# Calculating the optimal step of arc-eager parsing for non-projective trees

**Mark-Jan Nederhof**
School of Computer Science
University of St Andrews, UK
markjan.nederhof@googlemail.com

## Abstract

It is shown that the optimal next step of an arc-eager parser relative to a non-projective dependency structure can be calculated in cubic time, solving an open problem in parsing theory. Applications are in training of parsers by means of a 'dynamic oracle'.

## 1 Introduction

A deterministic transition-based dependency parser is often driven by a classifier that determines the next step, given features extracted from the current configuration (Nivre et al., 2004). The classifier may be trained on parser configurations and steps that exactly correspond to 'gold' trees from a treebank. However, better accuracy is generally obtained by also including configurations reached by letting the parser stray from the gold trees, to let the classifier learn how best to recover from any mistakes. This is associated with the term *dynamic oracle*. If the parser is projective while the gold trees are non-projective moreover, then it is unavoidable that configurations be considered that do not correspond to the gold trees.

Determining the desired output of the classifier requires calculation of the best next step given an arbitrary configuration and a gold tree. Typically, this is the step that allows the most accurate tree to be reached, in terms of the gold tree.[1]

For a gold tree that is projective, the optimal step can be determined in linear time for arc-eager parsing (Goldberg and Nivre, 2012, 2013) and for shift-reduce parsing (Nederhof, 2019). For a non-projective gold tree, the optimal step can be determined for several types of non-projective parsers (Gómez-Rodríguez et al., 2014; Gómez-Rodríguez and Fernández-González, 2015; de Lhoneux et al., 2017; Fernández-González and Gómez-Rodríguez, 2018), as well as for shift-reduce parsing (Nederhof, 2019). However, for arc-eager parsing, the problem has been unsolved until now. Aufrant et al. (2018) propose an *approximation* of the optimal step, based on the procedure for projective gold trees, and demonstrate the advantages of training a projective parser directly on non-projective trees.

The current paper introduces an *exact* calculation of the optimal step for arc-eager parsing and a non-projective gold tree, within the same framework as Nederhof (2019), which consists of a generic cubic-time tabular dependency parsing algorithm and a fixed context-free grammar that is applied on a string extracted from the current configuration, with edge weights determined by the gold tree.

For arc-eager parsing, the context-free grammar is considerably more complex than in the case of shift-reduce parsing. This is a consequence of theoretical properties of arc-eager parsing, which we first need to investigate in detail before we can define 'optimality' of the next step.

## 2 Dependency structures

Let $w = a_1 \cdots a_n$ be a sentence consisting of $n$ tokens, which can be words or punctuation. Where we use indices between 1 and $n$, we also refer to these as tokens, relying on the assumption that given an index $i$ we can retrieve the actual token $a_i$. An additional index 0 represents an imaginary token prepended to the sentence.

An *unlabeled dependency structure* $T$ for $w$ is an unlabeled tree with $\{0, 1, \ldots, n\}$ as the set of nodes, of which 0 is the root. We represent such a tree as a set of edges, each represented as a pair $(a, b)$, where index $a$ is the *parent* and index $b$ is the *child*. The *descendants* of a node are the node itself and the descendants of its children. A dependency structure is *projective* if the set of de-

[1] There are alternative perspectives on how to determine the best next step; cf. Straka et al. (2015).



*Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics*, pages 2273–2283
April 19 - 23, 2021. 

scendants of each node in the tree can be written as $\{i, i+1, \ldots, j-1, j\}$ for some $i$ and $j$ $(0 \leq i \leq j \leq n)$.

We assume each sentence $w = a_1 \cdots a_n$ has a distinguished *gold* tree $T_g$. The *score* of an arbitrary tree $T$ for the same $w$ is defined as $|T \cap T_g|$. The *accuracy* of $T$ is its score divided by $n$.

A dependency parser is usually designed to find a tree that is as accurate as possible, given an input sentence. Such a parser can generally be extended in a natural way to find a *labeled* dependency structure, which is analogously defined as a *labeled* tree with root 0. An edge label in such a structure is called a *dependency relation*. This paper focuses on unlabeled dependency parsing.

## 3 Transition-based dependency parsing

Transition-based dependency parsing is commonly formalized in terms of a set of configurations and a finite set of transitions between configurations. For now, a *configuration* for input sentence $w = a_1 \cdots a_n$ is a 3-tuple $(\alpha, \beta, T)$, where $\alpha$ is the *stack*, $\beta$ is the *remaining input*, and $T$ is a subset of (the set of edges of) a dependency structure. We assume $\alpha\beta$ is a subsequence of $0\ 1\ \cdots\ n$, and $\beta$ is more specifically a suffix of $1\ \cdots\ n$. A *transition* is a partial function, mapping one configuration to another. A *step* is one application of a transition. A *computation* is a sequence of steps, starting with the *initial* configuration $(0, 1\ \cdots\ n, \emptyset)$ and ending in a *final* configuration $(0, \varepsilon, T)$ where $\varepsilon$ denotes the empty string; here $T$ is the resulting tree.

A transition may have a *precondition*, i.e. a condition on the current configuration that must hold for the transition to be applicable. Unrestricted preconditions are less than convenient for our purposes, and therefore we opt for a more uniform framework, in which a stack element is a pair $(a, A)$ consisting of a token $a$ and a *label* $A$ taken from a fixed set.[2] To avoid clutter, we write $a^A$ in place of $(a, A)$; this also emphasizes the relation to more traditional formulations of dependency parsing, which are obtained by omitting the superscripts.

A first illustration of this is traditional shift-reduce dependency parsing, defined by the transitions in Table 1, here without labels, or alternatively, one may consider there to be only one such

**shift**:
$(\alpha,\ b\beta,\ T) \vdash^{\text{SH}} (\alpha b,\ \beta,\ T)$
**reduce_left**:
$(\alpha a_1 a_2,\ \beta,\ T) \vdash^{\text{RL}} (\alpha a_1,\ \beta,\ T \cup \{(a_1, a_2)\})$
**reduce_right**:
$(\alpha a_1 a_2,\ \beta,\ T) \vdash^{\text{RR}} (\alpha a_2,\ \beta,\ T \cup \{(a_2, a_1)\})$,
if $|\alpha| > 0$

Table 1: Shift-reduce dependency parsing.

**shift**:
$(\alpha a^C,\ b\beta,\ T) \vdash^{\text{SH}} (\alpha a^C b^N,\ \beta,\ T)$
**complete**:
$(\alpha a^N,\ \beta,\ T) \vdash^{\text{CO}} (\alpha a^C,\ \beta,\ T)$
**reduce_left**:
$(\alpha a_1^C a_2^C,\ \beta,\ T) \vdash^{\text{RL}} (\alpha a_1^C,\ \beta,\ T \cup \{(a_1, a_2)\})$
**reduce_right**:
$(\alpha a_1^C a_2^N,\ \beta,\ T) \vdash^{\text{RR}} (\alpha a_2^N,\ \beta,\ T \cup \{(a_2, a_1)\})$,
if $|\alpha| > 0$

Table 2: Shift-reduce dependency parsing enforcing the left-before-right policy.

label, which is left implicit.[3]

This form of parsing suffers from spurious ambiguity in that left and right children may be attached in different orders. E.g. if token $b$ has left child $a_1$ and right child $a_2$, then after a **shift** of $a_1$ and $b$, there may be a **reduce_right** followed by a **shift** of $a_2$ followed by a **reduce_left**. Or there may be a **shift** of $a_2$ followed by a **reduce_left** followed by a **reduce_right**. This can be resolved by requiring that left children are attached before right children are. In our framework, this left-before-right policy can be enforced by introducing a label $C$, which is given to a token to signal that it is 'complete' with regard to its left children. Initially, shifted tokens carry label $N$ (for 'no restriction'). The 0 token always has label $C$. This results in Table 2.

There is a simple one-to-one correspondence between a computation according to Table 2 and a computation according to Table 1 satisfying the left-before-right policy. The difference is merely an application of **complete** just before a token ceases to be a topmost stack element, either because it is reduced into the token to its left, or because another token is pushed on top. If a token

[2] There is a close connection to bilexical context-free grammars (Eisner and Satta, 1999), on the basis of which one may alternatively choose to refer to such a label as a 'delexicalized stack symbol', in a kind of lexicalized pushdown automaton.

[3] Shift-reduce dependency parsing has been known at least since Fraser (1989) and Nasr (1995). It is also referred to as 'arc-standard' parsing (Nivre, 2008).

**shift**:
$(\alpha,\ b\beta,\ T) \vdash^{\text{SH}} (\alpha b,\ \beta,\ T)$
**left_arc**:
$(\alpha a,\ b\beta,\ T) \vdash^{\text{LA}} (\alpha,\ b\beta,\ T \cup \{(b,a)\}),$
if $a \neq 0 \wedge \nexists a'[(a',a) \in T]$
**right_arc**:
$(\alpha a,\ b\beta,\ T) \vdash^{\text{RA}} (\alpha a b,\ \beta,\ T \cup \{(a,b)\}),$
if $\nexists a'[(a',b) \in T]$
**reduce**:
$(\alpha a,\ \beta,\ T) \vdash^{\text{RE}} (\alpha,\ \beta,\ T),$
if $\exists a'[(a',a) \in T]$

Table 3: Arc-eager parsing (Nivre, 2008, p. 525).

**shift**:
$(\alpha,\ b\beta,\ T) \vdash^{\text{SH}} (\alpha b^L,\ \beta,\ T)$
**left_arc**:
$(\alpha a^L,\ b\beta,\ T) \vdash^{\text{LA}} (\alpha,\ b\beta,\ T \cup \{(b,a)\})$
**right_arc**:
$(\alpha a^X,\ b\beta,\ T) \vdash^{\text{RA}} (\alpha a^X b^R,\ \beta,\ T \cup \{(a,b)\})$
**reduce**:
$(\alpha a_1^X a_2^R,\ \beta,\ T) \vdash^{\text{RE}} (\alpha a_1^X,\ \beta,\ T)$
**reduce_correct**:
$(\alpha a_1^X a_2^L,\ \beta,\ T) \vdash^{\text{RC}} (\alpha a_1^X,\ \beta,\ T \cup \{(a_1,a_2)\})$

Table 4: Reformulated arc-eager parsing, with $X \in \{R, L\}$, with an extra fifth transition needed to make it work in practice.

becomes non-topmost and reappears later on top of the stack, after applications of **reduce_left** that give it right children, then it will still have label $C$, which prevents it from taking further left children.

Table 3 is almost verbatim the formulation of arc-eager parsing by Nivre (2008), except that we renamed symbols, and we ignore dependency relations; the formulations by e.g. Nivre (2003, 2004) and Nivre et al. (2004) are largely equivalent. It is easy to see that the condition $\nexists a'[(a',b) \in T]$ for **right_arc** is redundant, as no tokens in the remaining input can obtain parents before they are shifted to the stack.

Taking shift-reduce parsing as starting point, **reduce_right** corresponds roughly to **left_arc**, while the role of **reduce_left** is only partly fulfilled by **right_arc**, which postulates that $b$ is a right child of $a$, but without removing $b$ as yet, allowing $b$ to take right children, and only later is that $b$ removed by **reduce**. Here shift-reduce parsing would postpone the decision whether that $b$ is the left or the right child of its parent until all descendants of $b$ have been shifted and reduced into it.

From the perspective of parsing of artificial languages (Sippu and Soisalon-Soininen, 1990), this is counter-intuitive. The conventional wisdom of deterministic parsing is that one should postpone commitment to occurrences of grammar rules (or here, dependency edges) for as long as possible, until enough information is available to resolve any local ambiguity, assuming left-to-right processing of the input string, and the ability to inspect only the top of the stack and the next $k$ tokens of the remaining input, for a fixed, small number $k$.

Two arguments have been given why arc-eager parsing is nonetheless superior for processing natural language. The first is that this earlier commitment made by **right_arc**, in terms of the earlier creation of the dependency edge, offers additional information about the tree under construction, to better predict the next steps, using some type of classifier. The second argument in favor of arc-eager parsing is that the earlier creation of the dependency edge ensures that the partial tree under construction remains as connected as possible, which may help simultaneous syntactic and semantic processing. See Nivre (2004, 2008) and Damonte et al. (2017) for related discussions.

Next we rephrase arc-eager parsing to use labels to express preconditions, to prepare us for Section 4. Note that a token is transferred from the remaining input to the stack by either **shift** or **right_arc**. In the former case, it must eventually become a left child of its parent, and in the latter case, it becomes a right child. We use labels $L$ and $R$ for these cases.[4] In a configuration with set $T$ as third element, existence of a stack element $a^L$ implies $\nexists a'[(a',a) \in T]$ and $a^R$ implies $\exists a'[(a',a) \in T]$. We thereby obtain the first four transitions in Table 4. Token 0 always has label $R$, and cannot be popped by **reduce** due to the $a_1^X$. The fifth transition will be discussed later.

Arc-eager parsing in either of the above two formulations cannot work in practice. The problem is illustrated in Table 5. In the last configuration, none of the steps is applicable. The situation arises when the remaining input becomes empty while there is a $L$ label *anywhere* in the stack. Assuming the classifier used for predicting the next step cannot look unboundedly deep in the stack, this problem is unavoidable.

[4] $L$ and $R$ are akin to 0 and 1 from Kuhlmann et al. (2011).

$$
\begin{array}{llll}
(0^R & , \; 1\,2\,3, & \emptyset & ) \vdash^{\text{SH}} \\
(0^R\,1^L & , \; 2\,3, & \emptyset & ) \vdash^{\text{RA}} \\
(0^R\,1^L\,2^R & , \; 3, & \{(1,2)\} & ) \vdash^{\text{RA}} \\
(0^R\,1^L\,2^R\,3^R, & & \{(1,2),(2,3)\}) & \vdash^{\text{RE}} \\
(0^R\,1^L\,2^R & , & \{(1,2),(2,3)\}) & \vdash^{\text{RE}} \\
(0^R\,1^L & , & \{(1,2),(2,3)\}) &
\end{array}
$$

Table 5: Arc-eager parsing is stuck in a configuration without applicable transitions.

One possible fix is to add the **unshift** transition of Nivre and Fernández-González (2014); see also Honnibal and Johnson (2015). As this causes considerable complications to our framework, we will solve this in another way, reminiscent of Honnibal et al. (2013), which also helps to make a connection with shift-reduce parsing later. Our proposed fix is to allow a **reduce** even if the top of stack has label $L$, by means of the fifth transition of Table 4, **reduce_correct**. This transition is not needed during training if only computations are considered that most straightforwardly correspond to gold trees, with **left_arc** applied only on a token that is to become the left child of its parent. This may mean however that, in the case of labeled dependency parsing, the trained classifier has no basis to predict the dependency relation of the edge created by this transition when applied during testing. This can be solved by moving the creation of the edge from **right_arc** to **reduce**, and by then merging **reduce** and **reduce_correct**, as in Table 6.

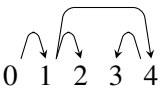
This formulation at first sight appears to nullify the property that has been argued to give arc-eager an advantage over shift-reduce parsing, namely the early availability of edges connecting a parent and a right child. However, these edges are still identified by investigating which tokens in the stack have label $R$: their parent is the token immediately left to it in the stack. In other words, a classifier predicting the next step can be made to have access to the exact same feature values as before.

This formulation of arc-eager parsing, as do the original formulations (Goldberg and Nivre, 2012, p. 963), allow the same dependency structure to be obtained in two different ways; cf. Table 7. There are few studies that compare parsing accuracy between the two ways of resolving this, by preferring either **shift** before **reduce**, or **reduce** before **shift**, and some literature suggests the choice is arbitrary,[5] although the results from one study

[5]Cf. "harmless SHIFT-REDUCE conflicts" (Nivre, 2006, p. 98).

**shift**:
$(\alpha,\ b\beta,\ T) \vdash^{\text{SH}} (\alpha b^L,\ \beta,\ T)$
**left_arc**:
$(\alpha a^L,\ b\beta,\ T) \vdash^{\text{LA}} (\alpha,\ b\beta,\ T \cup \{(b,a)\})$
**right_arc**:
$(\alpha a^X,\ b\beta,\ T) \vdash^{\text{RA}} (\alpha a^X b^R,\ \beta,\ T)$
**reduce**:
$(\alpha a_1^X a_2^Y,\ \beta,\ T) \vdash^{\text{RE}} (\alpha a_1^X,\ \beta,\ T \cup \{(a_1,a_2)\})$

Table 6: Corrected arc-eager parsing, with $X, Y \in \{R, L\}$.




$$
\begin{array}{llll}
(0^R\,1^R\,2^R & , 3\;4, & \emptyset & ) \vdash^{\text{SH}} \\
(0^R\,1^R\,2^R\,3^L, & \; 4, & \emptyset & ) \vdash^{\text{LA}} \\
(0^R\,1^R\,2^R & , \; 4, & \{(4,3)\} & ) \vdash^{\text{RE}} \\
(0^R\,1^R & , \; 4, & \{(1,2),(4,3)\}) & \vdash \ldots
\end{array}
$$

or

$$
\begin{array}{llll}
(0^R\,1^R\,2^R & , 3\;4, & \emptyset & ) \vdash^{\text{RE}} \\
(0^R\,1^R & , 3\;4, & \{(1,2)\} & ) \vdash^{\text{SH}} \\
(0^R\,1^R\,3^L & , \; 4, & \{(1,2)\} & ) \vdash^{\text{LA}} \\
(0^R\,1^R & , \; 4, & \{(1,2),(4,3)\}) & \vdash \ldots
\end{array}
$$

Table 7: Same structure obtained in two ways.

(Qi and Manning, 2017) suggest the shift-before-reduce policy could be slightly better.

One way to enforce the reduce-before-shift policy is to opt for a different division of labor between stack and the leftmost token of the remaining input, whereby we must shift a node from remaining input to stack before it obtains its first left child or before it is decided whether it is to be a left child or a right child. Table 8 presents this *normalized* arc-eager parsing. The **shift** is now simply the transfer of a token from remaining input to stack, without making a commitment whether it is to become a left or right child, which is indicated by the $N$ label. Where before we had **shift** and **right_arc**, we now have **left_child** and **right_child**, which commit a token on top of the stack to be a left or right child of its parent, respectively. Where before we had **left_arc**, we now have **reduce_right**, and **reduce** is more appropriately renamed to **reduce_left**. There is a simple one-to-one correspondence between a computation according to Table 8 and a computation according to Table 6 satisfying the reduce-before-

**shift**:
$(\alpha a^X,\ b\beta,\ T) \vdash^{\text{SH}} (\alpha a^X b^N,\ \beta,\ T)$
**left_child**:
$(\alpha a^N,\ \beta,\ T) \vdash^{\text{L}} (\alpha a^L,\ \beta,\ T)$
**right_child**:
$(\alpha a^N,\ \beta,\ T) \vdash^{\text{R}} (\alpha a^R,\ \beta,\ T)$
**reduce_left**:
$(\alpha a_1^X a_2^Y,\ \beta,\ T) \vdash^{\text{RL}} (\alpha a_1^X,\ \beta,\ T \cup \{(a_1, a_2)\})$
**reduce_right**:
$(\alpha a_1^L a_2^N,\ \beta,\ T) \vdash^{\text{RR}} (\alpha a_2^N,\ \beta,\ T \cup \{(a_2, a_1)\})$

Table 8: Normalized arc-eager parsing, with $X, Y \in \{R, L\}$.

shift policy. Moreover, the same feature values can be used, albeit after renaming. Specifically, if the top of stack has label $N$, then e.g. a feature referring to the top of stack in the case of Table 6 should instead refer to the first element underneath the top of stack in the case of Table 8.[6]

One advantage of the normalized formulation is that it clearly reveals the relation to shift-reduce parsing. In particular, instead of **complete** in Table 2, we have the more specific **left_child** and **right_child**, of which the latter constitutes the early commitment of a token to be right child, as explained before.

## 4 Calculating the optimal step

Assume there are $\tau$ transitions, denoted by $\vdash^1, \ldots, \vdash^\tau$. Let $\vdash$ represent an application of any of these transitions, and let $\vdash^*$ denote the reflexive transitive closure of $\vdash$. For a given configuration $(\alpha, \beta, T)$ for input sentence $w$ with gold tree $T_g$, there are up to $\tau$ steps $(\alpha, \beta, T) \vdash^i (\alpha_i, \beta_i, T_i)$, $i = 1, \ldots, \tau$. For each of these, the *score* is the maximal $\rho_i = |T_i' \cap T_g|$ with $(\alpha_i, \beta_i, T_i) \vdash^* (0^R, \epsilon, T_i')$ for some $T_i'$; if $\vdash^i$ is not applicable on $(\alpha, \beta, T)$, or if no final configuration is reachable from $(\alpha_i, \beta_i, T_i)$, then we set $\rho_i = -\infty$. The task is now to compute that $\rho_i$ for each $i$. This determines which transition to apply next, to eventually obtain the highest-scoring tree, irrespective of any 'incorrect' steps performed in the past, that is, steps that were inconsistent with the gold tree. Because $|T \cap T_g|$ is the same for all $i$, and because the value of $|T_i'' \cap T_g| \leq 1$ with $(\alpha, \beta, \emptyset) \vdash^i (\alpha_i, \beta_i, T_i'')$ is easily determined by a single lookup, the remaining problem is to compute the maximal $\sigma_i = |T_i''' \cap T_g|$ with $(\alpha_i, \beta_i, \emptyset) \vdash^* (0^R, \epsilon, T_i''')$.

As shown by Goldberg and Nivre (2012, 2013), the optimal step can be determined in linear time for (uncorrected) arc-eager parsing, provided the gold tree is projective. The procedure is defined in terms of *costs* of transitions, rather than in terms of scores. We revisit this in Section 5.

For normalized arc-eager parsing (Table 8) and projective gold trees, the problem appears to be no easier than for shift-reduce parsing, but can still be solved in linear time, by a straightforward refinement of the algorithm by Nederhof (2019), blocking a token from becoming a left child of its parent if its label is $R$.

Now assume the gold tree may be non-projective. For shift-reduce parsing, Nederhof (2019) presents a cubic-time algorithm for calculating $\sigma_i$, generalizing the procedure of Goldberg et al. (2014), which is applicable only on projective trees. The algorithm has a modular design, in terms of a generic tabular dependency parsing algorithm (Eisner and Satta, 1999), plus an explicitly 'split' bilexical context-free grammar (Eisner and Satta, 1999; Eisner, 2000; Johnson, 2007) that encodes computations of shift-reduce parsing. A given configuration is translated to an input string, and weights between pairs of input positions are set according to existence of edges between corresponding tokens in the gold tree. Exhaustive parsing of the string by the grammar, using an appropriate semiring, yields $\sigma_i$.

Here we show that the same framework is applicable on arc-eager parsing. The generic tabular dependency parsing algorithm remains the same, but a new grammar is needed to encode computations of arc-eager parsing. Following Nederhof (2019), nonterminals are either single symbols or pairs of symbols, and rules are of one of the forms: $A \to (B, C)$, $(B, C) \to a$, $(B', \_) \to A\ (B, \_)$ or $(\_, C') \to (\_, C)\ A$, where $a$ is a terminal. The last two forms are shorthand for any rules obtained by consistent substitution of the two underscores; which symbols can be meaningfully substituted is clear from context, as exemplified below.

We start with the normalized form (Table 8), which requires the grammar in Table 9, with the indicated translation from the configuration to an input string. The intuition behind this grammar is similar to the one by Nederhof (2019), but more cases need to be distinguished due to the labels. Grammar symbols $R$ and $R_t$ correspond to tokens

[6] The division of labor between stack and remaining input is also what distinguishes Table 2 from the *hybrid model* of Kuhlmann et al. (2011).

| | | | |
|---|---|---|---|
| 1) | $(R, R) \rightarrow \boldsymbol{r}$ | 10) | $(L, L) \rightarrow \boldsymbol{\ell}$ |
| 2) | $(R_t, R_t) \rightarrow \boldsymbol{r_t}$ | 11) | $(L_t, L_t) \rightarrow \boldsymbol{\ell_t}$ |
| 3) | $(\_, R_t) \rightarrow (\_, R)\ R_t$ | 12) | $(\_, L_t) \rightarrow (\_, L)\ R_t$ |
| 4) | $(\_, R_t) \rightarrow (\_, R)\ L_t$ | 13) | $(\_, L_t) \rightarrow (\_, L)\ L_t$ |
| 5) | $(\_, R_t) \rightarrow (\_, R)\ N$ | 14) | $(\_, L_t) \rightarrow (\_, L)\ N$ |
| 6) | $(\_, R_t) \rightarrow (\_, R_t)\ N$ | 15) | $(\_, L_t) \rightarrow (\_, L_t)\ N$ |
| 7) | $R \rightarrow (R, R)$ | 16) | $L \rightarrow (L, L)$ |
| 8) | $R_t \rightarrow (R, R_t)$ | 17) | $L_t \rightarrow (L, L_t)$ |
| 9) | $R_t \rightarrow (R_t, R_t)$ | 18) | $L_t \rightarrow (L_t, L_t)$ |
| 19) | $(N, N) \rightarrow \boldsymbol{n}$ | 24) | $(N, \_) \rightarrow L_b\ (N, \_)$ |
| 20) | $(\_, N) \rightarrow (\_, N)\ N$ | 25) | $N \rightarrow (N, N)$ |
| 21) | $(N, \_) \rightarrow N\ (N, \_)$ | 26) | $(L_b, L_b) \rightarrow \boldsymbol{\ell_b}$ |
| 22) | $(N, \_) \rightarrow L\ (N, \_)$ | 27) | $(\_, L_b) \rightarrow (\_, L_b)\ N$ |
| 23) | $(N, \_) \rightarrow L_t\ (N, \_)$ | 28) | $L_b \rightarrow (L_b, L_b)$ |

Table 9: Grammar for normalized arc-eager dependency parsing of a string in $\{\boldsymbol{r}, \boldsymbol{\ell}\}^{k-1}\{\boldsymbol{r_t}, \boldsymbol{\ell_t}, \boldsymbol{\ell_b}, \boldsymbol{n}\}\boldsymbol{n}^m$, representing a stack of length $k$ and a remaining input of length $m$. A label $R$ in the top of stack is translated to $\boldsymbol{r_t}$, and other occurrences of $R$ are translated to $\boldsymbol{r}$. A label $L$ in the top of stack is translated to $\boldsymbol{\ell_t}$, unless the candidate transition is **left_child**, in which case it is translated to $\boldsymbol{\ell_b}$; other occurrences of $L$ are translated to $\boldsymbol{\ell}$. A label $N$ in the top of stack is translated to $\boldsymbol{n}$.

in the stack with label $R$, where $R_t$ specifically means that the token is on top of the stack. Rules (3)–(4) distinguish the two cases $Y \in \{R, L\}$ of **reduce_left** with $X = R$. The rules are best read from right to left, as here for example "if the top of stack has label $R$ or $L$, and if the token underneath has label $R$, then the latter keeps its label $R$ and becomes the top of stack". Rules (5)–(6) allow for **reduce_left** with a right child that was in the remaining input, or that was the top of stack with label $N$. In (6), the underscore can be substituted by $R$ or $R_t$. In (3)–(5), the only meaningful substitution is by $R$. Rules (10)–(18) are analogous to (1)–(9). Rules (20)–(21) allow any projective parse of the remaining input (as well as of the top of stack if that had label $N$), and (22)–(24) handle a token in the remaining input taking a left child in the stack, provided it has label $L$. In (20)–(24), the only meaningful substitution of the underscore is by $N$.

If a token has already been given label $L$, then it becoming a right child by (4) or (13) amounts to correcting a mistake made earlier, and may be necessary so the computation does not get stuck (cf. Table 5). If the next transition to be considered is **left_child** however, which puts $L$ in the top of

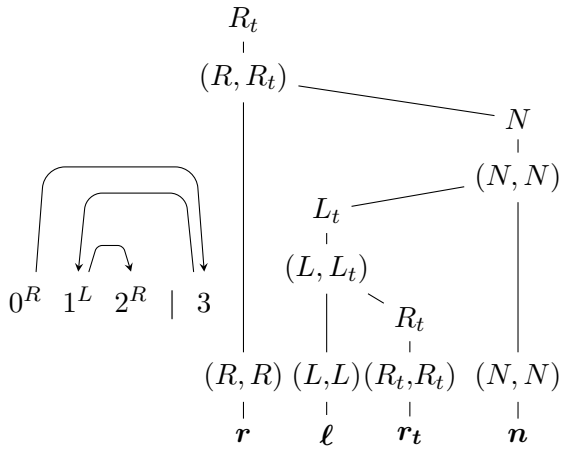



Figure 1: Dependency structure and configuration with stack of height 3 and remaining input of length 1, and corresponding derivation.

stack, then we do not wish the corresponding token to become a right child; **right_child** should be applied instead. Label $L$ is then translated to $\boldsymbol{\ell_b}$ with $\boldsymbol{b}$ for 'blocking' (4) and (13).

Figure 1 exemplifies a derivation encoding a computation. A formal proof of correctness is by induction, showing that existence of a subderivation of the grammar implies existence of a corresponding subcomputation with the same score, and vice versa. Cf. the proof sketch by Nederhof (2019) for shift-reduce parsing.

Unnormalized arc-eager parsing (Table 6) requires a different approach, due to the different division of labor between stack and remaining input. We now need to count the number of right children of a token in the stack that were themselves in the stack, up to but not exceeding 1. E.g. rule (5) in Table 10 counts the first right child, but there is no further rule with right-hand side $(\_, R_1)\ R$ to allow a second right child from among the stack elements; other children from the remaining input are allowed, as e.g. by rule (10).

We now also need to observe the chosen policy. With the shift-before-reduce policy, if the candidate transition is **reduce**, then the first symbol of the remaining input becomes $\boldsymbol{n_p}$ ($\boldsymbol{p}$ for 'policy'). There is a notable absence of a rule with right-hand side $N_p\ (N, \_)$, which means that this token cannot become a left child without first taking a child from the stack as by (39) and (43), because if it were, the policy would be violated: the token should have been shifted, and reduced into its parent on the right, *preceding* the **reduce**. There is no restriction on the token becoming a right child, as e.g.

by (4).

A *strict* reduce-before-shift policy implies that a token in the stack should not be reduced into the token to its right if other tokens were previously shifted on top, unless it is to obtain more right children. This is because by the policy, the reduction should have happened earlier. Alternatively we may opt for a *non-strict* reduce-before-shift policy that allows us to correct mistakes made earlier. Either variant uses $\boldsymbol{r_p}$, $\boldsymbol{\ell_p}$, $R_p$ and $L_p$ to enforce the policy. E.g. there are no rules with $R_p$ in the right-hand side, effectively blocking a derivation. Here rules (7)–(9) are needed to give an $R$-labeled stack element at least one right child, which by (14)–(15) allows the token to participate in a full derivation.

In order to compute the score for arc-eager parsing without our correction (starting in Table 4 with **reduce_correct**), one should omit the rules from Table 10 that correspond to $L$-labeled tokens becoming right children, i.e. (6), (9), (22), (25). Whether the **unshift** from Nivre and Fernández-González (2014) and Honnibal and Johnson (2015) can be handled in our framework requires further study.

| | | | | | |
|---|---|---|---|---|---|
| 1) | $(R, R)$ | $\rightarrow \boldsymbol{r}$ | 17) | $(L, L)$ | $\rightarrow \boldsymbol{\ell}$ |
| 2) | $(R_p, R_p)$ | $\rightarrow \boldsymbol{r_p}$ | 18) | $(L_p, L_p)$ | $\rightarrow \boldsymbol{\ell_p}$ |
| 3) | $(\_, R)$ | $\rightarrow (\_, R)\ N$ | 19) | $(\_, L)$ | $\rightarrow (\_, L)\ N$ |
| 4) | $(\_, R)$ | $\rightarrow (\_, R)\ N_p$ | 20) | $(\_, L)$ | $\rightarrow (\_, L)\ N_p$ |
| 5) | $(\_, R_1)$ | $\rightarrow (\_, R)\ R$ | 21) | $(\_, L_1)$ | $\rightarrow (\_, L)\ R$ |
| 6) | $(\_, R_1)$ | $\rightarrow (\_, R)\ L$ | 22) | $(\_, L_1)$ | $\rightarrow (\_, L)\ L$ |
| 7) | $(\_, R)$ | $\rightarrow (\_, R_p)\ N$ | 23) | $(\_, L)$ | $\rightarrow (\_, L_p)\ N$ |
| 8) | $(\_, R_1)$ | $\rightarrow (\_, R_p)\ R$ | 24) | $(\_, L_1)$ | $\rightarrow (\_, L_p)\ R$ |
| 9) | $(\_, R_1)$ | $\rightarrow (\_, R_p)\ L$ | 25) | $(\_, L_1)$ | $\rightarrow (\_, L_p)\ L$ |
| 10) | $(\_, R_1)$ | $\rightarrow (\_, R_1)\ N$ | 26) | $(\_, L_1)$ | $\rightarrow (\_, L_1)\ N$ |
| 11) | $(\_, R_1)$ | $\rightarrow (\_, R_1)\ N_p$ | 27) | $(\_, L_1)$ | $\rightarrow (\_, L_1)\ N_p$ |
| 12) | $R$ | $\rightarrow (R, R)$ | 28) | $L$ | $\rightarrow (L, L)$ |
| 13) | $R$ | $\rightarrow (R, R_1)$ | 29) | $L$ | $\rightarrow (L, L_1)$ |
| 14) | $R$ | $\rightarrow (R_p, R)$ | 30) | $L$ | $\rightarrow (L_p, L)$ |
| 15) | $R$ | $\rightarrow (R_p, R_1)$ | 31) | $L$ | $\rightarrow (L_p, L_1)$ |
| 16) | $R_p$ | $\rightarrow (R_p, R_p)$ | 32) | $L_p$ | $\rightarrow (L_p, L_p)$ |
| 33) | $(N, N)$ | $\rightarrow \boldsymbol{n}$ | 41) | $(N, \_)$ | $\rightarrow L_b\ (N, \_)$ |
| 34) | $(N_p, N_p)$ | $\rightarrow \boldsymbol{n_p}$ | 42) | $N$ | $\rightarrow (N, N)$ |
| 35) | $(\_, N)$ | $\rightarrow (\_, N)\ N$ | 43) | $N$ | $\rightarrow (N, N_p)$ |
| 36) | $(\_, N_p)$ | $\rightarrow (\_, N_p)\ N$ | 44) | $N_p$ | $\rightarrow (N_p, N_p)$ |
| 37) | $(N, \_)$ | $\rightarrow N\ (N, \_)$ | 45) | $(L_b, L_b)$ | $\rightarrow \boldsymbol{\ell_b}$ |
| 38) | $(N, \_)$ | $\rightarrow L\ (N, \_)$ | 46) | $(\_, L_b)$ | $\rightarrow (\_, L_b)\ N$ |
| 39) | $(N, \_)$ | $\rightarrow L\ (N_p, \_)$ | 47) | $L_b$ | $\rightarrow (L_b, L_b)$ |
| 40) | $(N, \_)$ | $\rightarrow L_p\ (N, \_)$ | | | |

Table 10: Grammar for unnormalized arc-eager dependency parsing. With reduce-before-shift, the string is in $\boldsymbol{r}\{\boldsymbol{r}, \boldsymbol{r_p}, \boldsymbol{\ell}, \boldsymbol{\ell_p}\}^{k-2}\{\boldsymbol{r}, \boldsymbol{r_p}, \boldsymbol{\ell}, \boldsymbol{\ell_p}, \boldsymbol{\ell_b}\}\boldsymbol{n}^m$, for stack length $k$ and remaining input length $m$. Now $\boldsymbol{\ell_b}$ is used if the candidate transition is **shift**, and a non-bottommost symbol to the left of that becomes $\boldsymbol{r_p}$ or $\boldsymbol{\ell_p}$. For a *strict* reduce-before-shift policy moreover, the second to the $k-2$-th symbols become $\boldsymbol{r_p}$ or $\boldsymbol{\ell_p}$, and furthermore the $k-1$-th becomes $\boldsymbol{r_p}$ or $\boldsymbol{\ell_p}$ if the candidate transition is **left_arc** or **reduce**, and furthermore the $k$-th becomes $\boldsymbol{r_p}$ or $\boldsymbol{\ell_p}$ if the candidate transition is **left_arc**; otherwise, these symbols are $\boldsymbol{r}$ or $\boldsymbol{\ell}$. With shift-before-reduce, the string does not contain $\boldsymbol{r_p}$ or $\boldsymbol{\ell_p}$, and the first $\boldsymbol{n}$ is replaced by $\boldsymbol{n_p}$ if the candidate transition is **reduce**.

## 5 Calculation for projective trees

If the gold tree is projective, then the problem becomes much easier. Here we assume the formulation of arc-eager parsing as in Table 6. The number $\sigma_i$, as defined in Section 4, for a configuration with stack $\alpha_i = a_1 \cdots a_k$ and remaining input $\beta_i = b_1 \cdots b_m$, can be calculated by counting in the first instance:

- the number of gold edges $(a_{p-1}, a_p)$, where $1 < p \leq k$, plus
- the number of gold edges $(a_p, b_q)$, plus
- the number of gold edges $(b_p, a_q)$, such that $a_q$ has label $L$, plus
- the number of gold edges $(b_p, b_q)$,

but discounting a number of these, as follows. First, consider the case of the candidate transition being **shift**. If $m = 0$, the score becomes $-\infty$, as there is no available parent for the shifted token. If $m > 0$, we discount a possible gold edge $(a_k, b_p)$ if the rightmost descendant of $b_p$ is $b_m$, because no projective tree exists in which $a_k$ is a left child while its descendants include the end of the input. We further discount a possible gold edge $(a_{k-1}, a_k)$, because if $a_k$ is to become a right child of $a_{k-1}$, then the correct step is **right_arc** in place of **shift**.

Second, if the candidate transition is **reduce**, we discount up to one gold edge in case of the shift-before-reduce policy, as follows, and as illustrated by Figure 2. Let $r$ be largest such that, for some $p > 1$, there is a gold edge $(b_p, a_r)$ where $a_r$ has label $L$, or there is a gold edge $(a_r, b_p)$; if no such gold edge exists, let $r = 1$. If there is no $s$ ($r < s \leq k$) such that $a_s$ has label $L$ and $(a_{s-1}, a_s)$ is not a gold edge, then we discount any gold edge

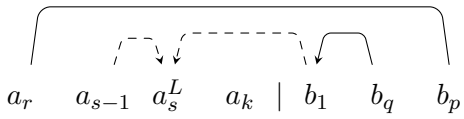



Figure 2: Discounting of $(b_q, b_1)$ if $s$ does not exist.

$(b_q, b_1)$. The rationale is that if $b_1$ can be given a child from among the tokens in the stack (by a gold edge or otherwise, and without discounting another gold edge elsewhere), then this justifies postponing the **shift** until after the **reduce**. If it cannot be, then $b_1$ becoming a left child violates the shift-before-reduce policy.

Lastly, if the candidate transition is **shift**, we discount further gold edges in case of the non-strict reduce-before-right policy, which requires $a_{k-1}$ to either become a child of some $b_p$ or take some $b_p$ as child, to justify it not having been reduced into $a_{k-2}$ before the shift.[7] From among the cases to be distinguished, we choose the one that discounts the fewest edges. First, if the label of $a_{k-1}$ is $L$, we can let it become a left child, but should then discount a possible gold edge $(a_{k-2}, a_{k-1})$. Second, if there is a gold edge $(a_{k-1}, b_p)$, then no edges need be discounted. Otherwise, we need to find a child $b_p$ of $a_{k-1}$, for which there are five options, illustrated in Figure 3: (A) The first option is applicable if $a_k$ has descendants among the remaining input or has a parent $b_q$ ($q > 1$) among the remaining input. In the former case, choose $b_p$ to be the rightmost among the descendants (but let $p = m - 1$ if the rightmost descendant is $b_m$), and in the latter case choose $p = 1$. In effect we assume non-gold edges $(a_{k-1}, b_p)$ and $(b_p, a_k)$, and consequently we discount any gold edge $(b_q, b_p)$ and any gold edge to $a_k$. (B1) If $a_k$ has a parent $b_q$ in the remaining input, choose $b_p$ to be $b_q$. In effect we assume non-gold edge $(a_{k-1}, b_q)$, and consequently we discount any gold edge $(a_r, b_q)$ with $r \leq k-2$ or any gold edge $(b_r, b_q)$, as well as any gold edges $(b_q, a_s)$ with $s \leq k-2$. (B2) If $a_k$ does not have a parent in the remaining input, let $b_q$ be the token immediately to the right of the rightmost descendant of $a_k$ among the remaining input (but let $q = m$ if the rightmost descendant is $b_m$), and let $q = 1$ if $a_k$ has no descendants among the remaining input. As in (B1), we in effect assume non-gold edge $(a_{k-1}, b_q)$, and discount any gold edge $(a_r, b_q)$ with $r \leq k-2$ or any gold edge $(b_u, b_q)$, as well as any gold edges $(b_q, a_s)$ with $s \leq k-2$.

[7] The *strict* reduce-before-right policy is more difficult to realize, and discussion is omitted for space reasons.

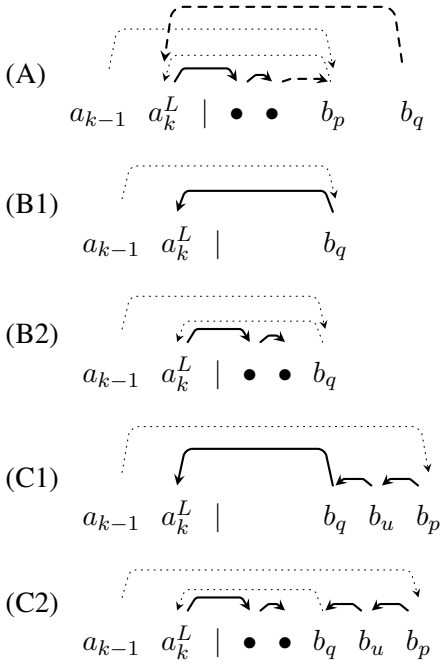



Figure 3: Non-strict reduce-before-shift policy. Bold edges are gold. Dashed edges are discounted. Dotted edges are non-gold.

(C1) and (C2) are similar to (B1) and (B2), but $b_p$ is chosen to be the first ancestor of $b_q$ that does not have a parent in the remaining input (but it may have in the stack). Much as before, we discount any gold edge $(a_r, b_p)$ with $r \leq k-2$, as well as any gold edges $(b_u, a_s)$ with $s \leq k-2$, where $b_u$ is $b_q$ or $b_p$ or any other token on the path of gold edges from $b_q$ to $b_p$. One can show that choices of $b_p$ other than in (A), (B1), (B2), (C1), (C2) would entail discounting of at least as many edges.

Aufrant et al. (2018) propose approximating the calculation of the optimal step for a non-projective gold tree, by a procedure defined in terms of *costs* of transitions, analogous to the procedure by Goldberg and Nivre (2012, 2013), but without taking full account of edges that violate projectivity. Similarly, if the above procedure to calculate *scores* is applied on a non-projective tree, then an approximation is obtained. The advantage is the simplicity and the linear time complexity.

## 6 Empirical results

The advantage of 'dynamic oracles' for improving parsing accuracy has been demonstrated before. Our experiments have therefore concentrated on

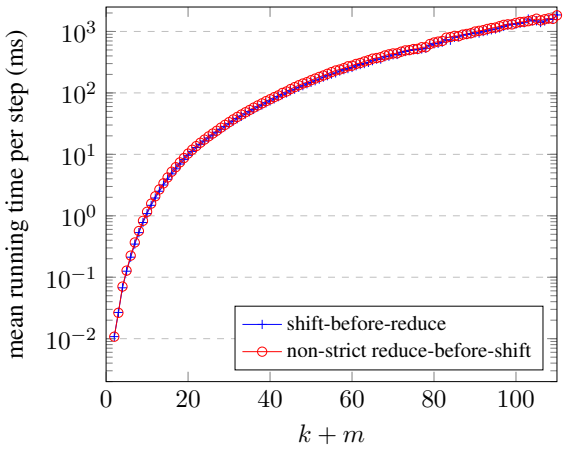



Figure 4: Mean running time per (exact) calculation of the optimal step (milliseconds) against the total length $k + m$ of configurations.

| exact SH-b.-RE | approximation | proportion |
|---|---|---|
| {SH, RA} | {SH} | 29.4% |
| {SH, LA} | {SH} | 20.5% |
| {SH, RA, RE} | {SH} | 10.8% |
| {SH, RA, RE} | {RE} | 7.5% |
| {SH, RA, LA, RE} | {SH, LA} | 6.0% |
| {SH, RA, LA, RE} | {LA, RE} | 4.3% |
| {LA, RE} | {SH,RA,LA,RE} | 3.2% |
| **exact n.-s. RE-b.-SH** | **approximation** | **proportion** |
| {SH, RA} | {SH} | 31.4% |
| {SH, LA, RE} | {SH} | 17.9% |
| {RA, RE} | {RE} | 16.2% |
| {RA, LA, RE} | {LA, RE} | 6.6% |
| {SH, RA, LA, RE} | {SH} | 4.0% |
| {LA, RE} | {SH,RA,LA,RE} | 3.0% |
| {RE} | {SH,RA,RE} | 2.9% |

Table 11: Proportions of the seven most frequent errors made by the approximation of the optimal transition(s).

two obvious questions, viz. whether the cubic-time calculation is feasible in practice, and whether the higher time costs are rewarded with a more accurate output, relative to a linear-time approximation of the kind discussed in Section 5.

Considered here is unnormalized arc-eager parsing. The classifier, implemented in Java and DL4J, uses simple features (gold POS of the three rightmost elements of the stack and three leftmost elements of the remaining input, and leftmost and rightmost dependency relations in the topmost two stack elements).

The parser was first trained on configurations corresponding to projectivized gold trees from the German (GSD) corpus of Universal Dependencies v2.2. The trained parser was then applied on the unprojectivized trees, and the optimal step was calculated for each configuration thus visited.

### 6.1 Running time

Figure 4 presents running time, on a laptop with an Intel i7-7500U processor (4 cores, 2.70 GHz) with 8 GB of RAM. The larger context-free grammar of Table 10, relative to the one for shift-reduce parsing, leads to a higher constant factor in the time complexity. Nonetheless, the calculation is feasible even for long sentences.

### 6.2 Accuracy of the approximation

In 8.0% and 8.1% of the visited configurations, one or more of the values $\rho_1, \ldots, \rho_4$ for the four transitions differed between the exact calculation (Section 4) and the approximation (Section 5), for the shift-before-reduce and non-strict reduce-before-shift policies respectively. However, we are less interested in the absolute values of the scores than in which of them is highest. Note further that more than one may be equal to their maximum. By comparing the sets of transitions with the maximum calculated score, we found that the true set and the approximate set differed for only 0.4% and 0.5% of the total number of configurations, for the two policies respectively. The most frequent errors are listed in Table 11. Somewhat surprisingly, in the great majority of cases, the approximate set was contained in the true set; these cases sum to 89.0% and 87.8% of the total number of errors, respectively. The implication is that if a parser trained with a 'dynamic oracle' does arbitrary tie breaking between multiple optimal transitions, then there are few immediate prospects to improve parsing accuracy by incorporating the exact calculation. The situation may change if future research reveals better alternatives to arbitrary tie breaking.

## 7 Conclusions

Our exact calculation of the optimal step solves an open problem in parsing theory. Further research into the application of 'dynamic oracles' is needed to determine whether this can be exploited to improve parsing accuracy.

## Acknowledgements

Many thanks go to the reviewers, whose reports were very detailed and helpful.

## References


L. Aufrant, G. Wisniewski, and F. Yvon. 2018. Exploiting dynamic oracles to train projective dependency parsers on non-projective trees. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, volume 2, pages 413–419, New Orleans, Louisiana.

M. Damonte, S.B. Cohen, and G. Satta. 2017. An incremental parser for Abstract Meaning Representation. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, volume 1, pages 536–546, Valencia, Spain.

J. Eisner. 2000. Bilexical grammars and their cubic-time parsing algorithms. In H. Bunt and A. Nijholt, editors, *Advances in Probabilistic and other Parsing Technologies*, chapter 3, pages 29–61. Kluwer Academic Publishers.

J. Eisner and G. Satta. 1999. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *37th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 457–464, Maryland, USA.

D. Fernández-González and C. Gómez-Rodríguez. 2018. A dynamic oracle for linear-time 2-planar dependency parsing. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, volume 2, pages 386–392, New Orleans, Louisiana.

N. Fraser. 1989. Parsing and dependency grammar. *UCL Working Papers in Linguistics*, 1:296–319.

Y. Goldberg and J. Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *The 24th International Conference on Computational Linguistics*, pages 959–976, Mumbai, India.

Y. Goldberg and J. Nivre. 2013. Training deterministic parsers with non-deterministic oracles. *Transactions of the Association for Computational Linguistics*, 1:403–414.

Y. Goldberg, F. Sartorio, and G. Satta. 2014. A tabular method for dynamic oracles in transition-based parsing. *Transactions of the Association for Computational Linguistics*, 2:119–130.

C. Gómez-Rodríguez and D. Fernández-González. 2015. An efficient dynamic oracle for unrestricted non-projective parsing. In *53rd Annual Meeting of the Association for Computational Linguistics and 7th International Joint Conference on Natural Language Processing*, volume 2, pages 256–261, Beijing.

C. Gómez-Rodríguez, F. Sartorio, and G. Satta. 2014. A polynomial-time dynamic oracle for non-projective dependency parsing. In *Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, pages 917–927, Doha, Qatar.

M. Honnibal, Y. Goldberg, and M. Johnson. 2013. A non-monotonic arc-eager transition system for dependency parsing. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 163–172, Sofia, Bulgaria.

M. Honnibal and M. Johnson. 2015. An improved non-monotonic transition system for dependency parsing. In *Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, pages 1373–1378, Lisbon, Portugal.

M. Johnson. 2007. Transforming projective bilexical dependency grammars into efficiently-parsable CFGs with Unfold-Fold. In *45th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 168–175, Prague, Czech Republic.

M. Kuhlmann, C. Gómez-Rodríguez, and G. Satta. 2011. Dynamic programming algorithms for transition-based dependency parsers. In *49th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 673–682, Portland, Oregon.

M. de Lhoneux, S. Stymne, and J. Nivre. 2017. Arc-hybrid non-projective dependency parsing with a static-dynamic oracle. In *15th International Conference on Parsing Technologies*, pages 99–104, Pisa, Italy.

A. Nasr. 1995. A formalism and a parser for lexicalised dependency grammars. In *Fourth International Workshop on Parsing Technologies*, pages 186–195, Prague and Karlovy Vary, Czech Republic.

M.-J. Nederhof. 2019. Calculating the optimal step in shift-reduce dependency parsing: From cubic to linear time. *Transactions of the Association for Computational Linguistics*, 7:283–296.

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

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

J. Nivre. 2006. *Inductive Dependency Parsing*. Springer-Verlag.

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

J. Nivre and D. Fernández-González. 2014. Arc-eager parsing with the tree constraint. *Computational Linguistics*, 40(2):259–267.

J. Nivre, J. Hall, and J. Nilsson. 2004. Memory-based dependency parsing. In *Proceedings of the Eighth Conference on Computational Natural Language Learning*, pages 49–56, Boston, Massachusetts.

P. Qi and C.D. Manning. 2017. Arc-swift: A novel transition system for dependency parsing. In *55th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, volume 2, pages 110–117, Vancouver, Canada.

S. Sippu and E. Soisalon-Soininen. 1990. *Parsing Theory, Vol. II: LR($k$) and LL($k$) Parsing*, volume 20 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag.

M. Straka, J. Hajič, J. Straková, and J. Hajič, jr. 2015. Parsing universal dependency treebanks using neural networks and search-based oracle. In *Proceedings of the Fourteenth International Workshop on Treebanks and Linguistic Theories*, pages 208–220, Warsaw, Poland.