A Tale of Two Parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search

Yue Zhang and Stephen Clark
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford OX1 3QD, UK
{yue.zhang,stephen.clark}@comlab.ox.ac.uk

Abstract

Graph-based and transition-based approaches to dependency parsing adopt very different views of the problem, each view having its own strengths and limitations. We study both approaches under the framework of beam-search. By developing a graph-based and a transition-based dependency parser, we show that a beam-search decoder is a competitive choice for both methods. More importantly, we propose a beam-search-based parser that combines both graph-based and transition-based parsing into a single system for training and decoding, showing that it outperforms both the pure graph-based and the pure transition-based parsers. Testing on the English and Chinese Penn Treebank data, the combined system gave state-of-the-art accuracies of 92.1% and 86.2%, respectively.

1 Introduction

Graph-based (McDonald et al., 2005; McDonald and Pereira, 2006; Carreras et al., 2006) and transition-based (Yamada and Matsumoto, 2003; Nivre et al., 2006) parsing algorithms offer two different approaches to data-driven dependency parsing. Given an input sentence, a graph-based algorithm finds the highest scoring parse tree from all possible outputs, scoring each complete tree, while a transition-based algorithm builds a parse by a sequence of actions, scoring each action individually.

The terms “graph-based” and “transition-based” were used by McDonald and Nivre (2007) to describe the difference between MSTParser (McDonald and Pereira, 2006), which is a graph-based parser with an exhaustive search decoder, and MaltParser (Nivre et al., 2006), which is a transition-based parser with a greedy search decoder. In this paper, we do not differentiate graph-based and transition-based parsers by their search algorithms: a graph-based parser can use an approximate decoder while a transition-based parser is not necessarily deterministic. To make the concepts clear, we classify the two types of parser by the following two criteria:

1. whether or not the outputs are built by explicit transition-actions, such as “Shift” and “Reduce”; 
2. whether it is dependency graphs or transition-actions that the parsing model assigns scores to.

By this classification, beam-search can be applied to both graph-based and transition-based parsers.

Representative of each method, MSTParser and MaltParser gave comparable accuracies in the CoNLL-X shared task (Buchholz and Marsi, 2006). However, they make different types of errors, which can be seen as a reflection of their theoretical differences (McDonald and Nivre, 2007). MSTParser has the strength of exact inference, but its choice of features is constrained by the requirement of efficient dynamic programming. MaltParser is deterministic, yet its comparatively larger feature range is an advantage. By comparing the two, three interesting research questions arise: (1) how to increase the flexibility in defining features for graph-based parsing; (2) how to add search to transition-based parsing; and (3) how to combine the two parsing approaches so that the strengths of each are utilized.

In this paper, we study these questions under one framework: beam-search. Beam-search has been successful in many NLP tasks (Koehn et al., 2003;
Inputs: training examples \((x_i, y_i)\)
Initialization: set \(\vec{w} = 0\)

Algorithm:
// R training iterations; N examples
for \(t = 1..R\), \(i = 1..N\):
\[ z_i = \arg \max_{y \in \text{GEN}(x_i)} \Phi(y) \cdot \vec{w} \]
if \(z_i \neq y_i:\)
\[ \vec{w} = \vec{w} + \Phi(y_i) - \Phi(z_i) \]

Output: \(\vec{w}\)

Figure 1: The perceptron learning algorithm

Collins and Roark, 2004), and can achieve accuracy that is close to exact inference. Moreover, a beam-search decoder does not impose restrictions on the search problem in the way that an exact inference decoder typically does, such as requiring the “optimal subproblem” property for dynamic programming, and therefore enables a comparatively wider range of features for a statistical system.

We develop three parsers. Firstly, using the same features as MSTParser, we develop a graph-based parser to examine the accuracy loss from beam-search compared to exact-search, and the accuracy gain from extra features that are hard to encode for exact inference. Our conclusion is that beam-search is a competitive choice for graph-based parsing. Secondly, using the transition actions from MaltParser, we build a transition-based parser and show that search has a positive effect on its accuracy compared to deterministic parsing. Finally, we show that by using a beam-search decoder, we are able to combine graph-based and transition-based parsing into a single system, with the combined system significantly outperforming each individual system. In experiments with the English and Chinese Penn Treebank data, the combined parser gave 92.1% and 86.2% accuracy, respectively, which are comparable to the best parsing results for these data sets, while the Chinese accuracy outperforms the previous best reported by 1.8%. In line with previous work on dependency parsing using the Penn Treebank, we focus on projective dependency parsing.

2 The graph-based parser

Following MSTParser (McDonald et al., 2005; McDonald and Pereira, 2006), we define the graph-based parser to projective dependency parsing using the Penn Treebank, we focus on projective dependency parsing.

Variables: \(\text{agenda} \rightarrow \) the beam for state items
\(\text{item} \rightarrow \) partial parse tree
\(\text{output} \rightarrow \) a set of output items
\(\text{index}, \text{prev} \rightarrow \) word indexes

Input: \(x \rightarrow \) POS-tagged input sentence.
Initialization: \(\text{agenda} = ["""]\)

Algorithm:
for \(\text{index} \in 1..x\text{.length}\):
\(\text{clear output}\)
for \(\text{item} \in \text{agenda}\):
\(\text{// for all prev words that can be linked with}\)
\(\text{// the current word at index}\)
\(\text{prev} = \text{index} - 1\)
while \(\text{prev} \neq 0\): // while prev is valid
\(\text{// add link making prev parent of index}\)
\(\text{newitem} = \text{item} \) // duplicate item
\(\text{newitem}\text{.link(prev, index)} \) // modify
\(\text{output}\text{.append(newitem)} \) // record
\(\text{// if prev does not have a parent word,}\)
\(\text{// add link making index parent of prev}\)
if \(\text{item}\text{.parent(prev)} == 0\):
\(\text{item}\text{.link(index, prev)} \) // modify
\(\text{output}\text{.append(item)} \) // record
\(\text{prev} = \) the index of the first word before
\(\text{prev}\) whose parent does not exist
or is on its left; 0 if no match
\(\text{clear agenda}\)
put the best items from \(\text{output}\) to \(\text{agenda}\)

Output: the best item in \(\text{agenda}\)

Figure 2: A beam-search decoder for graph-based parsing, developed from the deterministic Covington algorithm for projective parsing (Covington, 2001).

Based parsing problem as finding the highest scoring tree \(y\) from all possible outputs given an input \(x\):

\[
F(x) = \arg \max_{y \in \text{GEN}(x)} \text{Score}(y)
\]

where \(\text{GEN}(x)\) denotes the set of possible parses for the input \(x\). To repeat our earlier comments, in this paper we do not consider the method of finding the \(\arg \max\) to be part of the definition of graph-based parsing, only the fact that the dependency graph itself is being scored, and factored into scores attached to the dependency links.

The score of an output parse \(y\) is given by a linear model:

\[
\text{Score}(y) = \Phi(y) \cdot \vec{w}
\]
where $\Phi(y)$ is the global feature vector from $y$ and $\vec{w}$ is the weight vector of the model.

We use the discriminative perceptron learning algorithm (Collins, 2002; McDonald et al., 2005) to train the values of $\vec{w}$. The algorithm is shown in Figure 1. Averaging parameters is a way to reduce overfitting for perceptron training (Collins, 2002), and is applied to all our experiments.

While the MSTParser uses exact-inference (Eisner, 1996), we apply beam-search to decoding. This is done by extending the deterministic Covington algorithm for projective dependency parsing (Covington, 2001). As shown in Figure 2, the decoder works incrementally, building a state item (i.e. partial parse tree) word by word. When each word is processed, links are added between the current word and its predecessors. Beam-search is applied by keeping the $B$ best items in the agenda at each processing stage, while partial candidates are compared by scores from the graph-based model, according to partial graph up to the current word.

Before decoding starts, the agenda contains an empty sentence. At each processing stage, existing partial candidates from the agenda are extended in all possible ways according to the Covington algorithm. The top $B$ newly generated candidates are then put to the agenda. After all input words are processed, the best candidate output from the agenda is taken as the final output.

The projectivity of the output dependency trees is guaranteed by the incremental Covington process. The time complexity of this algorithm is $O(n^2)$, where $n$ is the length of the input sentence.

During training, the “early update” strategy of Collins and Roark (2004) is used: when the correct state item falls out of the beam at any stage, parsing is stopped immediately, and the model is updated using the current best partial item. The intuition is to improve learning by avoiding irrelevant information: when all the items in the current agenda are incorrect, further parsing steps will be irrelevant because the correct partial output no longer exists in the candidate ranking.

Table 1 shows the feature templates from the MSTParser (McDonald and Pereira, 2006), which are defined in terms of the context of a word, its parent and its sibling. To give more templates, features from templates 1 – 5 are also conjoined with the link direction and distance, while features from template 6 are also conjoined with the direction and distance between the child and its sibling. Here “distance” refers to the difference between word indexes. We apply all these feature templates to the graph-based parser. In addition, we define two extra feature templates (Table 2) that capture information about grandchildren and arity (i.e. the number of children to the left or right). These features are not conjoined with information about direction and distance. They are difficult to include in an efficient dynamic programming decoder, but easy to include in a beam-search decoder.

|   | Feature Templates |
|---|-------------------|
| 1 | Parent word (P)   | $P_w; P_t; P_{wt}$ |
| 2 | Child word (C)    | $C_w; C_t; C_{wt}$ |
| 3 | P and C           | $P_{wt}C_{wt}; P_{wt}C_w; P_{wt}C_t; P_tC_w; P_tC_t; P_tC_{wt}$ |
| 4 | A tag Bt between P, C | $P_tB_tC_t$ |
| 5 | Neighbour words of P, C, left (PL/CL) and right (PR/CR) | $P_tPL_tC_tCL_t; P_tPL_tC_tCR_t; P_tPR_tC_tCL_t; P_tPR_tC_tCR_t; P_tPL_tC_tCL_t; P_tPL_tC_tCR_t; P_tPR_tC_tCL_t; P_tPR_tC_tCR_t; P_tC_tCL_t; P_tC_tCR_t; P_tPL_tC_t; P_tC_tSt; P_tC_tSw; P_tC_tSt$ |
| 6 | Sibling (S) of C   | $C_wS_t; C_tS_w$ |

Table 1: Feature templates from MSTParser

|   | Extra Feature Templates |
|---|-------------------------|
| 1 | leftmost (CLC) and rightmost (CRC) children of C | $PtCL_tCt; PtCRC_tCt$ |
| 2 | left (la) and right (ra) arity of P | $Ptla; Ptra; Pwtla; Pwtra$ |

Table 2: Additional feature templates for the graph-based parser
The next three words from the input (the leftmost child of \( ST \)), the parent (\( STP \)) of \( ST \), the leftmost (\( STLC \)) and rightmost child (\( STRC \)) of \( ST \), the current word (\( N0 \)), the next three words from the input (\( N1, N2, N3 \)) and the leftmost child of \( N0 \) (\( N0LC \)). Given the context \( s \), the next action \( T \) is decided as follows:

\[
T(s) = \arg \max_{T \in \text{ACTION}} \text{Score}(T, s)
\]

where \( \text{ACTION} = \{ \text{Shift}, \text{ArcRight}, \text{ArcLeft}, \text{Reduce} \} \).

One drawback of deterministic parsing is error propagation, since once an incorrect action is made, the output parse will be incorrect regardless of the subsequent actions. To reduce such error propagation, a parser can keep track of multiple candidate outputs and avoid making decisions too early. Suppose that the parser builds a set of candidates \( \text{GEN}(x) \) for the input \( x \), the best output \( F(x) \) can be decided by considering all actions:

\[
F(x) = \arg \max_{y \in \text{GEN}(x)} \sum_{T' \in \text{act}(y)} \text{Score}(T', s_{T'})
\]

Here \( T' \) represents one action in the sequence (\( \text{act}(y) \)) by which \( y \) is built, and \( s_{T'} \) represents the corresponding context when \( T' \) is taken.

Our transition-based algorithm keeps \( B \) different sequences of actions in the agenda, and chooses the one having the overall best score as the final parse. Pseudo code for the decoding algorithm is shown in Figure 4. Here each state item contains a partial parse tree as well as a stack configuration, and state items are built incrementally by transition actions. Initially the stack is empty, and the agenda contains an empty sentence. At each processing stage, one transition action is applied to existing state items as a step to build the final parse. Unlike the MaltParser, which makes a decision at each stage, our transition-based parser applies all possible actions to each existing state item in the agenda to generate new items; then from all the newly generated items, it takes the \( B \) with the highest overall score and puts them onto the agenda. In this way, some ambiguity is retained for future resolution.

Note that the number of transition actions needed to build different parse trees can vary. For example, the three-word sentence “A B C” can be parsed by the sequence of three actions “Shift ArcRight ArcRight” (B modifies A; C modifies B) or the sequence of four actions “Shift ArcLeft Shift ArcRight” (both A and C modifies B). To ensure that all final state items are built by the same number of transition actions, we require that the final state
Variables: agenda – the beam for state items
item – (partial tree, stack config)
output – a set of output items
index – iteration index
Input: x – POS-tagged input sentence.
Initialization: agenda = [(“”, [])]
Algorithm:
for index in 1 .. 2 × x.length() − 1:
    clear output
    for item in agenda:
        // when all input words have been read, the
        // parse tree has been built; only pop.
        if item.length() == x.length():
            if item.stacksize() > 1:
                item.Reduce()
                output.append(item)
            else:
                if item.lastaction() != Reduce:
                    newitem = item
                    newitem.Shift()
                    output.append(newitem)
                else:
                    newitem = item
                    newitem.Reduce()
                    output.append(newitem)
        else:
            if item.stacksize() > 0:
                newitem = item
                newitem.ArcRight()
                output.append(newitem)
            else:
                newitem = item
                newitem.ArcLeft()
                output.append(newitem)
    if (item.parent(item.stacktop()) == 0):
        newitem = item
        newitem.ArcLeft()
        output.append(newitem)
    else:
        newitem = item
        newitem.Reduce()
        output.append(newitem)
    clear agenda
    transfer the best items from output to agenda
Output: the best item in agenda

Figure 4: A beam-search decoding algorithm for transition-based parsing

Items must 1) have fully-built parse trees; and 2) have only one root word left on the stack. In this way, popping actions should be made even after a complete parse tree is built, if the stack still contains more than one word.

Now because each word excluding the root must be pushed to the stack once and popped off once during the parsing process, the number of actions needed to parse a sentence is always 2n − 1, where n is the length of the sentence. Therefore, the decoder has linear time complexity, given a fixed beam size. Because the same transition actions as the MaltParser are used to build each item, the productivity of the output dependency tree is ensured.

We use a linear model to score each transition action, given a context:

\[
Score(T, s) = \Phi(T, s) \cdot \vec{w}
\]

\(\Phi(T, s)\) is the feature vector extracted from the action \(T\) and the context \(s\), and \(\vec{w}\) is the weight vector. Features are extracted according to the templates shown in Table 3, which are based on the context in Figure 3. Note that our feature definitions are similar to those used by MaltParser, but rather than using a kernel function with simple features (e.g. STw,
N0t, but not STwt or STwN0w), we combine features manually.

As with the graph-based parser, we use the discriminative perceptron (Collins, 2002) to train the transition-based model (see Figure 5). It is worth noticing that, in contrast to MaltParser, which trains each action decision individually, our training algorithm globally optimizes all action decisions for a parse. Again, “early update” and averaging parameters are applied to the training process.

4 The combined parser

The graph-based and transition-based approaches adopt very different views of dependency parsing. McDonald and Nivre (2007) showed that the MST-Parser and MaltParser produce different errors. This observation suggests a combined approach: by using both graph-based information and transition-based information, parsing accuracy can be improved.

The beam-search framework we have developed facilitates such a combination. Our graph-based and transition-based parsers share many similarities. Both build a parse tree incrementally, keeping an agenda of comparable state items. Both rank state items by their current scores, and use the averaged perceptron with early update for training. The key differences are the scoring models and incremental parsing processes they use, which must be addressed when combining the parsers.

Firstly, we combine the graph-based and the transition-based score models simply by summation. In particular, the transition-based model can be written as:

\[
\text{Score}_T(y) = \sum_{T' \in \text{act}(y)} \Phi(T', s_{T'}) \cdot \vec{w}_T
\]

If we take \(\sum_{T' \in \text{act}(y)} \Phi(T', s_{T'})\) as the global feature vector \(\Phi_T(y)\), we have:

\[
\text{Score}_T(y) = \Phi_T(y) \cdot \vec{w}_T
\]

which has the same form as the graph-based model:

\[
\text{Score}_G(y) = \Phi_G(y) \cdot \vec{w}_G
\]

Concatenating the feature vectors \(\Phi_G(y)\) and \(\Phi_T(y)\) to give a global feature vector \(\Phi_C(y)\), and the weight vectors \(\vec{w}_G\) and \(\vec{w}_T\) to give a weight vector \(\vec{w}_C\), the combined model can be written as:

\[
\text{Score}_C(y) = \Phi_C(y) \cdot \vec{w}_C
\]

which is a linear model with exactly the same form as both sub-models, and can be trained with the perceptron algorithm in Figure 1. Because the global feature vectors from the sub models are concatenated, the feature set for the combined model is the union of the sub model feature sets.

Second, the transition-based decoder can be used for the combined system. Both the graph-based decoder in Figure 2 and the transition-based decoder in Figure 4 construct a parse tree incrementally. However, the graph-based decoder works on a per-word basis, adding links without using transition actions, and so is not appropriate for the combined model. The transition-based algorithm, on the other hand, uses state items which contain partial parse trees, and so provides all the information needed by the graph-based parser (i.e. dependency graphs), and hence the combined system.

In summary, we build the combined parser by using a global linear model, the union of feature templates and the decoder from the transition-based parser.

5 Experiments

We evaluate the parsers using the English and Chinese Penn Treebank corpora. The English data is prepared by following McDonald et al. (2005). Bracketed sentences from the Penn Treebank (PTB) 3 are split into training, development and test sets.

|          | Sections | Sentences | Words  |
|----------|----------|-----------|--------|
| Training | 2–21     | 39,832    | 950,028|
| Dev      | 22       | 1,700     | 40,117 |
| Test     | 23       | 2,416     | 56,684 |

Table 4: The training, development and test data from PTB

We therefore combine the two models to give:

\[
\text{Score}_C(y) = \text{Score}_G(y) + \text{Score}_T(y) = \Phi_G(y) \cdot \vec{w}_G + \Phi_T(y) \cdot \vec{w}_T
\]
Figure 6: The influence of beam size on the transition-based parser, using the development data
X-axis: number of training iterations
Y-axis: word precision

as shown in Table 4, and then translated into dependency structures using the head-finding rules from Yamada and Matsumoto (2003).

Before parsing, POS tags are assigned to the input sentence using our reimplementation of the POS-tagger from Collins (2002). Like McDonald et al. (2005), we evaluate the parsing accuracy by the precision of lexical heads (the percentage of input words, excluding punctuation, that have been assigned the correct parent) and by the percentage of complete matches, in which all words excluding punctuation have been assigned the correct parent.

5.1 Development experiments
Since the beam size affects all three parsers, we study its influence first; here we show the effect on the transition-based parser. Figure 6 shows different accuracy curves using the development data, each with a different beam size $B$. The X-axis represents the number of training iterations, and the Y-axis the precision of lexical heads.

The parsing accuracy generally increases as the beam size increases, while the quantity of increase becomes very small when $B$ becomes large enough. The decoding times after the first training iteration are 10.2s, 27.3s, 45.5s, 79.0s, 145.4s, 261.3s and 469.5s, respectively, when $B = 1, 2, 4, 8, 16, 32, 64$.

In the rest of the experiments, we set $B = 64$ in order to obtain the highest possible accuracy.

When $B = 1$, the transition-based parser becomes a deterministic parser. By comparing the curves when $B = 1$ and $B = 2$, we can see that, while the use of search reduces the parsing speed, it improves the quality of the output parses. Therefore, beam-search is a reasonable choice for transition-based parsing.

5.2 Accuracy comparisons
The test accuracies are shown in Table 5, where each row represents a parsing model.

| Model                | Word | Complete |
|----------------------|------|----------|
| MSTParser 1          | 90.7 | 36.7     |
| Graph [M]            | 91.2 | 40.8     |
| Transition           | 91.4 | 41.8     |
| Graph [MA]           | 91.4 | 42.5     |
| MSTParser 2          | 91.5 | 42.1     |
| Combined [TM]        | 92.0 | 45.0     |
| Combined [TMA]       | 92.1 | 45.4     |

Table 5: Accuracy comparisons using PTB 3

As can be seen from the table, beam-search reduced the head word accuracy from 91.5%/42.1% (“MSTParser 2”) to 91.2%/40.8% (“Graph [M]”) with the same features as exact-inference. However, with only two extra feature templates from Table 2, which are not conjoined with direction or distance information, the accuracy is improved to 91.4%/42.5% (“Graph [MA]”). This improvement can be seen as a benefit of beam-search, which allows the definition of more global features.
split the corpus into training, development and test data as shown in Table 6, and use the head-finding rules in Table 8 in the Appendix to turn the bracketed sentences into dependency structures. Most of the head-finding rules are from Sun and Jurafsky (2004), while we added rules to handle NN and FRAG, and a default rule to use the rightmost node as the head for the constituent that are not listed.

Like Duan et al. (2007), we use gold-standard POS-tags for the input. The parsing accuracy is evaluated by the percentage of non-root words that have been assigned the correct head, the percentage of correctly identified root words, and the percentage of complete matches, all excluding punctuation.

The accuracies are shown in Table 7. Rows “Graph [MA]”, “Transition”, “Combined [TM]” and “Combined [TMA]” show our models in the same way as for the English experiments from Section 5.2. Row “Duan 2007” represents the transition-based model from Duan et al. (2007), which applies beam-search to the deterministic model from Yamada and Matsumoto (2003), and achieved the previous best accuracy on the data.

Our observations on parsing Chinese are essentially the same as for English. Our combined parser outperforms both the pure graph-based and the pure transition-based parsers. It gave the best accuracy we are aware of for dependency parsing using CTB.

6 Related work

Our graph-based parser is derived from the work of McDonald and Pereira (2006). Instead of performing exact inference by dynamic programming, we incorporated the linear model and feature templates from McDonald and Pereira (2006) into our beam-search framework, while adding new global features. Nakagawa (2007) and Hall (2007) also showed the effectiveness of global features in improving the accuracy of graph-based parsing, using the approximate Gibbs sampling method and a reranking approach, respectively.

Our transition-based parser is derived from the deterministic parser of Nivre et al. (2006). We incorporated the transition process into our beam-search framework, in order to study the influence of search on this algorithm. Existing efforts to add search to deterministic parsing include Sagae
and Lavie (2006b), which applied best-first search to constituent parsing, and Johansson and Nugues (2006) and Duan et al. (2007), which applied beam-search to dependency parsing. All three methods estimate the probability of each transition action, and score a state item by the product of the probabilities of all its corresponding actions. But different from our transition-based parser, which trains all transitions for a parse globally, these models train the probability of each action separately. Based on the work of Johansson and Nugues (2006), Johansson and Nugues (2007) studied global training with an approximated large-margin algorithm. This model is the most similar to our transition-based model, while the differences include the choice of learning and decoding algorithms, the definition of feature templates and our application of the “early update” strategy.

Our combined parser makes the biggest contribution of this paper. In contrast to the models above, it includes both graph-based and transition-based components. An existing method to combine multiple parsing algorithms is the ensemble approach (Sagae and Lavie, 2006a), which was reported to be useful in improving dependency parsing (Hall et al., 2007). A more recent approach (Nivre and McDonald, 2008) combined MSTParser and MaltParser by using the output of one parser for features in the other. Both Hall et al. (2007) and Nivre and McDonald (2008) can be seen as methods to combine separately defined models. In contrast, our parser combines two components in a single model, in which all parameters are trained consistently.

7 Conclusion and future work

We developed a graph-based and a transition-based projective dependency parser using beam-search, demonstrating that beam-search is a competitive choice for both parsing approaches. We then combined the two parsers into a single system, using discriminative perceptron training and beam-search decoding. The appealing aspect of the combined parser is the incorporation of two largely different views of the parsing problem, thus increasing the information available to a single statistical parser, and thereby significantly increasing the accuracy. When tested using both English and Chinese dependency data, the combined parser was highly competitive compared to the best systems in the literature.

The idea of combining different approaches to the same problem using beam-search and a global model could be applied to other parsing tasks, such as constituent parsing, and possibly other NLP tasks.

Acknowledgements

This work is supported by the ORS and Clarendon Fund. We thank the anonymous reviewers for their detailed comments.

Appendix

| Constituent | Rules |
|-------------|-------|
| ADJP        | r ADJP JJ AD; r |
| ADVP        | r ADVP AD CS JJ NP PP P VA VV; r |
| CLP         | r CLP M NN NP; r |
| CP          | r CP IP VP; r |
| DNP         | r DEG DNP DEC QP; r |
| DP          | r M; l DP DT OD; l |
| DVP         | r DEV AD VP; r |
| FRAG        | r VV NR NN NT; r |
| IP          | r VP IP NP; r |
| LCP         | r LCP LC; r |
| LST         | r CD NP QP; r |
| NP          | r NP NN IP NR NT; r |
| NN          | r NP NN IP NR NT; r |
| PP          | l P PP; l |
| PRN         | l PU; l |
| QP          | r QP CLP CD; r |
| UCP         | l IP NP VP; l |
| VCD         | l VV VA VE; l |
| VP          | l VE VC VV VNV VPT VRD VSB VCD VP; l |
| VPT         | l VA VV; l |
| VRD         | l VV VA; l |
| VSB         | r VV VE; r |
| default     | r |

Table 8: Head-finding rules to extract dependency data from CTB
References

Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In Proceedings of CoNLL, pages 149–164, New York City, USA, June.

Xavier Carreras, Mihai Surdeanu, and Lluís Marquez. 2006. Projective dependency parsing with perceptron. In Proceedings of CoNLL, New York City, USA, June.

Xavier Carreras. 2007. Experiments with a higher-order projective dependency parser. In Proceedings of the CoNLL Shared Task Session of EMNLP/CoNLL, pages 957–961, Prague, Czech Republic, June.

Michael Collins and Brian Roark. 2004. Incremental parsing with the perceptron algorithm. In Proceedings of ACL, pages 111–118, Barcelona, Spain, July.

Michael Collins. 2002. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In Proceedings of EMNLP, pages 1–8, Philadelphia, USA, July.

Michael A. Covington. 2001. A fundamental algorithm for dependency parsing. In Proceedings of the ACM Southeast Conference, Athens, Georgia, March.

Xiangyu Duan, Jun Zhao, and Bo Xu. 2007. Probabilistic models for action-based chinese dependency parsing. In Proceedings of ECML/ECPKDD, Warsaw, Poland, September.

Jason Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In Proceedings of COLING, pages 340–345, Copenhagen, Denmark, August.

Johan Hall, Jens Nilsson, Joakim Nivre, Gülsen Eryiğit, Beáta Megyesi, Mattias Nilsson, and Markus Saers. 2007. Single malt or blended? a study in multilingual parser optimization. In Proceedings of the CoNLL Shared Task Session of EMNLP/CoNLL, pages 933–939, Prague, Czech Republic, June.

Keith Hall. 2007. K-best spanning tree parsing. In Proceedings of ACL, Prague, Czech Republic, June.

Richard Johansson and Pierre Nugues. 2006. Investigating multilingual dependency parsing. In Proceedings of CoNLL, pages 206–210, New York City, USA, June.

Richard Johansson and Pierre Nugues. 2007. Incremental dependency parsing using online learning. In Proceedings of the CoNLL/EMNLP, pages 1134–1138, Prague, Czech Republic.

Philip Koehn, Franz Och, and Daniel Marcu. 2003. Statistical phrase-based translation. In Proceedings of NAACL/HLT, Edmonton, Canada, May.

Terry Koo, Xavier Carreras, and Michael Collins. 2008. Simple semi-supervised dependency parsing. In Proceedings of ACL/HLT, pages 595–603, Columbus, Ohio, June.

Ryan McDonald and Joakim Nivre. 2007. Characterizing the errors of data-driven dependency parsing models. In Proceedings of EMNLP/CoNLL, pages 122–131, Prague, Czech Republic, June.

R McDonald and F Pereira. 2006. Online learning of approximate dependency parsing algorithms. In Proc. of EACL, pages 81–88, Trento, Italy, April.

Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005. Online large-margin training of dependency parsers. In Proceedings of ACL, pages 91–98, Ann Arbor, Michigan, June.

Tetsuji Nakagawa. 2007. Multilingual dependency parsing using global features. In Proceedings of the CoNLL Shared Task Session of EMNLP/CoNLL, pages 952–956, Prague, Czech Republic, June.

Joakim Nivre and Ryan McDonald. 2008. Integrating graph-based and transition-based dependency parsers. In Proceedings of ACL/HLT, pages 950–958, Columbus, Ohio, June.

Joakim Nivre, Johan Hall, Jens Nilsson, Gülsen Eryiğit, and Svetoslav Marinov. 2006. Labeled pseudo-projective dependency parsing with support vector machines. In Proceedings of CoNLL, pages 221–225, New York City, USA, June.

K Sagae and A Lavie. 2006a. Parser combination by reparsing. In In Proc. HLT/NAACL, pages 129–132, New York City, USA, June.

Kenji Sagae and Alon Lavie. 2006b. A best-first probabilistic shift-reduce parser. In Proceedings of COLING/ACL (poster), pages 691–698, Sydney, Australia, July.

Honglin Sun and Daniel Jurafsky. 2004. Shallow semantic parsing of Chinese. In Proceedings of NAACL/HLT, Boston, USA, May.

H Yamada and Y Matsumoto. 2003. Statistical dependency analysis using support vector machines. In Proceedings of IWPT, Nancy, France, April.