Comparing Advanced Graph-based and Transition-based Dependency Parsers

Bernd Bohnet
University of Stuttgart
Institut für Maschinelle Sprachverarbeitung
bohnet@ims.uni-stuttgart.de

Abstract

In this paper, we compare a higher order graph-based parser and a transition-based parser with beam search. These parsers provide a higher accuracy than a second order MST parser and a deterministic transition-based parser. We apply and compare the output on languages, which have not been in the research focus of Shared Tasks. The parser are implemented in a uniform framework. The transition-based parser was newley implemented and we revised the graph-based parser. The graph-based parser has to our knowlege the highest published scores for French and Czech with 90.40 and 81.43 labeled accuracy score.

1 Introduction

The two main approaches to data-driven dependency parsing are transition-based dependency parsing (Yamada and M., 2003; Nivre, 2003; Nivre et al., 2004; Titov and Henderson, 2007) and maximum spanning tree based dependency parsing (Eisner, 1996; Eisner, 2000; McDonald and Pereira, 2006; Carreras, 2007; Johansson and Nugues, 2008).

The transition-based approach, might not provide the highest score for instance for English. Nevertheless, it can be justified to improve one of the approaches on its own because for some languages such as Catalan and Spanish, it had higher scores in the CoNLL shared task 2009, cf. (Gesmundo et al., 2009). The transition-based approach has a lower complexity and it is easier to implement. For stacking experiments with both approaches, each has to be optimized separately towards speed and accuracy.

A statistical **transition-based parser** learns which actions to perform for building a dependency graph while scanning a sentence. The parser builds the dependency trees by going left-to-right (or right-to-left) through the words of a sentence. At each step, a classifier selects the appropriate parsing action for the current state based on a set of features. Transition-based parsers typically have a linear or quadratic complexity (Nivre et al., 2004; Attardi, 2006). Nivre (2009) introduces a transition-based non-projective parsing algorithm that has a worst case quadratic complexity and an expected linear parsing time. Titov and Henderson (2007) combine a transition-based parsing algorithm that uses a beam search with a latent variable machine learning technique. The latest update of this parser provided the best accuracy for transition-based dependency parsing in the CoNLL shared task 2009 (Gesmundo et al., 2009).

Graph-based dependency parsers start with a completely connected graph whose edges are weighted according to a statistical model. They then try to find the spanning tree that covers all nodes in the graph (the words) and at the same time maximizes the sum of the edges belonging to the spanning tree. The original non-projective formulation by McDonald et al. (2005) had a complexity of O(n^2) but was not capable of taking second-order features into account (making the choice for an edge depending on already chosen edges). Second order MST parsing was shown to significantly improve results compared to first-

order parsing (McDonald et al., 2006; Carreras, 2007) but at the cost of a higher complexity (McDonald and Satta, 2007). Carreras (2007) also fully integrated edge labels into the parsing procedure by adding an additional loop over the set of edge labels (L), thus raising performance as well as theoretical complexity (O(n^4L)). Johansson and Nugues (2008) reduced the number of loops over the edge labels by looking only at those edges that existed in the training corpus for a distinct head and child part-of-speech tag combination. Recently, Koo and Collins (2010) introduced an efficient third-order dependency parsing algorithm. The algorithm considers substructures containing three dependencies, and is called efficient because it requires only O(n^4) time. The parsing algorithm can utilize both features with sibling and grandchild information.

We apply a discriminative training method that employs a hash kernel to transition-based dependency parsing. Results show state-of-the-art unlabeled and labeled accuracy scores and fast parsing times. We illustrate that negative features can improve the accuracy of transition-based dependency parsers. Zhang and Clark (2008) as well as Gesmundo et al. (2009) applied a beam search to improve the accuracy of transition-based parsers.

2 Transition-based Parsing

We define a deterministic transition-based edge eager parser formally $T_e = \langle \sigma, \beta, \Omega, \epsilon, L, \Pi \rangle$ consisting of a list σ (stack), the list β (input), a set of operations $\Omega = \{$shift, left-arc, right-arc, reduce$\}$, a set of edges ϵ and a set of states Π. A state π_i=$\{\sigma_i, \beta_i, \epsilon_i\}$, $\pi_i \in \Pi$ consists of a list σ_i, an input buffer β_i and a set of edges ϵ_i.

(1) The initial state π_1 has an empty list σ_1, the input buffer β_1 contains the words of a sentence, and the edge set ϵ is empty. (2) A transition $\tau_i(\pi_i, \omega)$: Π x $\Omega \rightarrow \Pi$ is a binary function that maps a state and an operation to a new state π_{i+1}. We write a transition as $\pi_i \rightarrow_\omega \pi_{i+1}$. (3) The final state π_f has an empty input buffer β_f and therefore, no further operations are applicable.

The history of the (partial) parse h is a list of operations. We can define the operations and preconditions for the operations as follows:

The **shift** transition τ_s : $\pi_i \rightarrow_{shift} \pi_{i+1}$ removes the first element of the input buffer $w_n \in \beta_i$ where: $\beta_i = \{w_n, ...\}$ and adds the word to the end of the list $\sigma_{i+1} \leftarrow \sigma_i \cup \{w_n\}$. We obtain the next state with $\pi_{i+1} = \{\sigma_{i+1}, \beta_{i+1}, \epsilon_i\}$. **Precondition:** $\beta_i \neq \emptyset$

The **left-arc** transition τ_l : $\pi_i \rightarrow^{label}_{left-arc} \pi_{i+1}$ adds the last element [+ s_l] of the list σ_i and the first element [+ b_1] of β_i: $\epsilon_{i+1} \leftarrow \epsilon_i \cup \{(b_1, label, s_l)\}$; the element s_l is removed from σ_i: $\sigma_{i+1} \leftarrow \sigma_i - \{s_l\}$ and the first element of the input buffer is added as the last element to the list σ_i: $\sigma_{i+1} \leftarrow \sigma_i \cup \{b_1\}$: $\pi_{i+1} = \{\sigma_{i+1}, \beta_{i+1}, \epsilon_{i+1}\}$ **Precondition:** $\beta_i \neq \emptyset$ and $\sigma_i \neq \emptyset$ and not has-head(s_l)

The **right-arc** transition τ_r : $\pi_i \rightarrow^{label}_{right-arc} \pi_{i+1}$ adds an edge between the last element $s_e \in \sigma_i$ and the first element $b_o \in \beta_i$; $\sigma_{i+1} \leftarrow \sigma \cup \{b_0\}$ and $\beta_{i+1} \leftarrow \beta_i - \{b_0\}$ $\epsilon_{i+1} \leftarrow \epsilon_i \cup \{(s_l, label, b_1)\}$ **Precondition:** $\beta_i \neq \emptyset$, & $\sigma_i \neq \emptyset$ and has-head(s_r)

The **reduce** transition τ_r : $\pi_i \rightarrow_{reduce} \pi_{i+1}$ removes the last element of $s_l \in \sigma_i$: $\sigma_{i+1} \leftarrow \sigma - \{s_l\}$ **Precondition:** $\sigma_i \neq \emptyset$

Applying this definition, we define the transition-based dependency parser with beam search in Algorithm 1. We score a transition sequence h as the sum of the scores for the individual transitions $w_i \in h$:

score (π) = $\sum_{i=0}^{|h|} F(\pi_i, \omega_i)$

Note that a state π_i contains the stack σ_i, input buffer β_i, and set of edges ϵ_i. These elements are taken into account to create the feature set. The feature set is the input for the support vector machine; it provides the score due to the features.

Algorithm 1: Transition-based parser with beam search

```
// x_c is a input sentence
σ_0 = ∅, β_0 = x_c, ϵ_0 = ∅, h = ∅
π_0 ← {σ_0, β_0, ϵ_0, h_0} // initial parts of a state
beam_0 ← {π_0} // create initial state
n ← 0// iteration
repeat
  for all π_j ∈ beam_n do
    operations ← possible-applicable-operation (π_j)
    // if no operation is applicable keep state π_j
    if operations = ∅ then beam_n ← beam_n ∪ {π_j}
    else for all ω_i ∈ operations do
        π ← τ(π_j, ω_i)// apply the operation i to state j
        beam_n ← beam_n ∪ {π}
    // end for
  // end for
  // the score function is defined in the next section
  sort beam_n due to the score(π_j)
  beam_n ← sublist (beam_n, 0, beam-size)
  n ← n + 1
until beam_{n-1} = beam_n // stop when the beam is not changed
```

3 Hash Kernel

We use a linear support vector machine with a Hash Kernel as classifier for our dependency parser, cf. (Shi et al., 2009; Bohnet, 2010). The Hash Kernel uses instead of a table to map the features to the indexes in the weight vector a random function. Therefore, the Hash Kernel can quickly process large numbers of features and hence we can use "negative" features. In most parsing approaches features are collected prior to the training phase which are derived from the gold trees and in the training phase, the feature set is not extended further. However, the decoder creates wrong structures and the features derived from predicted trees are often not found since they do not occur in the gold trees. We counted 9 times more negative features than positive ones. A Hash Kernel for structured data uses the hash function $h : J \rightarrow \{1...n\}$ to index ϕ. ϕ maps π_i to a feature space. We define $\phi(\pi_i)$ as the numeric feature representation indexed by J. Let $\overline{\phi}_k(x, y) = \phi_j(x, y)$ the hash based feature–index mapping, where $h(j) = k$. The scoring function of the Hash Kernel is

$$F(\pi_i) = \overrightarrow{w} * \overline{\phi}(\sigma_i, \beta_i, \epsilon_i, \omega_i)$$

where $\overrightarrow{w}$ is the weight vector and the size of $\overrightarrow{w}$ is n.

Algorithm 2: Update of the Hash Kernel

```
//π_i is the state of a predicted state and
//π_g the gold state including the transition
//sequences h_g and h_i for the gold and predicted state
// update(w, v, π_i, π_g, γ)
   err= Δ(h_g, h_i) // number of wrong transitions
   if err > 0 then
      u ← (φ(π_i) − φ(π_g))
      ν = (err − (F(π_i) − F(π_g))) / ||u||^2
      w ← w − ν * u
      v ← v − γ * ν * u
return w, v
```

Algorithm 2 illustrates the update function of the Hash Kernel. The update function is similar to that of (Crammer et al., 2006). The parameters of the function are the weight vectors $\overrightarrow{w}$ and $\overrightarrow{v}$, the predicted state π_i, the gold state π_g, which should been have built by the parsing algorithm so far, and the update weight γ. The function Δ calculates the number of wrong transitions. The update function updates the weight vectors, when a transition is wrong. It calculates the difference $\overrightarrow{u}$ of the feature vectors of the gold dependency structure $\overline{\phi}(\pi_i)$ and the predicted transition $\overline{\phi}(\pi_g)$. The hash function f_h maps the features to integer numbers between 1 and $|\overrightarrow{w}|$. After that the update function calculates the margin ν and updates $\overrightarrow{w}$ and $\overrightarrow{v}$ respectively. The second weight vector is used for averaging in order to avoid overfitting and collects the weight of all training rounds with a passive-aggressive update.

4 Feature Selection

Transition-based dependency parsers are most frequently used with polynomial kernels of degree two since it is very convenient to specify features, cf. (Hall et al., 2006; Nivre et al., 2007; Nivre, 2009). SVMs of degree two use automatically derived combinations of at most two simple features. On the other hand, linear support vector machines provide faster training and classification times. Linear SVMs require a higher manual effort to select the features and combination of simple features because that involves many experiments where each time a parser has to be trained in order to find good combinations of simple features. Therefore, we had to perform a feature selection considering feature and combination. The feature templates are shown in Table 1.

5 Efficient Implementation

We want to emphasize similar to Goldberg and Elhadad (2010) that the parsing time is to a large degree determined by the feature extraction, the score calculation and the implementation.

We use a rich feature set and negative features. Nevertheless the parser is still fast with 47 sentences per second. This is because of the efficiency of the Hash Kernel, which is about four times faster than our implementation of the perceptron algorithm. With our baseline perceptron algorithm, we use about about 6 million features. The hash kernel uses about 50 million features including negative features. Our algorithms provides labeled trees, which distinguishes it from (Zhang and Clark, 2008) and (Goldberg and Elhadad, 2010). Some further optimizations are: (1) For the implementation of the beam, we store and reuse the calculated scores. We use a two step approach. We extract and store the values of the features that do not contain structural elements or elements of the stack σ except the last elements. This part of the weight is only calculated once and stored. Goldberg and Elhadad (2010) use a similar technique. We have to calculate the second part

Standard Features	
L,t,x,y :x$\in$ {sF,sP}&$y \in$ {bF,bP}	L,t,sP,bP,x : x$\in$ {s-1P,s+1P,s+2P,s-2P}
L,t,x,y,z :x$\in$ {s-1P,s+1P} & y$\in$ {b-1P,b+1P,b-2P,b+2P} & z$\in$ {sP,bP}	
L,t,x,y,z :x$\in$ {s-1F,s+1F} & y$\in$ {b-1F,b+1F,b-2F,b+2F} & z$\in$ {sP,bP}	
L,t,x,y,z :x$\in$ {s-1F,s+1F} & y$\in$ {b-1P,b+1P,b-2P,b+2P} & z$\in$ {sP,bP}	
L,t,s-1F,s-2F,bP ; L,t,s-2F,s-3F,bP ; L,t,s+1F,s+2F,bP	L,t,s+2F,s+3F,bP
L,t,b-1F,b-2F,sP ; L,t,b-2F,b-3F,sP ; L,t,b+1F,b+2F,sP	L,t,b+1F,b+2F,sP
L,t,s-1P,s-2P,bP ; L,t,s-2P,s-3P,bP ; L,t,s+1P,s+2P,bP	L,t,s+2P,s+3P,bP
L,t,b-1P,b-2P,sP ; L,t,b-2P,b-3P,sP ; L,t,b+1P,b+2P,sP	L,t,b+2P,b+3P,sP
L,t,sP,x,y : $x \in$ {s+1F,s+2F,s+3F} & $y \in$ {b+1P,b+2P,b+3P}	
L,t,sP,bP,x,y : $x \in$ {s-1P,s+1P} & $y \in$ {b-1P,b+1P}	
Structural Features	
L,t,x,bP,h(s) : $x \in$ {s+1F,s+1F}	L,t,x,sP,h(s) : $x \in$ { b+1F,b+1P}
L,t,sP,bP,x : $x \in$ {leftsib(s)L,head(s_1)L,leftsib(s_1)L,head(s_2)L}	L,t,s_1P,s_2P
L,t,sP,bP,x : $x \in$ {leftsib(s)P,head(s_1)P,leftsib(s_1)P,head(s_2)P}	
L,t,bP,x,y: x$\in$ {b+1F,b+2F,b+3F} & y$\in$ {s+1P,s+2P,s+3P}	

Table 1: t represents a transition type, s the last word of σ, b the first word of the input β. P represents the part-of-speech-tag, F the form and L the label. -1,+1,+2, etc. denote the location one or two word before or after an element. h denotes the head and *leftsib* the the leftmost sibling and *rightsib* rightmost sibling. s_1, s_2, b_1, etc. are the last but one of σ, etc.

	wrong edges		total number	% of correct edges	
	G	T	of edges	G	T
PMOD	315	334	5593	94.36	93.84
VC	15	14	1771	99.15	99.2
SBAR	56	57	1195	95.31	95.23
SUB	145	161	4108	96.47	96.08
PRD	47	56	832	94.35	93.26
P	875	931	7301	88.01	87.24
AMOD	339	339	2072	83.63	83.63
OBJ	142	155	1960	92.75	92.09
ROOT	92	121	2416	96.19	94.99
NMOD	1206	1235	21002	94.25	94.11
VMOD	938	997	8175	88.52	87.8
DEP	95	122	259	63.32	52.89

Table 2: Labeled accuracy scores of the graph-based (G) and transition-based (T) parse for distinct edge types using the Penn2Malt conversion.

of the score each time anew since this depends on structural parts (e.g left-most sibling of s_l) and the elements of σ. The space complexity is O(n^2L) for the feature caching. (2) Furthermore, the calculation of each score is optimized: We calculate for each location determined by the last element $s_l \in \sigma_i$ and the first element of $b_0 \in \beta_i$ a numeric feature representation. This is kept fix and we add only the numeric value for each of the edge labels plus a value for the operation left-arc or right-arc. In this way, we create the features incrementally. (3) Further, we applied edge filtering as it is used in graph-based dependency parsing, cf. (Johansson and Nugues, 2008), i.e., we calculate the edge weights only for the labels that were found for the part-of-speech combination of the head and dependent in the training data.

6 Graph-based Parser

The basis for the graph-based parser is a higher order graph-based dependency parser developed by Bohnet (2010). We contribute two parts to this parser, which will become publicly available. A revision of the feature set and a new random function for the hash kernel that allows to create features incrementally. Based on the incremental features creation, we can provide a faster feature extraction. For the higher order dependency parser, the feature extraction iterates for all edges over all possible labels since the labels are part of the feature. Johansson and Nugues (2008) introduced the concept of edge filters. Edge filter constrain the possible edges labels to the labels, which occur in the training set for the part-of-speech tag combination of the head and its dependent.

Features are extracted for each possible edge label due to the edge constrains. However, the parser

System	Czech	French	English
Malt		87.32/89.73[2]	
MST		88.24/90.91[2]	
Merlo	80.38/-		
transition-based	77.75/84.58[1,2)]	88.12/90.93[2])	89.22/91.82[1,2)]
graph-based	81.43/88.01[1,2)]	90.4/92.81[2)]	90.48/92.58[1,2)]

Table 3: Labeled dependency scores / unlabeled dependency scores for top scoring transition-based dependency parsers. [1] including punctuation, [2] predicted POS-tags

does not need to build always the complete features for each of the edge labels. It can extract once the features for an edge and add later the part for the edge label. The same strategy again is possible with the parts of the features of a head and the set of dependents. The parser extracts once the properties of the head and iterates over the possible dependents and adds to the feature part of the head a part for each of the dependents. The same strategy is possible for the sibling and grandchildren features.

We could save 81% of the feature creation time and improve the speed of the parser by 25%. For instance, for French the parsing time went down from 0.079 seconds/sentence to 0.059 [1]

The features consists usually of several components. For instance, a standard second order features consists of the part-of-speech tag of a head, a dependent and a grandchild. These parts describe properties of a edge. There are in addition functional parts of a feature, which are the type of a feature and the edge label. The feature type is used to distinguish features for instance, a sibling features from a grandchild feature. Both types might have the same parts (equal number of part-of-speech tags) but they have of cause a different meaning.

The feature creation function composes parts to features. This can be done by different operations. A standard operator is the bit shift operator. For instance, a tag set might have 52 different tags. Therefore, 6 bits are needed to encode part-of-speech-tags. In order to encode several part-of-speech tags as a long value, we add the integer value of a part-of-speech tag and shift it by 6 bits. This procedure is repeated until all parts are encoded. This method wastes some of the encoding space since the 6 bit space could enumerate 64 values. Therefore, we use to encode the values the multiplication operator and multiply the value by the number of elements in the set, we want to encode.

The revised feature set combines systematically each part-of-speech tag, word form, lemma, distance features of the governor, dependent, sibling and grandchild. We used instead of a features for each words between the head and the lemmata, a single features that is a sorted bag of part-of-speech tags. The accuracy improved because of this for Czech and slightly for English as Table 3 shows.

7 Experiments

We trained the parser on English dependency trees as provided by the CoNLL shared task 2009 and on dependency trees converted with Penn2Malt using the head-finding rules of (Yamada and Matsumoto, 2003). Table 4 gives an overview of the data used with the these head-finding rules. The training data was 10-fold jackknifed with the tagger included in the Mate-Tools[2].

	Section	Sentences	PoS Acc.
Training	2-21	39.832	97.08
Dev	24	1.394	97.18
Test	23	2.416	97.30

Table 4: Overview of the training, development and test data split converted to dependency graphs with head-finding rules of (Yamada and Matsumoto, 2003). The last row shows the accuracy of Part-of-Speech tags.

We optimized our parser on section 24 and used section 23 of Penn Treebank for evaluation, which was the test set in the CoNLL shared task.

Table 6 summarizes the results and compares the result with Zhang and Clark (2008) as well as Goldberg and Elhadad (2010). We have taken the results for the Malt parser from Goldberg and Elhadad (2010).

[1] We used a computer with 12 cores, Intel Westmere and 3.33 Ghz.

[2] http://code.google.com/p/mate-tools/

System	LAS/UAS	Speed (sent./sec.)
Merlo	88.79/-	
Clear	89.15/91.18	430
this work	89.22/91.82	30

Table 5: **CoNLL Shared Task 2009 Data:** Labeled and unlabeled dependency scores of (Gesmundo et al., 2009) (Merlo), Choi and Palmer (2011) (Clear) and the parser introduced here. [1] including punctuation, [2] predicted POS-tags as provided in Shared Task.

System	UAS	Speed (sent./sec)
	including punctuation	
Malt	88.36	
NonDir	89.70	40
this work	91.81	47
	excluding punctuation	
Z&C08	91.4	50
Z&N11	92.90	29
this work	92.60	47

Table 6: **Penn2Malt, Train 2-21, Test 23**, predicted POS-tags: Unlabeled dependency scores of transition-based dependency parsers Zhang and Clark (2008) (Z&C08), Zhang and Nivre (2011) (Z&N11), Malt, NonDir (Goldberg and Elhadad, 2010).

In Table 5, we compare the scores of the our transition-based dependency parser with other transition-based parsers. The top score in the CoNLL Shared task 2009 was obtained by the parser of Gesmundo et al. (2009). This parser was ranked first in average for all languages and third for English, which was the best score of a transition-based parser for English. The labeled accuracy score of the dependency parser with Hash Kernel using the CoNLL data is about 0.4 percentage points higher than that of Gesmundo et al. (2009) and only slightly higher than the transition-based parser of Choi and Palmer (2011).

Table 6 shows results for the same data set but converted with Penn2Malt. The first three rows compare the result with other papers that included punctuation in their evaluation. The Malt and NonDir parser do not employ a beam search, which is probably the reason for the lower accuracy scores. The parser of Zhang and Clark (2008) is similar to our parser except that we use the Hash Kernel, which uses negative features in addition. The 2011 version (Zhang and Nivre, 2011) was published in the revision phase of this paper. Their parser uses a richer feature set and obtains 0.3 higher unlabeled accuracy scores. Remarkable is that our parser as well as the parser of Zhang get close to the results of the second order and third order graph-based dependency parser that carries out an exhaustive search and obtains 93.04 UAS on the test set (Koo and Collins, 2010). Our parser is fast with 47 sentences/second and a beam size of 80 on a MacBook Pro (2.8 Ghz). Gesmundo et al. (2009) uses a beam size of 80 as well and Zhang and Clark (2008) of 64. We use 25 training rounds.

System	English
this work (transition)	92.60/91.48
this work (graph)	93.06/91.96
Z&N11 (transition)	92.9/91.8
KC10	93.04
CCK08	93.50
SICC09	93.79

Table 7: Results obtained by graph-based dependency parser compared with selected transition-based parsers: Z&N11 (Zhang and Nivre, 2011), SICC09 (Suzuki et al., 2009), KC10 (Koo and Collins, 2010), and KCC08 (Koo et al., 2008)

In Table 7, we compare results of transition-based and graph-based parsers. The upper part of the table shows results obtained by parsing systems that do not exploit additional resources. Our updated second order graph-based parser obtain competitive results with 93.06 UAS. Table 2 shows a more detailed analysis on the level of edge labels. Both parsers are similar good on majority of the dependency edges. The transition-based parser has still a bit lower accuracy for the attachment of the root node (ROOT), punctuation marks, and verb modifiers (VMOD). Reviewing the errors in dependence to the distance, we could only observe a very slight tendency that long distance relations are more worse in the case of transition-based parsers.[3] An advantage of the transition-based parser is that it can observe some third order features, which the parser has already build, and also some subcategorization features.

Table 3 shows results of the graph-based and transition-based parser for Czech and English on the data of the CoNLL shared task 2009. For French, we use the data of Candito et al. (2010) as well as the same training, development and test data split. We obtain in line to English higher scores for the graph-based parser but the

[3]The graph-based parser has only 15% error rate on dependency spanning over more than 7 words in contrast to transition-based parser that has a error rate of 16.8%.

difference between the graph-based parser and transition-based parser for instance for Chzech is still much higher. We think that the reason for this are the higher portion of non-projective edges.

8 Conclusion and Future Work

We have presented a fast transition-based dependency parser with competitive labeled and unlabeled scores. We have shown that a transition-based parser can benefit from a support vector machine with Hash Kernel that enables the use of negative features, which improve the accuracy.

Our transition-based and graph-based parser performance quite different on the two English data sets. The graph-based parser has a higher accuracy than the transition-based parser with about 1.2 percentage point for English and 3.7 for Czech on the data of the CoNLL Shared Task 2009. The difference between the conversion of the CoNLL and conversion obtained with the Yamada and Matsumoto (2003) head finding rules is high. We observed a difference of 1.2/0.7 LAS/UAS on the CoNLL data and only 0.4/0.48 LAS/UAS with the Yamada and Matsumoto (2003) rules. The cause of this is is probably the larger number of edge labels and the non-projective edges contained in the CoNLL data.

References

G. Attardi. 2006. Experiments with a Multilanguage Non-Projective Dependency Parser. In *Proceedings of CoNLL*, pages 166–170.

B. Bohnet. 2010. Top accuracy and fast dependency parsing is not a contradiction. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 89–97, Beijing, China, August. Coling 2010 Organizing Committee.

M. Candito, B. Crabb, and P. Denis. 2010. Statistical french dependency parsing: Treebank conversion and first results. In Nicoletta Calzolari (Conference Chair), Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, Mike Rosner, and Daniel Tapias, editors, *Proceedings of the Seventh conference on International Language Resources and Evaluation (LREC'10)*, Valletta, Malta, may. European Language Resources Association (ELRA).

X. Carreras. 2007. Experiments with a Higher-order Projective Dependency Parser. In *EMNLP/CoNLL*.

J. D. Choi and Martha Palmer. 2011. Getting the most out of transition-based dependency parsing. In *ACL (Short Papers)*, pages 687–692.

K. Crammer, O. Dekel, S. Shalev-Shwartz, and Y. Singer. 2006. Online Passive-Aggressive Algorithms. *Journal of Machine Learning Research*, 7:551–585.

J. Eisner. 1996. Three New Probabilistic Models for Dependency Parsing: An Exploration. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING-96)*, pages 340–345, Copenhaen.

J. Eisner, 2000. *Bilexical Grammars and their Cubic-time Parsing Algorithms*, pages 29–62. Kluwer Academic Publishers.

A. Gesmundo, J. Henderson, P. Merlo, and I. Titov. 2009. A Latent Variable Model of Synchronous Syntactic-Semantic Parsing for Multiple Languages. In *Proceedings of the 13th Conference on Computational Natural Language Learning (CoNLL-2009)*, Boulder, Colorado, USA., June 4-5.

Y. Goldberg and M. Elhadad. 2010. An efficient algorithm for easy-first non-directional dependency parsing. In *HLT-NAACL*, pages 742–750.

J. Hall, J. Nivre, and J. Nilsson. 2006. Discriminative classifiers for deterministic dependency parsing. In *Proceedings of the COLING/ACL 2006 Main Conference Poster Sessions*, pages 316–323, Sydney, Australia, July. Association for Computational Linguistics.

R. Johansson and P. Nugues. 2008. Dependency-based Syntactic–Semantic Analysis with PropBank and NomBank. In *Proceedings of the Shared Task Session of CoNLL-2008*, Manchester, UK.

T. Koo and M. Collins. 2010. Efficient third-order dependency parsers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1–11, Uppsala, Sweden, July. Association for Computational Linguistics.

T. Koo, X. Carreras, and M. Collins. 2008. Simple semi-supervised dependency parsing. In *ACL*, pages 595–603.

R. McDonald and F. Pereira. 2006. Online Learning of Approximate Dependency Parsing Algorithms. In *In Proc. of EACL*, pages 81–88.

R. McDonald and G. Satta. 2007. On the Complexity of Non-projective Data-driven Dependency Parsing. In *IWPT '07: Proceedings of the 10th International Conference on Parsing Technologies*, pages 121–132, Morristown, NJ, USA.

R. McDonald, K. Crammer, and F. Pereira. 2005. Online Large-margin Training of Dependency Parsers. In *Proc. ACL*, pages 91–98.

R. McDonald, K. Lerman, K. Crammer, and F. Pereira. 2006. Multilingual Dependency Parsing with a Two-Stage Discriminative Parser. In *Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, pages 91–98.

J. Nivre, J. Hall, and J. Nilsson. 2004. Memory-Based Dependency Parsing. In *Proceedings of the 8th CoNLL*, pages 49–56, Boston, Massachusetts.

J. Nivre, J. Hall, S. Kübler, R. McDonald, J. Nilsson, S. Riedel, and D. Yuret. 2007. The conll 2007 shared task on dependency parsing. In *Proc. of the CoNLL 2007 Shared Task. Joint Conf. on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, June.

J. Nivre. 2003. An Efficient Algorithm for Projective Dependency Parsing. In *8th International Workshop on Parsing Technologies*, pages 149–160, Nancy, France.

J. Nivre. 2009. Non-Projective Dependency Parsing in Expected Linear Time. In *Proceedings of the 47th Annual Meeting of the ACL and the 4th IJCNLP of the AFNLP*, pages 351–359, Suntec, Singapore.

Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, and S.V.N. Vishwanathan. 2009. Hash Kernels for Structured Data. In *Journal of Machine Learning*.

J. Suzuki, H. Isozaki, X. Carreras, and M Collins. 2009. An empirical study of semi-supervised structured conditional models for dependency parsing. In *EMNLP*, pages 551–560.

I. Titov and J. Henderson. 2007. A Latent Variable Model for Generative Dependency Parsing. In *Proceedings of IWPT*, pages 144–155.

H. Yamada and Yuji M. 2003. Statistical dependency analysis with support vector machines. In *In Proceedings of IWPT*, pages 195–206.

H. Yamada and Y. Matsumoto. 2003. Statistical Dependency Analysis with Support Vector Machines. In *Proceedings of IWPT*, pages 195–206.

Y. Zhang and S. Clark. 2008. A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In *Proceedings of EMNLP*, Hawaii, USA.

Y. Zhang and J. 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*, pages 188–193, Portland, Oregon, USA, June. Association for Computational Linguistics.