Efficient Parsing for Head-Split Dependency Trees

Giorgio Satta
Dept. of Information Engineering
University of Padua, Italy
satta@dei.unipd.it

Marco Kuhlmann
Dept. of Linguistics and Philology
Uppsala University, Sweden
marco.kuhlmann@lingfil.uu.se

Abstract

Head splitting techniques have been successfully exploited to improve the asymptotic runtime of parsing algorithms for projective dependency trees, under the arc-factored model. In this article we extend these techniques to a class of non-projective dependency trees, called well-nested dependency trees with block-degree at most 2, which has been previously investigated in the literature. We define a structural property that allows head splitting for these trees, and present two algorithms that improve over the runtime of existing algorithms at no significant loss in coverage.

1 Introduction

Much of the recent work on dependency parsing has been aimed at finding a good balance between accuracy and efficiency. For one end of the spectrum, Eisner (1997) showed that the highest-scoring *projective* dependency tree under an arc-factored model can be computed in time $\mathcal{O}(n^3)$, where n is the length of the input string. Later work has focused on making projective parsing viable under more expressive models (Carreras, 2007; Koo and Collins, 2010).

At the same time, it has been observed that for many standard data sets, the coverage of projective trees is far from complete (Kuhlmann and Nivre, 2006), which has led to an interest in parsing algorithms for *non-projective* trees. While non-projective parsing under an arc-factored model can be done in time $\mathcal{O}(n^2)$ (McDonald et al., 2005), parsing with more informed models is intractable (McDonald and Satta, 2007). This has led several authors to investigate 'mildly non-projective' classes of trees, with the goal of achieving a balance between expressiveness and complexity (Kuhlmann and Nivre, 2006).

In this article we focus on a class of mildly non-projective dependency structures called *well-nested dependency trees with block-degree at most 2*. This class was first introduced by Bodirsky et al. (2005), who showed that it corresponds, in a natural way, to the class of derivation trees of lexicalized tree-adjoining grammars (Joshi and Schabes, 1997). While there are linguistic arguments against the restriction to this class (Maier and Lichte, 2011; Chen-Main and Joshi, 2010), Kuhlmann and Nivre (2006) found that it has excellent coverage on standard data sets. Assuming an arc-factored model, well-nested dependency trees with block-degree ≤ 2 can be parsed in time $\mathcal{O}(n^7)$ using the algorithm of Gómez-Rodríguez et al. (2011). Recently, Pitler et al. (2012) have shown that if an additional restriction called *1-inherit* is imposed, parsing can be done in time $\mathcal{O}(n^6)$, without any additional loss in coverage on standard data sets.

Standard context-free parsing methods, when adapted to the parsing of projective trees, provide $\mathcal{O}(n^5)$ time complexity. The $\mathcal{O}(n^3)$ time result reported by Eisner (1997) has been obtained by exploiting more sophisticated dynamic programming techniques that 'split' dependency trees at the position of their heads, in order to save bookkeeping. Splitting techniques have also been exploited to speed up parsing time for other lexicalized formalisms, such as bilexical context-free grammars and head automata (Eisner and Satta, 1999). However, to our knowledge no attempt has been made in the literature to extend these techniques to non-projective dependency parsing.

In this article we leverage the central idea from Eisner's algorithm and extend it to the class of well-nested dependency trees with block-degree at most 2.

Transactions of the Association for Computational Linguistics, 1 (2013) 267–278. Action Editor: Brian Roark.
Submitted 3/2013; Published 7/2013.

We introduce a structural property, called *head-split*, that allows us to split these trees at the positions of their heads. The property is restrictive, meaning that it reduces the class of trees that can be generated. However, we show that the restriction to head-split trees comes at no significant loss in coverage, and it allows parsing in time $\mathcal{O}(n^6)$, an asymptotic improvement of one order of magnitude over the algorithm by Gómez-Rodríguez et al. (2011) for the unrestricted class. We also show that restricting the class of head-split trees by imposing the already mentioned 1-inherit property does not cause any additional loss in coverage, and that parsing for the combined class is possible in time $\mathcal{O}(n^5)$, one order of magnitude faster than the algorithm by Pitler et al. (2012) for the 1-inherit class without the head-split condition.

The above results have consequences also for the parsing of other related formalisms, such as the already mentioned lexicalized tree-adjoining grammars. This will be discussed in the final section.

2 Head Splitting

To introduce the basic idea of this article, we briefly discuss in this section two well-known algorithms for computing the set of all projective dependency trees for a given input sentence: the naïve, CKY-style algorithm, and the improved algorithm with head splitting, in the version of Eisner and Satta (1999).[1]

CKY parsing The CKY-style algorithm works in a pure bottom-up way, building dependency trees by combining subtrees. Assuming an input string $w = a_1 \cdots a_n$, $n \geq 1$, each subtree t is represented by means of a finite signature $[i, j, h]$, called *item*, where i, j are the boundary positions of t's span over w and h is the position of t's root. This is the only information we need in order to combine subtrees under the arc-factored model. Note that the number of possible signatures is $\mathcal{O}(n^3)$.

The main step of the algorithm is displayed in Figure 1(a). Here we introduce the graphical convention, used throughout this article, of representing a subtree by a shaded area, with an horizontal line indicating the spanned fragment of the input string, and of marking the position of the head by a bullet. The illustrated step attaches a tree with signature $[k, j, d]$ as a dependent of a tree with signature $[i, k, h]$. There can be $\mathcal{O}(n^5)$ instantiations of this step, and this is also the running time of the algorithm.

[1] Eisner (1997) describes a slightly different algorithm.

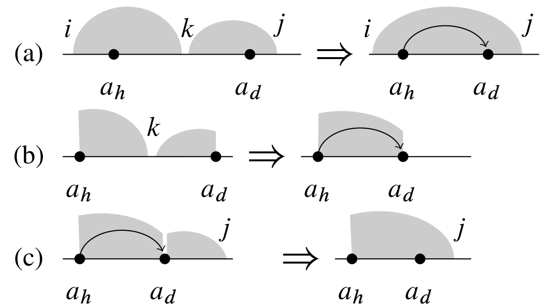

Figure 1: Basic steps for (a) the CKY-style algorithm and (b, c) the head splitting algorithm.

Eisner's algorithm Eisner and Satta (1999) improve over the CKY algorithm by reducing the number of position records in an item. They do this by 'splitting' each tree into a left and a right fragment, so that the head is always placed at one of the two boundary positions of a fragment, as opposed to being placed at an internal position. In this way items need only two indices. Left and right fragments can be processed independently, and merged afterwards.

Let us consider a right fragment t with head a_h. Attachment at t of a right dependent tree with head a_d is now performed in two steps. The first step attaches a left fragment with head a_d, as in Figure 1(b). This results in a new type of fragment/item that has both heads a_h and a_d placed at its boundaries. The second step attaches a right fragment with head a_d, as in Figure 1(c). The number of possible instantiations of these steps, and the asymptotic runtime of the algorithm, is $\mathcal{O}(n^3)$.

In this article we extend the splitting technique to the class of well-nested dependency trees with block-degree at most 2. This amounts to defining a factorization for these trees into fragments, each with its own head at one of its boundary positions, along with some unfolding of the attachment operation into intermediate steps. While for projective trees head splitting can be done without any loss in coverage, for the extended class head splitting turns out to be a proper restriction. The empirical relevance of this will be discussed in §7.

3 Head-Split Trees

In this section we introduce the class of well-nested dependency trees with block-degree at most 2, and define the subclass of head-split dependency trees.

3.1 Preliminaries

For non-negative integers i, j we write $[i, j]$ to denote the set $\{i, i+1, \ldots, j\}$; when $i > j$, $[i, j]$ is the empty set. For a string $w = a_1 \cdots a_n$, where $n \geq 1$ and each a_i is a lexical token, and for $i, j \in [0, n]$ with $i \leq j$, we write $w_{i,j}$ to denote the substring $a_{i+1} \cdots a_j$ of w; $w_{i,i}$ is the empty string.

A **dependency tree** t over w is a directed tree whose nodes are a subset of the tokens a_i in w and whose arcs encode a dependency relation between two nodes. We write $a_i \to a_j$ to denote the arc (a_i, a_j) in t; here, the node a_i is the head, and the node a_j is the dependent. If each token a_i, $i \in [1, n]$, is a node of t, then t is called **complete**. Sometimes we write t_{a_i} to emphasize that tree t is rooted in node a_i. If a_i is a node of t, we also write $t[a_i]$ to denote the subtree of t composed by node a_i as its root and all of its descendant nodes.

The nodes of t uniquely identify a set of maximal substrings of w, that is, substrings separated by tokens not in t. The sequence of such substrings, ordered from left to right, is the **yield** of t, written $yd(t)$. Let a_i be some node of t. The **block-degree** of a_i in t, written $bd(a_i, t)$, is defined as the number of string components of $yd(t[a_i])$. The block-degree of t, written $bd(t)$, is the maximal block-degree of its nodes. Tree t is **non-projective** if $bd(t) > 1$. Tree t is **well-nested** if, for each node a_i of t and for every pair of outgoing dependencies $a_i \to a_{d_1}$ and $a_i \to a_{d_2}$, the string components of $yd(t[a_{d_1}])$ and $yd(t[a_{d_2}])$ do not 'interleave' in w. More precisely, it is required that, if some component of $yd(t[a_{d_i}])$, $i \in [1, 2]$, occurs in w in between two components s_1, s_2 of $yd(t[a_{d_j}])$, $j \in [1, 2]$ and $j \neq i$, then *all* components of $yd(t[a_{d_i}])$ occur in between s_1, s_2.

Throughout this article, whenever we consider a dependency tree t we always implicitly assume that t is over w, that t has block-degree at most 2, and that t is well-nested. Let t_{a_i} be such a tree, with $bd(a_i, t_{a_i}) = 2$. We call the portion of w in between the two substrings of $yd(t_{a_i})$ the **gap** of t_{a_i}, denoted by $gap(t_{a_i})$.

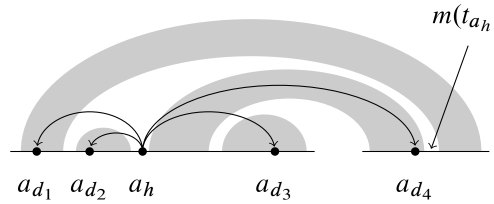

Figure 2: Example of a node a_h with block-degree 2 in a non-projective, well-nested dependency tree t_{a_h}. Integer $m(t_{a_h})$, defined in §3.2, is also marked.

Example 1 Figure 2 schematically depicts a well-nested tree t_{a_h} with block-degree 2; we have marked the root node a_h and its dependent nodes a_{d_i}. For each node a_{d_i}, a shaded area highlights $t[a_{d_i}]$. We have $bd(a_h, t_{a_h}) = bd(a_{d_1}, t_{a_h}) = bd(a_{d_4}, t_{a_h}) = 2$ and $bd(a_{d_2}, t_{a_h}) = bd(a_{d_3}, t_{a_h}) = 1$. □

3.2 The Head-Split Property

We say that a dependency tree t has the **head-split** property if it satisfies the following condition. Let $a_h \to a_d$ be any dependency in t with $bd(a_h, t) = bd(a_d, t) = 2$. Whenever $gap(t[a_d])$ contains a_h, it must also contain $gap(t[a_h])$. Intuitively, this means that if $yd(t[a_d])$ 'crosses over' the lexical token a_h in w, then $yd(t[a_d])$ must also 'cross over' $gap(t[a_h])$.

Example 2 Dependency $a_h \to a_{d_1}$ in Figure 3 violates the head-split condition, since $yd(t[a_{d_1}])$ crosses over the lexical token a_h in w, but does not cross over $gap(t[a_h])$. The remaining outgoing dependencies of a_h trivially satisfy the head-split condition, since the child nodes have block-degree 1. □

Let t_{a_h} be a dependency tree satisfying the head-split property and with $bd(a_h, t_{a_h}) = 2$. We specify below a construction that 'splits' t_{a_h} with respect to the position of the head a_h in $yd(t_{a_h})$, resulting in two dependency trees sharing the root a_h and having all of the remaining nodes forming two disjoint sets. Furthermore, the resulting trees have block-degree at most 2.

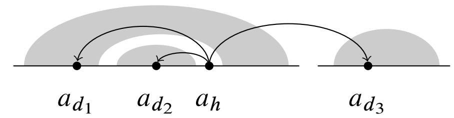

Figure 3: Arc $a_h \to a_{d_1}$ violates the head-split condition.

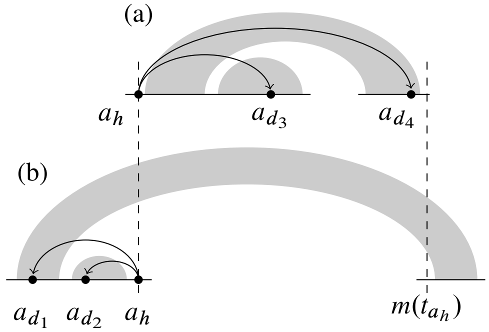

Figure 4: Lower tree (a) and upper tree (b) fragments for the dependency tree in Figure 2.

Let $yd(t_{a_h}) = \langle w_{i,j}, w_{p,q} \rangle$ and assume that a_h is placed within $w_{i,j}$. (A symmetric construction should be used in case a_h is placed within $w_{p,q}$.) The **mirror image** of a_h with respect to $gap(t_{a_h})$, written $m(t_{a_h})$, is the largest integer in $[p, q]$ such that there are no dependencies linking nodes in $w_{i,h-1}$ and nodes in $w_{p,m(t_{a_h})}$ and there are no dependencies linking nodes in $w_{h,j}$ and nodes in $w_{m(t_{a_h}),q}$. It is not hard to see that such an integer always exists, since t_{a_h} is well-nested.

We classify every dependent a_d of a_h as being an 'upper' dependent or a 'lower' dependent of a_h, according to the following conditions: (i) If $d \in [i, h-1] \cup [m(t_{a_h})+1, q]$, then a_d is an upper dependent of a_h. (ii) If $d \in [h+1, j] \cup [p, m(t_{a_h})]$, then a_d is a lower dependent of a_h.

The **upper tree** of t_{a_h} is the dependency tree rooted in a_h and composed of all dependencies $a_h \rightarrow a_d$ in t_{a_h} with a_d an upper dependent of a_h, along with all subtrees $t_{a_h}[a_d]$ rooted in those dependents. Similarly, the **lower tree** of t_{a_h} is the dependency tree rooted in a_h and composed of all dependencies $a_h \rightarrow a_d$ in t_{a_h} with a_d a lower dependent of a_h, along with all subtrees $t_{a_h}[a_d]$ rooted in those dependents. As a general convention, in this article we write t_{U,a_h} and t_{L,a_h} to denote the upper and the lower trees of t_{a_h}, respectively. Note that, in some degenerate cases, the set of lower or upper dependents may be empty; then t_{U,a_h} or t_{L,a_h} consists of the root node a_h only.

Example 3 Consider the tree t_{a_h} displayed in Figure 2. Integer $m(t_{a_h})$ denotes the boundary between the right component of $yd(t_{a_h}[a_{d_4}])$ and the right component of $yd(t_{a_h}[a_{d_1}])$. Nodes a_{d_3} and a_{d_4} are lower dependents, and nodes a_{d_1} and a_{d_2} are upper dependents. Trees t_{L,a_h} and t_{U,a_h} are displayed in Figure 4 (a) and (b), respectively. □

The importance of the head-split property can be informally explained as follows. Let $a_h \rightarrow a_d$ be a dependency in t_{a_h}. When we take apart the upper and the lower trees of t_{a_h}, the entire subtree $t_{a_h}[a_d]$ ends up in either of these two fragments. This allows us to represent upper and lower fragments for some head independently of the other, and to freely recombine them. More formally, our algorithms will make use of the following three properties, stated here without any formal proof:

P1 Trees t_{U,a_h} and t_{L,a_h} are well-nested, have block-degree ≤ 2, and satisfy the head-split property.

P2 Trees t_{U,a_h} and t_{L,a_h} have their head a_h always placed at one of the boundaries in their yields.

P3 Let t'_{U,a_h} and t''_{L,a_h} be the upper and lower trees of distinct trees t'_{a_h} and t''_{a_h}, respectively. If $m(t'_{a_h}) = m(t''_{a_h})$, then there exists a tree t_{a_h} such that $t_{U,a_h} = t'_{U,a_h}$ and $t_{L,a_h} = t''_{L,a_h}$.

4 Parsing Items

Let $w = a_1 \cdots a_n$, $n \geq 1$, be the input string. We need to compactly represent trees that span substrings of w by recording only the information that is needed to combine these trees into larger trees during the parsing process. We do this by associating each tree with a signature, called **item**, which is a tuple $[i, j, p, q, h]_X$, where $h \in [1, n]$ identifies the token a_h, i, j with $0 \leq i \leq j \leq n$ identify a substring $w_{i,j}$, and p, q with $j < p \leq q \leq n$ identify a substring $w_{p,q}$. We also use the special setting $p = q = -$.

The intended meaning is that each item represents some tree t_{a_h}. If $p, q \neq -$ then $yd(t_{a_h}) = \langle w_{i,j}, w_{p,q} \rangle$. If $p, q = -$ then

$$yd(t_{a_h}) = \begin{cases} \langle w_{i,j} \rangle & \text{if } h \in [i+1, j] \\ \langle w_{h,h}, w_{i,j} \rangle & \text{if } h < i \\ \langle w_{i,j}, w_{h,h} \rangle & \text{if } h > j+1 \end{cases}$$

The two cases $h < i$ and $h > j+1$ above will be used when the root node a_h of t_{a_h} has not yet collected all of its dependents.

Note that $h \in \{i, j+1\}$ is not used in the definition of item. This is meant to avoid different items representing the same dependency tree, which is undesired for the specification of our algorithm. As an example, items $[i, j, -, -, i+1]_X$ and $[i+1, j, -, -, i+1]_X$ both represent a dependency tree $t_{a_{i+1}}$ with $yd(t_{a_{i+1}}) = \langle w_{i,j} \rangle$. This and other similar cases are avoided by the ban against $h \in \{i, j+1\}$, which amounts to imposing some normal form for items. In our example, only item $[i, j, -, -, i+1]_X$ is a valid signature.

Finally, we distinguish among several item types, indicated by the value of subscript X. These types are specific to each parsing algorithm, and will be defined in later sections.

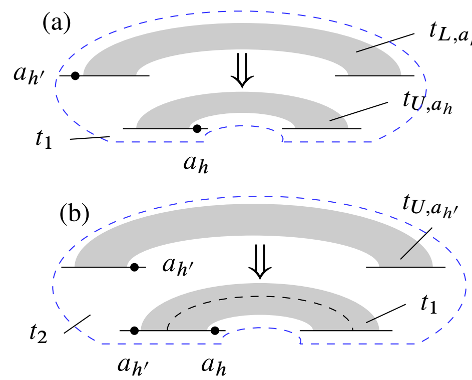

Figure 5: Two step attachment of $t_{a_{h'}}$ at t_{U,a_h}: (a) attachment of $t_{L,a_{h'}}$; (b) attachment of $t_{U,a_{h'}}$.

5 Parsing of Head-Split Trees

We present in this section our first tabular algorithm for computing the set of all dependency trees for an input sentence w that have the head-split property, under the arc-factored model. Recall that t_{a_i} denotes a tree with root a_i, and t_{L,a_i} and t_{U,a_i} are the lower and upper trees of t_{a_i}. The steps of the algorithm are specified by means of deduction rules over items, following the approach of Shieber et al. (1995).

5.1 Basic Idea

Our algorithm builds trees step by step, by attaching a tree $t_{a_{h'}}$ as a dependent of a tree t_{a_h} and creating the new dependency $a_h \to a_{h'}$. Computationally, the worst case for this operation is when both t_{a_h} and $t_{a_{h'}}$ have a gap; then, for each tree we need to keep a record of the four boundaries, along with the position of the head, as done by Gómez-Rodríguez et al. (2011). However, if we are interested in parsing trees that satisfy the head-split property, we can avoid representing a tree with a gap by means of a single item. We instead follow the general idea of §2 for projective parsing, and use different items for the upper and the lower trees of the source tree.

When we need to attach $t_{a_{h'}}$ as an upper dependent of t_{a_h}, defined as in §3.2, we perform two consecutive steps. First, we attach $t_{L,a_{h'}}$ to t_{U,a_h}, resulting in a new intermediate tree t_1. As a second step, we attach $t_{U,a_{h'}}$ to t_1, resulting in a new tree t_2 which is t_{U,a_h} with $t_{a_{h'}}$ attached as an upper dependent, as desired. Both steps are depicted in Figure 5; here we introduce the convention of indicating tree grouping through a dashed line. A symmetric procedure can be used to attach $t_{a_{h'}}$ as a lower dependent to t_{L,a_h}. The correctness of the two step approach follows from properties P1 and P3 in §3.2.

By property P2 in §3.2, in both steps above the lexical heads a_h and $a_{h'}$ can be read from the boundaries of the involved trees. Then these steps can be implemented more efficiently than the naïve method of attaching $t_{a_{h'}}$ to t_{a_h} in a single step. A more detailed computational analysis will be provided in §5.7. To simplify the presentation, we restrict the use of head splitting to trees with a gap and parse trees with no gap with the naïve method; this does not affect the computational complexity.

5.2 Item Types

We distinguish five different types of items, indicated by the subscript $X \in \{0, L, U, /L, /U\}$, as described in what follows.

- If $X = 0$, we have $p = q = -$ and $yd(a_h)$ is specified as in §4.
- If $X = L$, we use the item to represent some lower tree. We have therefore $p, q \neq -$ and $h \in \{i+1, q\}$.
- If $X = U$, we use the item to represent some upper tree. We have therefore $p, q \neq -$ and $h \in \{j, p+1\}$.
- If $X = /L$ or $X = /U$, we use the item to represent some intermediate step in the parsing process, in which only the lower or upper tree of some dependent has been collected by the head a_h, and we are still missing the upper ($/U$) or the lower ($/L$) tree.

We further specialize symbol $/U$ by writing $/U_<$ ($/U_>$) to indicate that the missing upper tree should have its head to the left (right) of its gap. We also use $/L_<$ and $/L_>$ with a similar meaning.

5.3 Item Normal Form

It could happen that our algorithm produces items of type 0 that do not satisfy the normal form condition discussed in §4. To avoid this problem, we assume that every item of type 0 that is produced by the algorithm is converted into an equivalent normal form item, by means of the following rules:

$$\frac{[i, j, -, -, i]_0}{[i-1, j, -, -, i]_0} \tag{1}$$

$$\frac{[i, j, -, -, j+1]_0}{[i, j+1, -, -, j+1]_0} \tag{2}$$

5.4 Items of Type 0

We start with deduction rules that produce items of type 0. As already mentioned, we do not apply the head splitting technique in this case.

The next rule creates trees with a single node, representing the head, and no dependents. The rule is actually an axiom (there is no antecedent) and the statement $i \in [1, n]$ is a side condition.

$$\frac{}{[i-1, i, -, -, i]_0} \; \{ i \in [1, n] \tag{3}$$

The next rule takes a tree headed in $a_{h'}$ and makes it a dependent of a new head a_h. This rule implements what has been called the 'hook trick'. The first side condition enforces that the tree headed in $a_{h'}$ has collected all of its dependents, as discussed in §4. The second side condition enforces that no cycle is created. We also write $a_h \to a_{h'}$ to indicate that a new dependency is created in the parse forest.

$$\frac{[i, j, -, -, h']_0}{[i, j, -, -, h]_0} \; \begin{cases} h' \in [i+1, j] \\ h \notin [i+1, j] \\ a_h \to a_{h'} \end{cases} \tag{4}$$

The next two rules combine gap-free dependents of the same head a_h.

$$\frac{[i, k, -, -, h]_0 \quad [k, j, -, -, h]_0}{[i, j, -, -, h]_0} \tag{5}$$

$$\frac{[i, h, -, -, h]_0 \quad [h-1, j, -, -, h]_0}{[i, j, -, -, h]_0} \tag{6}$$

We need the special case in (6) to deal with the concatenation of two items that share the head a_h at the concatenation point. Observe the apparent mismatch in step (6) between index h in the first antecedent and index $h-1$ in the second antecedent. This is because in our normal form, both the first and the second antecedent have already incorporated a copy of the shared head a_h.

The next two rules collect a dependent of a_h that wraps around the dependents that have already been collected. As already discussed, this operation is performed by two successive steps: We first collect the lower tree and then the upper tree. We present the case in which the shared head of the two trees is placed at the left of the gap. The case in which the head is placed at the right of the gap is symmetric.

$$\frac{\begin{array}{c}[i', j', -, -, h]_0 \\ [i, i', j', j, i+1]_L\end{array}}{[i, j, -, -, h]_{/U_<}} \; \begin{cases} h \notin [i+1, i'] \\ \quad \cup [j'+1, j] \end{cases} \tag{7}$$

$$\frac{\begin{array}{c}[i', j', -, -, h]_{/U_<} \\ [i, i'+1, j', j, i'+1]_U\end{array}}{[i, j, -, -, h]_0} \; \begin{cases} h \notin [i+1, i'+1] \\ \quad \cup [j'+1, j] \\ a_h \to a_{i'+1} \end{cases} \tag{8}$$

Again, there is an overlap in rule (8) between the two antecedents, due to the fact that both items have already incorporated copies of the same head.

5.5 Items of Type U

We now consider the deduction rules that are needed to process upper trees. Throughout this subsection we assume that the head of the upper tree is placed at the left of the gap. The other case is symmetric. The next rule creates an upper tree with a single node, representing its head, and no dependents. We construct an item for all possible right gap boundaries j.

$$\frac{}{[i-1, i, j, j, i]_U} \; \begin{cases} i \in [1, n] \\ j \in [i+1, n] \end{cases} \tag{9}$$

The next rule adds to an upper tree a group of new dependents that do not have any gap. We present the case in which the new dependents are placed at the left of the gap of the upper tree.

$$\frac{[i, i', -, -, j]_0 \quad [i', j, p, q, j]_U}{[i, j, p, q, j]_U} \tag{10}$$

The next two rules collect a new dependent that wraps around the upper tree. Again, this operation is performed by two successive steps: We first collect the lower tree, then the upper tree. We present the case in which the shared head of the two trees is placed at the left of the gap.

$$\frac{[i', j, p, q', j]_U \quad [i, i', q', q, i+1]_L}{[i, j, p, q, j]_{/U_<}} \tag{11}$$

$$\frac{\begin{array}{c}[i', j, p, q', j]_{/U_<} \\ [i, i'+1, q', q, i'+1]_U\end{array}}{[i, j, p, q, j]_U} \; \{a_j \to a_{i'+1} \tag{12}$$

5.6 Items of Type L

So far we have always expanded items (type 0 or U) at their external boundaries. When dealing with lower trees, we have to reverse this strategy and expand items (type L) at their internal boundaries. Apart from this difference, the deduction rules below are entirely symmetric to those in §5.5. Again, we assume that the head of the lower tree is placed at the left of the gap, the other case being symmetric. Our first rule creates a lower tree with a single node, representing its head. We blindly guess the right boundary of the gap of such a tree.

$$\frac{}{[i-1, i, j, j, i]_L} \; \begin{cases} i \in [1, n] \\ j \in [i+1, n] \end{cases} \tag{13}$$

The next rule adds to a lower tree a group of new dependents that do not have any gap. We present the case in which the new dependents are placed at the left of the gap of the lower tree.

$$\frac{[j', j, -, -, i+1]_0 \quad [i, j', p, q, i+1]_L}{[i, j, p, q, i+1]_L} \tag{14}$$

The next two rules collect a new dependent with a gap and embed it within the gap of our lower tree, creating a new dependency. Again, this operation is performed by two successive steps, and we present the case in which the common head of the lower and upper trees that are embedded is placed at the left of the gap, the other case being symmetric.

$$\frac{[i, j', p', q, i+1]_L \quad [j', j, p, p', j]_U}{[i, j, p, q, i+1]_{/L_<}} \tag{15}$$

$$\frac{\begin{array}{c}[i, j', p', q, i+1]_{/L_<} \\ [j'-1, j, p, p', j']_L\end{array}}{[i, j, p, q, i+1]_L} \; \{a_{i+1} \to a_{j'} \tag{16}$$

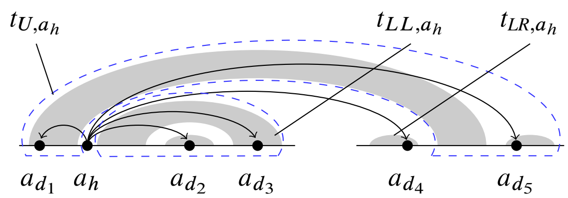

Figure 6: Node a_h satisfies both the 1-inherit and head-split conditions. Accordingly, tree t_{a_h} can be split into three fragments t_{U,a_h}, t_{LL,a_h} and t_{LR,a_h}.

5.7 Runtime

The algorithm runs in time $\mathcal{O}(n^6)$, where n is the length of the input sentence. The worst case is due to deduction rules that combine two items, each of which represents trees with one gap. For instance, rule (11) involves six free indices ranging over $[1, n]$, and thus could be instantiated $\mathcal{O}(n^6)$ many times. If the head-split property does not hold, attachment of a dependent in one step results in time $\mathcal{O}(n^7)$, as seen for instance in Gómez-Rodríguez et al. (2011).

6 Parsing of 1-Inherit Head-Split Trees

In this section we specialize the parsing algorithm of §5 to a new, more efficient algorithm for a restricted class of trees.

6.1 1-Inherit Head-Split Trees

Pitler et al. (2012) introduce a restriction on well-nested dependency trees with block-degree at most 2. A tree t satisfies the **1-inherit** property if, for every node a_h in t with $bd(a_h, t) = 2$, there is at most one dependency $a_h \to a_{d^*}$ such that $gap(t[a_{d^*}])$ contains $gap(t[a_h])$. Informally, this means that $yd(t[a_{d^*}])$ 'crosses over' $gap(t[a_h])$, and we say that a_{d^*} 'inherits' the gap of a_h. In this section we investigate the parsing of head-split trees that also have the 1-inherit property.

Example 4 Figure 6 shows a head node a_h along with dependents a_{d_i}, satisfying the head-split condition. Only $t_{a_{d_1}}$ has its yield crossing over $gap(t_{a_h})$. Thus a_h also satisfies the 1-inherit condition. □

6.2 Basic Idea

Let t_{a_h} be some tree satisfying both the head-split property and the 1-inherit propery. Assume that the dependent node a_{d^*} which inherits the gap of t_{a_h} is placed within t_{U,a_h}. This means that, for every dependency $a_h \rightarrow a_d$ in t_{L,a_h}, $yd(t[a_d])$ does not cross over $gap(t_{L,a_h})$. Then we can further split t_{L,a_h} into two trees, both with root a_h. We call these two trees the **lower-left** tree, written t_{LL,a_h}, and the **lower-right** tree, written t_{LR,a_h}; see again Figure 6.

The basic idea behind our algorithm is to split t_{a_h} into three dependency trees t_{U,a_h}, t_{LL,a_h} and t_{LR,a_h}, all sharing the same root a_h. This means that t_{a_h} can be attached to an existing tree through three successive steps, each processing one of the three trees above. The correctness of this procedure follows from a straightforward extension of properties P1 and P3 from §3.2, stating that the tree fragments t_{U,a_h}, t_{LL,a_h} and t_{LR,a_h} can be represented and processed one independently of the others, and freely combined if certain conditions are satisfied by their yields.

In case a_{d^*} is placed within t_{L,a_h}, we introduce the **upper-left** and the **upper-right** trees, written t_{UL,a_h} and t_{UR,a_h}, and apply a similar idea.

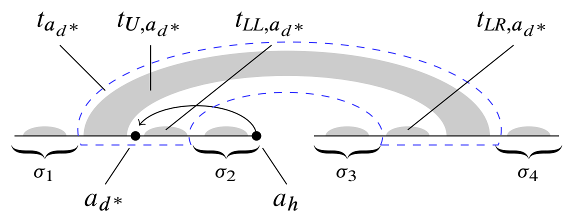

Figure 7: Tree t_{U,a_h} is decomposed into $t_{a_{d*}}$ and subtrees covering substrings σ_i, $i \in [1,4]$. Tree $t_{a_{d*}}$ is in turn decomposed into three fragments (trees $t_{LL,a_{d*}}$, $t_{LR,a_{d*}}$, and $t_{U,a_{d*}}$ in this example).

6.3 Item Types

When processing an attachment, the order in which the algorithm assembles the three tree fragments of t_{a_h} defined in §6.2 is not always the same. Such an order is chosen on the basis of where the head a_h and the dependent a_{d^*} inheriting the gap are placed within the involved trees. As a consequence, in our algorithm we need to represent several intermediate parsing states. Besides the item types from §5.2, we therefore need additional types. The specification of these new item types is rather technical, and is therefore delayed until we introduce the relevant deduction rules.

6.4 Items of Type 0

We start with the deduction rules for parsing of trees t_{LL,a_h} and t_{LR,a_h}; trees t_{UL,a_h} and t_{UR,a_h} can be treated symmetrically. The yields of t_{LL,a_h} and t_{LR,a_h} have the form specified in §4 for the case $p = q = -$. We can therefore use items of type 0 to parse these trees, adopting a strategy similar to the one in §5.4. The main difference is that, when a tree $t_{a_{h'}}$ with a gap is attached as a dependent to the head a_h, we use three consecutive steps, each processing a single fragment of $t_{a_{h'}}$. We assume below that $t_{a_{h'}}$ can be split into trees $t_{U,a_{h'}}$, $t_{LL,a_{h'}}$ and $t_{LR,a_{h'}}$, the other case can be treated in a similar way.

We use rules (3), (4) and (5) from §5.4. Since in the trees t_{LL,a_h} and t_{LR,a_h} the head is never placed in the middle of the yield, rule (6) is not needed now and it can safely be discarded. Rule (7), attaching a lower tree, needs to be replaced by two new rules, processing a lower-left and a lower-right tree. We assume here that the common head of these trees is placed at the left boundary of the lower-left tree; we leave out the symmetric case.

$$\frac{\begin{array}{c}[i, i', -, -, i+1]_0 \\ [i', j, -, -, h]_0\end{array}}{[i, j, -, -, h]_{/LR_<}} \left\{h \notin [i+1, i'] \right. \tag{17}$$

$$\frac{\begin{array}{c}[j', j, -, -, i+1]_0 \\ [i, j', -, -, h]_{/LR_<}\end{array}}{[i, j, -, -, h]_{/U_<}} \left\{h \notin [j'+1, j] \right. \tag{18}$$

The first antecedent in (17) encodes a lower-left tree with its head at the left boundary. The consequent item has then the new type $/LR_<$, meaning that a lower-right tree is missing that must have its head at the left. The first antecedent in (18) provides the missing lower-right tree, having the same head as the already incorporated lower-left tree. After these rules are applied, rule (8) from §5.4 can be applied to the consequent item of (18). This completes the attachment of a 'wrapping' dependent of a_h, with the incorporation of the missing upper tree and with the construction of the new dependency.

6.5 Items of Type U

We now assume that node a_{d^*} is realized within t_{U,a_h}, so that t_{a_h} can be split into trees t_{U,a_h}, t_{LL,a_h} and t_{LR,a_h}. We provide deduction rules to parse of t_{U,a_h}; this is the most involved part of the algorithm. In case a_{d^*} is realized within t_{L,a_h}, t_{a_h} must be split into t_{L,a_h}, t_{UL,a_h} and t_{UR,a_h}, and a symmetrical strategy can be applied to parse t_{L,a_h}.

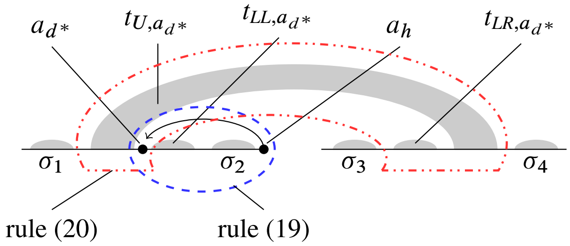

Figure 8: Decomposition of t_{U,a_h} as in Figure 7, with highlighted application of rules (19) and (20).

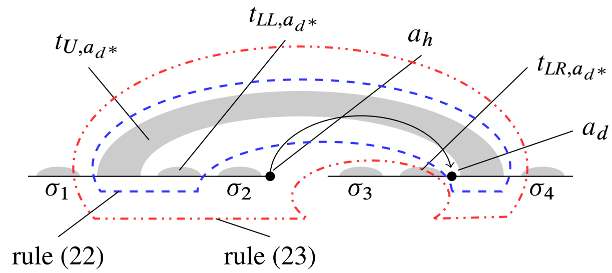

Figure 9: Decomposition of t_{U,a_h} as in Figure 7, with highlighted application of rules (22) and (23).

We start by observing that $yd(t_{a_{d*}})$ splits $yd(t_{U,a_h})$ into at most four substrings σ_i; see Figure 7.[2] Because of the well-nested property, within the tree t_{U,a_h} each dependent of a_h other than a_{d*} has a yield that is entirely placed within one of the σ_i's substrings. This means that each substring σ_i can be parsed independently of the other substrings.

As a first step in the process of parsing t_{U,a_h}, we parse each substring σ_i. We do this following the parsing strategy specified in §6.4. As a second step, we assume that each of the three fragments resulting from the decomposition of tree $t_{a_{d*}}$ has already been parsed; see again Figure 7. We then 'merge' these three fragments and the trees for segments σ_i's into a complete parse tree representing t_{U,a_h}. This is described in detail in what follows.

We assume that a_h is placed at the left of the gap of t_{U,a_h} (the right case being symmetrical) and we distinguish four cases, depending on the two ways in which $t_{a_{d*}}$ can be split, and the two side positions of the head a_{d*} with respect to $gap(t_{a_{d*}})$.

Case 1 We assume that $t_{a_{d*}}$ can be split into trees $t_{U,a_{d*}}$, $t_{LL,a_{d*}}$, $t_{LR,a_{d*}}$, and the head a_{d*} is placed at the left of $gap(t_{a_{d*}})$; see again Figure 7.

Rule (19) below combines $t_{LL,a_{d*}}$ with a parse for segment σ_2, which has its head a_h placed at its right boundary; see Figure 8 for a graphical representation of rule (19). The result is an item of the new type *HH*. This item is used to represent an intermediate tree fragment with root of block-degree 1, where both the left and the right boundaries are heads; a dependency between these heads will be constructed later.

$$\frac{[i, i', -, -, i+1]_0 \quad [i', j, -, -, j]_0}{[i, j, -, -, j]_{HH}} \tag{19}$$

Rule (20) combines $t_{U,a_{d*}}$ with a type 0 item representing $t_{LR,a_{d*}}$; see again Figure 8. Note that this combination operation expands an upper tree at one of its internal boundaries, something that was not possible with the rules specified in §5.5.

$$\frac{[i, j, p', q, j]_U \quad [p, p', -, -, j]_0}{[i, j, p, q, j]_U} \tag{20}$$

Finally, we combine the consequents of (19) and (20), and process the dependency that was left pending in the item of type *HH*.

$$\frac{\begin{array}{c}[i, j', p, q, j']_U \\ [j'-1, j, -, -, j]_{HH}\end{array}}{[i, j, p, q, j]_U} \, \{a_j \to a_{j'} \tag{21}$$

After the above steps, parsing of t_{U,a_h} can be completed by combining item $[i, j, p, q, j]_U$ from (21) with items of type 0 representing parses for the substrings σ_1, σ_3 and σ_4.

Case 2 We assume that $t_{a_{d*}}$ can be split into trees $t_{U,a_{d*}}$, $t_{LL,a_{d*}}$, $t_{LR,a_{d*}}$, and the head a_{d*} is placed at the right of $gap(t_{a_{d*}})$, as depicted in Figure 9.

Rule (22) below, graphically represented in Figure 9, combines $t_{U,a_{d*}}$ with a type 0 item representing $t_{LL,a_{d*}}$. This can be viewed as the symmetric version of rule (20) of Case 1, expanding an upper tree at one of its internal boundaries.

$$\frac{[i, j', p, q, p+1]_U \quad [j', j, -, -, p+1]_0}{[i, j, p, q, p+1]_U} \tag{22}$$

[2]According to our definition of $m(t_{a_h})$ in §3.2, σ_3 is always the empty string. However, here we deal with the general formulation of the problem in order to claim in §8 that our algorithm can be directly adapted to parse some subclasses of lexicalized tree-adjoining grammars.

		Arabic		Czech		Danish		Dutch		Portuguese		Swedish	
Number of trees		1,460		72,703		5,190		13,349		9,071		11,042	
WN2	$\mathcal{O}(n^7)$	1,458	99.9%	72,321	99.5%	5,175	99.7%	12,896	96.6%	8,650	95.4%	10,955	99.2%
Classes considered in this paper													
WN2 + HS	$\mathcal{O}(n^6)$	1,457	99.8%	72,182	99.3%	5,174	99.7%	12,774	95.7%	8,648	95.3%	10,951	99.2%
WN2 + HS + 1I	$\mathcal{O}(n^5)$	1,457	99.8%	72,182	99.3%	5,174	99.7%	12,774	95.7%	8,648	95.3%	10,951	99.2%
Classes considered by Pitler et al. (2012)													
WN2 + 1I	$\mathcal{O}(n^6)$	1,458	99.9%	72,321	99.5%	5,175	99.7%	12,896	96.6%	8,650	95.4%	10,955	99.2%
WN2 + 0I	$\mathcal{O}(n^5)$	1,394	95.5%	70,695	97.2%	4,985	96.1%	12,068	90.4%	8,481	93.5%	10,787	97.7%
Projective	$\mathcal{O}(n^3)$	1,297	88.8%	55,872	76.8%	4,379	84.4%	8,484	63.6%	7,353	81.1%	9,963	90.2%

Table 1: Coverage of various classes of dependency trees on the training sets used in the CoNLL-X shared task (WN2 = well-nested, block-degree ≤ 2; HS = head-split; 1I = 1-inherit; 0I = 0-inherit, 'gap-minding')

Next, we combine the result of (22) with a parse for substring σ_2. The result is an item of the new type $/LR_>$. This item is used to represent an intermediate tree fragment that is missing a lower-right tree with its head at the right. In this fragment, two heads are left pending, and a dependency relation will be eventually established between them.

$$\frac{[i, j', p, q, p+1]_U \quad [j', j, -, -, j]_0}{[i, j, p, q, j]_{/LR_>}} \tag{23}$$

The next rule combines the consequent item of (23) with a tree $t_{LR,a_{d*}}$ having its head at the right boundary, and processes the dependency that was left pending in the $/LR_>$ item.

$$\frac{\begin{array}{c}[i, j, p', q, j]_{/LR_>} \\ [p, p'+1, -, -, p'+1]_0\end{array}}{[i, j, p, q, j]_U} \{a_j \to a_{p'+1} \tag{24}$$

After the above rules, parsing of t_{U,a_h} continues by combining the consequent item $[i, j, p, q, j]_U$ from rule (24) with items representing parses for the substrings σ_1, σ_3 and σ_4.

Cases 3 and 4 We informally discuss the cases in which $t_{a_{d*}}$ can be split into trees $t_{L,a_{d*}}$, $t_{UL,a_{d*}}$, $t_{UR,a_{d*}}$, for both positions of the head a_{d*} with respect to $gap(t_{a_{d*}})$. In both cases we can adopt a strategy similar to the one of Case 2.

We first expand $t_{L,a_{d*}}$ externally, at the side opposite to the head a_{d*}, with a tree fragment $t_{UL,a_{d*}}$ or $t_{UR,a_{d*}}$, similarly to rule (22) of Case 2. This results in a new fragment t_1. Next, we merge t_1 with a parse for σ_2 containing the head a_h, similarly to rule (23) of Case 2. This results in a new fragment t_2 where a dependency relation involving the heads a_{d*} and a_h is left pending. Finally, we merge t_2 with a missing tree $t_{UL,a_{d*}}$ or $t_{UR,a_{d*}}$, and process the pending dependency, similarly to rule (24). One should contrast this strategy with the alternative strategy adopted in Case 1, where the fragment of $t_{a_{d*}}$ having block-degree 2 *cannot* be merged with a parse for the segment containing the head a_h (σ_2 in Case 1), because of an intervening fragment of $t_{a_{d*}}$ with block-degree 1 ($t_{LL,a_{d*}}$ in Case 1).

Finally, if there is no node a_{d*} in t_{U,a_h} that inherits the gap of a_h, we can split t_{U,a_h} into two dependency trees, as we have done for t_{L,a_h} in §6.2, and parse the two fragments using the strategy of §6.4.

6.6 Runtime

Our algorithm runs in time $\mathcal{O}(n^5)$, where n is the length of the input sentence. The reason of the improvement with respect to the $\mathcal{O}(n^6)$ result of §5 is that we no longer have deduction rules where both antecedents represent trees with a gap. In the new algorithm, the worst case is due to rules where only one antecedent has a gap. This leads to rules involving a maximum of five indices, ranging over $[1, n]$. These rules can be instantiated in $\mathcal{O}(n^5)$ ways.

7 Empirical Coverage

We have seen that the restriction to head-split dependency trees enables us to parse these trees one order of magnitude faster than the class of well-nested dependency trees with block-degree at most 2.

In connection with the 1-inherit property, this even increases to two orders of magnitude. However, as already stated in §2, this improvement is paid for by a loss in coverage; for instance, trees of the form shown in Figure 3 cannot be parsed any longer.

7.1 Quantitative Evaluation

In order to assess the empirical loss in coverage that the restriction to head-split trees incurs, we evaluated the coverage of several classes of dependency trees on standard data sets. Following Pitler et al. (2012), we report in Table 1 figures for the training sets of six languages used in the CoNLL-X shared task on dependency parsing (Buchholz and Marsi, 2006). As we can see, the $\mathcal{O}(n^6)$ class of head-split trees has only slightly lower coverage on this data than the baseline class of well-nested dependency trees with block-degree at most 2. The losses are up to 0.2 percentage points on five of the six languages, and 0.9 points on the Dutch data. Our even more restricted $\mathcal{O}(n^5)$ class of 1-inherit head-split trees has the same coverage as our $\mathcal{O}(n^6)$ class, which is expected given the results of Pitler et al. (2012): Their $\mathcal{O}(n^6)$ class of 1-inherit trees has exactly the same coverage as the baseline (and thereby more coverage than our $\mathcal{O}(n^6)$ class). Interestingly though, their $\mathcal{O}(n^5)$ class of 'gap-minding' trees has a significantly smaller coverage than our $\mathcal{O}(n^5)$ class. We conclude that our class seems to strike a good balance between expressiveness and parsing complexity.

7.2 Qualitative Evaluation

While the original motivation behind introducing the head-split property was to improve parsing complexity, it is interesting to also discuss the linguistic relevance of this property. A first inspection of the structures that violate the head-split property revealed that many such violations disappear if one ignores gaps caused by punctuation. Some decisions about what nodes should function as the heads of punctuation symbols lead to more gaps than others. In order to quantify the implications of this, we recomputed the coverage of the class of head-split trees on data sets where we first removed all punctuation. The results are given in Table 2. We restrict ourselves to the five native dependency treebanks used in the CoNLL-X shared task, ignoring treebanks that have been converted from phrase structure representations.

	Arabic	Czech	Danish	Slovene	Turkish
with	1	139	1	2	2
without	1	46	0	0	2

Table 2: Violations against the head-split property (relative to the class of well-nested trees with block-degree ≤ 2) with and without punctuation.

We see that when we remove punctuation from the sentences, the number of violations against the head-split property at most decreases. For Danish and Slovene, removing punctuation even has the effect that *all* well-nested dependency trees with block-degree at most 2 become head-split. Overall, the absolute numbers of violations are extremely small—except for Czech, where we have 139 violations with and 46 without punctuation. A closer inspection of the Czech sentences reveals that many of these feature rather complex coordinations. Indeed, out of the 46 violations in the punctuation-free data, only 9 remain when one ignores those with coordination. For the remaining ones, we have not been able to identify any clear patterns.

8 Concluding Remarks

In this article we have extended head splitting techniques, originally developed for parsing of projective dependency trees, to two subclasses of well-nested dependency trees with block-degree at most 2. We have improved over the asymptotic runtime of two existing algorithms, at no significant loss in coverage. With the same goal of improving parsing efficiency for subclasses of non-projective trees, in very recent work Pitler et al. (2013) have proposed an $\mathcal{O}(n^4)$ time algorithm for a subclass of non-projective trees that are not well-nested, using an approach that is orthogonal to the one we have explored here.

Other than for dependency parsing, our results have also implications for mildly context-sensitive phrase structure formalisms. In particular, the algorithm of §5 can be adapted to parse a subclass of lexicalized tree-adjoining grammars, improving the result by Eisner and Satta (2000) from $\mathcal{O}(n^7)$ to $\mathcal{O}(n^6)$. Similarly, the algorithm of §6 can be adapted to parse a lexicalized version of the tree-adjoining grammars investigated by Satta and Schuler (1998), improving a naïve $\mathcal{O}(n^7)$ algorithm to $\mathcal{O}(n^5)$.

References

Manuel Bodirsky, Marco Kuhlmann, and Mathias Möhl. 2005. Well-nested drawings as models of syntactic structure. In *Proceedings of the 10th Conference on Formal Grammar (FG) and Ninth Meeting on Mathematics of Language (MOL)*, pages 195–203, Edinburgh, UK.

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

Xavier Carreras. 2007. Experiments with a higher-order projective dependency parser. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, pages 957–961, Prague, Czech Republic.

Joan Chen-Main and Aravind K. Joshi. 2010. Unavoidable ill-nestedness in natural language and the adequacy of tree local-MCTAG induced dependency structures. In *Proceedings of the Tenth International Conference on Tree Adjoining Grammars and Related Formalisms (TAG+)*, New Haven, USA.

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 (ACL)*, pages 457–464, College Park, MD, USA.

Jason Eisner and Giorgio Satta. 2000. A faster parsing algorithm for lexicalized Tree-Adjoining Grammars. In *Proceedings of the Fifth Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+)*, pages 14–19, Paris, France.

Jason Eisner. 1997. Bilexical grammars and a cubic-time probabilistic parser. In *Proceedings of the Fifth International Workshop on Parsing Technologies (IWPT)*, pages 54–65, Cambridge, MA, USA.

Carlos Gómez-Rodríguez, John Carroll, and David J. Weir. 2011. Dependency parsing schemata and mildly non-projective dependency parsing. *Computational Linguistics*, 37(3):541–586.

Aravind K. Joshi and Yves Schabes. 1997. Tree-Adjoining Grammars. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 69–123. Springer.

Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1–11, Uppsala, Sweden.

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

Wolfgang Maier and Timm Lichte. 2011. Characterizing discontinuity in constituent treebanks. In Philippe de Groote, Markus Egg, and Laura Kallmeyer, editors, *Formal Grammar. 14th International Conference, FG 2009, Bordeaux, France, July 25–26, 2009, Revised Selected Papers*, volume 5591 of *Lecture Notes in Computer Science*, pages 167–182. Springer.

Ryan McDonald and Giorgio Satta. 2007. On the complexity of non-projective data-driven dependency parsing. In *Proceedings of the 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.

Emily Pitler, Sampath Kannan, and Mitchell Marcus. 2012. Dynamic programming for higher order parsing of gap-minding trees. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing (EMNLP) and Computational Natural Language Learning (CoNLL)*, pages 478–488, Jeju Island, Republic of Korea.

Emily Pitler, Sampath Kannan, and Mitchell Marcus. 2013. Finding optimal 1-endpoint-crossing trees. *Transactions of the Association for Computational Linguistics*.

Giorgio Satta and William Schuler. 1998. Restrictions on tree adjoining languages. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics (ACL) and 17th International Conference on Computational Linguistics (COLING)*, pages 1176–1182, Montréal, Canada.

Stuart M. Shieber, Yves Schabes, and Fernando Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36.