Calculating the Optimal Step in Shift-Reduce Dependency Parsing: From Cubic to Linear Time

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

Abstract

We present a new cubic-time algorithm to calculate the optimal next step in shift-reduce dependency parsing, relative to ground truth, commonly referred to as dynamic oracle. Unlike existing algorithms, it is applicable if the training corpus contains non-projective structures. We then show that for a projective training corpus, the time complexity can be improved from cubic to linear.

1 Introduction

A deterministic parser may rely on a classifier that predicts the next step, given features extracted from the present configuration (Yamada and Matsumoto, 2003; Nivre et al., 2004). It was found that accuracy improves if the classifier is trained not just on configurations that correspond to the ground-truth, or “gold”, tree, but also on configurations that a parser would typically reach when a classifier strays from the optimal predictions. This is known as a dynamic oracle.

The effective calculation of the optimal step for some kinds of parsing relies on ‘arc-decomposibility’, as in the case of Goldberg and Nivre (2012, 2013). This generally requires a projective training corpus; an attempt to extend this to non-projective training corpora had to resort to an approximation (Aufrant et al., 2018). It is known how to calculate the optimal step for a number of non-projective parsing algorithms, however (Gómez-Rodríguez et al., 2014; Gómez-Rodríguez and Fernández-González, 2015; Fernández-González Gómez-Rodríguez, 2018a); see also de Lhoneux et al. (2017).

Ordinary shift-reduce dependency parsing is known at least since Fraser (1989); see also Nasr (1995). Nivre (2008) calls it “arc-standard parsing.” For shift-reduce dependency parsing, calculation of the optimal step is regarded to be difficult. The best known algorithm is cubic and is only applicable if the training corpus is projective (Goldberg et al., 2014). We present a new cubic-time algorithm that is also applicable to non-projective training corpora. Moreover, its architecture is modular, expressible as a generic tabular algorithm for dependency parsing plus a context-free grammar that expresses the allowable transitions of the parsing strategy. This differs from approaches that require specialized tabular algorithms for different kinds of parsing (Gómez-Rodríguez et al., 2008; Huang and Sagae, 2010; Kuhlmann et al., 2011).

The generic tabular algorithm is interesting in its own right, and can be used to determine the optimal projectivization of a non-projective tree. This is not to be confused with pseudo-projectivization (Kahane et al., 1998; Nivre and Nilsson, 2005), which generally has a different architecture and is used for a different purpose, namely, to allow a projective parser to produce non-projective structures, by encoding non-projectivity into projective structures before training, and then reconstructing potential non-projectivity after parsing.

A presentational difference with earlier work is that we do not define optimality in terms of “loss” or “cost” functions but directly in terms of attainable accuracy. This perspective is shared by Straka et al. (2015), who also relate accuracies of competing steps, albeit by means of actual parser output and not in terms of best attainable accuracies.
We further show that if the training corpus is projective, then the time complexity can be reduced to linear. To achieve this, we develop a new approach of excluding computations whose accuracies are guaranteed not to exceed the accuracies of the remaining computations. The main theoretical conclusion is that arc-decomposibility is not a necessary requirement for efficient calculation of the optimal step.

Despite advances in unrestricted non-projective parsing, as, for example, Fernández-González and Gómez-Rodríguez (2018b), many state-of-the-art dependency parsers are projective, as, for example, Qi and Manning (2017). One main practical contribution of the current paper is that it introduces new ways to train projective parsers using non-projective trees, thereby enlarging the portion of trees from a corpus that is available for training. This can be done either after applying optimal projectivization, or by computing optimal steps directly for non-projective trees. This can be expected to lead to more accurate parsers, especially if a training corpus is small and a large proportion of it is non-projective.

2 Preliminaries

In this paper, a configuration (for sentence length \( n \)) is a 3-tuple \((\alpha, \beta, T)\) consisting of a stack \(\alpha\), which is a string of integers each between 0 and \( n \), a remaining input \(\beta\), which is a suffix of the string \(1 \cdots n\), and a set \( T \) of pairs \((a, a')\) of integers, with \(0 \leq a \leq n\) and \(1 \leq a' \leq n\). Further, \(\alpha\beta\) is a subsequence of \(0 1 \cdots n\), starting with 0. Integer 0 represents an artificial input position, not corresponding to any actual token of an input sentence.

An integer \(a'\) (\(1 \leq a' \leq n\)) occurs as second element of a pair \((a, a')\) \(\in T\) if and only if it does not occur in \(\alpha\beta\). Furthermore, for each \(a'\) there is at most one \(a\) such that \((a, a')\) \(\in T\). If \((a, a')\) \(\in T\) then \(a'\) is generally called a dependent of \(a\), but as we will frequently need concepts from graph theory in the remainder of this article, we will consistently call \(a'\) a child of \(a\) and \(a\) the parent of \(a'\); if \(a' < a\) then \(a'\) is a left child and if \(a < a'\) then it is a right child. The terminology is extended in the usual way to include descendants and ancestors. Pairs \((a, a')\) will henceforth be called edges.

For sentence length \(n\), the initial configuration is \((0, 1 2 \cdots n, \emptyset)\), and a final configuration is of the form \((0, \varepsilon, T)\), where \(\varepsilon\) denotes the empty string. The three transitions of shift-reduce dependency parsing are given in Table 1. By step we mean the application of a transition on a particular configuration. By computation we mean a series of steps, the formal notation of which uses \(\vdash\), the reflexive, transition closure of \(\vdash\). If \((0, 1 2 \cdots n, \emptyset) \vdash (0, \varepsilon, T)\), then \(T\) represents a tree, with \(0\) as root element, and \(T\) is projective, which means that for each node, the set of its descendants (including that node itself) is of the form \(\{a, a + 1, \ldots, a' - 1, a'\}\), for some \(a\) and \(a'\).

In general, a dependency tree is any tree of nodes labelled \(0, 1, \ldots, n\), with \(0\) being the root.

The score of a tree \(T\) for a sentence is the number of edges that it has in common with a given gold tree \(T_g\) for that sentence, or formally \(|T \cap T_g|\). The accuracy is the score divided by \(n\).

Note that neither tree need be projective for the score to be defined, but in this paper the first tree, \(T\), will normally be projective. Where indicated, also \(T_g\) is assumed to be projective.

Assume an arbitrary configuration \((\alpha, \beta, T)\) for sentence length \(n\) and assume a gold tree \(T_g\) for a sentence of that same length, and assume three steps \((\alpha, \beta, T) \vdash (\alpha_i, \beta_i, T_i)\), with \(i = 1, 2, 3\), obtainable by a shift, reduce_left or reduce_right, respectively. (If \(\beta = \varepsilon\), or \(|\alpha| \leq 2\), then naturally some of the three transitions need to be left out of consideration.)

We now wish to calculate, for each of \(i = 1, 2, 3\), the maximum value of \(|T_i \cap T_g|\), for any \(T_i\) such that \((\alpha_i, \beta_i, T_i) \vdash (0, \varepsilon, T_g)\). For \(i = 1, 2, 3\), let \(\sigma_i\) be this maximum value. The absolute scores \(\sigma_i\) are strictly speaking irrelevant; the relative values determine which is the optimal step, or which are the optimal steps, to reach a tree with the highest score. Note that \(|\{i \mid \sigma_i = \max_j \sigma_j\}|\) is either 1, 2, or 3. In the remainder of this article, we find it more convenient to calculate \(\sigma_i - |T \cap T_g|\) for each \(i\)—or, in other words, gold edges that were previously found are left out of consideration.
We can put restrictions on the set of allowable computations \((\alpha, \beta, T) \vdash^* (0, \varepsilon, T U T')\). The **left-before-right** strategy demands that all edges \((a, a') \in T'\) with \(a' < a\) are found before any edges \((a, a') \in T'\) with \(a < a'\), for each \(a\) that is rightmost in \(\alpha\) or that occurs in \(\beta\). The **strict left-before-right** strategy in addition disallows edges \((a, a') \in T'\) with \(a' < a\) for each \(a\) in \(\alpha\) other than the rightmost element. The intuition is that a non-strict strategy allows us to correct mistakes already made: If we have already pushed other elements on top of a stack element \(a\), then \(a\) will necessarily obtain right children before it occurs on top of the stack again, when it can take (more) left children. By contrast, the strict strategy would not allow these left children.

The definition of the **right-before-left** strategy is symmetric to that of the left-before-right strategy, but there is no independent **strict right-before-left** strategy. In this paper we consider all three strategies in order to emphasize the power of our framework. It is our understanding that Goldberg et al. (2014) does not commit to any particular strategy.

### 3 Tabular Dependency Parsing

We here consider context-free grammars (CFGs) of a special form, with nonterminals in \(N \cup (N_t \times N_r)\), for appropriate finite sets \(N, N_t, N_r\), which need not be disjoint. The finite set of terminals is denoted \(\Sigma\). There is a single start symbol \(S \in N\). Rules are of one of the forms:

- \((B, C) \rightarrow a,\)
- \(A \rightarrow (B, C),\)
- \((B', C) \rightarrow A (B, C),\)
- \((B', C') \rightarrow (B, C) A,\)

where \(A \in N, B, B' \in N_t, C, C' \in N_r, a \in \Sigma\). A first additional requirement is that if \((B', C) \rightarrow A (B, C)\) is a rule, then \((B', C') \rightarrow A (B, C')\), for any \(C' \in N_r\), is also a rule, and if \((B, C') \rightarrow (B, C) A\) is a rule, then \((B', C') \rightarrow (B', C) A\), for any \(B' \in N_t\), is also a rule. This justifies our notation of such rules in the remainder of this paper as \((B', \_) \rightarrow A (B, \_)\) and \((\_, C') \rightarrow (\_, C) A\), respectively. These two kinds of rules correspond to attachment of left and right children, respectively, in dependency parsing. Secondly, we require that there is precisely one rule \((B, C) \rightarrow a\) for each \(a \in \Sigma\).

\[
W_\ell(B, i, i) = \begin{cases} 1, & \text{if } (B, C) \rightarrow a_i \\ 0, & \text{otherwise} \end{cases}
\]

\[
W_{\ell}(C, i, i) = \begin{cases} 1, & \text{if } (B, C) \rightarrow a_i \\ 0, & \text{otherwise} \end{cases}
\]

\[
W_{\ell}(B, C, i, j) = \bigoplus_k W_{\ell}(B, i, k) \odot W_B(k, j + 1) \odot w(j, i)
\]

\[
W_{\ell}(C, i, j) = \bigoplus_k W_{\ell}(B, i, k) \odot W_C(k, j + 1) \odot w(i, j)
\]

Table 2: Weighted parsing, for an arbitrary semiring, with \(0 \leq i < j \leq n\)

Note that the additional requirements make the grammar explicitly “split” in the sense of Eisner and Satta (1999), Eisner (2000), and Johnson (2007). That is, the two processes of attaching left and right children, respectively, are independent, with rules \((B, C) \rightarrow a\) creating “initial states” \(B\) and \(C\), respectively, for these two processes. Rules of the form \(A \rightarrow (B, C)\) then combine the end results of these two processes, possibly placing constraints on allowable combinations of \(B\) and \(C\).

To bring out the relation between our subclass of CFGs and bilexical grammars, one could explicitly write \((B, C)(a) \rightarrow a, A(a) \rightarrow (B, C)(a), (B', \_) (b) \rightarrow A(a) (B, \_) (b), \text{and } (\_, C')(c) \rightarrow (\_, C)(c) A(a)\).

Purely symbolic parsing is extended to weighted parsing much as usual, except that instead of attaching weights to rules, we attach a score \(w(i, j)\) to each pair \((i, j)\), which is a potential edge. This can be done for any semiring. In the semiring we will first use, a value is either a non-negative integer or \(-\infty\). Further, \(w_1 \odot w_2 = \max(w_1, w_2)\) and \(w_1 \otimes w_2 = w_1 + w_2\) if \(w_1 \neq -\infty\) and \(w_2 \neq -\infty\) and \(w_1 \otimes w_2 = -\infty\) otherwise. Naturally, the identity element of \(\odot\) is \(0 = -\infty\) and the identity element of \(\otimes\) is \(1 = 0\).

Tabular weighted parsing can be realized following Eisner and Satta (1999). We assume the input is a string \(a_0 a_1 \cdots a_n \in \Sigma^*\), with \(a_0\) being the prospective root of a tree. Table 2 presents the cubic-time algorithm in the form of a system of recursive equations. With the semiring we chose above, \(W_\ell(B, i, j)\) represents the highest score of any right-most derivation of the form \((B, \_) \Rightarrow A_1(B_1, \_) \Rightarrow A_1 A_2(B_2, \_) \Rightarrow^*\)
\[(S, S) \rightarrow a \]
\[S \rightarrow (S, S) \]
\[\varepsilon S \rightarrow (\varepsilon S) S \]
\[(S, \varepsilon) \rightarrow S (S, \varepsilon) \]

Table 3: Grammar for projective dependency parsing, with \(\Sigma = \{a\} \) and \(N = N_t = N_r = \{S\} \).

\[S \]
\[\left\lfloor (S, S) \right\rfloor \]
\[\left\lfloor (S, S) \right\rfloor \]
\[\left\lfloor (S, S) \right\rfloor \]
\[\left\lfloor (S, S) \right\rfloor \]

Figure 1: Dependency structure and corresponding parse tree that encodes a computation of a shift-reduce parser.

\[A_1 \cdots A_m (B_{m-1}) \Rightarrow A_1 \cdots A_m (a_j) \Rightarrow a_i \cdots a_j, \text{ for some } m \geq 0, \text{ and } W_r (A, i, j) \text{ has symmetric meaning. Intuitively, } W_f (A, i, j) \text{ considers } a_j \text{ and its left dependents and } W_r (A, i, j) \text{ considers } a_i \text{ and its right dependents. A value } W_f (A, i, j), \text{ or } W_r (A, i, j), \text{ represents the highest score combining } a_i \text{ and its right dependents and } a_j \text{ and its left dependents, meeting in the middle at some } k, \text{ including also an edge from } a_i \text{ to } a_j, \text{ or from } a_j \text{ to } a_i, \text{ respectively.} \]

One may interpret the grammar in Table 3 as encoding all possible computations of a shift-reduce parser, and thereby all projective trees. As there is only one way to instantiate the underscores, we obtain rule \((S, S) \rightarrow (S, S) S\), which corresponds to reduce_left, and rule \((S, S) \rightarrow S (S, S)\), which corresponds to reduce_right.

Figure 1 presents a parse tree for the grammar and the corresponding dependency tree. Note that if we are not given a particular strategy, such as left-before-right, then the parse tree underspecifies whether left children or right children are attached first. This is necessarily the case because the grammar is split. Therefore, the computation in this example may consist of three shifts, followed by one reduce_left, one reduce_right, and one reduce_left, or it may consist of two shifts, one reduce_right, one shift, and two reduce_lefts.

\[(P, P) \rightarrow p \]
\[(S, S) \rightarrow s \]
\[P \rightarrow (P, P) \]
\[S \rightarrow (P, S) \]
\[(S, \varepsilon) \rightarrow P (S, \varepsilon) \]
\[(S, \varepsilon) \rightarrow S (S, \varepsilon) \]
\[(\varepsilon, S) \rightarrow (\varepsilon, P) S \]
\[(\varepsilon, S) \rightarrow (\varepsilon, S) S \]
\[(S, \varepsilon) \rightarrow P (P, \varepsilon) \]

Table 4: Grammar for dependency parsing of \(p^k s^{m+1}\), representing a stack of length \(k + 1\) and remaining input of length \(m\), with \(\Sigma = \{p, s\}, \) \(N = N_r = N_t = \{P, S\} \). The last rule would be excluded for the strict left-to-right strategy, or alternatively one can set \(w(i, j) = -\infty\) for \(j < i < k\).

For a given gold tree \(T_g\), which may or may not be projective, we let \(w(i, j) = \delta_g (i, j)\), where we define \(\delta_g (i, j) = 1\) if \((i, j) \in T_g\) and \(\delta_g (i, j) = 0\) otherwise. With the grammar from Table 3, the value \(W\) found by weighted parsing is now the score of the most accurate projective tree. By backtracing from \(W\) as usual, we can construct the (or more correctly, a) tree with that highest accuracy. We have thereby found an effective way to projectivize a treebank in an optimal way. By a different semiring, we can count the number of trees with the highest accuracy, which reflects the degree of “choice” when projectivizing a treebank.

4 \(\mathcal{O}(n^3)\) Time Algorithm

In a computation starting from a configuration \((a_0 \cdots a_k b_1 \cdots b_m, T)\), not every projective parse of the string \(a_0 \cdots a_k b_1 \cdots b_m\) is achievable. The structures that are achievable are captured by the grammar in Table 4, with \(P\) for prefix and \(S\) for suffix (also for “start symbol”). Nonterminals \(P\) and \((P, P)\) correspond to a node \(a_i (0 \leq i < k)\) that does not have children. Nonterminal \(S\) corresponds to a node that has either \(a_k\) or some \(b_j (1 \leq j \leq m)\) among its descendants. This then means that the node will appear on top of the stack at some point in the computation. Nonterminal \((S, S)\) also corresponds to a node that has one of the rightmost \(m + 1\) nodes among its descendants, and, in addition, if it itself is not one of the rightmost \(m + 1\) nodes, then it must have a left child.
Nonterminal \((P, S)\) corresponds to a node \(a_i\) \((0 \leq i < k)\) that has \(a_k\) among its descendants but that does not have a left child. Nonterminal \((S, P)\) corresponds to a node \(a_i\) \((0 \leq i < k)\) that has a left child but no right children. For \(a_i\) to be given a left child, it is required that it eventually appear on top of the stack. This requirement is encoded in the absence of a rule with right-hand side \((S, P)\). In other words, \((S, P)\) cannot be part of a successful derivation, unless the rule \((S, S) \Rightarrow (S, P)\) \(S\) is subsequently used, which then corresponds to giving \(a_i\) a right child that has \(a_k\) among its descendants.

Figure 2 shows an example. Note that we can partition a parse tree into “columns”, each consisting of a path starting with a label in \(N\), then a series of labels in \(N_t \times N_r\) and ending with a label in \(\Sigma\).

A dependency structure that is not achievable, and that appropriately does not correspond to a parse tree, for a stack of height 4 and remaining input of length 1, is:

\[
\begin{array}{cccc}
\cdot & \cdot & \cdot & \cdot \\
\end{array}
\]

Suppose we have a configuration \((a_0 \cdots a_k, b_1 \cdots b_m, T)\) for sentence length \(n\), which implies \(k + m \leq n\). We need to decide whether a shift, reduce_left, or reduce_right should be done in order to achieve the highest accuracy, for given gold tree \(T_g\). For this, we calculate three values \(\sigma_1\), \(\sigma_2\) and \(\sigma_3\), and determine which is highest.

The first value \(\sigma_1\) is obtained by investigating the configuration \((a_0 \cdots a_k b_1, b_2 \cdots b_m, \emptyset)\) resulting after a shift. We run our generic tabular algorithm for the grammar in Table 4, for input \(p^{k+1}s^m\), to obtain \(\sigma_1 = W\). The scores are obtained by translating indices of \(a_0 \cdots a_k b_1 \cdots b_m = c_0 \cdots c_{k+m}\) to indices in the original input, that is, we let \(w(i, j) = \delta_g(c_i, c_j)\). However, the shift, which pushes an element on top of \(a_k\), implies that \(a_k\) will obtain right children before it can obtain left children. If we assume the left-before-right strategy, then we should avoid that \(a_k\) obtains left children. We could do that by refining the grammar, but find it easier to set \(w(k, i) = -\infty\) for all \(i < k\).

For the second value \(\sigma_2\), we investigate the configuration \((a_0 \cdots a_{k-1}, b_1 \cdots b_m, \emptyset)\) resulting after a reduce_left. The same grammar and algorithm are used, now for input \(p^{k-1}s^{m+1}\). With \(a_0 \cdots a_{k-1} b_1 \cdots b_m = c_0 \cdots c_{k+m-1}\), we let \(w(i, j) = \delta_g(c_i, c_j)\). We let \(\sigma_2 = W \oplus \delta_g(a_k, a_k)\). In case of a strict left-before-right strategy, we set \(w(k-1, i) = -\infty\) for \(i < k - 1\), to avoid that \(a_{k-1}\) obtains left children after having obtained a right child \(a_k\).

If \(k \leq 1\) then the third value is \(\sigma_3 = -\infty\), as no reduce_right is applicable. Otherwise we investigate \((a_0 \cdots a_{k-2}, a_k, b_1 \cdots b_m, \emptyset)\). The same grammar and algorithm are used as before, and \(w(i, j) = \delta_g(c_i, c_j)\) with \(a_0 \cdots a_{k-2} a_k b_1 \cdots b_m = c_0 \cdots c_{k+m-1}\). Now \(\sigma_3 = W \oplus \delta_g(a_k, a_{k-1})\). In case of a right-before-left strategy, we set \(w(k, i) = -\infty\) for \(k < i\).

We conclude that the time complexity of calculating the optimal step is three times the time complexity of the algorithm of Table 2, hence cubic in \(n\).

For a proof of correctness, it is sufficient to show that each parse tree by the grammar in Table 4 corresponds to a computation with the same score, and conversely that each computation corresponds to an equivalent parse tree. Our grammar has spurious ambiguity, just as the shift-reduce parser from Table 1, and this can be resolved in the same way, depending on whether the intended strategy is (non-)strict left-before-right or right-before-left, and whether the configuration is the result of a shift, reduce_left, or reduce_right. Concretely, we can restrict parse trees to attach children lower in the tree if they would be attached earlier in the computation, and thereby we obtain
a bijection between parse trees and computations. For example, in the middle column of the parse tree in Figure 2, the \((P, S)\) and its right child occur below the \((S, S)\) and its left child, to indicate the reduce_left precedes the reduce_right.

The proof in one direction assumes a parse tree, which is traversed to gather the steps of a computation. This traversal is post-order, from left to right, but skipping the nodes representing stack elements below the top of the stack, starting from the leftmost node labeled \(s\). Each node \(\nu\) with a label in \(N_L \times N_r\) corresponds to a step. If the child of \(\nu\) is labeled \(s\), then we have a shift, and if it has a right or left child with a label in \(N\), then it corresponds to a reduce_left or reduce_right, respectively. The configuration resulting from that step can be constructed as sketched in Figure 3. We follow the shortest path from \(\nu\) to the root. All the leaves to the right of the path correspond to the remaining input. For the stack, we gather the leaves in the columns of the nodes on the path, as well as those of the left children of nodes on the path. Compare this with the concept of right-sentential forms in the theory of context-free parsing.

For a proof in the other direction, we can make use of existing parsing theory, which tells us how to translate a computation of the shift-reduce parser to a dependency structure, which in turn is easily translated to an undecorated parse tree. It then remains to show that the nodes in that tree can be decorated (in fact in a unique way), according to the rules from Table 4. This is straightforward given the meanings of \(P\) and \(S\) described earlier in this section. Most notably, the absence of a rule with right-hand side \((\ldots, P)\) \(P\) does not prevent the decoration of a tree that was constructed out of a computation, because a reduction involving two nodes within the stack is only possible if the rightmost of these nodes eventually appears on top of the stack, which is only possible when the computation has previously made \(a_k\) a descendant of that node, hence we would have \(S\) rather than \(P\).

5 \(O(n^2)\) Time Algorithm

Assume a given configuration \((\alpha, \beta, T)\) as before, resulting from a shift or reduction. Let \(\alpha = a_0 \cdots a_k\), \(A = \{a_0, \ldots, a_k\}\), and let \(B\) be the set of nodes in \(\beta\). We again wish to calculate the maximum value of \(|T' \cap T_g|\) for any \(T'\) such that \((\alpha, \beta, \emptyset) \vdash^* (0, \varepsilon, T')\), but now under the assumption that \(T_g\) is projective. Let us call this value \(\sigma_{\text{max}}\). We define \(w\) in terms of \(\delta_g\) as in the previous section, setting \(w(i, j) = -\infty\) for an appropriate subset of pairs \((i, j)\) to enforce a strategy that is (non-)strict left-before-right or right-before-left.

The edges in \(T_g \cap (B \times B)\) partition the remaining input into maximal connected components. Within these components, a node \(b \in B\) is called critical if it satisfies one or both of the following two conditions:

- At least one descendant of \(b\) (according to \(T_g\)) is not in \(B\).
- The parent of \(b\) (according to \(T_g\)) is not in \(B\).

Let \(B_{\text{crit}} \subseteq B\) be the set of critical nodes, listed in order as \(b_1, \ldots, b_m\), and let \(B_{\text{ncrit}} = B \setminus B_{\text{crit}}\). Figure 4 sketches three components as well as edges in \(T_g \cap (A \times B)\) and \(T_g \cap (B \times A)\). Component \(C_1\), for example, contains the critical elements \(b_1, b_2,\) and \(b_3\). The triangles under \(b_1, \ldots, b_7\) represent subtrees consisting of edges leading to non-critical nodes. For each \(b \in B_{\text{crit}},\)

\[|T_g \cap \{b\} \times A|\]

is zero or more, or in words,
critical nodes have zero or more children in the stack. Further, if \((a, b) \in T_g \cap (A \times B_{\text{crit}})\), then \(b\) is the rightmost critical node in a component; examples are \(b_5\) and \(b_7\) in the figure.

Let \(T_{\text{max}}\) be any tree such that \((\alpha, \beta, \emptyset) \vdash^\ast (0, \varepsilon, T_{\text{max}})\) and \([T_{\text{max}} \cap T_g] = \sigma_{\text{max}}\). Then we can find another tree \(T'_{\text{max}}\) that has the same properties and in addition satisfies:

1. \(T_g \cap (B \times B_{\text{ncrit}}) \subseteq T'_{\text{max}}\),
2. \(T'_{\text{max}} \cap (B_{\text{ncrit}} \times A) = \emptyset\),
3. \(T'_{\text{max}} \cap (B \times B_{\text{crit}}) \subseteq T_g\),

or in words, (1) the subtrees rooted in the critical nodes are entirely included, (2) no child of a non-critical node is in the stack, and (3) within the remaining input, all edges to critical nodes are gold. Very similar observations were made before by Goldberg et al. (2014), and therefore we will not give full proofs here. The structure of the proof is in each case that all violations of a property can be systematically removed, by rearranging the computation, in a way that does not decrease the score.

We need two more properties:

4. If \((a, b) \in T'_{\text{max}} \cap (A \times B_{\text{crit}}) \setminus T_g\) then either:
   - \(b\) is the rightmost critical node in its component, or
   - there is \((b, a') \in T'_{\text{max}} \cap T_g\), for some \(a' \in A\) and there is at least one other critical node \(b'\) to the right of \(b\), but in the same component, such that \((b', a'') \in T'_{\text{max}} \cap T_g\) or \((a'', b') \in T'_{\text{max}} \cap T_g\), for some \(a'' \in A\).

5. If \((b, a) \in T'_{\text{max}} \cap (B_{\text{crit}} \times A) \setminus T_g\) then there is \((b, a') \in T'_{\text{max}}\) for some \(a' \in A\), such that \(a'\) is a sibling of \(a\) immediately to its right.

Figure 5, to be discussed in more detail later, illustrates property (4) for the non-gold edge from \(a_4\); this edge leads to \(b_4\) (which has outgoing gold edge to \(a_5\)) rather than to \(b_5\) or \(b_6\). It further respects property (4) because of the gold edges connected to \(b_7\) and \(b_6\), which occur to the right of \(b_1\) but in the same component. Property (5) is illustrated for the non-gold edge from \(b_6\) to \(a_8\), which has sibling \(a_9\) immediately to the right.

The proof that property (4) may be assumed to hold, without loss of generality, again involves making local changes to the computation, in particular replacing the \(b\) in an offending non-gold edge \((a, b) \in A \times B_{\text{crit}}\) by another critical node \(b'\) further to the left or at the right end of the component. Similarly, for property (5), if we have an offending non-gold edge \((b, a)\), then we can rearrange the computation, such that node \(a\) is reduced not into \(b\) but into one of the descendants of \(b\) in \(B\) that was given children in \(A\). If none of the descendants of \(b\) in \(B\) was given children in \(A\), then \(a\) can instead be reduced into its neighbor in the stack immediately to the left, without affecting the score.

By properties (1)–(3), we can from here on ignore non-critical nodes, so that the remaining task is to calculate \(\sigma_{\text{max}} - |B_{\text{crit}}|\). In fact, we go further than that and calculate \(\sigma_{\text{max}} - M\), where \(M = |T_g \cap (B \times B)|\). In other words, we take for granted that the score can be at least as much as the number of gold edges within the remaining input, which leaves us with the task of counting the additional gold edges in the optimal computation. For any given component \(C\) we can consider the sequence of edges that the computation creates between \(A\) and \(C\), in the order in which they are created:

- for the first gold edge between \(C\) and \(A\), we count +1,
- for each subsequent gold edge between \(C\) to \(A\), we count +1,
- we ignore interspersed non-gold edges from \(C\) to \(A\),
- but following a non-gold edge from \(A\) to \(C\), the immediately next gold edge between \(C\) and \(A\) is not counted, because that non-gold edge implies that another gold edge in \(B_{\text{crit}} \times B_{\text{crit}}\) cannot be created.

This is illustrated by Figure 5. For \((b_3, a_9)\) we count +1, it being the first gold edge connected

![Figure 5](image-url)
to the component. For the subsequent three gold edges, we count +1 for each, ignoring the non-gold edge \((b_3, a_3)\). The non-gold edge \((a_4, b_1)\) implies that the parent of \(b_1\) is already determined.

One would then perhaps expect \(nchildren(\cdot)\) to be a function of \(\tau(i,j)\), such that \(nchildren(\cdot)\) is a function of \(a_{i+1}\) and \(b_j\). However, \(nchildren(\cdot)\) counts the number of children of \(a_{i+1}\) and \(b_j\) in the component. We need to distinguish \(\tau(i,j)\) from \(nchildren(\cdot)\).

The main motivation for properties (1)–(5) is that they limit the input positions that can be relevant for a node that is on top of the stack, thereby eliminating one factor \(m\) from the time complexity. More specifically, the gold edges relate a stack element to a “current critical node”, in a “current component”. We need to distinguish between three possible states:

- \(\mathcal{N}\) (none): none of the critical nodes from the current component were shifted on to the stack yet,
- \(\mathcal{C}\) (consumed): the current critical node was “consumed” by it having been shifted and assigned a parent,
- \(\mathcal{F}\) (fresh): the current critical node was not consumed, but at least one of the preceding critical nodes in the same component was consumed.

For \(0 \leq i \leq k\), we define \(p(i)\) to be the index \(j\) such that \((b_j, a_i) \in T_g\), and if there is no such \(j\), then \(p(i) = \bot\), where \(\bot\) denotes “undefined”.

For \(0 \leq i < k\), we let \(p_{\geq}(i) = p(i)\) if \(p(i) \neq \bot\), and \(p_{\geq}(i) = p_{\geq}(i + 1)\) otherwise, and further \(p_{\geq}(k) = p(k)\). Intuitively, we seek a critical node that is the parent of \(a_i\), or if there is none, of \(a_{i+1}\). We define \(c(i)\) to be the smallest \(j\) such that \((a_i, b_j) \in T_g\), or in words, the index of the leftmost child in the remaining input, and \(c(i) = \bot\) if there is none.

As representative element of a component with critical element \(b_j\) we take the critical element that is rightmost in that component, or formally, we define \(R(j)\) to be the largest \(j'\) such that \(b_{j'}\) is an ancestor (by \(T_g \cap (B_{crit} \times B_{crit})\)) of \(b_j\). For completeness, we define \(R(\bot) = \bot\). We let \(P(i) = R(p(i))\) and \(P_{\geq}(i) = R(p_{\geq}(i))\). Note that \(\text{score}(i, j, q) = 0\) if \(i < 0\), otherwise:

\[
\text{score}(i, j, q) = \begin{cases} \\
\text{nchildren}(i) - \Delta(c(i) = P_j(j) \land q \neq \mathcal{N})] & \land \left( w(i, j) \otimes \text{score}(i - 1, 1, \tau(i, j, q)) \right. \\
\text{w}(j, i) \otimes \text{score}(i - 1, j, q) \oplus & \left. \text{if } p(j) = \bot \lor q = \mathcal{C} \text{ then } -\infty \right. \\
\text{else } \Delta(q = \mathcal{N}) \otimes \text{score}(i, p(j)) \right. \\
\text{score}'(i, j) = 0 \text{ if } i < 0, \text{ otherwise:} & \text{else} \right.
\end{cases}
\]

\[
\text{nchildren}(i) = \left[ \begin{array}{l} \text{if } p'(i, j) = \bot \text{ then } \text{score}'(i - 1, j) \\
\text{else } 1 \otimes \text{score}'(i - 1, p'(i, j)) \end{array} \right] \oplus \\
\text{nchildren}(i) \otimes \text{score}(i - 1, i, \tau'(i, j))
\]

\[
\tau(i, j, q) = \begin{cases} \\
\text{if } q = \mathcal{N} \lor P_{\geq}(i) \neq P_{\geq}(j) \text{ then } \mathcal{N} \\
\text{else if } p_{\geq}(i) \neq p_{\geq}(j) \text{ then } \mathcal{F} \\
\text{else } q \end{cases}
\]

Table 5: Quadratic-time algorithm.
after a reduce_right we would preclude right children of \(a_k\) by setting \(w(k, i) = -\infty\) for \(k < i\). The leftmost of the children, at index \(c(i)\), is not counted (or in other words, 1 is subtracted from the number of children) if it is in the current component \(P_2(j)\) and that component is anything other than ‘none’; here \(\Delta\) is the indicator function, which returns 1 if its Boolean argument evaluates to true, and 0 otherwise. Figure 6 illustrates one possible case.

The second value corresponds to a reduction of \(a_i\) into \(a_j\), which turns the stack into \(a_0 \cdots a_{i-1} a_j\), leaving the state unchanged as the top of the stack is unchanged. The third value is applicable if \(q = F\), assuming \(c(i) = P_2(j) = \ell\) and \(a_i\) further has children \(b_{\ell+1}\) and \(b_{\ell+2}\). Because \(q = F\), there was some other node \(b_{\ell'}\) in the same component that was shifted on to the stack earlier and given a (non-gold) parent; let us assume \(\ell' = \ell - 1\). We can add 3 children to the score, but should subtract \(\Delta(c(i) = P_2(j) \land q \neq N) = 1\), to compensate for the fact that edge \((b_{\ell'}, b_{\ell-1})\) cannot be constructed, as \(b_{\ell-1}\) can only have one parent. If we further assume \(a_i\) has a parent among the critical nodes, then that parent must be in a different component, and therefore \(\tau(i, j, q) = N\).

The top-level call is \(\text{score}(k - 1, k, N)\). As this does not account for right children of the top of stack \(a_k\), we need to add \(\text{nchildren}(k)\). Putting everything together, we have \(\sigma_{\text{max}} = M \otimes \text{score}(k - 1, k, N) \otimes \text{nchildren}(k)\). The time complexity is quadratic in \(k + m \leq n\), given the quadratically many combinations of \(i\) and \(j\) in \(\text{score}(i, j, q)\).

### 6 \(O(n)\) Time Algorithm

Under the same assumption as in the previous section, namely, that \(T_g\) is projective, we can further reduce the time complexity of computing \(\sigma_{\text{max}}\) by two observations. First, let us define \(\lambda(i, j)\) to be true if and only if there is an \(\ell < i\) such that \((a_i, a_j) \in T_g\) or \((a_j, a_i) \in T_g\). If \((a_j, a_i) \notin T_g\) and \(\lambda(i, j)\) is false, then the highest score attainable from a configuration \((a_0 \cdots a_{i-1} a_j, \beta, \emptyset)\) is no higher than the highest score attainable from \((a_0 \cdots a_{i-1} a_j, \beta, \emptyset)\), or, if \(a_j\) has a parent \(b_{\ell'}\), from \((a_0 \cdots a_{j-b_{\ell'}}, \beta', \emptyset)\), for appropriate suffix \(\beta'\) of \(\beta\). This means that in order to calculate \(\text{score}(i, j, q)\) we do not need to calculate \(\text{score}(i - 1, j, q)\) in this case.

Secondly, if \((a_j, a_i) \notin T_g\) and \(\lambda(i, j)\) is true, and if there is \(\ell' < i\) such that \((a_{\ell'}, a_i) \in T_g\) or \((a_i, a_{\ell'}) \in T_g\), then there are no edges between \(a_j\) and \(a_{\ell'}\) for any \(i'\) with \(\ell' < i' < i\), because of projectivity of \(T_g\). We therefore do not need to calculate \(\text{score}(i', j, q)\) for such values of \(i'\) in order to find the computation with the highest score. This is illustrated in Figure 7.
Let us define \( \kappa(i) \) to be the smallest \( \ell' \) such that \((a_{\ell'}, a_i) \in T_g\) or \((a_i, a_{\ell'}) \in T_g\), or \(i - 1\) if there is no such \( \ell' \). In the definition of \( \text{score} \), we may now replace \( w(j, i) \otimes \text{score}(i - 1, j, q) \) by:

\[
\begin{cases}
  \text{if } w(j, i) = 1 & \text{then } 1 \otimes \text{score}(i - 1, j, q) \\
  \text{else if } w(j, i) = 0 \land \lambda(i, j) & \text{then } \text{score}(\kappa(i), j, q) \\
  \text{else } -\infty
\end{cases}
\]

Similarly, we define \( \lambda'(i, j) \) to be true if and only if there is an \( \ell < i \) such that \((a_\ell, b_j') \in T_g\) or \((b_j, a_\ell) \in T_g\) for some \( j' \) with \( R(j') = R(j) \). In the definition of \( \text{score}' \), we may now replace \( \text{score}'(i - 1, j) \) by:

\[
\text{if } \lambda'(i, j) \text{ then } \text{score}'(\kappa(i), j) \text{ else } -\infty
\]

Thereby the algorithm becomes linear-time, because the number of values \( \text{score}(i, j, q) \) and \( \text{score}'(i, j) \) that are calculated for any \( i \) is now linear. To see this, consider that for any \( i \), \( \text{score}(i, j, q) \) would be calculated only if \( j = i + 1 \), if \((a_i, a_j) \in T_g\) or \((a_j, a_i) \in T_g\), if \((a_j, a_{i+1}) \in T_g\), or if \( j \) is smallest such that there is \( \ell < i \) with \((a_\ell, a_j) \in T_g\) or \((a_j, a_\ell) \in T_g\). Similarly, \( \text{score}(i, j') \) would be calculated only if \( \text{score}(i, j', q) \) would be calculated and \((b_j, a_{j'}) \in T_g\), if \((b_j, a_{i+1}) \in T_g\), or if \( j \) is smallest such that there is \( \ell \leq i \) with \((a_\ell, b_j) \in T_g\) or \((b_j, a_\ell) \in T_g\) for some \( j' \) such that \( b_j \) an ancestor of \( b_j \) in the same component.

7 Towards Constant Time Per Calculation

A typical application would calculate the optimal step for several or even all configurations within one computation. Between one configuration and the next, the stack differs at most in the two rightmost elements and the remaining input differs at most in that it loses its leftmost element. Therefore, all but a constant number of values of \( \text{score}(i, j, q) \) and \( \text{score}'(i, j) \) can be reused, to make the time complexity closer to constant time for each calculation of the optimal step. The practical relevance of this is limited however if one would typically reload the data structures containing the relevant values, which are of linear size. Hence we have not pursued this further.

8 Experiments

Our experiments were run on a laptop with an Intel i7-7500U processor (4 cores, 2.70 GHz) with 8 GB of RAM. The implementation language is Java, with DL4J\(^2\) for the classifier, realized as a neural network with a single layer of 256 hidden nodes. Training is with batch size 100, and 20 epochs. Features are the (gold) parts of speech and length-100 word2vec representations of the word forms of the top-most three stack elements, as well as of the left-most three elements of the remaining input, and the left-most and right-most dependency relations in the top-most two stack elements.

8.1 Optimal Projectivization

We need to projectivize our training corpus for the experiments in Section 8.2, using the algorithm described at the end of Section 3. As we are not aware of literature reporting experiments with optimal projectivization, we briefly describe our findings here.

Projectivizing all the training sets in Universal Dependencies v2.2\(^3\) took 244 sec in total, or 0.342 ms per tree. As mentioned earlier, there may be multiple projectivized trees that are optimal in terms of accuracy, for a single gold tree. We are not aware of meaningful criteria that tell us how to choose any particular one of them, and for our experiments in Section 8.2 we have chosen an arbitrary one. It is conceivable, however, that the choices of the projectivized trees would affect the accuracy of a parser trained on them. Figure 8 illustrates the degree of “choice” when projectiving trees. We consider

\(^2\)https://deeplearning4j.org/.
\(^3\)https://universaldependencies.org/.
Table 6: Accuracy (LAS or UAS, which here are identical) of pseudo-projectivization and of optimal projectivization.

| language | pseudo | optimal |
|----------|--------|---------|
| grc      | 91.41  | 92.50   |
| de       | 98.89  | 98.97   |
| ja       | 99.99  | 99.99   |

two languages that are known to differ widely in the prevalence of non-projectivity, namely Ancient Greek (PROIEL) and Japanese (BCCWJ), and we consider one more language, German (GSD), that falls in between (Straka et al., 2015).

As can be expected, the degree of choice grows roughly exponentially in sentence length.

Table 6 shows that pseudo-projectivization is non-optimal. We realized pseudo-projectivization using MaltParser 1.9.0.4

8.2 Computing the Optimal Step

To investigate the run-time behavior of the algorithms, we trained our shift-reduce dependency parser on the German training corpus, after it was projectivized as in Section 8.1. In a second pass over the same corpus, the parser followed the steps returned by the trained classifier. For each configuration that was obtained in this way, the running time was recorded of calculating the optimal step, with the non-strict left-before-right strategy. For each configuration, it was verified that the calculated scores, for \texttt{shift}, \texttt{reduce\_left}, and \texttt{reduce\_right}, were the same between the three algorithms from Sections 4, 5, and 6.

The two-pass design was inspired by Choi and Palmer (2011). We chose this design, rather than online learning, as we found it easiest to implement. Goldberg and Nivre (2012) discuss the relation between multi-pass and online learning approaches.

As Figure 9 shows, the running times of the algorithms from Sections 5 and 6 grow slowly as the summed length of stack and remaining input grows; note the logarithmic scale. The improvement of the linear-time algorithm over the quadratic-time algorithm is perhaps less than one may expect. This is because the calculation of the critical nodes and the construction of the necessary tables, such as \( p, p' \), and \( R \), is considerable compared to the costs of the memoized recursive calls of \texttt{score} and \texttt{score}'

Both these algorithms contrast with the algorithm from Section 4, applied on projectivized trees as above (hence tagged \texttt{proj} in Figure 9), and with the remaining input simplified to just its critical nodes. For \( k + m = 80 \), the cubic-time algorithm is slower than the linear-time algorithm by a factor of about 65. Nonetheless, we find that the cubic-time algorithm is practically relevant, even for long sentences.

The decreases at roughly \( k + m = 88 \), which are most visible for Section 4 (proj), are explained by the fact that the running time is primarily determined by \( k + m' \), where \( m' \) is the number of critical nodes. Because \( k + m \) is bounded by the sentence length and the stack height \( k \) tends to be much less than the sentence length, high values of \( k + m \) tend to result from the length \( m \) of the remaining input being large, which in turn implies that there will be more non-critical nodes that are removed before the most time-consuming part of the analyses is entered. This is confirmed by Figure 10.
The main advantage of the cubic-time algorithm is that it is also applicable if the training corpus has not been projectivized. To explore this we have run this algorithm on the same corpus again, but now without projectivization in the second pass (for training the classifier in the first pass, projectivization was done as before). In this case, we can no longer remove non-critical nodes (without it affecting correctness), and now the curve is monotone increasing, as shown by Section 4 (unproj) in Figure 9. Nevertheless, with mean running times below 0.25 sec even for input longer than 100 tokens, this algorithm is practically relevant.

8.3 Accuracy

If a corpus is large enough for the parameters of a classifier to be reliably estimated, or if the vast majority of trees is projective, then accuracy is not likely to be much affected by the work in this paper. We therefore also consider six languages that have some of the smallest corpora in UD v2.2 in combination with a relatively large proportion of non-projective trees: Danish, Basque, Greek, Old Church Slobonic, Gothic, and Hungarian. For these languages, Table 7 shows that accuracy is generally higher if training can benefit from all trees. In a few cases, it appears to be slightly better to train directly on non-projective trees rather than on optimally projectivized trees.

9 Conclusions

We have presented the first algorithm to calculate the optimal step for shift-reduce dependency parsing that is applicable on non-projective training corpora. Perhaps even more innovative than its functionality is its modular architecture, which implies that the same is possible for related kinds of parsing, as long as the set of allowable transitions can be described in terms of a split context-free grammar. The application of the framework to, among others, arc-eager dependency parsing is to be reported elsewhere.

We have also shown that calculation of the optimal step is possible in linear time if the training corpus is projective. This is the first time this has been shown for a form of projective, deterministic dependency parsing that does not have the property of arc-decomposibility.

Acknowledgments

The author wishes to thank the reviewers for comments and suggestions, which led to substantial improvements.

References

Lauriane Aufrant, Guillaume Wisniewski, and François 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, LA.

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

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

Daniel Fernández-González and Carlos Gómez-Rodríguez. 2018a. 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, LA.

Daniel Fernández-González and Carlos Gómez-Rodríguez. 2018b. Non-projective dependency parsing with non-local transitions. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, volume 2, pages 693–700. New Orleans, LA.

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

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

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

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

Carlos Gómez-Rodríguez, John Carroll, and David Weir. 2008. A deductive approach to dependency parsing. In *46th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 968–976. Columbus, OH.

Carlos Gómez-Rodríguez and Daniel 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.

Carlos Gómez-Rodríguez, Francesco Sartorio, and Giorgio 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.

Liang Huang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1077–1086. Uppsala.

Mark 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.

Sylvain Kahane, Alexis Nasr, and Owen Rambow. 1998. Pseudo-projectivity, a polynomially parsable non-projective dependency grammar. In *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics*, volume 1, pages 646–652. Montreal.

Martin Kay. 2000. Guides and oracles for linear-time parsing. In *Proceedings of the Sixth International Workshop on Parsing Technologies*, pages 6–9. Trento.

Marco Kuhlmann, Carlos Gómez-Rodríguez, and Giorgio 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, OR.

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

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

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

Joakim Nivre and Jens Nilsson. 2005. Pseudo-projective dependency parsing. In *43rd Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pages 99–106. Ann Arbor, MI.

Peng Qi and Christopher 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.

Milan Straka, Jan Hajič, Jana Straková, and Jan 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.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *8th International Workshop on Parsing Technologies*, pages 195–206. LORIA, Nancy.