Transforming Dependencies into Phrase Structures

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 PARSEVAL accuracy near to the state of the art on English and Chinese benchmarks.

1 Introduction

Natural language parsers typically produce phrase-structure (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 subcommunity or used in 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 (reviewed in Section 2). 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 (Section 3): 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. Instead, we present a data-driven algorithm using the structured prediction framework (Section 4). The approach can be understood as a specially-trained coarse-to-fine decoding algorithm where a d-parser provides "coarse" structure and the second stage refines it (Charniak and Johnson, 2005; Petrov and Klein, 2007).

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. Experiments show that with simple pruning, our approach achieves linear observable runtime, and that accuracy similar to state-of-the-art phrase-structure parsers without reranking or semi-supervised training (Section 7).

2 Background

We begin with 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 consider the reverse transformation.

2.1 CFG Parsing

The phrase-structure trees annotated in the Penn Treebank are derivation trees from a context-free

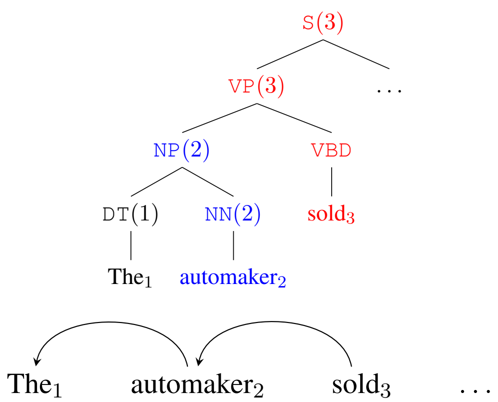


Figure 1: Illustration of c-parse to d-parse conversion with head rules $\{\texttt{VP} \rightarrow \texttt{NP}\ \texttt{VBD}^*, \texttt{NP} \rightarrow \texttt{DT}\ \texttt{NP}^*, \ldots\}$. The c-parse is an ordered tree with fringe $x_1, \ldots, x_n$. Each vertex is annotated with a nonterminal tag and a derived head index. The blue and red 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.

grammar. Define a binary[1] context-free grammar (CFG) as a 4-tuple $(\mathcal{N}, \mathcal{R}, \mathcal{T}, r)$ where $\mathcal{N}$ is a set of nonterminal symbols (e.g. NP, VP), $\mathcal{T}$ is a set of terminal symbols, consisting of the words in the language, $\mathcal{R}$ is a set of binary rule productions of the form $A \rightarrow \beta_1\ \beta_2$, and $r \in \mathcal{N}$ is 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 $\langle i, j \rangle$ with 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 representation of a sentence's structure (Collins, 2003; De Marneffe and Manning, 2008; Yamada and Matsumoto, 2003; Johansson and Nugues, 2007). In this work we consider transformations that are defined through local head rules.

For a binary CFG, define a collection of head rules as a function $\mathcal{H} : \mathcal{R} \mapsto \{L, R\}$ mapping each CFG rule to a 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 (d-parse). Define a sentence's dependencies as a sequence $d_1, \ldots d_n$ where word m is a dependent of word $d_m \in \{0, \ldots n\}$ 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 $d_m = h(v)$, else $d_m = 0$. Figure 1 illustrates the conversion.

These dependencies $d_1, \ldots, d_n$ 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 contiguous 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 we consider a flipped setup. There has been significant progress in developing efficient direct-to-dependency parsers. These d-parsers can be trained only on dependency annotations and do not require full phrase-structure trees.[2] Some prefer this setup, since it allows easy selection of the specific dependencies of interest in a downstream task (e.g., information extraction), and perhaps even training specifically for those dependencies. Some applications

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

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

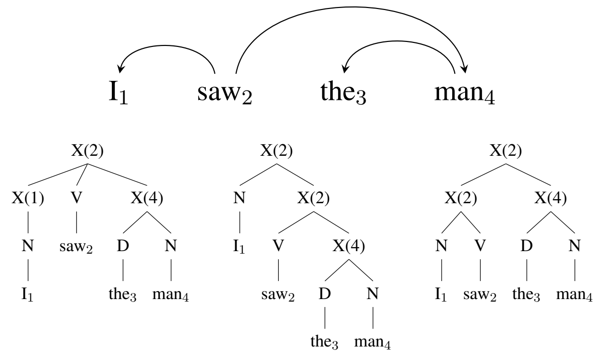


Figure 2: [Adapted from (Collins et al., 1999).] A d-parse (above) and several c-parses consistent with it (below). Our goal is to select the best parse from this set.

make use of phrase structures, so c-parsers enjoy wide use as well.

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

3.1 Parsing Algorithm

We start by considering the search problem with a given scoring function. First consider the standard problem of predicting the best c-parse under a CFG with head rules. Assume that we are given a binary 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. $\arg\max_{y \in \mathcal{Y}(x)} s(y; x)$ 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 CKY algorithm with lexicalized nonterminals. This algorithm is defined by the productions in Figure 3 (left). 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 not practical to run without heavy pruning or further assumptions on the scoring function (Eisner and Satta, 1999).

Now, consider the same setup, but constrained to c-parses that are consistent with a given d-parse, $d_1, \ldots, d_n$ (i.e., that could be converted, through the head rules, to this d-parse). Define this set as $\mathcal{Y}(x, d)$ and the search problem as, $\arg\max_{y \in \mathcal{Y}(x,d)} s(y; x, d)$

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 reduces 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 spans $\langle m_{\Leftarrow}, m_{\Rightarrow}\rangle$. Figure 3 shows the new algorithm.

For each word x_h in a d-parse, define $L(h)$ and $R(h)$ as the number of left and right dependents of h respectively. At any point in the algorithm word h may combine as head with its next left- or right-dependent. There are therefore $L(h) \times R(h)$ possible orderings of these combinations for each head. Since any rule in $\mathcal{R}$ can be used at each point, the running time of the algorithm is $O((\sum_h L(h) \times R(h))|\mathcal{R}|)$. While the number of dependents L and H could in theory be $O(n)$, in practice the number is often a constant term as we will show in Figure 6.

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. We can use this information to further prune the search space. We experimented with two heuristic pruning methods.

First, we observe that in training the part-of-speech of the head word x_h greatly restricts the possible rules $A \rightarrow \beta_1\ \beta_2$ available. To exploit this property we build tables $\mathcal{R}_{\mathrm{tag}(x_h)}$ where $\mathrm{tag}()$ is the part-of-speech tag of word x_h and further limit the search to rules seen in training for each head tag.

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. Left: 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. Right: 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 given the dependency parse. Finding the optimal c-parse with the new algorithm is $O\left((\sum_h L(h) \times R(h))|\mathcal{R}|\right)$ where L and R are the number of left and right dependents of word h.

This further reduces the run-time of the algorithm to $O(\sum_h (L(h) \times R(h) \times |\mathcal{R}_{\mathrm{tag}(x_h)}|))$.

Second, as noted above, at any point a word x_h with may combine with its next left or right dependent, leading to $L(h) \times R(h)$ possible orderings. However, in practice we can often predict which side will come next. To make this prediction, we estimate a distribution, $\eta(\text{side}) = p(\text{side} \mid \mathrm{tag}(x_h), \mathrm{tag}(x_l), \mathrm{tag}(x_r))$, where x_l is the next left dependent and x_r is the next right dependent. If the conditional probability of $\eta(\text{left})$ or $\eta(\text{right})$ is greater than a threshold parameter γ, we make a hard decision to combine with that side next. This pruning further reduces the impact of outlier words h with large $L(h) \times R(h)$.

We empirically measured how these pruning methods affect observed runtime and oracle parsing performance (i.e., how well a perfect scoring function could do with a pruned $\mathcal{Y}(x,d)$). Table **??** shows a comparison of these pruning methods on development data. These experiments show that the d-parse constraints contribute a large drop in oracle accuracy, while pruning contributes a relatively small one. Still, this upper-bound on accuracy is high enough to make it possible to still recover c-parses at least as accurate as state-of-the-art c-parsers. We will return to this discussion in Section 7.

Model	Complexity	Sent./s.	Ora. F_1
FULL*	$n^5\|\mathcal{R}\|$	0.25	100.0
DEP	$\sum_h L(h)R(h)\|\mathcal{R}\|$	71.2	92.6
PRUNE1	$\sum_h L(h)R(h)\|\mathcal{R}_{\mathrm{tag}(x_h)}\|$	336.0	92.5
PRUNE2	–	96.6	92.5
PRUNE1+2	–	425.1	92.5

Table 1: Comparison of three parsing setups: FULL* is the standard lexicalized grammar, but limited to only sentences less than 20 words, DEP restricts c-parses to be consistent with a predicted dependency skeleton, and PRUNED1, PRUNED2, and PRUNED1+2 are the pruning methods described in Section 3.2. The oracle is the best labeled F_1 achievable on the development data (see Section 7.1).

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.

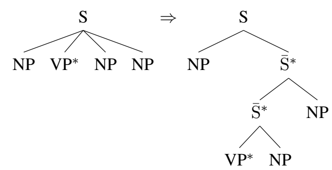
Consider a non-binarized rule of the form $A \rightarrow \beta_1 \dots \beta_m$ with head rule $\mathcal{H}(A \rightarrow \beta_1 \dots \beta_m) = k$. Relative to the head β_k the rule has left-side

$\beta_1 \ldots \beta_{k-1}$ and right-side $\beta_{k+1} \ldots \beta_m$.

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

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

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



We also explored binarization using horizontal and vertical markovization to include additional context of the tree, as found useful in unlexicalized approaches (Klein and Manning, 2003). Preliminary experiments showed that this increased the size of the grammar, and the run-time of the algorithm, without leading to improvements in accuracy.

4 Structured Prediction

While the parser only requires a d-parse at prediction time, the transformation it implements is learned from a treebank of c-parses. 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 parameters for scoring possible c-parses we use a standard structured prediction setup with a linear model. We first describe features, then parameter estimation.

4.1 Features

In theory, the scoring function for this system could be directly adapted from existing lexicalized 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.

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)
$(A, \beta_1, \mathrm{tag}(m))$
$(A, \beta_2, \mathrm{tag}(h))$

Span Features
(rule, x_i) (rule, x_{i-1})
(rule, x_j) (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 $f(x, d, y)$. For the span features, the symbol rule is expanded into both $A \rightarrow B \ C$ and A. The function $\mathrm{tag}(i)$ gives the part-of-speech tag of word x_i. The function $\mathrm{bin}(i)$ partitions a span length into one of 10 bins.

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 nonterminal, modifier word and part-of-speech, and 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 conjunctions 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, and binned length of the span.

The full feature set is shown in Figure 4. After training, there are a total of around 2 million nonzero features. For efficiency, we use lossy feature hashing. We found this had no impact on parsing accuracy but made the parsing significantly faster.

4.2 Training

The parameters θ are estimated using a structural support vector machine (Taskar et al., 2004). Given a set of gold-annotated parse examples, $(x^1, y^1), \ldots, (x^D, y^D)$, and dependency structures $d^1 \ldots d^D$ induced from the head rules, we estimate the parameters to minimize the regularized empirical risk

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

Model	PTB §22		
	Prec.	Rec.	F_1
Xia et al. (2009)	88.1	90.7	89.4
PARPAR (§19)	95.9	95.9	95.9
PARPAR (§2–21)	97.5	97.8	97.7

Table 2: Comparison with the rule-based system of Xia et al. (2009). Results are shown using gold-standard tags and dependencies. Xia et al. report results consulting only §19 in development and note that additional data had little effect. We show our system's results using §19 and the full training set.

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

where we define ℓ as $\ell(x, d, y, \theta) = -s(y) + \max_{y' \in \mathcal{Y}(x,d)} (s(y') + \Delta(y, y'))$ 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 for production rules firing over pairs of adjacent spans (i.e., i, j, k).

The objective is optimized using Adagrad (Duchi et al., 2011). The gradient calculation requires computing a loss-augmented max-scoring c-parse for each training example which is done using the algorithm of Figure 3 (right).

5 Related Work

[why we put the related work section between training and method? –lpk]

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 conversion of 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 dependency parses as input. Table 2 compares our system to that of Xia et al. (2009).

There has been successful work combining dependency and phrase-structure parsing. **[should probably cite the factored parser of Klein and Manning here –nas]** 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. This work differs in that we treat the dependency parse as a hard constraint, hence largely reduce the running time of a fully lexicalized phrase structure parsing model while maintaining the ability, at least in principle, to generate highly phrase-structure parses.

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 §2–21, development on §22, and testing on §23. We use the development set to tune regularization parameter, $\lambda = 1e-8$, and the pruning threshold, $\gamma = 0.95$.

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 articles 001–270 and 440–1151 for training, 301–325 for development, and 271–300 for test.

Part-of-speech tagging is performed for all models using TurboTagger (Martins et al., 2013). Prior to training the d-parser, the training sections are automatically processed using 10-fold jackknifing (**?**) for both dependency and phrase structure trees. Zhu et al. (2013) found this simple technique gives an improvement to dependency accuracy of 0.4% on English and 2.0% on Chinese.

Unless otherwise noted, the input d-parsing is done using the RedShift implementation[4] of the Zhang-Nivre Parser (Zhang and Nivre, 2011), trained to follow the conventions of Collins head rules (Collins, 2003). This parser is a transition-based beam search parser, and the size of the beam k controls a speed/accuracy trade-off. By default we use a beam of $k = 16$.

[4] `https://github.com/syllog1sm/redshift`

Additionally, while our parser does not use dependency labels, we found that labels have a significant impact on the performance of the RedShift parser. We therefore train a labeled parser, but discard the labels before conversion.

Evaluation for phrase-structure parses is performed using the `evalb`[5] script with the standard setup. We report labeled F_1 scores as well as recall and precision. For dependency parsing, we report 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 runtime efficiency, the effect of dependency parsing accuracy, and the effect of the amount of annotated phrase-structure data.

7.1 Parsing Accuracy

Our first experiments, shown in Table 3, give the accuracy and speed of the phrase-structure trees produced by the parser. For these experiments we treat our system and the Zhang-Nivre parser as an independently trained, but complete end-to-end c-parser. Run-time for these experiments includes both the time for d-parsing and conversion. This setup is limited in that the parser cannot change the dependencies given to it. Despite this limitation, the results show that the parser is comparable in accuracy to many widely-used systems, and is significantly faster. The only parser competitive in both speed and accuracy is the that of Zhu et al. (2013), a fast shift-reduce phrase-structure parser.

To apply our model on Chinese, we use the head rules complied by Ding and Palmer (2005) [7]. Since RedShift does not yet have Chinese support, we train our d-parser using TurboParser second order model [8] trained with projective constraints.

	PTB	
Model	F_1	Sent./s.
Charniak (2000)	89.5	-
Stanford PCFG (2003)	85.5	5.3
Petrov (2007)	90.1	8.6
Zhu (2013)	90.3	39.0
Carreras (2008)	91.1	-
CJ Reranking (2005)	91.5	4.3
Stanford RNN (2013)	90.0	2.8
ParPar	90.4	58.6
	CTB	
Model	F_1	
Charniak (2000)	80.8	
Bikel(2004)	80.6	
Petrov (2007)	83.3	
Zhu (2013)	83.2	
ParPar	81.0	

Table 3: Accuracy and speed on PTB §23 and CTB 5.1 test split. Comparisons are to state-of-the-art non-reranking supervised phrase-structure parsers (Charniak, 2000; Klein and Manning, 2003; Petrov and Klein, 2007; Carreras et al., 2008; Zhu et al., 2013; Bikel, 2004), and semi-supervised parsers (Charniak and Johnson, 2005; Socher et al., 2013).

The result in Table 3 suggests that, even with out doing any language specific changes in the feature system (e.g. using the last character of a Chinese word as features), we still achieve competitive parsing accuracy. Due to the time constraints, we leave explorations like adopting better d-parse systems such as ZPar (Zhang and Clark, 2011) and feature engineering a future work.

...

Effect of Dependencies Our next set of experiments look at the effect of different input dependency trees. We ran the same trained converter on seven different dependency inputs of varying quality measured by unlabeled accuracy score (UAS). Table 4 shows the results for each d-parser: its UAS, labeled F_1 when it is used with ParPar and with an oracle converter.[9] Figure 5 shows that there is a direct correlation between the UAS of the inputs and labeled F_1.

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

[6] URL withheld for review.

[7] `http://stp.lingfil.uu.se/~nivre/research/chn_headrules.txt`

[8] Martins et al. (2013) show that with third-order features, the UAS of parsing Chinese actually decreases.

[9] For a gold parse y and predicted dependencies $\hat{d}$, define the oracle parse as $y' = \arg\min_{y' \in \mathcal{Y}(x,\hat{d})} \Delta(y, y')$

Model	UAS	F_1	Sent./s.	Oracle
MALTPARSER	89.7	85.5	240.7	87.8
RS-K1	90.1	86.6	233.9	87.6
RS-K4	92.5	90.1	151.3	91.5
RS-K16	93.1	90.6	58.6	92.4
TP-BASIC	92.8	88.9	132.8	90.8
TP-STANDARD	93.3	90.9	27.2	92.6
TP-FULL	93.5	90.8	13.2	92.9

Table 4: Comparison of the effect of d-parsing accuracy (PTB §22) on PARPAR and an oracle converter. Runtime includes d-parsing and c-parsing. Inputs include MaltParser (Nivre et al., 2006), the RedShift implementation of the Zhang-Nivre parser (Zhang and Nivre, 2011) with beam size $K \in \{1, 4, 16\}$, and three versions of TurboParser trained with projective constraints (Martins et al., 2013).

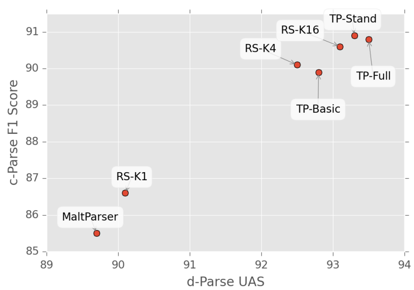


Figure 5: Relationship of d-parsing performance to c-parsing performance using PARPAR to convert different dependency parsers' output.

Runtime In Section 3 we considered the theoretical complexity of the parsing model and present the main speed results in Table **??**. Despite having a quadratic theoretical complexity the practical runtime was quite fast. Here we consider the empirical complexity of the model by measuring the time spent of individual sentences. Figure 6 shows a graph of the speed of sentences of varying length for both the full algorithm and for the algorithm with pruning. In both cases the observed runtime is linear.

Less Annotation in Phrase-structures, More Annotation in Dependencies? **[quick highlights... need rewrite –lpk]** Annotating unlabeled dependency has been proven to be cheaper and faster than annotating phrase structure. (**?**) What if we need the phrase-structure trees? Our parser given a chance to annotate more dependencies, while get a good phrase-structure parser.

Show in Figure **??**. We train a simple PCFG

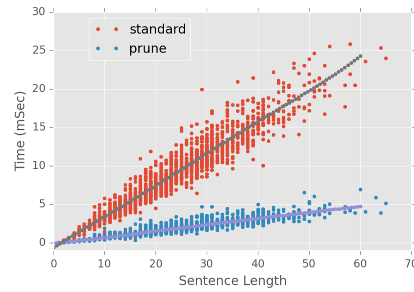


Figure 6: Empirical runtime of the parser on sentences of varying length, with and without pruning. Despite a worst-case quadratic complexity, observed runtime is linear. Pruning significantly improves the slope.

Class			Results	
Dep. (h, m)	Span $\langle i, j \rangle$	Split k	Count	Acc. A
+	+	+	32853	97.9
–	+	+	381	69.3
+	+	–	802	83.3
–	+	–	496	85.9
+	–	–	1717	0.0
–	–	–	1794	0.0

Table 5: Error analysis of binary CFG rules. Rules used are split into classes based on correct (+) identification of dependency (h, m), span $\langle i, j \rangle$, and split k. "Count" is the size of each class. "Acc." is the accuracy of span nonterminal identification.

model with $vMarkov = 2$ and $hMarkov = 2$ (**?**).

The are two scenarios here, blue and red.

Blue means we label whole phrase-structure parse tree, given 5%, 20%, 50%, 80% and 100% annotation of the PTB, the parser's performance goes up, but this way is expensive.

Red means we only label 5% phrase-structure parses, and then we train our parser on this 5% phrase-structure model (with dependencies extracted using collins headrule), then, we use our parser's predicted phrase-structure tree (predicted on gold dependency parses, assuming people are annotating that), from 20% to 100%, to train the phrase-structure parser.

We note that the accuracy of the PCFG model also goes higher with more dependency annotations, which suggests a cheaper and fast (?) way to build a language resource in phrase-structure.

Analysis Finally we consider an internal error analysis of the parser. For this analysis, we group each binary rule production selected by the parser by three properties: Is its dependency (h, m) correct? Is its span $\langle i, j \rangle$ correct? Is its split k correct? The first is fully determined by the input d-parse, the second are partially determined by PARPAR.

Table **??** gives the breakdown. As expected the conversion is almost always correct when it has this information **[what is "this information"? –nas]**. Many of the errors come from cases where the dependency was fully incorrect, or there was a span issue either from the dependency parser or from the conversion itself.

8 Conclusion

With recent advances in statistical dependency parsing, we find that fast, high-quality phrase-structure parsing is achievable using dependency parsing first, followed by a statistical conversion algorithm to fill in phrase-structure trees. Our implementation is available as open-source software at (URL withheld for review).

References

Daniel M Bikel. 2004. *On the parameter space of generative lexicalized statistical parsing models*. Ph.D. thesis, University of Pennsylvania.

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 and Mark Johnson. 2005. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 173–180. 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.

Marie-Catherine De Marneffe and Christopher D Manning. 2008. The stanford typed dependencies representation. In *Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation*, pages 1–8. Association for Computational Linguistics.

Yuan Ding and Martha Palmer. 2005. Machine translation using probabilistic synchronous dependency insertion grammars. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, pages 541–548. Association for Computational Linguistics.

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159.

Jason Eisner and Giorgio Satta. 1999. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, pages 457–464. Association for Computational Linguistics.

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

Richard Johansson and Pierre Nugues. 2007. Extended constituent-to-dependency conversion for english. In *16th Nordic Conference of Computational Linguistics*, pages 105–112. University of Tartu.

Dan Klein and Christopher D Manning. 2003. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 423–430. Association for Computational Linguistics.

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 and Dan Klein. 2007. Improved inference for unlexicalized parsing. In *HLT-NAACL*, pages 404–411. Citeseer.

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.

Richard Socher, John Bauer, Christopher D Manning, and Andrew Y Ng. 2013. Parsing with compositional vector grammars. In *In Proceedings of the ACL conference*.

Ben Taskar, Carlos Guestrin, and Daphne Koller. 2004. Max-margin Markov networks. In *Advances in Neural Information Processing Systems 16*.

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.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proceedings of IWPT*, volume 3.

Yue Zhang and Stephen Clark. 2011. Syntactic processing using the generalized perceptron and beam search. *Computational Linguistics*, 37(1):105–151.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2*, pages 188–193. Association for Computational Linguistics.

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.