

Parse Reranking Based on Higher-Order Lexical Dependencies

Zhiguo Wang and Chengqing Zong

National Laboratory of Pattern Recognition

Institute of Automation, Chinese Academy of Sciences, Beijing, China, 100190

{zgwang, cqzong}@nlpr.ia.ac.cn

Abstract

Existing work shows that lexical dependencies are helpful for constituent tree parsing. However, only first-order lexical dependencies have been employed and investigated in previous work. In this paper, we propose a method to employing higher-order¹ lexical dependencies for constituent tree evaluation. Our method is based on a parse reranking framework, which provides a constrained search space (via N -best lists or parse forests) and enables our parser to employ relatively complicated dependency features. We evaluate our models on the Penn Chinese Treebank. The highest F_1 score reaches 85.74%, thus outperforming all previously reported state-of-the-art systems. The dependency accuracy of constituent trees generated by our parser has been significantly improved as well.

1 Introduction

The most commonly used grammar for constituent structure parsing is probabilistic context-free grammar (PCFG). However, as demonstrated in Klein and Manning (2003a), PCFG estimated straightforwardly from Treebank does not perform well. The reason is that the basic PCFG has certain recognized drawbacks: its independence assumption is too strong, and it lacks of lexical conditioning (Jurafsky and Martin, 2008). To address these drawbacks, several variants of PCFG-based models have been proposed (Klein and Manning, 2003a; Matsuzaki et al., 2005; Petrov et al., 2006; Petrov and Klein, 2007). Lexicalized PCFG (LPCFG) (Collins, 1999; Charniak, 2000; Bikel, 2004) is a representative work that tries to ameliorate the deficiency of lexical conditioning. In LPCFG, non-terminals are annotated with lexical heads and the probabilities of CFG rules are estimated conditioned upon these lexical heads. Thus LPCFG becomes sensitive to lexical heads, and its performance is improved. However, the information provided by lexical heads is limited. To obtain higher parsing performance, we must seek additional informa-

tion. We believe that dependency trees are good candidates because they encode grammatical relations between words and provide much more lexical conditioning than lexical heads for PCFG.

Dependency trees are usually factored into sets of lexical dependency parts for evaluation. The order of a lexical dependency part can be defined according to the number of dependency arcs it contains. For example, in Figure 1, *dependency* is first-order, *sibling* and *grandchild* are second-order and *grand-sibling* and *tri-sibling* are third-order. During the past few years, higher-order¹ lexical dependencies have been successfully used for dependency parsing (McDonald et al., 2005; McDonald and Pereira, 2006; Koo and Collins, 2010). But for constituent tree evaluation, only first-order (bigram) lexical dependencies have been used (Collins, 1996; Klein and Manning, 2003a; Collins and Koo, 2005). However, first-order lexical dependency parts are quite limited and thus lose much of the contextual information within the dependency tree. To improve parsing performance, we propose to evaluate constituent trees with higher-order lexical dependencies.

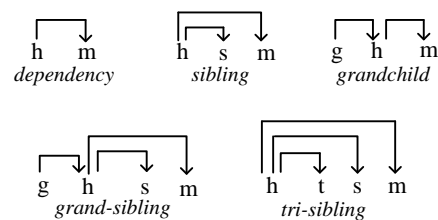


Figure 1. Lexical dependency types. The lower-case letters h, m, s, g are words in a sentence.

In this paper, we propose a method for evaluating constituent trees using higher-order lexical dependencies within a parse reranking framework. We evaluate our method on the Penn Chinese Treebank (CTB). The F_1 score reaches 85.74%, thus outperforming the best previously reported systems. Thanks to the lexical dependencies, the dependency accuracy of the generated constituent trees is improved as well. These experimental results show that higher-order lexi-

¹ Lexical dependency part which contains more than one dependency arcs is called higher-order, e.g., *sibling*, *grandchild* and *grand-sibling* in Figure 1.

cal dependencies are highly beneficial for constituent tree evaluation.

The remainder of this paper is organized as follows: Section 2 briefly reviews related work and proposes our ideas. Section 3 describes our parsing approach. Section 4 describes our parse reranking algorithms based on higher-order lexical dependencies. In Section 5, we describe our training algorithms. We discuss and analyze our experiments in Section 6. Finally, we conclude and mention future work in Section 7.

2 Related Work and Our Ideas

Over the past few years, two kinds of *parse reranking* methods have been proposed. The first is *N*-best reranking (Charniak and Johnson, 2005; Collins and Koo, 2005). In this method, an existing generative parser is used to enumerate *N*-best parse trees for an input sentence, and then a reranking model is used to rescore the *N*-best lists with the help of various sorts of features. However, the *N*-best reranking method suffers from the limited scope of the *N*-best list in that potentially good alternatives may have been ruled out. The second method, called the forest reranking model, was proposed by Huang (2008). In Huang’s method, a forest, instead of an *N*-best list, is generated first. Then a beam search algorithm is used to generate *N*-best sub-trees for each node in bottom-up order and the best-first sub-tree of the root node is chosen as the final parse tree.

In recent years, there have been many attempts to use dependency trees for constituent parsing. All these approaches can be classified into three types. The first type is *dependency-driven* constituent parsing (Hall et al., 2007; Hall and Nivre, 2008). Given an input sentence, this approach first parses it into a labeled dependency tree (with complex arc labels, which makes it possible to recover the constituent tree) and then transforms the dependency tree into a constituent tree. The second approach is *dependency-constrained* constituent parsing (Xia and Palmer, 2001; Xia et al., 2008; Wang and Zhang, 2010; Wang and Zong, 2010). In this approach, dependency trees, once generated, are used to constrain the search space of a constituent parser. The third approach is *dependency-based* constituent parsing (Collins, 1996; Klein and Manning, 2003b). In this approach, the constituent tree is evaluated with the help of its corresponding lexical dependencies.

All three existing approaches have certain limitations. In the first approach, the dependency-

driven constituent parser is not constrained by the Treebank grammar, so a constituent tree transformed from its corresponding dependency tree may contain context-free productions not seen in the Treebank grammar. Although this limitation may not affect the parsing F_1 score, it often has undesirable effects on applications. For the second approach, if the generated dependency tree includes some erroneous parts, the correct constituent tree may be pruned out directly, leaving no way to recover the correct tree again. The third approach parses sentences making use of first-order lexical dependencies only. As mentioned, first-order lexical dependencies are quite limited, and thus may lose much information about the grammatical relations between words. Consequently, the performance improvement of this approach is limited as well.

To overcome the drawbacks of the existing approaches, we propose to evaluate constituent trees using higher-order lexical dependencies within a parse reranking framework. Our approach has the following advantages: 1) It utilizes the higher-order lexical dependencies, which provide more contextual information within the dependency tree for constituent tree evaluation; 2) the parse reranking method provides high-quality candidates (*N*-best list or parse forest) which yields a small search space, enabling the use of relatively complicated features.

3 Our Approach

For a sentence x , we define constituent parsing as a search for the highest-scoring parse c^* of x :

$$c^* = \arg \max_{c \in GEN(x)} Score(x, c) \quad (1)$$

Where, $GEN(x)$ is a set of candidate parsers for x , and $Score(x, c)$ evaluates the event that tree c is the parse of sentence x .

In order to evaluate c with higher-order lexical dependencies, we define:

$$Score(x, c) = \Phi(x, c) \cdot \bar{\alpha} = \sum_i \alpha_i \Phi_i(x, c) \quad (2)$$

Where, Φ maps each $(x, c) \in X \times C$ to lexical dependency feature vector $\Phi(x, y) \in \mathfrak{R}^d$, and $\bar{\alpha} \in \mathfrak{R}^d$ is the corresponding weight vector.

3.1 Representation of Constituent Tree with Labeled Dependency Tree

The discriminative parsing model in Eq. (1) takes lexical dependencies as features, so we must design a method of representing constituent trees

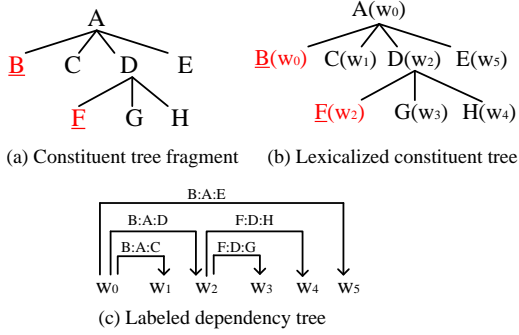


Figure 2. Representation of constituent tree with labeled dependency tree

with associated dependency trees. Our method includes the following two steps:

Step 1: Lexicalize the constituent tree, i.e. annotate each node in the constituent tree with its head-word. First, find the head-child of each non-terminal node using a head percolation table (Yamada and Matsumoto, 2003). For example, in Figure 2(a), node B is identified as the head-child of rule $A \rightarrow B C D E$. Then the head-words propagate up through the leaf nodes and each parent receives its head-word from its head-child. For example, in Figure 2(b), w_0 is propagated up from node B to A. According to this procedure, we can get the lexicalized constituent tree (shown in Figure 2(b)) for the constituent fragment shown in Figure 2(a).

Step 2: Transform the lexicalized tree into a labeled dependency tree. First, let the head-word of each non-head-child depend on the head-word of the head-child for each rule. For example, in Figure 2(b) for rule $A \rightarrow B C D E$, the head words of non-head-child (node C, D and E) which are w_1 , w_2 and w_5 should depend on w_0 which is the head word of head-child (node B). In order to encoding the syntactic symbols in the constituent tree into dependency tree, we annotate each dependency arc with a label $N_h : P : N_m$, where N_h is the head-child’s syntactic category, P is the parent’s syntactic category and N_m is the non-head-child’s syntactic category. For example, in Figure 2(c), the dependency arc between w_1 and w_0 is built through rule $A \rightarrow B C D E$, where w_0 associates with B, w_1 associates with C and the parent node is A, so we can annotate the dependency arc with B:A:C. According to the procedure, the lexicalized tree in Figure 2(b) can be transformed into the labeled dependency tree shown in Figure 2(c).

3.2 Mapping Higher-Order Lexical Dependencies into Feature Vectors

To map lexical dependencies into feature vectors,

Algorithm 1: Constituent Tree Evaluating

```

1: function Eval( $C$ )
2: for  $P \in C$  in bottom up topological order do
3:   EvalSubTree ( $C_p$ )
4: return Score( $C$ )
5:
6: procedure EvalSubTree ( $C_p$ )
7:    $\triangleright$  Assume the constituent is  $P \rightarrow N_1 \dots N_n$ 
8:   Find the head-child  $N_h$  for  $P$ 
9:    $W_p \leftarrow W_{N_h}$ 
10:   $\triangleright$  Building  $D_p$ 
11:  for  $N_i \in \{N_1, \dots, N_n\} \setminus N_h$  do
12:    Link  $D_{N_h}$  and  $D_{N_i}$  with a dependency arc
13:    Annotate the arc with label  $N_h : P : N_i$ 
14:    Make the root of  $D_{N_h}$  as  $D_p$ ’s root
15:    Extract all lexical dependencies for  $P$ 
        and map them into feature vector  $\Phi(P)$ 
16:   $Score(C_p) = \Phi(P) \cdot \bar{\alpha} + \sum_{i=1}^n Score(C_{N_i})$ 

```

we define certain feature templates, as shown in Table 1. We work with binary indicator features² for each lexical dependency. The feature vector $\Phi(x, C)$ of constituent tree C can be calculated through the dependency tree D transformed from C using the follow formula:

$$\Phi(x, C) = \sum_{d \in S(D)} \phi(d) \quad (3)$$

In this formula $S(D)$ is a set of all the lexical dependencies extracted from D , and d is a lexical dependency in $S(D)$. The function ϕ is used to map each lexical dependency d into feature vector according to the templates in Table 1.

4 Parse Reranking Algorithms

A critical problem when training the discriminative model in Eq. (1) is the extensive training time required, in which we must parse all the sentences in the training set repeatedly. In this paper, we adopt an approximate method: parse reranking. In parse reranking, $GEN(S)$ in Eq. (1) is an N -best list or a parse forest which provides a small and well-formed search space for constituent parsing. Given this small space, we can exploit higher-order lexical dependencies efficiently.

² Binary indicator features are defined as follows: if a certain feature is observed in an instance, the value of that feature is 1; otherwise, the value is 0.

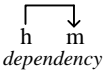
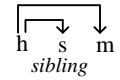
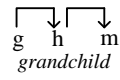
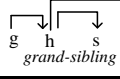
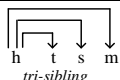
 <i>dependency</i>	Basic Uni-gram Features h , POS(h), N(h) h , POS(h) h , N(h) m , POS(m), N(m) m , POS(m) m , N(m)	 <i>sibling</i>	POS(h),N(h),POS(s),N(s),P(s),POS(m),N(m),P(m) POS(h),N(h),N(s),P(s),N(m),P(m) POS(h),N(h),POS(s),P(s),POS(m),P(m) POS(h),N(h),POS(s),N(s),POS(m),N(m) POS(h),POS(s),POS(m) N(h),N(s),N(m) N(h),P(s),P(m)
	Basic Bi-gram Features P(m) , h , POS(h), N(h), m , POS(m), N(m) h , POS(h), N(h), m , POS(m), N(m) P(m) .POS(h), N(h), POS(m), N(m) P(m) , h , N(h), m , N(m) P(m) .h , POS(h), m , POS(m) P(m) .h , m P(m) , POS(h),POS(m) P(m) , N(h), N(m)	 <i>grandchild</i>	POS(g),N(g),POS(h),N(h),P(h),POS(m),N(m),P(m) POS(g),N(g),N(h),P(h),N(m),P(m) POS(g),N(g),POS(h),P(h),POS(m),P(m) POS(g),N(g),POS(h),N(h),POS(m),N(m) POS(g),POS(h),POS(m) N(g),N(h),N(m) N(g),P(h),P(m)
	Surrounding Word POS Features P(m), N(h), POS(h), N(m), POS(m), POS(h)+1, POS(m)-1 P(m), N(h), POS(h), N(m), POS(m), POS(h)-1, POS(m)-1 P(m), N(h), POS(h), N(m), POS(m), POS(h)+1, POS(m)+1 P(m), N(h), POS(h), N(m), POS(m), POS(h)-1, POS(m)+1	 <i>grand-sibling</i>	POS(g),POS(h),POS(s),POS(m) N(g),N(h),N(s),N(m) N(g),P(h),P(s),P(m)
		 <i>tri-sibling</i>	POS(h),POS(t),POS(s),POS(m) N(h),N(t),N(s),N(m) N(h),P(t),P(s),P(m)

Table 1. Feature templates of various lexical dependency types. The lowercase letters h, m, s, g are words in a sentence. POS(x) is x’s POS tag. POS(x)+1 is the POS tag of the word to the right of x. POS(x)-1 is the POS tag of the word to the left of x. P(x), N(x) are syntactic categories of P and N_h (or N_m), which are annotated on dependency arcs (We ignore dependency arc labels in the table for simplicity. More details can be found in section 3.2).

4.1 N -best Reranking Based on Higher-Order Lexical Dependencies

The method of sub-section 3.1 determines that each constituent sub-tree must have a corresponding dependency sub-tree. Accordingly, we now describe an efficient algorithm for evaluating constituent trees with higher-order lexical dependencies. We define a quadruple $\langle C_N, D_N, score(C_N), W_N \rangle$ for each non-terminal node N , in which C_N is the constituent sub-tree rooted at N ; D_N is the dependency sub-tree transformed from C_N ; $score(C_N)$ is the score of C_N evaluated using Eq. (2); and W_N is the head-word of N in the tree.

Our algorithm (Algorithm 1) works bottom-up to fill $\langle C_N, D_N, score(C_N), W_N \rangle$ for each node N . For a constituent P in the parse tree, we first find the head-child N_h for P (line 8), then propagate the head-word of N_h to P (line 9). To build D_P , we simply build dependency arcs for current constituent P ; then link D_{N_1}, \dots, D_{N_n} with these dependency arcs; and then let the root of D_{N_n} be D_P ’s root (line 11 to line 14). We extract all the lexical dependencies rooted at P ’s head-word W_p through D_P . For example, in Figure 2(b), all the lexical dependencies rooted at node A’s head-word w_0 can be extracted from the dependency tree in Figure 2(c); and all the lexical dependencies have been shown in

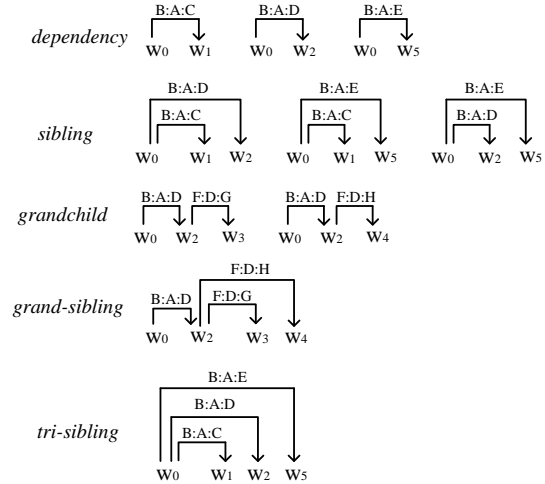


Figure 3. Lexical dependencies for w_0 in Figure 2(c).

Figure 3. Then we map the lexical dependencies into feature vectors and sum over them as the feature vector $\Phi(P)$ for P . Finally, we evaluate the score of C_p using formula (4) below:

$$Score(C_p) = \Phi(P) \cdot \bar{\alpha} + \sum_{i=1}^n Score(C_{N_i}) \quad (4)$$

4.2 Forest Reranking Based on Higher-Order Lexical Dependencies

As mentioned, N -best reranking suffers from the limited scope of N -best list. Forest reranking, by contrast, can rerank a packed forest of exponentially many parses, and thus provides a good way to overcome these limitations. Thus we also use the forest reranking method, based on higher-order lexical dependencies.

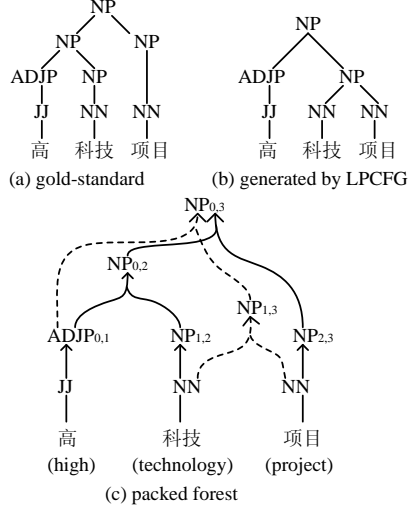


Figure 4. Constituent trees and forest

A forest is a compact representation of many parse trees. Figure 4(c) is a sample forest which is the compact representation of the constituent trees shown in Figures 4(a) and 4(b). To obtain forests, Huang (2008) tried to modify the Charniak parser to output forest directly. Inspired by parser combination methods (Sagae and Lavie, 2006; Fossum and Knight, 2009), we have designed a simple method of building forests starting from N -best lists. First, we convert each parse tree in an N -best list into context-free productions and label each constituent in each production with its span and syntactic category. Then these converted context-free productions are used to build the forest. For example, in Figure 4, given two candidates (Figure 4(a) and Figure 4(b)), we first convert them into context-free productions, e.g. $NP_{0,3} \rightarrow ADJP_{0,1} NP_{1,3}$, $NP_{0,3} \rightarrow NP_{0,2} NP_{2,3}$ and so on. Then we combine these productions into the forest shown in Figure 4(c). The recombinced forest probably contains some parse trees that are not included in the N -best list, as will be shown in sub-section 6.1.

Our algorithm for forest reranking is similar to Algorithm 1. The only difference is that there may be more than one hyperedge for each node in a forest. So we make use of a beam search algorithm (Huang and Chiang, 2005) and store N -best sub-trees for each internal node. Finally, we choose the best-first sub-tree of the root node as the result.

5 Training Algorithm

The training task is to tune the parameter weights $\bar{\alpha}$ in Eq. (1) using the training examples as evidence. We employ the online-learning algorithm shown in Algorithm 2 because it has been proven

Algorithm 2: Generic online learning algorithm

- 1: **Input:** training data (x_t, c_t) for $t = 1 \dots T$
 - 2: $\bar{\alpha}^{(0)} \leftarrow 0$; $\mathbf{v} \leftarrow 0$; $i \leftarrow 0$ \triangleright initial weights
 - 3: **for** n in $1 \dots N$ **do** $\triangleright N$ iterations
 - 4: **for** t in $1 \dots T$ **do** $\triangleright T$ training instances
 - 5: $\bar{\alpha}^{(i+1)} \leftarrow$ update $\bar{\alpha}^{(i)}$ according to (x_t, c_t)
 - 6: $\mathbf{v} \leftarrow \mathbf{v} + \bar{\alpha}^{(i+1)}$
 - 7: $i \leftarrow i + 1$
 - 8: $\bar{\alpha} \leftarrow \mathbf{v} / (N * T)$ \triangleright averaging weights
 - 9: **return** $\bar{\alpha}$
-

to be effective and efficient in many studies (Collins, 2002; Collins and Roark, 2004; McDonald et al., 2005). For Algorithm 2, we define two parameter update strategies (line 5 in Algorithm 2) as follows.

The first strategy is *perceptron updating*. We first obtain the oracle tree c_t^+ that has the highest F_1 score according to the gold-standard tree c_t ,

$$c_t^+ = \arg \max_{c \in GEN(x_t)} F_1(c, c_t) \quad (5)$$

Then we get the highest scoring tree \hat{c}_t with current weights $\bar{\alpha}^{(i)}$,

$$\hat{c}_t = \arg \max_{c \in GEN(x_t)} \Phi(x_t, c) \cdot \bar{\alpha}^{(i)} \quad (6)$$

If \hat{c}_t is not equal to c_t^+ , the weights will be updated through

$$\bar{\alpha}^{(i+1)} \leftarrow \bar{\alpha}^{(i)} + \Phi(c_t^+) - \Phi(\hat{c}_t) \quad (7)$$

Otherwise, the current weights are kept.

Although the perceptron updating strategy works well, parameter updating must wait until the entire tree has been built. We believe that this strategy probably misses the best opportunity for parameter updating and introduces some noise into the updating procedure. So, inspired by Collins and Roark (2004), we propose an *early updating strategy* for forest reranking. The key idea is to insert the parameter updating procedure into the forest reranking procedure. We parse a forest bottom up with the current parameter $\bar{\alpha}^{(i)}$. When the best-first sub-tree \hat{s}_N for internal node N is different from oracle sub-tree s_N^+ , we stop the parsing procedure and update the parameters immediately using the following formula:

$$\bar{\alpha}^{(i+1)} \leftarrow \bar{\alpha}^{(i)} + \Phi(s_t^+) - \Phi(\hat{s}_t).$$

Then we continue to parse the current forest with the newer parameters $\bar{\alpha}^{(i+1)}$. Unlike the perceptron updating strategy, this strategy updates parameters at the moment that an error sub-tree is built, and this is why we call it the *early updating strategy*.

6 Experiments and Analysis

We evaluate our method on the Penn Chinese Treebank Version 5.0 with the standard division: Art.301-325 as the development set, Art. 271-300 as the test set and others as the training set. All the F_1 scores are evaluated with EVALB³.

6.1 To Obtain N -best Lists and Forests

We first employ existing parsers to generate N -best lists and then recombine the N -best lists into forests according to the method described in sub-section 4.2. We split the training set into 20 folds averagely and generate 50-best lists for one fold with both the Berkeley parser⁴ and the Charniak parser⁵ (trained on the remaining 19 folds) individually. The development set and the test set are parsed with models trained on the entire training set.

	Berkley(50)	Charniak(50)	Comb(100)
Nbest	89.13	89.20	91.61
Forest	90.22	90.38	94.05

Table 2. Oracle F_1 (%) of N -best lists and forests

The oracle F_1 scores of N -best lists and forests on test set are listed in Table 2, where ‘Berkeley(50)’ means the performance of 50-best lists from Berkeley parser; ‘Charniak(50)’ means the performance of 50-best list from Charniak parser; ‘Comb(100)’ means the performance of 100-best lists by combining the two 50-best lists; ‘Nbest’ means the oracle F_1 of N -best lists; and ‘Forest’ means the oracle F_1 of forests which are evaluated through the Forest Oracle Algorithm proposed in Huang (2008). In Table 2, we can see that the oracle F_1 scores of forests are much better than associated N -best lists. This result clearly demonstrates that the approach of obtaining forests by recombining N -best lists is effective.

6.2 Parameter Tuning on Development Set

We tuned some parameters manually for our models in the sub-section, including the number of iterations in the training algorithm, and the beam size k in the forest reranking algorithm. Models are trained with training set’s 100-best lists and evaluated on development set’s 100-best lists.

The F_1 score curves varying with iteration times are shown in Figure 5. Although there are some fluctuations, we can see that the F_1 score

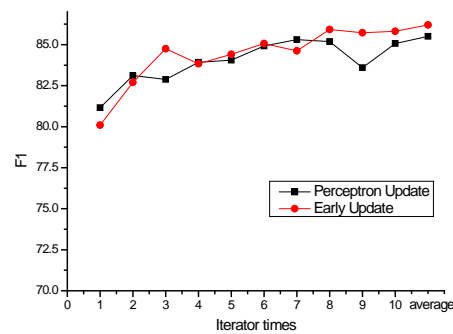


Figure 5. The F_1 score curves on the development set varying with iteration times in Algorithm 2.

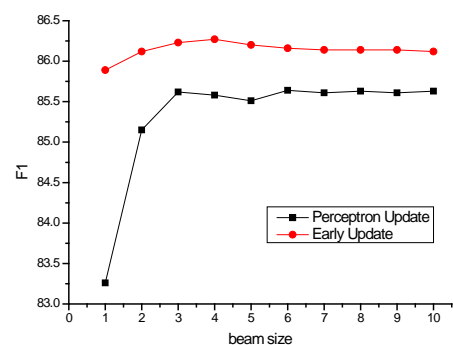


Figure 6. The F_1 score curves on the development set varying with beam size in forest reranking.

tends to improve with the incremental iteration times, and that the average model yields additional improvement. To avoid the problem of overfitting to the training set, we fix the iteration times at 10 in the following experiments. Figure 6 shows F_1 score curves varying with beam size. We see that when the beam size exceeds 5, the performance fluctuates slightly, so we fix the beam size at 5 in our experiments. In Figure 6, we can also see that the model trained with the early updating strategy can obtain better performance than with the perceptron updating strategy.

6.3 Evaluation on Test Set

In this sub-section, we build three parsing systems using the methods described in the previous sections. For brevity, we annotate the N -best reranking system trained with the perceptron updating strategy as ‘NbestRerank’; the forest reranking system trained with the perceptron updating strategy as ‘ForestRerank’; and the forest reranking system trained with the early updating strategy as ‘EarlyUpdate’. We also employ the Charniak parser (Charniak) and the Berkeley

³ <http://nlp.cs.nyu.edu/evalb/>

⁴ <http://code.google.com/p/berkeleyparser/>

⁵ <http://bllip.cs.brown.edu/download/reranking-parserAug06.tar.gz>

	Berkeley	Charniak	Combine
Baseline	83.13	82.41	-----
NbestRerank	84.68	83.29	84.68
ForestRerank	84.31	83.11	85.72
EarlyUpdate	85.06	83.32	85.74

Table 3. F1 (%) scores on Test Set. The column headed by “Berkeley” is trained and tested with Berkeley parser’s 50-best list; the column headed by “Charniak” is trained and tested with Charniak parser’s 50-best list; the column headed by “Combine” is trained and tested with 100-best list generated by Berkeley parser and Charniak parser.

Parsers	UA(%)
Charniak	82.31
Berkeley	84.05
NbestRerank	85.89
ForestRerank	85.69
EarlyUpdate	86.26
MST 1-ord (automatic POS)	79.62
MST 2-ord (automatic POS)	80.24
MST 1-ord (gold-standard POS)	85.23
MST 2-ord (gold-standard POS)	86.66

Table 4. Unlabeled dependency accuracy (UA). NbestRerank, ForestRerank and EarlyUpdate are trained and tested with combined 100-best lists

parser (Berkeley) as our baselines.

Using the parameter configuration tuned on development set, we have evaluated all the systems on test set. The F_1 scores are shown in Table 3. We can find that the F_1 scores are improved enormously when we make use of higher-order lexical dependencies. No matter which N -best list is used, EarlyUpdate system gets the highest F_1 . However, the improved ranges vary with N -best list. The improvement is 1.93% for Berkeley parser’s 50-best list, while it is 0.91% for Charniak parser’s 50-best list. In our opinion, the reason is that Charniak parser has made use of headword information during parsing, so it is less sensitive to lexical dependencies than Berkeley parser. When we use the combined 100-best lists for training and testing, all the three systems are improved. NbestRerank gets 1.55% improvements than Berkeley does, ForestRerank gets 1.04% improvements further than NbestRerank does, and EarlyUpdate makes the final performance up to 85.74%.

Intuitively, since they benefit from the higher-order lexical dependencies, the generated constituent trees should show better dependency accuracy as well. So we convert the generated constituent trees into dependency trees and calculate their unlabeled dependency accuracy (UA)⁶.

⁶ To compare with dependency parsing systems whose de-

	F_1 (%)
Baseline	84.59
+ <i>dependency</i> (first-order)	85.46
+ <i>sibling & grandchild</i> (second-order)	86.20
+ <i>grand-sibling & tri-sibling</i> (third-order)	86.37

Table 5. F_1 (%) score on development set of the EarlyUpdate system using different lexical dependency types.

To demonstrate the effectiveness of our systems, we also train a 1-order MSTParser⁷ (MST 1-ord) and a 2-order MSTParser (MST 2-ord), and then use them to parse the test set with gold-standard POS tags and automatically annotated POS tags (accuracy is 95.17%). All of the results are shown in Table 4. We see that the UAs of our systems are much better than those of Charniak and Berkeley. Although our systems employ no gold-standard POS tags during parsing, their UAs exceed those of MST 1-ord, which does employ such tags; and the UA of EarlyUpdate is even comparable with those of MST 2-ord, which also employs such tags.

The figures shown in Table 3 and Table 4 clearly reveal that our parsing approach obtains constituent trees with both better F_1 scores and better UAs.

6.4 Ablation studies

The experimental results above have shown that reranking parses based on higher-order lexical dependencies is effective. To verify the contributions of different lexical dependency types, we further evaluate the development set using the EarlyUpdate system trained with combined forests. First, we reranked forests with first-order (*dependency*) lexical dependencies. Then we added the second-order (*sibling* and *grandchild*) lexical dependencies into our system. Finally, we added the third-order (*grand-sibling* and *tri-sibling*) lexical dependencies. All of the parsing results are shown in Table 5. It is clear that all of the lexical dependency types are helpful for constituent tree evaluation.

6.5 Comparison with State-of-the-art Results

Table 6 compares our best results with that of state-of-the-art parsers. Compared to the

pendency arc labels are different from ours, we simply calculate the UAs.

⁷ <http://sourceforge.net/projects/mstparser/>

Individual System	
(Petrov and Klein, 2007)	83.32
(Huang and Harper, 2009)	84.15
N-best Reranking	
Charniak & Johnson Reranker ⁸	83.30
Our NbestRerank System	84.68
Parsers Combination	
(Zhang et al., 2009)	85.45
Using Extra Resource	
(Burkett and Klein, 2008)	84.24
(Huang and Harper, 2009)	85.18
(Niu et al., 2009)	85.20
Reranking with Lexical Dependencies	
Our EarlyUpdate System	85.74

Table 6. F1 (%) scores of state-of-the-art methods compared with ours on the Chinese Treebank.

“Charniak & Johnson Reranker”⁸ which is a parse reranking system and exploits various sorts of features including 1-order lexical dependencies (Charniak and Johnson, 2005), our NbestRerank parser, which uses higher-order lexical dependency features, gets a higher F_1 . Comparing with the parsers combination system (Zhang et al., 2009) which combines scores evaluated by Berkeley parser and Charniak parser to evaluate a parse tree, our EarlyUpdate system haven’t used scores evaluated by first stage parsers and still gets a higher F1 score. Although our EarlyUpdate system uses no resources other than CTB, it still obtains better results than other parsers which have employed extra resources (Burkett and Klein, 2008; Huang and Harper, 2009; Niu et al., 2009). These comparisons allow us to confidently conclude that exploitation of higher-order lexical dependencies is highly beneficial for constituent parsing.

7 Conclusion and Future Work

We have presented a method for evaluating constituent trees using higher-order lexical dependencies. Within a parse reranking framework, our models rerank N -best lists and forests based on dependency features. Experimental results show that higher-order lexical dependencies can yield greater improvements in constituent parsing performance than commonly used first-order lexical dependencies. The best results of our models outperformed all previous results on the CTB, and the dependency accuracy of generated constituent trees is significantly improved as well. All of the results demonstrate that exploitation of

higher-order lexical dependencies provides significant benefits for constituent tree evaluation.

Although all of our experiments were carried out only on the Chinese Treebank, our method is language independent. It can be adapted to any languages which can represent constituent trees with labeled dependency trees. We will apply our methods to other languages in the future.

Acknowledgments

The research work has been funded by the Natural Science Foundation of China under Grant No. 60975053 and 61003160, supported by the External Cooperation Program of the Chinese Academy of Sciences, and also partially supported by the China-Singapore Institute of Digital Media (CSIDM) project under grant No. CSIDM-200804 as well. Sincere thanks to Mark Seligman for his careful revision work.

References

- Daniel M. Bikel, 2004. Intricacies of Collins' parsing model. *Computational Linguistics*, 30 (4). pages 479-511.
- David Burkett and Dan Klein, 2008. Two languages are better than one (for syntactic parsing). In *EMNLP 2008*.
- Eugene Charniak, 2000. A maximum-entropy-inspired parser. In *NAACL 2000*.
- Eugene Charniak and Mark Johnson, 2005. Coarse-to-fine n -best parsing and MaxEnt discriminative reranking. In *ACL 2005*.
- Michael Collins, 1999. Head-driven statistical models for natural language parsing. University of Pennsylvania.
- Michael Collins, 2002. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *EMNLP 2002*.
- Michael John Collins, 1996. A new statistical parser based on bigram lexical dependencies. In *ACL 1996*, pages 184-191.
- Michael Collins and Terry Koo, 2005. Discriminative reranking for natural language parsing. *Computational Linguistics*, 31 (1). pages 25-70.
- Michael Collins and Brian Roark, 2004. Incremental parsing with the perceptron algorithm. In *ACL 2004*.
- Victoria Fossum and Kevin Knight, 2009. Combining constituent parsers. In *NAACL 2009*, pages 253-256.

⁸ The F1 score of Charniak & Johnson Reranker on CTB was reported in Niu et al. (2009).

- Johan Hall and Joakim Nivre, 2008. A dependency-driven parser for German dependency and constituency representations. In PaGe-08, pages 47-54.
- Johan Hall, Joakim Nivre and Jens Nilsson, 2007. A Hybrid Constituency-Dependency Parser for Swedish. In NODALIDA 2007, pages 284-287.
- Liang Huang, 2008. Forest reranking: Discriminative parsing with non-local features. In ACL 2008.
- Liang Huang and David Chiang, 2005. Better k-best parsing. In IWPT 2005.
- Zhongqiang Huang and Mary Harper, 2009. Self-Training PCFG grammars with latent annotations across languages. In ACL 2009.
- Daniel Jurafsky and James H. Martin, 2008. *Speech and Language Processing: An introduction to natural language processing, computational linguistics, and speech recognition*: Prentice Hall.
- Dan Klein and Christopher D. Manning, 2003a. Accurate unlexicalized parsing. In ACL2003, pages 423-430.
- Dan Klein and Christopher D. Manning, 2003b. Fast exact inference with a factored model for natural language parsing. In NIPS 2003.
- Terry Koo and Michael Collins, 2010. Efficient Third-Order Dependency Parsers. In ACL 2010.
- Takuya Matsuzaki, Yusuke Miyao and Jun'ichi Tsujii, 2005. Probabilistic CFG with latent annotations. In ACL 2005.
- Ryan McDonald, Koby Crammer and Fernando Pereira, 2005. Online large-margin training of dependency parsers. In ACL 2005.
- Ryan McDonald and Fernando Pereira, 2006. Online learning of approximate dependency parsing algorithms. In EACL 2006.
- Zheng-Yu Niu, Haifeng Wang and Hua Wu, 2009. Exploiting heterogeneous treebanks for parsing. In ACL 2009.
- Slav Petrov, Leon Barrett, Romain Thibaux and Dan Klein, 2006. Learning accurate, compact, and interpretable tree annotation. In ACL 2006.
- Slav Petrov and Dan Klein, 2007. Improved inference for unlexicalized parsing. In ACL 2007.
- Kenji Sagae and Alon Lavie, 2006. Parser combination by reparsing. In NAACL 2006, pages 129-132.
- Rui Wang and Yi Zhang, 2010. Hybrid Constituent and Dependency Parsing with Tsinghua Chinese Treebank. In LREC 2010.
- Zhiguo Wang and Chengqing Zong, 2010. Phrase Structure Parsing with Dependency Structure. In Coling 2010.
- Fei Xia and Martha Palmer, 2001. Converting dependency structures to phrase structures. In The 1st Human Language Technology Conference (HLT-2001).
- Fei Xia, Owen Rambow, Rajesh Bhatt, Martha Palmer and Dipti Misra Sharma, 2008. Towards a multi-representational treebank. Proc. of the 7th Int'l Workshop on Treebanks and Linguistic Theories (TLT-7). pages.
- Hiroyasu Yamada and Yuji Matsumoto, 2003. Statistical dependency analysis with support vector machines. In IWPT 2003.
- Hui Zhang, Min Zhang, Chew Lim Tan and Haizhou Li, 2009. K-best combination of syntactic parsers. In EMNLP 2009.
- Ying Zhang, Stephan Vogel and Alex Waibel, 2004. Interpreting BLEU/NIST scores: How much improvement do we need to have a better system. In LREC 2004.