A Transition-Based Dependency Parser Using a Dynamic Parsing Strategy

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

Joakim Nivre
Department of
Linguistics and Philology
Uppsala University, Sweden
joakim.nivre@lingfil.uu.se

Abstract

We present a novel transition-based, greedy dependency parser which implements a flexible mix of bottom-up and top-down strategies. The new strategy allows the parser to postpone difficult decisions until the relevant information becomes available. The novel parser has a $\sim$12% error reduction in unlabeled attachment score over an arc-eager parser, with a slow-down factor of 2.8.

1 Introduction

Dependency-based methods for syntactic parsing have become increasingly popular during the last decade or so. This development is probably due to many factors, such as the increased availability of dependency treebanks and the perceived usefulness of dependency structures as an interface to downstream applications, but a very important reason is also the high efficiency offered by dependency parsers, enabling web-scale parsing with high throughput. The most efficient parsers are greedy transition-based parsers, which only explore a single derivation for each input and relies on a locally trained classifier for predicting the next parser action given a compact representation of the derivation history, as pioneered by Yamada and Matsumoto (2003), Nivre (2003), Attardi (2006), and others. However, while these parsers are capable of processing tens of thousands of tokens per second with the right choice of classifiers, they are also known to perform slightly below the state-of-the-art because of search errors and subsequent error propagation (McDonald and Nivre, 2007), and recent research on transition-based dependency parsing has therefore explored different ways of improving their accuracy.

The most common approach is to use beam search instead of greedy decoding, in combination with a globally trained model that tries to minimize the loss over the entire sentence instead of a locally trained classifier that tries to maximize the accuracy of single decisions (given no previous errors), as first proposed by Zhang and Clark (2008). With these methods, transition-based parsers have reached state-of-the-art accuracy for a number of languages (Zhang and Nivre, 2011; Bohnet and Nivre, 2012). However, the drawback with this approach is that parsing speed is proportional to the size of the beam, which means that the most accurate transition-based parsers are not nearly as fast as the original greedy transition-based parsers. Another line of research tries to retain the efficiency of greedy classifier-based parsing by instead improving the way in which classifiers are learned from data. While the classical approach limits training data to parser states that result from oracle predictions (derived from a treebank), these novel approaches allow the classifier to explore states that result from its own (sometimes erroneous) predictions (Choi and Palmer, 2011; Goldberg and Nivre, 2012).

In this paper, we explore an orthogonal approach to improving the accuracy of transition-based parsers, without sacrificing their advantage in efficiency, by introducing a new type of transition system. While all previous transition systems assume a static parsing strategy with respect to top-down and bottom-up processing, our new system allows a dynamic strategy for ordering parsing decisions. This has the advantage that the parser can postpone difficult decisions until the relevant information becomes available, in a way that is not possible in existing transition systems. A second advantage of dynamic parsing is that we can extend the feature inventory of previous systems. Our experiments show that these advantages lead to significant improvements in parsing accuracy, compared to a baseline parser that uses the arc-eager transition system of Nivre (2003), which is one of the most

Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, pages 135–144,
Sofia, Bulgaria, August 4-9 2013. ©2013 Association for Computational Linguistics

widely used static transition systems.

2 Static vs. Dynamic Parsing

The notions of bottom-up and top-down parsing strategies do not have a general mathematical definition; they are instead specified, often only informally, for individual families of grammar formalisms. In the context of dependency parsing, a parsing strategy is called purely **bottom-up** if every dependency $h \rightarrow d$ is constructed only after all dependencies of the form $d \rightarrow i$ have been constructed. Here $h \rightarrow d$ denotes a dependency with h the head node and d the dependent node. In contrast, a parsing strategy is called purely **top-down** if $h \rightarrow d$ is constructed before any dependency of the form $d \rightarrow i$.

If we consider transition-based dependency parsing (Nivre, 2008), the purely bottom-up strategy is implemented by the arc-standard model of Nivre (2004). After building a dependency $h \rightarrow d$, this model immediately removes from its stack node d, preventing further attachment of dependents to this node. A second popular parser, the arc-eager model of Nivre (2003), instead adopts a mixed strategy. In this model, a dependency $h \rightarrow d$ is constructed using a purely bottom-up strategy if it represents a left-arc, that is, if the dependent d is placed to the left of the head h in the input string. In contrast, if $h \rightarrow d$ represents a right-arc (defined symmetrically), then this dependency is constructed before any right-arc $d \rightarrow i$ (top-down) but after any left-arc $d \rightarrow i$ (bottom-up).

What is important to notice about the above transition-based parsers is that the adopted parsing strategies are **static**. By this we mean that each dependency is constructed according to some *fixed* criterion, depending on structural conditions such as the fact that the dependency represents a left or a right arc. This should be contrasted with **dynamic** parsing strategies in which several parsing options are *simultaneously* available for the dependencies being constructed.

In the context of left-to-right, transition-based parsers, dynamic strategies are attractive for several reasons. One argument is related to the well-known PP-attachment problem, illustrated in Figure 1. Here we have to choose whether to attach node P as a dependent of V (arc α_2) or else as a dependent of N1 (arc α_3). The purely bottom-up arc-standard model has to take a decision as soon as N1 is placed into the stack. This is so

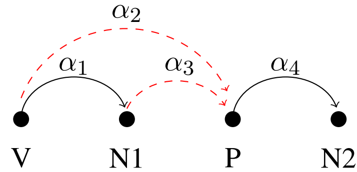

Figure 1: PP-attachment example, with dashed arcs identifying two alternative choices.

because the construction of α_1 excludes α_3 from the search space, while the alternative decision of shifting P into the stack excludes α_2. This is bad, because the information about the correct attachment could come from the lexical content of node P. The arc-eager model performs slightly better, since it can delay the decision up to the point in which α_1 has been constructed and P is read from the buffer. However, at this point it must make a commitment and either construct α_3 or pop N1 from the stack (implicitly committing to α_2) before N2 is read from the buffer. In contrast with this scenario, in the next sections we implement a dynamic parsing strategy that allows a transition system to decide between the attachments α_2 and α_3 after it has seen all of the four nodes V, N1, P and N2.

Other additional advantages of dynamic parsing strategies with respect to static strategies are related to the increase in the feature inventory that we apply to parser states, and to the increase of spurious ambiguity. However, these arguments are more technical than the PP-attachment argument above, and will be discussed later.

3 Dependency Parser

In this section we present a novel transition-based parser for projective dependency trees, implementing a dynamic parsing strategy.

3.1 Preliminaries

For non-negative integers i and j with $i \leq j$, we write $[i, j]$ to denote the set $\{i, i+1, \ldots, j\}$. When $i > j$, $[i, j]$ is the empty set.

We represent an input sentence as a string $w = w_0 \cdots w_n$, $n \geq 1$, where token w_0 is a special root symbol and, for each $i \in [1, n]$, token $w_i = (i, a_i, t_i)$ encodes a lexical element a_i and a part-of-speech tag t_i associated with the i-th word in the sentence.

A **dependency tree** for w is a directed, ordered tree $T_w = (V_w, A_w)$, where $V_w = \{w_i \mid i \in$

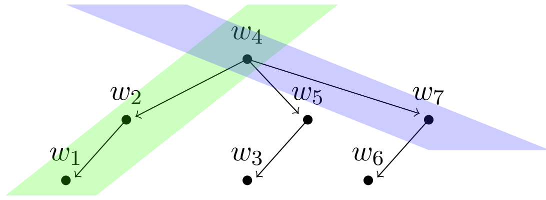

Figure 2: A dependency tree with left spine $\langle w_4, w_2, w_1\rangle$ and right spine $\langle w_4, w_7\rangle$.

$[0, n]\}$ is the set of nodes, and $A_w \subseteq V_w \times V_w$ is the set of arcs. Arc (w_i, w_j) encodes a dependency $w_i \rightarrow w_j$. A sample dependency tree (excluding w_0) is displayed in Figure 2. If $(w_i, w_j) \in A_w$ for $j < i$, we say that w_j is a left child of w_i; a right child is defined in a symmetrical way.

The **left spine** of T_w is an ordered sequence $\langle u_1, \dots, u_p\rangle$ with $p \geq 1$ and $u_i \in V_w$ for $i \in [1, p]$, consisting of all nodes in a descending path from the root of T_w taking the leftmost child node at each step. More formally, u_1 is the root node of T_w and u_i is the leftmost child of u_{i-1}, for $i \in [2, p]$. The **right spine** of T_w is defined symmetrically; see again Figure 2. Note that the left and the right spines share the root node and no other node.

3.2 Basic Idea

Transition-based dependency parsers use a stack data structure, where each stack element is associated with a tree spanning some (contiguous) substring of the input w. The parser can combine two trees T and T' through attachment operations, called left-arc or right-arc, under the condition that T and T' appear at the two topmost positions in the stack. Crucially, only the roots of T and T' are available for attachment; see Figure 3(a).

In contrast, a stack element in our parser records the *entire* left spine and right spine of the associated tree. This allows us to extend the inventory of the attachment operations of the parser by including the attachment of tree T as a dependent of any node in the left or in the right spine of a second tree T', provided that this does not violate projectivity.[1] See Figure 3(b) for an example.

The new parser implements a mix of bottom-up and top-down strategies, since after any of the attachments in Figure 3(b) is performed, additional dependencies can still be created for the root of T. Furthermore, the new parsing strategy is clearly dynamic, due to the free choice in the timing for these attachments. The new strategy is more powerful than the strategy of the arc-eager model, since we can use top-down parsing at left arcs, which is not allowed in arc-eager parsing, and we do not have the restrictions of parsing right arcs ($h \rightarrow d$) before the attachment of right dependents at node d.

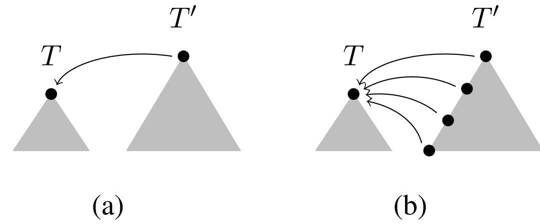

Figure 3: Left-arc attachment of T to T' in case of (a) standard transition-based parsers and (b) our parser.

To conclude this section, let us resume our discussion of the PP-attachment example in Figure 1. We observe that the new parsing strategy allows the construction of a tree T' consisting of the only dependency V $\rightarrow$ N1 and a tree T, placed at the right of T', consisting of the only dependency P $\rightarrow$ N2. Since the right spine of T' consists of nodes V and N1, we can freely choose between attachment V $\rightarrow$ P and attachment N1 $\rightarrow$ P. Note that this is done after we have seen node N2, as desired.

3.3 Transition-based Parser

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. To keep the notation at a simple level, we only discuss here the unlabeled version of our parser; however, a labeled extension is used in §5 for our experiments.

Our transition-based parser uses a **stack** data structure to store partial parses for the input string w. We represent the stack as an ordered sequence $\sigma = [\sigma_d, \dots, \sigma_1]$, $d \geq 0$, of stack elements, 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.

A **stack element** is a pair

$$\sigma_k = (\langle u_{k,1}, \dots, u_{k,p}\rangle, \langle v_{k,1}, \dots, v_{k,q}\rangle)$$

where the ordered sequences $\langle u_{k,1}, \dots, u_{k,p}\rangle$ and

[1] A dependency tree for w is projective if every subtree has a contiguous yield in w.

$\langle v_{k,1}, \ldots, v_{k,q} \rangle$ are the left and the right spines, respectively, of the tree associated with σ_k. Recall that $u_{k,1} = v_{k,1}$, since the root node of the associated tree is shared by the two spines.

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 $\beta = [w_i, \ldots, w_n]$, $i \geq 0$, of tokens from w, with the first element placed at the left. Note that β always represents a (non-necessarily proper) suffix of w. When $i > n$, we have the empty buffer $\beta = []$. Sometimes we use the vertical bar to denote the append operator for β, and write $\beta = w_i|\beta'$ to indicate that w_i is the first token of β; consequently, we have $\beta' = [w_{i+1}, \ldots, w_n]$.

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 $([], [w_0, \ldots, w_n], \emptyset)$. The set of terminal configurations consists of all configurations of the form $([\sigma_1], [], A)$, where σ_1 is associated with a tree having root w_0, that is, $u_{1,1} = v_{1,1} = w_0$, and A is any set of arcs.

The core of a transition-based parser is the set of its transitions. Each **transition** is a binary relation defined over the set of configurations of the parser. Since the set of configurations is infinite, a transition is infinite as well, when viewed as a set. However, transitions can always be specified by some finite means. Our parser uses three types of transitions, defined in what follows.

- SHIFT, or sh for short. This transition 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. More formally

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

- LEFT-ARC$_k$, $k \geq 1$, or la_k for short. Let h be the k-th node in the left spine of the topmost tree in the stack, and let d be the root node of the second topmost tree in the stack. This transition creates a new arc $h \rightarrow d$. Furthermore, the two topmost stack elements are replaced by a new element associated with the tree resulting from the $h \rightarrow d$ attachment. The transition does not advance with the reading of the buffer. More formally

$$(\sigma'|\sigma_2|\sigma_1, \beta, A) \vdash_{\mathsf{la}_k} (\sigma'|\sigma_{\mathsf{la}}, \beta, A \cup \{h \rightarrow d\})$$

where

$$\begin{aligned} \sigma_1 &= (\langle u_{1,1}, \ldots, u_{1,p} \rangle, \langle v_{1,1}, \ldots, v_{1,q} \rangle) \,, \\ \sigma_2 &= (\langle u_{2,1}, \ldots, u_{2,r} \rangle, \langle v_{2,1}, \ldots, v_{2,s} \rangle) \,, \\ \sigma_{\mathsf{la}} &= (\langle u_{1,1}, \ldots, u_{1,k}, u_{2,1}, \ldots, u_{2,r} \rangle, \\ &\qquad \langle v_{1,1}, \ldots, v_{1,q} \rangle) \,, \end{aligned}$$

and where we have set $h = u_{1,k}$ and $d = u_{2,1}$.

- RIGHT-ARC$_k$, $k \geq 1$, or ra_k for short. This transition is defined symmetrically with respect to la_k. We have

$$(\sigma'|\sigma_2|\sigma_1, \beta, A) \vdash_{\mathsf{ra}_k} (\sigma'|\sigma_{\mathsf{ra}}, \beta, A \cup \{h \rightarrow d\})$$

where σ_1 and σ_2 are as in the la_k case,

$$\begin{aligned} \sigma_{\mathsf{ra}} = (&\langle u_{2,1}, \ldots, u_{2,r} \rangle, \\ &\langle v_{2,1}, \ldots, v_{2,k}, v_{1,1}, \ldots, v_{1,q} \rangle) \,, \end{aligned}$$

and we have set $h = v_{2,k}$ and $d = v_{1,1}$.

Transitions la_k and ra_k are parametric in k, where k is bounded by the length of the input string and not by a fixed constant (but see also the experimental findings in §5). Thus our system uses an unbounded number of transition relations, which has an apparent disadvantage for learning algorithms. We will get back to this problem in §4.3.

A **complete computation** relative to w is a sequence of configurations $c_1, c_2, \ldots, c_t$, $t \geq 1$, such that c_1 and c_t are initial and final configurations, respectively, and for each $i \in [2, t]$, c_i is produced by the application of some transition to c_{i-1}. It is not difficult to see that the transition-based parser specified above is sound, meaning that the set of arcs constructed in any complete computation on w is always a dependency tree for w. The parser is also complete, meaning that every (projective) dependency tree for w is constructed by some complete computation on w. A mathematical proof of this statement is beyond the scope of this paper, and will not be provided here.

3.4 Deterministic Parsing Algorithm

The transition-based parser of the previous section is a nondeterministic device, since several transitions can be applied to a given configuration. This might result in several complete computations

Algorithm 1 Parsing Algorithm

Input: string $w = w_0 \cdots w_n$, function $\mathsf{score}()$
Output: dependency tree T_w

```
c = (σ, β, A) ← ([], [w_0, ..., w_n], ∅)
while |σ| > 1 ∨ |β| > 0 do
    while |σ| < 2 do
        update c with sh
    p ← length of left spine of σ_1
    s ← length of right spine of σ_2
    T ← {la_k | k ∈ [1, p]} ∪
        {ra_k | k ∈ [1, s]} ∪ {sh}
    bestT ← argmax_{t∈T} score(t, c)
    update c with bestT
return T_w = (V_w, A)
```

Algorithm 2 Learning Algorithm

Input: pair $(w = w_0 \cdots w_n, A_g)$, vector $\vec{\omega}$
Output: vector $\vec{\omega}$

```
c = (σ, β, A) ← ([], [w_0, ..., w_n], ∅)
while |σ| > 1 ∨ |β| > 0 do
    while |σ| < 2 do
        update c with SHIFT
    p ← length of left spine of σ_1
    s ← length of right spine of σ_2
    T ← {la_k | k ∈ [1, p]} ∪
        {ra_k | k ∈ [1, s]} ∪ {sh}
    bestT ← argmax_{t∈T} score(t, c)
    bestCorrectT ←
        argmax_{t∈T ∧ isCorrect(t)} score(t, c)
    if bestT ≠ bestCorrectT then
        ω ← ω − φ(bestT, c)
              + φ(bestCorrectT, c)
    update c with bestCorrectT
```

for w. We present here an algorithm that runs the parser in pseudo-deterministic mode, greedily choosing at each configuration the transition that maximizes some score function. Algorithm 1 takes as input a string w and a scoring function $\mathsf{score}()$ defined over parser transitions and parser configurations. The scoring function will be the subject of §4 and is not discussed here. The output of the parser is a dependency tree for w.

At each iteration the algorithm checks whether there are at least two elements in the stack and, if this is not the case, it shifts elements from the buffer to the stack. Then the algorithm uses the function $\mathsf{score}()$ to evaluate all transitions that can be applied under the current configuration $c = (\sigma, \beta, A)$, and it applies the transition with the highest score, updating the current configuration.

To parse a sentence of length n (excluding the root token w_0) the algorithm applies exactly $2n + 1$ transitions. In the worst case, each transition application involves $1 + p + s$ transition evaluations. We therefore conclude that the algorithm always reaches a configuration with an empty buffer and a stack which contains only one element. Then the algorithm stops, returning the dependency tree whose arc set is defined as in the current configuration.

4 Model and Training

In this section we introduce the adopted learning algorithm and discuss the model parameters.

4.1 Learning Algorithm

We use a linear model for the score function in Algorithm 1, and define $\mathsf{score}(t, c) = \vec{\omega} \cdot \phi(t, c)$. Here $\vec{\omega}$ is a weight vector and function ϕ provides a feature vector representation for a transition t applying to a configuration c. The function ϕ will be discussed at length in §4.3. The vector $\vec{\omega}$ is trained using the perceptron algorithm in combination with the averaging method to avoid overfitting; see Freund and Schapire (1999) and Collins and Duffy (2002) for details.

The training data set consists of pairs (w, A_g), where w is a sentence and A_g is the set of arcs of the gold (desired) dependency tree for w. At training time, each pair (w, A_g) is processed using the learning algorithm described as Algorithm 2. The algorithm is based on the notions of correct and incorrect transitions, discussed at length in §4.2.

Algorithm 2 parses w following Algorithm 1 and using the current $\vec{\omega}$, until the highest score selected transition $bestT$ is incorrect according to A_g. When this happens, $\vec{\omega}$ is updated by decreasing the weights of the features associated with the incorrect $bestT$ and by increasing the weights of the features associated with the transition $bestCorrectT$ having the highest score among all possible correct transitions. After each update, the learning algorithm resumes parsing from the current configuration by applying $bestCorrectT$, and moves on using the updated weights.

4.2 Correct and Incorrect Transitions

Standard transition-based dependency parsers are trained by associating each gold tree with a canonical complete computation. This means that, for each configuration of interest, only one transition

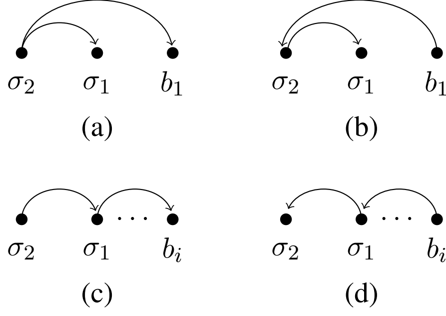

Figure 4: Graphical representation of configurations; drawn arcs are in A_g but have not yet been added to the configuration. Transition sh is incorrect for configuration (a) and (b); sh and ra_1 are correct for (c); sh and la_1 are correct for (d).

leading to the gold tree is considered as correct. In this paper we depart from such a methodology, and follow Goldberg and Nivre (2012) in allowing more than one correct transition for each configuration, as explained in detail below.

Let (w, A_g) be a pair in the training set. In §3.3 we have mentioned that there is always a complete computation on w that results in the construction of the set A_g. In general, there might be more than one computation for A_g. This means that the parser shows **spurious ambiguity**.

Observe that all complete computations for A_g share the same initial configuration $c_{I,w}$ and final configuration c_{F,A_g}. Consider now the set $\mathcal{C}(w)$ of all configurations c that are reachable from $c_{I,w}$, meaning that there exists a sequence of transitions that takes the parser from $c_{I,w}$ to c. A configuration $c \in \mathcal{C}(w)$ is **correct** for A_g if c_{F,A_g} is reachable from c; otherwise, c is **incorrect** for A_g.

Let $c \in \mathcal{C}(w)$ be a correct configuration for A_g. A transition t is correct for c and A_g if $c \vdash_t c'$ and c' is correct for A_g; otherwise, t is incorrect for c and A_g. The next lemma provides a characterization of correct and incorrect transitions; see Figure 4 for examples. We use this characterization in the implementation of predicate $isCorrect()$ in Algorithm 2.

Lemma 1 *Let (w, A_g) be a pair in the training set and let $c \in \mathcal{C}(w)$ with $c = (\sigma, \beta, A)$ be a correct configuration for A_g. Let also $v_{1,k}$, $k \in [1, q]$, be the nodes in the right spine of σ_1.*

(i) la_k *and* ra_k *are incorrect for c and A_g if and only if they create a new arc $(h \rightarrow d) \notin A_g$;*

(ii) sh *is incorrect for c and A_g if and only if the following conditions are both satisfied:*

(a) *there exists an arc $(h \rightarrow d)$ in A_g such that h is in σ and $d = v_{1,1}$;*

(b) *there is no arc $(h' \rightarrow d')$ in A_g with $h' = v_{1,k}$, $k \in [1, q]$, and d' in β.* □

PROOF (SKETCH) To prove part (i) we focus on transition ra_k; a similar argument applies to la_k. The 'if' statement in part (i) is self-evident.

'Only if'. Assuming that transition ra_k creates a new arc $(h \rightarrow d) \in A_g$, we argue that from configuration c' with $c \vdash_{\mathsf{ra}_k} c'$ we can still reach the final configuration associated with A_g. We have $h = v_{2,k}$ and $d = u_{1,1}$. The tree fragments in σ with roots $v_{2,k+1}$ and $u_{1,1}$ must be adjacent siblings in the tree associated with A_g, since c is a correct configuration for A_g and $(v_{2,k} \rightarrow u_{1,1}) \in A_g$. This means that each of the nodes $v_{2,k+1}, \ldots, v_{2,s}$ in the right spine in σ_2 in c must have already acquired all of its right dependents, since the tree is projective. Therefore it is safe for transition ra_k to eliminate the nodes $v_{2,k+1}, \ldots, v_{2,s}$ from the right spine in σ_2.

We now deal with part (ii). Let $c \vdash_{\mathsf{sh}} c'$, $c' = (\sigma', \beta', A)$.

'If'. Assuming (ii)a and (ii)b, we argue that c' is incorrect. Node d is the head of σ'_2. Arc $(h \rightarrow d)$ is not in A, and the only way we could create $(h \rightarrow d)$ from c' is by reaching a new configuration with d in the topmost stack symbol, which amounts to say that σ'_1 can be reduced by a correct transition. Node h is in some σ'_i, $i > 2$, by (ii)a. Then reduction of σ'_1 implies that the root of σ'_1 is reachable from the root of σ'_2, which contradicts (ii)b.

'Only if'. Assuming (ii)a is not satisfied, we argue that sh is correct for c and A_g. There must be an arc $(h \rightarrow d)$ not in A with $d = v_{1,1}$ and h is some token w_i in β. From stack $\sigma' = \sigma''|\sigma'_2|\sigma'_1$ it is always possible to construct $(h \rightarrow d)$ consuming the substring of β up to w_i and ending up with stack $\sigma''|\sigma_{red}$, where σ_{red} is a stack element with root w_i. From there, the parser can move on to the final configuration c_{F,A_g}. A similar argument applies if we assume that (ii)b is not satisfied. ■

From condition (i) in Lemma 1 and from the fact that there are no cycles in A_g, it follows that there is at most one correct transition among the transitions of type la_k or ra_k. From condition (ii) in the lemma we can also see that the existence of a correct transition of type la_k or ra_k for some configuration does not imply that the sh transition is incorrect

for the same configuration; see Figures 4(c,d) for examples. It follows that for a correct configuration there might be at most 2 correct transitions. In our training experiments for English in §5 we observe 2 correct transitions for 42% of the reached configurations. This nondeterminism is a byproduct of the adopted dynamic parsing strategy, and eventually leads to the spurious ambiguity of the parser.

As already mentioned, we do not impose any canonical form on complete computations that would hardwire a preference for some correct transition and get rid of spurious ambiguity. Following Goldberg and Nivre (2012), we instead regard spurious ambiguity as an additional resource of our parsing strategy. Our main goal is that the training algorithm learns to prefer a sh transition in a configuration that does not provide enough information for the choice of the correct arc. In the context of dependency parsing, the strategy of delaying arc construction when the current configuration is not informative is called the *easy-first* strategy, and has been first explored by Goldberg and Elhadad (2010).

4.3 Feature Extraction

In existing transition-based parsers a set of atomic features is statically defined and extracted from each configuration. These features are then combined together into complex features, according to some feature template, and joined with the available transition types. This is not possible in our system, since the number of transitions la_k and ra_k is not bounded by a constant. Furthermore, it is not meaningful to associate transitions la_k and ra_k, for any $k \geq 1$, always with the same features, since the constructed arcs impinge on nodes at different depths in the involved spines. It seems indeed more significant to extract information that is local to the arc $h \rightarrow d$ being constructed by each transition, such as for instance the grandparent and the great grandparent nodes of d. This is possible if we introduce a higher level of abstraction than in existing transition-based parsers. We remark here that this abstraction also makes the feature representation more similar to the ones typically found in graph-based parsers, which are centered on arcs or subgraphs of the dependency tree.

We index the nodes in the stack σ relative to the head node of the arc being constructed, in case of the transitions la_k or ra_k, or else relative to the root node of σ_1, in case of the transition sh. More precisely, let $c = (\sigma, \beta, A)$ be a configuration and let t be a transition. We define the **context** of c and t as the tuple $C(c,t) = (s_3, s_2, s_1, q_1, q_2, gp, gg)$, whose components are placeholders for word tokens in σ or in β. All these placeholders are specified in Table 1, for each c and t. Figure 5 shows an example of feature extraction for the displayed configuration $c = (\sigma, \beta, A)$ and the transition la_2. In this case we have $s_3 = u_{3,1}$, $s_2 = u_{2,1}$, $s_1 = u_{1,2}$, $q_1 = gp = u_{1,1}$, $q_2 = b_1$; $gg = \mathsf{none}$ because the head of gp is not available in c.

Note that in Table 1 placeholders are dynamically assigned in such a way that s_1 and s_2 refer to the nodes in the constructed arc $h \rightarrow d$, and gp, gg refer to the grandparent and the great grandparent nodes, respectively, of d. Furthermore, the node assigned to s_3 is the parent node of s_2, if such a node is defined; otherwise, the node assigned to s_3 is the root of the tree fragment in the stack underneath σ_2. Symmetrically, placeholders q_1 and q_2 refer to the parent and grandparent nodes of s_1, respectively, when these nodes are defined; otherwise, these placeholders get assigned tokens from the buffer. See again Figure 5.

Finally, from the placeholders in $C(c,t)$ we extract a standard set of atomic features and their complex combinations, to define the function ϕ. Our feature template is an extended version of the feature template of Zhang and Nivre (2011), originally developed for the arc-eager model. The extension is obtained by adding top-down features for left-arcs (based on placeholders gp and gg), and by adding right child features for the first stack element. The latter group of features is usually exploited for the arc-standard model, but is undefined for the arc-eager model.

5 Experimental Assessment

Performance evaluation is carried out on the Penn Treebank (Marcus et al., 1993) converted to Stanford basic dependencies (De Marneffe et al., 2006). We use sections 2-21 for training, 22 as development set, and 23 as test set. The part-of-speech tags are assigned by an automatic tagger with accuracy 97.1%. The tagger used on the training set is trained on the same data set by using four-way jackknifing, while the tagger used on the development and test sets is trained on all the training set. We train an arc-labeled version of our parser.

In the first three lines of Table 2 we compare

context placeholder	sh	la_k $k=1$	$k=2$	$k>2$	ra_k $k=1$	$k=2$	$k>2$
s_1	$u_{1,1}=v_{1,1}$	$u_{1,k}$			$u_{1,1}=v_{1,1}$		
s_2	$u_{2,1}=v_{2,1}$	$u_{2,1}=v_{2,1}$			$v_{2,k}$		
s_3	$u_{3,1}=v_{3,1}$	$u_{3,1}=v_{3,1}$			$u_{3,1}=v_{3,1}$	$v_{2,k-1}$	
q_1	b_1	b_1	$u_{1,k-1}$		b_1		
q_2	b_2	b_2	b_1	$u_{1,k-2}$	b_2		
gp	none	none	$u_{1,k-1}$		none	$v_{2,k-1}$	
gg	none	none	none	$u_{1,k-2}$	none	none	$v_{2,k-2}$

Table 1: Definition of $C(c,t) = (s_3, s_2, s_1, q_1, q_2, gp, gg)$, for $c = (\sigma'|\sigma_3|\sigma_2|\sigma_1, b_1|b_2|\beta, A)$ and t of type sh or la_k, ra_k, $k \geq 1$. Symbols $u_{j,k}$ and $v_{j,k}$ are the k-th nodes in the left and right spines, respectively, of stack element σ_j, with $u_{j,1} = v_{j,1}$ being the shared root of σ_j; none is an artificial element used when some context's placeholder is not available.

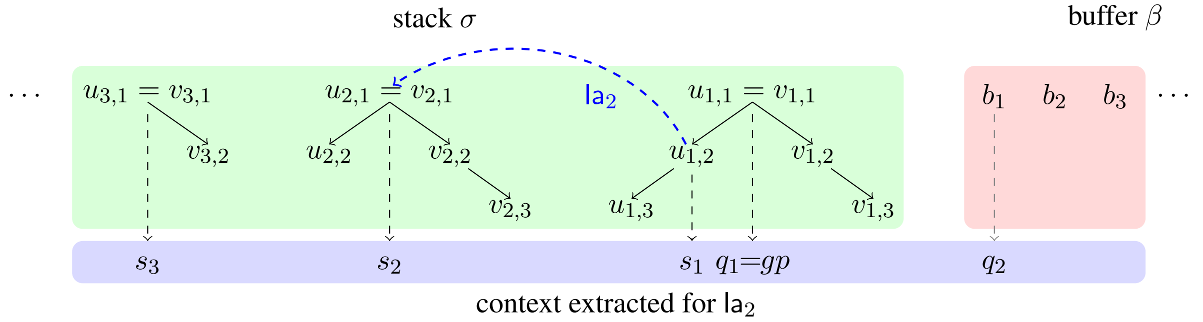

Figure 5: Extraction of atomic features for context $C(c, \mathsf{la}_2) = (s_3, s_2, s_1, q_1, q_2, gp, gg)$, $c = (\sigma, \beta, A)$.

parser	iter	UAS	LAS	UEM
arc-standard	23	90.02	87.69	38.33
arc-eager	12	90.18	87.83	40.02
this work	30	91.33	89.16	42.38
arc-standard + easy-first	21	90.49	88.22	39.61
arc-standard + spine	27	90.44	88.23	40.27

Table 2: Accuracy on test set, excluding punctuation, for unlabeled attachment score (UAS), labeled attachment score (LAS), unlabeled exact match (UEM).

the accuracy of our parser against our implementation of the arc-eager and arc-standard parsers. For the arc-eager parser, we use the feature template of Zhang and Nivre (2011). The same template is adapted to the arc-standard parser, by removing the top-down parent features and by adding the right child features for the first stack element. It turns out that our feature template, described in §4.3, is the exact merge of the templates used for the arc-eager and the arc-standard parsers.

We train all parsers up to 30 iterations, and for each parser we select the weight vector $\vec{\omega}$ from the iteration with the best accuracy on the development set. All our parsers attach the root node at the end of the parsing process, following the 'None' approach discussed by Ballesteros and Nivre (2013). Punctuation is excluded in all evaluation metrics. Considering UAS, our parser provides an improvement of 1.15 over the arc-eager parser and an improvement of 1.31 over the arc-standard parser, that is an error reduction of $\sim$12% and $\sim$13%, respectively. Considering LAS, we achieve improvements of 1.33 and 1.47, with an error reduction of $\sim$11% and $\sim$12%, over the arc-eager and the arc-standard parsers, respectively.

We speculate that the observed improvement of our parser can be ascribed to two distinct components. The first component is the left-/right-spine representation for stack elements, introduced in §3.3. The second component is the easy-first strategy, implemented on the basis of the spurious ambiguity of our parser and the definition of correct/incorrect transitions in §4.2. In this perspective, we observe that our parser can indeed be viewed as an arc-standard model *augmented* with (i) the spine representation, and (ii) the easy-first strategy. More specifically, (i) generalizes the la/ra transitions to the la_k/ra_k transitions, introducing a top-down component into the purely bottom-up arc-standard. On the other hand, (ii) drops the limitation of canonical computations for the arc-standard, and leverages

on the spurious ambiguity of the parser to enlarge the search space.

The two components above are mutually independent, meaning that we can individually implement each component on top of an arc-standard model. More precisely, the arc-standard + spine model uses the transitions la_k/ra_k but retains the definition of canonical computation, defined by applying each la_k/ra_k transition as soon as possible. On the other hand, the arc-standard + easy-first model retains the original la/ra transitions but is trained allowing any correct transition at each configuration. In this case the characterization of correct and incorrect configurations in Lemma 1 has been adapted to transitions la/ra, taking into account the bottom-up constraint.

With the purpose of incremental comparison, we report accuracy results for the two 'incremental' models in the last two lines of Table 2. Analyzing these results, and comparing with the plain arc-standard, we see that the spine representation and the easy-first strategy individually improve accuracy. Moreover, their combination into our model (third line of Table 2) works very well, with an overall improvement larger than the sum of the individual contributions.

We now turn to a computational analysis. At each iteration our parser evaluates a number of transitions bounded by $\gamma + 1$, with γ the maximum value of the sum of the lengths of the left spine in σ_1 and of the right spine in σ_2. Quantity γ is bounded by the length n of the input sentence. Since the parser applies exactly $2n + 1$ transitions, worst case running time is $\mathcal{O}(n^2)$. We have computed the average value of γ on our English data set, resulting in 2.98 (variance 2.15) for training set, and 2.95 (variance 1.96) for development set. We conclude that, in the expected case, running time is $\mathcal{O}(n)$, with a slow down constant which is rather small, in comparison to standard transition-based parsers. Accordingly, when running our parser against our implementation of the arc-eager and arc-standard models, we measured a slow-down of 2.8 and 2.2, respectively. Besides the change in representation, this slow-down is also due to the increase in the number of features in our system. We have also checked the worst case value of γ in our data set. Interestingly, we have seen that for strings of length smaller than 40 this value linearly grows with n, and for longer strings the growth stops, with a maximum worst case observed value of 22.

6 Concluding Remarks

We have presented a novel transition-based parser using a dynamic parsing strategy, which achieves a $\sim$12% error reduction in unlabeled attachment score over the static arc-eager strategy and even more over the (equally static) arc-standard strategy, when evaluated on English.

The idea of representing the right spine of a tree within the stack elements of a shift-reduce device is quite old in parsing, predating empirical approaches. It has been mainly exploited to solve the PP-attachment problem, motivated by psycholinguistic models. The same representation is also adopted in applications of discourse parsing, where right spines are usually called right frontiers; see for instance Subba and Di Eugenio (2009). In the context of transition-based dependency parsers, right spines have also been exploited by Kitagawa and Tanaka-Ishii (2010) to decide where to attach the next word from the buffer. In this paper we have generalized their approach by introducing the symmetrical notion of left spine, and by allowing attachment of full trees rather than attachment of a single word.[2]

Since one can regard a spine as a stack in itself, whose elements are tree nodes, our model is reminiscent of the embedded pushdown automata of Schabes and Vijay-Shanker (1990), used to parse tree adjoining grammars (Joshi and Schabes, 1997) and exploiting a stack of stacks. However, by imposing projectivity, we do not use the extra-power of the latter class.

An interesting line of future research is to combine our dynamic parsing strategy with a training method that allows the parser to explore transitions that apply to incorrect configurations, as in Goldberg and Nivre (2012).

Acknowledgments

We wish to thank Liang Huang and Marco Kuhlmann for discussion related to the ideas reported in this paper, and the anonymous reviewers for their useful suggestions. The second author has been partially supported by MIUR under project PRIN No. 2010LYA9RH_006.

[2]Accuracy comparison of our work with Kitagawa and Tanaka-Ishii (2010) is not meaningful, since these authors have evaluated their system on the same data set but based on gold part-of-speech tags (personal communication).

References

Giuseppe Attardi. 2006. Experiments with a multilanguage non-projective dependency parser. In *Proceedings of the 10th Conference on Computational Natural Language Learning (CoNLL)*, pages 166–170.

Miguel Ballesteros and Joakim Nivre. 2013. Going to the roots of dependency parsing. *Computational Linguistics*, 39(1):5–13.

Bernd Bohnet and Joakim Nivre. 2012. A transition-based system for joint part-of-speech tagging and labeled non-projective dependency parsing. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 1455–1465.

Jinho D. Choi and Martha Palmer. 2011. Getting the most out of transition-based dependency parsing. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 687–692.

Michael Collins and Nigel Duffy. 2002. New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 263–270, Philadephia, Pennsylvania.

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 Freund and Robert E. Schapire. 1999. Large margin classification using the perceptron algorithm. *Machine Learning*, 37(3):277–296, December.

Yoav Goldberg and Michael Elhadad. 2010. An efficient algorithm for easy-first non-directional dependency parsing. In *Proceedings of Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, pages 742–750, Los Angeles, USA.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of the 24th International Conference on Computational Linguistics (COLING)*, pages 959–976.

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

Kotaro Kitagawa and Kumiko Tanaka-Ishii. 2010. Tree-based deterministic dependency parsing — an application to Nivre's method —. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL) Short Papers*, pages 189–193.

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

Ryan McDonald and Joakim Nivre. 2007. Characterizing the errors of data-driven dependency parsing models. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 122–131.

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.

Yves Schabes and K. Vijay-Shanker. 1990. Deterministic left to right parsing of tree adjoining languages. In *Proceedings of the 28th annual meeting of the Association for Computational Linguistics (ACL)*, pages 276–283, Pittsburgh, Pennsylvania.

Rajen Subba and Barbara Di Eugenio. 2009. An effective discourse parser that uses rich linguistic information. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 566–574.

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 Stephen Clark. 2008. A tale of two parsers: Investigating and combining graph-based and transition-based dependency parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 562–571.

Yue Zhang and Joakim Nivre. 2011. Transition-based parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 188–193.