A Crossing-Sensitive Third-Order Factorization for Dependency Parsing

Emily Pitler[*]
Google Research
76 9th Avenue
New York, NY 10011
epitler@google.com

Abstract

Parsers that parametrize over wider scopes are generally more accurate than edge-factored models. For graph-based non-projective parsers, wider factorizations have so far implied large increases in the computational complexity of the parsing problem. This paper introduces a "crossing-sensitive" generalization of a third-order factorization that trades off complexity in the *model* structure (i.e., scoring with features over multiple edges) with complexity in the *output* structure (i.e., producing crossing edges). Under this model, the optimal 1-Endpoint-Crossing tree can be found in $O(n^4)$ time, matching the asymptotic run-time of *both* the third-order *projective* parser and the *edge-factored* 1-Endpoint-Crossing parser. The crossing-sensitive third-order parser is significantly more accurate than the third-order projective parser under many experimental settings and significantly less accurate on none.

1 Introduction

Conditioning on wider syntactic contexts than simply individual head-modifier relationships improves parsing accuracy in a wide variety of parsers and frameworks (Charniak and Johnson, 2005; McDonald and Pereira, 2006; Hall, 2007; Carreras, 2007; Martins et al., 2009; Koo and Collins, 2010; Zhang and Nivre, 2011; Bohnet and Kuhn, 2012; Martins et al., 2013). This paper proposes a new graph-based dependency parser that efficiently produces the globally optimal dependency tree according to a third-order model (that includes features over grandparents and siblings in the tree) in the class of 1-Endpoint-Crossing trees (that includes all projective trees and the vast majority of *non-projective* structures seen in dependency treebanks).

Within graph-based *projective* parsing, the *third-order* parser of Koo and Collins (2010) has a runtime of $O(n^4)$, just one factor of n more expensive than the edge-factored model of Eisner (2000). Incorporating richer features *and* producing trees with crossing edges has traditionally been a challenge, however, for graph-based dependency parsers. If parsing is posed as the problem of finding the optimal scoring directed spanning tree, then the problem becomes NP-hard when trees are scored with a grandparent and/or sibling factorization (McDonald and Pereira, 2006; McDonald and Satta, 2007). For various definitions of mildly non-projective trees, even edge-factored versions are expensive, with *edge-factored* running times between $O(n^4)$ and $O(n^7)$ (Gómez-Rodríguez et al., 2011; Pitler et al., 2012; Pitler et al., 2013; Satta and Kuhlmann, 2013).

The third-order projective parser of Koo and Collins (2010) and the edge-factored 1-Endpoint-Crossing parser described in Pitler et al. (2013) have some similarities: both use $O(n^4)$ time and $O(n^3)$ space, using sub-problems over intervals with one exterior vertex, which are constructed using one free split point. The two parsers differ in *how the exterior vertex is used*: Koo and Collins (2010) use the exterior vertex to store a grandparent index, while Pitler et al. (2013) use the exterior vertex to introduce crossed edges between the point and

[*]The majority of this work was done while at the University of Pennsylvania.

Transactions of the Association for Computational Linguistics, 2 (2014) 41–54. Action Editor: Joakim Nivre.
Submitted 9/2013; Revised 11/2013; Published 2/2014. ©2014 Association for Computational Linguistics.

	Projective	1-Endpoint-Crossing
Edge	$O(n^3)$ Eisner (2000)	$O(n^4)$ Pitler et al. (2013)
CS-GSib	$O(n^4)$ Koo and Collins (2010)	$\mathbf{O(n^4)}$ This paper

Table 1: Parsing time for various output spaces and model factorizations. CS-GSib refers to the (crossing-sensitive) grand-sibling factorization described in this paper.

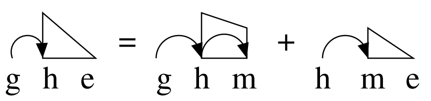
the interval. This paper proposes *merging* the two parsers to achieve the best of both worlds – producing the best tree in the wider range of 1-Endpoint-Crossing trees while incorporating the identity of the grandparent and/or sibling of the child in the score of an edge whenever the local neighborhood of the edge does not contain crossing edges. The crossing-sensitive grandparent-sibling 1-Endpoint-Crossing parser proposed here takes $O(n^4)$ time, matching the runtime of both the third-order projective parser and of the edge-factored 1-Endpoint-Crossing parser (see Table 1).

The parsing algorithms of Koo and Collins (2010) and Pitler et al. (2013) are reviewed in Section 2. The proposed crossing-sensitive factorization is defined in Section 3. The parsing algorithm that finds the optimal 1-Endpoint-Crossing tree according to this factorization is described in Section 4. The implemented parser is significantly more accurate than the third-order projective parser in a variety of languages and treebank representations (Section 5). Section 6 discusses the proposed approach in the context of prior work on non-projective parsing.

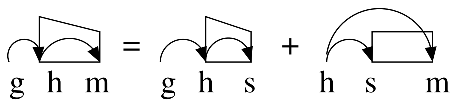
2 Preliminaries

In a *projective* dependency tree, each subtree forms one consecutive interval in the sequence of input words; equivalently (assuming an artificial root node placed as either the first or last token), when all edges are drawn in the half-plane above the sentence, no two edges *cross* (Kübler et al., 2009). Two vertex-disjoint edges *cross* if their endpoints interleave. A *1-Endpoint-Crossing* tree is a dependency tree such that for each edge, all edges that cross it share a common vertex (Pitler et al., 2013). Note that the class of projective trees is properly included within the class of 1-Endpoint-Crossing trees.

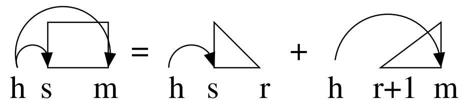
To avoid confusion between intervals and edges,



(a) m is the child of h that e is descended from



(b) The edge $\vec{e}_{hm}$ is added to the tree; s is m's adjacent inner sibling



(c) r is s's outermost descendant; $r+1$ is m's innermost descendant

Figure 1: Algorithm for grand-sibling projective parsing; the figures replicate Figure 6 in Koo and Collins (2010).

$\vec{e}_{ij}$ denotes the *directed edge* from i to j (i.e., i is the parent of j). Interval notation ((i,j), $[i,j]$, $(i,j]$, or $[i,j)$) is used to denote *sets of vertices* between i and j, with square brackets indicating closed intervals and round brackets indicating open intervals.

2.1 Grand-Sibling Projective Parsing

A grand-sibling factorization allows features over 4-tuples of (g,h,m,s), where h is the parent of m, g is m's grandparent, and s is m's adjacent inner sibling. Features over these grand-sibling 4-tuples are referred to as "third-order" because they scope over three *edges* simultaneously ($\vec{e}_{gh}$, $\vec{e}_{hs}$, and $\vec{e}_{hm}$). The parser of Koo and Collins (2010) produces the highest-scoring projective tree according to this grand-sibling model by adding an external grandparent index to each of the sub-problems used in the sibling factorization (McDonald and Pereira, 2006). Figure 6 in Koo and Collins (2010) provided a pictorial view of the algorithm; for convenience, it is replicated in Figure 1. An edge $\vec{e}_{hm}$ is added to the tree in the "trapezoid" step (Figure 1b); this allows the edge to be scored conditioned on m's grandparent (g) and its adjacent inner sibling (s), as all four relevant indices are accessible.

2.2 Edge-factored 1-Endpoint-Crossing Parsing

The edge-factored 1-Endpoint-Crossing parser of Pitler et al. (2013) produces the highest scoring 1-

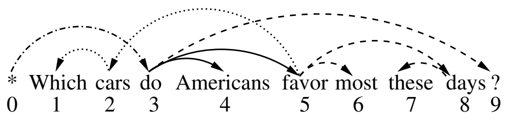


Figure 2: A 1-Endpoint-Crossing non-projective English sentence from the WSJ Penn Treebank (Marcus et al., 1993), converted to dependencies with PennConverter (Johansson and Nugues, 2007).

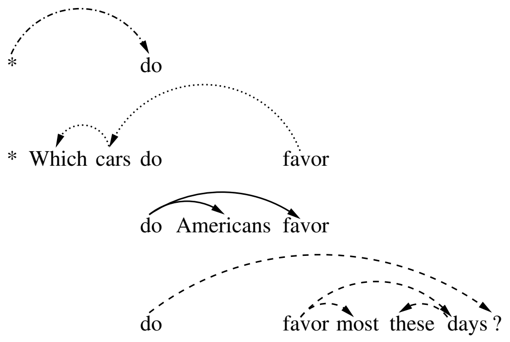


Figure 3: Constructing a 1-Endpoint-Crossing tree with intervals with one exterior vertex (Pitler et al., 2013).

Endpoint-Crossing tree with each edge $\vec{e}_{hm}$ scored according to $\mathrm{Score}(\mathrm{Edge}(h, m))$. The 1-Endpoint-Crossing property allows the tree to be built up in edge-disjoint pieces each consisting of intervals with one exterior point that has edges into the interval. For example, the tree in Figure 2 would be built up with the sub-problems shown in Figure 3.

To ensure that crossings *within* a sub-problem are consistent with the crossings that happen as a result of combination steps, the algorithm uses four different "types" of sub-problems, indicating whether the edges incident to the exterior point may be internally crossed by edges incident to the left boundary point (L), the right (R), either (LR), or neither (N). In Figure 3, the sub-problem over [*, *do*] $\cup$ {*favor*} would be of type R, and [*favor*, *?*] $\cup$ {*do*} of type L.

2.2.1 Naïve Approach to Including Grandparent Features

The example in Figure 3 illustrates the difficulty of incorporating grandparents into the scoring of all edges in 1-Endpoint-Crossing parsing. The vertex *favor* has a parent or child in *all three* of the sub-problems. In order to use grandparent scoring for the edges from *favor* to *favor*'s children in the other two sub-problems, we would need to augment the problems with the grandparent index *do*. We also must add the parent index *do* to the middle sub-problem to ensure consistency (i.e., that *do* is in fact the parent assigned). Thus, a first attempt to score all edges with grandparent features within 1-Endpoint-Crossing trees raises the runtime from $O(n^4)$ to $O(n^7)$ (all of the four indices need a "predicted parent" index; at least one edge is always implied so one of these additional indices can be dropped).

3 Crossing-Sensitive Factorization

Factorizations for projective dependency parsing have often been designed to allow efficient parsing. For example, the algorithms in Eisner (2000) and McDonald and Pereira (2006) achieve their efficiency by assuming that children to the left of the parent and to the right of the parent are independent of each other. The algorithms of Carreras (2007) and Model 2 in Koo and Collins (2010) include grandparents for only the outermost grand-children of each parent for efficiency reasons. In a similar spirit, this paper introduces a variant of the Grand-Sib factorization that scores crossed edges independently (as a $\mathrm{CrossedEdge}$ part) and uncrossed edges under either a grandparent-sibling, grandparent, sibling, or edge-factored model depending on whether relevant edges in its local neighborhood are crossed.

A few auxiliary definitions are required. For any parent h and grandparent g, h's children are partitioned into *interior* children (those between g and h) and *exterior children* (the complementary set of children).[1] Interior children are numbered from closest to h through furthest from h; exterior children are first numbered on the side closer to h from closest to h through furthest, then the enumeration wraps around to include the vertices on the side closer to g. Figure 4 shows a parent h, its grandparent g, and a possible sequence of three interior and four exterior children. Note that for a projective tree, there would not be any children on the far side of g.

Definition 1. *Let h be m's parent.* $\mathbf{Outer}(m)$ *is the set of siblings of m that are in the same subset of h's children and are later in the enumeration than m is.*

For example, in the tree in Figure 2,

[1]Because dependency trees are directed trees, each node except for the artificial root has a unique parent. To ensure that *grandparent* is defined for the root's children, assume an artificial parent of the root for notational convenience.

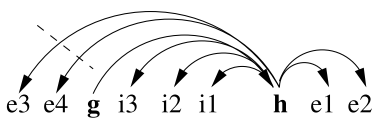


Figure 4: The exterior children are numbered first beginning on the side closest to the parent, then the side closest to the grandparent. There must be a path from the root to g, so the edges from h to its exterior children on the far side of g are guaranteed to be crossed.

	$\mathit{Crossed}(\vec{e}_{hs})$	$\neg\mathit{Crossed}(\vec{e}_{hs})$
$\neg\mathit{GProj}(\vec{e}_{hm})$	$\mathrm{Edge}(h,m)$	$\mathrm{Sib}(h,m,s)$
$\mathit{GProj}(\vec{e}_{hm})$	$\mathrm{Grand}(g,h,m)$	$\mathrm{GrandSib}(g,h,m,s)$

Table 2: Part type for an uncrossed edge $\vec{e}_{hm}$ for the crossing-sensitive third-order factorization (g is m's grandparent; s is m's inner sibling).

$\mathit{Outer}(\mathit{most}) = \{\mathit{days}, \mathit{cars}\}$.

Definition 2. *An uncrossed edge $\vec{e}_{hm}$ is GProj if both of the following hold:*

1. *The edge $\vec{e}_{gh}$ from h's parent to h is not crossed*

2. *None of the edges from h to $\mathit{Outer}(m)$ (m's outer siblings) are crossed*

Uncrossed *GProj* edges include the grandparent in the part. The part includes the sibling if the edge $\vec{e}_{hs}$ from the parent to the sibling is not crossed. Table 2 gives the factorization for uncrossed edges.

The parser in this paper finds the optimal 1-Endpoint-Crossing tree according to this factorized form. A fully projective tree would decompose into *exclusively* GrandSib parts (as all edges would be uncrossed and *GProj*). As all projective trees are within the 1-Endpoint-Crossing search space, the optimization problem that the parser solves includes all projective trees scored with grand-sibling features everywhere. Projective parsing with grand-sibling scores can be seen as a special case, as the crossing-sensitive 1-Endpoint-Crossing parser can simulate a grand-sibling projective parser by setting all $\mathrm{Crossed}(h,m)$ scores to $-\infty$.

In Figure 2, the edge from *do* to *Americans* is not *GProj* because Condition (1) is violated, while the edge from *favor* to *most* is not *GProj* because Condition (2) is violated. Under this definition, the vertices *do* and *favor* (which have children in multiple sub-problems) do not need external grandparent indices in *any* of their sub-problems. Table 3 lists the parts in the tree in Figure 2 according to this crossing-sensitive third-order factorization.

CrossedEdge(**,do*)	Sib(*cars*, *Which*, -)
CrossedEdge(*favor,cars*)	Sib(*do*, *Americans*, -)
Sib(*do*, *favor*, *Americans*)	CrossedEdge(*do,?*)
Sib(*favor*, *most*, -)	Sib(*favor*, *days*, *most*)
GSib(*favor*, *days*, *these*, -)	

Table 3: Decomposing Figure 2 according to the crossing-sensitive third-order factorization described in Section 3. Null inner siblings are indicated with -.

4 Parsing Algorithm

The parser finds the maximum scoring 1-Endpoint-Crossing tree according to the factorization in Section 3 with a dynamic programming procedure reminiscent of Koo and Collins (2010) (for scoring uncrossed edges with grandparent and/or sibling features) and of Pitler et al. (2013) (for including crossed edges). The parser also uses novel sub-problems for transitioning between portions of the tree with and without crossed edges. This formulation of the parsing problem presents two difficulties:

1. The parser must know whether an edge is crossed when it is added.

2. For *uncrossed* edges, the parser must use the appropriate part for scoring according to whether *other* edges are crossed (Table 2).

Difficulty 1 is solved by adding crossed and uncrossed edges to the tree in distinct sub-problems (Section 4.1). Difficulty 2 is solved by producing different versions of subtrees over the same sets of vertices, both with and without a grandparent index, which differ in their assumptions about the tree *outside* of that set (Section 4.2). The list of all sub-problems with their invariants and the full dynamic program are provided in the supplementary material.

4.1 Enforcing Crossing Edges

The parser adds crossed and uncrossed edges in distinct portions of the dynamic program. Uncrossed edges are added *only* through trapezoid sub-problems (that may or may not have a grandparent index), while crossed edges are added in *non*-trapezoid sub-problems. To add *all* uncrossed edges in trapezoid sub-problems, the parser (a) enforces that any edge added anywhere else must be crossed, and (b) includes transitional sub-problems to build trapezoids when the edge $\vec{e}_{hm}$ is not crossed, but the edge to its inner sibling $\vec{e}_{hs}$ is (and so the construction step shown in Figure 1b cannot be used).

4.1.1 Crossing Conditions

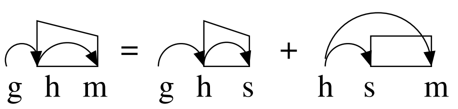
Pitler et al. (2013) included crossing edges by using "crossing region" sub-problems over intervals with an external vertex that *optionally* contained edges between the interval and the external vertex. An uncrossed edge could then be included either by a derivation that prohibited it from being crossed or a derivation which allowed (but did not force) it to be crossed. This ambiguity is removed by enforcing that (1) each crossing region contains at least one edge incident to the exterior vertex, and (2) all such edges are crossed by edges in another sub-problem. For example, by requiring at least one edge between *do* and (*favor*, *?*] and also between *favor* and (*, *do*), the edges in the two sets are guaranteed to cross.

4.1.2 Trapezoids with Edge to Inner Sibling Crossed

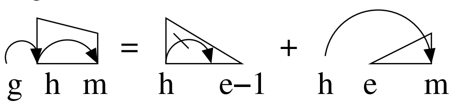
To add *all* uncrossed edges in trapezoid-style sub-problems, we must be able to construct a trapezoid over vertices $[h, m]$ whenever the edge $\vec{e}_{hm}$ is not crossed. The construction used in Koo and Collins (2010), repeated graphically in Figure 5a, *cannot* be used if the edge $\vec{e}_{hs}$ is crossed, as there would then exist edges between (h, s) and (s, m), making s an invalid split point. The parser therefore includes some "transitional glue" to allow alternative ways to construct the trapezoid over $[h, m]$ when $\vec{e}_{hm}$ is not crossed but the edge $\vec{e}_{hs}$ to m's inner sibling is.

The two additional ways of building trapezoids are shown graphically in Figures 5b and 5c. Consider the "chain of crossing edges" that includes the edge $\vec{e}_{hs}$. If none of these edges are in the subtree rooted at m, then we can build the tree involving m and its inner descendants separately (Figure 5b) from the rest of the tree rooted at h. Within the interval $[h, e-1]$ the furthest edge incident to h ($\vec{e}_{hs}$) must be crossed: these intervals are parsed choosing s and the crossing point of $\vec{e}_{hs}$ simultaneously (as in Figure 4 in Pitler et al. (2013)).

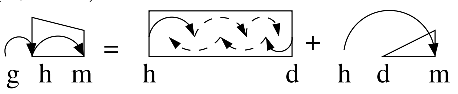
Otherwise, the sub-tree rooted at m is involved in



(a) Edge from h to inner sibling s is *not* crossed (repeats Figure 1b)



(b) $\vec{e}_{hs}$ is crossed, but the chain of crossing edges involving $\vec{e}_{hs}$ does not include any descendants of m. e is m's descendant furthest from m within (h, m). $s \in (h, e-1)$.



(c) $\vec{e}_{hs}$ is crossed, and the chain of crossing edges involving $\vec{e}_{hs}$ includes descendants of m. Of m's descendants that are incident to edges in the chain, d is the one closest to m (d can be m itself). $s \in (h, d)$.

Figure 5: Ways to build a trapezoid when the edge $\vec{e}_{hs}$ to m's inner sibling may be crossed.

the chain of crossing edges (Figure 5c). The chain of crossing edges between h and d (m's descendant, which may be m itself) is built up first then concatenated with the triangle rooted at m containing m's inner descendants not involved in the chain.

Chains of crossing edges are constructed by repeatedly applying two specialized types of L items that alternate between adding an edge from the interval to the exterior point (right-to-left) or from the exterior point to the interval (left-to-right) (Figure 6). The boundary edges of the chain can be crossed more times without violating the 1-Endpoint-Crossing property, and so the beginning and end of the chain can be unrestricted crossing regions. These specialized chain sub-problems are also used to construct *boxes* (Figure 1c) over $[s, m]$ with shared parent h when neither edge $\vec{e}_{hs}$ nor $\vec{e}_{hm}$ is crossed, but the subtrees rooted at m and at s cross each other (Figure 7).

Lemma 1. *The GrandSib-Crossing parser adds all uncrossed edges and only uncrossed edges in a tree in a "trapezoid" sub-problem.*

Proof. The *only* part is easy: when a trapezoid is built over an interval $[h, m]$, all edges are internal to the interval, so no earlier edges could cross $\vec{e}_{hm}$. Af-

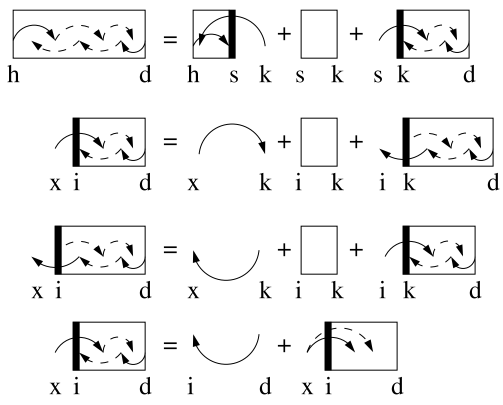


Figure 6: Constructing a chain of crossing edges

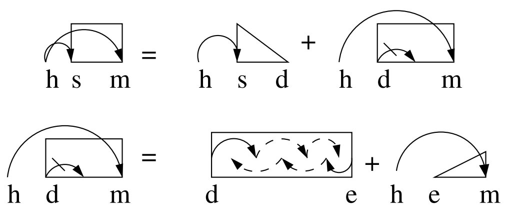


Figure 7: Constructing a box when edges in m and s's subtrees cross each other.

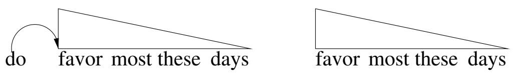
ter the trapezoid is built, only the interval endpoints h and m are accessible for the rest of the dynamic program, and so an edge between a vertex in (h, m) and a vertex $\notin [h, m]$ can never be added. The Crossing Conditions ensure that every edge added in a non-trapezoid sub-problem is crossed. $\square$

Lemma 2. *The GrandSib-Crossing parser considers* all *1-Endpoint-Crossing trees and* only *1-Endpoint-Crossing trees.*

Proof. All trees that could have been built in Pitler et al. (2013) are still possible. It can be verified that the additional sub-problems added all obey the 1-Endpoint-Crossing property. $\square$

4.2 Reduced Context in Presence of Crossings

A crossed edge (added in a non-trapezoid sub-problem) is scored as a $\mathrm{CrossedEdge}$ part. An uncrossed edge added in a trapezoid sub-problem, however, may need to be scored according to a $\mathrm{GrandSib}$, Grand, Sib, or Edge part, depending on whether the relevant *other* edges are crossed. In this section we show that sibling and grandparent features are included in the GrandSib-Crossing parser as specified by Table 2.



(a) For good contexts (b) For bad contexts

Figure 8: For each of the interval sub-problems in Koo and Collins (2010), the parser constructs versions with and without the additional grandparent index. Figure 8b is used if the edge from *do* to *favor* is crossed, or if there are any crossed edges from *favor* to children to the left of *do* or to the right of *days*. Otherwise, Figure 8a is used.

4.2.1 Sibling Features

Lemma 3. *The GrandSib-Crossing parser scores an uncrossed edge $\vec{e}_{hm}$ with a* Sib *or* $\mathrm{GrandSib}$ *part if and only if $\vec{e}_{hs}$ is not crossed.*

Proof. Whether the edge to an uncrossed edge's inner sibling is crossed is known bottom-up through how the trapezoid is constructed, since the inner sibling is *internal* to the sub-problem. When $\vec{e}_{hs}$ is not crossed, the trapezoid is constructed as in Figure 5a, using the inner sibling as the split point. When the edge $\vec{e}_{hs}$ is crossed, the trapezoid is constructed as in Figure 5b or 5c; note that both ways force the edge to the inner sibling to be crossed. $\square$

4.2.2 Grandparent Features for *GProj* Edges

Koo and Collins (2010) include an external grandparent index for each of the sub-problems that the edges within use for scoring. We want to avoid adding such an external grandparent index to *any* of the crossing region sub-problems (to stay within the desired time and space constraints) or to interval sub-problems when the external context would make all internal edges $\neg GProj$.

For each interval sub-problem, the parser constructs versions both with and without a grandparent index (Figure 8). Which version is used depends on the external context. In a *bad context*, all edges to children within an interval are guaranteed to be $\neg GProj$. This section shows that all boundary points in crossing regions are placed in bad contexts, and then that edges are scored with grandparent features if and only if they are $GProj$.

Bad Contexts for Interval Boundary Points For *exterior vertex* boundary points, all edges from it to its children will be crossed (Section 4.1.1), so it does not need a grandparent index.

Lemma 4. *If a boundary point i's parent (call it g) is within a sub-problem over vertices* $[i, j]$ *or* $[i, j] \cup \{x\}$*, then for all uncrossed edges* $\vec{e}_{im}$ *with* m *in the sub-problem, the tree outside of the sub-problem is irrelevant to whether* $\vec{e}_{im}$ *is GProj.*

Proof. The sub-problem contains the edge $\vec{e}_{gi}$, so Condition (1) is checked internally. m cannot be x, since $\vec{e}_{im}$ is uncrossed. If g is x, then $\vec{e}_{im}$ is $\neg GProj$ regardless of the outer context. If both g and $m \in (i, j]$, then $Outer(m) \subseteq (i, j]$: If m is an interior child of i ($m \in (i, g)$) then $Outer(m) \subseteq (m, g) \subseteq (i, j]$. Otherwise, if m is an exterior child ($m \in (g, j]$), by the "wrapping around" definition of $Outer$, $Outer(m) \subseteq (g, m) \subseteq (i, j]$. Thus Condition (2) is also checked internally. □

We can therefore focus on interval boundary points with their parent outside of the sub-problem.

Definition 3. *The left boundary vertex of an interval* $[i, j]$ *is in a* bad context *(*$BadContext_L(i, j)$*) if* i *receives its parent (call it g) from outside of the sub-problem and either of the following hold:*

1. **Grand-Edge Crossed**: $\vec{e}_{gi}$ *is crossed*
2. **Outer-Child-Edge Crossed**: *An edge from i to a child of i outside of* $[i, j]$ *and Outer to j will be crossed (recall this includes children on the far side of g if g is to the left of i)*

$BadContext_R(i, j)$ *is defined symmetrically regarding j and j's parent and children.*

Corollary 1. *If* $BadContext_L(i, j)$*, then for all* $\vec{e}_{im}$ *with* $m \in (i, j]$*,* $\vec{e}_{im}$ *is* $\neg GProj$*. Similarly, if* $BadContext_R(i, j)$*, for all* $\vec{e}_{jm}$ *with* $m \in [i, j)$*,* $\vec{e}_{jm}$ *is* $\neg GProj$*.*

No Grandparent Indices for Crossing Regions We would exceed the desired $O(n^4)$ run-time if *any* crossing region sub-problems needed *any* grandparent indices. In Pitler et al. (2013), LR sub-problems with edges from the exterior point crossed by both the left and the right boundary points were constructed by concatenating an L and an R sub-problem. Since the split point was not necessarily incident to a crossed edge, the split point might have $GProj$ edges to children on the side other than where it gets its parent; accommodating this would add another factor of n to the running time and space

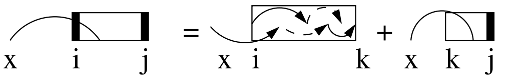
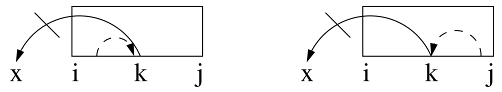


Figure 9: For all split points k, the edge from k's parent to k is crossed, so all edges from k to children on either side were $\neg GProj$. The case when the split point's parent is from the right is symmetric.



(a) x is Outer to all children of k in $(k, j]$.

(b) x is Outer to all children of k in $[i, k)$.

Figure 10: The edge $\vec{e}_{kx}$ is guaranteed to be crossed, so k is in a BadContext for whichever side it does not get its parent from.

to store the split point's parent. To avoid this increase in running time, they are instead built up as in Figure 9, which chooses the split point so that the edge from the parent of the split point to it is crossed.

Lemma 5. *For all crossing region sub-problems* $[i, j] \cup \{x\}$ *with i's parent* $\notin [i, j] \cup \{x\}$*,* $BadContext_L(i, j)$*. Similarly, when j's parent* $\notin [i, j] \cup \{x\}$*,* $BadContext_R(i, j)$*.*

Proof. Crossing region sub-problems either combine to form intervals or larger crossing regions. When they combine to form intervals as in Figure 3, it can be verified that all boundary points are in a bad context. LR sub-problems were discussed above. Split points for the $L/R/N$ sub-problems by construction are incident to a crossed edge to a further vertex. If that edge is from the split point's parent to the split point, then the grand-edge is crossed and so both sides are in a bad context. If the crossed edge is from the split point to a child, then that child is Outer to all other children on the side in which it does not get its parent (see Figure 10). □

Corollary 2. *No grandparent indices are needed for any crossing region sub-problem.*

Triangles and Trapezoids with and without Grandparent Indices The presentation that follows assumes left-headed versions. Uncrossed edges are added in two distinct types of trapezoids: (1) $\mathrm{TrapG[h, m, g, L]}$ with an external grandparent index g, scores the edge $\vec{e}_{hm}$ with grandparent features, and (2) $\text{Trap}[h, m, L]$ *without* a grandparent index, scores the edge $\vec{e}_{hm}$ without grandparent features. Triangles also have versions with ($\text{TriG}[\text{h}, \text{e}, \text{g}, \text{L}]$ and without ($\text{Tri}[\text{h}, \text{e}, \text{L}]$) a grandparent index. What follows shows that all *GProj* edges are added in TrapG sub-problems, and all $\neg$*GProj* uncrossed edges are added in Trap sub-problems.

Lemma 6. *For all $k \in (i, j)$, if $BadContext_L(i, j)$, then $BadContext_L(i, k)$. Similarly, if $BadContext_R(i, j)$, then $BadContext_R(k, j)$.*

Proof. $BadContext_L(i, j)$ implies either the edge from i's parent to i is crossed and/or an edge from i to a child of i outer to j is crossed. If the edge from i's parent to i is crossed, then $BadContext_L(i, k)$. If a child of i is outer to j, then since $k \in (i, j)$, such a child is also outer to k. □

Lemma 7. *All left-rooted triangle sub-problems* $\text{Tri}[\text{i}, \text{j}, \text{L}]$ *without a grandparent index are in a $BadContext_L(i, j)$. Similarly for all* $\text{Tri}[\text{i}, \text{j}, \text{R}]$, *$BadContext_R(i, j)$.*

Proof. All triangles without grandparent indices are either placed immediately into a bad context (by adding a crossed edge to the triangle's root from its parent, or a crossed edge from the root to an outer child) or are combined with other sub-trees to form larger crossing regions (and therefore the triangle is in a bad context, using Lemmas 5 and 6). □

Lemma 8. *All triangle sub-problems with a grandparent index* $\text{TriG}[\text{h}, \text{e}, \text{g}, \text{L}]$ *are placed in a $\neg BadContext_L(h, e)$. Similarly,* $\text{TriG}[\text{e}, \text{h}, \text{g}, \text{R}]$ *are only placed in $\neg BadContext_R(h, e)$.*

Proof. Consider where a non-empty triangle ($h \neq e$) with a grandparent index $\text{TriG}[\text{h}, \text{e}, \text{g}, \text{L}]$ can be placed in the full dynamic program and what each step would imply about the rest of the tree.

If the triangle contains exterior children of h (e and g are on opposite sides of h), then it can either combine with a trapezoid to form another larger triangle (as in Figure 1a) or it can combine with another sub-problem to form a box with a grandparent index (Figure 1c or 7). Boxes with a grandparent index can only combine with another trapezoid to form a larger trapezoid (Figure 1b). Both cases force $\vec{e}_{gh}$ to not be crossed and prevent h from having any outer crossed children, as h becomes an internal node within the larger sub-problem.

If the triangle contains interior children of h (e lies between g and h), then it can either form a trapezoid from g to h by combining with a triangle (Figure 5b) or a chain of crossing edges (Figure 5c), or it can be used to build a box with a grandparent index (Figures 1c and 7), which then can only be used to form a trapezoid from g to h. In either case, a trapezoid is constructed from g to h, enforcing that $\vec{e}_{gh}$ cannot be crossed. These steps prevent h from having any additional children between g and e (since h does not appear in the adjacent sub-problems at all whenever $h \neq e$), so again the children of h in (e, h) have no outer siblings. □

Lemma 9. *In a* $\text{TriG}[\text{h}, \text{e}, \text{g}, \text{L}]$ *sub-problem, if an edge $\vec{e}_{hm}$ is not crossed and no edges from i to siblings of m in $(m, e]$ are crossed, then $\vec{e}_{hm}$ is GProj.*

Proof. This follows from (1) the edge $\vec{e}_{hm}$ is not crossed, (2) the edge $\vec{e}_{gh}$ is not crossed by Lemma 8, and (3) no outer siblings are crossed (outer siblings in $(m, e]$ are not crossed by assumption and siblings outer to e are not crossed by Lemma 8). □

Lemma 10. *An edge $\vec{e}_{hm}$ scored with a* GrandSib *or* Grand *part (added through a* $\text{TrapG}[h, m, g, L]$ *or* $\text{TrapG}[m, h, g, R]$ *sub-problem) is GProj.*

Proof. A TrapG can either (1) combine with descendants of m to form a triangle with a grandparent index rooted at h (indicating that m is the outermost inner child of h) or (2) combine with descendants of m and of m's adjacent outer sibling (call it o), forming a trapezoid from h to o (indicating that $\vec{e}_{ho}$ is not crossed). Such a trapezoid could again only combine with further uncrossed outer siblings until eventually the final triangle rooted at h with grandparent index g is built. As $\vec{e}_{hm}$ was not crossed, no edges from h to outer siblings within the triangle are crossed, and $\vec{e}_{hm}$ is within a TriG sub-problem, $\vec{e}_{hm}$ is *GProj* by Lemma 9. □

Lemma 11. *An uncrossed edge $\vec{e}_{hm}$ scored with a* Sib *or* Edge *part (added through a* $\text{Trap}[\text{h}, \text{m}, \text{L}]$ *or* $\text{Trap}[\text{m}, \text{h}, \text{R}]$ *sub-problem) is $\neg$GProj.*

Proof. A Trap can only (1) form a triangle without a grandparent index, or (2) form a trapezoid to an outer sibling of m, until eventually a final triangle rooted at h without a grandparent index is built. This triangle without a grandparent index is then placed in a bad context (Lemma 7) and so $\vec{e}_{hm}$ is $\neg \mathit{GProj}$ (Corollary 1). $\square$

4.3 Main Results

Lemma 12. *The crossing-sensitive third-order parser runs in $O(n^4)$ time and $O(n^3)$ space when the input is an unpruned graph. When the input to the parser is a pruned graph with at most k incoming edges per node, the crossing-sensitive third-order parser runs in $O(kn^3)$ time and $O(n^3)$ space.*

Proof. All sub-problems are either over intervals (two indices), intervals with a grandparent index (three indices), or crossing regions (three indices). No crossing regions require any grandparent indices (Corollary 2). The only sub-problems that require a maximization over two internal split points are over intervals and need no grandparent indices (as the furthest edges from each root are guaranteed to be crossed within the sub-problem). All steps either contain an edge in their construction step or in the invariant of the sub-problem, so with a pruned graph as input, the running time is the number of edges ($O(kn)$) times the number of possibilities for the other two free indices ($O(n^2)$). The space is not reduced as there is not necessarily an edge relationship between the three stored vertices. $\square$

Theorem 1. *The GrandSib-Crossing parser correctly finds the maximum scoring 1-Endpoint-Crossing tree according to the crossing-sensitive third-order factorization (Section 3) in $O(n^4)$ time and $O(n^3)$ space. When the input to the parser is a pruned graph with at most k incoming edges per node, the GrandSib-Crossing parser correctly finds the maximum scoring 1-Endpoint-Crossing tree that uses only unpruned edges in $O(kn^3)$ time and $O(n^3)$ space.*

Proof. The correctness of scoring follows from Lemmas 3, 10, and 11. The search space of 1-Endpoint-Crossing trees was in Lemma 2 and the time and space complexity in Lemma 12. $\square$

The parser produces the optimal tree in a well-defined output space. Pruning edges restricts the output space the same way that constraints enforcing projectivity or the 1-Endpoint-Crossing property also restrict the output space. Note that if the optimal unconstrained 1-Endpoint-Crossing tree does not include any pruned edges, then whether the parser uses pruning or not is irrelevant; both the pruned and unpruned parsers will produce the exact same tree.

5 Experiments

The crossing-sensitive third-order parser was implemented as an alternative parsing algorithm within *dpo3* (Koo and Collins, 2010).[2] To ensure a fair comparison, all code relating to input/output, features, learning, etc. was re-used from the original projective implementation, and so the only substantive differences between the projective and 1-Endpoint-Crossing parsers are the dynamic programming charts, the parsing algorithms, and the routines that extract the maximum scoring tree from the completed chart.

The treebanks used to prepare the CoNLL shared task data (Buchholz and Marsi, 2006; Nivre et al., 2007) vary widely in their conventions for representing conjunctions, modal verbs, determiners, and other decisions (Zeman et al., 2012). The experiments use the newly released HamleDT software (Zeman et al., 2012) that normalizes these treebanks into one standard format and also provides built-in transformations to other conjunction styles. The unnormalized treebanks input to HamleDT were from the CoNLL 2006 Shared Task (Buchholz and Marsi, 2006) for Danish, Dutch, Portuguese, and Swedish and from the CoNLL 2007 Shared Task (Nivre et al., 2007) for Czech.

The experiments include the default Prague style (Böhmová et al., 2001), Mel'čukian style (Mel'čuk, 1988), and Stanford style (De Marneffe and Manning, 2008) for conjunctions. Under the grandparent-sibling factorization, the two words being conjoined would never appear in the same scope for the Prague style (as they are siblings on different sides of the conjunct head). In the Mel'čukian style, the two conjuncts are in a grandparent relationship and in the Stanford style the two conjuncts

[2] `http://groups.csail.mit.edu/nlp/dpo3/`

are in a sibling relationship, and so we would expect to see larger gains for including grandparents and siblings under the latter two representations. The experiments also include a nearly projective dataset, the English Penn Treebank (Marcus et al., 1993), converted to dependencies with PennConverter (Johansson and Nugues, 2007).

The experiments use marginal-based pruning based on an edge-factored directed spanning tree model (McDonald et al., 2005). Each word's set of potential parents is limited to those with a marginal probability of at least .1 times the probability of the most probable parent, and cut off this list at a maximum of 20 potential parents per word. To ensure that there is always at least one projective and/or 1-Endpoint-Crossing tree achievable, the artificial root is always included as an option. The pruning parameters were chosen to keep 99.9% of the true edges on the English development set.

Following Carreras (2007) and Koo and Collins (2010), before training the training set trees are transformed to be the best achievable within the model class (i.e., the closest projective tree or 1-Endpoint-Crossing tree). All models are trained for five iterations of averaged structured perceptron training. For English, the model after the iteration that performs best on the development set is used; for all other languages, the model after the fifth iteration is used.

Model	Du	Cz	Pt	Da	Sw
	Prague				
Proj GSib	80.45	85.12	88.85	**88.17**	85.50
Proj Edge	80.38	84.04	88.14	**88.29**	**86.09**
1-EC CS-GSib	**82.78**	**85.90**	**89.74**	**88.64**	85.70
1-EC Edge	**83.33**	84.97	**89.21**	**88.19**	**86.46**
	Mel'čukian				
Proj GSib	82.26	**87.96**	89.19	**90.23**	**89.59**
Proj Edge	82.09	86.18	88.73	89.29	**89.00**
1-EC CS-GSib	**86.03**	**87.89**	**90.34**	**90.50**	**89.34**
1-EC Edge	85.28	**87.57**	**89.96**	**90.14**	88.97
	Stanford				
Proj GSib	81.16	86.83	88.80	**88.84**	87.27
Proj Edge	80.56	86.18	88.61	**88.69**	**87.92**
1-EC CS-GSib	**84.67**	**88.34**	**90.20**	**89.22**	**88.15**
1-EC Edge	83.62	87.13	89.43	**88.74**	87.36

Table 4: Overall Unlabeled Attachment Scores (UAS) for all words.[3] CS-GSib is the proposed crossing-sensitive grandparent-sibling factorization. For each data set, we bold the most accurate model and those not significantly different from the most accurate (sign test, $p < .05$). Languages are sorted in increasing order of projectivity.

Model	UAS
Proj GSib	**93.10**
Proj Edge	92.63
1-EC CS-GSib	**93.22**
1-EC Edge	92.80

Table 5: English results

5.1 Results

Results for edge-factored and (crossing-sensitive) grandparent-sibling factored models for both projective and 1-Endpoint-Crossing parsing are in Tables 4 and 5. In 14 out of the 16 experimental set-ups, the third-order 1-Endpoint-Crossing parser is more accurate than the third-order projective parser. It is significantly better than the projective parser in 9 of the set-ups and significantly worse in none.

Table 6 shows how often the 1-EC CS-GSib parser used each of the GrandSib, Grand, Sib, Edge, and CrossedEdge parts for the Mel'čukian and Stanford style test sets. In both representations, the parser is able to score with a sibling context more often than it is able to score with a grandparent, perhaps explaining why the datasets using the Stanford conjunction representation saw the largest gains from including the higher order factors into the 1-Endpoint-Crossing parser.

Across languages, the third-order 1-Endpoint-Crossing parser runs 2.1-2.7 times slower than the third-order projective parser (71-104 words per second, compared with 183-268 words per second). Parsing speed is correlated with the amount of pruning. The level of pruning mentioned earlier is relatively permissive, retaining 39.0-60.7% of the edges in the complete graph; a higher level of pruning could likely achieve much faster parsing times with the same underlying parsing algorithms.

[3]Following prior work in graph-based dependency parsing (for example, Rush and Petrov (2012)), English results use automatically produced part-of-speech tags and results exclude punctuation, while the results for all other languages use gold part-of-speech tags and include punctuation.

Part Used	Du	Cz	Pt	Da	Sw
	Mel'čukian				
CrossedEdge	8.5	4.5	3.2	1.4	1.2
GrandSib	81.2	89.1	90.7	95.7	96.2
Grand	1.1	0.5	0.8	0.3	0.2
Sib	9.0	5.8	5.2	2.6	2.3
Edge	< 0.1	< 0.1	0	< 0.1	0
	Stanford				
CrossedEdge	8.4	5.1	3.3	2.0	1.8
GrandSib	81.4	87.8	90.5	94.2	95.2
Grand	1.1	0.5	0.7	0.3	0.3
Sib	8.9	6.5	5.2	3.5	2.6
Edge	< 0.1	0.1	0	< 0.1	0

Table 6: The proportion of edges in the predicted output trees from the CS-GSib 1-Endpoint-Crossing parser that would have used each of the five part types for scoring.

6 Discussion

There have been many other notable approaches to non-projective parsing with larger scopes than single edges, including transition-based parsers, directed spanning tree graph-based parsers, and mildly non-projective graph-based parsers.

Transition-based parsers score actions that the parser may take to transition between different configurations. These parsers typically use either greedy or beam search, and can condition on any tree context that is in the history of the parser's actions so far. Zhang and Nivre (2011) significantly improved the accuracy of an arc-eager transition system (Nivre, 2003) by adding several additional classes of features, including some third-order features. Basic arc-eager and arc-standard (Nivre, 2004) models that parse left-to-right using a stack produce projective trees, but transition-based parsers can be modified to produce crossing edges. Such modifications include pseudo-projective parsing in which the dependency labels encode transformations to be applied to the tree (Nivre and Nilsson, 2005), adding actions that add edges to words in the stack that are not the topmost item (Attardi, 2006), adding actions that swap the positions of words (Nivre, 2009), and adding a second stack (Gómez-Rodríguez and Nivre, 2010).

Graph-based approaches to non-projective parsing either consider all directed spanning trees or restricted classes of mildly non-projective trees. Directed spanning tree approaches with higher order features either use approximate learning techniques, such as loopy belief propagation (Smith and Eisner, 2008), or use dual decomposition to solve relaxations of the problem (Koo et al., 2010; Martins et al., 2013). While not guaranteed to produce optimal trees within a fixed number of iterations, these dual decomposition techniques do give certificates of optimality on the instances in which the relaxation is tight and the algorithm converges quickly.

This paper described a mildly non-projective graph-based parser. Other parsers in this class find the optimal tree in the class of well-nested, block degree two trees (Gómez-Rodríguez et al., 2011), or in a class of trees further restricted based on gap inheritance (Pitler et al., 2012) or the head-split property (Satta and Kuhlmann, 2013), with edge-factored running times of $O(n^5) - O(n^7)$. The factorization used in this paper is not immediately compatible with these parsers: the complex cases in these parsers are due to *gaps*, not *crossings*. However, there may be analogous "gap-sensitive" factorizations that could allow these parsers to be extended without large increases in running times.

7 Conclusion

This paper proposed an exact, graph-based algorithm for non-projective parsing with higher order features. The resulting parser has the same asymptotic run time as a third-order projective parser, and is significantly more accurate for many experimental settings. An exploration of other factorizations that facilitate non-projective parsing (for example, an analogous "gap-sensitive" variant) may be an interesting avenue for future work. Recent work has investigated faster variants for third-order graph-based projective parsing (Rush and Petrov, 2012; Zhang and McDonald, 2012) using structured prediction cascades (Weiss and Taskar, 2010) and cube pruning (Chiang, 2007). It would be interesting to extend these lines of work to the crossing-sensitive third-order parser as well.

Acknowledgments

I would like to thank Sampath Kannan, Mitch Marcus, Chris Callison-Burch, Michael Collins, Mark Liberman, Ben Taskar, Joakim Nivre, and the three anonymous reviewers for valuable comments on earlier versions of this material.

References

G. Attardi. 2006. Experiments with a multilanguage non-projective dependency parser. In *Proceedings of CoNLL*, pages 166–170.

A. Böhmová, J. Hajič, E. Hajičová, and B. Hladká. 2001. The Prague Dependency Treebank: Three-level annotation scenario. In Anne Abeillé, editor, *Treebanks: Building and Using Syntactically Annotated Corpora*, pages 103–127. Kluwer Academic Publishers.

B. Bohnet and J. Kuhn. 2012. The best of both worlds – a graph-based completion model for transition-based parsers. In *Proceedings of EACL*, pages 77–87.

S. Buchholz and E. Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of CoNLL*, pages 149–164.

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

E. Charniak and M. Johnson. 2005. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of ACL*, pages 173–180.

D. Chiang. 2007. Hierarchical phrase-based translation. *Computational Linguistics*, 33(2):201–228.

M. De Marneffe and C. Manning. 2008. Stanford typed dependencies manual.

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

C. Gómez-Rodríguez and J. Nivre. 2010. A transition-based parser for 2-planar dependency structures. In *Proceedings of ACL*, pages 1492–1501.

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

K. Hall. 2007. K-best spanning tree parsing. In *Proceedings of ACL*, pages 392–399.

R. Johansson and P. Nugues. 2007. Extended constituent-to-dependency conversion for English. In *Proceedings of the 16th Nordic Conference on Computational Linguistics (NODALIDA)*, pages 105–112.

T. Koo and M. Collins. 2010. Efficient third-order dependency parsers. In *Proceedings of ACL*, pages 1–11.

T. Koo, A. M. Rush, M. Collins, T. Jaakkola, and D. Sontag. 2010. Dual decomposition for parsing with non-projective head automata. In *Proceedings of EMNLP*, pages 1288–1298.

T. Koo. 2010. *Advances in discriminative dependency parsing*. Ph.D. thesis, Massachusetts Institute of Technology.

S. Kübler, R. McDonald, and J. Nivre. 2009. Dependency parsing. *Synthesis Lectures on Human Language Technologies*, 2(1):1–127.

M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

A. F. T. Martins, N. A. Smith, and E. P. Xing. 2009. Concise integer linear programming formulations for dependency parsing. In *Proceedings of ACL*, pages 342–350.

A. Martins, M. Almeida, and N. A. Smith. 2013. Turning on the turbo: Fast third-order non-projective turbo parsers. In *Proceedings of ACL (Short Papers)*, pages 617–622.

R. McDonald and F. Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *Proceedings of EACL*, pages 81–88.

R. McDonald and G. Satta. 2007. On the complexity of non-projective data-driven dependency parsing. In *Proceedings of the 10th International Conference on Parsing Technologies*, pages 121–132.

R. McDonald, F. Pereira, K. Ribarov, and J. Hajič. 2005. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of HLT/EMNLP*, pages 523–530.

I. Mel'čuk. 1988. *Dependency Syntax: Theory and Practice*. State University of New York Press.

J. Nivre and J. Nilsson. 2005. Pseudo-projective dependency parsing. In *Proceedings of ACL*, pages 99–106.

J. Nivre, J. Hall, S. Kübler, R. McDonald, J. Nilsson, S. Riedel, and D. Yuret. 2007. The CoNLL 2007 shared task on dependency parsing. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL*, pages 915–932.

J. Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies*, pages 149–160.

J. Nivre. 2004. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57.

J. Nivre. 2009. Non-projective dependency parsing in expected linear time. In *Proceedings of ACL*, pages 351–359.

E. Pitler, S. Kannan, and M. Marcus. 2012. Dynamic programming for higher order parsing of gap-minding trees. In *Proceedings of EMNLP*, pages 478–488.

E. Pitler, S. Kannan, and M. Marcus. 2013. Finding optimal 1-Endpoint-Crossing trees. *Transactions of the Association for Computational Linguistics*, 1(Mar):13–24.

A. Rush and S. Petrov. 2012. Vine pruning for efficient multi-pass dependency parsing. In *Proceedings of NAACL*, pages 498–507.

G. Satta and M. Kuhlmann. 2013. Efficient parsing for head-split dependency trees. *Transactions of the Association for Computational Linguistics*, 1(July):267–278.

D. A. Smith and J. Eisner. 2008. Dependency parsing by belief propagation. In *Proceedings of EMNLP*, pages 145–156.

D. Weiss and B. Taskar. 2010. Structured Prediction Cascades. In *AISTATS*, pages 916–923.

D. Zeman, D. Mareček, M. Popel, L. Ramasamy, J. Štěpánek, Z. Žabokrtský, and J. Hajič. 2012. HamleDT: To parse or not to parse? In *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, pages 2735–2741.

H. Zhang and R. McDonald. 2012. Generalized higher-order dependency parsing with cube pruning. In *Proceedings of EMNLP*, pages 320–331.

Y. Zhang and J. Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proceedings of ACL (Short Papers)*, pages 188–193.