A Tabular Method for Dynamic Oracles in Transition-Based Parsing

Yoav Goldberg
Department of
Computer Science
Bar Ilan University, Israel
yoav.goldberg@gmail.com

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

We develop parsing oracles for two transition-based dependency parsers, including the arc-standard parser, solving a problem that was left open in (Goldberg and Nivre, 2013). We experimentally show that using these oracles during training yields superior parsing accuracies on many languages.

1 Introduction

Greedy transition-based dependency parsers (Nivre, 2008) incrementally process an input sentence from left to right. These parsers are very fast and provide competitive parsing accuracies (Nivre et al., 2007). However, greedy transition-based parsers still fall behind search-based parsers (Zhang and Clark, 2008; Huang and Sagae, 2010) with respect to accuracy.

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. A parsing oracle is deterministic if it returns a single canonical transition. Furthermore, an oracle is partial if it is defined only for configurations that can reach the gold tree, that is, configurations representing parsing histories with no mistake. Oracles that are both deterministic and partial are called **static**. Traditionally, only static oracles have been exploited in training of transition-based parsers.

Recently, Goldberg and Nivre (2012; 2013) showed that the accuracy of greedy parsers can be substantially improved without affecting their parsing speed. This improvement relies on the introduction of novel oracles that are nondeterministic and complete. An oracle is nondeterministic if it returns the set of all transitions that are optimal with respect to the gold tree, and it is complete if it is well-defined and correct for every configuration that is reachable by the parser. Oracles that are both nondeterministic and complete are called **dynamic**.

Goldberg and Nivre (2013) develop dynamic oracles for several transition-based parsers. The construction of these oracles is based on a property of transition-based parsers that they call arc decomposition. They also prove that the popular arc-standard system (Nivre, 2004) is not arc-decomposable, and they leave as an open research question the construction of a dynamic oracle for the arc-standard system. In this article, we develop one such oracle (§4) and prove its correctness (§5).

An extension to the arc-standard parser was presented by Sartorio et al. (2013), which relaxes the bottom-up construction order and allows mixing of bottom-up and top-down strategies. This parser, called here the LR-spine parser, achieves state-of-the-art results for greedy parsing. Like the arc-standard system, the LR-spine parser is not arc-decomposable, and a dynamic oracle for this system was not known. We extend our oracle for the arc-standard system to work for the LR-spine system as well (§6).

The dynamic oracles developed by Goldberg and Nivre (2013) for arc-decomposable systems are based on local properties of computations. In contrast, our novel dynamic oracle algorithms rely on arguably more complex structural properties of computations, which are computed through dynamic programming. This leaves open the question of whether a machine-learning model can learn to effectively simulate such complex processes: will the

Transactions of the Association for Computational Linguistics, 2 (2014) 119–130. Action Editor: Ryan McDonald.
Submitted 11/2013; Revised 2/2014; Published 4/2014.

benefit of training with the dynamic oracle carry over to the arc-standard and LR-spine systems? We show experimentally that this is indeed the case (§8), and that using the training-with-exploration method of (Goldberg and Nivre, 2013) with our dynamic programming based oracles yields superior parsing accuracies on many languages.

2 Arc-Standard Parser

In this section we introduce the arc-standard parser of Nivre (2004), which is the model that we use in this article. To keep the notation at a simple level, we only discuss the unlabeled version of the parser; however, a labeled extension is used in §8 for our experiments.

2.1 Preliminaries and Notation

The set of non-negative integers is denoted as $\mathbb{N}_0$. For $i, j \in \mathbb{N}_0$ with $i \leq j$, we write $[i, j]$ to denote the set $\{i, i+1, \ldots, j\}$. When $i > j$, $[i, j]$ denotes the empty set.

We represent an input sentence as a string $w = w_0 \cdots w_n$, $n \in \mathbb{N}_0$, where token w_0 is a special root symbol, and each w_i with $i \in [1, n]$ is a lexical token. For $i, j \in [0, n]$ with $i \leq j$, we write $w[i, j]$ to denote the substring $w_i w_{i+1} \cdots w_j$ of w.

We write $i \rightarrow j$ to denote a grammatical **dependency** of some unspecified type between lexical tokens w_i and w_j, where w_i is the head and w_j is the dependent. A **dependency tree** for w is a directed, ordered tree $t = (V_w, A)$, such that $V_w = [0, n]$ is the set of nodes, $A \subseteq V_w \times V_w$ is the set of arcs, and node 0 is the root. Arc (i, j) encodes a dependency $i \rightarrow j$, and we will often use the latter notation to denote arcs.

2.2 Transition-Based Dependency Parsing

We assume the reader is familiar with the formal framework of transition-based dependency parsing originally introduced by Nivre (2003); see Nivre (2008) for an introduction. We only summarize here our notation.

Transition-based dependency parsers use a stack data structure, where each stack element is associated with a tree spanning (generating) some substring of the input w. The parser processes the input string incrementally, from left to right, applying at each step a transition that updates the stack and/or consumes one token from the input. Transitions may also construct new dependencies, which are added to the current configuration of the parser.

We represent the **stack** data structure as an ordered sequence $\sigma = [\sigma_d, \ldots, \sigma_1]$, $d \in \mathbb{N}_0$, of nodes $\sigma_i \in V_w$, with the topmost element placed at the right. When $d = 0$, we have the empty stack $\sigma = []$. Sometimes we use the vertical bar to denote the append operator for σ, and write $\sigma = \sigma'|\sigma_1$ to indicate that σ_1 is the topmost element of σ.

The parser also uses a **buffer** to store the portion of the input string still to be processed. 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. In this way β always encodes a (non-necessarily proper) suffix of w. We denote the empty buffer as $\beta = []$. Sometimes we use the vertical bar to denote the append operator for β, and write $\beta = i|\beta'$ to indicate that i is the first token of β; consequently, we have $\beta' = [i+1, \ldots, n]$.

When processing w, the parser reaches several states, technically called configurations. 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 $A \subseteq V_w \times V_w$ is a set of arcs. The **initial** configuration for w is $([], [0, \ldots, n], \emptyset)$. For the purpose of this article, a configuration is **final** if it has the form $([0], [], A)$, and in a final configuration arc set A always defines a dependency tree for w.

The core of a transition-based parser is the set of its transitions, which are specific to each family of parsers. A **transition** is a binary relation defined over the set of configurations of the parser. We use symbol $\vdash$ to denote the union of all transition relations of a parser.

A **computation** of the parser on w is a sequence $c_0, \ldots, c_m$, $m \in \mathbb{N}_0$, of configurations (defined relative to w) such that $c_{i-1} \vdash c_i$ for each $i \in [1, m]$. We also use the reflexive and transitive closure relation $\vdash^*$ to represent computations. A computation is called **complete** whenever c_0 is initial and c_m is final. In this way, a complete computation is uniquely associated with a dependency tree for w.

2.3 Arc-Standard Parser

The arc-standard model uses the three types of transitions formally specified in Figure 1

$$(\sigma, i|\beta, A) \vdash_{\mathsf{sh}} (\sigma|i, \beta, A)$$
$$(\sigma|i|j, \beta, A) \vdash_{\mathsf{la}} (\sigma|j, \beta, A \cup \{j \to i\})$$
$$(\sigma|i|j, \beta, A) \vdash_{\mathsf{ra}} (\sigma|i, \beta, A \cup \{i \to j\})$$

Figure 1: Transitions in the arc-standard model.

- Shift (sh) removes the first node in the buffer and pushes it into the stack;
- Left-Arc (la) creates a new arc with the topmost node on the stack as the head and the second-topmost node as the dependent, and removes the second-topmost node from the stack;
- Right-Arc (ra) is symmetric to la in that it creates an arc with the second-topmost node as the head and the topmost node as the dependent, and removes the topmost node.

Notation We sometimes use the functional notation for a transition $\tau \in \{\mathsf{sh}, \mathsf{la}, \mathsf{ra}\}$, and write $\tau(c) = c'$ in place of $c \vdash_\tau c'$. Naturally, sh applies only when the buffer is not empty, and la,ra require two elements on the stack. We denote by $\mathsf{valid}(c)$ the set of valid transitions in a given configuration.

2.4 Arc Decomposition

Goldberg and Nivre (2013) show how to derive dynamic oracles for any transition-based parser which has the arc decomposition property, defined below. They also show that the arc-standard parser is not arc-decomposable.

For a configuration c, we write A_c to denote the associated set of arcs. A transition-based parser is **arc-decomposable** if, for every configuration c and for every set of arcs A that can be extended to a projective tree, we have

$$\forall (i \to j) \in A, \exists c'[c \vdash^* c' \wedge (i \to j) \in A_{c'}] \Rightarrow \exists c''[c \vdash^* c'' \wedge A \subseteq A_{c''}] \; .$$

In words, if each arc in A is individually derivable from c, then the set A in its entirety can be derived from c as well. The arc decomposition property is useful for deriving dynamic oracles because it is relatively easy to investigate derivability for single arcs and then, using this property, draw conclusions about the number of gold-arcs that are simultaneously derivable from the given configuration.

Unfortunately, the arc-standard parser is not arc-decomposable. To see why, consider a configuration with stack $\sigma = [i, j, k]$. Consider also arc set $A = \{(i, j), (i, k)\}$. The arc (i, j) can be derived through the transition sequence ra, ra, and the arc (i, k) can be derived through the alternative transition sequence la, ra. Yet, it is easy to see that a configuration containing both arcs cannot be reached.

As we cannot rely on the arc decomposition property, in order to derive a dynamic oracle for the arc-standard model we need to develop more sophisticated techniques which take into account the interaction among the applied transitions.

3 Configuration Loss and Dynamic Oracles

We aim to derive a dynamic oracle for the arc-standard (and related) system. This is a function that takes a configuration c and a gold tree t_G and returns a set of transitions that are "optimal" for c with respect to t_G. As already mentioned in the introduction, a dynamic oracle can be used to improve training of greedy transition-based parsers. In this section we provide a formal definition for a dynamic oracle.

Let t_1 and t_2 be two dependency trees over the same string w, with arc sets A_1 and A_2, respectively. We define the **loss** of t_1 with respect to t_2 as

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

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 our parser relative to input string w. We write $\mathcal{D}(c)$ to denote the set of all dependency trees that can be obtained in a computation of the form $c \vdash^* c_f$, where c_f is some final configuration. We extend the loss function in (1) to configurations by letting

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

Assume some reference (desired) dependency tree t_G for w, which we call the **gold** tree. Quantity $\mathcal{L}(c, t_G)$ can be used to compute a dynamic oracle relating a parser configuration c to a set of optimal actions by setting

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

We therefore need to develop an algorithm for computing (2). We will do this first for the arc-standard parser, and then for an extension of this model.

Notation We also apply the loss function $\mathcal{L}(t, t_G)$ in (1) when t is a dependency tree for a substring of w. In this case the nodes of t are a subset of the nodes of t_G, and $\mathcal{L}(t, t_G)$ provides a count of the nodes of t that are assigned a wrong head node, when t_G is considered as the reference tree.

4 Main Algorithm

Throughout this section we assume an arc-standard parser. Our algorithm takes as input a projective gold tree t_G and a configuration $c = (\sigma_L, \beta, A)$. We call σ_L the **left stack**, in contrast with a right stack whose construction is specified below.

4.1 Basic Idea

The algorithm consists of two steps. Informally, in the first step we compute the largest subtrees, called here tree fragments, of the gold tree t_G that have their span entirely included in the buffer β. The root nodes of these tree fragments are then arranged into a stack data structure, according to the order in which they appear in β and with the leftmost root in β being the topmost element of the stack. We call this structure the right stack σ_R. Intuitively, σ_R can be viewed as the result of pre-computing β by applying all sequences of transitions that match t_G and that can be performed independently of the stack in the input configuration c, that is, σ_L.

In the second step of the algorithm we use dynamic programming techniques to simulate all computations of the arc-standard parser starting in a configuration with stack σ_L and with a buffer consisting of σ_R, with the topmost token of σ_R being the first token of the buffer. As we will see later, the search space defined by these computations includes the dependency trees for w that are reachable from the input configuration c and that have minimum loss. We then perform a Viterbi search to pick up such value.

The second step is very similar to standard implementations of the CKY parser for context-free grammars (Hopcroft and Ullman, 1979), running on an input string obtained as the concatenation of σ_L and σ_R. The main difference is that we restrict ourselves to parse only those constituents in $\sigma_L\sigma_R$ that dominate the topmost element of σ_L (the rightmost element, if σ_L is viewed as a string). In this way, we account for the additional constraint that we visit only those configurations of the arc-standard parser that can be reached from the input configuration c. For instance, this excludes the reduction of two nodes in σ_L that are not at the two topmost positions. This would also exclude the reduction of two nodes in σ_R: this is correct, since the associated tree fragments have been chosen as the largest such fragments in β.

The above intuitive explanation will be made mathematically precise in §5, where the notion of linear dependency tree is introduced.

4.2 Construction of the Right Stack

In the first step we process β and construct a stack σ_R, which we call the **right stack** associated with c and t_G. Each node of σ_R is the root of a tree t which satisfies the following properties

- t is a tree fragment of the gold tree t_G having span entirely included in the buffer β;
- t is **bottom-up complete** for t_G, meaning that for each node i of t different from t's root, the dependents of i in t_G cannot be in σ_L;
- t is **maximal** for t_G, meaning that every supertree of t in t_G violates the above conditions.

The stack σ_R is incrementally constructed by processig β from left to right. Each node i is copied into σ_R if it satisfies any of the following conditions

- the parent node of i in t_G is not in β;
- some dependent of i in t_G is in σ_L or has already been inserted in σ_R.

It is not difficult to see that the nodes in σ_R are the roots of tree fragments of t_G that satisfy the condition of bottom-up completeness and the condition of maximality defined above.

4.3 Computation of Configuration Loss

We start with some notation. Let $\ell_L = |\sigma_L|$ and $\ell_R = |\sigma_R|$. We write $\sigma_L[i]$ to denote the i-th element of σ_L and $t(\sigma_L[i])$ to denote the corresponding tree fragment; $\sigma_R[i]$ and $t(\sigma_R[i])$ have a similar meaning. In order to simplify the specification of the algorithm, we assume below that $\sigma_L[1] = \sigma_R[1]$.

Algorithm 1 Computation of the loss function for the arc-standard parser

1: $\mathcal{T}[1,1](\sigma_L[1]) \leftarrow \mathcal{L}(t(\sigma_L[1]), t_G)$
2: **for** $d \leftarrow 1$ **to** $\ell_L + \ell_R - 1$ **do** ▷ d is the index of a sub-anti-diagonal
3: **for** $j \leftarrow \max\{1, d - \ell_L + 1\}$ **to** $\min\{d, \ell_R\}$ **do** ▷ j is the column index
4: $i \leftarrow d - j + 1$ ▷ i is the row index
5: **if** $i < \ell_L$ **then** ▷ expand to the left
6: **for each** $h \in \Delta_{i,j}$ **do**
7: $\mathcal{T}[i+1,j](h) \leftarrow \min\{\mathcal{T}[i+1,j](h),\ \mathcal{T}[i,j](h) + \delta_G(h \rightarrow \sigma_L[i+1])\}$
8: $\mathcal{T}[i+1,j](\sigma_L[i+1]) \leftarrow \min\{\mathcal{T}[i+1,j](\sigma_L[i+1]),\ \mathcal{T}[i,j](h) + \delta_G(\sigma_L[i+1] \rightarrow h)\}$
9: **if** $j < \ell_R$ **then** ▷ expand to the right
10: **for each** $h \in \Delta_{i,j}$ **do**
11: $\mathcal{T}[i,j+1](h) \leftarrow \min\{\mathcal{T}[i,j+1](h),\ \mathcal{T}[i,j](h) + \delta_G(h \rightarrow \sigma_R[j+1])\}$
12: $\mathcal{T}[i,j+1](\sigma_R[j+1]) \leftarrow \min\{\mathcal{T}[i,j+1](\sigma_R[j+1]), \mathcal{T}[i,j](h) + \delta_G(\sigma_R[j+1] \rightarrow h)\}$
13: **return** $\mathcal{T}[\ell_L, \ell_R](0) + \sum_{i \in [1,\ell_L]} \mathcal{L}(t(\sigma_L[i]), t_G)$

Therefore the elements of σ_R which have been constructed in §4.2 are $\sigma_R[i]$, $i \in [2, \ell_R]$.

Algorithm 1 uses a two-dimensional array $\mathcal{T}$ of size $\ell_L \times \ell_R$, where each entry $\mathcal{T}[i,j]$ is an association list from integers to integers. An entry $\mathcal{T}[i,j](h)$ stores the minimum loss among dependency trees rooted at h that can be obtained by running the parser on the first i elements of stack σ_L and the first j elements of buffer σ_R. More precisely, let

$$\Delta_{i,j} = \{\sigma_L[k] \mid k \in [1,i]\} \cup \{\sigma_R[k] \mid k \in [1,j]\}\ . \quad (4)$$

For each $h \in \Delta_{i,j}$, the entry $\mathcal{T}[i,j](h)$ is the minimum loss among all dependency trees defined as above and with root h. We also assume that $\mathcal{T}[i,j](h)$ is initialized to $+\infty$ (not reported in the algorithm).

Algorithm 1 starts at the top-left corner of $\mathcal{T}$, visiting each individual sub-anti-diagonal of $\mathcal{T}$ in ascending order, and eventually reaching the bottom-right corner of the array. For each entry $\mathcal{T}[i,j]$, the left expansion is considered (lines 5 to 8) by combining with tree fragment $\sigma_L[i+1]$, through a left or a right arc reduction. This results in the update of $\mathcal{T}[i+1,j](h)$, for each $h \in \Delta_{i+1,j}$, whenever a smaller value of the loss is achieved for a tree with root h. The Kronecker-like function used at line 8 provides the contribution of each single arc to the loss of the current tree. Denoting with A_G the set of arcs of t_G, such a function is defined as

$$\delta_G(i \rightarrow j) = \begin{cases} 0, & \text{if } (i \rightarrow j) \in A_G; \\ 1, & \text{otherwise.} \end{cases} \quad (5)$$

A symmetrical process is implemented for the right expansion of $\mathcal{T}[i,j]$ through tree fragment $\sigma_R[j+1]$ (lines 9 to 12).

As we will see in the next section, quantity $\mathcal{T}[\ell_L, \ell_R](0)$ is the minimal loss of a tree composed only by arcs that connect nodes in σ_L and σ_R. By summing the loss of all tree fragments $t(\sigma_L[i])$ to the loss in $\mathcal{T}[\ell_L, \ell_R](0)$, at line 13, we obtain the desired result, since the loss of each tree fragment $t(\sigma_R[j])$ is zero.

5 Formal Properties

Throughout this section we let w, t_G, σ_L, σ_R and $c = (\sigma_L, \beta, A)$ be defined as in §4, but we no longer assume that $\sigma_L[1] = \sigma_R[1]$. To simplify the presentation, we sometimes identify the tokens in w with the associated nodes in a dependency tree for w.

5.1 Linear Trees

Algorithm 1 explores all dependency trees that can be reached by an arc-standard parser from configuration c, under the condition that (i) the nodes in the buffer β are pre-computed into tree fragments and collapsed into their root nodes in the right stack σ_R, and (ii) nodes in σ_R cannot be combined together prior to their combination with other nodes in the left stack σ_L. This set of dependency trees is char-

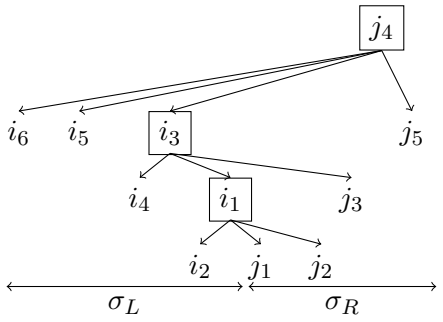

Figure 2: A possible linear tree for string pair (σ_L, σ_R), where $\sigma_L = i_6 i_5 i_4 i_3 i_2 i_1$ and $\sigma_R = j_1 j_2 j_3 j_4 j_5$. The spine of the tree consists of nodes j_4, i_3 and i_1.

acterized here using the notion of linear tree, to be used later in the correctness proof.

Consider two nodes $\sigma_L[i]$ and $\sigma_L[j]$ with $j > i > 1$. An arc-standard parser can construct an arc between $\sigma_L[i]$ and $\sigma_L[j]$, in any direction, only after reaching a configuration in which $\sigma_L[i]$ is at the top of the stack and $\sigma_L[j]$ is at the second topmost position. In such configuration we have that $\sigma_L[i]$ dominates $\sigma_L[1]$. Furthermore, consider nodes $\sigma_R[i]$ and $\sigma_R[j]$ with $j > i \geq 1$. Since we are assuming that tree fragments $t(\sigma_R[i])$ and $t(\sigma_R[j])$ are bottom-up complete and maximal, as defined in §4.2, we allow the construction of an arc between $\sigma_R[i]$ and $\sigma_R[j]$, in any direction, only after reaching a configuration in which $\sigma_R[i]$ dominates node $\sigma_L[1]$.

The dependency trees satisfying the restrictions above are captured by the following definition. A **linear tree** over (σ_L, σ_R) is a projective dependency tree t for string $\sigma_L \sigma_R$ satisfying both of the additional conditions reported below. The path from t's root to node $\sigma_L[1]$ is called the **spine** of t.

- Every node of t not in the spine is a dependent of some node in the spine.
- For each arc $i \rightarrow j$ in t with j in the spine, no dependent of i can be placed in between i and j within string $\sigma_L \sigma_R$.

An example of a linear tree is depicted in Figure 2. Observe that the second condition above forbids the reduction of two nodes i and j, in case none of these dominates node $\sigma_L[1]$. For instance, the ra reduction of nodes i_3 and i_2 would result in arc $i_3 \rightarrow i_2$ replacing arc $i_1 \rightarrow i_2$ in Figure 2. The new dependency tree is not linear, because of a violation of the second condition above. Similarly, the la reduction of nodes j_3 and j_4 would result in arc $j_4 \rightarrow j_3$ replacing arc $i_3 \rightarrow j_3$ in Figure 2, again a violation of the second condition above.

Lemma 1 *Any tree $t \in \mathcal{D}(c)$ can be decomposed into trees $t(\sigma_L[i])$, $i \in [1, \ell_L]$, trees t_j, $j \in [1, q]$ and $q \geq 1$, and a linear tree t_l over $(\sigma_L, \sigma_{R,t})$, where $\sigma_{R,t} = r_1 \cdots r_q$ and each r_j is the root node of t_j.* □

PROOF (SKETCH) Trees $t(\sigma_L[i])$ are common to every tree in $\mathcal{D}(c)$, since the arc-standard model can not undo the arcs already built in the current configuration c. Similar to the construction in §4.2 of the right stack σ_R from t_G, we let t_j, $j \in [1, q]$, be tree fragments of t that cover only nodes associated with the tokens in the buffer β and that are bottom-up complete and maximal for t. These trees are indexed according to their left to right order in β. Finally, t_l is implicitly defined by all arcs of t that are not in trees $t(\sigma_L[i])$ and t_j. It is not difficult to see that t_l has a spine ending with node $\sigma_L[1]$ and is a linear tree over $(\sigma_L, \sigma_{R,t})$. ■

5.2 Correctness

Our proof of correctness for Algorithm 1 is based on a specific dependency tree t^* for w, which we define below. Let $S_L = \{\sigma_L[i] \mid i \in [1, \ell_L]\}$ and let D_L be the set of nodes that are descendants of some node in S_L. Similarly, let $S_R = \{\sigma_R[i] \mid i \in [1, \ell_R]\}$ and let D_R be the set of descendants of nodes in S_R. Note that sets S_L, S_R, D_L and D_R provide a partition of V_w.

We choose any linear tree t_l^* over (σ_L, σ_R) having root 0, such that $\mathcal{L}(t_l^*, t_G) = \min_t \mathcal{L}(t, t_G)$, where t ranges over all possible linear trees over (σ_L, σ_R) with root 0. Tree t^* consists of the set of nodes V_w and the set of arcs obtained as the union of the set of arcs of t_l^* and the set of arcs of all trees $t(\sigma_L[i])$, $i \in [1, \ell_L]$, and $t(\sigma_R[j])$, $j \in [1, \ell_R]$.

Lemma 2 $t^* \in \mathcal{D}(c)$. □

PROOF (SKETCH) All tree fragments $t(\sigma_L[i])$ have already been parsed and are available in the stack associated with c. Each tree fragment $t(\sigma_R[j])$ can later be constructed in the computation, when a configuration c' is reached with the relevant segment of w at the start of the buffer. Note also that parsing of $t(\sigma_R[j])$ can be done in a way that does not depend on the content of the stack in c'.

Finally, the parsing of the tree fragments $t(\sigma_R[j])$ is interleaved with the construction of the arcs from the linear tree t_l^*, which are all of the form $(i \rightarrow j)$ with $i, j \in (S_L \cup S_R)$. More precisely, if $(i \rightarrow j)$ is an arc from t_l^*, at some point in the computation nodes i and j will become available at the two topmost positions in the stack. This follows from the second condition in the definition of linear tree. ■

We now show that tree t^* is "optimal" within the set $\mathcal{D}(c)$ and with respect to t_G.

Lemma 3 $\mathcal{L}(t^*, t_G) = \mathcal{L}(c, t_G)$. □

PROOF Consider an arbitrary tree $t \in \mathcal{D}(c)$. Assume the decomposition of t defined in the proof of Lemma 1, through trees $t(\sigma_L[i])$, $i \in [1, \ell_L]$, trees t_j, $j \in [1, q]$, and linear tree t_l over $(\sigma_L, \sigma_{R,t})$.

Recall that an arc $i \rightarrow j$ denotes an ordered pair (i, j). Let us consider the following partition for the set of arcs of any dependency tree for w

$$A_1 = (S_L \cup D_L) \times D_L \ ,$$
$$A_2 = (S_R \cup D_R) \times D_R \ ,$$
$$A_3 = (V_w \times V_w) \setminus (A_1 \cup A_2) \ .$$

In what follows, we compare the losses $\mathcal{L}(t, t_G)$ and $\mathcal{L}(t^*, t_G)$ by separately looking into the contribution to such quantities due to the arcs in A_1, A_2 and A_3.

Note that the arcs of trees $t(\sigma_L[i])$ are all in A_1, the arcs of trees $t(\sigma_R[j])$ are all in A_2, and the arcs of tree t_l^* are all in A_3. Since t and t^* share trees $t(\sigma_L[i])$, when restricted to arcs in A_1 quantities $\mathcal{L}(t, t_G)$ and $\mathcal{L}(t^*, t_G)$ are the same. When restricted to arcs in A_2, quantity $\mathcal{L}(t^*, t_G)$ is zero, by construction of the trees $t(\sigma_R[j])$. Thus $\mathcal{L}(t, t_G)$ can not be smaller than $\mathcal{L}(t^*, t_G)$ for these arcs. The difficult part is the comparison of the contribution to $\mathcal{L}(t, t_G)$ and $\mathcal{L}(t^*, t_G)$ due to the arcs in A_3. We deal with this below.

Let $A_{S,G}$ be the set of all arcs from t_G that are also in set $(S_L \times S_R) \cup (S_R \times S_L)$. In words, $A_{S,G}$ represents gold arcs connecting nodes in S_L and nodes in S_R, in any direction. Within tree t, these arcs can only be found in the t_l component, since nodes in S_L are all placed within the spine of t_l, or else at the left of that spine.

Let us consider an arc $(j \rightarrow i) \in A_{S,G}$ with $j \in S_L$ and $i \in S_R$, and let us assume that $(j \rightarrow i)$ is in t_l^*. If token a_i does not occur in $\sigma_{R,t}$, node i is not in t_l and $(j \rightarrow i)$ can not be an arc of t. We then have that $(j \rightarrow i)$ contributes one unit to $\mathcal{L}(t, t_G)$ but does not contribute to $\mathcal{L}(t^*, t_G)$. Similarly, let $(i \rightarrow j) \in A_{S,G}$ be such that $i \in S_R$ and $j \in S_L$, and assume that $(i \rightarrow j)$ is in t_l^*. If token a_i does not occur in $\sigma_{R,t}$, arc $(i \rightarrow j)$ can not be in t. We then have that $(i \rightarrow j)$ contributes one unit to $\mathcal{L}(t, t_G)$ but does not contribute to $\mathcal{L}(t^*, t_G)$.

Intuitively, the above observations mean that the winning strategy for trees in $\mathcal{D}(c)$ is to move nodes from S_R as much as possible into the linear tree component t_l, in order to make it possible for these nodes to connect to nodes in S_L, in any direction. In this case, arcs from A_3 will also move into the linear tree component of a tree in $\mathcal{D}(c)$, as it happens in the case of t^*. We thus conclude that, when restricted to the set of arcs in A_3, quantity $\mathcal{L}(t, t_G)$ is not smaller than $\mathcal{L}(t^*, t_G)$, because stack σ_R has at least as many tokens corresponding to nodes in S_R as stack $\sigma_{R,t}$, and because t_l^* has the minimum loss among all the linear trees over (σ_L, σ_R).

Putting all of the above observations together, we conclude that $\mathcal{L}(t, t_G)$ can not be smaller than $\mathcal{L}(t^*, t_G)$. This concludes the proof, since t has been arbitrarily chosen in $\mathcal{D}(c)$. ■

Theorem 1 *Algorithm 1 computes* $\mathcal{L}(c, t_G)$. □

PROOF (SKETCH) Algorithm 1 implements a Viterbi search for trees with smallest loss among all linear trees over (σ_L, σ_R). Thus $\mathcal{T}[\ell_L, \ell_R](0) = \mathcal{L}(t_l^*, t_G)$. The loss of the tree fragments $t(\sigma_R[j])$ is 0 and the loss of the tree fragments $t(\sigma_L[i])$ is added at line 13 in the algorithm. Thus the algorithm returns $\mathcal{L}(t^*, t_G)$, and the statement follows from Lemma 2 and Lemma 3. ■

5.3 Computational Analysis

Following §4.2, the right stack σ_R can be easily constructed in time $\mathcal{O}(n)$, n the length of the input string. We now analyze Algorithm 1. For each entry $\mathcal{T}[i, j]$ and for each $h \in \Delta_{i,j}$, we update $\mathcal{T}[i, j](h)$ a number of times bounded by a constant which does not depend on the input. Each updating can be computed in constant time as well. We thus conclude that Algorithm 1 runs in time $\mathcal{O}(\ell_L \cdot \ell_R \cdot (\ell_L + \ell_R))$. Quantity $\ell_L + \ell_R$ is bounded by n, but in practice the former is significantly smaller. When measured over the sentences in the Penn

Treebank, the average value of $\frac{\ell_L+\ell_R}{n}$ is 0.29. In terms of runtime, training is 2.3 times slower when using our oracle instead of a static oracle.

6 Extension to the LR-Spine Parser

In this section we consider the transition-based parser proposed by Sartorio et al. (2013), called here the LR-spine parser. This parser is not arc-decomposable: the same example reported in §2.4 can be used to show this fact. We therefore extend to the LR-spine parser the algorithm developed in §4.

6.1 The LR-Spine Parser

Let t be a dependency tree. The **left spine** of t is an ordered sequence $\langle i_1, \ldots, i_p \rangle$, $p \geq 1$, consisting of all nodes in a descending path from the root of t taking the leftmost child node at each step. The **right spine** of t is defined symmetrically. We use $\oplus$ to denote sequence concatenation.

In the LR-spine parser each stack element $\sigma[i]$ denotes a partially built subtree $t(\sigma[i])$ and is represented by a pair (ls_i, rs_i), with ls_i and rs_i the left and the right spine, respectively, of $t(\sigma[i])$. We write $ls_i[k]$ ($rs_i[k]$) to represent the k-th element of ls_i (rs_i, respectively). We also write $r(\sigma[i])$ to denote the root of $t(\sigma[i])$, so that $r(\sigma[i]) = ls_i[1] = rs_i[1]$.

Informally, the LR-spine parser uses the same transition typologies as the arc-standard parser. However, an arc $(h \rightarrow d)$ can now be created with the head node h chosen from any node in the spine of the involved tree. The transition types of the LR-spine parser are defined as follows.

- Shift (sh) removes the first node from the buffer and pushes into the stack a new element, consisting of the left and right spines of the associated tree

$$(\sigma, i|\beta, A) \vdash_{\mathsf{sh}} (\sigma|(\langle i \rangle, \langle i \rangle), \beta, A) \ .$$

- Left-Arc k (la_k) creates a new arc $h \rightarrow d$ from the k-th node in the left spine of the topmost tree in the stack to the head of the second element in the stack. Furthermore, the two topmost stack elements are replaced by a new element associated with the resulting tree

$$(\sigma'|\sigma[2]|\sigma[1], \beta, A) \vdash_{\mathsf{la}_k} (\sigma'|\sigma_{\mathsf{la}_k}, \beta, A \cup \{h \rightarrow d\})$$

  where we have set $h = ls_1[k]$, $d = r(\sigma[2])$ and $\sigma_{\mathsf{la}_k} = (\langle ls_1[1], \ldots, ls_1[k] \rangle \oplus ls_2, rs_1)$.

- Right-Arc k (ra_k for short) is defined symmetrically with respect to la_k

$$(\sigma'|\sigma[2]|\sigma[1], \beta, A) \vdash_{\mathsf{ra}_k} (\sigma'|\sigma_{\mathsf{ra}_k}, \beta, A \cup \{h \rightarrow d\})$$

  where we have set $h = rs_2[k]$, $d = r(\sigma[1])$ and $\sigma_{\mathsf{ra}_k} = (ls_2, \langle rs_2[1], \ldots, rs_2[k] \rangle \oplus rs_1)$.

Note that, at each configuration in the LR-spine parser, there are $|ls_1|$ possible la_k transitions, one for each choice of a node in the left spine of $t(\sigma[1])$; similarly, there are $|rs_2|$ possible ra_k transitions, one for each choice of a node in the right spine of $t(\sigma[2])$.

6.2 Configuration Loss

We only provide an informal description of the extended algorithm here, since it is very similar to the algorithm in §4.

In the first phase we use the procedure of §4.2 for the construction of the right stack σ_R, considering only the roots of elements in σ_L and ignoring the rest of the spines. The only difference is that each element $\sigma_R[j]$ is now a pair of spines $(ls_{R,j}, rs_{R,j})$. Since tree fragment $t(\sigma_R[j])$ is bottom-up complete (see §4.1), we now restrict the search space in such a way that only the root node $r(\sigma_R[j])$ can take dependents. This is done by setting $ls_{R,j} = rs_{R,j} = \langle r(\sigma_R[j]) \rangle$ for each $j \in [1, \ell_R]$. In order to simplify the presentation we also assume $\sigma_R[1] = \sigma_L[1]$, as done in §4.3.

In the second phase we compute the loss of an input configuration using a two-dimensional array $\mathcal{T}$, defined as in §4.3. However, because of the way transitions are defined in the LR-spine parser, we now need to distinguish tree fragments not only on the basis of their roots, but also on the basis of their left and right spines. Accordingly, we define each entry $\mathcal{T}[i, j]$ as an association list with keys of the form (ls, rs). More specifically, $\mathcal{T}[i, j](ls, rs)$ is the minimum loss of a tree with left and right spines *ls* and *rs*, respectively, that can be obtained by running the parser on the first i elements of stack σ_L and the first j elements of buffer σ_R.

We follow the main idea of Algorithm 1 and expand each tree in $\mathcal{T}[i, j]$ at its left side, by combining with tree fragment $t(\sigma_L[i+1])$, and at its right side, by combining with tree fragment $t(\sigma_R[j+1])$.

Tree combination deserves some more detailed discussion, reported below.

We consider the combination of a tree t_a from $\mathcal{T}[i,j]$ and tree $t(\sigma_L[i+1])$ by means of a left-arc transition. All other cases are treated symmetrically. Let (ls_a, rs_a) be the spine pair of t_a, so that the loss of t_a is stored in $\mathcal{T}[i,j](ls_a, rs_a)$. Let also (ls_b, rs_b) be the spine pair of $t(\sigma_L[i+1])$. In case there exists a gold arc in t_G connecting a node from ls_a to $r(\sigma_L[i+1])$, we choose the transition la_k, $k \in [1, |ls_a|]$, that creates such arc. In case such gold arc does not exists, we choose the transition la_k with the maximum possible value of k, that is, $k = |ls_a|$. We therefore explore only one of the several possible ways of combining these two trees by means of a left-arc transition.

We remark that the above strategy is safe. In fact, in case the gold arc exists, no other gold arc can ever involve the nodes of ls_a eliminated by la_k (see the definition in §6.1), because arcs can not cross each other. In case the gold arc does not exist, our choice of $k = |ls_a|$ guarantees that we do not eliminate any element from ls_a.

Once a transition la_k is chosen, as described above, the reduction is performed and the spine pair (ls, rs) for the resulting tree is computed from (ls_a, rs_a) and (ls_b, rs_b), as defined in §6.1. At the same time, the loss of the resulting tree is computed, on the basis of the loss $\mathcal{T}[i,j](ls_a, rs_a)$, the loss of tree $t(\sigma_L[i+1])$, and a Kronecker-like function defined below. This loss is then used to update $\mathcal{T}[i+1,j](ls, rs)$.

Let t_a and t_b be two trees that must be combined in such a way that t_b becomes the dependent of some node in one of the two spines of t_a. Let also $p_a = (ls_a, rs_a)$ and $p_b = (ls_b, rs_b)$ be spine pairs for t_a and t_b, respectively. Recall that A_G is the set of arcs of t_G. The new Kronecker-like function for the computation of the loss is defined as

$$\delta_G(p_a, p_b) = \begin{cases} 0, & \text{if } r(t_a) < r(t_b) \wedge \\ & \quad \exists k[(rs_a^k \rightarrow r(t_b)) \in A_G]; \\ 0, & \text{if } r(t_a) > r(t_b) \wedge \\ & \quad \exists k[(ls_a^k \rightarrow r(t_b)) \in A_G]; \\ 1, & \text{otherwise.} \end{cases}$$

6.3 Efficiency Improvement

The algorithm in §6.2 has an exponential behaviour. To see why, consider trees in $\mathcal{T}[i,j]$. These trees are produced by the combination of trees in $\mathcal{T}[i-1,j]$ with tree $t(\sigma_L[i])$, or by the combination of trees in $\mathcal{T}[i,j-1]$ with tree $t(\sigma_R[j])$. Since each combination involves either a left-arc or a right-arc transition, we obtain a recursive relation that resolves into a number of trees in $\mathcal{T}[i,j]$ bounded by 4^{i+j-2}.

We introduce now two restrictions to the search space of our extended algorithm that result in a huge computational saving. For a spine s, we write $\mathcal{N}(s)$ to denote the set of all nodes in s. We also let $\Delta_{i,j}$ be the set of all pairs (ls, rs) such that $\mathcal{T}[i,j](ls, rs) \neq +\infty$.

- Every time a new pair (ls, rs) is created in $\Delta[i,j]$, we remove from ls all nodes different from the root that do not have gold dependents in $\{r(\sigma_L[k]) \mid k < i\}$, and we remove from rs all nodes different from the root that do not have gold dependents in $\{r(\sigma_R[k]) \mid k > j\}$.

- A pair $p_a = (ls_a, rs_a)$ is removed from $\Delta[i,j]$ if there exists a pair $p_b = (ls_b, rs_b)$ in $\Delta[i,j]$ with the same root node as p_a and with $(ls_a, rs_a) \neq (ls_b, rs_b)$, such that $\mathcal{N}(ls_a) \subseteq \mathcal{N}(ls_b)$, $\mathcal{N}(rs_a) \subseteq \mathcal{N}(rs_b)$, and $\mathcal{T}[i,j](p_a) \geq \mathcal{T}[i,j](p_b)$.

The first restriction above reduces the size of a spine by eliminating a node if it is irrelevant for the computation of the loss of the associated tree. The second restriction eliminates a tree t_a if there is a tree t_b with smaller loss than t_a, such that in the computations of the parser t_b provides exactly the same context as t_a. It is not difficult to see that the above restrictions do not affect the correctness of the algorithm, since they always leave in our search space some tree that has optimal loss.

A mathematical analysis of the computational complexity of the extended algorithm is quite involved. In Figure 3, we plot the worst case size of $\mathcal{T}[i,j]$ for each value of $j + i - 1$, computed over all configurations visited in the training phase (see §7). We see that $|\mathcal{T}[i,j]|$ grows linearly with $j + i - 1$, leading to the same space requirements of Algorithm 1. Empirically, training with the dynamic

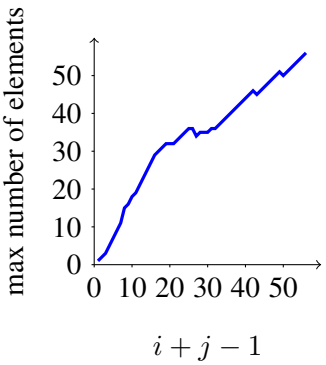

Figure 3: Empirical worst case size of $\mathcal{T}[i, j]$ for each value of $i + j - 1$ as measured on the Penn Treebank corpus.

Algorithm 2 Online training for greedy transition-based parsers

$\mathbf{w} \leftarrow 0$
for k iterations **do**
  shuffle(corpus)
  for sentence w and gold tree t_G in corpus **do**
    $c \leftarrow \text{INITIAL}(w)$
    while not $\text{FINAL}(c)$ **do**
      $\tau_p \leftarrow \text{argmax}_{\tau \in \mathsf{valid}(c)}\ \mathbf{w} \cdot \phi(c, \tau)$
      $\tau_o \leftarrow \text{argmax}_{\tau \in \mathsf{oracle}(c, t_G)}\ \mathbf{w} \cdot \phi(c, \tau)$
      if $\tau_p \notin \mathsf{oracle}(c, t_G)$ **then**
        $\mathbf{w} \leftarrow \mathbf{w} + \phi(c, \tau_o) - \phi(c, \tau_p)$
      $\tau \leftarrow \begin{cases} \tau_p & \text{if EXPLORE} \\ \tau_o & \text{otherwise} \end{cases}$
      $c \leftarrow \tau(c)$
return *averaged*$(\mathbf{w})$

oracle is only about 8 times slower than training with the oracle of Sartorio et al. (2013) without exploring incorrect configurations.

7 Training

We follow the training procedure suggested by Goldberg and Nivre (2013), as described in Algorithm 2. The algorithm performs online learning using the averaged perceptron algorithm. A weight vector $\mathbf{w}$ (initialized to 0) is used to score the valid transitions in each configuration based on a feature representation ϕ, and the highest scoring transition τ_p is predicted. If the predicted transition is not optimal according to the oracle, the weights $\mathbf{w}$ are updated away from the predicted transition and towards the highest scoring oracle transition τ_o. The parser then moves to the next configuration, by taking either the predicted or the oracle transition. In the "error exploration" mode (EXPLORE is true), the parser follows the predicted transition, and otherwise the parser follows the oracle transition. Note that the error exploration mode requires the completeness property of a dynamic oracle.

We consider three training conditions: *static*, in which the oracle is deterministic (returning a single canonical transition for each configuration) and no error exploration is performed; *nondet*, in which we use a nondeterministic partial oracle (Sartorio et al., 2013), but do not perform error exploration; and *explore* in which we use the dynamic oracle and perform error exploration. The *static* setup mirrors the way greedy parsers are traditionally trained. The *nondet* setup allows the training procedure to choose which transition to take in case of spurious ambiguities. The *explore* setup increases the configuration space explored by the parser during training, by exposing the training procedure to non-optimal configurations that are likely to occur during parsing, together with the optimal transitions to take in these configurations. It was shown by Goldberg and Nivre (2012; 2013) that the *nondet* setup outperforms the *static* setup, and that the *explore* setup outperforms the *nondet* setup.

8 Experimental Evaluation

Datasets Performance evaluation is carried out on CoNLL 2007 multilingual dataset, as well as on the Penn Treebank (PTB) (Marcus et al., 1993) converted to Stanford basic dependencies (De Marneffe et al., 2006). For the CoNLL datasets we use gold part-of-speech tags, while for the PTB we use automatically assigned tags. As usual, the PTB parser is trained on sections 2-21 and tested on section 23.

Setup We train labeled versions of the arc-standard (std) and LR-spine (lrs) parsers under the *static*, *nondet* and *explore* setups, as defined in §7. In the *nondet* setup we use a nondeterministic partial oracle and in the *explore* setup we use the nondeterministic complete oracles we present in this paper. In the *static* setup we resolve oracle ambiguities and choose a canonic transition sequence by attaching arcs as soon as possible. In the *explore* setup,

parser:train	Arabic	Basque	Catalan	Chinese	Czech	English	Greek	Hungarian	Italian	Turkish	PTB
	UAS										
std:static	81.39	75.37	90.32	85.17	78.90	85.69	79.90	77.67	82.98	77.04	89.86
std:nondet	81.33	74.82	90.75	84.80	79.92	86.89	81.19	77.51	84.15	76.85	90.56
std:explore	82.56	74.39	90.95	85.65	81.01	87.70	81.85	78.72	84.37	77.21	90.92
lrs:static	81.67	76.07	91.47	84.24	77.93	86.36	79.43	76.56	84.64	77.00	90.33
lrs:nondet	83.14	75.53	91.31	84.98	80.03	88.38	81.12	76.98	85.29	77.63	91.18
lrs:explore	84.54	75.82	91.92	86.72	81.19	89.37	81.78	77.48	85.38	78.61	91.77
	LAS										
std:static	71.93	65.64	84.90	80.35	71.39	84.60	72.25	67.66	78.77	65.90	87.56
std:nondet	71.09	65.28	85.36	80.06	71.47	85.91	73.40	67.77	80.06	65.81	88.30
std:explore	72.89	65.27	85.82	81.28	72.92	86.79	74.22	69.57	80.25	66.71	88.72
lrs:static	72.24	66.21	86.02	79.36	70.48	85.38	72.36	66.79	80.38	66.02	88.07
lrs:nondet	72.94	65.66	86.03	80.47	71.32	87.45	73.09	67.70	81.32	67.02	88.96
lrs:explore	74.54	66.91	86.83	82.38	72.72	88.44	74.04	68.76	81.50	68.06	89.53

Table 1: Scores on the CoNLL 2007 dataset (including punctuation, gold parts of speech) and on Penn Tree Bank (excluding punctuation, predicted parts of speech). Label 'std' refers to the arc-standard parser, and 'lrs' refers to the LR-spine parser. Each number is an average over 5 runs with different randomization seeds.

from the first round of training onward, we always follow the predicted transition (EXPLORE is true). For all languages, we deal with non-projectivity by skipping the non-projective sentences during training but not during test. For each parsing system, we use the same feature templates across all languages.[1] The arc-standard models are trained for 15 iterations and the LR-spine models for 30 iterations, after which all the models seem to have converged.

Results In Table 1 we report the labeled (LAS) and unlabeled (UAS) attachment scores. As expected, the LR-spine parsers outperform the arc-standard parsers trained under the same setup. Training with the dynamic oracles is also beneficial: despite the arguable complexity of our proposed oracles, the trends are consistent with those reported by Goldberg and Nivre (2012; 2013). For the arc-standard model we observe that the move from a static to a nondeterministic oracle during training improves the accuracy for most of languages. Making use of the completeness of the dynamic oracle and moving to the error exploring setup further improve results. The only exceptions are Basque, that has a small dataset with more than 20% of non-projective sentences, and Chinese. For Chinese we observe a reduction of accuracy in the *nondet* setup, but an increase in the *explore* setup.

For the LR-spine parser we observe a practically constant increase of performance by moving from the static to the nondeterministic and then to the error exploring setups.

[1] Our complete code, together with the description of the feature templates, is available on the second author's homepage.

9 Conclusions

We presented dynamic oracles, based on dynamic programming, for the arc-standard and the LR-spine parsers. Empirical evaluation on 10 languages showed that, despite the apparent complexity of the oracle calculation procedure, the oracles are still learnable, in the sense that using these oracles in the error exploration training algorithm presented in (Goldberg and Nivre, 2012) considerably improves the accuracy of the trained parsers.

Our algorithm computes a dynamic oracle using dynamic programming to explore a forest of dependency trees that can be reached from a given parser configuration. For the arc-standard parser, the computation takes cubic time in the size of the largest of the left and right input stacks. Dynamic programming for the simulation of arc-standard parsers have been proposed by Kuhlmann et al. (2011). That algorithm could be adapted to compute minimum loss for a given configuration, but the running time is $\mathcal{O}(n^4)$, n the size of the input string: besides being asymptotically slower by one order of magnitude, in practice n is also larger than the stack size above.

Acknowledgments We wish to thank the anonymous reviewers. In particular, we are indebted to one of them for two important technical remarks. The third author has been partially supported by MIUR under project PRIN No. 2010LYA9RH_006.

References

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.

John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA.

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, 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, Johan Hall, Sandra Kübler, Ryan McDonald, Jens Nilsson, Sebastian Riedel, and Deniz Yuret. 2007. The CoNLL 2007 shared task on dependency parsing. In *Proceedings of EMNLP-CoNLL*.

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.

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

Francesco Sartorio, Giorgio Satta, and Joakim Nivre. 2013. A transition-based dependency parser using a dynamic parsing strategy. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 135–144, Sofia, Bulgaria, August. Association for Computational Linguistics.

Yue Zhang and Stephen Clark. 2008. A tale of two parsers: Investigating and combining graph-based and transition-based dependency parsing. In *Proceedings of EMNLP*.