Parsing Dependencies

Abstract

We present a new algorithm for transforming dependency parse trees into phrase-structure parse trees. We cast the problem as structured prediction and learn a statistical model. Our algorithm is faster than traditional phrase-structure parsing and achieves accuracy near to the state of the art on English and Chinese benchmarks.

1 Introduction

[rewrote the intro. I think the paper will be more exciting if, instead of making claims about informativeness or suggesting preferences, we just say the way the world turned out, and then ask how things might have turned out if it had been different. –nas]

Natural language parsers typically produce phrase-structure (or constituent) trees or dependency trees. These representations capture some of the same syntactic phenomena, and the two can be produced jointly (Carreras et al., 2008; Rush et al., 2010). Yet it appears to be completely unpredictable which will be preferred by a particular application. Both continue to receive the attention of parsing researchers.

Further, it appears to be a historical accident that phrase-structure syntax was used in annotating the Penn Treebank, and that English dependency annotations are largely derived through mechanical, rule-based transformations applied to the Penn Treebank. Indeed, despite extensive work on direct-to-dependency parsing algorithms (which we call *d-parsing*), the most accurate dependency parsers for English still involve phrase-structure parsing (which we call *c-parsing*) followed by rule-based extraction of dependencies (Kong and Smith, 2014).

What if dependency annotations had come first? Because d-parsers are generally much faster than c-parsers, we consider an alternate pipeline: d-parse first, then transform the dependency representation into a phrase-structure tree constrained to be consistent with the dependency parse. This idea was explored by Xia and Palmer (2001) and Xia et al. (2009) using hand-written rules. Here we present a data-driven algorithm in the structured prediction framework. The approach can be understood as a pipeline, or as a specially-trained coarse-to-fine decoding algorithm where a d-parser provides "coarse" structure and the second stage refines it (**?**).

Our lexicalized phrase-structure parser is asymptotically faster than parsing with a lexicalized context-free grammar: $O(n^2)$ plus d-parsing, vs. $O(n^5)$ worst case runtime in sentence length n, with the same grammar constant. With simple pruning, our approach achieves linear observable runtime. The accuracy of the approach is similar to state-of-the-art phrase-structure parsers without reranking or semisupervised training.

[I suggest giving a brief roadmap here, or (better) integrating it into the discussion above –nas]

2 Background

We begin by following the conventional development by first introducing c-parsing and then defining d-parses through a mechanical conversion using head-rules. In the next section, we flip this development and consider the reverse transformation.

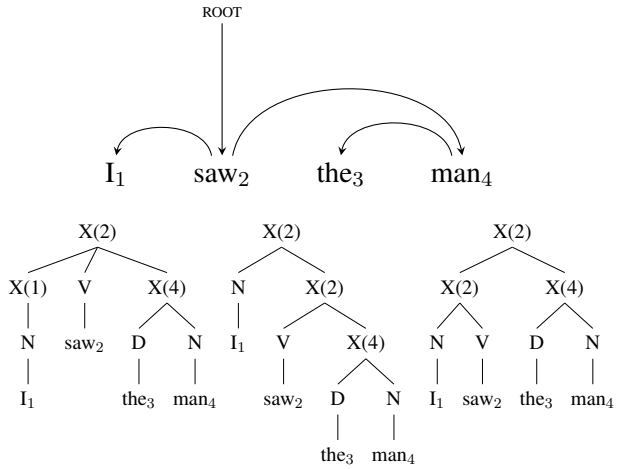


Figure 1: Head rules specify a deterministic transformation from c-parses to d-parses. The figure, adapted from (Collins et al., 1999), shows several simple CFG trees that all map the same d-parse. The parentheses $X(h)$ indicate the head h of each internal vertex.

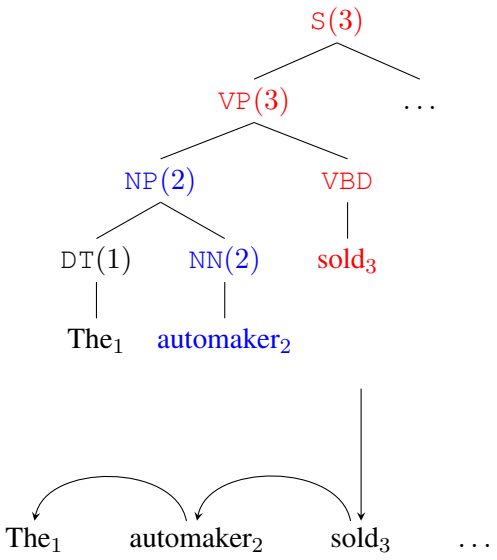


Figure 2: Illustration of c-parse to d-parse conversion with head rules $\{\texttt{VP} \rightarrow \texttt{NP}^* \texttt{ VBD}, \texttt{NP} \rightarrow \texttt{DT NP}^*, \ldots\}$. The c-parse is an ordered tree with fringe $x_1, \ldots, x_n$. Each vertex is annotated with a span, head, and syntactic tag. The blue and vertices have the words $\texttt{automaker}_2$ and $\texttt{sold}_3$ as heads respectively. The vertex $\texttt{VP}_3$ implies that $\texttt{automaker}_2$ is a dependent of $\texttt{sold}_3$, and that $d_2 = 3$ in the d-parse.

2.1 CFG Parsing

The phrase-structure trees annotated in the Penn Treebank are derivation trees from a context-free grammar. Define a binarized[1] context-free grammar (CFG) as a 4-tuple $(\mathcal{N}, \mathcal{R}, \mathcal{T}, r)$ where:

- $\mathcal{N}$; a set of nonterminal symbols, e.g. NP, VP,
- $\mathcal{T}$; a set of terminal symbols, consisting of the words in the language,
- $\mathcal{R}$; a set of binarized rule productions of the form $A \rightarrow \beta_1\ \beta_2$
- $r \in \mathcal{N}$; a distinguished root symbol.

Given an input sentence $x_1, \ldots, x_n$ of terminal symbols from $\mathcal{T}$, define the set of c-parses for the sentence as $\mathcal{Y}(x)$. This set consists of all binary ordered trees with fringe $x_1, \ldots, x_n$, internal nodes labeled from $\mathcal{N}$, all tree productions $A \rightarrow \beta$ consisting of members of $\mathcal{R}$, and root label r.

For a c-parse $y \in \mathcal{Y}(x)$, we further associate a span $\langle i, j \rangle$, each vertex in the tree. This specifies the sequence $\{x_i, \ldots, x_j\}$ of the sentence covered by this vertex.

2.2 Dependency Parsing

Dependency trees provide an alternative, and in some sense simpler, representation of sentence structure. These d-parses can be derived through mechanical transformation from context-free trees. There are several popular transformation in wide use; each provides a different representations of a sentences structure (). In this work we consider transformations that are defined through local transformations known as head rules.

For a binarized CFG, define a collection of head rules as a function $\mathcal{H} : \mathcal{R} \mapsto \{L, R\}$ mapping each CFG rule to its head preference for its left- or right-child. We use the notation $A \rightarrow \beta_1^*\ \beta_2$ and $A \rightarrow \beta_1\ \beta_2^*$ to indicate a left- or right-headed rule respectively.

For each vertex v of a c-parse let the head of the vertex $h(v)$ be defined recursively,

1. If the vertex is leaf x_m, then $h(v) = m$.
2. Otherwise, $h(v)$ matches the head child where $A \rightarrow \beta_1^*\ \beta_2$ or $A \rightarrow \beta_1\ \beta_2^*$ is the rule production at this vertex.

The head rules can be used to map a c-parse to a dependency tree. Define a sentence's dependencies as a sequence $d_1, \ldots d_n$ where word m is a depen-

[1] For notational simplicity we ignore unary rules for this section.

dent of word $d_m\{0, \dots n\}$ and 0 is a special pseudo-root symbol. Let vertex v be the first parent vertex of x_m not headed by m, i.e. $h(v) \neq m$. If this vertex exists, then $d_m = h(v)$, otherwise $d_m = 0$. Figure **??** illustrates the conversion.

These dependencies $d_1, \dots, d_n$ can be viewed as a directed tree with arcs (d_m, m) for all m. This tree differs from the original c-parse since the words are directly connected. However we can relate the two trees through their spans. Define a dependency span $\langle m_{\Leftarrow}, m_{\Rightarrow} \rangle$ as the sequence of words reachable from word m. By construction this span is the same as the span $\langle i, j \rangle$ of the top vertex v with $h(v) = m$.

3 Parsing Dependencies

Now let us flip this setup. In recent years there has been significant progress in developing efficient direct-to-dependency parsers. These parsers can be trained only on dependency annotations and do not require full phrase-structure trees.[2] In many ways this setup is preferable since the parser can be trained on the specific dependencies required for a downstream task. However, there are many applications that require or prefer full c-parses as input, and so c-parsers are still widely-used.

With these applications in mind, we consider the problem of converting a d-parser to a c-parser by converting fixed d-parses into c-parses. Since this problem is more challenging than its inverse, we use a structured prediction setup: learn a function to score possible c-parse conversions, and then generate the highest-scoring c-parse possible.

We start by considering the search problem of finding the best c-parse conversion under a given scoring function.

3.1 Parsing Algorithm

First consider the standard problem of predicting the best c-parse under a CFG with head rules. Assume that we are given a binarized CFG defining a set of valid c-parses $\mathcal{Y}(x)$. The parsing problem is to find the highest-scoring parse in this set, i.e.

$$\hat{y} \leftarrow \arg\max_{y \in \mathcal{Y}(x)} s(y; x)$$

where s is a scoring function that factors over rule productions.

This problem is known as lexicalized context-free parsing. It can be solved using the standard lexicalized CKY parsing algorithm. This algorithm is defined by the productions in Figure **??**. The productions are of the form

$$\frac{(\langle i, k \rangle, m, \beta_1) \quad (\langle k+1, j \rangle, h, \beta_2)}{(\langle i, j \rangle, h, A)}$$

for all rules $A \rightarrow \beta_1\ \beta_2^* \in \mathcal{R}$ and spans $i \leq k < j$. This particular production indicates that rule $A \rightarrow \beta_1\ \beta_2^*$ was applied at a vertex covering $\langle i, j \rangle$ to produce two vertices covering $\langle i, k \rangle$ and $\langle k+1, j \rangle$, and that the new head is index h which is modified by index m.

The highest-scoring parse can found by bottom-up dynamic programming over these productions. The standard lexicalized CKY algorithm requires $O(n^5|\mathcal{R}|)$ time. While simple to implement, this algorithm is intractable to run without heavy pruning or further assumptions on the scoring function.

Now, consider the same setup, but constrained to parses that will be converted to a given d-parse, $d_1, \dots, d_n$. Define this set as $\mathcal{Y}(x, d)$.

$$\arg\max_{y \in \mathcal{Y}(x,d)} s(y; x, d)$$

This new problem has a nice property. For any word x_m the span $\langle i, j \rangle$ of the highest v with $h(v) = n$ is the same as span $\langle m_{\Leftarrow}, m_{\Rightarrow} \rangle$ in the dependency parse. Since we have the dependency parse these spans can be efficiently pre-computed..

This property greatly limits the search space of the parsing problem. Instead of searching over all possible spans $\langle i, j \rangle$ of each modifier, we simple pre-compute the span $\langle m_{\Leftarrow}, m_{\Rightarrow} \rangle$. Figure **??** shows the new algorithm. While the productions are the same as the original algorithm, there are many fewer of them. There is one production for each index and each possible modifier, leading to an algorithm with $O(n^2|\mathcal{R}|)$ running time.

3.2 Pruning

In addition to constraining the number of c-parses possible, the d-parse also provides valuable information about the labeling and structure of the c-parse.

[2] For English these parsers are still often trained on trees converted from c-parses; however for other languages d-parse-only treebanks are common.

Premise:

$$(\langle i, i\rangle, i, A) \quad \forall i \in \{1 \dots n\}, A \in \mathcal{N}$$

Rules:

For $i \leq h \leq k < m \leq j$, and rule $A \rightarrow \beta_1^* \ \beta_2$,

$$\frac{(\langle i, k\rangle, h, \beta_1) \quad (\langle k+1, j\rangle, m, \beta_2)}{(\langle i, j\rangle, h, A)}$$

For $i \leq m \leq k < h \leq j$, rule $A \rightarrow \beta_1 \ \beta_2^*$,

$$\frac{(\langle i, k\rangle, m, \beta_1) \quad (\langle k+1, j\rangle, h, \beta_2)}{(\langle i, j\rangle, h, A)}$$

Goal:

$$(\langle 1, n\rangle, m, r) \text{ for any } m$$

Premise:

$$(\langle i, i\rangle, i, A) \quad \forall i \in \{1 \dots n\}, A \in \mathcal{N}$$

Rules:

For all $i < m, h = d_m$ and rule $A \rightarrow \beta_1^* \ \beta_2$,

$$\frac{(\langle i, m_{\Leftarrow} - 1\rangle, h, \beta_1) \quad (\langle m_{\Leftarrow}, m_{\Rightarrow}\rangle, m, \beta_2)}{(\langle i, m_{\Rightarrow}\rangle, h, A)}$$

For all $m < j, h = d_m$ and rule $A \rightarrow \beta_1 \ \beta_2^*$,

$$\frac{(\langle m_{\Leftarrow}, m_{\Rightarrow}\rangle, m, \beta_1) \quad (\langle m_{\Rightarrow} + 1, j\rangle, h, \beta_2)}{(\langle m_{\Leftarrow}, j\rangle, h, A)}$$

Goal:

$$(\langle 1, n\rangle, m, r) \text{ for any } m \text{ s.t. } d_m = 0$$

Figure 3: The two algorithms written as deductive parsers. Starting from the *premise*, any valid application of *rules* that leads to a *goal* is a valid parse. (a) Lexicalized CKY algorithm for CFG parsing with head rules. For this algorithm there are $O(n^5|\mathcal{R}|)$ rules where n is the length of the sentence. (b) The constrained CKY parsing algorithm for $\mathcal{Y}(x, d)$. The algorithm is nearly identical except that many of the free indices are now fixed to the dependency parse. Finding the optimal parse is now $O(n^2|\mathcal{R}|)$.

We can use this information to provide further pruning of the search space. We experimented with two heuristic pruning methods based on this information

- We observe that in training the part-of-speech of the head word x_h greatly limits the possible rules $A \rightarrow \beta_1 \ \beta_2$ available. To exploit this property we build tables $\mathcal{R}_t$ for each part-of-speech tag t and limit the search to rules seen for the current head tag. This reduces the effect of $|\mathcal{R}|$ at runtime.

- A word x_h with L dependents to its left and R dependents to its right may at any point combine with a left or right dependent. There are therefore $L \times R$ ($O(n^2)$) possible orderings of these dependents.

  However, we can often predict with very high accuracy which side will come next. To make this prediction, we estimate a distribution

  $$p(\text{side}|\text{tag}(x_h), \text{tag}(x_l), \text{tag}(x_r))$$

  where x_l is the next left dependent, x_r is the next right dependent, and $\text{tag}()$ is the part-of-speech tag of the word. This reduces the effect of the n^2 at runtime.

Model	Sym	Comp.	Speed	Oracle
LCFG(< 20)	$\mathcal{Y}(x)$	$O(n^5\|\mathcal{R}\|)$	0.25	100.0
LCFG(dep)	$\mathcal{Y}(x, d)$	$O(n^2\|\mathcal{R}\|)$	63.2	92.8
LCFG(prune)	-	$O(n^2\|\mathcal{R}\|)$	173.5	92.7

Table 1: Comparison of three parsing setups: LCFG(< 20) is the standard full lexicalized grammar limited to sentence of length less than 20 words, LCFG(dep) is limited to the dependency skeleton, and LCFG(prune) is the pruning described in Section 3.2. *Oracle* is the oracle f-score on the development data (described in Section 7.3). *Speed* is the efficiency of the parser on development data in sentences per second.

Table **??** shows a comparison of these pruning methods. In addition to speed, an important question whether it is even possible to recover c-parses under these constraints. These experiments show that there is a small drop from pruning, but that there is a sizable drop in oracle accuracy with the d-parse constraints. Still this upper-bound on accuracy is high enough to make it possible to still recover high-accuracy parses. We return to this discussion in Section **??**.

3.3 Binarization

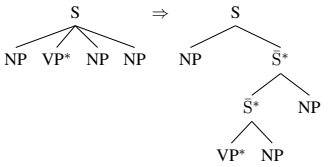
To this point, we have assumed that the CFG and head rules are binary; however the standard treebank grammars have arbitrarily large rules. In order to apply the algorithm, we need to binarize the grammar and preserve the head relations.

Consider a non-binarized rule of the form $A \rightarrow \beta_1 \dots \beta_m$ with head rule $\mathcal{H}(A \rightarrow \beta_1 \dots \beta_m) = k$. Relative to the head β_k the rule has left-side $\beta_1 \dots \beta_{k-1}$ and right-side $\beta_{k+1} \dots \beta_m$.

We replace this rule with binary rules that consume each side independently as a simple chain. The transformation introduces new rules[3]

- $\bar{A} \rightarrow \beta_i \ \bar{A}^*$ for $i \in \{2, \dots, k\}$
- $\bar{A} \rightarrow \bar{A}^* \ \beta_i$ for $i \in \{k, \dots, m\}$
- $A \rightarrow \beta_1 \ \bar{A}^*$

As an example consider the transformation of a rule with four children:



Note that each rule contains at most two nonterminals from the original grammar, so the size of the new binarized rule set is bounded by $O(\mathcal{N}^2)$. The small size of the grammar further helps the speed of the algorithm.

We experimented with binarization that use horizontal and vertical markovization to include additional context of the tree (), but we found that this increased the size of the grammar without leading to improvements in accuracy.

[it's quite important to adding the pruning algorithm stuff... they are very interesting i think –lpk]

4 Structured Prediction

While the parser only requires a d-parse at prediction time, it uses annotated c-parses to learn the transformation. Define this scoring function s as

$$s(y; x, d, \theta) = \theta^\top f(x, d, y)$$

where θ is a parameter vector and $f(x, d, y)$ is a feature function that maps parse productions to sparse feature vectors.

To learn the parameter for scoring possible c-parses we use a standard structured prediction setup. In this section, we describe the features of the model and then the learning algorithm.

4.1 Features

In theory, the scoring function for this system could be directly adapted from existing c-parsers. However, the existing structure of the dependency parse limits the number of decisions that need to be made, and allows for a smaller set of features.

We model our features after two bare-bones parsing systems. The first set is the basic arc-factored features used by McDonald (2006). These features include combinations of:

- rule and top non-terminal
- modifier word and part-of-speech
- head word and part-of-speech

The second set of features is modeled after the span features described in the X-Bar style parser of Hall et al. (2014). These include conjunction of the rule with:

- first and last word of current span.
- preceding and following word of current span
- adjacent words at split of current span
- length of the span

The full feature set is shown in Figure **??**.

4.2 Training

The parameters θ are estimated using a structured SVM setup. Given a set of gold-annotated parse examples, $(x^1, y^1), \dots, (x^D, y^D)$, and paired dependency structures $d^1 \dots d^D$ induced from the head rules, we estimate the parameters to minimize the regularized empirical risk

$$\min_\theta \sum_{i=1}^{D} \max\{0, \ell(x^i, d^i, y^i, \theta)\} + \frac{\lambda}{2}||\theta||_1$$

[3] These rules are slightly modified when $k = 1$

For a production $\dfrac{(\langle i,k\rangle, m, \beta_1) \quad (\langle k+1,j\rangle, h, \beta_2)}{(\langle i,j\rangle, h, A)}$

Nonterm Features
(A, β_1)
(A, β_2)
$(A, \beta_1, \text{tag}(m))$
$(A, \beta_2, \text{tag}(h))$

Span Features
(rule, x_i)
(rule, x_j)
(rule, x_{i-1})
(rule, x_{j+1})
(rule, x_k)
(rule, x_{k+1})
$(\text{rule}, \text{bin}(j-i))$

Rule Features
(rule)
$(\text{rule}, x_h, \text{tag}(m))$
$(\text{rule}, \text{tag}(h), x_m)$
$(\text{rule}, \text{tag}(h), \text{tag}(m))$
(rule, x_h)
$(\text{rule}, \text{tag}(h))$
(rule, x_m)
$(\text{rule}, \text{tag}(m))$

Figure 4: The feature templates used in the function $f(x, d, y)$. For the span features, the symbol rule is expanded into both $A \rightarrow B\ C$ and A. The function $\text{tag}(i)$ gives the part-of-speech tag of word x_i. The function $\text{bin}(i)$ partitions a span length into one of 10 bins.

where we define ℓ as

$$\ell(x, d, y, \theta) = s(y) + \max_{y' \in \mathcal{Y}(x,d)} \left(s(y') + \Delta(y, y')\right)$$

and where Δ is a problem specific cost-function. In experiments, we use a hamming loss $\Delta(y, y') = ||y - y'||$ where y is an indicator of rule productions.

The objective is optimized using Adagrad (). The gradient calculation requires computing a loss-augmented argmax for each training example which is done using the algorithm of Figure **??**.

5 Related Work

The problem of converting dependency to phrase-structured trees has been studied previously from the perspective of building multi-representational treebanks. Xia and Palmer (2001) and Xia et al. (2009) develop a rule-based system for the converting human-annotated dependency parses. Our work differs in that we learn a data-driven structured prediction model that is also able to handle automatically predicted input.**[mention the table in the exp section where the comparison number show our approach better here? –lpk]**

There has been successful work combining dependency and phrase-structure parsing. Carreras et al. (2008) build a high-accuracy parser that uses a dependency parsing model both for pruning and within a richer lexicalized parser. Similarly Rush et al. (2010) use dual decomposition to combine a dependency parser with a simple phrase-structure model. We take this approach a step further by fixing the dependency structure entirely before parsing.**[only fixing the dependency structure entirely before parsing sounds more like a step backward than a step further... i am not sure if this is a good way to say that... i think there is a important difference here, which is, when combining the phrase-structure and dep parsing, when ppl do this, the binarization does not imply dep arcs, so even you can push info from dep to phrase structure parsing, you can't directly pruning so much like we do here...right? –lpk]**

Finally there have also been several papers that use ideas from dependency parsing to simplify and speed up phrase-structure prediction. Zhu et al. (2013) build a high-accuracy phrase-structure parser using a transition-based system. Hall et al. (2014) use a stripped down parser based on a simple X-bar grammar and a small set of lexicalized features.

6 Methods

We ran a series of experiments to assess the accuracy, efficiency, and applicability of the parser to several tasks. These experiments use the following setup.

For English experiments we use the standard Penn Treebank (PTB) experimental setup (Marcus et al., 1993). Training is done on section 2-21, development on section 22, and test of section 23.

For Chinese experiments, we use version 5.1 of the Penn Chinese Treebank 5.1 (CTB) (Xue et al., 2005). We followed previous work and used 001-270 and 440-1151 for training, articles 301-325 as development, and articles 271-300 as test.

Part-of-speech tagging is done using TurboTagger (Martins et al., 2013). Unless otherwise noted, the input d-parsing is done using TurboParser (Martins et al., 2013). Prior to training the d-parser, the train sections are automatically tagged using 10-fold jackknifing. The d-parser is trained to produce Collins head rules (Collins, 2003).

For our model the "gold" d-parses $d^1, \ldots, d^D$ used at training come from the gold c-parse con-

verted with the Collins head rules[4]. Since the trained model is independent of the d-parser, for all experiments with other d-parsers we used the same model.

Evaluation for phrase-structure parses is performed using the `evalb`[5] script using the standard setup. We report F1-Score as well as recall and precision. For dependency parsing we use unlabeled accuracy score (UAS).

We implemented the grammar binarization, head rules, and pruning tables in Python, and the parser, features, and training in C++. The core run-time decoding algorithm is self contained and requires less than 400 lines of code. Both are publicly available.[6] Experiments are performed on a Lenovo ThinkCentre desktop computer with 32GB of memory and Core i7-3770 3.4GHz 8M cache CPU.

7 Experiments

We ran experiments to assess the accuracy of the method, its run-time efficiency, the amount of phrase-structure data required, and the effect of dependency accuracy.**[if space allowed, probably add the label extension as a way to get higher acc in the first half of the experiment section. –lpk]**

7.1 Parsing Accuracy

Our first experiments, shown in Table **??**, examine the accuracy of the phrase-structure trees produced by the parser. For these experiments, we use TurboParser (Martins et al., 2013) to predict downstream dependencies.

...

7.2 Efficiency

Our next set of experiments consider the efficiency of the model. For these experiments we consider both the full and pruned version of the parser using the pruning described in section 3.2. Table **??** shows that in practice the parser is quite fast, averaging around % tokens per second at high accuracy.

We also consider the end-to-end speed of the parser when combined with different downstream dependencies. We look at

Finally we consider the practical run-time of the parser on sentences of different length. Figure **??** shows the graph.

	PTB		
Model	22 FScore	22 UAS	23 FScore
Charniak			89.5
Petrov[07]		92.66	90.1
Carreras[08]			91.1
Zhu[13]			90.4
CJ		93.92	
Stanford[]		88.88	
StanfordRNN		92.23	
PARPAR	91.04	93.59	
	CTB		
model	dev fscore		test fscore
Bikel			80.6
Petrov[07]			83.3
Carreras[08]			
Zhu[13]			83.2
Stanford[]			
CJ			82.3
PARPAR			

Table 2: Accuracy results on the Penn Treebank and Chinese Treebank datasets. Comparisons are to state-of-the-art non-reranking phrase-structure parsers including: Petrov[07] (Petrov et al., 2006), Carraras[08] (Carreras et al., 2008), Zhu[13] (Zhu et al., 2013), Charniak[00] (Charniak, 2000), and Stanford[] ().

7.3 Analysis

To gauge the upper bound of the accuracy of this system we consider an oracle version of the parser. For a gold parse y and predicted dependencies $\hat{d}$, define the oracle parse y' as

$$y' = \arg\min_{y' \in \mathcal{Y}(x,\hat{d})} \Delta(y, y')$$

Table **??** shows the oracle accuracy of TurboParser and several other commonly used dependency parsers.

We also consider the mistakes that are made by the parser compared to the mistakes made. For each of the bracketing errors made by the parser, we can classify it as a bracketing mistake, a dependency mistake or neither.

[4] We experimented with using jackknifed dependency parses d' at training time with oracle tree structures, i.e. $\arg\min_{y' \in \mathcal{Y}(x,d')} \Delta(y, y)$, but found that this did not improve performance.

[5] http://nlp.cs.nyu.edu/evalb/

[6] Withheld for review

Model	Oracle	FScore	Speed
TURBOPARSER	92.90	91.04	
MALTPARSER			20
ZPAR			
MIT			

Table 3: Comparison of the effect of downstream dependency prediction. Experiments are run on the development section with different input dependencies. *Oracle* is the oracle F1 on the development data. *Speed* is the efficiency of the parser in sentences per second. Inputs include TurboParser (Martins et al., 2013), MaltParser (Nivre et al., 2006), and MIT ().

Class						
Dep	Span	Split	Total	Pre	Rec	F1
+	+	+				
+	+	-				
-	+	+				
-	-	-				

7.4 Conversion

Previous work on this problem has looked at converting dependency trees to phrase-structure trees using linguistic rules (Xia and Palmer, 2001; Xia et al., 2009). This work is targeted towards the development of treebanks, particularly converting dependency treebanks to phrase-structure treebanks. For this application, it is useful to convert gold trees as opposed to predicted trees.

To compare to this work, we train our parser with gold tags and run on gold dependency trees in development. Table 4 give the results for this task.

	Dev		
Model	Prec	Rec	F1
Xia[09]	88.1	90.7	89.4
PARPAR(Sec19)	95.9	95.9	95.9
PARPAR	97.5	97.8	97.7

Table 4: Comparison with the rule-based system of Xia et al. (2009). Results are from PTB development section 22 using gold tags and gold dependencies. Xia[09] report results from training on only on Section 19, but include a note that further data had little effect. For comparison we report result on complete training as well as just Sec. 19.

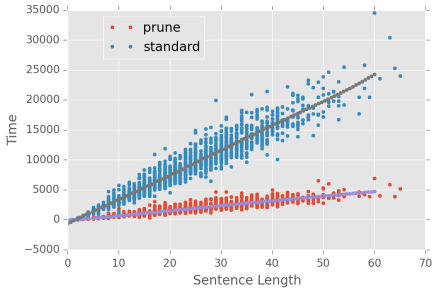


Table 5: Experiments of parsing speed. (a) The speed of the parser on its own and with pruning. (b) The end-to-end speed of the parser when combined with different dependency parsers.

8 Conclusion

With recent advances in statistical dependency parsing, state-of-the-art parsers have reached the comparable dependency accuracy as the best phrase structure parsers. However, these parser cannot be directly used in applications that require phrase-structure prediction. In this work we have described a simple parsing algorithm and structured prediction system for this comparison, and show that it can produce phrase-structure parses at comparable accuracy to state-of-the-art system.

One question for future work is whether these results are language dependent, or whether these transformation can be projected across languages. If this were possible, we could use a system of this form to learn phrase structure parsers on languages with only dependency annotations.

References

Xavier Carreras, Michael Collins, and Terry Koo. 2008. Tag, dynamic programming, and the perceptron for efficient, feature-rich parsing. In *Proceedings of the Twelfth Conference on Computational Natural Language Learning*, pages 9–16. Association for Computational Linguistics.

Eugene Charniak. 2000. A maximum-entropy-inspired parser. In *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*, pages 132–139. Association for Computational Linguistics.

Michael Collins, Lance Ramshaw, Jan Hajič, and Christoph Tillmann. 1999. A statistical parser for czech. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on*

Computational Linguistics, pages 505–512. Association for Computational Linguistics.

Michael Collins. 2003. Head-driven statistical models for natural language parsing. *Computational linguistics*, 29(4):589–637.

David Hall, Greg Durrett, and Dan Klein. 2014. Less grammar, more features. In *ACL*.

Lingpeng Kong and Noah A Smith. 2014. An empirical comparison of parsing methods for stanford dependencies. *arXiv preprint arXiv:1404.4314*.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330.

André FT Martins, Miguel Almeida, and Noah A Smith. 2013. Turning on the turbo: Fast third-order non-projective turbo parsers. In *ACL (2)*, pages 617–622.

Ryan McDonald. 2006. *Discriminative learning and spanning tree algorithms for dependency parsing*. Ph.D. thesis, University of Pennsylvania.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2006. Maltparser: A data-driven parser-generator for dependency parsing. In *Proceedings of LREC*, volume 6, pages 2216–2219.

Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. 2006. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*, pages 433–440. Association for Computational Linguistics.

Alexander M Rush, David Sontag, Michael Collins, and Tommi Jaakkola. 2010. On dual decomposition and linear programming relaxations for natural language processing. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1–11. Association for Computational Linguistics.

Fei Xia and Martha Palmer. 2001. Converting dependency structures to phrase structures. In *Proceedings of the first international conference on Human language technology research*, pages 1–5. Association for Computational Linguistics.

Fei Xia, Owen Rambow, Rajesh Bhatt, Martha Palmer, and Dipti Misra Sharma. 2009. Towards a multi-representational treebank. In *The 7th International Workshop on Treebanks and Linguistic Theories. Groningen, Netherlands*.

Naiwen Xue, Fei Xia, Fu-Dong Chiou, and Martha Palmer. 2005. The penn chinese treebank: Phrase structure annotation of a large corpus. *Natural language engineering*, 11(02):207–238.

Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and accurate shift-reduce constituent parsing. In *ACL (1)*, pages 434–443.