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

Before describing our method, We begin by developing notation for a c-parsing and the conversion process to d-parses.

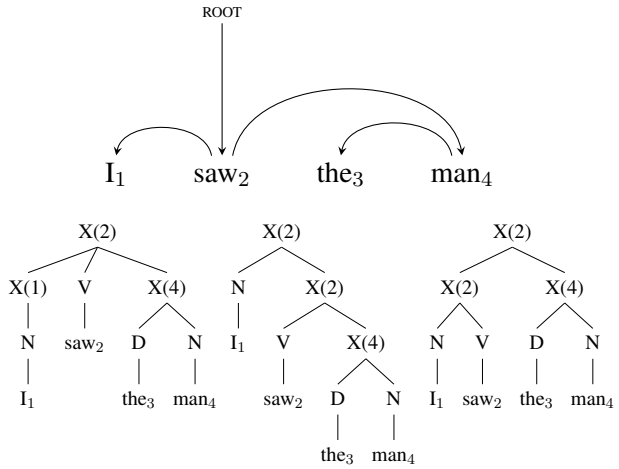


Figure 1: It is standard to transform c-parses to d-parses using deterministic head rules. The figure, adapted from (Collins et al., 1999), shows several X-bar trees that all produce the same dependency structure. The parentheses $X(h)$ indicate the head h of each internal vertex.

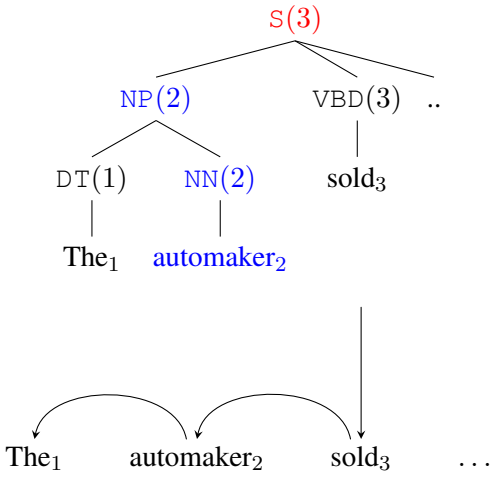


Figure 2: Figure illustrating an LCFG parse. The parse is an ordered tree with fringe $x_1, \ldots, x_n$. Each vertex is annotated with a span, head, and syntactic tag. The blue vertices represent the 3-vertex spine v_1, v_2, v_3 of the word $\texttt{automaker}_2$. The root vertex is v_4, which implies that $\texttt{automaker}_2$ modifies $\texttt{sold}_3$ in the induced dependency graph.

2.1 CFG Parsing

The phrase-structure trees annotated in the Penn Treebank are derivation trees from a context-free grammar. A c-parser is any system that directly predicts the best context-free parse for a sentence.

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.

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

2.2 Dependency Parsing

Dependency trees provide an alternative, and in some sense simpler, representation of sentence structure. These trees were originally derived through mechanical transformation from context-free trees, known as head rules. There are several popular head rules in use that provide different representations of the sentences structures (). In this work we consider head rules that are local to the context-free rules.

For a binarized CFG, define the head rules as a function $\mathcal{H} : \mathcal{R} \mapsto \{L, R\}$ mapping each CFG rule to its preferred, left or right head 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. The head rules can be used to map a c-parse to a dependency tree through the following transformation.

First, for each vertex v of the tree let the head of the vertex $h(v)$ be defined recursively,

1. If the vertex is leaf x_i, then $h(v) = i$.
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.

Next, for each word x_i let the *spine* of the word be the longest chain of connected vertices with x_i

as head, i.e. $v_1, \ldots, v_p$ where $h(v_j) = i$ for $j \in \{1, \ldots, p\}$. An example spine is shown in Figure **??**.

Finally, define the d-parse $d_1 \ldots d_n$ for a sentence through the spines. If it exists, we define v_{p+1} be the parent of v_p, and let x_i be a dependent of the head of this vertex, $d_i = h(v_{p+1})$. Otherwise we say x_i is the root word of the sentence, and let $d_i = 0$.

These dependency relations can be viewed as a directed tree with arcs (d_i, i) for all i. 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 i. By construction this span is the same as the span $\langle i, j \rangle$ of the top vertex v_p of the spine of x_i.

[I think here maybe instead of directly giving the dependneny tree notation like this, it would be easier for ppl to understand if we fisrt when we have the phrase-structure tree, through head rules (popluar head rules including cite collins, cite ym and cite stanford. They basically defined in all the children a NT has, which one he loves the most. –lpk]

3 Parsing Dependencies

The aim of this work is to predict the best c-parse from a given d-parse. Since this problem is ill-posed, we will learn a function to score possible c-parses and generate the highest-scoring parse. In this section we assume we have this function and discuss the prediction problem. We return to training in the next section.

3.1 Constrained Parsing Algorithm

Our prediction algorithm will be a simple extension of the standard lexicalized CKY parsing algorithm.

Assume that we are given a binarized LCFG 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.

We further assume that scoring function factors over rule productions, and so the highest-scoring parse can be found using the lexicalized CKY algorithm. This algorithm is defined deductively 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, and the running time is linear in the number of production. The standard lexicalized CKY algorithm requires $O(n^5|\mathcal{R}|)$ time, which is intractable to run without heavy pruning.

Fortunately, the parsing problem becomes much simpler if we are constrained to a given d-parse, $d_1, \ldots, d_n$. Define the set $\mathcal{Y}(x, d)$ as all valid LCFG parses that map to this dependency parse (as defined in the previous section). Our aim is to find

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

This new problem has a nice property. For any word x_m with spine $v_1, \ldots, v_p$ the LCFG span $\langle i, j \rangle$ of v_p is equal to the dependency span $\langle m_{\Leftarrow}, m_{\Rightarrow} \rangle$ of x_m. These dependency spans can be efficiently computed directly from the dependency parse d.

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 can precompute $\langle m_{\Leftarrow}, m_{\Rightarrow} \rangle$. 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.

Finally in addition to constraining the topology of c-parses, we also experiment with pruning to limit the space of rules. We observe that in training the part-of-speech of the head word x_h greatly limits the possible rules $A \rightarrow \beta_1\ \beta_2$. 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.

An important concern is whether it is even possible to recover c-parses under these constraints. To answer this question we ran oracle experiments

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 LCFG parsing. 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 $O(n^2|\mathcal{R}|)$.[d_m **is a little confusing when i first saw that, we can mention that in the caption. And the premise of the second algorithm is on the bottom of the first algorithm, probably we want to move it to the right. –lpk]**

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 **??**. *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.

to find the best c-parse possible under each algorithm. Table **??** shows a comparison among full LCFG parsing, dependency limited parsing, and dependency-limit pruned parsing. These experiments show that there is a sizable drop in oracle accuracy with the d-parse constraints, but that the upper-bound on accuracy is still high enough to make this task interesting.

3.2 Binarization

[This section is wrong, start over –smr]

To this point we have assumed the grammar is binarized, While the algorithm itself is not dependent on the binarization used **[but one important thing is, we must binarize in a way preserve the head each time. (i.e. one binarization rule implies one arc.) –lpk]**, this choice affects the run-time of the algorithm, through $\mathcal{R}$, as well as the structure of the scoring function.

For simplicity, we consider binarizing rule $\langle A \rightarrow \beta_1 \dots \beta_m, k\rangle$ with $m > 2$. Relative to the head β_k the rule has left-side $\beta_1 \dots \beta_{k-1}$ and right-side $\beta_{k+1} \dots \beta_m$.

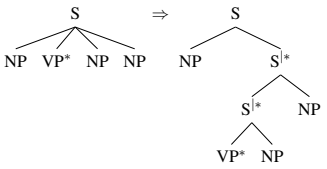
We replace this rule with binary rules that consume each side independently as a first-order Markov chain (horizontal Markovization). The main transformation is to introduce rules

- $A^{|} \rightarrow A^{|*} \ \beta_i$ for $k < i < m$
- $A^{|} \rightarrow \beta_i \ A^{|*}$ for $1 < i < k$

Additionally we introduce several additional rules to handle the boundary cases of starting a new rule, finishing the right side, and completing a rule.

(These rules are slightly modified when $k \leq 2$ or $k = m$).

For example the transformation of a common rule looks like



[I change the rule based on current system. I think I want to add the lesson we learnt about binarization here. That is, the NT should remember as less as possible. This is saying the the fully lexiconlized model has the ability to figure out what rule to use as well as the topological structure. The more information only adds to the runtime and make estimation more sparse. –lpk]

Each rule contains at most 3 original nonterminals so the size of the new binarized rule set is bounded by $O(\mathcal{N}^3)$.**[it's quite important to adding the pruning algorithm stuff... they are very interesting i think – lpk]**

4 Structured Prediction

To learn the scoring function for the transformation from d-parses to c-parses, we use a standard structured prediction setup. Define the 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. We first discuss the features used and then estimating the parameters of the model.

4.1 Features

We implemented a small set of standard dependency and phrase-structure features.

For the dependency style features, we replicated the basic arc-factored features used by McDonald (2006). These include combinations of:

- nonterminal combinations
- rule and top nonterminal
- modifier word and part-of-speech

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 $f(x, d, y)$. 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.

- head word word and part-of-speech

Additionally we included the span features described for 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 standard structured SVM training. We assume that we are given a set of gold-annotated parse examples: $(x^1, y^1), \ldots, (x^D, y^D)$. We also define $d^1 \ldots d^D$ as the dependency structures induced from $y^1 \ldots y^D$. We select parameters to minimize the regularized empirical risk

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

where we define ℓ as

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

where Δ is a problem specific cost-function that we assume is linear in either arguments. In experiments, we use a hamming loss $\Delta(y, \bar{y}) = ||y - \bar{y}||$ 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 Setup

6.1 Data and Methods

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). Prior to training, the train sections are automatically tagged using 10-fold jackknifing. At training time, the gold dependency structures are computed using the Collins head rules (Collins, 2003).[2]

Evaluation for phrase-structure parses is performed using the `evalb`[3] script using the standard setup. We report F1-Score as well as recall and precision. For dependency parsing using 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.[4] 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]**

[2] 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.

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

[4] Withheld for review

	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[] ().

7.1 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 **??**. 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.

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
+	+	+				
+	+	-				
-	+	+				
-	-	-				

7.3 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.

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

Model	Dev Prec	Dev Rec	Dev 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.

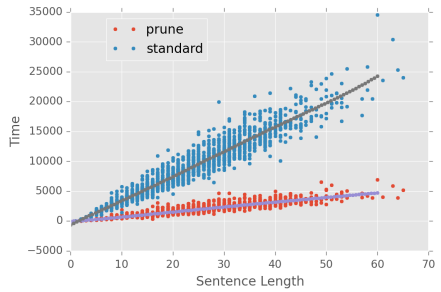


Table 5: Experiments of parsing speed. (a) The speed of the parser on its own and with pruning. (b) The end-to-end speed of the parser when combined with different dependency parsers.

gold tags and run on gold dependency trees in development. Table 4 give the results for this task.

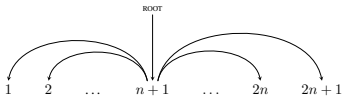
8 Conclusion

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.

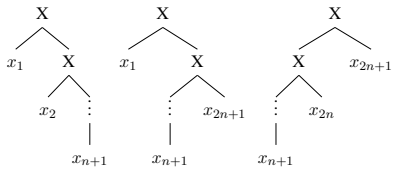
One question for future work is whether these results are language dependent, or whether these transformation can be projected across languages. If this were possible, we could use a system of this form to learn phrase structure parsers on languages with only dependency annotations.

A Proof of PS Size

Consider the LCFG grammar with two rules $A = X \rightarrow X^* \; X$ and $B = X \rightarrow X \; X^*$ and a sentence $x_1, \ldots, x_{2n+1}$. Let the dependency parse be defined as $d_{n+1} = 0$ and $d_i = n + 1$ for all $i \neq n + 1$, i.e.



Since all rules have $h = x_n$ as head, a parse is a chain of $2n$ rules with each rule in $\{A, B\}$, e.g. the following are BB..., BA..., AA...



Since there must be equal As and Bs and all orders are possible, there are $\binom{2n}{n}$ valid parses and $|\mathcal{Y}(x, d)|$ is $O(2^n)$.

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.