). For each instance, scores for all labels are measured by the logistic regression function $f(x, y)$ in Section 3.3. These scores are subtracted from an output of the indicator function $I(y, y')$, which forces our model to keep learning this instance until the prediction is 100% confident (in other words, until the score of y_i becomes 1). Then, a subgradient is measured by taking all feature vectors together weighted by $\mathbf{Q}$ (line 6). This subgradient is used to update $\mathbf{G}$ and $\mathbf{w}$, where $\circ$ is the Hadamard product (lines 7-9). ρ is a ridge term to keep the inverse covariance well-conditioned.

4 Experiments

4.1 Corpora

For projective parsing experiments, the Penn English Treebank (Marcus et al., 1993) is used with the standard split: sections 2-21 for training, 22 for development, and 23 for evaluation. All constituent trees are converted with the head-finding rules of Yamada and Matsumoto (2003) and the labeling rules of Nivre (2006). For non-projective parsing experiments, four languages from the CoNLL-X shared task are used: Danish, Dutch, Slovene, and Swedish (Buchholz and Marsi, 2006). These languages are selected because they contain non-projective trees and are publicly available from the CoNLL-X webpage.[6] Since the CoNLL-X data we have does not come with development sets, the last 10% of each training set is used for development.

4.2 Feature engineering

For English, we mostly adapt features from Zhang and Nivre (2011) who have shown state-of-the-art parsing accuracy for transition-based dependency parsing. Their distance features are not included in our approach because they do not seem to show meaningful improvement. Feature selection is done on the English development set.

For the other languages, the same features are used with the addition of morphological features provided by CoNLL-X; specifically, morphological features from the top of σ and the front of β are added as unigram features. Moreover, all POS tag features from English are duplicated with coarse-grained POS tags provided by CoNLL-X. No more feature engineering is done for these languages; it is possible to achieve higher performance by using different features, especially when these languages contain non-projective dependencies whereas English does not, which we will explore in the future.

4.3 Development

Several parameters need to be optimized during development. For ADAGRAD, T, α, and ρ need to be tuned (Section 3.6). For bootstrapping, the number of iterations, say r, needs to be tuned (Section 3.5). For selectional branching, the margin threshold m and the beam size b need to be tuned (Section 3.3). First, all parameters are tuned on the English development set by using grid search on $T = [1, \ldots, 10]$, $\alpha = [0,01, 0,02]$, $\rho = [0.1, 0.2]$, $r = [1, 2, 3]$, $m = [0.83, \ldots, 0.92]$, and $b = [16, 32, 64, 80]$. As a result, the following parameters are found: $\alpha = 0.02$, $\rho = 0.1$, $m = 0.88$, and $b = 64|80$. For this development set, the beam size of 64 and 80 gave the exact same result, so we kept the one with a larger beam size ($b = 80$).

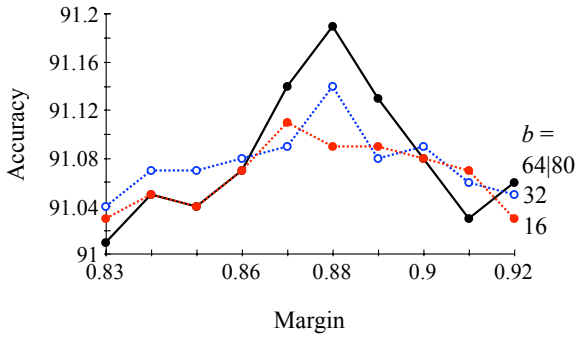


Figure 3: Parsing accuracies with respect to margins and beam sizes on the English development set. $b = 64|80$: the black solid line with solid circles, $b = 32$: the blue dotted line with hollow circles, $b = 16$: the red dotted line with solid circles.

Figure 3 shows parsing accuracies with respect to different margins and beam sizes on the English development set. These parameters need to be tuned jointly because different margins prefer different beam sizes. For instance, $m = 0.85$ gives the highest accuracy with $b = 32$, but $m = 0.88$ gives the highest accuracy with $b = 64|80$.

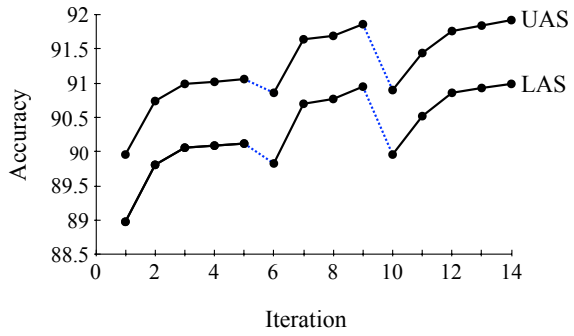


Figure 4: Parsing accuracies with respect to ADAGRAD and bootstrap iterations on the English development set when $\alpha = 0.02$, $\rho = 0.1$, $m = 0.88$, and $b = 64|80$. UAS: unlabeled attachment score, LAS: labeled attachment score.

Figure 4 shows parsing accuracies with respect to ADAGRAD and bootstrap iterations on the English development set. The range 1-5 shows results of 5 ADAGRAD iterations before bootstrapping, the range 6-9 shows results of 4 iterations during the

[6] http://ilk.uvt.nl/conll/

first bootstrapping, and the range 10-14 shows results of 5 iterations during the second bootstrapping. Thus, the number of bootstrap iteration is 2 where each bootstrapping takes a different number of ADAGRAD iterations. Using an Intel Xeon 2.57GHz machine, it takes less than 40 minutes to train the entire Penn Treebank, which includes times for IO, feature extraction and bootstrapping.

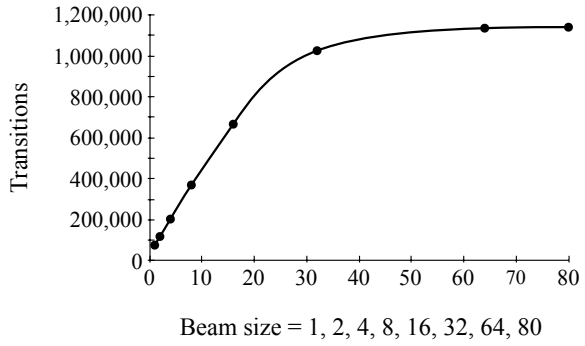


Figure 5: The total number of transitions performed during decoding with respect to beam sizes on the English development set.

Figure 5 shows the total number of transitions performed during decoding with respect to beam sizes on the English development set (1,700 sentences, 40,117 tokens). With selectional branching, the number of transitions grows logarithmically as the beam size increases whereas it would have grown linearly if beam search were used. We also checked how often the one best sequence is chosen as the final sequence during decoding. Out of 1,700 sentences, the one best sequences are chosen for 1,095 sentences. This implies that about 64% of time, our greedy parser performs as accurately as our non-greedy parser using selectional branching.

For the other languages, we use the same values as English for α, ρ, m, and b; only the ADAGRAD and bootstrap iterations are tuned on the development sets of the other languages.

4.4 Projective parsing experiments

Before parsing, POS tags were assigned to the training set by using 20-way jackknifing. For the automatic generation of POS tags, we used the domain-specific model of Choi and Palmer (2012a)'s tagger, which gave 97.5% accuracy on the English evaluation set (0.2% higher than Collins (2002)'s tagger).

Table 4 shows comparison between past and current state-of-the-art parsers and our approach. The first block shows results from transition-based dependency parsers using beam search. The second block shows results from other kinds of parsing approaches (e.g., graph-based parsing, ensemble parsing, linear programming, dual decomposition). The third block shows results from parsers using external data. The last block shows results from our approach. The Time column show how many seconds per sentence each parser takes.[7]

Approach	UAS	LAS	Time
Zhang and Clark (2008)	92.1		
Huang and Sagae (2010)	92.1		0.04
Zhang and Nivre (2011)	92.9	91.8	0.03
Bohnet and Nivre (2012)	93.38	92.44	0.4
McDonald et al. (2005)	90.9		
Mcdonald and Pereira (2006)	91.5		
Sagae and Lavie (2006)	92.7		
Koo and Collins (2010)	93.04		
Zhang and McDonald (2012)	93.06	91.86	
Martins et al. (2010)	93.26		
Rush et al. (2010)	93.8		
Koo et al. (2008)	93.16		
Carreras et al. (2008)	93.54		
Bohnet and Nivre (2012)	93.67	92.68	
Suzuki et al. (2009)	93.79		
$b_t = 80, b_d = 80, m = 0.88$	**92.96**	**91.93**	**0.009**
$b_t = 80, b_d = 64, m = 0.88$	92.96	91.93	0.009
$b_t = 80, b_d = 32, m = 0.88$	92.96	91.94	0.009
$b_t = 80, b_d = 16, m = 0.88$	92.96	91.94	0.008
$b_t = 80, b_d = 8, m = 0.88$	92.89	91.87	0.006
$b_t = 80, b_d = 4, m = 0.88$	92.76	91.76	0.004
$b_t = 80, b_d = 2, m = 0.88$	92.56	91.54	0.003
$b_t = 80, b_d = 1, m = 0.88$	92.26	91.25	0.002
$b_t = 1, b_d = 1, m = 0.88$	92.06	91.05	0.002

Table 4: Parsing accuracies and speeds on the English evaluation set, excluding tokens containing only punctuation. b_t and b_d indicate the beam sizes used during training and decoding, respectively. UAS: unlabeled attachment score, LAS: labeled attachment score, Time: seconds per sentence.

For evaluation, we use the model trained with $b = 80$ and $m = 0.88$, which is the best setting found during development. Our parser shows higher accuracy than Zhang and Nivre (2011), which is the current state-of-the-art transition-based parser that uses beam search. Bohnet and Nivre (2012)'s transition-based system jointly performs POS tagging and dependency parsing, which shows higher accuracy than ours. Our parser gives a comparative accuracy to Koo and Collins (2010) that is a 3rd-order graph-based parsing approach. In terms of speed, our parser outperforms all other transition-based parsers; it takes about 9 milliseconds per

[7] Dhillon et al. (2012) and Rush and Petrov (2012) also have shown good results on this data but they are excluded from our comparison because they use different kinds of constituent-to-dependency conversion methods.

Approach	Danish		Dutch		Slovene		Swedish	
	LAS	UAS	LAS	UAS	LAS	UAS	LAS	UAS
Nivre et al. (2006)	84.77	89.80	78.59	81.35	70.30	78.72	84.58	89.50
McDonald et al. (2006)	84.79	90.58	79.19	83.57	73.44	83.17	82.55	88.93
Nivre (2009)	84.2	-	-	-	75.2	-	-	-
F.-González and G.-Rodríguez (2012)	85.17	90.10	-	-	-	-	83.55	89.30
Nivre and McDonald (2008)	86.67	-	81.63	-		75.94	84.66	
Martins et al. (2010)	-	**91.50**	-	84.91	-	**85.53**	-	89.80
$b_t = 80, b_d = 1, m = 0.88$	86.75	91.04	80.75	83.59	75.66	83.29	86.32	91.12
$b_t = 80, b_d = 80, m = 0.88$	**87.27**	91.36	**82.45**	**85.33**	**77.46**	84.65	**86.80**	**91.36**

Table 5: Parsing accuracies on four languages with non-projective dependencies, excluding punctuation.

sentence using the beam size of 80. Our parser is implemented in Java and tested on an Intel Xeon 2.57GHz. Note that we do not include input/output time for our speed comparison.

For a proof of concept, we run the same model, trained with $b_t = 80$, but decode with different beam sizes using the same margin. Surprisingly, our parser gives the same accuracy (0.01% higher for labeled attachment score) on this data even with $b_d = 16$. More importantly, $b_d = 16$ shows about the same parsing speed as $b_d = 80$, which indicates that selectional branching automatically reduced down the beam size by estimating low confidence predictions, so even if we assigned a larger beam size for decoding, it would have performed as efficiently. This implies that we no longer need to be so conscious about the beam size during decoding.

Another interesting part is that $(b_t = 80, b_d = 1)$ shows higher accuracy than $(b_t = 1, b_d = 1)$; this implies that our training method of bootstrapping transition sequences can improve even a greedy parser. Notice that our greedy parser shows higher accuracy than many other greedy parsers (Hall et al., 2006; Goldberg and Elhadad, 2010) because it uses the non-local features of Zhang and Nivre (2011) and the bootstrapping technique of Choi and Palmer (2011) that had not been used for most other greedy parsing approaches.

4.5 Non-projective parsing experiments

Table 5 shows comparison between state-of-the-art parsers and our approach for four languages with non-projective dependencies. Nivre et al. (2006) uses a pseudo-projective transition-based parsing approach. McDonald et al. (2006) uses a 2nd-order maximum spanning tree approach. Nivre (2009) and Fernández-González and Gómez-Rodríguez (2012) use different non-projective transition-based parsing approaches. Nivre and McDonald (2008) uses an ensemble model between transition-based and graph-based parsing approaches. Martins et al. (2010) uses integer linear programming for the optimization of their parsing model.

Some of these approaches use greedy parsers, so we include our results from models using ($b_t = 80$, $b_d = 1$, $m = 0.88$), which finds only the one-best sequences during decoding although it is trained on multiple transition sequences (see Section 4.4). Our parser shows higher accuracies for most languages except for unlabeled attachment scores in Danish and Slovene. Our greedy approach outperforms both Nivre (2009) and Fernández-González and Gómez-Rodríguez (2012) who use different non-projective parsing algorithms.

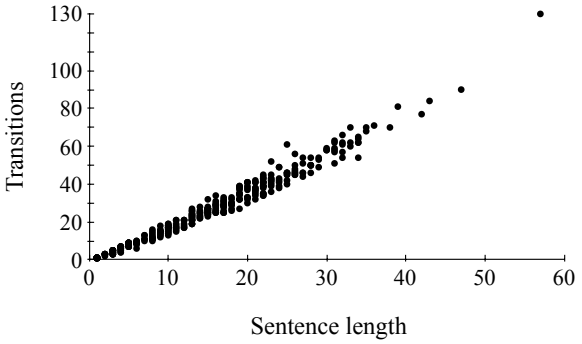


Figure 6: The # of transitions performed during decoding with respect to sentence lengths for Dutch.

Figure 6 shows the number of transitions performed during decoding with respect to sentence lengths for Dutch using $b_d = 1$. Our parser still shows a linear growth in transition during decoding.

5 Related work

Our parsing algorithm is most similar to Choi and Palmer (2011) who integrated our LEFT-REDUCE transition into Nivre's list-based algorithm. Our algorithm is distinguished from theirs because ours gives different parsing complexities of $O(n)$ and $O(n^2)$ for projective and non-projective parsing, respectively, whereas their algorithm gives $O(n^2)$

for both cases; this is possible because of the new integration of the RIGHT-SHIFT and NO-REDUCE transitions. There are other transition-based dependency parsing algorithms that take a similar approach; Nivre (2009) integrated a SWAP transition into Nivre's arc-standard algorithm (Nivre, 2004) and Fernández-González and Gómez-Rodríguez (2012) integrated a buffer transition into Nivre's arc-eager algorithm to handle non-projectivity.

Our selectional branching method is most relevant to Zhang and Clark (2008) who introduced a transition-based dependency parsing model that uses beam search. Huang and Sagae (2010) later applied dynamic programming to this approach and showed improved efficiency. Zhang and Nivre (2011) added non-local features to this approach and showed improved parsing accuracy. Bohnet and Nivre (2012) introduced a transition-based system that jointly performed POS tagging and dependency parsing. Our work is distinguished from theirs because we use selectional branching instead.

6 Conclusion

We present selectional branching that uses confidence estimates to decide when to employ a beam. Coupled with our new hybrid parsing algorithm, ADAGRAD, rich non-local features, and bootstrapping, our parser gives higher parsing accuracy than most other transition-based dependency parsers in multiple languages and shows faster parsing speed. It is interesting to see that our greedy parser outperformed most other greedy dependency parsers. This is because our parser used both bootstrapping and Zhang and Nivre (2011)'s non-local features, which had not been used by other greedy parsers.

In the future, we will experiment with more advanced dependency representations (de Marneffe and Manning, 2008; Choi and Palmer, 2012b) to show robustness of our approach. Furthermore, we will evaluate individual methods of our approach separately to show impact of each method on parsing performance. We also plan to implement the typical beam search approach to make a direct comparison to our selectional branching.[8]

Acknowledgments

Special thanks are due to Luke Vilnis of the University of Massachusetts Amherst for insights on the ADAGRAD derivation. We gratefully acknowledge a grant from the Defense Advanced Research Projects Agency (DARPA) under the DEFT project, solicitation #: DARPA-BAA-12-47.

[8]Our parser is publicly available under an open source project, ClearNLP (`clearnlp.googlecode.com`).

References

Bernd Bohnet and Joakim Nivre. 2012. A Transition-Based System for Joint Part-of-Speech Tagging and Labeled Non-Projective Dependency Parsing. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP'12, pages 1455–1465.

Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, CoNLL'06, pages 149–164.

Xavier Carreras, Michael Collins, and Terry Koo. 2008. TAG, Dynamic Programming, and the Perceptron for Efficient, Feature-rich Parsing. In *Proceedings of the 12th Conference on Computational Natural Language Learning*, CoNLL'08, pages 9–16.

Daniel Cer, Marie-Catherine de Marneffe, Daniel Jurafsky, and Christopher D. Manning. 2010. Parsing to Stanford Dependencies: Trade-offs between speed and accuracy. In *Proceedings of the 7th International Conference on Language Resources and Evaluation*, LREC'10.

Jinho D. Choi and Martha Palmer. 2011. Getting the Most out of Transition-based Dependency Parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, ACL:HLT'11, pages 687–692.

Jinho D. Choi and Martha Palmer. 2012a. Fast and Robust Part-of-Speech Tagging Using Dynamic Model Selection. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics*, ACL'12, pages 363–367.

Jinho D. Choi and Martha Palmer. 2012b. Guidelines for the Clear Style Constituent to Dependency Conversion. Technical Report 01-12, University of Colorado Boulder.

Michael Collins. 2002. Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. In *Proceedings of the conference on Empirical methods in natural language processing*, EMNLP'02, pages 1–8.

Marie-Catherine de Marneffe and Christopher D. Manning. 2008. The Stanford typed dependencies representation. In *Proceedings of the COLING workshop on Cross-Framework and Cross-Domain Parser Evaluation*.

Paramveer S. Dhillon, Jordan Rodu, Michael Collins, Dean P. Foster, and Lyle H. Ungar. 2012. Spectral Dependency Parsing with Latent Variables. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP'12, pages 205–213.

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *The Journal of Machine Learning Research*, 12(39):2121–2159.

Daniel Fernández-González and Carlos Gómez-Rodríguez. 2012. Improving Transition-Based Dependency Parsing with Buffer Transitions. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP'12, pages 308–319.

Yoav Goldberg and Michael Elhadad. 2010. An Efficient Algorithm for Easy-First Non-Directional Dependency Parsing. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT:NAACL'10, pages 742–750.

Yoav Goldberg and Joakim Nivre. 2012. A Dynamic Oracle for Arc-Eager Dependency Parsing. In *Proceedings of the 24th International Conference on Computational Linguistics*, COLING'12.

Johan Hall, Joakim Nivre, and Jens Nilsson. 2006. Discriminative Classifiers for Deterministic Dependency Parsing. In *In Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, COLING-ACL'06, pages 316–323.

Cho-Jui Hsieh, Kai-Wei Chang, Chih-Jen Lin, S. Sathiya Keerthi, and S. Sundararajan. 2008. A Dual Coordinate Descent Method for Large-scale Linear SVM. In *Proceedings of the 25th international conference on Machine learning*, ICML'08, pages 408–415.

Liang Huang and Kenji Sagae. 2010. Dynamic Programming for Linear-Time Incremental Parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL'10.

Liang Huang, Suphan Fayong, and Yang Guo. 2012. Structured Perceptron with Inexact Search. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, NAACL-HLT'12, pages 142–151.

Terry Koo and Michael Collins. 2010. Efficient Third-order Dependency Parsers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL'10.

Terry Koo, Xavier Carreras, and Michael Collins. 2008. Simple Semi-supervised Dependency Parsing. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, ACL:HLT'08, pages 595–603.

Sandra Kübler, Ryan T. McDonald, and Joakim Nivre. 2009. *Dependency Parsing*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

André F. T. Martins, Noah A. Smith, Eric P. Xing, Pedro M. Q. Aguiar, and Mário A. T. Figueiredo. 2010. Turbo Parsers: Dependency Parsing by Approximate Variational Inference. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, EMNLP'10, pages 34–44.

Ryan Mcdonald and Fernando Pereira. 2006. Online Learning of Approximate Dependency Parsing Algorithms. In *Proceedings of the Annual Meeting of the European American Chapter of the Association for Computational Linguistics*, EACL'06, pages 81–88.

Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005. Online Large-Margin Training of Dependency Parsers. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 91–98.

Ryan McDonald, Kevin Lerman, and Fernando Pereira. 2006. Multilingual Dependency Analysis with a Two-Stage Discriminative Parser. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, CoNLL'06, pages 216–220.

Joakim Nivre and Ryan McDonald. 2008. Integrating Graph-based and Transition-based Dependency Parsers. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, ACL:HLT'08, pages 950–958.

Joakim Nivre and Jens Nilsson. 2005. Pseudo-Projective Dependency Parsing. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, ACL'05, pages 99–106.

Joakim Nivre, Johan Hall, Jens Nilsson, Gülşen Eryiğit, and Svetoslav Marinov. 2006. Labeled pseudo-projective dependency parsing with support vector machines. In *Proceedings of the 10th Conference on Computational Natural Language Learning*, CoNLL'06, pages 221–225.

Joakim Nivre. 2003. An Efficient Algorithm for Projective Dependency Parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies*, IWPT'03, pages 149–160.

Joakim Nivre. 2004. Incrementality in Deterministic Dependency Parsing. In *Proceedings of the ACL'04 Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57.

Joakim Nivre. 2006. *Inductive Dependency Parsing*. Springer.

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553.

Joakim Nivre. 2009. Non-Projective Dependency Parsing in Expected Linear Time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, ACL-IJCNLP'09, pages 351–359.

Alexander M. Rush and Slav Petrov. 2012. Vine Pruning for Efficient Multi-Pass Dependency Parsing. In *Proceedings of the 12th Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, NAACL:HLT'12.

Alexander M. Rush, David Sontag, Michael Collins, and Tommi Jaakkola. 2010. On Dual Decomposition and Linear Programming Relaxations for Natural Language Processing. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, EMNLP'10, pages 1–11.

Kenji Sagae and Alon Lavie. 2006. Parser Combination by Reparsing. In *In Proceedings of the Human Language Technology Conference of the NAACL*, NAACL'06, pages 129–132.

Jun Suzuki, Hideki Isozaki, Xavier Carreras, and Michael Collins. 2009. An Empirical Study of Semi-supervised Structured Conditional Models for Dependency Parsing. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, EMNLP'09, pages 551–560.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machine. In *Proceedings of the 8th International Workshop on Parsing Technologies*, IWPT'03, pages 195–206.

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 *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP'08, pages 562–571.

Hao Zhang and Ryan McDonald. 2012. Generalized Higher-Order Dependency Parsing with Cube Pruning. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP-CoNLL'12, pages 320–331.

Yue Zhang and Joakim Nivre. 2011. Transition-based Dependency Parsing with Rich Non-local Features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, ACL'11, pages 188–193.