Shift-Reduce CCG Parsing with a Dependency Model

Wenduan Xu
University of Cambridge
Computer Laboratory
wx217@cam.ac.uk

Stephen Clark
University of Cambridge
Computer Laboratory
sc609@cam.ac.uk

Yue Zhang
Singapore University
of Technology and Design
yue_zhang@sutd.edu.sg

Abstract

This paper presents the first dependency model for a shift-reduce CCG parser. Modelling dependencies is desirable for a number of reasons, including handling the "spurious" ambiguity of CCG; fitting well with the theory of CCG; and optimizing for structures which are evaluated at test time. We develop a novel training technique using a dependency oracle, in which all derivations are hidden. A challenge arises from the fact that the oracle needs to keep track of exponentially many gold-standard derivations, which is solved by integrating a packed parse forest with the beam-search decoder. Standard CCGBank tests show the model achieves up to 1.05 labeled F-score improvements over three existing, competitive CCG parsing models.

1 Introduction

Combinatory Categorial Grammar (CCG; Steedman (2000)) is able to derive typed dependency structures (Hockenmaier, 2003; Clark and Curran, 2007), providing a useful approximation to the underlying predicate-argument relations of "who did what to whom". To date, CCG remains the most competitive formalism for recovering "deep" dependencies arising from many linguistic phenomena such as raising, control, extraction and coordination (Rimell et al., 2009; Nivre et al., 2010).

To achieve its expressiveness, CCG exhibits so-called "spurious" ambiguity, permitting many non-standard surface derivations which ease the recovery of certain dependencies, especially those arising from type-raising and composition. But this raises the question of what is the most suitable model for CCG: *should we model the derivations, the dependencies, or both?* The choice for some existing parsers (Hockenmaier, 2003; Clark and Curran, 2007) is to model derivations directly, restricting the gold-standard to be the normal-form derivations (Eisner, 1996) from CCGBank (Hockenmaier and Steedman, 2007).

Modelling dependencies, as a proxy for the semantic interpretation, fits well with the theory of CCG, in which Steedman (2000) argues that the derivation is merely a "trace" of the underlying syntactic process, and that the structure which is built, and predicated over when applying constraints on grammaticality, is the semantic interpretation. The early dependency model of Clark et al. (2002), in which model features were defined over *only* dependency structures, was partly motivated by these theoretical observations.

More generally, dependency models are desirable for a number of reasons. First, modelling dependencies provides an elegant solution to the spurious ambiguity problem (Clark and Curran, 2007). Second, obtaining training data for dependencies is likely to be easier than for syntactic derivations, especially for incomplete data (Schneider et al., 2013). Clark and Curran (2006) show how the dependency model from Clark and Curran (2007) extends naturally to the partial-training case, and also how to obtain dependency data cheaply from gold-standard lexical category sequences alone. And third, it has been argued that dependencies are an ideal representation for parser evaluation, especially for CCG (Briscoe and Carroll, 2006; Clark and Hockenmaier, 2002), and so optimizing for dependency recovery makes sense from an evaluation perspective.

In this paper, we fill a gap in the literature by developing the first dependency model for a shift-reduce CCG parser. Shift-reduce parsing applies naturally to CCG (Zhang and Clark, 2011), and the left-to-right, incremental nature of the decoding fits with CCG's cognitive claims. The discriminative model is global and trained with the structured perceptron. The decoder is based on beam-search

(Zhang and Clark, 2008) with the advantage of linear-time decoding (Goldberg et al., 2013).

A main contribution of the paper is a novel technique for training the parser using a dependency oracle, in which all derivations are hidden. A challenge arises from the potentially exponential number of derivations leading to a gold-standard dependency structure, which the oracle needs to keep track of. Our solution is an integration of a packed parse *forest*, which efficiently stores all the derivations, with the beam-search decoder *at training time*. The derivations are not explicitly part of the data, since the forest is built from the gold-standard dependencies. We also show how perceptron learning with beam-search (Collins and Roark, 2004) can be extended to handle the additional ambiguity, by adapting the "violation-fixing" perceptron of Huang et al. (2012).

Results on the standard CCGBank tests show that our parser achieves absolute labeled F-score gains of up to 0.5 over the shift-reduce parser of Zhang and Clark (2011); and up to 1.05 and 0.64 over the normal-form and hybrid models of Clark and Curran (2007), respectively.

2 Shift-Reduce with Beam-Search

This section describes how shift-reduce techniques can be applied to CCG, following Zhang and Clark (2011). First we describe the deterministic process which a parser would follow when tracing out a single, correct derivation; then we describe how a model of normal-form derivations — or, more accurately, a sequence of shift-reduce actions leading to a normal-form derivation — can be used with beam-search to develop a non-deterministic parser which selects the highest scoring sequence of actions. Note this section only describes a normal-form derivation model for shift-reduce parsing. Section 3 explains how we extend the approach to dependency models.

The shift-reduce algorithm adapted to CCG is similar to that of shift-reduce dependency parsing (Yamada and Matsumoto, 2003; Nivre and McDonald, 2008; Zhang and Clark, 2008; Huang and Sagae, 2010). Following Zhang and Clark (2011), we define each item in the parser as a pair $\langle s, q \rangle$, where q is a queue of remaining input, consisting of words and a set of possible lexical categories for each word (with q_0 being the front word), and s is the stack that holds subtrees $s_0, s_1, ...$ (with s_0 at the top). Subtrees on the stack are partial derivations that have been built as part of the shift-reduce process. SHIFT, REDUCE and UNARY are the three types of actions that can be applied to an item. A SHIFT action shifts one of the lexical categories of q_0 onto the stack. A REDUCE action combines s_0 and s_1 according to a CCG combinatory rule, producing a new category on the top of the stack. A UNARY action applies either a type-raising or type-changing rule to the stack-top category s_0.[1]

step	stack $(s_n, ..., s_1, s_0)$	queue $(q_0, q_1, ..., q_m)$	action
0		Mr. President visited Paris	
1	N/N	President visited Paris	SHIFT
2	$N/N\ N$	visited Paris	SHIFT
3	N	visited Paris	REDUCE
4	NP	visited Paris	UNARY
5	$NP\ (S[dcl]\backslash NP)/NP$	Paris	SHIFT
6	$NP\ (S[dcl]\backslash NP)/NP\ N$		SHIFT
7	$NP\ (S[dcl]\backslash NP)/NP\ NP$		UNARY
8	$NP\ S[dcl]\backslash NP$		REDUCE
9	$S[dcl]$		REDUCE

Figure 1: Deterministic example of shift-reduce CCG parsing (lexical categories omitted on queue).

Figure 1 shows a deterministic example for the sentence *Mr. President visited Paris*, giving a single sequence of shift-reduce actions which produces a correct derivation (i.e. one producing the correct set of dependencies). Starting with the initial item $\langle s, q \rangle_0$ (row 0), which has an empty stack and a full queue, a total of nine actions are applied to produce the complete derivation.

Applying beam-search to a statistical shift-reduce parser is a straightforward extension to the deterministic example. At each step, a beam is used to store the top-k highest-scoring items, resulting from expanding all items in the previous beam. An item becomes a candidate output once it has an empty queue, and the parser keeps track of the highest scored candidate output and returns the best one as the final output. Compared with greedy local-search (Nivre and Scholz, 2004), the use of a beam allows the parser to explore a larger search space and delay difficult ambiguity-resolving decisions by considering multiple items in parallel.

We refer to the shift-reduce model of Zhang and Clark (2011) as the normal-form model, where the oracle for each sentence specifies a unique sequence of gold-standard actions which produces the corresponding normal-form derivation. No dependency structures are involved at training and test time, except for evaluation. In the next section, we describe a dependency oracle which considers all sequences of actions producing a gold-standard dependency structure to be correct.

[1] See Hockenmaier (2003) and Clark and Curran (2007) for a description of CCG rules.

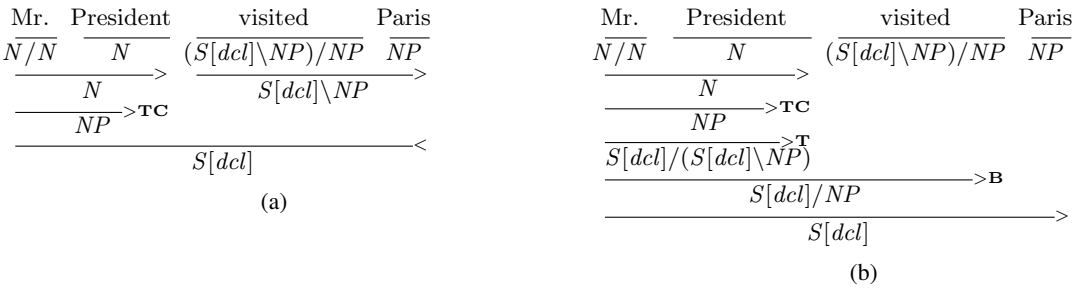


Figure 2: Two derivations leading to the same dependency structure. **TC** denotes type-changing.

3 The Dependency Model

Categories in CCG are either basic (such as NP and PP) or complex (such as $(S[dcl]\backslash NP)/NP$). Each complex category in the lexicon defines one or more predicate-argument relations, which can be realized as a predicate-argument dependency when the corresponding argument slot is consumed. For example, the transitive verb category above defines two relations: one for the subject NP and one for the object NP. In this paper a CCG predicate-argument dependency is a 4-tuple: $\langle h_f, f, s, h_a \rangle$ where h_f is the lexical item of the lexical category expressing the relation; f is the lexical category; s is the argument slot; and h_a is the head word of the argument. Since the lexical items in a dependency are indexed by their sentence positions, all dependencies for a sentence form a set, which is referred to as a CCG *dependency structure*. Clark and Curran (2007) contains a detailed description of dependency structures.

Fig. 2 shows an example demonstrating spurious ambiguity in relation to a CCG dependency structure. In both derivations, the first two lexical categories are combined using forward application ($>$) and the following dependency is realized: $\langle \textit{Mr.}, N/N_1, 1, \textit{President} \rangle$. In the normal-form derivation (a), the dependency $\langle \textit{visited}, (S\backslash NP_1)/NP_2, 2, \textit{Paris} \rangle$ is created by combining the transitive verb category with the object NP using forward application. One final dependency, $\langle \textit{visited}, (S\backslash NP_1)/NP_2, 1, \textit{President} \rangle$, is realized when the root node $S[dcl]$ is produced through backward application ($<$).

Fig. 2(b) shows a non-normal-form derivation which uses type-raising (**T**) and composition (**B**) (which are not required to derive the correct dependency structure). In this alternative derivation, the dependency $\langle \textit{visited}, (S\backslash NP_1)/NP_2, 1, \textit{President} \rangle$ is realized using forward composition (**B**), and $\langle \textit{visited}, (S\backslash NP_1)/NP_2, 2, \textit{Paris} \rangle$ is realized when the $S[dcl]$ root is produced.

The chart-based dependency model of Clark and Curran (2007) treats all derivations as hidden, and defines a probabilistic model for a dependency structure by summing probabilities of all derivations leading to a particular structure. Features are defined over both derivations and CCG predicate-argument dependencies. We follow a similar approach, but rather than define a probabilistic model (which requires summing), we define a linear model over sequences of shift-reduce actions, as for the normal-form shift-reduce model. However, the difference compared to the normal-form model is that we do not assume a single gold-standard sequence of actions.

Similar to Goldberg and Nivre (2012), we define an *oracle* which determines, for a gold-standard dependency structure, G, what the valid transition sequences are (i.e. those sequences corresponding to derivations leading to G). More specifically, the oracle can determine, given G and an item $\langle s, q \rangle$, what the valid actions are for that item (i.e. what actions can potentially lead to G, starting with $\langle s, q \rangle$ and the dependencies already built on s). However, there can be exponentially many valid action sequences for G, which we represent efficiently using a packed parse forest. We show how the forest can be used, during beam-search decoding, to determine the valid actions for a parse item (Section 3.2). We also show, in Section 3.3, how perceptron training with early-update (Collins and Roark, 2004) can be used in this setting.

3.1 The Oracle Forest

A CCG parse forest efficiently represents an exponential number of derivations. Following Clark and Curran (2007) (which builds on Miyao and Tsujii (2002)), and using the same notation, we define a CCG parse forest Φ as a tuple $\langle C, D, R, \gamma, \delta \rangle$, where C is a set of conjunctive

Algorithm 1 (Clark and Curran, 2007)

Input: A packed forest $\langle C, D, R, \gamma, \delta \rangle$, with $dmax(c)$ and $dmax(d)$ already computed

1: **function** MAIN
2: **for each** $d_r \in R$ s.t. $dmax(d_r) = |G|$ **do**
3: MARK(d_r)
4: **procedure** MARK(d)
5: mark d as a correct node
6: **for each** $c \in \gamma(d)$ **do**
7: **if** $dmax(c) == dmax(d)$ **then**
8: mark c as a correct node
9: **for each** $d' \in \delta(c)$ **do**
10: **if** d' has not been visited **then**
11: MARK(d')

nodes and D is a set of disjunctive nodes.[2] Conjunctive nodes are individual CCG categories in Φ, and are either obtained from the lexicon, or by combining two disjunctive nodes using a CCG rule, or by applying a unary rule to a disjunctive node. Disjunctive nodes are equivalence classes of conjunctive nodes. Two conjunctive nodes are equivalent iff they have the same category, head and unfilled dependencies (i.e. they will lead to the same derivation, and produce the same dependencies, in any future parsing). $R \subseteq D$ is a set of root disjunctive nodes. $\gamma : D \rightarrow 2^C$ is the conjunctive child function and $\delta : C \rightarrow 2^D$ is the disjunctive child function. The former returns the set of all conjunctive nodes of a disjunctive node, and the latter returns the disjunctive child nodes of a conjunctive node.

The dependency model requires all the conjunctive and disjunctive nodes of Φ that are part of the derivations leading to a gold-standard dependency structure G. We refer to such derivations as *correct* derivations and the packed forest containing all these derivations as the *oracle forest*, denoted as Φ_G, which is a subset of Φ. It is prohibitive to enumerate all correct derivations, but it is possible to identify, from Φ, all the conjunctive and disjunctive nodes that are part of Φ_G. Clark and Curran (2007) gives an algorithm for doing so, which we use here. The main intuition behind the algorithm is that a gold-standard dependency structure decomposes over derivations; thus gold-standard dependencies realized at conjunctive nodes can be counted when Φ is built, and all nodes that are part of Φ_G can then be marked out of Φ by traversing it top-down. A key idea in understanding the algorithm is that dependencies are created when disjunctive nodes are combined, and hence are associated with, or "live on", conjunctive nodes in the forest.

Following Clark and Curran (2007), we also define the following three values, where the first decomposes only over local rule productions, while the other two decompose over derivations:

$$cdeps(c) = \begin{cases} * & \text{if } \exists\, \tau \in deps(c), \tau \notin G \\ |deps(c)| & \text{otherwise} \end{cases}$$

$$dmax(c) = \begin{cases} * & \text{if } cdeps(c) == * \\ * & \text{if } dmax(d) == * \text{ for some } d \in \delta(c) \\ \sum_{d \in \delta(c)} dmax(d) + cdeps(c) & \text{otherwise} \end{cases}$$

$$dmax(d) = \max\{dmax(c) \mid c \in \gamma(d)\}$$

$deps(c)$ is the set of all dependencies on conjunctive node c, and $cdeps(c)$ counts the number of *correct* dependencies on c. $dmax(c)$ is the maximum number of correct dependencies over any sub-derivation headed by c and is calculated recursively; $dmax(d)$ returns the same value for a disjunctive node. In all cases, a special value $*$ indicates the presence of incorrect dependencies. To obtain the oracle forest, we first pre-compute $dmax(c)$ and $dmax(d)$ for all d and c in Φ when Φ is built using CKY, which are then used by Algorithm 1 to identify all the conjunctive and disjunctive nodes in Φ_G.

3.2 The Dependency Oracle Algorithm

We observe that the canonical shift-reduce algorithm (as demonstrated in Fig. 1) applied to a single parse tree exactly resembles bottom-up *post*-order traversal of that tree. As an example, consider the derivation in Fig. 2a, where the corresponding sequence of actions is: sh N/N, sh N, re N, un NP, sh $(S[dcl]\backslash NP)/NP$, sh NP, re $S[dcl]\backslash NP$, re $S[dcl]$.[3] The order of traversal is left-child, right-child and parent. For a single parse, the corresponding shift-reduce action sequence is unique, and for a given item this canonical order restricts the possible derivations that can be formed using further actions. We now extend this observation to the more general case of an oracle forest, where there may be more than one gold-standard action for a given item.

Definition 1. Given a gold-standard dependency

[2] Under the hypergraph framework (Gallo et al., 1993; Huang and Chiang, 2005), a conjunctive node corresponds to a hyperedge and a disjunctive node corresponds to the head of a hyperedge or hyperedge bundle.

[3] The derivation is "upside down", following the convention used for CCG, where the root is $S[dcl]$. We use sh, re and un to denote the three types of shift-reduce action.

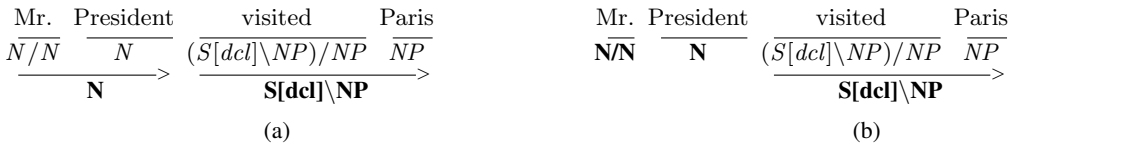


Figure 3: Example subtrees on two stacks, with two subtrees in (a) and three in (b); roots of subtrees are in bold.

structure G, an oracle forest Φ_G, and an item $\langle s, q\rangle$, we say s is a **realization** of G, denoted $s \simeq G$, if $|s| = 1$, q is empty and the single derivation on s is correct. If $|s| > 0$ and the subtrees on s can lead to a correct derivation in Φ_G using further actions, we say s is a **partial-realization** of G, denoted as $s \sim G$. And we define $s \sim G$ for $|s| = 0$.

As an example, assume that Φ_G contains only the derivation in Fig. 2a; then a stack containing the two subtrees in Fig. 3a is a partial-realization, while a stack containing the three subtrees in Fig. 3b is not. Note that each of the three subtrees in Fig. 3b is present in Φ_G; however, these subtrees cannot be combined into the single correct derivation, since the correct sequence of shift-reduce actions must first combine the lexical categories for *Mr.* and *President* before shifting the lexical category for *visited*.

We denote an action as a pair (x, c), where $x \in \{\text{SHIFT}, \text{REDUCE}, \text{UNARY}\}$ and c is the root of the subtree resulting from that action. For all three types of actions, c also corresponds to a unique conjunctive node in the *complete* forest Φ; and we use c_{s_i} to denote the conjunctive node in Φ corresponding to subtree s_i on the stack. Let $\langle s', q'\rangle = \langle s, q\rangle \circ (x, c)$ be the resulting item from applying the action (x, c) to $\langle s, q\rangle$; and let the set of all possible actions for $\langle s, q\rangle$ be $\mathcal{X}_{\langle s,q\rangle} = \{(x, c) \mid (x, c) \text{ is applicable to } \langle s, q\rangle\}$.

Definition 2. Given Φ_G and an item $\langle s, q\rangle$ s.t. $s \sim G$, we say an applicable action (x, c) for the item is **valid** iff $s' \sim G$ or $s' \simeq G$, where $\langle s', q'\rangle = \langle s, q\rangle \circ (x, c)$.

Definition 3. Given Φ_G, the dependency oracle function f_d is defined as:

$$f_d(\langle s, q\rangle, (x, c), \Phi_G) = \begin{cases} \textit{true} & \text{if } s' \sim G \text{ or } s' \simeq G \\ \textit{false} & \text{otherwise} \end{cases}$$

where $(x, c) \in \mathcal{X}_{\langle s,q\rangle}$ and $\langle s', q'\rangle = \langle s, q\rangle \circ (x, c)$.

The pseudocode in Algorithm 2 implements f_d. It determines, for a given item, whether an applicable action is valid in Φ_G.

It is trivial to determine the validity of a SHIFT action for the initial item, $\langle s, q\rangle_0$, since the SHIFT action is valid iff its category matches the gold-standard lexical category of the first word in the sentence. For any subsequent SHIFT action (SHIFT, c) to be valid, the *necessary* condition is $c \equiv c_{lex_0}$, where c_{lex_0} denotes the gold-standard lexical category of the front word in the queue, q_0 (line 3). However, this condition is not sufficient; a counterexample is the case where all the gold-standard lexical categories for the sentence in Figure 2 are shifted in succession. Hence, in general, the conditions under which an action is valid are more complex than the trivial case above.

First, suppose there is only one correct derivation in Φ_G. A SHIFT action $(\text{SHIFT}, c_{lex_0})$ is valid whenever c_{s_0} (the conjunctive node in Φ_G corresponding to the subtree s_0 on the stack) and c_{lex_0} (the conjunctive node in Φ_G corresponding to the next gold-standard lexical category from the queue) are both dominated by the conjunctive node parent p of c_{s_0} in Φ_G.[4] A REDUCE action (REDUCE, c) is valid if c matches the category of the conjunctive node parent of c_{s_0} and c_{s_1} in Φ_G. A UNARY action (UNARY, c) is valid if c matches the conjunctive node parent of c_{s_0} in Φ_G. We now generalize the case where Φ_G contains a single correct parse to the case of an oracle forest, where each parent p is replaced by a set of conjunctive nodes in Φ_G.

Definition 4. The **left parent set** $p_L(c)$ of conjunctive node $c \in \Phi_G$ is the set of all parent conjunctive nodes of c in Φ_G, which have the disjunctive node d containing c (i.e. $c \in \gamma(d)$) as a left child.

Definition 5. The **ancestor set** $\mathcal{A}(c)$ of conjunctive node $c \in \Phi_G$ is the set of all reachable ancestor conjunctive nodes of c in Φ_G.

Definition 6. Given an item $\langle s, q\rangle$, if $|s| = 1$ we say s is a **frontier stack**.

[4] Strictly speaking, the conjunctive node parent is a parent of the disjunctive node containing the conjunctive node c_{s_0}. We will continue to use this shorthand for parents of conjunctive nodes throughout the paper.

Algorithm 2 The Dependency Oracle Function f_d

Input: Φ_G, an item $\langle s, q\rangle$ s.t. $s \sim G$, $(x, c) \in \mathcal{X}_{\langle s,q\rangle}$
Let s' be the stack of $\langle s', q'\rangle = \langle s, q\rangle \circ (x, c)$

function MAIN($\langle s, q\rangle$, (x, c), Φ_G)
  if x is SHIFT **then**
    if $c \not\equiv c_{lex_0}$ **then** ▷ c not gold lexical category
      return false
    else if $c \equiv c_{lex_0}$ and $|s| = 0$ **then** ▷ the initial item
      return true
    else if $c \equiv c_{lex_0}$ and $|s| \neq 0$ **then**
      compute $\mathcal{R}(c_{s'_1}, \mathbf{c}_{s'_0})$
      return $\mathcal{R}(c_{s'_1}, \mathbf{c}_{s'_0}) \neq \emptyset$

  if x is REDUCE **then** ▷ s is non-frontier
    if $c \in \mathcal{R}(c_{s_1}, \mathbf{c}_{s_0})$ **then**
      compute $\mathcal{R}(c_{s'_1}, \mathbf{c}_{s'_0})$
      return true
    else return false

  if x is UNARY **then**
    if $|s| = 1$ **then** ▷ s is frontier
      return $c \in \Phi_G$
    if $|s| \neq 1$ **and** $c \in \Phi_G$ **then** ▷ s is non-frontier
      compute $\mathcal{R}(c_{s'_1}, \mathbf{c}_{s'_0})$
      return $\mathcal{R}(c_{s'_1}, \mathbf{c}_{s'_0}) \neq \emptyset$

A key to defining the dependency oracle function is the notion of a **shared ancestor set**. Intuitively, shared ancestor sets are built up through shift actions, and contain sets of nodes which can potentially become the results of reduce or unary actions. A further intuition is that shared ancestor sets define the space of possible correct derivations, and nodes in these sets are "ticked off" when reduce and unary actions are applied, as a single correct derivation is built through the shift-reduce process (corresponding to a bottom-up post-order traversal of the derivation). The following definition shows how the dependency oracle function builds shared ancestor sets for each action type.

Definition 7. Let $\langle s, q\rangle$ be an item and let $\langle s', q'\rangle = \langle s, q\rangle \circ (x, c)$. We define the **shared ancestor set** $\mathcal{R}(c_{s'_1}, \mathbf{c}_{s'_0})$ of $\mathbf{c}_{s'_0}$, after applying action (x, c), as:

- $\{c' \mid c' \in p_L(c_{s_0}) \cap \mathcal{A}(c)\}$, if s is frontier and $x =$ SHIFT
- $\{c' \mid c' \in p_L(c_{s_0}) \cap \mathcal{A}(c)$ and there is some $c'' \in \mathcal{R}(c_{s_1}, \mathbf{c}_{s_0})$ s.t. $c'' \in \mathcal{A}(c')\}$, if s is non-frontier and $x =$ SHIFT
- $\{c' \mid c' \in \mathcal{R}(c_{s_2}, \mathbf{c}_{s_1}) \cap \mathcal{A}(c)\}$, if $x =$ REDUCE
- $\{c' \mid c' \in \mathcal{R}(c_{s_1}, \mathbf{c}_{s_0}) \cap \mathcal{A}(c)\}$, if s is non-frontier and $x =$ UNARY
- $\mathcal{R}(\epsilon, \mathbf{c}^0_{s_0}) = \emptyset$ where $\mathbf{c}^0_{s_0}$ is the conjunctive node corresponding to the gold-standard lexical category of the first word in the sentence (ϵ is a dummy symbol indicating the bottom of stack).

The base case for Definition 7 is when the gold-standard lexical category of the first word in the sentence has been shifted, which creates an empty shared ancestor set. Furthermore, the shared ancestor set is always empty when the stack is a frontier stack.

The dependency oracle algorithm checks the validity of applicable actions. A SHIFT action is valid if $\mathcal{R}(c_{s'_1}, \mathbf{c}_{s'_0}) \neq \emptyset$ for the resulting stack s'. A valid REDUCE action consumes s_1 and s_0. For the new node, its shared ancestor set is the subset of the conjunctive nodes in $\mathcal{R}(c_{s_2}, \mathbf{c}_{s_1})$ which dominate the resulting conjunctive node of a valid REDUCE action. The UNARY case for a frontier stack is trivial: any UNARY action applicable to s in Φ_G is valid. For a non-frontier stack, the UNARY case is similar to REDUCE except the resulting shared ancestor set is a subset of $\mathcal{R}(c_{s_1}, \mathbf{c}_{s_0})$.

We now turn to the problem of finding the shared ancestor sets. In practice, we do not do this by traversing Φ_G top-down from the conjunctive nodes in $p_L(c_{s_0})$ on-the-fly to find each member of $\mathcal{R}$. Instead, when we build Φ_G in bottom-up topological order, we pre-compute the set of reachable disjunctive nodes of each conjunctive node c in Φ_G as:

$$\mathcal{D}(c) = \delta(c) \cup (\cup_{c' \in \gamma(d), d \in \delta(c)}(\mathcal{D}(c')))$$

Each $\mathcal{D}$ is implemented as a hash map, which allows us to test the membership of one potential conjunctive node in $\mathcal{O}(1)$ time. For example, a conjunctive node $c \in p_L(c_{s_0})$ is reachable from c_{lex_0} if there is a *disjunctive node* $d \in \mathcal{D}(c)$ s.t. $c_{lex_0} \in \gamma(d)$. With this implementation, the complexity of checking each valid SHIFT action is then $\mathcal{O}(|p_L(c_{s_0})|)$.

3.3 Training

We use the averaged perceptron (Collins, 2002) to train a global linear model and score each action. The normal-form model of Zhang and Clark (2011) uses an early update mechanism (Collins and Roark, 2004), where decoding is stopped to update model weights whenever the single gold action falls outside the beam. In our parser, there can be multiple gold items in a beam. One option would be to apply early update whenever at least

Algorithm 3 Dependency Model Training

Input: (y, G) and beam size k
1: $\mathbf{w} \leftarrow \mathbf{0}$; $\mathcal{B}_0 \leftarrow \emptyset$; $i \leftarrow 0$
2: $\mathcal{B}_0$.push($\langle s, q \rangle_0$) ▷ the initial item
3: $cand \leftarrow \emptyset$ ▷ candidate output priority queue
4: $gold \leftarrow \emptyset$ ▷ gold output priority queue
5: **while** $\mathcal{B}_i \neq \emptyset$ **do**
6: **for each** $\langle s, q \rangle \in \mathcal{B}_i$ **do**
7: **if** $|q| = 0$ **then** ▷ candidate output
8: $cand$.push($\langle s, q \rangle$)
9: **if** $s \simeq G$ **then** ▷ s is a realization of G
10: $gold$.push($\langle s, q \rangle$)
11: expand $\langle s, q \rangle$ into $\mathcal{B}_{i+1}$
12: $\mathcal{B}_{i+1} \leftarrow \mathcal{B}_{i+1}[1:k]$ ▷ apply beam
13: **if** $\Pi_G \neq \emptyset, \Pi_G \cap \mathcal{B}_{i+1} = \emptyset$ **and** $cand[0] \not\simeq G$ **then**
14: $\mathbf{w} \leftarrow \mathbf{w} + \phi(\Pi_G[0]) - \phi(\mathcal{B}_{i+1}[0])$ ▷ early update
15: **return**
16: $i \leftarrow i + 1$ ▷ continue to next step
17: **if** $cand[0] \not\simeq G$ **then** ▷ final update
18: $\mathbf{w} \leftarrow \mathbf{w} + \phi(gold[0]) - \phi(cand[0])$

one of these gold items falls outside the beam. However, this may not be a true violation of the gold-standard (Huang et al., 2012). Thus, we use a relaxed version of early update, in which *all* gold-standard actions must fall outside the beam before an update is performed. This update mechanism is provably correct under the violation-fixing framework of Huang et al. (2012).

Let (y, G) be a training sentence paired with its gold-standard dependency structure and let $\Pi_{\langle s,q \rangle}$ be the following set for an item $\langle s, q \rangle$:

$$\{\langle s, q \rangle \circ (x, c) \mid f_d(\langle s, q \rangle, (x, c), \Phi_G) = true\}$$

$\Pi_{\langle s,q \rangle}$ contains all *correct* items at step $i + 1$ obtained by expanding $\langle s, q \rangle$. Let the set of *all* correct items at a step $i + 1$ be:[5]

$$\Pi_G = \bigcup_{\langle s,q \rangle \in \mathcal{B}_i} \Pi_{\langle s,q \rangle}$$

Algorithm 3 shows the pseudocode for training the dependency model with early update for one input (y, G). The score of an item $\langle s, q \rangle$ is calculated as $\mathbf{w} \cdot \phi(\langle s, q \rangle)$ with respect to the current model $\mathbf{w}$, where $\phi(\langle s, q \rangle)$ is the feature vector for the item. At step i, all items are expanded and added onto the next beam $\mathcal{B}_{i+1}$, and the top-k retained. Early update is applied when all gold items first fall outside the beam, and any candidate output is incorrect (line 14). Since there are potentially many gold items, and one gold item is required for the perceptron update, a decision needs to be made regarding which gold item to update against. We choose to reward the highest scoring gold item, in line with the violation-fixing framework; and penalize the highest scoring incorrect item, using the standard perceptron update. A final update is performed if no more expansions are possible but the final output is incorrect.

[5] In Algorithm 3 we abuse notation by using $\Pi_G[0]$ to denote the highest scoring gold item in the set.

4 Experiments

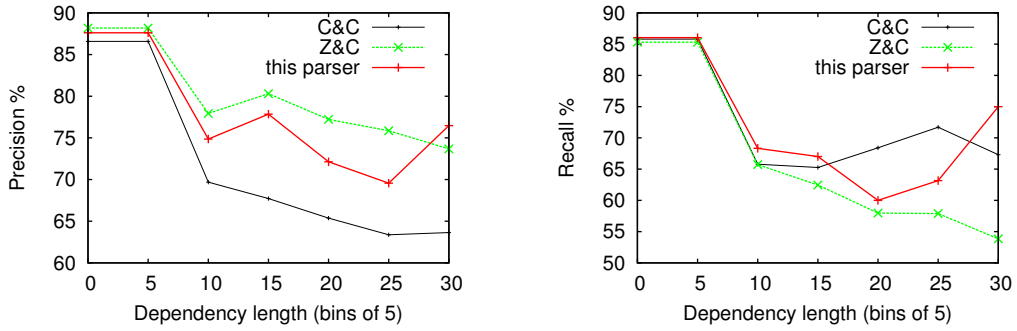
We implement our shift-reduce parser on top of the core C&C code base (Clark and Curran, 2007) and evaluate it against the shift-reduce parser of Zhang and Clark (2011) (henceforth Z&C) and the chart-based normal-form and hybrid models of Clark and Curran (2007). For all experiments, we use CCGBank with the standard split: sections 2-21 for training (39,604 sentences), section 00 for development (1,913 sentences) and section 23 (2,407 sentences) for testing.

The way that the CCG grammar is implemented in C&C has some implications for our parser. First, unlike Z&C, which uses a context-free cover (Fowler and Penn, 2010) and hence is able to use all sentences in the training data, we are only able to use 36,036 sentences. The reason is that the grammar in C&C does not have complete coverage of CCGBank, due to the fact that e.g. not all rules in CCGBank conform to the combinatory rules of CCG. Second, our parser uses the unification mechanism from C&C to output dependencies directly, and hence does not need a separate post-processing step to convert derivations into CCG dependencies, as required by Z&C.

The feature templates of our model consist of all of those in Z&C, except the ones which require lexical heads to come from either the left or right child, as such features are incompatible with the head passing mechanism used by C&C. Each Z&C template is defined over a parse item, and captures various aspects of the stack and queue context. For example, one template returns the top category on the stack plus its head word, together with the first word and its POS tag on the queue. Another template returns the second category on the stack, together with the POS tag of its head word. Every Z&C feature is defined as a pair, consisting of an instantiated context template and a parse action. In addition, we use all the CCG predicate-argument dependency features from Clark and Curran (2007), which contribute to the score of a REDUCE action when dependencies

	LP %	LR %	LF %	LSent. %	CatAcc. %	coverage %
this parser	86.29	**84.09**	**85.18**	**34.40**	92.75	100
Z&C	**87.15**	82.95	85.00	33.82	**92.77**	100
C&C (normal-form)	85.22	82.52	83.85	31.63	92.40	100
this parser	86.76	**84.90**	**85.82**	**34.72**	**93.20**	99.06 (C&C coverage)
Z&C	**87.55**	83.63	85.54	34.14	93.11	99.06 (C&C coverage)
C&C (hybrid)	–	–	85.25	–	–	99.06 (C&C coverage)
C&C (normal-form)	85.22	84.29	84.76	31.93	92.83	99.06 (C&C coverage)

Table 1: Accuracy comparison on Section 00 (auto POS).



(a) precision vs. dependency length

(b) recall vs. dependency length

Figure 4: Labeled precision and recall relative to dependency length on the development set. C&C normal-form model is used.

are realized. Detailed descriptions of all the templates in our model can be found in the respective papers. We run 20 training iterations and the resulting model contains 16.5M features with a non-zero weight.

We use 10-fold cross validation for POS tagging and supertagging the training data, and automatically assigned POS tags for all experiments. A probability cut-off value of 0.0001 for the β parameter in the supertagger is used for both training and testing. The β parameter determines how many lexical categories are assigned to each word; $\beta = 0.0001$ is a relatively small value which allows in a large number of categories, compared to the default value used in Clark and Curran (2007). For training only, if the gold-standard lexical category is not supplied by the supertagger for a particular word, it is added to the list of categories.

4.1 Results and Analysis

The beam size was tuned on the development set, and a value of 128 was found to achieve a reasonable balance of accuracy and speed; hence this value was used for all experiments. Since C&C always enforces non-fragmentary output (i.e. it can only produce spanning analyses), it fails on some sentences in the development and test sets, and thus we also evaluate on the reduced sets, following Clark and Curran (2007). Our parser does not fail on any sentences because it permits fragmentary output (those cases where there is more than one subtree left on the final stack). The results for Z&C, and the C&C normal-form and hybrid models, are taken from Zhang and Clark (2011).

Table 1 shows the accuracies of all parsers on the development set, in terms of labeled precision and recall over the predicate-argument dependencies in CCGBank. On both the full and reduced sets, our parser achieves the highest F-score. In comparison with C&C, our parser shows significant increases across all metrics, with 0.57% and 1.06% absolute F-score improvements over the hybrid and normal-form models, respectively. Another major improvement over the other two parsers is in sentence level accuracy, LSent, which measures the number of sentences for which the dependency structure is completely correct.

Table 1 also shows that our parser has improved recall over Z&C at some expense of precision. To probe this further we compare labeled precision and recall relative to dependency length, as measured by the distance between the two words in a dependency, grouped into bins of 5 values. Fig. 4 shows clearly that Z&C favors precision over recall, giving higher precision scores for almost all dependency lengths compared to our parser. In

category	LP % (o)	LP % (z)	LP % (c)	LR % (o)	LR % (z)	LR % (c)	LF % (o)	LF % (z)	LF % (c)	freq.
N/**N**	95.53	**95.77**	95.28	**95.83**	95.79	95.62	95.68	**95.78**	95.45	7288
NP/**N**	96.53	**96.70**	96.57	**97.12**	96.59	96.03	**96.83**	96.65	96.30	4101
$(NP\backslash NP)$/**NP**	81.64	**83.19**	82.17	**90.63**	89.24	88.90	85.90	**86.11**	85.40	2379
$(NP\backslash$**NP**$)/NP$	81.70	**82.53**	81.58	**88.91**	87.99	85.74	85.15	**85.17**	83.61	2174
$((S\backslash NP)\backslash(S\backslash NP))$/**NP**	**77.64**	77.60	71.94	72.97	71.58	**73.32**	**75.24**	74.47	72.63	1147
$((S\backslash NP)\backslash($**S**$\backslash$**NP**$))/NP$	75.78	**76.30**	70.92	71.27	70.60	**71.93**	**73.45**	73.34	71.42	1058
$((S[dcl]\backslash NP)$/**NP**	83.94	**85.60**	81.57	86.04	84.30	**86.37**	**84.98**	84.95	83.90	917
PP/**NP**	**77.06**	73.76	75.06	**73.63**	72.83	70.09	**75.31**	73.29	72.49	876
$((S[dcl]\backslash$**NP**$)/NP$	82.03	**85.32**	81.62	83.26	82.00	**85.55**	82.64	**83.63**	83.54	872
$((S\backslash NP)\backslash($**S**$\backslash$**NP**$))$	86.42	84.44	**86.85**	86.19	86.60	**86.73**	86.31	85.51	**86.79**	746

Table 2: Accuracy comparison on most frequent dependency types, for our parser (o), Z&C (z) and C&C hybrid model (c). Categories in bold indicate the argument slot in the relation.

	LP %	LR %	LF %	LSent. %	CatAcc. %	coverage %
our parser	87.03	**85.08**	**86.04**	**35.69**	93.10	100
Z&C	**87.43**	83.61	85.48	35.19	**93.12**	100
C&C (normal-form)	85.58	82.85	84.20	32.90	92.84	100
our parser	87.04	**85.16**	**86.09**	**35.84**	93.13	99.58 (C&C coverage)
Z&C	**87.43**	83.71	85.53	35.34	**93.15**	99.58 (C&C coverage)
C&C (hybrid)	86.17	84.74	85.45	32.92	92.98	99.58 (C&C coverage)
C&C (normal-form)	85.48	84.60	85.04	33.08	92.86	99.58 (C&C coverage)

Table 3: Accuracy comparison on section 23 (auto POS).

terms of recall (Fig. 4b), our parser outperforms Z&C over all dependency lengths, especially for longer dependencies ($x \geq 20$). When compared with C&C, the recall of the Z&C parser drops quickly for dependency lengths over 10. While our parser also suffers from this problem, it is less severe and is able to achieve higher recall at $x \geq 30$.

Table 2 compares our parser with Z&C and the C&C hybrid model, for the most frequent dependency relations. While our parser achieved lower precision than Z&C, it is more balanced and gives higher recall for all of the dependency relations except the last one, and higher F-score for over half of them.

Table 3 presents the final test results on Section 23. Again, our parser achieves the highest scores across all metrics (for both the full and reduced test sets), except for precision and lexical category assignment, where Z&C performed better.

5 Conclusion

We have presented a dependency model for a shift-reduce CCG parser, which fully aligns CCG parsing with the left-to-right, incremental nature of a shift-reduce parser. Our work is in part inspired by the dependency models of Clark and Curran (2007) and, in the use of a dependency oracle, is close in spirit to that of Goldberg and Nivre (2012). The difference is that the Goldberg and Nivre parser builds, and scores, dependency structures directly, whereas our parser uses a unification mechanism to create dependencies, and scores the CCG derivations, allowing great flexibility in terms of what dependencies can be realized. Another related work is Yu et al. (2013), which introduced a similar technique to deal with spurious ambiguity in MT. Finally, there may be potential to integrate the techniques of Auli and Lopez (2011), which currently represents the state-of-the-art in CCGBank parsing, into our parser.

Acknowledgements

We thank the anonymous reviewers for their helpful comments. Wenduan Xu is fully supported by the Carnegie Trust and receives additional funding from the Cambridge Trusts. Stephen Clark is supported by ERC Starting Grant DisCoTex (306920) and EPSRC grant EP/I037512/1. Yue Zhang is supported by Singapore MOE Tier2 grant T2MOE201301.

References

Michael Auli and Adam Lopez. 2011. A comparison of loopy belief propagation and dual decomposition for integrated CCG supertagging and parsing. In *Proc. ACL 2011*, pages 470–480, Portland, OR.

Ted Briscoe and John Carroll. 2006. Evaluating the accuracy of an unlexicalized statistical parser on the

PARC DepBank. In *Proc. of COLING/ACL*, pages 41–48, Sydney, Australia.

Stephen Clark and James R. Curran. 2006. Partial training for a lexicalized-grammar parser. In *Proc. NAACL-06*, pages 144–151, New York, USA.

Stephen Clark and James R. Curran. 2007. Wide-coverage efficient statistical parsing with CCG and log-linear models. *Computational Linguistics*, 33(4):493–552.

Stephen Clark and Julia Hockenmaier. 2002. Evaluating a wide-coverage CCG parser. In *Proc. of the LREC 2002 Beyond Parseval Workshop*, pages 60–66, Las Palmas, Spain.

Stephen Clark, Julia Hockenmaier, and Mark Steedman. 2002. Building deep dependency structures with a wide-coverage CCG parser. In *Proc. ACL*, pages 327–334, Philadelphia, PA.

Michael Collins and Brian Roark. 2004. Incremental parsing with the perceptron algorithm. In *Proc. of ACL*, pages 111–118, Barcelona, Spain.

Michael Collins. 2002. Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms. In *Proc. of EMNLP*, pages 1–8, Philadelphia, USA.

Jason Eisner. 1996. Efficient normal-form parsing for Combinatory Categorial Grammar. In *Proc. ACL*, pages 79–86, Santa Cruz, CA.

Timothy AD Fowler and Gerald Penn. 2010. Accurate context-free parsing with Combinatory Categorial Grammar. In *Proc. ACL*, pages 335–344, Uppsala, Sweden.

Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. 1993. Directed hypergraphs and applications. *Discrete applied mathematics*, 42(2):177–201.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *Proc. COLING*, Mumbai, India.

Yoav Goldberg, Kai Zhao, and Liang Huang. 2013. Efficient implementation for beam search incremental parsers. In *Proceedings of the Short Papers of ACL*, Sofia, Bulgaria.

Julia Hockenmaier and Mark Steedman. 2007. CCGbank: A corpus of CCG derivations and dependency structures extracted from the Penn Treebank. *Computational Linguistics*, 33(3):355–396.

Julia Hockenmaier. 2003. *Data and Models for Statistical Parsing with Combinatory Categorial Grammar*. Ph.D. thesis, University of Edinburgh.

Liang Huang and David Chiang. 2005. Better k-best parsing. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 53–64, Vancouver, Canada.

Liang Huang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In *Proc. ACL*, pages 1077–1086, Uppsala, Sweden.

Liang Huang, Suphan Fayong, and Yang Guo. 2012. Structured perceptron with inexact search. In *Proc. NAACL*, pages 142–151, Montreal, Canada.

Yusuke Miyao and Jun'ichi Tsujii. 2002. Maximum entropy estimation for feature forests. In *Proceedings of the Human Language Technology Conference*, San Diego, CA.

Joakim Nivre and Ryan McDonald. 2008. Integrating graph-based and transition-based dependency parsers. In *Proc. of ACL/HLT*, pages 950–958, Columbus, Ohio.

J. Nivre and M Scholz. 2004. Deterministic dependency parsing of English text. In *Proceedings of COLING 2004*, pages 64–70, Geneva, Switzerland.

Joakim Nivre, Laura Rimell, Ryan McDonald, and Carlos Gomez-Rodriguez. 2010. Evaluation of dependency parsers on unbounded dependencies. In *Proc. of COLING*, Beijing, China.

Laura Rimell, Stephen Clark, and Mark Steedman. 2009. Unbounded dependency recovery for parser evaluation. In *Proc. EMNLP*, pages 813–821, Edinburgh, UK.

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. In *Proc. of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, Sofia, Bulgaria.

Mark Steedman. 2000. *The Syntactic Process*. The MIT Press, Cambridge, Mass.

H Yamada and Y Matsumoto. 2003. Statistical dependency analysis using support vector machines. In *Proc. of IWPT*, Nancy, France.

Heng Yu, Liang Huang, Haitao Mi, and Kai Zhao. 2013. Max-violation perceptron and forced decoding for scalable mt training. In *Proc. EMNLP, Seattle, Washington, USA*.

Yue Zhang and Stephen Clark. 2008. A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In *Proc. of EMNLP*, Hawaii, USA.

Yue Zhang and Stephen Clark. 2011. Shift-reduce CCG parsing. In *Proc. ACL 2011*, pages 683–692, Portland, OR.