Treebank Grammar Techniques for Non-Projective Dependency Parsing

Marco Kuhlmann
Uppsala University
Uppsala, Sweden
marco.kuhlmann@lingfil.uu.se

Giorgio Satta
University of Padua
Padova, Italy
satta@dei.unipd.it

Abstract

An open problem in dependency parsing is the accurate and efficient treatment of non-projective structures. We propose to attack this problem using chart-parsing algorithms developed for mildly context-sensitive grammar formalisms. In this paper, we provide two key tools for this approach. First, we show how to reduce non-projective dependency parsing to parsing with Linear Context-Free Rewriting Systems (LCFRS), by presenting a technique for extracting LCFRS from dependency treebanks. For efficient parsing, the extracted grammars need to be transformed in order to minimize the number of nonterminal symbols per production. Our second contribution is an algorithm that computes this transformation for a large, empirically relevant class of grammars.

1 Introduction

Dependency parsing is the task of predicting the most probable dependency structure for a given sentence. One of the key choices in dependency parsing is about the class of candidate structures for this prediction. Many parsers are confined to *projective* structures, in which the yield of a syntactic head is required to be continuous. A major benefit of this choice is computational efficiency: an exhaustive search over all projective structures can be done in cubic, greedy parsing in linear time (Eisner, 1996; Nivre, 2003). A major drawback of the restriction to projective dependency structures is a potential loss in accuracy. For example, around 23% of the analyses in the Prague Dependency Treebank of Czech (Hajič et al., 2001) are non-projective, and for German and Dutch treebanks, the proportion of non-projective structures is even higher (Havelka, 2007).

The problem of non-projective dependency parsing under the joint requirement of accuracy and efficiency has only recently been addressed in the literature. Some authors propose to solve it by techniques for recovering non-projectivity from the output of a projective parser in a post-processing step (Hall and Novák, 2005; Nivre and Nilsson, 2005), others extend projective parsers by heuristics that allow at least certain non-projective constructions to be parsed (Attardi, 2006; Nivre, 2007). McDonald et al. (2005) formulate dependency parsing as the search for the most probable spanning tree over the *full* set of all possible dependencies. However, this approach is limited to probability models with strong independence assumptions. Exhaustive non-projective dependency parsing with more powerful models is intractable (McDonald and Satta, 2007), and one has to resort to approximation algorithms (McDonald and Pereira, 2006).

In this paper, we propose to attack non-projective dependency parsing in a principled way, using polynomial chart-parsing algorithms developed for mildly context-sensitive grammar formalisms. This proposal is motivated by the observation that most dependency structures required for the analysis of natural language are very nearly projective, differing only minimally from the best projective approximation (Kuhlmann and Nivre, 2006), and by the close link between such 'mildly non-projective' dependency structures on the one hand, and grammar formalisms with mildly context-sensitive generative capacity on the other (Kuhlmann and Möhl, 2007). Furthermore, as pointed out by McDonald and Satta (2007), chart-parsing algorithms are amenable to augmentation by non-local information such as arity constraints and Markovization, and therefore should allow for more predictive statistical models than those used by current systems for non-projective dependency parsing. Hence, mildly non-projective dependency parsing promises to be both efficient and accurate.

Proceedings of the 12th Conference of the European Chapter of the ACL, pages 478–486,
Athens, Greece, 30 March – 3 April 2009. ©2009 Association for Computational Linguistics

Contributions In this paper, we contribute two key tools for making the mildly context-sensitive approach to accurate and efficient non-projective dependency parsing work.

First, we extend the standard technique for extracting context-free grammars from phrase-structure treebanks (Charniak, 1996) to mildly context-sensitive grammars and dependency treebanks. More specifically, we show how to extract, from a given dependency treebank, a lexicalized Linear Context-Free Rewriting System (LCFRS) whose derivations capture the dependency analyses in the treebank in the same way as the derivations of a context-free treebank grammar capture phrase-structure analyses. Our technique works for arbitrary, even non-projective dependency treebanks, and essentially reduces non-projective dependency to parsing with LCFRS. This problem can be solved using standard chart-parsing techniques.

Our extraction technique yields a grammar whose parsing complexity is polynomial in the length of the sentence, but exponential in both a measure of the non-projectivity of the treebank and the maximal number of dependents per word, reflected as the *rank* of the extracted LCFRS. While the number of highly non-projective dependency structures is negligible for practical applications (Kuhlmann and Nivre, 2006), the rank cannot easily be bounded. Therefore, we present an algorithm that transforms the extracted grammar into a normal form that has rank 2, and thus can be parsed more efficiently. This contribution is important even independently of the extraction procedure: While it is known that a rank-2 normal form of LCFRS does not exist in the general case (Rambow and Satta, 1999), our algorithm succeeds for a large and empirically relevant class of grammars.

2 Preliminaries

We start by introducing dependency trees and Linear Context-Free Rewriting Systems (LCFRS). Throughout the paper, for positive integers i and j, we write $[i, j]$ for the **interval** $\{ k \mid i \leq k \leq j \}$, and use $[n]$ as a shorthand for $[1, n]$.

2.1 Dependency Trees

Dependency parsing is the task to assign dependency structures to a given sentence w. For the purposes of this paper, dependency structures are edge-labelled trees. More formally, let w be a sentence, understood as a sequence of tokens over some given alphabet T, and let L be an alphabet of edge labels. A **dependency tree** for w is a construct $D = (w, E, \lambda)$, where E forms a rooted tree (in the standard graph-theoretic sense) on the set $[|w|]$, and λ is a total function that assigns every edge in E a label in L. Each node of D represents a (position of a) token in w.

Example 1 Figure 2 shows a dependency tree for the sentence *A hearing is scheduled on the issue today*, which consists of 8 tokens and the edges $\{ (2,1), (2,5), (3,2), (3,4), (4,8), (5,7), (7,6) \}$. The edges are labelled with syntactic functions such as sbj for 'subject'. The root node is marked by a dotted line. □

Let u be a node of a dependency tree D. A node u' is a **descendant** of u, if there is a (possibly empty) path from u to u'. A **block** of u is a maximal interval of descendants of u. The number of blocks of u is called the **block-degree** of u. The block-degree of a dependency tree is the maximum among the block-degrees of its nodes. A dependency tree is **projective**, if its block-degree is 1.

Example 2 The tree shown in Figure 2 is not projective: both node 2 (*hearing*) and node 4 (*scheduled*) have block-degree 2. Their blocks are $\{2\}, \{5, 6, 7\}$ and $\{4\}, \{8\}$, respectively.

2.2 LCFRS

Linear Context-Free Rewriting Systems (LCFRS) have been introduced as a generalization of several mildly context-sensitive grammar formalisms. Here we use the standard definition of LCFRS (Vijay-Shanker et al., 1987) and only fix our notation; for a more thorough discussion of this formalism, we refer to the literature.

Let G be an LCFRS. Recall that each nonterminal symbol A of G comes with a positive integer called the **fan-out** of A, and that a production p of G has the form

$$A \to g(A_1, \dots, A_r) \; ; \; g(\vec{x}_1, \dots, \vec{x}_r) = \vec{\alpha} \,,$$

where $A, A_1, \dots, A_r$ are nonterminals with fan-out $f, f_1, \dots, f_r$, respectively, g is a function symbol, and the equation to the right of the semicolon specifies the semantics of g. For each $i \in [r]$, $\vec{x}_i$ is an f_i-tuple of variables, and $\vec{\alpha} = \langle \alpha_1, \dots, \alpha_f \rangle$ is a tuple of strings over the variables on the left-hand side of the equation and the alphabet of terminal symbols in which each variable appears exactly once. The production p is said to have **rank** r, **fan-out** f, and **length** $|\alpha_1| + \cdots + |\alpha_f| + (f - 1)$.

3 Grammar Extraction

We now explain how to extract an LCFRS from a dependency treebank, in very much the same way as a context-free grammar can be extracted from a phrase-structure treebank (Charniak, 1996).

3.1 Dependency Treebank Grammars

A simple way to induce a context-free grammar from a phrase-structure treebank is to read off the productions of the grammar from the trees. We will specify a procedure for extracting, from a given dependency treebank, a lexicalized LCFRS G that is **adequate** in the sense that for every analysis D of a sentence w in the treebank, there is a derivation tree of G that is isomorphic to D, meaning that it becomes equal to D after a suitable renaming and relabelling of nodes, and has w as its derived string. Here, a **derivation tree** of an LCFRS G is an ordered tree such that each node u is labelled with a production p of G, the number of children of u equals the rank r of p, and for each $i \in [r]$, the ith child of u is labelled with a production that has as its left-hand side the ith nonterminal on the right-hand side of p.

The basic idea behind our extraction procedure is that, in order to represent the compositional structure of a possibly non-projective dependency tree, one needs to represent the decomposition and relative order not of subtrees, but of blocks of subtrees (Kuhlmann and Möhl, 2007). We introduce some terminology. A **component** of a node u in a dependency tree is either a block B of some child u' of u, or the singleton interval that contains u; this interval will represent the position in the string that is occupied by the lexical item corresponding to u. We say that u' **contributes** B, and that u contributes $[u, u]$ to u. Notice that the number of components that u' contributes to its parent u equals the block-degree of u'. Our goal is to construct for u a production of an LCFRS that specifies how each block of u decomposes into components, and how these components are ordered relative to one another. These productions will make an adequate LCFRS, in the sense defined above.

3.2 Annotating the Components

The core of our extraction procedure is an efficient algorithm that annotates each node u of a given dependency tree with the list of its components, sorted by their left endpoints. It is helpful to think of this algorithm as of two independent parts, one that annotates each node u with the list of the left endpoints of its components (ANNOTATE-L) and one that annotates the corresponding right endpoints (ANNOTATE-R). The list of components can then be obtained by zipping the two lists of endpoints together in linear time.

```
1: Function ANNOTATE-L(D)
2: for each u of D, from left to right do
3:   if u is the first node of D then
4:     b := the root node of D
5:   else
6:     b := the lca of u and its predecessor
7:   for each u' on the path b ··· u do
8:     left[u'] := left[u'] · u
```

Figure 1: Annotation with components

Figure 1 shows pseudocode for ANNOTATE-L; the pseudocode for ANNOTATE-R is symmetric. We do a single left-to-right sweep over the nodes of the input tree D. In each step, we annotate all nodes u' that have the current node u as the left endpoint of one of their components. Since the sweep is from left to right, this will get us the left endpoints of u' in the desired order. The nodes that we annotate are the nodes u' on the path between u and the **least common ancestor (lca)** b of u and its predecessor, or the path from the root node to u, in case that u is the leftmost node of D.

Example 3 For the dependency tree in Figure 2, ANNOTATE-L constructs the following lists left[u] of left endpoints, for $u = 1, \dots, 8$:

$$1,\ 1\cdot 2\cdot 5,\ 1\cdot 3\cdot 4\cdot 5\cdot 8,\ 4\cdot 8,\ 5\cdot 6,\ 6,\ 6\cdot 7,\ 8$$

The following Lemma establishes the correctness of the algorithm:

Lemma 1 *Let D be a dependency tree, and let u and u' be nodes of D. Let b be the least common ancestor of u and its predecessor, or the root node in case that u is the leftmost node of D. Then u is the left endpoint of a component of u' if and only if u' lies on the path from b to u.* □

PROOF It is clear that u' must be an ancestor of u. If u is the leftmost node of D, then u is the left endpoint of the leftmost component of all of its ancestors. Now suppose that u is not the leftmost node of D, and let $\hat{u}$ be the predecessor of u. Distinguish three cases: If u' is not an ancestor of $\hat{u}$, then $\hat{u}$ does not belong to any component of u'; therefore, u is the left endpoint of a component

of u'. If u' is an ancestor of $\hat{u}$ but $u' \neq b$, then $\hat{u}$ and u belong to the same component of u'; therefore, u is not the left endpoint of this component. Finally, if $u' = b$, then $\hat{u}$ and u belong to different components of u'; therefore, u is the left endpoint of the component it belongs to. ■

We now turn to an analysis of the runtime of the algorithm. Let n be the number of components of D. It is not hard to imagine an algorithm that performs the annotation task in time $O(n \log n)$: such an algorithm could construct the components for a given node u by essentially merging the list of components of the children of u into a new sorted list. In contrast, our algorithm takes time $O(n)$. The crucial part of the analysis is the assignment in line 6, which computes the least common ancestor of u and its predecessor. Using markers for the path from the root node to u, it is straightforward to implement this assignment in time $O(|\pi|)$, where π is the path $b \cdots u$. Now notice that, by our correctness argument, line 8 of the algorithm is executed exactly n times. Therefore, the sum over the lengths of all the paths π, and hence the amortized time of computing all the least common ancestors in line 6, is $O(n)$. This runtime complexity is optimal for the task we are solving.

3.3 Extraction Procedure

We now describe how to extend the annotation algorithm into a procedure that extracts an LCFRS from a given dependency tree D. The basic idea is to transform the list of components of each node u of D into a production p. This transformation will only rename and relabel nodes, and therefore yield an adequate derivation tree. For the construction of the production, we actually need an extended version of the annotation algorithm, in which each component is annotated with the node that contributed it. This extension is straightforward, and does not affect the linear runtime complexity.

Let D be a dependency tree for a sentence w. Consider a single node u of D, and assume that u has r children, and that the block-degree of u is f. We construct for u a production p with rank r and fan-out f. For convenience, let us order the children of u, say by their leftmost descendants, and let us write u_i for the ith child of u according to this order, and f_i for the block-degree of u_i, $i \in [r]$. The production p has the form

$$L \to g(L_1, \dots, L_r) \; ; \; g(\vec{x}_1, \dots, \vec{x}_r) = \vec{\alpha} \, ,$$

where L is the label of the incoming edge of u (or the special label root in case that u is the root node of D) and for each $i \in [r]$: L_i is the label of the incoming edge of u_i; $\vec{x}_i$ is a f_i-tuple of variables of the form $x_{i,j}$, where $j \in [f_i]$; and $\vec{\alpha}$ is an f-tuple that is constructed in a single left-to-right sweep over the list of components computed for u as follows. Let $k \in [f_i]$ be a pointer to a current segment of $\vec{\alpha}$; initially, $k = 1$. If the current component is not adjacent (as an interval) to the previous component, we increase k by one. If the current component is contributed by the child u_i, $i \in [r]$, we add the variable $x_{i,j}$ to α_k, where j is the number of times we have seen a component contributed by u_i during the sweep. Notice that $j \in [f_i]$. If the current component is the (unique) component contributed by u, we add the token corresponding to u to α_k. In this way, we obtain a complete specification of how the blocks of u (represented by the segments of the tuple $\vec{\alpha}$) decompose into the components of u, and of the relative order of the components. As an example, Figure 2 shows the productions extracted from the tree above.

3.4 Parsing the Extracted Grammar

Once we have extracted the grammar for a dependency treebank, we can apply any parsing algorithm for LCFRS to non-projective dependency parsing. The generic chart-parsing algorithm for LCFRS runs in time $O(|P| \cdot |w|^{f(r+1)})$, where P is the set of productions of the input grammar G, w is the input string, r is the maximal rank, and f is the maximal fan-out of a production in G (Seki et al., 1991). For a grammar G extracted by our technique, the number f equals the maximal block-degree per node. Hence, without any further modification, we obtain a parsing algorithm that is polynomial in the length of the sentence, but exponential in both the block-degree and the rank. This is clearly unacceptable in practical systems. The relative frequency of analyses with a block-degree ≥ 2 is almost negligible (Havelka, 2007); the bigger obstacle in applying the treebank grammar is the rank of the resulting LCFRS. Therefore, in the remainder of the paper, we present an algorithm that can transform the productions of the input grammar G into an equivalent set of productions with rank at most 2, while preserving the fan-out. This transformation, if it succeeds, yields a parsing algorithm that runs in time $O(|P| \cdot r \cdot |w|^{3f})$.

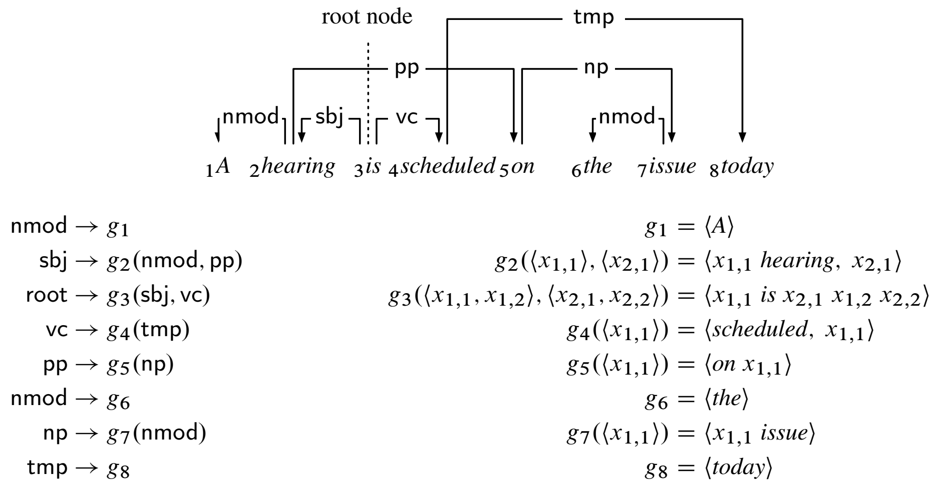

Figure 2: A dependency tree, and the LCFRS extracted for it

4 Adjacency

In this section we discuss a method for factorizing an LCFRS into productions of rank 2. Before starting, we get rid of the 'easy' cases. A production p is **connected** if any two strings α_i, α_j in p's definition share at least one variable referring to the same nonterminal. It is not difficult to see that, when p is *not* connected, we can always split it into new productions of lower rank. Therefore, throughout this section we assume that LCFRS only have connected productions. We can split p into its connected components using standard methods for finding the strongly connected components of an undirected graph. This can be implemented in time $O(r \cdot f)$, where r and f are the rank and the fan-out of p, respectively.

4.1 Adjacency Graphs

Let p be a production with length n and fan-out f, associated with function a g. The set of **positions** of p is the set $[n]$. Informally, each position represents a variable or a lexical element in one of the components of the definition of g, or else a 'gap' between two of these components. (Recall that n also accounts for the $f - 1$ gaps in the body of g.)

Example 4 The set of positions of the production for *hearing* in Figure 2 is [4]: 1 for variable x_1, 2 for *hearing*, 3 for the gap, and 4 for y_1. □

Let $i_1, j_1, i_2, j_2 \in [n]$. An interval $[i_1, j_1]$ is **adjacent** to an interval $[i_2, j_2]$ if either $j_1 = i_2 - 1$ (left-adjacent) or $i_1 = j_2 + 1$ (right-adjacent). A multi-interval, or **m-interval** for short, is a set v of pairwise disjoint intervals such that no interval in v is adjacent to any other interval in v. The **fan-out** of v, written $f(v)$, is defined as $|v|$.

We use m-intervals to represent the nonterminals and the lexical element heading p. The ith nonterminal on the right-hand side of p is represented by the m-interval obtained by collecting all the positions of p that represent a variable from the ith argument of g. The head of p is represented by the m-interval containing the associated position. Note that all these m-intervals are pairwise disjoint.

Example 5 Consider the production for *is* in Figure 2. The set of positions is [5]. The first nonterminal is represented by the m-interval $\{[1,1],[4,4]\}$, the second nonterminal by $\{[3,3],[5,5]\}$, and the lexical head by $\{[2,2]\}$. □

For disjoint m-intervals v_1, v_2, we say that v_1 is **adjacent** to v_2, denoted by $v_1 \to v_2$, if for every interval $I_1 \in v_1$, there is an interval $I_2 \in v_2$ such that I_1 is adjacent to I_2. Adjacency is not symmetric: if $v_1 = \{[1,1],[4,4]\}$ and $v_2 = \{[2,2]\}$, then $v_2 \to v_1$, but not vice versa.

Let V be some collection of pairwise disjoint m-intervals representing p as above. The **adjacency graph** associated with p is the graph $G = (V, \to_G)$ whose vertices are the m-intervals in V, and whose edges $\to_G$ are defined by restricting the adjacency relation $\to$ to the set V.

For m-intervals $v_1, v_2 \in V$, the **merger** of v_1 and v_2, denoted by $v_1 \oplus v_2$, is the (uniquely determined) m-interval whose span is the union of the spans of v_1 and v_2. As an example, if $v_1 = \{[1,1],[3,3]\}$ and $v_2 = \{[2,2]\}$, then $v_1 \oplus v_2 = \{[1,3]\}$. Notice that the way in which we defined m-intervals ensures that a merging operation collapses all adjacent intervals. The proof of the following lemma is straightforward and omitted for space reasons:

```
1: Function FACTORIZE(G = (V, →_G))
2: R := ∅;
3: while →_G ≠ ∅ do
4:     choose (v1, v2) ∈ →_G;
5:     R := R ∪ { (v1, v2) };
6:     V := V − { v1, v2 } ∪ { v1 ⊕ v2 };
7:     →_G := { (v, v') | v, v' ∈ V, v → v' };
8: if |V| = 1 then
9:     output R and accept;
10: else
11:     reject;
```

Figure 3: Factorization algorithm

Lemma 2 *If $v_1 \to v_2$, then $f(v_1 \oplus v_2) \leq f(v_2)$.*

4.2 The Adjacency Algorithm

Let $G = (V, \to_G)$ be some adjacency graph, and let $v_1 \to_G v_2$. We can **derive** a new adjacency graph from G by merging v_1 and v_2. The resulting graph G' has vertices $V' = V - \{ v_1, v_2 \} \cup \{ v_1 \oplus v_2 \}$ and set of edges $\to_{G'}$ obtained by restricting the adjacency relation $\to$ to V'. We denote the derive relation as $G \Rightarrow_{(v_1,v_2)} G'$.

Informally, if G represents some LCFRS production p and v_1, v_2 represent nonterminals A_1, A_2, then G' represents a production p' obtained from p by replacing A_1, A_2 with a fresh nonterminal A. A new production p'' can also be constructed, expanding A into A_1, A_2, so that p', p'' together will be equivalent to p. Furthermore, p' has a rank smaller than the rank of p and, from Lemma 2, A does not increase the overall fan-out of the grammar.

In order to simplify the notation, we adopt the following convention. Let $G \Rightarrow_{(v_1,v_2)} G'$ and let $v \to_G v_1$, $v \neq v_2$. If $v \to_{G'} v_1 \oplus v_2$, then edges (v, v_1) and $(v, v_1 \oplus v_2)$ will be identified, and we say that G' **inherits** $(v, v_1 \oplus v_2)$ from G. If $v \not\to_{G'} v_1 \oplus v_2$, then we say that (v, v_1) does not **survive** the derive step. This convention is used for all edges incident upon v_1 or v_2.

Our factorization algorithm is reported in Figure 3. We start from an adjacency graph representing some LCFRS production that needs to be factorized. We arbitrarily choose an edge e of the graph, and push it into a set R, in order to keep a record of the candidate factorization. We then merge the two m-intervals incident to e, and we recompute the adjacency relation for the new set of vertices. We iterate until the resulting graph has an empty edge set. If the final graph has one one vertex, then we have managed to factorize our production into a set of productions with rank at most two that can be computed from R.

Example 6 Let $V = \{ v_1, v_2, v_3 \}$ with $v_1 = \{ [4, 4] \}$, $v_2 = \{ [1, 1], [3, 3] \}$, and $v_3 = \{ [2, 2], [5, 5] \}$. Then $\to_G = \{ (v_1, v_2) \}$. After merging v_1, v_2 we have a new graph G with $V = \{ v_1 \oplus v_2, v_3 \}$ and $\to_G = \{ (v_1 \oplus v_2, v_3) \}$. We finally merge $v_1 \oplus v_2, v_3$ resulting in a new graph G with $V = \{ v_1 \oplus v_2 \oplus v_3 \}$ and $\to_G = \emptyset$. We then accept and stop. □

4.3 Mathematical Properties

We have already argued that, if the algorithm accepts, then a binary factorization that does not increase the fan-out of the grammar can be built from R. We still need to prove that the algorithm answers consistently on a given input, despite of possibly different choices of edges at line 4. We do this through several intermediate results.

A **derivation** for an adjacency graph G is a sequence of edges $d = \langle e_1, \ldots, e_n \rangle$, $n \geq 1$, such that $G = G_0$ and $G_{i-1} \Rightarrow_{e_i} G_i$ for every i with $1 \leq i \leq n$. For short, we write $G_0 \Rightarrow_d G_n$. Two derivations for G are **competing** if one is a permutation of the other.

Lemma 3 *If $G \Rightarrow_{d_1} G_1$ and $G \Rightarrow_{d_2} G_2$ with d_1 and d_2 competing derivations, then $G_1 = G_2$.*

PROOF We claim that the statement of the lemma holds for $|d_1| = 2$. To see this, let $G \Rightarrow_{e_1} G'_1 \Rightarrow_{e_2} G_1$ and $G \Rightarrow_{e_2} G'_2 \Rightarrow_{e_1} G_2$ be valid derivations. We observe that G_1 and G_2 have the same set of vertices. Since the edges of G_1 and G_2 are defined by restricting the adjacency relation to their set of vertices, our claim immediately follows.

The statement of the lemma then follows from the above claim and from the fact that we can always obtain the sequence d_2 starting from d_1 by repeatedly switching consecutive edges. ■

We now consider derivations for the same adjacency graph that are not competing, and show that they always lead to isomorphic adjacency graphs. Two graphs are **isomorphic** if they become equal after some suitable renaming of the vertices.

Lemma 4 *The out-degree of G is bounded by 2.*

PROOF Assume $v \to_G v_1$ and $v \to_G v_2$, with $v_1 \neq v_2$, and let $I \in v$. I must be adjacent to some interval $I_1 \in v_1$. Without loss of generality, assume that I is left-adjacent to I_1. I must also be adjacent to some interval $I_2 \in v_2$. Since v_1 and v_2

are disjoint, I must be right-adjacent to I_2. This implies that I cannot be adjacent to an interval in any other m-interval v' of G. ■

A vertex v of G such that $v \rightarrow_G v_1$ and $v \rightarrow_G v_2$ is called a **bifurcation**.

Example 7 Assume $v = \{[2,2]\}$, $v_1 = \{[3,3],[5,5]\}$, $v_2 = \{[1,1]\}$ with $v \rightarrow_G v_1$ and $v \rightarrow_G v_2$. The m-interval $v \oplus v_1 = \{[2,3],[5,5]\}$ is no longer adjacent to v_2. □

The example above shows that, when choosing one of the two outgoing edges in a bifurcation for merging, the other edge might not survive. Thus, such a choice might lead to distinguishable derivations that are not competing (one derivation has an edge that is not present in the other). As we will see (in the proof of Theorem 1), bifurcations are the only cases in which edges might not survive a merging.

Lemma 5 *Let v be a bifurcation of G with outgoing edges e_1, e_2, and let $G \Rightarrow_{e_1} G_1$, $G \Rightarrow_{e_2} G_2$. Then G_1 and G_2 are isomorphic.*

Proof (Sketch) Assume e_1 has the form $v \rightarrow_G v_1$ and e_2 has the form $v \rightarrow_G v_2$. Let also V_S be the set of vertices shared by G_1 and G_2. We show that the statement holds under the isomorphism mapping $v \oplus v_1$ and v_2 in G_1 to v_1 and $v \oplus v_2$ in G_2, respectively.

When restricted to V_S, the graphs G_1 and G_2 are equal. Let us then consider edges from G_1 and G_2 involving exactly one vertex in V_S. We show that, for $v' \in V_S$, $v' \rightarrow_{G_1} v \oplus v_1$ if and only if $v' \rightarrow_{G_2} v_1$. Consider an arbitrary interval $I' \in v'$. If $v' \rightarrow_{G_1} v \oplus v_1$, then I' must be adjacent to some interval $I_1 \in v \oplus v_1$. If $I_1 \in v_1$ we are done. Otherwise, I_1 must be the concatenation of two intervals I_{1v} and I_{1v_1} with $I_{1v} \in v$ and $I_{1v_1} \in v_1$. Since $v \rightarrow_{G_2} v_2$, I_{1v} is also adjacent to some interval in v_2. However, v' and v_2 are disjoint. Thus I' must be adjacent to $I_{1v_1} \in v_1$. Conversely, if $v' \rightarrow_{G_2} v_1$, then I' must be adjacent to some interval $I_1 \in v_1$. Because v' and v are disjoint, I' must also be adjacent to some interval in $v \oplus v_1$.

Using very similar arguments, we can conclude that G_1 and G_2 are isomorphic when restricted to edges with at most one vertex in V_S.

Finally, we need to consider edges from G_1 and G_2 that are not incident upon vertices in V_S. We show that $v \oplus v_1 \rightarrow_{G_1} v_2$ only if $v_1 \rightarrow_{G_2} v \oplus v_2$; a similar argument can be used to prove the converse. Consider an arbitrary interval $I_1 \in v \oplus v_1$. If $v \oplus v_1 \rightarrow_{G_1} v_2$, then I_1 must be adjacent to some interval $I_2 \in v_2$. If $I_1 \in v_1$ we are done. Otherwise, I_1 must be the concatenation of two adjacent intervals I_{1v} and I_{1v_1} with $I_{1v} \in v$ and $I_{1v_1} \in v_1$. Since I_{1v} is also adjacent to some interval $I'_2 \in v_2$ (here I'_2 might as well be I_2), we conclude that $I_{1v_1} \in v_1$ is adjacent to the concatenation of I_{1v} and I'_2, which is indeed an interval in $v \oplus v_2$. Note that our case distinction is exhaustive. We thus conclude that $v_1 \rightarrow_{G_2} v \oplus v_2$.

A symmetrical argument can be used to show that $v_2 \rightarrow_{G_1} v \oplus v_1$ if and only if $v \oplus v_2 \rightarrow_{G_2} v_1$, which concludes our proof. ■

Theorem 1 *Let d_1 and d_2 be derivations for G, describing two different computations c_1 and c_2 of the algorithm of Figure 3 on input G. Computation c_1 is accepting if and only if c_2 is accepting.*

Proof First, we prove the claim that if e is not an edge outgoing from a bifurcation vertex, then in the derive relation $G \Rightarrow_e G'$ all of the edges of G but e and its reverse are inherited by G'. Let us write e in the form $v_1 \rightarrow_G v_2$. Obviously, any edge of G not incident upon v_1 or v_2 will be inherited by G'. If $v \rightarrow_G v_2$ for some m-interval $v \neq v_1$, then every interval $I \in v$ is adjacent to some interval in v_2. Since v and v_1 are disjoint, I will also be adjacent to some interval in $v_1 \oplus v_2$. Thus we have $v \rightarrow_{G'} v_1 \oplus v_2$. A similar argument shows that $v \rightarrow_G v_1$ implies $v \rightarrow_{G'} v_1 \oplus v_2$.

If $v_2 \rightarrow_G v$ for some $v \neq v_1$, then every interval $I \in v_2$ is adjacent to some interval in v. From $v_1 \rightarrow_G v_2$ we also have that each interval $I_{12} \in v_1 \oplus v_2$ is either an interval in v_2 or else the concatenation of exactly two intervals $I_1 \in v_1$ and $I_2 \in v_2$. (The interval I_2 cannot be adjacent to more than an interval in v_1, because $v_2 \rightarrow_G v$). In both cases I_{12} is adjacent to some interval in v, and hence $v_1 \oplus v_2 \rightarrow_{G'} v$. This concludes the proof of our claim.

Let d_1, d_2 be as in the statement of the theorem, with $G \Rightarrow_{d_1} G_1$ and $G \Rightarrow_{d_2} G_2$. If d_1 and d_2 are competing, then the theorem follows from Lemma 3. Otherwise, assume that d_1 and d_2 are not competing. From our claim above, some bifurcation vertices must appear in these derivations. Let us reorder the edges in d_1 in such a way that edges outgoing from a bifurcation vertex are processed last and in some canonical order. The resulting derivation has the form dd'_1, where d'_1 involves the processing of all bifurcation vertices. We can also reorder edges in d_2 to obtain dd'_2, where d'_2 involves the processing of all bifurcation

not context-free	102 687	100.00%
not binarizable	24	0.02%
not well-nested	622	0.61%

Table 1: Properties of productions extracted from the CoNLL 2006 data (3 794 605 productions)

vertices in exactly the same order as in d'_1, but with possibly different choices for the outgoing edges.

Let $G \Rightarrow_d G_d \Rightarrow_{d'_1} G'_1$ and $G \Rightarrow_d G_d \Rightarrow_{d'_2} G'_2$. Derivations dd'_1 and d_1 are competing. Thus, by Lemma 3, we have $G'_1 = G_1$. Similarly, we can conclude that $G'_2 = G_2$. Since bifurcation vertices in d'_1 and in d'_2 are processed in the same canonical order, from repeated applications of Lemma 5 we have that G'_1 and G'_2 are isomorphic. We then conclude that G_1 and G_2 are isomorphic as well. The statement of the theorem follows immediately. ∎

We now turn to a computational analysis of the algorithm of Figure 3. Let G be the representation of an LCFRS production p with rank r. G has r vertices and, following Lemma 4, $O(r)$ edges. Let v be an m-interval of G with fan-out f_v. The incoming and outgoing edges for v can be detected in time $O(f_v)$ by inspecting the $2 \cdot f_v$ endpoints of v. Thus we can compute G in time $O(|p|)$.

The number of iterations of the while cycle in the algorithm is bounded by r, since at each iteration one vertex of G is removed. Consider now an iteration in which m-intervals v_1 and v_2 have been chosen for merging, with $v_1 \rightarrow_G v_2$. (These m-intervals might be associated with nonterminals in the right-hand side of p, or else might have been obtained as the result of previous merging operations.) Again, we can compute the incoming and outgoing edges of $v_1 \oplus v_2$ in time proportional to the number of endpoints of such an m-interval. By Lemma 2, this number is bounded by $O(f)$, f the fan-out of the grammar. We thus conclude that a run of the algorithm on G takes time $O(r \cdot f)$.

5 Discussion

We have shown how to extract mildly context-sensitive grammars from dependency treebanks, and presented an efficient algorithm that attempts to convert these grammars into an efficiently parseable binary form. Due to previous results (Rambow and Satta, 1999), we know that this is not always possible. However, our algorithm may fail even in cases where a binarization exists—our notion of adjacency is not strong enough to capture all binarizable cases. This raises the question about the practical relevance of our technique.

In order to get at least a preliminary answer to this question, we extracted LCFRS productions from the data used in the 2006 CoNLL shared task on data-driven dependency parsing (Buchholz and Marsi, 2006), and evaluated how large a portion of these productions could be binarized using our algorithm. The results are given in Table 1. Since it is easy to see that our algorithm always succeeds on context-free productions (productions where each nonterminal has fan-out 1), we evaluated our algorithm on the 102 687 productions with a higher fan-out. Out of these, only 24 (0.02%) could *not* be binarized using our technique. We take this number as an indicator for the usefulness of our result.

It is interesting to compare our approach with techniques for **well-nested** dependency trees (Kuhlmann and Nivre, 2006). Well-nestedness is a property that implies the binarizability of the extracted grammar; however, the classes of well-nested trees and those whose corresponding productions can be binarized using our algorithm are incomparable—in particular, there are well-nested productions that cannot be binarized in our framework. Nevertheless, the coverage of our technique is actually higher than that of an approach that relies on well-nestedness, at least on the CoNLL 2006 data (see again Table 1).

We see our results as promising first steps in a thorough exploration of the connections between non-projective and mildly context-sensitive parsing. The obvious next step is the evaluation of our technique in the context of an actual parser.

As a final remark, we would like to point out that an alternative technique for efficient non-projective dependency parsing, developed by Gómez Rodríguez et al. independently of this work, is presented elsewhere in this volume.

Acknowledgements We would like to thank Ryan McDonald, Joakim Nivre, and the anonymous reviewers for useful comments on drafts of this paper, and Carlos Gómez Rodríguez and David J. Weir for making a preliminary version of their paper available to us. The work of the first author was funded by the Swedish Research Council. The second author was partially supported by MIUR under project PRIN No. 2007TJNZRE_002.

References

Giuseppe Attardi. 2006. Experiments with a multilanguage non-projective dependency parser. In *Tenth Conference on Computational Natural Language Learning (CoNLL)*, pages 166–170, New York, USA.

Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Tenth Conference on Computational Natural Language Learning (CoNLL)*, pages 149–164, New York, USA.

Eugene Charniak. 1996. Tree-bank grammars. In *13th National Conference on Artificial Intelligence*, pages 1031–1036, Portland, Oregon, USA.

Jason Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *16th International Conference on Computational Linguistics (COLING)*, pages 340–345, Copenhagen, Denmark.

Carlos Gómez-Rodríguez, David J. Weir, and John Carroll. 2009. Parsing mildly non-projective dependency structures. In *Twelfth Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, Athens, Greece.

Jan Hajič, Barbora Vidova Hladka, Jarmila Panevová, Eva Hajičová, Petr Sgall, and Petr Pajas. 2001. Prague Dependency Treebank 1.0. Linguistic Data Consortium, 2001T10.

Keith Hall and Václav Novák. 2005. Corrective modelling for non-projective dependency grammar. In *Ninth International Workshop on Parsing Technologies (IWPT)*, pages 42–52, Vancouver, Canada.

Jiří Havelka. 2007. Beyond projectivity: Multilingual evaluation of constraints and measures on non-projective structures. In *45th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 608–615, Prague, Czech Republic.

Marco Kuhlmann and Mathias Möhl. 2007. Mildly context-sensitive dependency languages. In *45th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 160–167, Prague, Czech Republic.

Marco Kuhlmann and Joakim Nivre. 2006. Mildly non-projective dependency structures. In *21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (COLING-ACL), Main Conference Poster Sessions*, pages 507–514, Sydney, Australia.

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

Ryan McDonald and Giorgio Satta. 2007. On the complexity of non-projective data-driven dependency parsing. In *Tenth International Conference on Parsing Technologies (IWPT)*, pages 121–132, Prague, Czech Republic.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005. Non-projective dependency parsing using spanning tree algorithms. In *Human Language Technology Conference (HLT) and Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 523–530, Vancouver, Canada.

Joakim Nivre and Jens Nilsson. 2005. Pseudo-projective dependency parsing. In *43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 99–106, Ann Arbor, USA.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Eighth International Workshop on Parsing Technologies (IWPT)*, pages 149–160, Nancy, France.

Joakim Nivre. 2007. Incremental non-projective dependency parsing. In *Human Language Technologies: The Conference of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL)*, pages 396–403, Rochester, NY, USA.

Owen Rambow and Giorgio Satta. 1999. Independent parallelism in finite copying parallel rewriting systems. *Theoretical Computer Science*, 223(1–2):87–120.

Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. 1991. On Multiple Context-Free Grammars. *Theoretical Computer Science*, 88(2):191–229.

K. Vijay-Shanker, David J. Weir, and Aravind K. Joshi. 1987. Characterizing structural descriptions produced by various grammatical formalisms. In *25th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 104–111, Stanford, CA, USA.