Transforming Dependencies into Phrase Structures

Lingpeng Kong
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
lingpenk@cs.cmu.edu

Alexander M. Rush
Facebook AI Research
New York, NY, USA
srush@seas.harvard.edu

Noah A. Smith
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
nasmith@cs.cmu.edu

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 90.4% English parsing accuracy and 82.4% Chinese parsing accuracy, near to the state of the art on both 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 (Klein and Manning, 2002; Hall and Nivre, 2008; 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, PAD, 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 our approach achieves linear observable runtime, and 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 grammar. Define a binary[1] context-free grammar

[1]For notational simplicity, we defer discussion of non-binary rules to Section 3.3.

(CFG) as a 4-tuple $(\mathcal{N}, \mathcal{G}, \mathcal{T}, r)$ 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, $\mathcal{G}$ is a set of binary rules of the form $A \rightarrow \beta_1\ \beta_2$, and $r \in \mathcal{N}$ is a distinguished root nonterminal 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 $A \rightarrow \beta_1\ \beta_2$ consisting of members of $\mathcal{G}$, and root label r.

For a c-parse $y \in \mathcal{Y}(x)$, we further associate a span $\langle v_{\Leftarrow}, v_{\Rightarrow} \rangle$ with each vertex in the tree. This specifies the subsequence $\{x_{v_{\Leftarrow}}, \ldots, x_{v_{\Rightarrow}}\}$ of the sentence covered by this vertex.

2.2 Dependency Parsing

Dependency parses 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 transformations 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).

We consider the class of transformations that are defined through local *head rules*. For a binary CFG, define a collection of head rules as a mapping from 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.

The head rules can be used to map a c-parse to a dependency tree (d-parse). In a d-parse, each word in the sentence is assigned as a dependent to a head word, $h \in \{0, \ldots, n\}$, where 0 is a special symbol indicating the pseudo-root of the sentence. For each h we define $\mathcal{L}(h) \subset \{1, \ldots, h-1\}$ as the set of left dependencies of h, and $\mathcal{R}(h) \subset \{h+1, \ldots, n\}$ as the set of right dependencies.

A d-parse can be constructed recursively from a c-parse and the head rules. For each c-parse vertex v with potential children v_{L} and v_{R} in bottom-up order, we apply the following procedure to both assign heads to the c-parse and construct the d-parse:

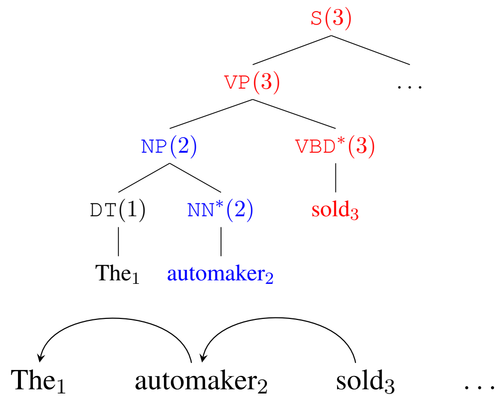


Figure 1: Illustration of c-parse to d-parse conversion with head rules $\{\text{VP} \rightarrow \text{NP VBD}^*, \text{NP} \rightarrow \text{DT NN}^*, \ldots\}$. The c-parse is an ordered tree with fringe $x_1, \ldots, x_n$. Each vertex is annotated with a terminal or nonterminal symbol and a derived head index. The blue and red vertices have the words automaker$_2$ and sold$_3$ as heads respectively. The vertex VP(3) implies that automaker$_2$ is a left-dependent of sold$_3$, and that $2 \in \mathcal{L}(3)$ in the d-parse.

1. If the vertex is leaf x_m, then $\text{head}(v) = m$.
2. If the next rule is $A \rightarrow \beta_1^*\ \beta_2$ then $\text{head}(v) = \text{head}(v_{\text{L}})$ and $\text{head}(v_{\text{R}}) \in \mathcal{R}(\text{head}(v))$, i.e. the head of the right-child is a dependent of the head word.
3. If the next rule is $A \rightarrow \beta_1\ \beta_2^*$ then $\text{head}(v) = \text{head}(v_{\text{R}})$ and $\text{head}(v_{\text{L}}) \in \mathcal{L}(\text{head}(v))$, i.e. the head of the left-child is a dependent of the head word.

Figure 1 shows an example conversion of a c-parse to d-parse using this procedure.

By construction, these dependencies form a directed tree with arcs (h, m) for all $h \in \{0, \ldots, n\}$ and $m \in \mathcal{L}(h) \cup \mathcal{R}(h)$. While this tree differs from the original c-parse, we can relate the two trees through their spans. Define the dependency tree span $\langle h_{\Leftarrow}, h_{\Rightarrow} \rangle$ as the contiguous sequence of words reachable from word h in this tree.[2] This span is equivalent to the maximal span $\langle v_{\Leftarrow}, v_{\Rightarrow} \rangle$ of any c-parse vertex with $\text{head}(v) = h$. This property will be important for the parsing algorithm presented in the next section.

[2]The conversion from a standard CFG tree to a d-parse preserves this sequence property, known as *projectivity*. We leave the question of non-projective d-parses and the related question of traces and co-indexation in c-parses to future work.

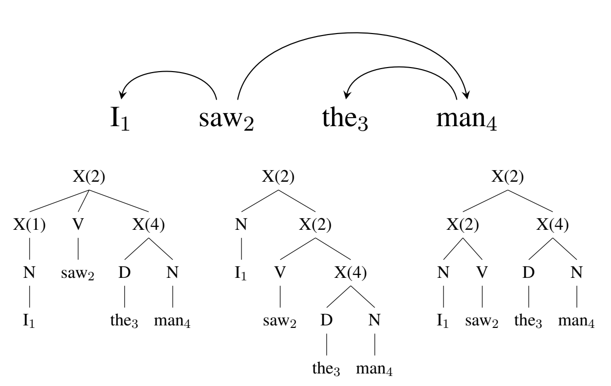


Figure 2: [Adapted from (Collins et al., 1999).] A d-parse (left) and several c-parses consistent with it (right). Our goal is to select the best parse from this set.

3 Parsing Dependencies

Now we consider flipping this setup. There has been significant progress in developing efficient direct-to-dependency parsers. These d-parsers are trained only on dependency annotations and do not require full phrase-structure trees.[3] 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. Other applications 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 fixed d-parse into a c-parse, with the intent of using off-the-shelf d-parsers for constructing phrase-structure parses. 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. A toy example of the problem is shown in Figure 2.

3.1 Parsing Algorithm

Consider the classical problem of predicting the best c-parse under a CFG with head rules, known as lexicalized context-free parsing. 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 lexicalized tree productions.

This problem can be solved by extending the CKY algorithm to propagate head information. The algorithm can be compactly defined by the productions in Figure 3 (left). For example, one type of production is 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 $A \rightarrow \beta_1\ \beta_2^* \in \mathcal{G}$ 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 has dependent index m. We say this production "completes" word m since it can no longer be the head of a larger span.

Running the algorithm consists of bottom-up dynamic programming over these productions. However, applying this version of the CKY algorithm requires $O(n^5|\mathcal{G}|)$ time (linear in the number of productions), which is not practical to run without heavy pruning. Most lexicalized parsers therefore make further assumptions on the scoring function which can lead to asymptotically faster algorithms (Eisner and Satta, 1999).

Instead, we consider the same objective, but constrain the c-parses to be consistent with a given d-parse, d. By "consistent," we mean that the c-parse will be converted by the head rules to this exact d-parse.[4] Define the set of consistent c-parses as $\mathcal{Y}(x, d)$ and the constrained search problem as $\arg\max_{y \in \mathcal{Y}(x,d)} s(y; x, d)$.

Figure 3 (right) shows the algorithm for this new problem. The algorithm has several nice properties. All rules now must select words h and m that are consistent with the dependency parse (i.e., there is an arc (h, m)) so these variables are no longer free. Furthermore, since we have the full d-parse, we can precompute the dependency span of each word $\langle m_{\Leftarrow}, m_{\Rightarrow} \rangle$. By our definition of consistency, this gives us the c-parse span of m before it is completed, and fixes two more free variables. Finally the head item must have its alternative side index match

[3]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.

[4]An alternative, soft version of consistency, might enforce that the c-parse is close to the d-parse. While this allows the algorithm to potentially correct d-parse mistakes, it is much more computationally expensive.

Premise:

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

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:

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

Rules:
For all $h, m \in \mathcal{R}(h)$, rule $A \rightarrow \beta_1^* \; \beta_2$,
and $i \in \{m'_{\Leftarrow} : m' \in \mathcal{L}(h)\} \cup \{h\}$,

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

For all $h, m \in \mathcal{L}(h)$, rule $A \rightarrow \beta_1 \; \beta_2^*$,
and $j \in \{m'_{\Rightarrow} : m' \in \mathcal{R}(h)\} \cup \{h\}$,

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

Goal:

$$(\langle 1, n\rangle, m, r) \text{ for any } m \in \mathcal{R}(0)$$

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 $O(n^5|\mathcal{G}|)$ 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 now requires $O\left((\sum_h |\mathcal{L}(h)||\mathcal{R}(h)|)|\mathcal{G}|\right)$ time where $\mathcal{L}(h)$ and $\mathcal{R}(h)$ are the left and right dependents of word h.

a valid dependency span. For example, if for a word h there are $|\mathcal{L}(h)| = 3$ left dependents, then when taking the next right-dependent there can only be 4 valid left boundary indices.

The runtime of the final algorithm reduces to $O(\sum_h |\mathcal{L}(h)||\mathcal{R}(h)||\mathcal{G}|)$. While the terms $|\mathcal{L}(h)|$ and $|\mathcal{R}(h)|$ could in theory make the runtime quadratic, in practice the number of dependents is almost always constant in the length of the sentence. This leads to linear observed runtime in practice as we will show in Section 7.

3.2 Pruning

In addition to constraining the number of c-parses, 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 employ two pruning methods:

Method 1 uses the part-of-speech tag of x_h, $\text{tag}(h)$, to limit the possible rule productions at a given span. We build tables $\mathcal{G}_{\text{tag}(h)}$ and restrict the search to rules seen in training for a particular part-of-speech tag.

Method 2 prunes based on the order in which dependent words are added. By the constraints of the algorithm, a head word x_h must combine with each of its left and right dependents. However, the order of combination can lead to different tree structures (as illustrated in Figure 2). In total there are $|\mathcal{L}(h)| \times |\mathcal{R}(h)|$ possible orderings of dependents.

In practice, though, it is often easy to predict which side, left or right, will come next. We do this by estimating the distribution,

$$p(\text{side} \mid \text{tag}(h), \text{tag}(m), \text{tag}(m')),$$

where $m \in \mathcal{L}(h)$ is the next left dependent and $m' \in \mathcal{R}(h)$ is the next right dependent. If the conditional probability of left or 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 outliers with multiple dependents on both sides.

We empirically measure 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 1 shows a comparison of these pruning methods on development data. The constrained parsing algorithm is much faster than standard lexicalized parsing, and

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

Table 1: Comparison of three parsing setups: LEX CKY* is the complete lexicalized c-parser on $\mathcal{Y}(x)$, but limited to only sentences less than 20 words for tractability, DEP CKY is the constrained c-parser on $\mathcal{Y}(x, d)$, PRUNE1, PRUNE2, and PRUNE1+2 are combinations of the pruning methods described in Section 3.2. The oracle is the best labeled F_1 achievable on the development data (§22, see Section 7).

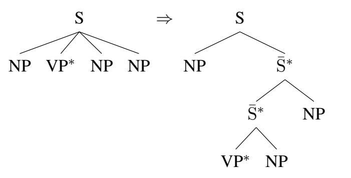
pruning contributes even greater speed-ups. The oracle experiments show that the d-parse constraints do contribute a large drop in oracle accuracy, while pruning contributes a relatively small one. Still, this upper-bound on accuracy is high enough to make it possible to still recover c-parses at least as accurate as state-of-the-art c-parsers. We will return to this discussion in Section 7.

3.3 Binarization and Unary Rules

We have to this point developed the algorithm for a strictly binary-branching grammar; however, we need to produce trees have rules with varying size. In order to apply the algorithm, we binarize the grammar and add productions to handle unary rules.

Consider a non-binarized rule of the form $A \rightarrow \beta_1 \dots \beta_m$ with head child β_k^*. Relative to the head child 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 new binary rules and non-terminal symbols to produce each side independently as a simple chain, left-side first. The transformation introduces the following new rules:[5] $A \rightarrow \beta_1\ \bar{A}^*$, $\bar{A} \rightarrow \beta_i\ \bar{A}^*$ for $i \in \{2, \dots, k\}$, and $\bar{A} \rightarrow \bar{A}^*\ \beta_i$ for $i \in \{k, \dots, m\}$.

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



These rules can then be reversed deterministically to produce a non-binary tree.

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

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 runtime of the algorithm, without leading to improvements in accuracy.

Phrase-structure trees also include unary rules of the form $A \rightarrow \beta_1^*$. To handle unary rules we modify the parsing algorithms in Figure 3 to include a unary completion rule,

$$\frac{(\langle i, j\rangle, h, \beta_1)}{(\langle i, j\rangle, h, A)}$$

for all indices $i \leq h \leq j$ that are consistent with the dependency parse. In order to avoid unary recursion, we limit the number of applications of this rule at each span (preserving the runtime of the algorithm). Preliminary experiments looked at collapsing the unary rules into the nonterminal symbols, but we found that this hurt performance compared to explicit unary rules.

4 Structured Prediction

We learn the d-parse to c-parse conversion using a standard structured prediction setup. Define the linear scoring function s for a conversion 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. While the parser only requires a d-parse at prediction time, the parameters of this scoring function are learned directly from a treebank of c-parses and a set of head rules. The structured prediction model, in effect, learns to invert the head rule transformation.

4.1 Features

The scoring function requires specifying a set of parse features f which, in theory, could be directly adapted from existing lexicalized c-parsers. However, the structure of the dependency parse greatly limits the number of decisions that need to be made, and allows for a smaller set of features.

We model our features after two bare-bones parsing systems. The first set is the basic arc-factored features used by McDonald (2006). These features include combinations of: rule and top nonterminal, 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	Rule Features
(A, β_1) $(A, \beta_1, \mathrm{tag}(m))$	(rule)
(A, β_2) $(A, \beta_2, \mathrm{tag}(h))$	$(\mathrm{rule}, x_h, \mathrm{tag}(m))$
Span Features	$(\mathrm{rule}, \mathrm{tag}(h), x_m)$
(rule, x_i) (rule, x_{i-1})	$(\mathrm{rule}, \mathrm{tag}(h), \mathrm{tag}(m))$
(rule, x_j) (rule, x_{j+1})	(rule, x_h)
(rule, x_k) (rule, x_{k+1})	$(\mathrm{rule}, \mathrm{tag}(h))$
$(\mathrm{rule}, \mathrm{bin}(j-i))$	(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 backoff symbol A. The function $\mathrm{bin}(i)$ partitions a span length into one of 10 bins.

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 c-parse examples, $(x^1, y^1), \ldots, (x^D, y^D)$, and d-parses $d^1 \ldots d^D$ induced from the head rules, we estimate the parameters to minimize the regularized empirical risk

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

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).

	PTB §22		
Model	Prec.	Rec.	F_1
Xia et al. (2009)	88.1	90.7	89.4
PAD (§19)	95.9	95.9	95.9
PAD (§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.

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

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. This work focuses on modeling the conversion decisions made and capturing how researchers annotate specific phenomena. Our work focuses on a different problem of learning a data-driven structured prediction model that is also able to handle automatically predicted dependency parses as input. While the aim is different, Table 2 does give a direct comparison of our system to that of Xia et al. (2009) on gold d-parse data.

An important line of previous work also uses dependency parsers to produce phrase-structure trees. In particular Hall et al. (2007) and Hall and Nivre (2008) develop a specialized dependency label set to encode phrase-structure information in the d-parse. After predicting a d-parse this label information can be used to assemble a predicted c-parse. Our work differs in that it does not make any assumptions on the labeling of the dependency tree used and it uses structured prediction to produce the final c-parse.

Very recently, Fernández-González and Martins (2015) also show that an off-the-shelf, trainable, dependency parser is enough to build a highly-competitive constituent parser. They proposed a new intermediate representation called "head-ordered dependency trees", which encode head ordering information in dependeny labels. Their algorithm is based on a reduction of the constituent parsing to dependency parsing of such trees.

There has been successful work combining dependency and phrase-structure information to build accurate c-parsers. Klein and Manning (2002) construct a factored generative model that scores both context-free syntactic productions and semantic dependencies. Carreras et al. (2008) construct a state-of-the-art 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 powerful dependency parser with a lexicalized phrase-structure model. This work differs in that we treat the dependency parse as a hard constraint, hence largely reduce the runtime of a fully lexicalized phrase structure parsing model while maintaining the ability, at least in principle, to generate highly accurate 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 our parser, PAD, 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 the 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. We also use the development set to tune the regularization parameter, $\lambda = 1e-3$.

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 (Collins and Koo, 2005) 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 in their system.

During training, we use the d-parses induced by the head rules from the gold c-parses as constraints. There is a slight mismatch here with test, since these d-parses are guaranteed to be consistent with the target c-parse. We also experimented with using 10-fold jacknifing of the d-parser during training to produce more realistic parses; however, we found that this hurt performance of the parser.

Unless otherwise noted, in English the test d-parsing is done using the RedShift implementation[6] of the parser of 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 $k = 16$. We found that dependency labels have a significant impact on the performance of the RedShift parser, but not on English dependency conversion. We therefore train a labeled parser, but discard the labels.

For Chinese, we use the head rules compiled by Ding and Palmer (2005)[7]. For this data-set we trained the d-parser using the YaraParser implementation[8] of the parser of Zhang and Nivre (2011), because it has a better Chinese implementation. We use a beam of $k = 64$. In experiments, we found that Chinese labels were quite helpful, and added four additional features templates conjoining the label with the non-terminals of a rule.

Evaluation for phrase-structure parses is performed using the `evalb`[9] script with the standard setup. We report labeled F_1 scores as well as recall and precision. For dependency parsing, we report

[6]https://github.com/syllog1sm/redshift

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

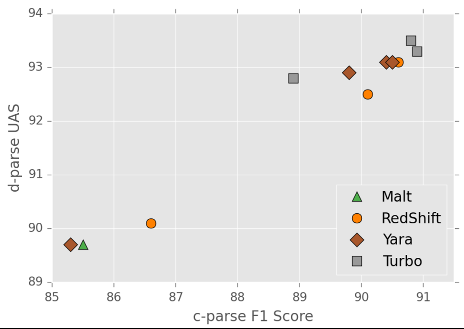
[8]https://github.com/yahoo/YaraParser

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

Model	PTB §23 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
PAD	90.4	34.3
PAD (Pruned)	90.3	58.6

Model	CTB F_1
Charniak (2000)	80.8
Bikel (2004)	80.6
Petrov (2007)	83.3
Zhu (2013)	83.2
PAD	82.4

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 and reranking parsers (Charniak and Johnson, 2005; Socher et al., 2013).



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	92.5
YARA-K1	89.7	85.3	1265.8	86.7
YARA-K16	92.9	89.8	157.5	91.7
YARA-K32	93.1	90.4	48.3	92.0
YARA-K64	93.1	90.5	47.3	92.2
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: The effect of d-parsing accuracy (PTB §22) on PAD and an oracle converter. Runtime includes d-parsing and c-parsing. Inputs include MaltParser (Nivre et al., 2006), the RedShift and the Yara implementations of the parser of Zhang and Nivre (2011) with various beam size, and three versions of TurboParser trained with projective constraints (Martins et al., 2013).

unlabeled accuracy score (UAS).

We implemented the grammar binarization, head rules, and pruning tables in Python, and the parser, features, and training in C++. 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.

Parsing Accuracy Table 3 compares 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. Runtime for these experiments includes both the time for d-parsing and conversion. Despite the fixed dependency constraints, the English results show that the parser is comparable in accuracy to many widely-used systems, and is significantly faster. The parser most competitive in both speed and accuracy is that of Zhu et al. (2013), a fast shift-reduce phrase-structure parser.

Furthermore, the Chinese results suggest that, even without making language-specific changes in the feature system we can still achieve competitive parsing accuracy.

Effect of Dependencies Table 4 shows experiments comparing the effect of different input d-parses. For these experiments we used the same version of PAD with 11 different d-parsers of varying quality and speed. We measure for each parser: its UAS, speed, and labeled F_1 when used with PAD and with an oracle converter.[10] The paired figure

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

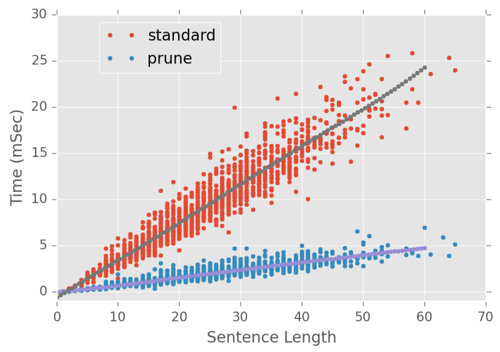


Figure 5: Empirical runtime of the parser on sentences of varying length, with and without pruning. Despite a worst-case quadratic complexity, observed runtime is linear.

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.

shows that there is a direct correlation between the UAS of the inputs and labeled F_1.

Runtime In Section 3 we considered the theoretical complexity of the parsing model and presented the main speed results in Table 1. 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 on individual sentences. Figure 5 shows parser speed for sentences of varying length for both the full algorithm and with pruning. In both cases the observed runtime is linear.

Recovering Phrase-Structure Treebanks Annotating phrase-structure trees is often more expensive and slower than annotating unlabeled dependency trees (Schneider et al., 2013). For low-resource languages, an alternative approach to developing fully annotated phrase-structure treebanks might be to label a small amount of c-parses and a large amount of cheaper d-parses. Assuming this setup, we ask how many c-parses would be necessary to obtain reasonable performance?

For this experiment, we train PAD on only 5% of the PTB training set and apply it to predicted d-parses from a fully-trained model. Even with this small amount of data, we obtain a parser with development score of $F_1 = 89.1\%$, which is comparable to Charniak (2000) and Stanford PCFG (Klein and Manning, 2003) trained on the complete c-parse training set. Additionally, if the gold dependencies are available, PAD with 5% training achieves $F_1 = 95.8\%$ on development, demonstrating a strong ability to recover the phrase-structure trees from dependency annotations.

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 property is fully determined by the input d-parse, the others are partially determined by PAD itself.

Table 5 shows the breakdown. The conversion is almost always accurate ($\sim$98%) when the parser has correct span and dependency information. As expected, the difficult cases come when the dependency was fully incorrect, or there is a propagated span mistake. As dependency parsers improve, the performance of PAD should improve as well.

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 `https://github.com/ikekonglp/PAD`.

Acknowledgments The authors thank the anonymous reviewers and André Martins, Chris Dyer, and Slav Petrov for helpful feedback. This research was supported in part by NSF grant IIS-1352440 and computing resources provided by Google and the Pittsburgh Supercomputing Center.

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 and Terry Koo. 2005. Discriminative reranking for natural language parsing. *Computational Linguistics*, 31(1):25–70.

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.

Daniel Fernández-González and André FT Martins. 2015. Parsing as reduction. *arXiv preprint arXiv:1503.00030*.

Johan Hall and Joakim Nivre. 2008. A dependency-driven parser for german dependency and constituency representations. In *Proceedings of the Workshop on Parsing German*, pages 47–54. Association for Computational Linguistics.

Johan Hall, Joakim Nivre, and Jens Nilsson. 2007. A hybrid constituency-dependency parser for swedish. In *Proceedings of NODALIDA*, pages 284–287.

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. 2002. Fast exact inference with a factored model for natural language parsing. In *Advances in neural information processing systems*, pages 3–10.

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.

Nathan Schneider, Brendan O'Connor, Naomi Saphra, David Bamman, Manaal Faruqui, Noah A Smith, Chris Dyer, and Jason Baldridge. 2013. A framework for (under) specifying dependency syntax without overloading annotators. *arXiv preprint arXiv:1306.2091*.

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 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.