Parsing Dependencies

Abstract

Motivated by the increase in accuracy of statistical dependency parsers, we consider the problem of decoding phrase-structure parses directly from predicted dependency trees. Unlike past rule-based approaches, we treat this as a structured prediction problem using a specialized context-free parser. Since our parser makes use of the predicted dependency structure it is asymptotically faster and much simpler than a standard lexicalized parser. However, despite its simplicity, it still yields high-accuracy phrase-structure parses on experiments in both English and Chinese.

1 Introduction

There are two main grammatical frameworks used for statistical syntactic prediction: phrase-structure parsers and dependency parsers (). The two offer a trade-off: phrase-structure parsing is very accurate and provides a full context-free syntactic representation; dependency parsing is often faster, but still predicts much of the useful syntactic structure.

We can quantify this difference by looking at the dependency prediction accuracy of phrase-structure parsers. Table 1 shows this comparison across several widely-used parsers. Note that the best models are phrase-structure parsers, but that recent advances in dependency parsing have led to comparably high-accuracy dependency parsers.

The natural reverse question is how well do these dependency parsers perform at phrase-structure prediction? Is it reasonable to use dependency parsers for tasks that require phrase-structure trees as input?

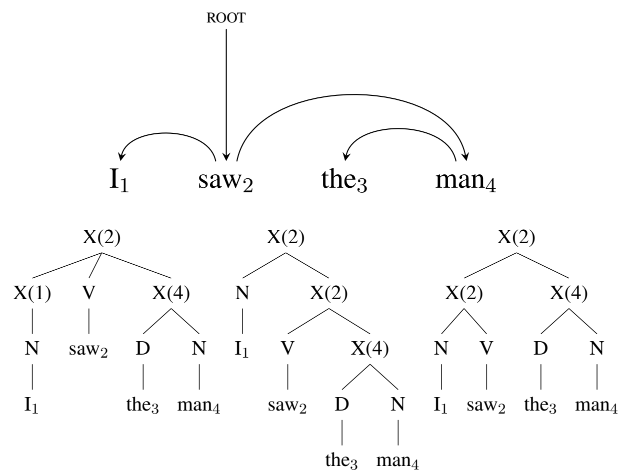


Figure 1: While a phrase-structure parse determines a unique dependency parse, the inverse problem is non-deterministic. The figure, adapted from (Collins et al., 1999), shows several X-bar trees that all produce the same dependency structure. The parentheses $X(h)$ indicate the head h of each internal vertex.

Unfortunately, as illustrated in Figure **??**, recovering the best phrase-structure tree is a non-trivial problem. Given a predicted dependency tree, there is a large space of possible output phrase-structure trees, each tree may have many possible labelings, and there are likely errors in the input dependency tree.

In this work, we pose phrase-structure recovery as a structured prediction problem, and train a full lexicalized phrase-structure parser to predict the syntactic tree for a given sentence. Crucially, though, we limit the search space of the parser to the search space that produces a given dependency tree. We show that,

- The constrained parser is asymptotically faster than standard phrase-structure parser for lexicalized context-free grammar. While the standard algorithm is $O(n^5|\mathcal{R}|)$, the constrained dependency parser has worst case complexity $O(n^2|\mathcal{R}|)$.

- In practice using simple pruning the parser is linear time in the length of the sentence and as efficient as the fastest high-accuracy dependency parsers. %

- Despite being constrained to hard downstream dependency decisions, the parser is comparably accurate to non-reranked phrase-structure parsers. %

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.

There has been successful work using dependency parsing structure to guide phrase-structure parsers. 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.

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.

2 Background

We begin by developing notation for a lexicalized context-free formalism and for dependency parsing. The notation aims to highlight the similarity between the two formalisms.

Type	Model	UAS
Phrase Structure	Petrov[06]	92.66
	Stanford PCFG	88.88
	CJ Reranking	93.92
	Stanford RNN	92.23
Dependency	TurboParser	93.59

Table 1: Dependency accuracy for several several widely used phrase-structure and dependency parsers. Score are reported as the unlabeled accuracy score (UAS) of dependencies on PTB Section 22. Conversion are performed using the Collins head rules (Collins, 2003). Note that the best scoring is a reranking phrase-structure parser, but that state-of-the-art dependency parsers are comparable with the best parsers.

2.1 Lexicalized CFG Parsing

A lexicalized context-free grammar (LCFG) is a context-free grammar where each vertex in a parse has a unique lexical head. Define an binarized[1] LCFG 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 lexicalized rule productions either of the form $A \rightarrow \beta_1^* \ \beta_2$ or $A \rightarrow \beta_1 \ \beta_2^*$ consisting of a parent nonterminal $A \in \mathcal{N}$, a sequence of children $\beta_i \in \mathcal{N} \cup \mathcal{T}$ for $i \in \{1, 2\}$, and a distinguished head child annotated with $*$. The head child comes from the head rules associated with the grammar.

- r; a distinguished root symbol $r \in \mathcal{N}$.

Given an input sentence $x_1, \ldots, x_n$ of terminal symbols from $\mathcal{T}$, define $\mathcal{Y}(x)$ as the set of valid lexicalized parses for the sentence. 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 an LCFG parse $y \in \mathcal{Y}(x)$, we further associate a triple $v = (\langle i, j \rangle, h, A)$ with each vertex in the tree, where

- $\langle i, j \rangle$; the *span* of the vertex, i.e. the contiguous sequence $\{x_i, \ldots, x_j\}$ of the sentence covered by the vertex.

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

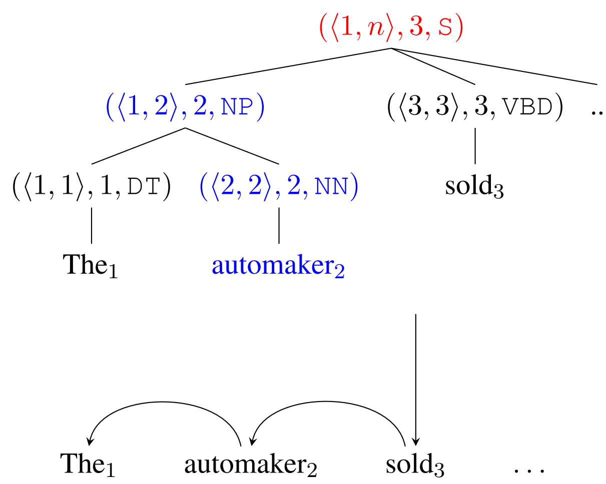


Figure 2: Figure illustrating an LCFG parse. The parse is an ordered tree with fringe $x_1, \ldots, x_n$. Each vertex is annotated with a span, head, and syntactic tag. The blue vertices represent the 3-vertex spine v_1, v_2, v_3 of the word $\texttt{automaker}_2$. The root vertex is v_4, which implies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

- $h(v) \in \{1, \ldots, n\}$; index indicating that x_h is the *head* of the vertex, defined recursively by the following rules:
  1. If the vertex is leaf x_i, then $h = i$.
  2. Otherwise, h 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.
- $A \in \mathcal{T} \cup \mathcal{N}$; the terminal or nonterminal symbol of the vertex.

Note that all but one word x_i has an ancestor vertex v where $h(v) \neq i$. Define the *spine* of word x_i to be the longest of chain connected vertices $v_1, \ldots, v_p$ where $h(v_j) = i$ for $j \in \{1, \ldots, p\}$. Also if it exists, let vertex v_{p+1} be the parent of vertex v_p, where $h(v_{p+1}) \neq i$. The full notation is illustrated in Figure 2.

2.2 Dependency Parsing

Dependency trees provide an alternative, and in some sense simpler, representation of grammatical structure.

For sentence $x_1, \ldots, x_n$, define a dependency parse d as a sequence $d_1, \ldots, d_n$ where for all i, $d_i \in \{0, \ldots, n\}$. These dependency relations can be seen as arcs (d_i, i) in a directed graph over the sentence, where w_0 is a special pseudo-root vertex. A dependency parse is valid if the corresponding directed graph is a directed tree rooted at vertex 0. Figure 2 contains an example of a dependency tree.

For a valid dependency tree, define the *span* of any word x_m as the set of indices reachable from vertex m in the directed tree. A dependency parse is *projective* if the descendants of every word in the tree form a contiguous span of the original sentence (). We use the notation $m_\Leftarrow$ and $m_\Rightarrow$ to represent the left- and right-boundaries of this span.

Any lexicalized context-free parse can be converted to a unique projective dependency tree. For an input symbol x_m with spine $v_1, \ldots, v_p$,

1. If v_p is the root of the tree, then $d_m = 0$.
2. Otherwise let v_{p+1} be the parent vertex of v_p and $d_m = h(v_{p+1})$. The span $\langle i, j \rangle$ of v_p in the lexicalized parse is equivalent to $\langle m_\Leftarrow, m_\Rightarrow \rangle$ in the induced dependency parse.

However the conversion from dependency tree to phrase-structure tree not unique, and in fact, it can be shown that in the worst-case there are an exponential number of possible unlabeled phrase-structure trees that induce the same dependency parse (proof given in Appendix A).

3 Parsing Dependencies

Since the inverse problem is ill-posed, our goal will be to learn a scoring function to help predict a phrase-structure tree for any dependency parse. In this section we assume this function is given and describe the prediction problem. In the next section we consider the learning problem.

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, define the set of valid parses for a sentence as $\mathcal{Y}(x)$. 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.

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: (a) Standard CKY algorithm for LCFG parsing stated as inductive rules. Starting from the *premise*, any valid application of *rules* that leads to a *goal* is a valid parse. Finding the optimal parse with dynamic programming is linear in the number of 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 $O(n^2|\mathcal{R}|)$.

If the scoring function factors over rule productions, then the highest-scoring parse can be found using the lexicalized CKY algorithm. This algorithm is defined as a collection of inductive rules shown in Figure **??**. The inductives rules 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 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 parse can found by bottom-up dynamic programming (CKY) over this set. The running time is linear in the number of rules. This algorithm requires $O(n^5|\mathcal{R}|)$ time, which is intractable to run without heavy pruning.

However, in this work, we are interested in a constrained variant of this problem. We assume that we additionally have access to a projective dependency parse for the sentence, $d_1, \dots, d_n$. Define the set $\mathcal{Y}(x, d)$ as all valid LCFG parses that match this dependency parse. For all inductive rules with head h and modifier m, there must be a dependency $d_m = h$. 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 $v_1, \dots, v_p$ 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 set inductive rules. While these rules are very similar to the original, the quantifiers are much more constrained. Given that there are n dependency links and n indices, the new algorithm has $O(n^2|\mathcal{R}|)$ running time.

3.2 Extension: Labels

Finish this section

Model	Sym	Comp.	Speed	Oracle
LCFG($<$ 20)	$\mathcal{Y}(x)$	$O(n^5\lvert\mathcal{N}\rvert^3)$	0.25	100.0
LCFG(dep)	$\mathcal{Y}(x,d)$	$O(n^2\lvert\mathcal{N}\rvert^3)$	63.2	92.8
LCFG(prune)	-	$O(n^2\lvert\mathcal{N}\rvert^3)$	173.5	92.7

Table 2: 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 3.3. *Oracle* is the oracle f-score on the development data (described in Section 6.3). *Speed* is the efficiency of the parser on development data in sentences per second.

Standard dependency parsers also predict labels from a set $\mathcal{L}$ on each dependency link. In a labeled dependency parser a would be of the form (i, j, l).

This label information can be used to encode further information about the parse structure. For instance if we use the label set $\mathcal{L} = \mathcal{N} \times \mathcal{N} \times \mathcal{N}$, encoding the binary rule decisions $A \to \beta_1\ \beta_2$.

3.3 Extension: Pruning

We also experiment with a simple pruning dictionary pruning technique. For each context-free rule $A \to \beta_1\ \beta_2$ and POS tag a we remove all rules that were not seen with that tag as a head in training.

Pruning discussion

3.4 Binarization

In order to have an efficient binary LCFG grammar, we must convert the non-binary treebank grammars to binary form. While the algorithm itself is not dependent on the binarization used, this choice affects the run-time of the algorithm, through $\mathcal{R}$, as well as the structure of the scoring function.

Our binarization decomposes non-binary rules into fragments for each head-modifier pair.

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

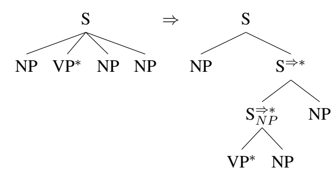
We replace this rule with binary rules that consume each side independently as a first-order Markov chain (horizontal Markovization). The main transformation is to introduce rules

- $A^{\Rightarrow}_{\beta_i} \to A^{\Rightarrow *}_{\beta_{i-1}}\ \beta_i$ for $k < i < m$
- $A^{\Leftarrow}_{\beta_i} \to \beta_i\ A^{\Leftarrow *}_{\beta_{i+1}}$ 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 $k = m$).

$$A^{\Rightarrow}_{\beta_{k+1}} \to \beta_k^*\ \beta_{k+1} \qquad A^{\Rightarrow *} \to A^{\Rightarrow}_{\beta_{m-1}}\ \beta_m$$
$$A^{\Leftarrow}_{\beta_{k-1}} \to \beta_{k-1}\ A^{\Rightarrow *} \qquad A \to \beta_1\ A^{\Leftarrow *}_{\beta_2}$$

For example the transformation of a common rule looks like



Each rule contains at most 3 original nonterminals so the size of the new binarized rule set is bounded by $O(\mathcal{N}^3)$.

4 Structured Prediction

To learn the scoring function for the transformation from dependency trees to phrase-structure trees, we use a standard structured prediction setup. We define the scoring function s as

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

where $\theta \in \mathbb{R}^D$ is a weight vector and $f(x, d, y)$ is a feature function that maps parse production (as in Figure **??**) to sparse feature vectors in $\{0, 1\}^D$. In this section we first discuss the features used and then training for the weight vector.

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
- head word word and part-of-speech

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

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

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

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

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

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 **??**. After training there are # non-zero features.

4.2 Training

We train the parameters θ using standard structured SVM training ().

We assume that we are given a set of gold-annotated parse examples: $(x^1, y^1), \dots, (x^D, y^D)$. We also define $d^1 \dots d^D$ as the dependency structures induced from $y^1 \dots y^D$. 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 Setup

5.1 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 experiment, we the standard Chinese Treebank 5.1 (CTB) experimental setup (Xue et al., 2005).

Part-of-speech tagging is done using TurboTagger (Martins et al., 2013). Prior to training, the train sections are automatically tagged using 10-fold jack-knifing. 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 a standard setup and 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 500 lines of code. Both are publicly available.[4]

6 Experiments

We ran experiments to assess the accuracy of the method, its run-time efficiency, the amount of

[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	
ParseDep	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
ParseDep		

Table 3: 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[] ().

phrase-structure data required, and the effect of dependency accuracy.

6.1 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.

...

6.2 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 3.3. 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			

oracle	dep	dev

Table 4: (a) Oracle accuracy for the predicted dependency trees of commonly used dependency parsers including TurboParser (Martins et al., 2013), MaltParser (Nivre et al., 2006), and MIT (). (b) Classification of the bracketing mistakes made by the parser.

6.3 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.

6.4 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 5 give the results for this task.

7 Conclusion

With recent advances in statistical dependency parsing, state-of-the-art parsers have reached the comparable dependency accuracy as the best phrase

Model	Dev Prec	Rec	F1
Xia[09]	88.1	90.7	89.4
PARSEDEP(Sec19)	98.5	98.6	98.5
PARSEDEP	99.5	99.7	99.6

Table 5: Comparison with the rule-based system of Xia et al. (2009). This experiment is run on gold dependencies and tags of Section 22. Xia[09] only trains on Sec. 19 but includes a footnote saying that more data would not help. For comparison we train on full PTB as well as just Sec. 19.

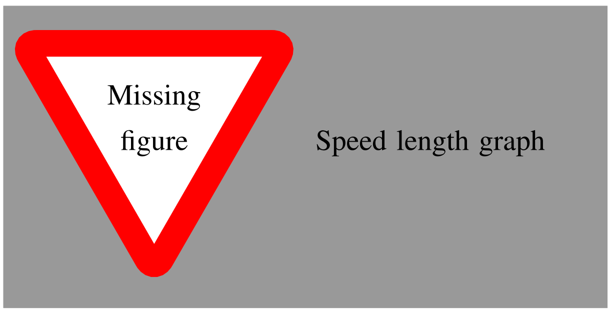


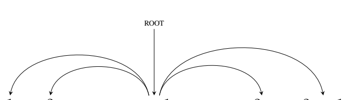
Table 6: Experiments of parsing speed. (a) The speed of the parser on its own and with pruning. (b) The end-to-end speed of the parser when combined with different dependency parsers.

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 problem, and show that it performs at the accuracy of phrase-structure parsers.

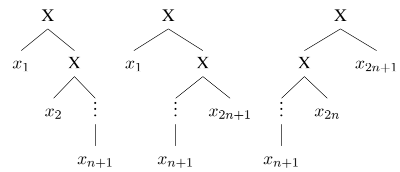
One question for future work is whether these results are language dependent, or whether these transformation can be projected across languages. If this were possible, we could use a system of this form to learn phrase structure parsers on languages with only dependency annotations.

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*.

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.