Transforming Projective Bilexical Dependency Grammars into efficiently-parsable CFGs with Unfold-Fold

Mark Johnson
Microsoft Research
Redmond, WA
t-majoh@microsoft.com
Brown University
Providence, RI
Mark_Johnson@Brown.edu

Abstract

This paper shows how to use the Unfold-Fold transformation to transform Projective Bilexical Dependency Grammars (PBDGs) into ambiguity-preserving weakly equivalent Context-Free Grammars (CFGs). These CFGs can be parsed in $O(n^3)$ time using a CKY algorithm with appropriate indexing, rather than the $O(n^5)$ time required by a naive encoding. Informally, using the CKY algorithm with such a CFG mimics the steps of the Eisner-Satta $O(n^3)$ PBDG parsing algorithm. This transformation makes all of the techniques developed for CFGs available to PBDGs. We demonstrate this by describing a maximum posterior parse decoder for PBDGs.

1 Introduction

Projective Bilexical Dependency Grammars (PBDGs) have attracted attention recently for two reasons. First, because they capture bilexical head-to-head dependencies they are capable of producing extremely high-quality parses: state-of-the-art discriminatively trained PBDG parsers rival the accuracy of the very best statistical parsers available today (McDonald, 2006). Second, Eisner-Satta $O(n^3)$ PBDG parsing algorithms are extremely fast (Eisner, 1996; Eisner and Satta, 1999; Eisner, 2000).

This paper investigates the relationship between Context-Free Grammar (CFG) parsing and the Eisner/Satta PBDG parsing algorithms, including their extension to second-order PBDG parsing (McDonald, 2006; McDonald and Pereira, 2006). Specifically, we show how to use an off-line preprocessing step, the Unfold-Fold transformation, to transform a PBDG into an equivalent CFG that can be parsed in $O(n^3)$ time using a version of the CKY algorithm with suitable indexing (Younger, 1967), and extend this transformation so that it captures second-order PBDG dependencies as well. The transformations are ambiguity-preserving, i.e., there is a one-to-one mapping between dependency parses and CFG parses, so it is possible to map the CFG parses back to the PBDG parses they correspond to.

The PBDG to CFG reductions make techniques developed for CFGs available to PBDGs as well. For example, incremental CFG parsing algorithms can be used with the CFGs produced by this transform, as can the Inside-Outside estimation algorithm (Lari and Young, 1990) and more exotic methods such as estimating adjoined hidden states (Matsuzaki et al., 2005; Petrov et al., 2006). As an example application, we describe a maximum posterior parse decoder for PBDGs in Section 8.

The Unfold-Fold transformation is a calculus for transforming functional and logic programs into equivalent but (hopefully) faster programs (Burstall and Darlington, 1977). We use it here to transform CFGs encoding dependency grammars into other CFGs that are more efficiently parsable. Since CFGs can be expressed as Horn-clause logic programs (Pereira and Shieber, 1987) and the Unfold-Fold transformation is provably correct for such programs (Sato, 1992; Pettorossi and Proeitti, 1992), it follows that its application to CFGs is provably correct as well. The Unfold-Fold transformation is used here to derive the CFG schemata presented in sections 5–7. A system that uses these schemata (such as the one described in section 8) can implement

Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics, pages 168–175,
Prague, Czech Republic, June 2007.

these schemata directly, so the Unfold-Fold transformation plays a theoretical role in this work, justifying the resulting CFG schemata.

The closest related work we are aware of is McAllester (1999), which also describes a reduction of PBDGs to efficiently-parsable CFGs and directly inspired this work. However, the CFGs produced by McAllester's transformation include epsilon-productions so they require a specialized CFG parsing algorithm, while the CFGs produced by the transformations described here have binary productions so they can be parsed with standard CFG parsing algorithms. Further, our approach extends to second-order PBDG parsing, while McAllester only discusses first-order PBDGs.

The rest of this paper is structured as follows. Section 2 defines projective dependency graphs and grammars and Section 3 reviews the "naive" encoding of PBDGs as CFGs with an $O(n^5)$ parse time, where n is the length of the string to be parsed. Section 4 introduces the "split-head" CFG encoding of PBDGs, which has an $O(n^4)$ parse time and serves as the input to the Unfold-Fold transform. Section 5 uses the Unfold-Fold transform to obtain a weakly-equivalent CFG encoding of PBDGs which can be parsed in $O(n^3)$ time, and presents timing results showing that the transformation does speed parsing. Sections 6 and 7 apply Unfold-Fold in slightly more complex ways to obtain CFG encodings of PBDGs that also make second-order dependencies available in $O(n^3)$ time parsable CFGs. Section 8 applies a PBDG to CFG transform to obtain a maximum posterior decoding parser for PBDGs.

2 Projective bilexical dependency parses and grammars

Let Σ be a finite set of $terminals$ (e.g., words), and let 0 be the *root terminal* not in Σ. If $w = (w_1, \ldots, w_n) \in \Sigma^\star$, let $w^\star = (0, w_1, \ldots, w_n)$, i.e., $w^\star$ is obtained by prefixing w with 0. A *dependency parse* G for w is a tree whose root is labeled 0 and whose other n vertices are labeled with each of the n terminals in w. If G contains an arc from u to v then we say that v is a *dependent* of u, and if G contains a path from u to v then we say that v is a *descendant* of u. If v is dependent of u that also precedes u in $w^\star$ then we say that v is a *left dependent* of u (right dependent and left and right descendants are defined similarly).

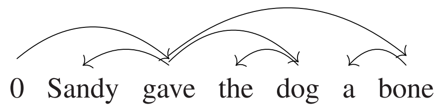

Figure 1: A projective dependency parse for the sentence "Sam gave the dog a bone".

A dependency parse G is *projective* iff whenever there is a path from u to v then there is also a path from u to every word between u and v in $w^\star$ as well. Figure 1 depicts a projective dependency parse for the sentence "Sam gave the dog a bone".

A projective dependency grammar defines a set of projective dependency parses. A *Projective Bilexical Dependency Grammar* (PBDG) consists of two relations $\curvearrowleft$ and $\curvearrowright$, both defined over $(\Sigma \cup \{0\}) \times \Sigma$. A PBDG generates a projective dependency parse G iff $u \curvearrowright v$ for all right dependencies (u, v) in G and $v \curvearrowleft u$ for all left dependencies (u, v) in G. The language generated by a PBDG is the set of strings that have projective dependency parses generated by the grammar. The following dependency grammar generates the dependency parse in Figure 1.

$$
\begin{array}{cc}
0 \curvearrowright \text{gave} & \text{Sandy} \curvearrowleft \text{gave} \\
\text{gave} \curvearrowright \text{dog} & \text{the} \curvearrowleft \text{dog} \\
\text{gave} \curvearrowright \text{bone} & \text{a} \curvearrowleft \text{bone}
\end{array}
$$

This paper does not consider stochastic dependency grammars directly, but see Section 8 for an application involving them. However, it is straightforward to associate weights with dependencies, and since the dependencies are preserved by the transformations, obtain a weighted CFG. Standard methods for converting weighted CFGs to equivalent PCFGs can be used if required (Chi, 1999). Alternatively, one can transform a corpus of dependency parses into a corpus of the corresponding CFG parses, and estimate CFG production probabilities directly from that corpus.

3 A naive encoding of PBDGs

There is a well-known method for encoding a PBDG as a CFG in which each terminal $u \in \Sigma$ is associated with a corresponding nonterminal X_u that expands to u and all of u's descendants. The nonterminals of the naive encoding CFG consist of the start symbol S and symbols X_u for each terminal $u \in \Sigma$, and the productions of the CFG are the instances of the following schemata:

$$
\begin{array}{rcll}
\mathrm{S} & \rightarrow & \mathrm{X}_u & \text{where } 0 \curvearrowright u \\
\mathrm{X}_u & \rightarrow & u & \\
\mathrm{X}_u & \rightarrow & \mathrm{X}_v\, \mathrm{X}_u & \text{where } v \curvearrowleft u \\
\mathrm{X}_u & \rightarrow & \mathrm{X}_u\, \mathrm{X}_v & \text{where } u \curvearrowright v
\end{array}
$$

The dependency annotations associated with each production specify how to interpret a local tree generated by that production, and permit us to map a CFG parse to the corresponding dependency parse. For example, the top-most local tree in Figure 2 was generated by the production $\mathrm{S} \rightarrow \mathrm{X}_{\mathrm{gave}}$, and indicate that in this parse $0 \curvearrowright \mathrm{gave}$.

Given a terminal vocabulary of size m the CFG contains $O(m^2)$ productions, so it is impractical to enumerate all possible productions for even modest vocabularies. Instead productions relevant to a particular sentence are generated on the fly.

The naive encoding CFG in general requires $O(n^5)$ parsing time with a conventional CKY parsing algorithm, since tracking the head annotations u and v multiplies the standard $O(n^3)$ CFG parse time requirements by an additional factor proportional to the $O(n^2)$ productions expanding X_u.

An additional problem with the naive encoding is that the resulting CFG in general exhibits spurious ambiguities, i.e., a single dependency parse may correspond to more than one CFG parse, as shown in Figure 2. Informally, this is because the CFG permits left and the right dependencies to be arbitrarily intermingled.

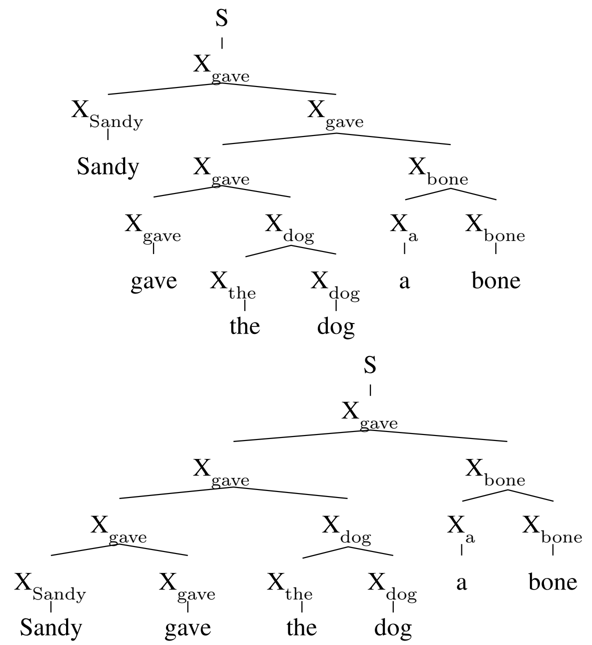

Figure 2: Two parses using the naive CFG encoding that both correspond to the dependency parse of Figure 1.

4 Split-head encoding of PBDGs

There are several ways of removing the spurious ambiguities in the naive CFG encoding just described. This section presents a method we call the "split-head encoding", which removes the ambiguities and serves as starting point for the grammar transforms described below.

The split-head encoding represents each word u in the input string w by *two* unique terminals u_l and u_r in the CFG parse. A split-head CFG's terminal vocabulary is $\Sigma' = \{u_l, u_r : u \in \Sigma\}$, where Σ is the set of terminals of the PBDG. A PBDG parse with yield $w = (u_1, \ldots, u_n)$ is transformed to a split-head CFG parse with yield $w' = (u_{1,l}, u_{1,r}, \ldots, u_{n,l}, u_{n,r})$, so $|w'| = 2|w|$.

The split-head CFG for a PBDG is given by the following schemata:

$$
\begin{array}{rcll}
\mathrm{S} & \rightarrow & \mathrm{X}_u & \text{where } 0 \curvearrowright u \\
\mathrm{X}_u & \rightarrow & \mathrm{L}_u\ {}_u\mathrm{R} & \text{where } u \in \Sigma \\
\mathrm{L}_u & \rightarrow & u_l & \\
\mathrm{L}_u & \rightarrow & \mathrm{X}_v\ \mathrm{L}_u & \text{where } v \curvearrowleft u \\
{}_u\mathrm{R} & \rightarrow & u_r & \\
{}_u\mathrm{R} & \rightarrow & {}_u\mathrm{R}\ \mathrm{X}_v & \text{where } u \curvearrowright v
\end{array}
$$

The dependency parse shown in Figure 1 corresponds to the split-head CFG parse shown in Figure 3. Each X_u expands to two new categories, L_u and ${}_u\mathrm{R}$. L_u consists of u_l and all of u's left descendants, while ${}_u\mathrm{R}$ consists of u_r and all of u's right descendants. The spurious ambiguity present in the naive encoding does not arise in the split-head encoding because the left and right dependents of a head are assembled independently and cannot intermingle.

As can be seen by examining the split-head schemata, the *rightmost* descendant of L_u is either L_u or u_l, which guarantees that the rightmost terminal dominated by L_u is always u_l; similarly the *leftmost* terminal dominated by ${}_u\mathrm{R}$ is always u_r. Thus

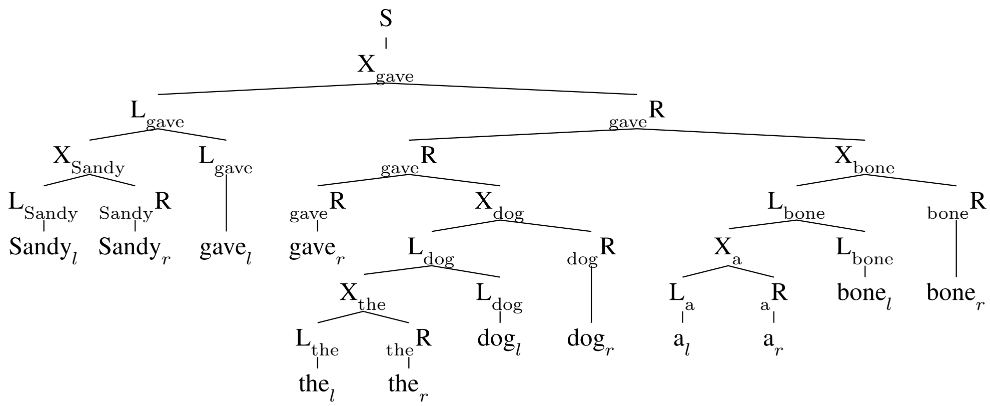

Figure 3: The split-head parse corresponding to the dependency graph depicted in Figure 1. Notice that u_l is always the rightmost descendant of L_u and u_r is always the leftmost descendant of ${}_u\mathrm{R}$, which means that these indices are redundant given the constituent spans.

these subscript indices are redundant given the string positions of the constituents, which means we do not need to track the index u in L_u and ${}_u\mathrm{R}$ but can parse with just the two categories L and R, and determine the index from the constituent's span when required.

It is straight-forward to extend the split-head CFG to encode the additional state information required by the head automata of Eisner and Satta (1999); this corresponds to splitting the non-terminals L_u and ${}_u\mathrm{R}$. For simplicity we work with PBDGs in this paper, but all of the Unfold-Fold transformations described below extend to split-head grammars with the additional state structure required by head automata.

Implementation note: it is possible to directly parse the "undoubled" input string w by modifying both the CKY algorithm and the CFGs described in this paper. Modify L_u and ${}_u\mathrm{R}$ so they both ultimately expand to the same terminal u, and special-case the implementation of production $\mathrm{X}_u \rightarrow \mathrm{L}_u\, {}_u\mathrm{R}$ and all productions derived from it to permit L_u and ${}_u\mathrm{R}$ to overlap by the terminal u.

The split-head formulation explains what initially seem unusual properties of existing PBDG algorithms. For example, one of the standard "sanity checks" for the Inside-Outside algorithm—that the outside probability of each terminal is equal to the sentence's inside probability—fails for these algorithms. In fact, the outside probability of each terminal is *double* the sentence's inside probability because these algorithms implicitly collapse the two terminals u_l and u_r into a single terminal u.

5 A $O(n^3)$ split-head grammar

The split-head encoding described in the previous section requires $O(n^4)$ parsing time because the index v on X_v is not redundant. We can obtain an equivalent grammar that only requires $O(n^3)$ parsing time by transforming the split-head grammar using Unfold-Fold. We describe the transformation on L_u; the transformation of ${}_u\mathrm{R}$ is symmetric.

We begin with the definition of L_u in the split-head grammar above ("|" separates the right-hand sides of productions).

$$\mathrm{L}_u \;\rightarrow\; u_l \mid \mathrm{X}_v\ \mathrm{L}_u \quad \text{where } v \curvearrowleft u$$

Our first transformation step is to unfold X_v in L_u, i.e., replace X_v by its expansion, producing the following definition for L_u (ignore the underlining for now).

$$\mathrm{L}_u \;\rightarrow\; u_l \mid \mathrm{L}_v\ \underline{{}_v\mathrm{R}\ \mathrm{L}_u} \quad \text{where } v \curvearrowleft u$$

This removes the offending X_v in L_u, but the resulting definition of L_u contains ternary productions and so still incurs $O(n^4)$ parse time. To address this we define new nonterminals ${}_x\mathrm{M}_y$ for each $x, y \in \Sigma$:

$${}_x\mathrm{M}_y \;\rightarrow\; {}_x\mathrm{R}\ \mathrm{L}_y$$

and fold the underlined children in L_u into ${}_v\mathrm{M}_u$:

$$\begin{array}{lll} {}_x\mathrm{M}_y & \rightarrow\ {}_x\mathrm{R}\ \mathrm{L}_y & \text{where } x, y \in \Sigma \\ \mathrm{L}_u & \rightarrow\ u_l \mid \mathrm{L}_v\ {}_v\mathrm{M}_u & \text{where } v \curvearrowleft u \end{array}$$

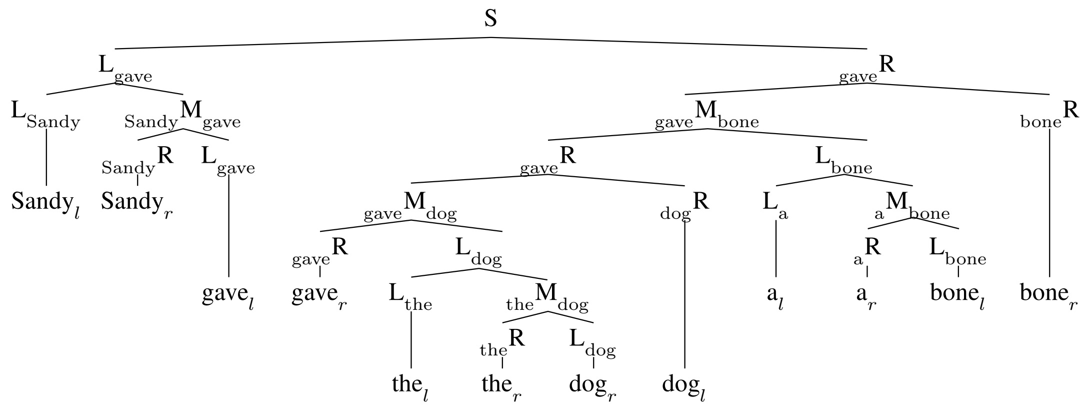

Figure 4: The $O(n^3)$ split-head parse corresponding to the dependency graph of Figure 1.

The $O(n^3)$ split-head grammar is obtained by unfolding the occurence of X_u in the S production and dropping the X_u schema as X_u no longer appears on the right-hand side of any production. The resulting $O(n^3)$ split-head grammar schemata are as follows:

$$
\begin{array}{rcll}
\mathrm{S} & \to & \mathrm{L}_u \ {}_u\mathrm{R} & \text{where } 0 \curvearrowright u \\
\mathrm{L}_u & \to & u_l & \\
\mathrm{L}_u & \to & \mathrm{L}_v \ {}_v\mathrm{M}_u & \text{where } v \curvearrowleft u \\
{}_u\mathrm{R} & \to & u_r & \\
{}_u\mathrm{R} & \to & {}_u\mathrm{M}_v \ {}_v\mathrm{R} & \text{where } u \curvearrowright v \\
{}_x\mathrm{M}_y & \to & {}_x\mathrm{R} \ \mathrm{L}_y & \text{where } x, y \in \Sigma
\end{array}
$$

As before, the dependency annotations on the production schemata permit us to map CFG parses to the corresponding dependency parse. This grammar requires $O(n^3)$ parsing time to parse because the indices are redundant given the constituent's string positions for the reasons described in section 4. Specifically, the rightmost terminal of L_u is always u_l, the leftmost terminal of ${}_u\mathrm{R}$ is always u_r and the leftmost and rightmost terminals of ${}_v\mathrm{M}_u$ are v_l and u_r respectively.

The $O(n^3)$ split-head grammar is closely related to the $O(n^3)$ PBDG parsing algorithm given by Eisner and Satta (1999). Specifically, the steps involved in parsing with this grammar using the CKY algorithm are essentially the same as those performed by the Eisner/Satta algorithm. The primary difference is that the Eisner/Satta algorithm involves two separate categories that are collapsed into the single category M here.

To confirm their relative performance we implemented stochastic CKY parsers for the three CFG schemata described so far. The production schemata were hard-coded for speed, and the implementation trick described in section 4 was used to avoid doubling the terminal string. We obtained dependency weights from our existing discriminatively-trained PBDG parser (not cited to preserve anonymity). We compared the parsers' running times on section 24 of the Penn Treebank. Because all three CFGs implement the same dependency grammar their Viterbi parses have the same dependency accuracy, namely 0.8918. We precompute the dependency weights, so the times include just the dynamic programming computation on a 3.6GHz Pentium 4.

CFG schemata	sentences parsed / second
Naive $O(n^5)$ CFG	45.4
$O(n^4)$ CFG	406.2
$O(n^3)$ CFG	3580.0

6 An $O(n^3)$ adjacent-head grammar

This section shows how to further transform the $O(n^3)$ grammar described above into a form that encodes second-order dependencies between adjacent dependent heads in much the way that a Markov PCFG does (McDonald, 2006; McDonald and Pereira, 2006). We provide a derivation for the L_u constituents; there is a parallel derivation for ${}_u\mathrm{R}$.

We begin by unfolding X_v in the definition of L_u in the split-head grammar, producing as before:

$$\mathrm{L}_u \ \to \ u_l \ | \ \mathrm{L}_v \ \underline{{}_v\mathrm{R} \ \mathrm{L}_u}$$

Now introduce a new nonterminal ${}_v\mathrm{M}_u^{\mathrm{L}}$, which is a specialized version of M requiring that v is a left-dependent of u, and fold the underlined constituents

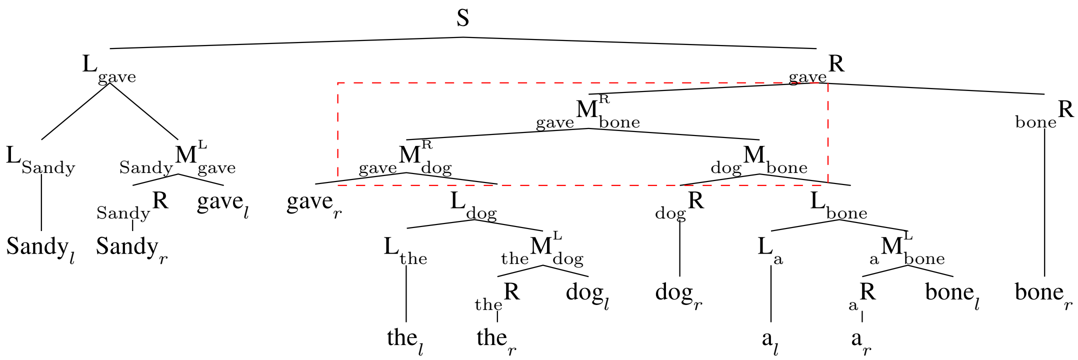

Figure 5: The $O(n^3)$ adjacent-head parse corresponding to the dependency graph of Figure 1. The boxed local tree indicates *bone* is the dependent of *give* following the dependent *dog*, i.e., $\mathit{give} \curvearrowright \mathit{dog} \curvearrowright \mathit{bone}$.

into ${}_v\mathrm{M}^{\mathrm{L}}_u$.

$$\begin{array}{rcll} {}_v\mathrm{M}^{\mathrm{L}}_u & \rightarrow & {}_v\mathrm{R}\ \ \mathrm{L}_u & \text{where } v \curvearrowleft u \\ \mathrm{L}_u & \rightarrow & u_l \mid \mathrm{L}_v\ {}_v\mathrm{M}^{\mathrm{L}}_u & \text{where } v \curvearrowleft u \end{array}$$

Now unfold L_u in the definition of ${}_v\mathrm{M}^{\mathrm{L}}_u$, producing:

$$\begin{array}{rcll} {}_v\mathrm{M}^{\mathrm{L}}_u & \rightarrow & {}_v\mathrm{R}\ u_l \mid \underline{{}_v\mathrm{R}\ \mathrm{L}_{v'}}\ {}_{v'}\mathrm{M}^{\mathrm{L}}_u; & v \curvearrowleft v' \curvearrowleft u \end{array}$$

Note that in the first production expanding ${}_v\mathrm{M}^{\mathrm{L}}_u$, v is the *closest* left dependent of u, and in the second production v and v' are *adjacent* left-dependents of u. ${}_v\mathrm{M}^{\mathrm{L}}_u$ has a ternary production, so we introduce ${}_x\mathrm{M}_y$ as before to fold the underlined constituents into.

$$\begin{array}{rcll} {}_x\mathrm{M}_y & \rightarrow & {}_x\mathrm{R}\ \mathrm{L}_y & \text{where } x, y \in \Sigma \\ {}_v\mathrm{M}^{\mathrm{L}}_u & \rightarrow & {}_v\mathrm{R}\ u_l \mid {}_v\mathrm{M}_{v'}\ {}_{v'}\mathrm{M}^{\mathrm{L}}_u; & v \curvearrowleft v' \curvearrowleft u \end{array}$$

The resulting grammar schema is as below, and a sample parse is given in Figure 5.

$$\begin{array}{rcll} \mathrm{S} & \rightarrow & \mathrm{L}_u\ {}_u\mathrm{R} & \text{where } 0 \curvearrowright u \\ \mathrm{L}_u & \rightarrow & u_l & u \text{ has no left dependents} \\ \mathrm{L}_u & \rightarrow & \mathrm{L}_v\ {}_v\mathrm{M}^{\mathrm{L}}_u & v \text{ is } u\text{'s last left dep.} \\ {}_v\mathrm{M}^{\mathrm{L}}_u & \rightarrow & {}_v\mathrm{R}\ u_l & v \text{ is } u\text{'s closest left dep.} \\ {}_v\mathrm{M}^{\mathrm{L}}_u & \rightarrow & {}_v\mathrm{M}_{v'}\ {}_{v'}\mathrm{M}^{\mathrm{L}}_u & v \curvearrowleft v' \curvearrowleft u \\ {}_u\mathrm{R} & \rightarrow & u_r & u \text{ has no right dependents} \\ {}_u\mathrm{R} & \rightarrow & {}_u\mathrm{M}^{\mathrm{R}}_v\ {}_v\mathrm{R} & v \text{ is } u\text{'s last right dep.} \\ {}_u\mathrm{M}^{\mathrm{R}}_v & \rightarrow & u_r\ \mathrm{L}_v & v \text{ is } u\text{'s closest right dep.} \\ {}_u\mathrm{M}^{\mathrm{R}}_v & \rightarrow & {}_u\mathrm{M}^{\mathrm{R}}_{v'}\ {}_{v'}\mathrm{M}_v & u \curvearrowright v' \curvearrowright v \\ {}_x\mathrm{M}_y & \rightarrow & {}_x\mathrm{R}\ \mathrm{L}_y & \text{where } x, y \in \Sigma \end{array}$$

As before, the indices on the nonterminals are redundant, as the heads are always located at an edge of each constituent, so they need not be computed or stored and the CFG can be parsed in $O(n^3)$ time. The steps involved in CKY parsing with this grammar correspond closely to those of the McDonald (2006) second-order PBDG parsing algorithm.

7 An $O(n^3)$ dependent-head grammar

This section shows a different application of Unfold-Fold can capture head-to-head-to-head dependencies, i.e., "vertical" second-order dependencies, rather than the "horizontal" ones captured by the transformation described in the previous section. Because we expect these vertical dependencies to be less important linguistically than the horizontal ones, we only sketch the transformation here.

The derivation differs from the one in Section 6 in that the dependent ${}_v\mathrm{R}$, rather than the head L_u, is unfolded in the initial definition of ${}_v\mathrm{M}^{\mathrm{L}}_u$. This results in a grammar that tracks vertical, rather than horizontal, second-order dependencies. Since left-hand and right-hand derivations are assembled separately in a split-head grammar, the grammar in fact only tracks zig-zag type dependencies (e.g., where a grandparent has a right dependent, which in turn has a left dependent).

The resulting grammar is given below, and a sample parse using this grammar is shown in Figure 6. Because the subscripts are redundant they can be omitted and the resulting CFG can be parsed in

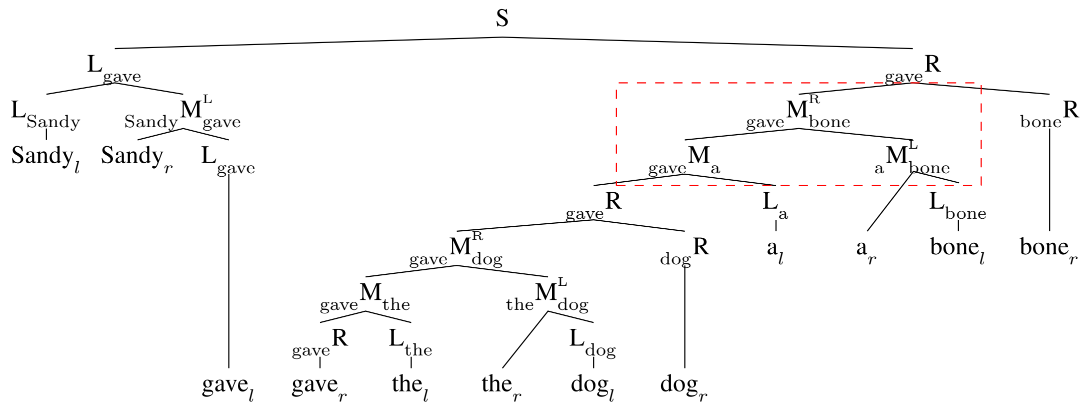

Figure 6: The n^3 dependent-head parse corresponding to the dependency graph of Figure 1. The boxed local tree indicates that *a* is a left-dependent of *bone*, which is in turn a right-dependent of *gave*, i.e., $gave \quad a \curvearrowleft bone$.

$O(n^3)$ time using the CKY algorithm.

$$\begin{array}{rcll}
\mathrm{S} & \to & \mathrm{L}_u\ {}_u\mathrm{R} & \text{where } 0 \curvearrowright u \\
\mathrm{L}_u & \to & u_l & \\
\mathrm{L}_u & \to & \mathrm{L}_v\ {}_v\mathrm{M}^{\mathrm{L}}_u & \text{where } v \curvearrowleft u \\
{}_v\mathrm{M}^{\mathrm{L}}_u & \to & v_r\ \mathrm{L}_u & \text{where } v \curvearrowleft u \\
{}_v\mathrm{M}^{\mathrm{L}}_u & \to & {}_v\mathrm{M}^{\mathrm{R}}_w\ {}_w\mathrm{M}_u & \text{where } v \quad w \quad u \\
{}_u\mathrm{R} & \to & u_r & \\
{}_u\mathrm{R} & \to & {}_u\mathrm{M}^{\mathrm{R}}_v\ {}_v\mathrm{R} & \text{where } u \curvearrowright v \\
{}_u\mathrm{M}^{\mathrm{R}}_v & \to & {}_u\mathrm{R}\ v_l & \text{where } u \curvearrowright v \\
{}_u\mathrm{M}^{\mathrm{R}}_v & \to & {}_u\mathrm{M}_w\ {}_w\mathrm{M}^{\mathrm{L}}_v & \text{where } u \quad w \quad u \\
{}_x\mathrm{M}_y & \to & {}_x\mathrm{R}\ \mathrm{L}_y & \text{where } x, y \in \Sigma
\end{array}$$

8 Maximum posterior decoding

As noted in the introduction, one consequence of the PBDG to CFG reductions presented in this paper is that CFG parsing and estimation techniques are now available for PBDGs as well. As an example application, this section describes Maximum Posterior Decoding (MPD) for PBDGs.

Goodman (1996) observed that the Viterbi parse is in general not the optimal parse for evaluation metrics such as f-score that are based on the number of correct constituents in a parse. He showed that MPD improves f-score modestly relative to Viterbi decoding for PCFGs.

Since dependency parse accuracy is just the proportion of dependencies in the parse that are correct, Goodman's observation should hold for PBDG parsing as well. MPD for PBDGs selects the parse that maximizes the sum of the marginal probabilities of each of the dependencies in the parse. Such a decoder might plausibly produce parses that score better on the dependency accuracy metric than Viterbi parses.

MPD is straightforward given the PBDG to CFG reductions described in this paper. Specifically, we use the Inside-Outside algorithm to compute the posterior probability of the CFG constituents corresponding to each PBDG dependency, and then use the Viterbi algorithm to find the parse tree that maximizes the sum of these posterior probabilities.

We implemented MPD for first-order PBDGs using dependency weights from our existing discriminatively-trained PBDG parser (not cited to preserve anonymity). These weights are estimated by an online procedure as in McDonald (2006), and are not intended to define a probability distribution. In an attempt to heuristically correct for this, in this experiment we used $\exp(\alpha w_{u,v})$ as the weight of the dependency between head u and dependent v, where $w_{u,v}$ is the weight provided by the discriminatively-trained model and α is an adjustable scaling parameter tuned to optimize MPD accuracy on development data.

Unfortunately we found no significant difference between the accuracy of the MPD and Viterbi parses. Optimizing MPD on the development data (section 24 of the PTB) set the scale factor $\alpha = 0.21$ and produced MPD parses with an accuracy of 0.8921, which is approximately the same as the Viterbi accuracy of 0.8918. On the blind test data (section 23) the two accuracies are essentially identical (0.8997).

There are several possible explanations for the failure of MPD to produce more accurate parses than Viterbi decoding. Perhaps MPD requires weights that define a probability distribution (e.g., a Max-Ent model). It is also possible that discriminative training adjusts the weights in a way that ensures that the Viterbi parse is close to the maximum posterior parse. This was the case in our experiment, and if this is true with discriminative training in general, then maximum posterior decoding will not have much to offer to discriminative parsing.

9 Conclusion

This paper shows how to use the Unfold-Fold transform to translate PBDGs into CFGs that can be parsed in $O(n^3)$ time. A key component of this is the split-head construction, where each word u in the input is split into two terminals u_l and u_r of the CFG parse. We also showed how to systematically transform the split-head CFG into grammars which track second-order dependencies. We provided one grammar which captures horizontal second-order dependencies (McDonald, 2006), and another which captures vertical second-order head-to-head-to-head dependencies.

The grammars described here just scratch the surface of what is possible with Unfold-Fold. Notice that both of the second-order grammars have more nonterminals than the first-order grammar. If one is prepared to increase the number of nonterminals still further, it may be possible to track additional information about constituents (although if we insist on $O(n^3)$ parse time we will be unable to track the interaction of more than three heads at once).

References

R.M. Burstall and John Darlington. 1977. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67.

Zhiyi Chi. 1999. Statistical properties of probabilistic context-free grammars. *Computational Linguistics*, 25(1):131–160.

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*, pages 457–480, University of Maryland.

Jason Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *COLING96: Proceedings of the 16th International Conference on Computational Linguistics*, pages 340–345, Copenhagen. Center for Sprogteknologi.

Jason Eisner. 2000. Bilexical grammars and their cubic-time parsing algorithms. In Harry Bunt and Anton Nijholt, editors, *Advances in Probabilistic and Other Parsing Technologies*, pages 29–62. Kluwer Academic Publishers.

Joshua T. Goodman. 1996. Parsing algorithms and metrics. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 177–183, Santa Cruz, Ca.

K. Lari and S.J. Young. 1990. The estimation of Stochastic Context-Free Grammars using the Inside-Outside algorithm. *Computer Speech and Language*, 4(35-56).

Takuya Matsuzaki, Yusuke Miyao, and Jun'ichi Tsujii. 2005. Probabilistic CFG with latent annotations. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 75–82, Ann Arbor, Michigan, June. Association for Computational Linguistics.

David McAllester. 1999. A reformulation of Eisner and Sata's cubic time parser for split head automata grammars. Available from http://ttic.uchicago.edu/~dmcallester/.

Ryan McDonald and Fernando Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *11th Conference of the European Chapter of the Association for Computational Linguistics*, pages 81–88, Trento, Italy.

Ryan McDonald. 2006. *Discriminative Training and Spanning Tree Algorithms for Dependency Parsing*. Ph.D. thesis, University of Pennyslvania, Philadelphia, PA.

Fernando Pereira and Stuart M. Shieber. 1987. *Prolog and Natural Language Analysis*. Center for the Study of Language and Information, Stanford, CA.

Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. 2006. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 433–440, Sydney, Australia, July. Association for Computational Linguistics.

A. Pettorossi and M. Proeitti. 1992. Transformation of logic programs. In *Handbook of Logic in Artificial Intelligence*, volume 5, pages 697–787. Oxford University Press.

Taisuke Sato. 1992. Equivalence-preserving first-order unfold/fold transformation systems. *Theoretical Computer Science*, 105(1):57–84.

Daniel H. Younger. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208.