A Polynomial-Time Dynamic Oracle for Non-Projective Dependency Parsing

Carlos Gómez-Rodríguez
Departamento de Computación
Universidade da Coruña, Spain
cgomezr@udc.es

Francesco Sartorio
Department of Information Engineering
University of Padua, Italy
sartorio@dei.unipd.it

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

Abstract

The introduction of dynamic oracles has considerably improved the accuracy of greedy transition-based dependency parsers, without sacrificing parsing efficiency. However, this enhancement is limited to projective parsing, and dynamic oracles have not yet been implemented for parsers supporting non-projectivity. In this paper we introduce the first such oracle, for a non-projective parser based on Attardi's parser. We show that training with this oracle improves parsing accuracy over a conventional (static) oracle on a wide range of datasets.

1 Introduction

Greedy transition-based parsers for dependency grammars have been pioneered by Yamada and Matsumoto (2003) and Nivre (2003). These methods incrementally process the input sentence from left to right, predicting the next parsing action, called transition, on the basis of a compact representation of the derivation history.

Greedy transition-based parsers can be very efficient, allowing web-scale parsing with high throughput. However, the accuracy of these methods still falls behind that of transition-based parsers using beam-search, where the accuracy improvement is obtained at the cost of a decrease in parsing efficiency; see for instance Zhang and Nivre (2011), Huang and Sagae (2010), Choi and McCallum (2013). As an alternative to beam-search, recent research on transition-based parsing has therefore explored possible ways of improving accuracy at no extra cost in parsing efficiency.

The training of transition-based parsers relies on a component called the parsing oracle, which maps parser configurations to optimal transitions with respect to a gold tree. A discriminative model is then trained to simulate the oracle's behavior, and is later used for decoding. Traditionally, so-called static oracles have been exploited in training, where a static oracle is defined only for configurations that have been reached by computations with no mistake, and it returns a single canonical transition among those that are optimal.

Very recently, Goldberg and Nivre (2012), Goldberg and Nivre (2013) and Goldberg et al. (2014) showed that the accuracy of transition-based parsers can be substantially improved using dynamic oracles. A dynamic oracle returns the set of all transitions that are optimal for a given configuration, with respect to the gold tree, and is well-defined and correct for every configuration that is reachable by the parser.

Naïve implementations of dynamic oracles run in exponential time, since they need to simulate all possible computations of the parser for the input configuration. Polynomial-time implementations of dynamic oracles have been proposed by the above mentioned authors for several projective dependency parsers. To our knowledge, no polynomial-time algorithm has been published for transition-based parsers based on non-projective dependency grammars.

In this paper we consider a restriction of a transition-based, non-projective parser originally presented by Attardi (2006). This restriction was further investigated by Kuhlmann and Nivre (2010) and Cohen et al. (2011). We provide an implementation for a dynamic oracle for this parser running in polynomial time.

We experimentally compare the parser trained with the dynamic oracle to a baseline obtained by training with a static oracle. Significant accuracy improvements are achieved on many languages when using our dynamic oracle. To our knowledge, these are the first experimental results on non-projective parsing based on a dynamic oracle.

2 Preliminary Definitions

Transition-based dependency parsing was originally introduced by Yamada and Matsumoto (2003) and Nivre (2003). In this section we briefly summarize the notation we use for this framework and introduce the notion of dynamic oracle.

2.1 Transition-Based Dependency Parsing

We represent an input sentence as a string $w = w_0 \cdots w_n$, $n \geq 1$, where each w_i with $i \neq 0$ is a lexical symbol and w_0 is a special symbol called root. Set $V_w = \{i \mid 0 \leq i \leq n\}$ denotes the symbol occurrences in w. For $i, j \in V_w$ with $i \neq j$, we write $i \rightarrow j$ to denote a grammatical **dependency** of some unspecified type between w_i and w_j, where w_i is the head and w_j is the dependent.

A **dependency tree** t for w is a directed tree with node set V_w and with root node 0. An arc of t is a pair (i, j), encoding a dependency $i \rightarrow j$; we will often use the latter notation to denote arcs.

A transition-based dependency parser typically uses a stack data structure to process the input string from left to right, in a way very similar to the classical push-down automaton for context-free languages (Hopcroft et al., 2006). Each stack element is a node from V_w, representing the root of a dependency tree spanning some portion of the input w, and no internal state is used. At each step the parser applies some transition that updates the stack and/or consumes one symbol from the input. Transitions may also construct new dependencies, which are added to the current configuration of the parser.

We represent the **stack** as an ordered sequence $\sigma = [h_d, \ldots, h_1]$, $d \geq 0$, of nodes $h_i \in V_w$, with the topmost element placed at the right. When $d = 0$, we have the empty stack $\sigma = []$. We use the vertical bar to denote the append operator for σ, and write $\sigma = \sigma'|h_1$ to indicate that h_1 is the topmost element of σ.

The portion of the input string still to be processed by the parser is called the **buffer**. We represent the buffer as an ordered sequence $\beta = [i, \ldots, n]$ of nodes from V_w, with i the first element of the buffer. We denote the empty buffer as $\beta = []$. Again, we use the vertical bar to denote the append operator, and write $\beta = i|\beta'$ to indicate that i is the first symbol occurrence of β; consequently, we have $\beta' = [i + 1, \ldots, n]$.

In a transition-based parser, the parsing process is defined through the technical notions of configuration and transition. A **configuration** of the parser relative to w is a triple $c = (\sigma, \beta, A)$, where σ and β are a stack and a buffer, respectively, and A is the set of arcs that have been built so far. A **transition** is a partial function mapping the set of parser configurations into itself. Each transition-based parser is defined by means of some finite inventory of transitions. We will later introduce the specific inventory of transitions for the parser that we investigate in this paper. We use the symbol $\vdash$ to denote the binary relation formed by the union of all transitions of a parser.

With the notions of configuration and transition in place, we can define a **computation** of the parser on w as a sequence $c_0, c_1, \ldots, c_m$, $m \geq 0$, of configurations relative to w, under the condition that $c_{i-1} \vdash c_i$ for each i with $1 \leq i \leq m$. We use the reflexive and transitive closure of $\vdash$, written $\vdash^*$, to represent computations.

2.2 Configuration Loss and Dynamic Oracles

A transition-based dependency parser is a non-deterministic device, meaning that a given configuration can be mapped into several configurations by the available transitions. However, in several implementations the parser is associated with a discriminative model that, on the basis of some features of the current configuration, always chooses a single transition. In other words, the model is used to run the parser as a pseudo-deterministic device. The training of the discriminative model relies on a component called the parsing **oracle**, which maps parser configurations to "optimal" transitions with respect to some reference dependency tree, which we call the **gold** tree.

Traditionally, so-called **static** oracles have been used which return a single, canonical transition and they do so only for configurations that can reach the gold tree, that is, configurations representing parsing histories with no mistake. In recent work, Goldberg and Nivre (2012), Goldberg and Nivre (2013) and Goldberg et al. (2014) have introduced **dynamic** oracles, which return the set of all transitions that are optimal with respect to a gold tree, and are well-defined and correct for every configuration that is reachable by the parser. These authors have shown that the accuracy of transition-based dependency parsers can be substantially improved if dynamic oracles are used in place of static ones. In what follows, we provide a mathematical definition of dynamic oracles, following Goldberg et al. (2014).

$$\begin{aligned}
(\sigma, k|\beta, A) &\vdash_{\mathsf{sh}} (\sigma|k, \beta, A)\\
(\sigma|i|j, \beta, A) &\vdash_{\mathsf{la}} (\sigma|j, \beta, A \cup \{j \to i\})\\
(\sigma|i|j, \beta, A) &\vdash_{\mathsf{ra}} (\sigma|i, \beta, A \cup \{i \to j\})\\
(\sigma|i|j|k, \beta, A) &\vdash_{\mathsf{la}_2} (\sigma|j|k, \beta, A \cup \{k \to i\})\\
(\sigma|i|j|k, \beta, A) &\vdash_{\mathsf{ra}_2} (\sigma|i|j, \beta, A \cup \{i \to k\})
\end{aligned}$$

Figure 1: Transitions of the non-projective parser.

Let t_1 and t_2 be dependency trees for w, with arc sets A_1 and A_2, respectively. The **loss** of t_1 with respect to t_2 is defined as

$$\mathcal{L}(t_1, t_2) = |A_1 \setminus A_2| \ . \qquad (1)$$

Note that $\mathcal{L}(t_1, t_2) = \mathcal{L}(t_2, t_1)$, since $|A_1| = |A_2|$. Furthermore $\mathcal{L}(t_1, t_2) = 0$ if and only if t_1 and t_2 are the same tree.

Let c be a configuration of a transition-based parser relative to w. Let also $\mathcal{D}(c)$ be the set of all dependency trees that can be obtained in a computation of the form $c \vdash^* c_f$, where c_f is a final configuration, that is, a configuration that has constructed a dependency tree for w. We extend the loss function in (1) to configurations by letting

$$\mathcal{L}(c, t_2) = \min_{t_1 \in \mathcal{D}(c)} \mathcal{L}(t_1, t_2) \ . \qquad (2)$$

Let t_G be the gold tree for w. Quantity $\mathcal{L}(c, t_G)$ can be used to define a dynamic oracle as follows. For any transition $\vdash_\tau$ in the finite inventory of our parser, we use the functional notation $\tau(c) = c'$ in place of $c \vdash_\tau c'$. We then let

$$\mathsf{oracle}(c, t_G) = \{\tau \mid \mathcal{L}(\tau(c), t_G) - \mathcal{L}(c, t_G) = 0\} \ . \qquad (3)$$

In words, (3) provides the set of transitions that do not increase the loss of c; we call these transitions optimal for c.

A naïve way of implementing (3) would be to explicitly compute the set $\mathcal{D}(c)$ in (2), which has exponential size. More interestingly, the implementation of dynamic oracles proposed by the above cited authors all run in polynomial time. These oracles are all defined for projective parsing. In this paper, we present a polynomial-time oracle for a non-projective parser.

3 Non-Projective Dependency Parsing

In this section we introduce a parser for non-projective dependency grammars that is derived from the transition-based parser originally presented by Attardi (2006), and was further investigated by Kuhlmann and Nivre (2010) and Cohen et al. (2011). Our definitions follow the framework introduced in Section 2.1.

We start with some additional notation. Let t be a dependency tree for w and let k be a node of t. Consider the complete subtree t' of t rooted at k, that is, the subtree of t induced by k and all of the descendants of k in t. The **span** of t' is the subsequence of tokens in w represented by the nodes of t'. Node k has **gap-degree** 0 if the span of t' forms a (contiguous) substring of w. A dependency tree is called **projective** if all of its nodes have gap-degree 0; a dependency tree which is not projective is called **non-projective**.

Given w as input, the parser starts with the initial configuration $([], [0, \dots, n], \emptyset)$, consisting of an empty stack, a buffer with all the nodes representing the symbol occurrences in w, and an empty set of constructed dependencies (arcs). The parser stops when it reaches a final configuration of the form $([0], [], A)$, consisting of a stack with only the root node and of an empty buffer; in any such configuration, set A always implicitly defines a valid dependency tree (rooted in node 0).

The core of the parser consists of an inventory of five transitions, defined in Figure 1. Each transition is specified using the free variables σ, β, A, i, j and k. As an example, the schema $(\sigma|i|j, \beta, A) \vdash_{\mathsf{la}} (\sigma|j, \beta, A \cup \{j \to i\})$ means that if a configuration c matches the antecedent, then a new configuration is obtained by instantiating the variables in the consequent accordingly.

The transition $\vdash_{\mathsf{sh}}$, called shift, reads a new token from the input sentence by removing it from the buffer and pushing it into the stack. Each of the other transitions, collectively called reduce transitions, has the effect of building a dependency between two nodes in the stack, and then removing the dependent node from the stack. The removal of the dependent ensures that the output dependency tree is built in a bottom-up order, collecting all of the dependents of each node i before linking i to its head.

The transition $\vdash_{\mathsf{la}}$, called left-arc, creates a leftward arc where the topmost stack node is the head and the second topmost node is the dependent, and removes the latter from the stack. The transition $\vdash_{\mathsf{ra}}$, called right-arc, is defined symmetrically, so that the topmost stack node is at-

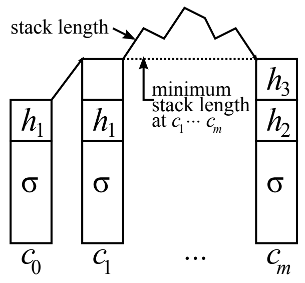


Figure 2: General form of the computations associated with an item $[h_1, h_2, h_3]$.

tached as a dependent of the second topmost node. The combination of the shift, left-arc and right-arc transitions provides complete coverage of projective dependency trees, but no support for non-projectivity, and corresponds to the so-called arc-standard parser introduced by Nivre (2004).

Support for non-projective dependencies is achieved by adding the transitions $\vdash_{\mathsf{la}_2}$ and $\vdash_{\mathsf{ra}_2}$, which are variants of the left-arc and right-arc transitions, respectively. These new transitions create dependencies involving the first and the *third* topmost nodes in the stack. The creation of dependencies between non-adjacent stack nodes might produce crossing arcs and is the key to the construction of non-projective trees.

Recall that transitions are partial functions, meaning that they might be undefined for some configurations. Specifically, the shift transition is only defined for configurations with a non-empty buffer. Similarly, the left-arc and right-arc transitions can only be applied if the length of the stack is at least 2, while the transitions $\vdash_{\mathsf{la}_2}$ and $\vdash_{\mathsf{ra}_2}$ require at least 3 nodes in the stack.

Transitions $\vdash_{\mathsf{la}_2}$ and $\vdash_{\mathsf{ra}_2}$ were originally introduced by Attardi (2006) together with other, more complex transitions. The parser we define here is therefore more restrictive than Attardi (2006), meaning that it does not cover all the non-projective trees that can be processed by the original parser. However, the restricted parser has recently attracted some research interest, as it covers the vast majority of non-projective constructions appearing in standard treebanks (Attardi, 2006; Kuhlmann and Nivre, 2010), while keeping simplicity and interesting properties like being compatible with polynomial-time dynamic programming (Cohen et al., 2011).

4 Representation of Computations

Our oracle algorithm exploits a dynamic programming technique which, given an input string, combines certain pieces of a computation of the parser from Section 3 to obtain larger pieces. In order to efficiently encode pieces of computations, we borrow a representation proposed by Cohen et al. (2011), which is introduced in this section.

Let $w = a_0 \cdots a_n$ and V_w be specified as in Section 2, and let w' be some substring of w. (The specification of w' is not of our concern in this section.) Let also $h_1, h_2, h_3 \in V_w$. We are interested in computations of the parser processing the substring w' and having the form $c_0, c_1, \ldots, c_m$, $m \geq 1$, that satisfy both of the following conditions, exemplified in Figure 2.

- For some sequence of nodes σ with $|\sigma| \geq 0$, the stack associated with c_0 has the form $\sigma|h_1$ and the stack associated with c_m has the form $\sigma|h_2|h_3$.
- For each intermediate configuration c_i, $1 \leq i \leq m-1$, the stack associated with c_i has the form $\sigma\sigma_i$, where σ_i is a sequence of nodes with $|\sigma_i| \geq 2$.

An important property of the above definition needs to be discussed here, which is at the heart of the polynomial-time algorithm in the next section. If in $c_0, c_1, \ldots, c_m$ we replace σ with a different sequence σ', we obtain a valid computation for w' constructing exactly the same dependencies as the original computation. To see this, let $c_{i-1} \vdash_{\tau_i} c_i$ for each i with $1 \leq i \leq m$. Then $\vdash_{\tau_1}$ must be a shift, otherwise $|\sigma_1| \geq 2$ would be violated. Consider now a transition $\vdash_{\tau_i}$ with $2 \leq i \leq m$ that builds some dependency. From $|\sigma_i| \geq 2$ we derive $|\sigma_{i-1}| \geq 3$. We can easily check from Figure 1 that none of the nodes in σ can be involved in the constructed dependency.

Intuitively, the above property asserts that the sequence of transitions $\vdash_{\tau_1}, \vdash_{\tau_2}, \ldots, \vdash_{\tau_m}$ can be applied to parse substring w' independently of the context σ. This suggests that we can group into an equivalence class all the computations satisfying the conditions above, for different values of σ. We indicate such class by means of the tuple $[h_1, h_2h_3]$, called **item**. It is easy to see that each item represents an exponential number of computations. In the next section we will show how we can process items with the purpose of obtaining an efficient computation for dynamic oracles.

5 Dynamic Oracle Algorithm

Our algorithm takes as input a gold tree t_G for string w and a parser configuration $c = (\sigma, \beta, A)$ relative to w, specified as in Section 2. We assume that t_G can be parsed by the non-projective parser of Section 3 starting from the initial configuration.

5.1 Basic Idea

The algorithm consists of two separate stages, informally discussed in what follows. In the first stage we identify some tree fragments of t_G that can be constructed by the parser after reaching configuration c, in a way that does not depend on the content of σ. This means that these fragments can be precomputed by looking only into β. Furthermore, since these fragments are subtrees of t_G, their computation has no effect on the overall loss of a computation on w.

For each fragment t with the above properties, we replace all the nodes in β that are also nodes of t with the root node of t itself. The result of the first stage is therefore a new node sequence shorter than β, which we call the reduced buffer β_R.

In the second stage of the algorithm we use a variant of the tabular method developed by Cohen et al. (2011), which was originally designed to simulate all computations of the parser in Section 3 on an input string w. We run the above method on the concatenation of the stack and the reduced buffer, with some additional constraints that restrict the search space in two respects. First, we visit only those computations of the parser that step through configuration c. Second, we reach only those dependency trees that contain all the tree fragments precomputed in the first stage. We can show that such search space always contains at least one dependency tree with the desired loss, which we then retrieve performing a Viterbi search.

5.2 Preprocessing of the Buffer

Let t be a complete subtree of t_G, having root node k in β. Consider the following two conditions, defined on t.

- *Bottom-up completeness*: No arc $i \rightarrow j$ in t is such that i is a node in β, $i \neq k$, and j is a node in σ.
- *Zero gap-degree*: The nodes of t that are in β form a (contiguous) substring of w.

We claim that if t satisfies the above conditions, then we can safely reduce the nodes of t appearing in β, replacing them with node k. We only report here an informal discussion of this claim, and omit a formal proof.

As a first remark, recall that our parser implements a purely bottom-up strategy. This means that after a tree has been constructed, all of its nodes but the root are removed from the parser configuration. Then the Bottom-up completeness condition guarantees that if we remove from β all nodes of t but k, the nodes of t that are in σ can still be processed in a way that does not affect the loss, since their parent must be either k or a node that is neither in β nor in σ. Note that the nodes of t that are neither in β nor in σ are irrelevant to the precomputation of t from β, since these nodes have already been attached and are no longer available to the parser.

As a second remark, the Zero gap-degree condition guarantees that the span of t over the nodes of β is not interleaved by nodes that do not belong to t. This is also an important requirement for the precomputation of t from β, since a tree fragment having a discontinuous span over β might not be constructable independently of σ. More specifically, parsing such fragment implies dealing with the nodes in the discontinuities, and this might require transitions involving nodes from σ.

We can now use the sufficient condition above to compute β_R. We process β from left to right. For each node k, we can easily test the Bottom-up completeness condition and the Zero gap-degree condition for the complete subtree t of t_G rooted at k, and perform the reduction if both conditions are satisfied. Note that in this process a node k resulting from the reduction of t might in turn be removed from β if, at some later point, we reduce a supertree of t.

5.3 Computation of the Loss

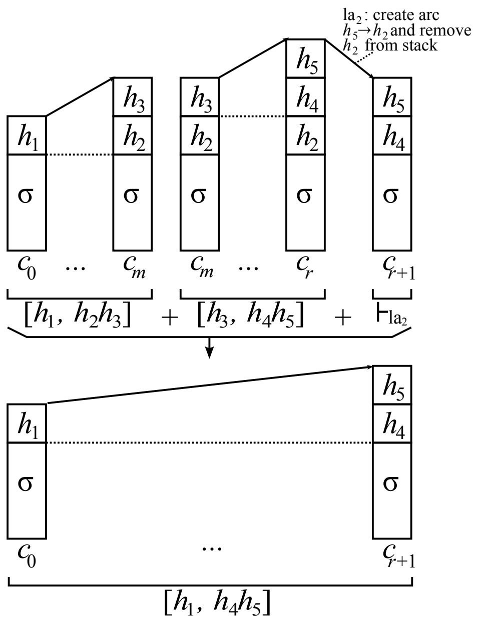
We describe here our dynamic programming algorithm for the computation of the loss of an input configuration c. We start with some additional notation. Let $\gamma = \sigma\beta_R$ be the concatenation of σ and β_R, which we treat as a string of nodes. For integers i with $0 \leq i \leq |\gamma| - 1$, we write $\gamma[i]$ to denote the $(i + 1)$-th node of γ. Let also $\ell = |\sigma|$. Symbol ℓ is used to mark the boundary between the stack and the reduced buffer in γ, thus $\gamma[i]$ with $i < \ell$ is a node of σ, while $\gamma[i]$ with $i \geq \ell$ is a node of β_R.

Algorithm 1 computes the loss of c by processing the sequence γ in a way quite similar to the standard nested loop implementation of the CKY parser for context-free grammars (Hopcroft et al., 2006). The algorithm uses a two-dimensional array $\mathcal{T}$ whose indexes range from 0 to $|\gamma| = \ell + |\beta_R|$, and only the cells $\mathcal{T}[i, j]$ with $i < j$ are filled.

We view each $\mathcal{T}[i, j]$ as an association list whose keys are items $[h_1, h_2h_3]$, defined in the context of the substring $\gamma[i] \cdots \gamma[j-1]$ of γ; see Section 4. The value stored at $\mathcal{T}[i, j]([h_1, h_2h_3])$ is the minimum loss contribution due to the computations represented by $[h_1, h_2h_3]$. For technical reasons, we assume that our parser starts with a symbol $\$ \notin V_w$ in the stack, denoting the bottom of the stack.

We initialize the table by populating the cells of the form $\mathcal{T}[i, i+1]$ with information about the trivial computations consisting of a single $\vdash_{\mathsf{sh}}$ transition that shifts the node $\gamma[i]$ into the stack. These computations are known to have zero loss contribution, because a $\vdash_{\mathsf{sh}}$ transition does not create any arcs. In the case where the node $\gamma[i]$ belongs to σ, i.e., $i < \ell$, we assign loss contribution 0 to the entry $\mathcal{T}[i, i+1]([\gamma[i-1], \gamma[i-1]\gamma[i]])$ (line 3 of Algorithm 1), because $\gamma[i]$ is shifted with $\gamma[i-1]$ at the top of the stack. On the other hand, if $\gamma[i]$ is in β, i.e., $i \geq \ell$, we assign loss contribution 0 to several entries in $\mathcal{T}[i, i+1]$ (line 6) because, at the time $\gamma[i]$ is shifted, the content of the stack depends on the transitions executed before that point.

After the above initialization, we consider pairs of contiguous substrings $\gamma[i] \cdots \gamma[k-1]$ and $\gamma[k] \cdots \gamma[j-1]$ of γ. At each inner iteration of the nested loops of lines 7-11 we update cell $\mathcal{T}[i, j]$ based on the content of the cells $\mathcal{T}[i, k]$ and $\mathcal{T}[k, j]$. We do this through the procedure PROCESSCELL$(\mathcal{T}, i, k, j)$, which considers all pairs of keys $[h_1, h_2h_3]$ in $\mathcal{T}[i, k]$ and $[h_3, h_4h_5]$ in $\mathcal{T}[k, j]$. Note that we require the index h_3 to match between both items, meaning that their computations can be concatenated. In this way, for each reduce transition τ in our parser, we compute the loss contribution for a new piece of computation defined by concatenating a computation with minimum loss contribution in the first item and a computation with minimum loss contribution in the second item, followed by the transition τ. The fact that the new piece of computation can be represented by an item is exemplified in Figure 3 for the case $\tau = \vdash_{\mathsf{ra}_2}$.



la_2: create arc $h_5 \to h_2$ and remove h_2 from stack
h_1
σ
h_3
h_2
h_4
h_5
c_0 … c_m
c_m … c_r
c_{r+1}
$[h_1,\ h_2h_3]$ + $[h_3,\ h_4h_5]$ + $\vdash_{\mathsf{la}_2}$
$[h_1,\ h_4h_5]$

Figure 3: Concatenation of two computations/items and transition $\vdash_{\mathsf{ra}_2}$, resulting in a new computation/item.

The computed loss contribution is used to update the entry in $\mathcal{T}[i, j]$ corresponding to the item associated with the new computation. Observe how the loss contribution provided by the arc created by τ is computed by the δ_G function at lines 17, 20, 23 and 26, which is defined as:

$$\delta_G(i \to j) \quad = \quad \begin{cases} 0, & \text{if } i \to j \text{ is in } t_G; \\ 1, & \text{otherwise.} \end{cases} \qquad (4)$$

We remark that the nature of our problem allows us to apply several shortcuts and optimizations that would not be possible in a setting where we actually needed to parse the string γ. First, the range of variable i in the loop in line 8 starts at $\max\{0, \ell - d\}$, rather than at 0, because we do not need to combine pairs of items originating from nodes in σ below the topmost node, as the items resulting from such combinations correspond to computations that do not contain our input configuration c. Second, when we have set values for i such that $i+2 < \ell$, we can omit calling PROCESSCELL for values of the parameter k ranging from $i+2$ to $\ell-1$, as those calls would use as their input one of the items described above, which are not of interest. Finally, when processing substrings that are entirely in β_R ($i \geq \ell$) we can restrict the transitions that we explore to those that generate arcs that either are in the gold tree t_G, or have a parent node which is not present in γ (see conditions in

Algorithm 1 Computation of the loss function

```
 1: $\mathcal{T}[0,1]([\$, \$0]) \leftarrow 0$                         ▷ shift node 0 on top of empty stack symbol $
 2: for $i \leftarrow 1$ to $\ell - 1$ do
 3:     $\mathcal{T}[i,i+1]([\gamma[i-1], \gamma[i-1]\gamma[i]]) \leftarrow 0$      ▷ shift node $\gamma[i]$ with $\gamma[i-1]$ on top of the stack
 4: for $i \leftarrow \ell$ to $|\gamma|$ do
 5:     for $h \leftarrow 0$ to $i - 1$ do
 6:         $\mathcal{T}[i,i+1]([\gamma[h], \gamma[h]\gamma[i]]) \leftarrow 0$      ▷ shift node $\gamma[i]$ with $\gamma[h]$ on top of the stack
 7: for $d \leftarrow 2$ to $|\gamma|$ do                                 ▷ consider substrings of length $d$
 8:     for $i \leftarrow \max\{0, \ell - d\}$ to $|\gamma| - d$ do                ▷ $i$ = beginning of substring
 9:         $j \leftarrow i + d$                                            ▷ $j - 1$ = end of substring
10:         ProcessCell($\mathcal{T}, i, i+1, j$)          ▷ We omit the range $k = i+2$ to $\max\{i+2, \ell\} - 1$
11:         for $k \leftarrow \max\{i+2, \ell\}$ to $j$ do                  ▷ factorization of substring at $k$
12:             ProcessCell($\mathcal{T}, i, k, j$)
13: return $\mathcal{T}[0, |\gamma|]([\$, \$0]) + \sum_{i \in [0, \ell-1]} \mathcal{L}_c(\sigma[i], t_G)$

14: procedure ProcessCell($\mathcal{T}, i, k, j$)
15:     for each key $[h_1, h_2h_3])$ defined in $\mathcal{T}[i,k]$ do
16:         for each key $[h_3, h_4h_5])$ defined in $\mathcal{T}[k,j]$ do          ▷ $h_3$ must match between the two entries
17:             $loss_{\mathsf{la}} \leftarrow \mathcal{T}[i,k]([h_1, h_2h_3]) + \mathcal{T}[k,j]([h_3, h_4h_5]) + \delta_G(h_5 \rightarrow h_4)$
18:             if $(i < \ell) \vee \delta_G(h_5 \rightarrow h_4) = 0 \vee (h_5 \not\in \gamma)$ then
19:                 $\mathcal{T}[i,j]([h_1, h_2h_5]) \leftarrow \min\{loss_{\mathsf{la}}, \mathcal{T}[i,j]([h_1, h_2h_5])\}$     ▷ cell update $\vdash_{\mathsf{la}}$
20:             $loss_{\mathsf{ra}} \leftarrow \mathcal{T}[i,k]([h_1, h_2h_3]) + \mathcal{T}[k,j]([h_3, h_4h_5]) + \delta_G(h_4 \rightarrow h_5)$
21:             if $(i < \ell) \vee \delta_G(h_4 \rightarrow h_5) = 0 \vee (h_4 \not\in \gamma)$ then
22:                 $\mathcal{T}[i,j]([h_1, h_2h_4]) \leftarrow \min\{loss_{\mathsf{ra}}, \mathcal{T}[i,j]([h_1, h_2h_4])\}$     ▷ cell update $\vdash_{\mathsf{ra}}$
23:             $loss_{\mathsf{la}_2} \leftarrow \mathcal{T}[i,k]([h_1, h_2h_3]) + \mathcal{T}[k,j]([h_3, h_4h_5]) + \delta_G(h_5 \rightarrow h_2)$
24:             if $(i < \ell) \vee \delta_G(h_5 \rightarrow h_2) = 0 \vee (h_5 \not\in \gamma)$ then
25:                 $\mathcal{T}[i,j]([h_1, h_4h_5]) \leftarrow \min\{loss_{\mathsf{la}_2}, \mathcal{T}[i,j]([h_1, h_4h_5])\}$     ▷ cell update $\vdash_{\mathsf{la}_2}$
26:             $loss_{\mathsf{ra}_2} \leftarrow \mathcal{T}[i,k]([h_1, h_2h_3]) + \mathcal{T}[k,j]([h_3, h_4h_5]) + \delta_G(h_2 \rightarrow h_5)$
27:             if $(i < \ell) \vee \delta_G(h_2 \rightarrow h_5) = 0 \vee (h_2 \not\in \gamma)$ then
28:                 $\mathcal{T}[i,j]([h_1, h_2h_4]) \leftarrow \min\{loss_{\mathsf{ra}_2}, \mathcal{T}[i,j]([h_1, h_2h_4])\}$     ▷ cell update $\vdash_{\mathsf{ra}_2}$
```

lines 18, 21, 24, 27), because we know that incorrectly attaching a buffer node as a dependent of another buffer node, when the correct head is available, can never be an optimal decision in terms of loss.

Once we have filled the table $\mathcal{T}$, the loss for the input configuration c can be obtained from the value of the entry $\mathcal{T}[0, |\gamma|]([\$, \$0])$, representing the minimum loss contribution among computations that reach the input configuration c and parse the whole input string. To obtain the total loss, we add to this value the loss contribution accumulated by the dependency trees with root in the stack σ of c. This is represented in Algorithm 1 as $\sum_{i \in [0, \ell-1]} \mathcal{L}_c(\sigma[i], t_G)$, where $\mathcal{L}_c(\sigma[i], t_G)$ is the count of the descendants of $\sigma[i]$ (the $(i+1)$-th element of σ) that had been assigned the wrong head by the parser with respect to t_G.

5.4 Sample Run

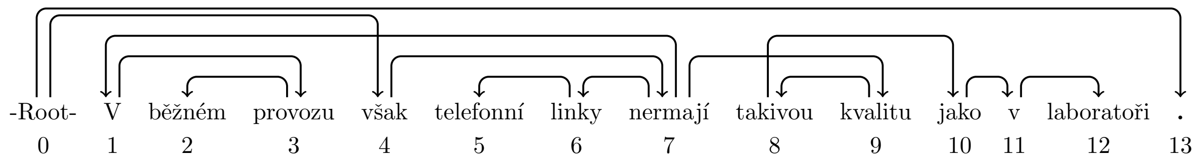
Consider the Czech sentence and the gold dependency tree t_G shown in Figure 4(a). Given the configuration $c = (\sigma, \beta, A)$ where $\sigma = [0, 1, 3, 4]$, $\beta = [5, \ldots, 13]$ and $A = \{3 \rightarrow 2\}$, we trace the two stages of the algorithm.

Preprocessing of the buffer The complete subtree rooted at node 7 satisfies the Bottom-up completeness and the Zero gap-degree conditions in Section 5.2, so the nodes $5, \ldots, 12$ in β can be replaced with the root 7. Note that all the nodes in the span $5, \ldots, 12$ have all their (gold) dependents in that span, with the exception of the root 7, with its dependent node 1 still in the stack. No other reduction is possible, and we have $\beta_R = [7, 13]$. The corresponding fragment of t_G is represented in Figure 4(b).

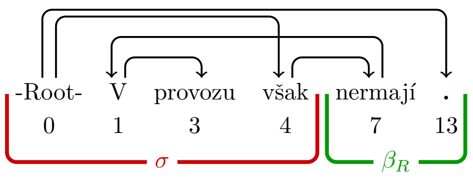
Computation of the loss Let $\gamma = \sigma\beta_R$. Algorithm 1 builds the two-dimensional array $\mathcal{T}$ in Figure 4(c). Each cell $\mathcal{T}[i,j]$ contains an association list, whose (key:value) pairs map items to their loss contribution. Figure 4(c) only shows the pairs involved in the minimum-loss computation.

Lines 1-6 of Algorithm 1 initialize the cells in the diagonal, $\mathcal{T}[0,1], \ldots, \mathcal{T}[5,6]$. The boundary between stack and buffer is $\ell = 4$, thus cells $\mathcal{T}[0,1]$, $\mathcal{T}[1,2]$, and $\mathcal{T}[2,3]$ contain only one element, while $\mathcal{T}[3,4]$, $\mathcal{T}[4,5]$ and $\mathcal{T}[5,6]$ contain as many as the previous elements in γ, although not all of them are shown in the figure.

Lines 7-12 fill the superdiagonals until $\mathcal{T}[0,6]$ is reached. The cells $\mathcal{T}[0,2]$, $\mathcal{T}[0,3]$ and $\mathcal{T}[1,3]$



(a) Non-projective dependency tree from the Prague Dependency Treebank.



(b) Fragment of dependency tree in (a) after buffer reduction.

i \ j	1	2	3	4	5	6
0	[\$,\$ 0]:0	∅	∅	…	[\$,\$ 0]:1	[\$,\$ 0]:1
1		[0,0 1]:0	∅	…	[0,0 4]:1	…
2			[1,1 3]:0	[1,1 4]:1	[1,4 7]:1	…
3				[3,3 4]:0	[3,4 7]:1	…
4					[4,4 7]:0	…
5						[0,0 13]:0

(c) Relevant portion of $\mathcal{T}$ computed by Algorithm 1, with the loss of c in the yellow entry.

Figure 4: Example of loss computation given the sentence in (a) and considering a configuration c with $\sigma = [0, 1, 3, 4]$ and $\beta = [5, \ldots, 13]$.

are left empty because $\ell = 4$. Once $\mathcal{T}[0, 6]$ is calculated, it contains only the entry with key $[\$, \$, 0]$, with the associated value 1 representing the minimum number of wrong arcs that the parsing algorithm has to build to reach a final configuration from c. Then, Line 13 retrieves the loss of the configuration, computed as the sum of $\mathcal{T}[0, 6]([\$, \$, 0])$ with the term $\mathcal{L}_c$, representing the erroneous arcs made *before* reaching c.

Note that in our example the loss of c is 1, even though $\mathcal{L}_c = 0$, meaning that there are no wrong arcs in A. Indeed, given c, there is no single computation that builds all the remaining arcs in t_G. This is reflected in $\mathcal{T}$, where the path to reach the item with minimum loss has to go through either $\mathcal{T}[3, 5]$ or $\mathcal{T}[2, 4]$, which implies building the erroneous arc $(w_7 \rightarrow w_3)$ or $(w_4 \rightarrow w_3)$, respectively.

6 Computational Analysis

The first stage of our algorithm can be easily implemented in time $\mathcal{O}(|\beta|\,|t_G|)$, where $|t_G|$ is the number of nodes in t_G, which is equal to the length n of the input string.

For the worst-case complexity of the second stage (Algorithm 1), note that the number of cell updates made by calling PROCESSCELL$(\mathcal{T}, i, k, j)$ with $k < \ell$ is $\mathcal{O}(|\sigma|^3\,|\gamma|^2\,|\beta_R|)$. This is because these updates can only be caused by procedure calls on line 10 (as those on line 12 always set $k \geq \ell$) and therefore the index k always equals $i + 1$, while h_2 must equal h_1 because the item $[h_1, h_2 h_3]$ is one of the initial items created on line 3. The variables i, h_1 and h_3 must index nodes on the stack σ as they are bounded by k, while j ranges over β_R and h_4 and h_5 can refer to nodes either on σ or on β_R.

On the other hand, the number of cell updates triggered by calls to PROCESSCELL such that $k \geq \ell$ is $\mathcal{O}(|\gamma|^4|\beta_R|^4)$, as they happen for four indices referring to nodes of β_R (k, j, h_4, h_5) and four indices that can range over σ or β_R (i, h_1, h_2, h_3).

Putting everything together, we conclude that the overall complexity of our algorithm is $\mathcal{O}(|\beta|\,|t_G| + |\sigma|^3\,|\gamma|^2\,|\beta_R| + |\gamma|^4\,|\beta_R|^4)$.

In practice, quantities $|\sigma|$, $|\beta_R|$ and $|\gamma|$ are significantly smaller than n, providing reasonable training times as we will see in Section 7. For instance, when measured on the Czech treebank, the average value of $|\sigma|$ is 7.2, with a maximum of 87. Even more interesting, the average value of $|\beta_R|$ is 2.6, with a maximum of 23. Comparing this to the average and maximum values of $|\beta|$, 11 and 192, respectively, we see that the buffer reduction is crucial in reducing training time.

Note that, when expressed as a function of n, our dynamic oracle has a worst-case time complexity of $\mathcal{O}(n^8)$. This is also the time complexity of the dynamic programming algorithm of Cohen et al. (2011) we started with, simulating all computations of our parser. In contrast, the dynamic oracle of Goldberg et al. (2014) for the projective case achieves a time complexity of $\mathcal{O}(n^3)$ from the dynamic programming parser by Kuhlmann et al. (2011) running in time $\mathcal{O}(n^5)$.

The reason why we do not achieve any asymptotic improvement is that some helpful properties that hold with projective trees are no longer satisfied in the non-projective case. In the projective (arc-standard) case, subtrees that are in the buffer can be completely reduced. As a consequence, each oracle step always combines an inferred entry in the table with either a node from the stack or a node from the reduced buffer, asymptotically reducing the time complexity. However, in the non-projective (Attardi) case, subtrees in the buffer can not always be completely reduced, for the reasons mentioned in the second-to-last paragraph of Section 5.2. As a consequence, the oracle needs to make cell updates in a more general way, which includes linking pairs of elements in the reduced buffer or pairs of inferred entries in the table.

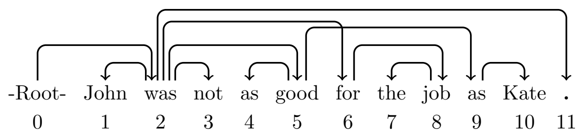


Figure 5: Non-projective dependency tree adapted from the Penn Treebank.

An example of why this is needed is provided by the gold tree in Figure 5. Assume a configuration $c = (\sigma, \beta, A)$ where $\sigma = [0, 1, 2, 3, 4]$, $\beta = [5, \ldots, 11]$, and $A = \emptyset$. It is easy to see that the loss of c is greater than zero, since the gold tree is not reachable from c: parsing the subtree rooted at node 5 requires shifting 6 into the stack, and this makes it impossible to build the arcs $2 \rightarrow 5$ and $2 \rightarrow 6$. However, if we reduced the subtree in the buffer with root 5, we would incorrectly obtain a loss of 0, as the resulting tree is parsable if we start with $\vdash_{\mathsf{sh}}$ followed by $\vdash_{\mathsf{la}}$ and $\vdash_{\mathsf{ra}_2}$. Note that there is no way of knowing whether it is safe to reduce the subtree rooted at 5 without using non-local information. For example, the arc $2 \rightarrow 6$ is crucial here: if 6 depended on 5 or 4 instead, the loss would be zero. These complications are not found in the projective case, allowing for the mentioned asymptotic improvement.

7 Experimental Evaluation

For comparability with previous work on dynamic oracles, we follow the experimental settings reported by Goldberg et al. (2014) for their arc-standard dynamic oracle. In particular, we use the same training algorithm, features, and root node position. However, we train the model for 20 iterations rather than 15, as the increased search space and spurious ambiguity of Attardi's non-projective parser implies that more iterations are required to converge to a stable model. A more detailed description of the experimental settings follows.

	static		dynamic	
	UAS	LAS	UAS	LAS
Arabic	80.90	71.56	**82.23**	**72.63**
Basque	**75.96**	**66.74**	74.32	65.59
Catalan	**90.55**	**85.20**	89.94	84.96
Chinese	84.72	79.93	**85.34**	**81.00**
Czech	79.83	72.69	**82.08**	**74.44**
English	85.52	84.46	**87.38**	**86.40**
Greek	79.84	72.26	**81.55**	**74.14**
Hungarian	**78.13**	**68.90**	76.27	68.14
Italian	83.08	78.94	**84.43**	**80.45**
Turkish	**79.57**	69.44	79.41	**70.32**
Bulgarian	**89.46**	**85.99**	89.32	85.92
Danish	85.58	81.25	**86.03**	**81.59**
Dutch	79.05	75.69	**80.13**	**77.22**
German	88.34	86.48	**88.86**	**86.94**
Japanese	93.06	91.64	**93.56**	**92.18**
Portuguese	84.80	81.38	**85.36**	**82.10**
Slovene	76.33	68.43	**78.20**	**70.22**
Spanish	79.88	76.84	**80.25**	**77.45**
Swedish	**87.26**	**82.77**	87.24	82.49
PTB	89.55	87.18	**90.47**	**88.18**

Table 1: Unlabelled Attachment Score (UAS) and Labelled Attachment Score (LAS) using a *static* and a *dynamic* oracle. Evaluation on CoNLL 2007 (first block) and CoNLL 2006 (second block) datasets is carried out including punctuation, evaluation on the Penn Treebank excludes it.

7.1 Experimental Setup

Training We train a global linear model using the averaged perceptron algorithm and a labelled version of the parser described in Section 3. We perform on-line training using the oracle defined in Section 5: at each parsing step, the model's weights are updated if the predicted transition results into an increase in configuration loss, but the process continues by following the predicted transition independently of the loss increase.

As our baseline we train the model using the static oracle defined by (Cohen et al., 2012). This oracle follows a canonical computation that creates arcs as soon as possible, and prioritizes the $\vdash_{\mathsf{la}}$ transition over the $\vdash_{\mathsf{la}_2}$ transition in situations where both create a gold arc. The static oracle is not able to deal with configurations that cannot reach the gold dependency tree, so we constrain the training algorithm to follow the zero-loss transition provided by the oracle.

While this version of Attardi's parser has been shown to cover the vast majority of non-projective sentences in several treebanks (Attardi, 2006; Cohen et al., 2012), there still are some sentences which are not parsable. These sentences are skipped during training, but not during test and evaluation of the model.

Datasets We evaluate the parser performance over CoNLL 2006 and CoNLL 2007 datasets. If a language is present in both datasets, we use the latest version. We also include results over the Penn Treebank (PTB) (Marcus et al., 1993) converted to Stanford basic dependencies (De Marneffe et al., 2006). For the CoNLL datasets we use the provided part-of-speech tags and the standard training/test partition; for the PTB we use automatically assigned tags, we train on sections 2-21 and test on section 23.

7.2 Results and Analysis

In Table 1 we report the unlabelled (UAS) and labelled (LAS) attachment scores for the static and the dynamic oracles. Each figure is an average over the accuracy provided by 5 models trained with the same setup but using a different random seed. The seed is only used to shuffle the sentences in random order during each iteration of training.

Our results are consistent with the results reported by Goldberg and Nivre (2013) and Goldberg et al. (2014). For most of the datasets, we obtain a relevant improvement in both UAS and LAS. For Dutch, Czech and German, we achieve an error reduction of 5.2%, 11.2% and 4.5%, respectively. Exceptions to this general trend are Swedish and Bulgarian, where the accuracy differences are negligible, and the Basque, Catalan and Hungarian datasets, where the performance actually decreases.

If instead of testing on the standard test sets we use 10-fold cross-validation and average the resulting accuracies, we obtain improvements for all languages in Table 1 but Basque and Hungarian. More specifically, measured (UAS, LAS) pairs for Swedish are (86.85, 82.17) with dynamic oracle against (86.6, 81.93) with static oracle; for Bulgarian (88.42, 83.91) against (88.20, 83.55); and for Catalan (88.33, 83.64) against (88.06, 83.13). This suggests that the negligible or unfavourable results in Table 1 for these languages are due to statistical variability given the small size of the test sets.

As for Basque, we measure (75.54, 67.58) against (76.77, 68.20); similarly, for Hungarian we measure (75.66, 67.66) against (77.22, 68.42). Unfortunately, we have no explanation for these performance decreases, in terms of the typology of the non-projective patterns found in these two datasets. Note that Goldberg et al. (2014) also observed a performance decrease on the Basque dataset in the projective case, although not on Hungarian.

The parsing times measured in our experiments for the static and the dynamic oracles are the same, since the oracle algorithm is only used during the training stage. Thus the reported improvements in parsing accuracy come at no extra cost for parsing time. In the training stage, the extra processing needed to compute the loss and to explore paths that do not lead to a gold tree made training about 4 times slower, on average, for the dynamic oracle model. This confirms that our oracle algorithm is fast enough to be of practical interest, in spite of its relatively high worst-case asymptotic complexity.

8 Conclusions

We have presented what, to our knowledge, are the first experimental results for a transition-based non-projective parser trained with a dynamic oracle. We have also shown significant accuracy improvements on many languages over a static oracle baseline.

The general picture that emerges from our approach is that dynamic programming algorithms originally conceived for the simulation of transition-based parsers can effectively be used in the development of polynomial-time algorithms for dynamic oracles.

Acknowledgments

The first author has been partially funded by Ministerio de Economía y Competitividad/FEDER (Grant TIN2010-18552-C03-02) and by Xunta de Galicia (Grant CN2012/008). The third author has been partially supported by MIUR under project PRIN No. 2010LYA9RH_006.

References

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

Jinho D. Choi and Andrew McCallum. 2013. Transition-based dependency parsing with selectional branching. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1052–1062, Sofia, Bulgaria, August. Association for Computational Linguistics.

Shay B. Cohen, Carlos Gómez-Rodríguez, and Giorgio Satta. 2011. Exact inference for generative probabilistic non-projective dependency parsing. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1234–1245, Edinburgh, Scotland, UK., July. Association for Computational Linguistics.

Shay B. Cohen, Carlos Gómez-Rodríguez, and Giorgio Satta. 2012. Elimination of spurious ambiguity in transition-based dependency parsing. *CoRR*, abs/1206.6735.

Marie-Catherine De Marneffe, Bill MacCartney, and Christopher D. Manning. 2006. Generating typed dependency parses from phrase structure parses. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC)*, volume 6, pages 449–454.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *Proc. of the 24th COLING*, Mumbai, India.

Yoav Goldberg and Joakim Nivre. 2013. Training deterministic parsers with non-deterministic oracles. *Transactions of the association for Computational Linguistics*, 1.

Yoav Goldberg, Francesco Sartorio, and Giorgio Satta. 2014. A tabular method for dynamic oracles in transition-based parsing. *Transactions of the Association for Computational Linguistics*, 2(April):119–130.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Liang Huang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, July.

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

Marco Kuhlmann, Carlos Gómez-Rodríguez, and Giorgio Satta. 2011. Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 673–682, Portland, Oregon, USA, June. Association for Computational Linguistics.

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

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

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In *Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57, Barcelona, Spain.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pages 195–206.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 188–193, Portland, Oregon, USA, June. Association for Computational Linguistics.