# A Linear-Time Transition System for Crossing Interval Trees

**Emily Pitler** **Ryan McDonald**
Google, Inc.
{epitler,ryanmcd}@google.com

## Abstract

We define a restricted class of non-projective trees that 1) covers many natural language sentences; and 2) can be parsed exactly with a generalization of the popular arc-eager system for projective trees (Nivre, 2003). Crucially, this generalization only adds constant overhead in run-time and space keeping the parser's total run-time linear in the worst case. In empirical experiments, our proposed transition-based parser is more accurate on average than both the arc-eager system or the swap-based system, an unconstrained non-projective transition system with a worst-case quadratic runtime (Nivre, 2009).

## 1 Introduction

Linear-time transition-based parsers that use either greedy inference or beam search are widely used today due to their speed and accuracy (Nivre, 2008; Zhang and Clark, 2008; Zhang and Nivre, 2011). Of the many proposed transition systems (Nivre, 2008), the arc-eager transition system of Nivre (2003) is one of the most popular for a variety of reasons. The arc-eager system has a well-defined output space: it can produce *all* projective trees and *only* projective trees. For an input sentence with $n$ words, the arc-eager system always performs $2n$ operations and each operation takes constant time. Another attractive property of the arc-eager system is the close connection between the parameterization of the parsing problem and the final predicted output structure. In the arc-eager model, each operation has a clear interpretation in terms of constraints on the final output tree (Goldberg and Nivre, 2012), which allows for more robust learning procedures (Goldberg and Nivre, 2012).

The arc-eager system, however, cannot produce trees with crossing arcs. Alternative systems can produce crossing dependencies, but at the cost of taking $O(n^2)$ transitions in the worst case (Nivre, 2008; Nivre, 2009; Choi and McCallum, 2013), requiring more transitions than arc-eager to produce projective trees (Nivre, 2008; Gómez-Rodríguez and Nivre, 2010), or producing trees in an unknown output class[1] (Attardi, 2006).

Graph-based non-projective parsing algorithms, on the other hand, have been able to preserve many of the attractive properties of their corresponding projective parsing algorithms by restricting search to classes of *mildly non-projective trees* (Kuhlmann and Nivre, 2006). Mildly non-projective classes of trees are *characterizable* subsets of directed trees. Classes of particular interest are those that both have high empirical coverage and that can be parsed efficiently. With appropriate definitions of feature functions and output spaces, exact higher-order graph-based non-projective parsers can match the asymptotic time and space of higher-order projective parsers (Pitler, 2014).

In this paper, we propose a class of mildly non-projective trees (§3) and a transition system (§4) that is sound and complete with respect to this class (§5) while preserving desirable properties of arc-eager: it runs in $O(n)$ time in the worst case (§6), and each operation can be interpreted as a prediction about

[1] A characterization independent of the transition system is unknown.



*Human Language Technologies: The 2015 Annual Conference of the North American Chapter of the ACL*, pages 662–671,
Denver, Colorado, May 31 – June 5, 2015. 

the final tree structure. At the same time, it can produce trees with crossing dependencies. Across ten languages, on average 96.7% of sentences have dependency trees in the proposed class (Table 1), compared with 79.4% for projective trees. The implemented mildly non-projective transition-based parser is more accurate than a fully projective parser (arc-eager, (Nivre, 2003)) and a fully non-projective parser (swap-based, (Nivre, 2009)) (§7.1).

## 2 Preliminaries

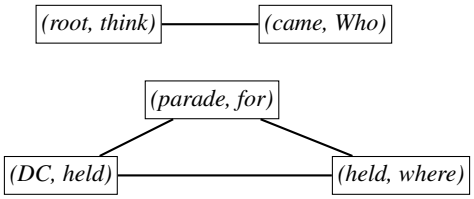
Given an input sentence $w_1 w_2 \dots w_n$, a dependency tree for that sentence is a set of vertices $V = \{0, 1, \dots, n\}$ and arcs $A \subset V \times V$. Each vertex $i$ corresponds to a word in the sentence and vertex 0 corresponds to an artificial *root* word, which is standard in the literature. An arc $(i, j) \in A$ represents a dependency between a modifier $w_j$ and a head $w_i$. Critically, the arc set $A$ is constrained to form a valid *dependency tree*: its root is at the leftmost vertex 0; each vertex $i$ has exactly one incoming arc (except 0, which has no incoming arcs); and there are no cycles. A common extension is to add labels of syntactic relations to each arc. For ease of exposition, we will focus on the unlabeled variant during the discussion but use a labeled variant during experiments.

A dependency tree is *projective* if and only if the nodes in the yield of each subtree form a contiguous interval with respect to the words and their order in the sentence. For instance, the tree in Figure 1a is *non-projective* since the subtrees rooted at *came* and *parade* do not cover a contiguous set of words. Equivalently, a dependency tree is non-projective if and only if the tree cannot be drawn in the plane above the sentence without crossing arcs. As we will see, these *crossing arcs* are a useful measure when defining sub-classes of non-projectivity. We will often reason about the set of vertices *incident* to a particular arc. The incident vertices of an arc are its endpoints: for an arc $(u, v)$, $u$ and $v$ are the two vertices incident to it.

## 3 $k$-Crossing Interval Trees

We begin by defining a class of trees based on restrictions on crossing dependencies. The class definition is independent of any transition system; it is easy to check whether a particular tree is within the


(a) A dependency tree with two disjoint sets (blue and dashed/red and dotted) of crossing arcs (bold).




(b) The auxiliary graph for the sentence above. There are two connected components of crossed arcs, one of which corresponds to the crossing interval *[root, came]* and the other *[DC, for]*.

Figure 1: A sentence with two crossing intervals.

class or not. We compare the coverage of this class on various natural language datasets with the coverage of the class of projective trees.

**Definition 1.** *Let $A$ be a set of unlabeled arcs. The* Interval *of $A$,* Interval(A)*, is the interval from the leftmost vertex in $A$ to the rightmost vertex in $A$, i.e., Interval(A) $= [min(V_A), max(V_A)]$, where $V_A = \{v : \exists u[(u, v) \in A \vee (v, u) \in A]\}$.*

**Definition 2.** *For any dependency tree $T$, the below procedure partitions the crossed arcs in $T$ into disjoint sets $A_1$, $A_2$, ...., $A_l$ such that Interval($A_1$), Interval($A_2$), ..., Interval($A_l$) are all vertex-disjoint. These intervals are the **crossing intervals** of the tree $T$.*

*Procedure:* Construct an auxiliary graph with a vertex for each crossed arc in the original tree. Two such vertices are connected by an arc if the intervals defined by the arcs they correspond to have a non-empty intersection. Figure 1b shows the auxiliary graph for the sentence in Figure 1a. The connected components of this graph form a partition of the graph's vertices, and so also partition the crossed arcs in the original sentence. The intervals defined by these groups cannot overlap, since then the crossed arcs that span the overlapping portion would have been connected by an arc in the auxiliary graph and hence been part of the same connected component. □

**Definition 3.** *A tree is a* $k$-Crossing Interval *tree if for each crossing interval, there exists at most $k$ ver-*

| Language | 2-Crossing Interval | 1-Endpoint-Crossing | Projective |
|---|---|---|---|
| Basque | 93.5 | 94.7 | 74.8 |
| Czech | 97.4 | 98.9 | 77.9 |
| Dutch | 91.4 | 95.8 | 63.6 |
| English | 99.2 | 99.3 | 93.4 |
| German | 94.7 | 96.4 | 72.3 |
| Greek | 99.1 | 99.7 | 84.4 |
| Hungarian | 95.3 | 96.3 | 74.7 |
| Portuguese | 99.0 | 99.6 | 83.3 |
| Slovene | 98.2 | 99.5 | 79.6 |
| Turkish | 99.1 | 99.3 | 89.9 |
| Average | 96.7 | 98.0 | 79.4 |

Table 1: Proportion of trees (excluding punctuation) in each tree class for the CoNLL shared tasks training sets: Dutch, German, Portuguese, and Slovene are from Buchholz and Marsi (2006); Basque, Czech, English, Greek, Hungarian, and Turkish data are from Nivre et al. (2007).

*tices such that a) all crossed arcs within the interval are incident to at least one of these vertices and b) any vertex in the interval that has a child on the far side of its parent is one of these $k$ vertices.*

Figure 1a shows a 2-Crossing Interval tree. For the first crossing interval, *think* and *came* satisfy the conditions; for the second, *parade* and *held* do. The coverage of 2-Crossing Interval trees is shown in Table 1. Across datasets from ten languages with a non-negligible proportion of crossing dependencies, on average 96.7% of dependency trees are 2-Crossing Interval, within 1.3% of the larger 1-Endpoint-Crossing class (Pitler et al., 2013) and substantially larger than the 79.4% coverage of projective trees. Coverage increases as $k$ increases; for 3-Crossing Interval trees, the average coverage reaches 98.6%. Punctuation tokens are excluded when computing coverage to better reflect language specific properties rather than treebank artifacts; for example, the Turkish CoNLL data attaches punctuation tokens to the artificial root, causing a 15% absolute drop in coverage for projective trees when punctuation tokens are included (89.9% vs. 74.7%).

### 3.1 Connections to Other Tree Classes

$k = 0$ or $k = 1$ gives exactly the class of projective trees (even a single crossing implies two vertex-disjoint crossed edges). 2-Crossing Interval trees are a subset of the linguistically motivated 1-Endpoint-Crossing trees (Pitler et al., 2013) (each crossed edge is incident to one of the two vertices for the interval, so all edges that cross it are incident to the *other* vertex for the interval); all of the examples from the linguistics literature provided in Pitler (2013, p.132-136) for 1-Endpoint-Crossing trees are 2-Crossing Interval trees as well. 2-Crossing Interval trees are *not* necessarily *well-nested* and can have unbounded *block degree* (Kuhlmann, 2013). Figure 2 shows an example of a 2-Crossing Interval tree (all crossed edges are incident to either $a$ or $b$; no children are on the far side of their parent) in which the subtrees rooted at $a$ and $b$ are ill-nested and each has a block degree of $n + 1$.

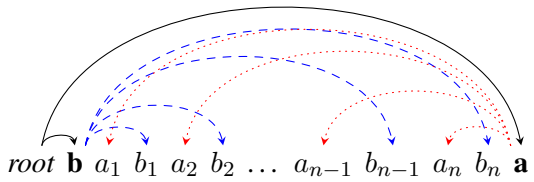



Figure 2: A 2-Crossing Interval tree that is not well-nested and has unbounded block degree.

## 4 Two-Registers Transition System

A transition system for dependency parsing comprises: 1) an initial configuration for an input sentence; 2) a set of final configurations after which the parsing derivation terminates; and 3) a set of deterministic *transitions* for transitioning from one configuration to another (Nivre, 2008).

Our transition system builds on one of the most commonly used transition systems for parsing projective trees, the arc-eager system (Nivre, 2003). An arc-eager configuration, $c$, is a tuple, $(\sigma, \beta, A)$, where 1) $\sigma$ is a *stack* consisting of a subset of processed tokens; 2) $\beta$ is a *buffer* consisting of unprocessed tokens; and 3) $A$ is the set of dependency arcs already added to the tree.

We define a new transition system called *two-registers*. Configurations are updated to include two *registers* $R1$ and $R2$, i.e., $c = (\sigma, \beta, R1, R2, A)$. A register contains one vertex or is empty: $R1, R2 \in V \cup \{null\}$. Table 2 defines both the arc-eager and two-registers transition systems. The two-registers system includes the arc-eager transitions (top half of Table 2) and three new transitions that make use of the registers (bottom half of Table 2):

- **Store**: Moves the token at the front of the buffer into the first available register, optionally

Arc-Eager
- Initial configuration: $(\{0\}, \{1, \ldots, n\}, \{\})$
- Terminal configurations $(\sigma, \{\}, A)$

Two-Registers
- Initial configuration: $(\{\}, \{0, \ldots, n\}, null, null, \{\})$
- Terminal configurations: $(\sigma, \{\}, null, null, A)$

| | Transition | $\sigma$ | $\beta$ | $R1$ | $R2$ | $A$ |
|---|---|---|---|---|---|---|
| Arc-Eager | Left-Arc | $\sigma_{m..2}$ | $\beta_{1..n}$ | $R1$ | $R2$ | $A \cup \{(\beta_1, \sigma_1)\}$ |
| | Right-Arc | $\sigma_{m..1}\|\beta_1$ | $\beta_{2..n}$ | $R1$ | $R2$ | $A \cup \{(\sigma_1, \beta_1)\}$ |
| | Shift | $\sigma_{m..1}\|\beta_1$ | $\beta_{2..n}$ | $R1$ | $R2$ | $A$ |
| | Reduce | $\sigma_{m..2}$ | $\beta_{1..n}$ | $R1$ | $R2$ | $A$ |
| + Two-Registers | Store(arc) | $\sigma_{m..1}$ | $\beta_{2..n}$ | $R1'$ | $R2'$ | $A \cup B$ |
| | | Where: | arc $\in$ {left, right, no-arc} | | | |
| | | | $B := \{(\beta_1, R1)\}$ if arc=left, $\{(R1, \beta_1)\}$ if arc=right, and $\emptyset$ otherwise. | | | |
| | | | $R1' := (R1 = null)\ ?\ \beta_1\ :\ R1$; $R2' := (R1 = null)\ ?\ R2\ :\ \beta_1$. | | | |
| | Clear | $\sigma_{m..2}\|\psi$ | $\gamma\|\beta_{1..n}$ | *null* | *null* | $A$ |
| | | Where: | $\gamma := (\sigma_1 = \beta_1 - 1)\ ?\ \sigma_1\ :\ (R2 = \beta_1 - 1)\ ?\ R2\ :\ null$ | | | |
| | | | $\psi := \{\sigma_1\} \cup NotCovered(R1) \cup NotCovered(R2) - \{\gamma\}$ in left-to-right order, | | | |
| | | | where *NotCovered*$(x) := x$ if no edges in $A$ cover $x$ and $\emptyset$ otherwise. | | | |
| | Register-Stack(k, dir) | $\sigma_{m..2}\|\psi$ | $\beta_{1..n}$ | $R1$ | $R2$ | $A \cup B$ |
| | | Where: | $k \in \{1, 2\}$ and $dir \in$ {to-register, to-stack} | | | |
| | | | $B := (dir =$ to-register$)\ ?\ \{(\sigma_1, Rk)\}\ :\ \{(Rk, \sigma_1)\}$ | | | |
| | | | $\psi := (dir =$ to-stack $\wedge\ \sigma_1 < Rk)\ ?\ null\ :\ \sigma_1$ | | | |

Table 2: Transitions and the resulting state after each is applied to the configuration $(\sigma_{m..2}|\sigma_1, \beta_1|\beta_{2..n}, R1, R2, A)$.

| Transition | $\sigma$ | $\beta$ | $R1$ | $R2$ | $A$ |
|---|---|---|---|---|---|
| ... | [that we Hans house] | [helped paint] | *null* | *null* | {(*house*, *the*)} |
| Store(no-arc) | [that we Hans house] | [paint] | helped | *null* | |
| Store(right) | [that we Hans house] | [] | helped | paint | $\cup$ {(*helped, paint*)} |
| Register-Stack(2, to-stack) | [that we Hans] | [] | helped | paint | $\cup$ {(*paint, house*)} |
| Register-Stack(1, to-stack) | [that we] | [] | helped | paint | $\cup$ {(*helped, Hans*)} |
| Register-Stack(1, to-stack) | [that] | [] | helped | paint | $\cup$ {(*helped, we*)} |
| Register-Stack(1, to-register) | [that] | [] | helped | paint | $\cup$ {(*that, helped*)} |
| Clear | [that] | [paint] | *null* | *null* | |

Table 3: An excerpt from a gold standard derivation of the sentence in Figure 3. The two words *paint* and *house* are added to the registers and then crossed arcs are added between them and the top of the stack.

| Transition | Precondition | Type |
|---|---|---|
| Left-Arc, Right-Arc | $R1 \notin (\sigma_1, \beta_1) \wedge R2 \notin (\sigma_1, \beta_1)$ | (2) |
| Store($\cdot$) | $(R1 = null \vee R2 =$ *null*$) \wedge (\beta_1 > last)$ | (1) |
| Clear | $(R1 \neq null) \wedge (R2 \neq null \vee \beta_1 = null) \wedge (\sigma_2 < R1) \wedge (\sigma_1 \notin (R1, R2))$ | (1) |
| Register-Stack(k, $\cdot$) | $(\sigma_1 > last) \vee (k = 1 \wedge \neg IsCovered(R1))$ | (1) |
| | $\sigma_2 < R_{right}$ | (2) |
| Register-Stack(k, to-register) | $(R_{close}, \sigma_1) \notin A$ | (3) |
| Register-Stack(k, to-stack) | $(\sigma_1, R_{far}) \notin A$ | (3) |

Table 4: Preconditions that ensure the 2-Crossing Interval property for trees output by the two-registers transition system, applied to a configuration $(\sigma_{m..1}, \beta_{1..n}, R1, R2, A)$. If $\sigma_1 < R1$, $R_{close} := R1$ and $R_{far} := R2$; otherwise, $R_{close} := R2$ and $R_{far} := R1$. $R_{right} := (R2 = null)\ ?\ R1\ :\ R2$. Preconditions of type (1) ensure each pair of registers defines a disjoint crossing interval; type (2) that only edges incident to registers are crossed; and type (3) that only registers can have children on the far side of their parent.

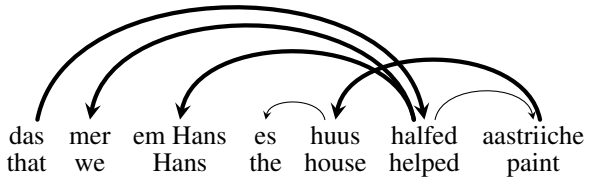



Figure 3: A clause with crossing edges (Shieber, 1985).

adding an arc between this token and the token in the first register.

- **Clear**: Removes tokens from the registers, reducing them completely if they are covered by an edge in $A$ or otherwise placing them back on the stack in order. If either $R2$ or the top of the stack is the token immediately to the left of the front of the buffer, that token is placed back on the buffer instead.
- **Register-Stack**: Adds an arc between the top of the stack and one of the registers.

A derivation excerpt for the clause in Figure 3 is shown in Table 3. The two tokens incident to all crossed arcs *helped* and *paint* are stored in the registers. The crossed arcs are then added through Register-Stack transitions, working outward from the registers through the previous words in the sentence: *(paint, house)*, then *(helped, Hans)*, etc. After all the crossed arcs incident to these two tokens have been added, the registers are cleared.

Preconditions related to rootedness, single-headedness, and acyclicity follow the arc-eager system straightforwardly: each transition that adds an arc $(h, m)$ checks that $m$ is not the root, $m$ does not already have a head, and that $h$ is not a descendant of $m$. Preconditions used to guarantee that trees output by the system are within the desired class are listed in Table 4. In particular, they ensure that all crossed arcs are incident to registers, and that each pair of registers entails an interval corresponding to a self-contained set of crossed edges. To avoid traversing $A$ while checking preconditions, two helper constants are used: *IsCovered(Rk)*[2] and *last*[3].

[2] $IsCovered(R1)$ is true if there exists an arc in $A$ with endpoints on either side of $R1$. Rather than enumerating arcs, this boolean can be updated in constant time by setting it to true only after a Register-Stack(2, dir) transition with $\sigma_1 < R1$; likewise R2 can only be covered with a Register-Stack(1, dir) transition with $\sigma_1 > R2$.

[3] *last* is used to indicate the rightmost partially processed unreduced vertex after the last pair of registers were cleared (set to the rightmost in $\gamma$, $\psi$ after each Clear transition).

**Lemma 1.** *In the two-registers system, all crossed arcs are added through register-stack operations.*

*Proof.* Suppose for the sake of contradiction that a right arc $(s, b)$ added when $\sigma_1 = s$ and $\beta_1 = b$ is crossed in the final output tree (the argument for left-arcs is identical). Let $(l, r)$ with $l < r$ be an arc that crosses $(s, b)$. One of $\{l, r\}$ must be within the open interval $(s, b)$ and one of $\{l, r\} \notin [s, b]$. When the arc $(s, b)$ is added, no tokens in the open interval $(s, b)$ remain. They cannot be in the stack or buffer since the stack and buffer always remain in order; they cannot be in registers by the precondition $R1 \notin (\sigma_1, \beta_1) \wedge R2 \notin (\sigma_1, \beta_1)$ for Right-Arc transitions. Thus, $(l, r)$ must already have been added. It cannot be that $l \in (s, b)$ and $r > b$, since the rest of the buffer has never been accessible to tokens left of $b$. The ordering must then be $l < s < r < b$. Figure 4 shows that for each way $(l, r)$ could have been added (Right-Arc, 4a; Store(right), 4b; Register-Stack(k, to-stack), 4c; Register-Stack(k, to-register), 4d), it is impossible to keep $s$ unreduced without violating one of the preconditions.

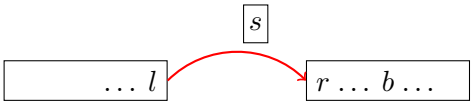
The only other type of arc-adding operation is Store. Similar logic holds: arcs added through Left-Arc and Right-Arc transitions cannot cross these arcs, since they would violate the preconditions $R1 \notin (\sigma_1, \beta_1) \wedge R2 \notin (\sigma_1, \beta_1)$; later arcs involving other registers would imply Clear operations that violate $\sigma_2 < R1 \wedge \sigma_1 \notin (R1, R2)$. □

## 5 Parsing 2-Crossing Interval Trees with the Two-Registers Transition System

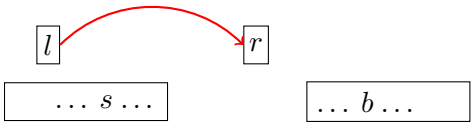
In this section we show the correspondence between the two-registers transition system and 2-Crossing Interval trees: each forest output by the transition system is a 2-Crossing Interval tree (*soundness*) and every 2-Crossing Interval tree can be produced by the two-registers system (*completeness*).

### 5.1 Soundness: Two-Registers System → 2-Crossing Interval trees

*Proof.* Every crossed arc is incident to a token that was in a register (Lemma 1). There cannot be any overlap between register arcs where the corresponding tokens were not in the registers simultaneously: the Clear transition updates the book-keeping constant $last$ to be the rightmost vertex associated with

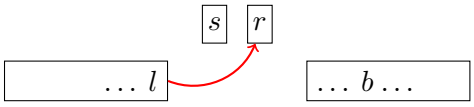



(a) Right-Arc: $s$ would have been in a register, and the Right-Arc would have violated $R1 \notin (\sigma_1, \beta_1) \wedge R2 \notin (\sigma_1, \beta_1)$.




(b) Store(right): $s$ would be on the stack when the registers were cleared, so Clear would have violated $\sigma_2 < R1 \wedge \sigma_1 \notin (R1, R2)$.


(c) Register-Stack(k, to-stack): If $s$ was on the stack, then if $s > R2$, Register-Stack(k, t-stack) would have violated $\sigma_2 < R2$; if $s < R2$, then $s \in (R1, R2)$, and Clear would have violated $\sigma_2 < R1 \wedge \sigma_1 \notin (R1, R2)$. If $s$ instead was in $R2$ (not shown), then it would get covered by $(l, r)$ and reduced by Clear.




(d) Register-Stack(k, to-register): $s$ must have been in $R2$. $s$ would get covered by $(l, r)$ and reduced by Clear.

Figure 4: If a stack-buffer arc $(s, b)$ is added in the two-registers system, there cannot have been an earlier arc $(l, r)$ with $l < s < r < b$, since it would then be impossible to keep $s$ unreduced without violating the preconditions.

the registers being cleared, and subsequent actions cannot introduce crossed arcs to the $last$ token or to its left (by the $\beta_1 > last$ and $\sigma_1 > last$ preconditions on storing and register-stack arcs, respectively). Thus, each set of tokens that were in registers simultaneously defines a crossing interval. Condition (a) of Definition 3 is satisfied, since all crossed arcs are incident to registers and at most two vertices are in registers at the same time.

Assume that a vertex $h$, $h \notin \{R1, R2\}$, has a child $m$ on the far side of its parent $g$ (i.e., either $h < g < m$ or $m < g < h$). The edge $(h, m)$ is guaranteed to be crossed and so was added through a register-stack arc (Lemma 1). The ordering $h < g < m$ is not possible, since if $(g, h)$ had been added through a left-arc, then $h$ would have been reduced, and if $(g, h)$ and $(h, m)$ were both added through register-stack arcs, then one of them would have violated the $(R_{close}, \sigma_1) \notin A$ or the $(\sigma_1, R_{far}) \notin A$ precondition. Similar reasoning can rule out $m < g < h$. Thus Condition (b) of Definition 3 is also satisfied. □

### 5.2 Completeness: 2-Crossing Interval trees → Two-Registers System

*Proof.* The portions of a 2-Crossing Interval tree in-between the crossing intervals can be constructed using the transitions from arc-eager. For a particular crossing interval $[l, r]$ and a particular choice of two vertices $a$ and $b$ incident to all all crossed arcs in the interval ($l \leq a < b \leq r$), $a$ and $b$ divide the interval into: $L = [l, a)$, $a$, $M = (a, b)$, $b$, $R = (b, r]$. All arcs incident to neither $a$ nor $b$ must lie entirely within $L$, $M$, or $R$.[4]

The parser begins by adding all arcs with both endpoints in $L$, using the standard arc-eager Shift/Reduce/Left-Arc/Right-Arc. It then shifts until $a$ is at the front of the buffer and stores $a$. It then repeats the same process to add the arcs lying entirely in $M$ until $b$ reaches the front of the buffer, adding the parent of $a$ with a Register-Stack(1, to-register) transition if the parent is in $M$ and the arc is uncrossed. $b$ is then stored, adding the arc between $a$ and $b$ if necessary. Throughout this process, the precondition $R1 \notin (\sigma_1, \beta_1) \wedge R2 \notin (\sigma_1, \beta_1)$ for left and right arcs is satisfied.

Next, the parser will repeatedly take Register-Stack transitions, interspersed with Reduce transitions, to add all the arcs with one endpoint in $\{a, b\}$ and the other in $L$ or $M$, working right-to-left from $b$ (i.e., from the top of the stack downwards). No shifts are done at this stage, so the $\sigma_2 < R2$ precondition on Register-Stack arcs is always satisfied. The $\sigma_1 > last$ precondition is also always satisfied since all vertices in the crossing interval will be to the right of the previous crossing interval boundary point. After all these arcs are done, if there are any uncrossed arcs incident to $a$ to the left that go outside of the crossing interval, they are added now with a Register-Stack transition.[5]

[4] E.g., if there were an arc not incident to $a$ or $b$ with one endpoint left of $a$ and one endpoint right of $a$, then this arc must be crossed or lie outside of the crossing interval.

[5] Only possible in the case $l = a$, in which case $\neg$IsCovered($a$) and the transition is allowed.

Finally, the arcs with at least one endpoint in $R$ are added, using Register-Stack arcs for those with the other endpoint in $\{a, b\}$ and Left-Arc/Right-Arc for those with both endpoints in $R$. Before any vertex incident to $a$ or $b$ is shifted onto the stack, all tokens on the stack to the right of $b$ are reduced.

After all these arcs are added, the crossing interval is complete. The boundary points of the interval that can still participate in uncrossed arcs with the exterior are left on the stack and buffer after the clear operation, so the rest of the tree is still parsable. □

## 6 Worst-case Runtime

The two-registers system runs in $O(n)$ time: it completes after at most $O(n)$ transitions and each transition takes constant time.

The total number of arc-adding actions (Left-Arc, Right-Arc, Register-Stack, or a Store that includes an arc) is bounded by $n$, as there are at most $n$ arcs in the final output. The net result of {Store, Store, Clear} triples of transitions decreases the number of tokens on the buffer by at least one, so these triples, plus the number of Shifts and Right-Arcs, are bounded by $n$. Finally, each token can be removed completely at most once, so the number of Left-Arcs and Reduces is bounded by $n$. Every transition fell into one of these categories, so the total number of transitions is bounded by $5n = O(n)$.

Each operation can be performed in constant time, as all operations involve moving vertices and/or adding arcs, and at most three vertices are ever moved (Clear) and at most one arc is ever added. Most preconditions can be trivially checked in constant time, such as checking whether a vertex already has a parent or not. The non-trivial precondition to check is acyclicity, and this can also be checked by adding some book-keeping variables that can be updated in constant time (full proof omitted due to space constraints). For example, in the derivation in Table 3, prior to the Register-Stack(2, to-stack) transition, $R1 \rightarrow^A R2$ (*helped* $\rightarrow^A$ *paint*). After the arc $(R2, \sigma_1)$ (*paint*, *house*) is added, $R2 \rightarrow^A \sigma_1$ and by transitivity, $R1 \rightarrow^A \sigma_1$. The top of the stack is then reduced, and since $\sigma_2$ does not have a parent to its right, it is not a descendant of $\sigma_1$, and so after *Hans* becomes the new $\sigma_1$, the system makes the update that $R1$, $R2 \twoheadrightarrow^A \sigma_1$.

## 7 Experiments

The experiments compare the two-registers transition system for mildly non-projective trees proposed here with two other transition systems: the arc-eager system for projective trees (Nivre, 2003) and the swap-based system for all non-projective trees (Nivre, 2009). We choose the swap-based system as our non-projective baseline as it currently represents the state-of-the-art in transition-based parsing (Bohnet et al., 2013), with higher empirical performance than the Attardi system or pseudo-projective parsing (Kuhlmann and Nivre, 2010).

The arc-eager system is a reimplementation of Zhang and Nivre (2011), using their rich feature set and beam search. The features for the two other transition systems are based on the same set, but with slight modifications to account for the different relevant domains of locality. In particular, for the swap transition system, we updated the features to account for the fact that this transition system is based on the arc-standard model and so the most relevant positions are the top two tokens on the stack. For the two-register system, we added features over properties of the tokens stored in each of the registers. All experiments use beam search with a beam of size 32 and are trained with ten iterations of averaged structured perceptron training. Training set trees that are outside of the reachable class (projective for arc-eager, 2-Crossing Intervals for two-registers) are transformed by lifting arcs (Nivre and Nilsson, 2005) until the tree is within the class. The test sets are left unchanged. We use the standard technique of parameterizing arc creating actions with dependency labels to produce labeled dependency trees.

Experiments use the ten datasets in Table 1 from the CoNLL 2006 and 2007 shared tasks (Buchholz and Marsi, 2006; Nivre et al., 2007). We report numbers using both gold and automatically predicted part-of-speech tags and morphological attribute-values as features. For the latter, the part of speech tagger is a first-order CRF model and the morphological tagger uses a greedy SVM per-attribute classifier. Evaluation uses CoNLL-X scoring conventions (Buchholz and Marsi, 2006) and we report both labeled and unlabeled attachment scores.

| Language | LAS (UAS) eager | swap | two-registers |
|---|---|---|---|
| Basque | 70.50 (78.06) | 69.66 (77.44) | **71.10** (**78.57**) |
| Czech | 79.60 (85.55) | **80.74** (**86.82**) | 79.75 (85.93) |
| Dutch | 78.69 (81.41) | 79.65 (82.69) | **80.77** (**83.91**) |
| English | 90.00 (91.18) | 90.16 (91.29) | **90.36** (**91.54**) |
| German | 88.34 (91.01) | 86.76 (89.56) | **89.08** (**91.95**) |
| Greek | 77.34 (**84.79**) | 76.90 (84.72) | **77.59** (84.77) |
| Hungarian | 80.00 (84.20) | 79.93 (84.40) | **80.21** (**84.91**) |
| Portuguese | **88.30** (**91.64**) | 87.92 (91.79) | 87.40 (91.20) |
| Slovene | 75.68 (83.97) | **76.34** (**84.47**) | 76.08 (84.33) |
| Turkish | 68.83 (77.34) | 70.71 (79.74) | **70.94** (**80.39**) |
| Average | 79.73 (84.92) | 79.88 (85.29) | **80.33** (**85.75**) |

Table 5: Labeled and Unlabeled Attachment Scores (LAS and UAS) on the CoNLL 2006/2007 Shared Task datasets (gold part-of-speech tags and morphology).

| Language | LAS (UAS) eager | swap | two-registers |
|---|---|---|---|
| Basque | **64.36** (**73.03**) | 63.23 (72.10) | 64.27 (72.32) |
| Czech | 75.92 (83.79) | **76.92** (**84.54**) | 76.37 (83.79) |
| Dutch | 78.59 (81.07) | 79.69 (83.03) | **80.77** (**83.71**) |
| English | 88.19 (89.77) | 88.68 (90.32) | **88.93** (**90.50**) |
| German | **87.74** (**90.62**) | 85.66 (88.40) | 87.60 (90.48) |
| Greek | **77.46** (**85.14**) | 76.29 (84.65) | 77.22 (84.82) |
| Hungarian | **75.88** (81.61) | 75.83 (81.89) | 75.71 (**82.43**) |
| Portuguese | **86.07** (**90.16**) | 85.65 (89.86) | 85.91 (**90.16**) |
| Slovene | **71.72** (**81.69**) | 71.36 (81.63) | 71.58 (81.43) |
| Turkish | 62.18 (74.22) | 63.12 (75.26) | **64.06** (**76.82**) |
| Average | 76.81 (83.11) | 76.64 (83.17) | **77.24** (**83.65**) |

Table 6: Labeled and Unlabeled Attachment Scores (LAS and UAS) on the CoNLL 2006/2007 Shared Task datasets (predicted part-of-speech tags and morphology).

### 7.1 Results

Table 5 shows the results using gold tags as features, which is the most common set-up in the literature. The two-registers transition system has on average 0.8% absolute higher unlabeled attachment accuracy than arc-eager across the ten datasets investigated. Its UAS is higher than arc-eager for eight out of the ten languages and is up to 2.5% (Dutch) or 3.0% (Turkish) absolute higher, while never more than 0.4% worse (Portuguese). The two-registers transition system is also more accurate than the alternate non-projective swap system on seven out of the ten languages, with more than 1% absolute improvements in UAS for Basque, Dutch, and German. The two-registers transition-system is still on average more accurate than either the arc-eager or swap systems using predicted tags as features (Table 6).

| Language | Crossed / Uncrossed eager | swap | two-registers |
|---|---|---|---|
| Basque | 33.10 / 83.32 | **39.37** / 82.52 | 34.49 / **83.58** |
| Czech | 43.98 / 87.37 | **68.76** / **87.63** | 55.42 / 87.24 |
| Dutch | 40.08 / **87.66** | **71.08** / 85.70 | 69.19 / 87.08 |
| English | 27.66 / 91.98 | **42.55** / 92.00 | **42.55** / **92.09** |
| German | 55.29 / 91.60 | 72.35 / 89.46 | **75.29** / **91.85** |
| Greek | 29.94 / 84.79 | **33.12** / 84.76 | 30.57 / **84.94** |
| Hungarian | 44.40 / **84.98** | 55.40 / 84.07 | **55.60** / 84.77 |
| Portuguese | 48.17 / **90.98** | **58.64** / 90.79 | 57.07 / 89.96 |
| Slovene | 41.83 / 83.60 | **47.91** / **84.05** | 44.11 / 83.65 |
| Turkish | 45.07 / 86.20 | **70.39** / 86.15 | 56.25 / **87.31** |
| Average | 32.51 / **87.25** | **55.96** / 86.72 | 52.05 / **87.25** |

Table 7: UAS from Table 5 for tokens in which the incoming arc in the gold tree is crossed or uncrossed (recall of both crossed and uncrossed arcs).

Finally, we analyzed the performance of each of these parsers on both crossed *and uncrossed* arcs. Even on languages with many non-projective sentences, the majority of arcs are not crossed. Table 7 partitions all scoring tokens into those whose incoming arc in the gold tree is crossed and those whose incoming arc is not crossed, and presents the UAS scores from Table 5 for each of these groups. On the crossed arcs, the swap system does the best, followed by the two-registers system, with the arc-eager system about 20% absolute less accurate. On the uncrossed arcs, the arc-eager and two-registers systems are tied, with the swap system less accurate.

## 8 Discussion and Related Work

There has been a significant amount of recent work on non-projective dependency parsing. In the transition-based parsing paradigm, the pseudo-projective parser of Nivre and Nilsson (2005) was an early attempt and modeled the problem by transforming non-projective trees into projective trees via transformations encoded in arc labels. While improving parsing accuracies for many languages, this method was both approximate and inefficient as the increase in the cardinality of the label set affected run time.

Attardi (2006) directly augmented the transition system to permit limited non-projectivity by allowing transitions between words not directly at the top of the stack or buffer. While this transition system had significant coverage, it is unclear how to precisely characterize the set of dependency trees that it

covers. Nivre (2009) introduced a transition system that covered all non-projective trees via a new *swap* transition that locally re-ordered words in the sentence. The downside of the swap transition is that it made worst-case run time quadratic. Also, as shown in Table 7, the attachment scores of *uncrossed* arcs decreases compared with arc-eager.

Two other transition systems that can be seen as generalizations of arc-eager are the 2-Planar transition system (Gómez-Rodríguez and Nivre, 2010; Gómez-Rodríguez and Nivre, 2013), which adds a second stack, and the transition system of Choi (Choi and McCallum, 2013), which adds a deque. The arc-eager, 2-registers, 2-planar, and the Choi transition systems can be seen as along a continuum for trading off various properties. In terms of coverage, projective trees (arc-eager) $\subset$ 2-Crossing Interval trees (this paper) $\subset$ 2-planar trees $\subset$ all directed trees (Choi). The Choi system uses a quadratic number of transitions in the worst case, while arc-eager, 2-registers, and 2-planar all use at most $O(n)$ transitions. Checking for cycles does not need to be done at all in the arc-eager system, can be with a few constant operations in the 2-registers system, and can be done in amortized constant time for the other systems (Gómez-Rodríguez and Nivre, 2013).

In the graph-based parsing literature, there has also been a plethora of work on non-projective parsing (McDonald et al., 2005; Martins et al., 2009; Koo et al., 2010). Recent work by Pitler and colleagues is the most relevant to the work described here (Pitler et al., 2012, 2013, 2014). Like this work, Pitler et al. define a restricted class of non-projective trees and then a graph-based parsing algorithm that parses exactly that set.

The register mechanism in two-registers transition parsing bears a resemblance to registers in Augmented Transition Networks (ATNs) (Woods, 1970). In ATNs, global registers are introduced to account for a wide range of natural language phenomena. This includes long-distance dependencies, which is a common source of non-projective trees. While transition-based parsing and ATNs use quite different control and data structures, this observation does raise an interesting question about the relationship between these two parsing paradigms.

There are many additional points of interest to explore based on this study. A first step would be to generalize the two-registers transition system to a $k$-registers system that can parse exactly $k$-Crossing Interval trees. This will necessarily lead to an asymptotic increase in run-time as $k$ approaches $n$. With larger values of $k$, the system would need additional transitions to add arcs between the registers (extending the Store transition to consider all subsets of arcs with the existing registers would become exponential in $k$). If $k$ were to increase all the way to $n$, such a system would probably look very similar to list-based systems that consider all pairs of arcs (Covington, 2001; Nivre, 2008).

Another direction would be to define dynamic oracles around the two-registers transition system (Goldberg and Nivre, 2012; Goldberg and Nivre, 2013). The additional transitions here have interpretations in terms of which trees are still reachable (Register-Stack($\cdot$) adds an arc; Store and Clear indicate that particular vertices should be incident to crossed arcs or are finished with crossed arcs, respectively). The two-registers system is not quite arc-decomposable (Goldberg and Nivre, 2013): if the wrong vertex is stored in a register then a later pair of crossed arcs might both be individually reachable but not jointly reachable. However, there may be a "crossing-sensitive" variant of arc-decomposability that takes into account the vertices crossed arcs are incident to that would apply here.

## 9 Conclusion

In this paper we presented $k$-Crossing Interval trees, a class of mildly non-projective trees with high empirical coverage. For the case of $k = 2$, we also presented a transition system that is sound and complete with respect to this class that is a generalization of the arc-eager transition system and maintains many of its desirable properties, most notably a linear worst-case run-time. Empirically, this transition system outperforms its projective counterpart as well as a quadratic swap-based transition system with larger coverage.

## Acknowledgments

We'd like to thank Mike Collins, Terry Koo, Joakim Nivre, Fernando Pereira, and Slav Petrov for helpful discussions and comments.

## References


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

B. Bohnet, J. Nivre, I. Boguslavsky, R. Farkas, F. Ginter, and J. Hajic. 2013. Joint morphological and syntactic analysis for richly inflected languages. *TACL*, 1:415–428.

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

J. D. Choi and A. McCallum. 2013. Transition-based dependency parsing with selectional branching. In *ACL*, pages 1052–1062.

M. A. Covington. 2001. A fundamental algorithm for dependency parsing. *Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102.

Y. Goldberg and J. Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *COLING*.

Y. Goldberg and J. Nivre. 2013. Training deterministic parsers with non-deterministic oracles. *TACL*, 1:403–414.

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 and J. Nivre. 2013. Divisible transition systems and multiplanar dependency parsing. *Computational Linguistics*, 39(4):799–845.

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.

M. Kuhlmann and J. Nivre. 2006. Mildly non-projective dependency structures. In *Proceedings of COLING/ACL*, pages 507–514.

M. Kuhlmann and J. Nivre. 2010. Transition-based techniques for non-projective dependency parsing. *Northern European Journal of Language Technology*, 2(1):1–19.

M. Kuhlmann. 2013. Mildly non-projective dependency grammar. *Computational Linguistics*, 39(2).

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.

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.

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. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553.

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. *TACL*, 1(Mar):13–24.

E. Pitler. 2013. Models for improved tractability and accuracy in dependency parsing. *University of Pennsylvania*.

E. Pitler. 2014. A crossing-sensitive third-order factorization for dependency parsing. *TACL*, 2(Feb):41–54.

S. M. Shieber. 1985. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8(3):333–343.

W. A. Woods. 1970. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606.

Y. Zhang and S. Clark. 2008. A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In *Proceedings of EMNLP*, pages 562–571.

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