Algorithms for Deterministic Incremental Dependency Parsing

Joakim Nivre*,**
Växjö University, Uppsala University

Parsing algorithms that process the input from left to right and construct a single derivation have often been considered inadequate for natural language parsing because of the massive ambiguity typically found in natural language grammars. Nevertheless, it has been shown that such algorithms, combined with treebank-induced classifiers, can be used to build highly accurate disambiguating parsers, in particular for dependency-based syntactic representations. In this article, we first present a general framework for describing and analyzing algorithms for deterministic incremental dependency parsing, formalized as transition systems. We then describe and analyze two families of such algorithms: stack-based and list-based algorithms. In the former family, which is restricted to projective dependency structures, we describe an arc-eager and an arc-standard variant; in the latter family, we present a projective and a non-projective variant. For each of the four algorithms, we give proofs of correctness and complexity. In addition, we perform an experimental evaluation of all algorithms in combination with SVM classifiers for predicting the next parsing action, using data from thirteen languages. We show that all four algorithms give competitive accuracy, although the non-projective list-based algorithm generally outperforms the projective algorithms for languages with a non-negligible proportion of non-projective constructions. However, the projective algorithms often produce comparable results when combined with the technique known as pseudo-projective parsing. The linear time complexity of the stack-based algorithms gives them an advantage with respect to efficiency both in learning and in parsing, but the projective list-based algorithm turns out to be equally efficient in practice. Moreover, when the projective algorithms are used to implement pseudo-projective parsing, they sometimes become less efficient in parsing (but not in learning) than the non-projective list-based algorithm. Although most of the algorithms have been partially described in the literature before, this is the first comprehensive analysis and evaluation of the algorithms within a unified framework.

1. Introduction

Because parsers for natural language have to cope with a high degree of ambiguity and nondeterminism, they are typically based on different techniques than the ones used for parsing well-defined formal languages—for example, in compilers for

* School of Mathematics and Systems Engineering, Växjö University, 35195 Växjö, Sweden.
E-mail: joakim.nivre@vxu.se.
** Department of Linguistics and Philology, Uppsala University, Box 635, 75126 Uppsala, Sweden.
E-mail: joakim.nivre@lingfil.uu.se.

Submission received: 29 May 2007; revised submission received 22 September 2007; accepted for publication: 3 November 2007.

programming languages. Thus, the mainstream approach to natural language parsing uses algorithms that efficiently derive a potentially very large set of analyses in parallel, typically making use of dynamic programming and well-formed substring tables or charts. When disambiguation is required, this approach can be coupled with a statistical model for parse selection that ranks competing analyses with respect to plausibility. Although it is often necessary, for efficiency reasons, to prune the search space prior to the ranking of complete analyses, this type of parser always has to handle multiple analyses.

By contrast, parsers for formal languages are usually based on deterministic parsing techniques, which are maximally efficient in that they only derive one analysis. This is possible because the formal language can be defined by a non-ambiguous formal grammar that assigns a single canonical derivation to each string in the language, a property that cannot be maintained for any realistically sized natural language grammar. Consequently, these deterministic parsing techniques have been much less popular for natural language parsing, except as a way of modeling human sentence processing, which appears to be at least partly deterministic in nature (Marcus 1980; Shieber 1983).

More recently, however, it has been shown that accurate syntactic disambiguation for natural language can be achieved using a pseudo-deterministic approach, where treebank-induced classifiers are used to predict the optimal next derivation step when faced with a nondeterministic choice between several possible actions. Compared to the more traditional methods for natural language parsing, this can be seen as a severe form of pruning, where parse selection is performed incrementally so that only a single analysis is derived by the parser. This has the advantage of making the parsing process very simple and efficient but the potential disadvantage that overall accuracy suffers because of the early commitment enforced by the greedy search strategy. Somewhat surprisingly, though, research has shown that, with the right choice of parsing algorithm and classifier, this type of parser can achieve state-of-the-art accuracy, especially when used with dependency-based syntactic representations.

Classifier-based dependency parsing was pioneered by Kudo and Matsumoto (2002) for unlabeled dependency parsing of Japanese with head-final dependencies only. The algorithm was generalized to allow both head-final and head-initial dependencies by Yamada and Matsumoto (2003), who reported very good parsing accuracy for English, using dependency structures extracted from the Penn Treebank for training and testing. The approach was extended to labeled dependency parsing by Nivre, Hall, and Nilsson (2004) (for Swedish) and Nivre and Scholz (2004) (for English), using a different parsing algorithm first presented in Nivre (2003). At a recent evaluation of data-driven systems for dependency parsing with data from 13 different languages (Buchholz and Marsi 2006), the deterministic classifier-based parser of Nivre et al. (2006) reached top performance together with the system of McDonald, Lerman, and Pereira (2006), which is based on a global discriminative model with online learning. These results indicate that, at least for dependency parsing, deterministic parsing is possible without a drastic loss in accuracy. The deterministic classifier-based approach has also been applied to phrase structure parsing (Kalt 2004; Sagae and Lavie 2005), although the accuracy for this type of representation remains a bit below the state of the art. In this setting, more competitive results have been achieved using probabilistic classifiers and beam search, rather than strictly deterministic search, as in the work by Ratnaparkhi (1997, 1999) and Sagae and Lavie (2006).

A deterministic classifier-based parser consists of three essential components: a parsing algorithm, which defines the derivation of a syntactic analysis as a sequence

of elementary parsing actions; a feature model, which defines a feature vector representation of the parser state at any given time; and a classifier, which maps parser states, as represented by the feature model, to parsing actions. Although different types of parsing algorithms, feature models, and classifiers have been used for deterministic dependency parsing, there are very few studies that compare the impact of different components. The notable exceptions are Cheng, Asahara, and Matsumoto (2005), who compare two different algorithms and two types of classifier for parsing Chinese, and Hall, Nivre, and Nilsson (2006), who compare two types of classifiers and several types of feature models for parsing Chinese, English, and Swedish.

In this article, we focus on parsing algorithms. More precisely, we describe two families of algorithms that can be used for deterministic dependency parsing, supported by classifiers for predicting the next parsing action. The first family uses a stack to store partially processed tokens and is restricted to the derivation of projective dependency structures. The algorithms of Kudo and Matsumoto (2002), Yamada and Matsumoto (2003), and Nivre (2003, 2006b) all belong to this family. The second family, represented by the algorithms described by Covington (2001) and recently explored for classifier-based parsing in Nivre (2007), instead uses open lists for partially processed tokens, which allows arbitrary dependency structures to be processed (in particular, structures with non-projective dependencies). We provide a detailed analysis of four different algorithms, two from each family, and give proofs of correctness and complexity for each algorithm. In addition, we perform an experimental evaluation of accuracy and efficiency for the four algorithms, combined with state-of-the-art classifiers, using data from 13 different languages. Although variants of these algorithms have been partially described in the literature before, this is the first comprehensive analysis and evaluation of the algorithms within a unified framework.

The remainder of the article is structured as follows. Section 2 defines the task of dependency parsing and Section 3 presents a formal framework for the characterization of deterministic incremental parsing algorithms. Sections 4 and 5 contain the formal analysis of four different algorithms, defined within the formal framework, with proofs of correctness and complexity. Section 6 presents the experimental evaluation; Section 7 reports on related work; and Section 8 contains our main conclusions.

2. Dependency Parsing

Dependency-based syntactic theories are based on the idea that syntactic structure can be analyzed in terms of binary, asymmetric dependency relations holding between the words of a sentence. This basic conception of syntactic structure underlies a variety of different linguistic theories, such as Structural Syntax (Tesnière 1959), Functional Generative Description (Sgall, Hajičová, and Panevová 1986), Meaning-Text Theory (Mel'čuk 1988), and Word Grammar (Hudson 1990). In computational linguistics, dependency-based syntactic representations have in recent years been used primarily in data-driven models, which learn to produce dependency structures for sentences solely from an annotated corpus, as in the work of Eisner (1996), Yamada and Matsumoto (2003), Nivre, Hall, and Nilsson (2004), and McDonald, Crammer, and Pereira (2005), among others. One potential advantage of such models is that they are easily ported to any domain or language in which annotated resources exist.

In this kind of framework the syntactic structure of a sentence is modeled by a *dependency graph*, which represents each word and its syntactic dependents through labeled directed arcs. This is exemplified in Figure 1, for a Czech sentence taken from the Prague

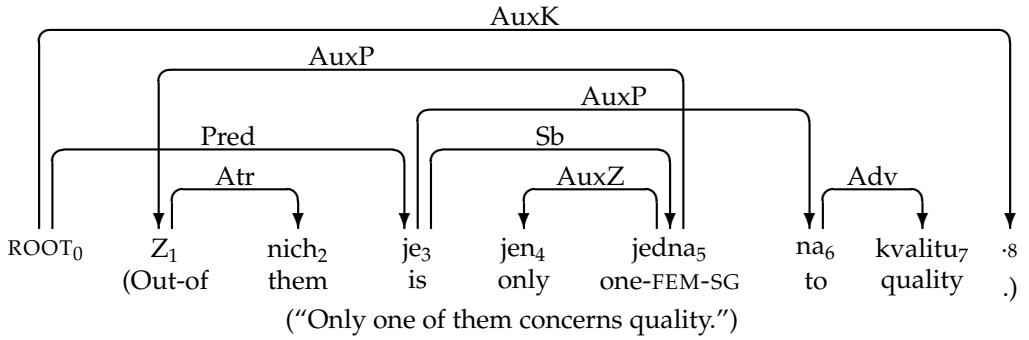

Figure 1
Dependency graph for a Czech sentence from the Prague Dependency Treebank.

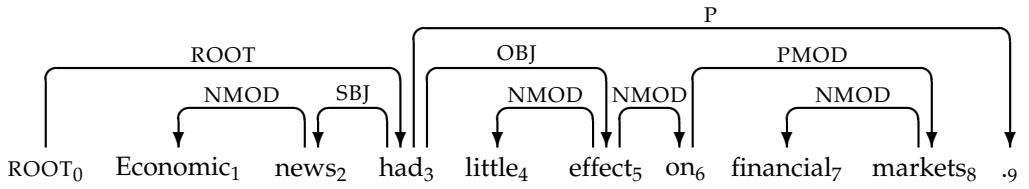

Figure 2
Dependency graph for an English sentence from the Penn Treebank.

Dependency Treebank (Hajič et al. 2001; Böhmová et al. 2003), and in Figure 2, for an English sentence taken from the Penn Treebank (Marcus, Santorini, and Marcinkiewicz 1993; Marcus et al. 1994).[1] An artificial word ROOT has been inserted at the beginning of each sentence, serving as the unique root of the graph. This is a standard device that simplifies both theoretical definitions and computational implementations.

Definition 1

Given a set $L = \{l_1, \ldots, l_{|L|}\}$ of dependency labels, a **dependency graph** for a sentence $x = (w_0, w_1, \ldots, w_n)$ is a labeled directed graph $G = (V, A)$, where

1. $V = \{0, 1, \ldots, n\}$ is a set of nodes,
2. $A \subseteq V \times L \times V$ is a set of labeled directed arcs.

The set V of **nodes** (or **vertices**) is the set of non-negative integers up to and including n, each corresponding to the linear position of a word in the sentence (including ROOT). The set A of **arcs** (or **directed edges**) is a set of ordered triples (i, l, j), where i and j are nodes and l is a dependency label. Because arcs are used to represent dependency relations, we will say that i is the **head** and l is the **dependency type** of j. Conversely, we say that j is a **dependent** of i.

1 In the latter case, the dependency graph has been derived automatically from the constituency-based annotation in the treebank, using the Penn2Malt program, available at `http://w3.msi.vxu.se/users/nivre/research/Penn2Malt.html`.

Definition 2

A dependency graph $G = (V, A)$ is **well-formed** if and only if:

1. The node 0 is a root, that is, there is no node i and label l such that $(i, l, 0) \in A$.
2. Every node has at most one head and one label, that is, if $(i, l, j) \in A$ then there is no node i' and label l' such that $(i', l', j) \in A$ and $i \neq i'$ or $l \neq l'$.
3. The graph G is acyclic, that is, there is no (non-empty) subset of arcs $\{(i_0, l_1, i_1), (i_1, l_2, i_2), \ldots, (i_{k-1}, l_k, i_k)\} \subseteq A$ such that $i_0 = i_k$.

We will refer to conditions 1–3 as ROOT, SINGLE-HEAD, and ACYCLICITY, respectively. Any dependency graph satisfying these conditions is a **dependency forest**; if it is also connected, it is a **dependency tree**, that is, a directed tree rooted at the node 0. It is worth noting that any dependency forest can be turned into a dependency tree by adding arcs from the node 0 to all other roots.

Definition 3

A dependency graph $G = (V, A)$ is **projective** if and only if, for every arc $(i, l, j) \in A$ and node $k \in V$, if $i < k < j$ or $j < k < i$ then there is a subset of arcs $\{(i, l_1, i_1), (i_1, l_2, i_2), \ldots (i_{k-1}, l_k, i_k)\} \in A$ such that $i_k = k$.

In a projective dependency graph, every node has a continuous projection, where the projection of a node i is the set of nodes reachable from i in the reflexive and transitive closure of the arc relation. This corresponds to the ban on discontinuous constituents in orthodox phrase structure representations. We call this condition PROJECTIVITY. When discussing PROJECTIVITY, we will often use the notation $i \rightarrow^* j$ to mean that j is reachable from i in the reflexive and transitive closure of the arc relation.

Example 1

For the graphs depicted in Figures 1 and 2, we have:

Figure 1:
$G_1 = (V_1, A_1)$
$V_1 = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$
$A_1 = \{(0, \text{Pred}, 3), (0, \text{AuxK}, 8), (1, \text{Atr}, 2), (3, \text{Sb}, 5), (3, \text{AuxP}, 6), (5, \text{AuxP}, 1), (5, \text{AuxZ}, 4), (6, \text{Adv}, 7)\}$

Figure 2:
$G_2 = (V_2, A_2)$
$V_2 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
$A_2 = \{(0, \text{ROOT}, 3), (2, \text{NMOD}, 1), (3, \text{SBJ}, 2), (3, \text{OBJ}, 5), (3, \text{P}, 9), (5, \text{NMOD}, 4), (5, \text{NMOD}, 6), (6, \text{PMOD}, 8), (8, \text{NMOD}, 7)\}$

Both G_1 and G_2 are well-formed dependency forests (dependency trees, to be specific), but only G_2 is projective. In G_1, the arc $(5, \text{AuxP}, 1)$ spans node 3, which is not reachable from node 5 by following dependency arcs.

3. Deterministic Incremental Dependency Parsing

In this section, we introduce a formal framework for the specification of deterministic dependency parsing algorithms in terms of two components: a **transition system**, which

is nondeterministic in the general case, and an **oracle**, which always picks a single transition out of every parser configuration. The use of transition systems to study computation is a standard technique in theoretical computer science, which is here combined with the notion of oracles in order to characterize parsing algorithms with deterministic search. In data-driven dependency parsing, oracles normally take the form of classifiers, trained on treebank data, but they can also be defined in terms of grammars and heuristic disambiguation rules (Nivre 2003).

The main reason for introducing this framework is to allow us to characterize algorithms that have previously been described in different traditions and to compare their formal properties within a single unified framework. In particular, whereas this type of framework has previously been used to characterize algorithms in the stack-based family (Nivre 2003, 2006b; Attardi 2006), it is here being used also for the list-based algorithms first discussed by Covington (2001).

Definition 4
A **transition system** for dependency parsing is a quadruple $S = (C, T, c_s, C_t)$, where

1. C is a set of **configurations**, each of which contains a buffer β of (remaining) nodes and a set A of dependency arcs,
2. T is a set of transitions, each of which is a (partial) function $t : C \rightarrow C$,
3. c_s is an initialization function, mapping a sentence $x = (w_0, w_1, \ldots, w_n)$ to a configuration with $\beta = [1, \ldots, n]$,
4. $C_t \subseteq C$ is a set of terminal configurations.

A configuration is required to contain at least a buffer β, initially containing the nodes $[1, \ldots, n]$ corresponding to the real words of a sentence $x = (w_0, w_1, \ldots, w_n)$, and a set A of dependency arcs, defined on the nodes in $V = \{0, 1, \ldots, n\}$, given some set of dependency labels L. The specific transition systems defined in Sections 4 and 5 will extend this basic notion of configuration with different data structures, such as stacks and lists. We use the notation β_c and A_c to refer to the value of β and A, respectively, in a configuration c; we also use $|\beta|$ to refer to the length of β (i.e., the number of nodes in the buffer) and we use [ ] to denote an empty buffer.

Definition 5
Let $S = (C, T, c_s, C_t)$ be a transition system. A **transition sequence** for a sentence $x = (w_0, w_1, \ldots, w_n)$ in S is a sequence $C_{0,m} = (c_0, c_1, \ldots, c_m)$ of configurations, such that

1. $c_0 = c_s(x)$,
2. $c_m \in C_t$,
3. for every i $(1 \leq i \leq m)$, $c_i = t(c_{i-1})$ for some $t \in T$.

The **parse** assigned to x by $C_{0,m}$ is the dependency graph $G_{c_m} = (\{0, 1, \ldots, n\}, A_{c_m})$, where A_{c_m} is the set of dependency arcs in c_m.

Starting from the initial configuration for the sentence to be parsed, transitions will manipulate β and A (and other available data structures) until a terminal configuration is reached. Because the node set V is given by the input sentence itself, the set A_{c_m} of dependency arcs in the terminal configuration will determine the output dependency graph $G_{c_m} = (V, A_{c_m})$.

Definition 6

A transition system $S = (C, T, c_s, C_t)$ is **incremental** if and only if, for every configuration $c \in C$ and transition $t \in T$, it holds that:

1. if $\beta_c = [\,]$ then $c \in C_t$,
2. $|\beta_c| \geq |\beta_{t(c)}|$,
3. if $a \in A_c$ then $a \in A_{t(c)}$.

The first two conditions state that the buffer β never grows in size and that parsing terminates as soon as it becomes empty; the third condition states that arcs added to A can never be removed. Note that this is only one of several possible notions of incrementality in parsing. A weaker notion would be to only require that the set of arcs is built monotonically (the third condition); a stronger notion would be to require also that nodes in β are processed strictly left to right.

Definition 7

Let $S = (C, T, c_s, C_t)$ be a transition system for dependency parsing.

1. S is **sound** for a class $\mathbb{G}$ of dependency graphs if and only if, for every sentence x and every transition sequence $C_{0,m}$ for x in S, the parse $G_{c_m} \in \mathbb{G}$.
2. S is **complete** for a class $\mathbb{G}$ of dependency graphs if and only if, for every sentence x and every dependency graph G_x for x in $\mathbb{G}$, there is a transition sequence $C_{0,m}$ for x in S such that $G_{c_m} = G_x$.
3. S is **correct** for a class $\mathbb{G}$ of dependency graphs if and only if it is sound and complete for $\mathbb{G}$.

The notions of soundness and completeness, as defined here, can be seen as corresponding to the notions of soundness and completeness for grammar parsing algorithms, according to which an algorithm is sound if it only derives parses licensed by the grammar and complete if it derives all such parses (Shieber, Schabes, and Pereira 1995).

Depending on the nature of a transition system S, there may not be a transition sequence for every sentence, or there may be more than one such sequence. The systems defined in Sections 4 and 5 will all be such that, for any input sentence $x = (w_0, w_1, \ldots, w_n)$, there is always at least one transition sequence for x (and usually more than one).

Definition 8

An **oracle** for a transition system $S = (C, T, c_s, C_t)$ is a function $o : C \rightarrow T$.

Given a transition system $S = (C, T, c_s, C_t)$ and an oracle o, deterministic parsing can be achieved by the following simple algorithm:

PARSE($x = (w_0, w_1, \ldots, w_n)$)
1 $c \leftarrow c_s(x)$
2 **while** $c \notin C_t$
3 $\quad c \leftarrow [o(c)](c)$
4 **return** G_c

It is easy to see that, provided that there is at least one transition sequence in S for every sentence, such a parser constructs exactly one transition sequence $C_{0,m}$ for a sentence x and returns the parse defined by the terminal configuration c_m, that is, $G_{c_m} = (\{0, 1, \ldots, n\}, A_{c_m})$. The reason for separating the oracle o, which maps a configuration c to a transition t, from the transition t itself, which maps a configuration c to a new configuration c', is to have a clear separation between the abstract machine defined by the transition system, which determines formal properties such as correctness and complexity, and the search mechanism used when executing the machine.

In the experimental evaluation in Section 6, we will use the standard technique of approximating oracles with classifiers trained on treebank data. However, in the formal characterization of different parsing algorithms in Sections 4 and 5, we will concentrate on properties of the underlying transition systems. In particular, assuming that both $o(c)$ and $t(c)$ can be performed in constant time (for every o, t and c), which is reasonable in most cases, the worst-case time complexity of a deterministic parser based on a transition system S is given by an upper bound on the length of transition sequences in S. And the space complexity is given by an upper bound on the size of a configuration $c \in C$, because only one configuration needs to be stored at any given time in a deterministic parser.

4. Stack-Based Algorithms

The stack-based algorithms make use of a stack to store partially processed tokens, that is, tokens that have been removed from the input buffer but which are still considered as potential candidates for dependency links, either as heads or as dependents. A parser configuration is therefore defined as a triple, consisting of a stack, an input buffer, and a set of dependency arcs.

Definition 9

A **stack-based** configuration for a sentence $x = (w_0, w_1, \ldots, w_n)$ is a triple $c = (\sigma, \beta, A)$, where

1. σ is a stack of tokens $i \leq k$ (for some $k \leq n$),
2. β is a buffer of tokens $j > k$,
3. A is a set of dependency arcs such that $G = (\{0, 1, \ldots, n\}, A)$ is a dependency graph for x.

Both the stack and the buffer will be represented as lists, although the stack will have its head (or top) to the right for reasons of perspicuity. Thus, $\sigma|i$ represents a stack with top i and tail σ, and $j|\beta$ represents a buffer with head j and tail β.[2] We use square brackets for enumerated lists, for example, $[1, 2, \ldots, n]$, with $[\,]$ for the empty list as a special case.

Definition 10

A **stack-based** transition system is a quadruple $S = (C, T, c_s, C_t)$, where

1. C is the set of all stack-based configurations,
2. $c_s(x = (w_0, w_1, \ldots w_n)) = ([0], [1, \ldots, n], \emptyset)$,

2 The operator | is taken to be left-associative for the stack and right-associative for the buffer.

Transitions	
LEFT-ARC$_l$	$(\sigma\|i, j\|\beta, A) \Rightarrow (\sigma, j\|\beta, A \cup \{(j, l, i)\})$
RIGHT-ARC$_l^s$	$(\sigma\|i, j\|\beta, A) \Rightarrow (\sigma, i\|\beta, A \cup \{(i, l, j)\})$
SHIFT	$(\sigma, i\|\beta, A) \Rightarrow (\sigma\|i, \beta, A)$
Preconditions	
LEFT-ARC$_l$	$\neg[i = 0]$ $\neg\exists k \exists l'[(k, l', i) \in A]$
RIGHT-ARC$_l^s$	$\neg\exists k \exists l'[(k, l', j) \in A]$

Figure 3
Transitions for the arc-standard, stack-based parsing algorithm.

3. T is a set of transitions, each of which is a function $t : C \rightarrow C$,
4. $C_t = \{c \in C | c = (\sigma, [\,], A)\}$.

A stack-based parse of a sentence $x = (w_0, w_1, \ldots, w_n)$ starts with the artificial root node 0 on the stack σ, all the nodes corresponding to real words in the buffer β, and an empty set A of dependency arcs; it ends as soon as the buffer β is empty. The transitions used by stack-based parsers are essentially composed of two types of actions: adding (labeled) arcs to A and manipulating the stack σ and input buffer β. By combining such actions in different ways, we can construct transition systems that implement different parsing strategies. We will now define two such systems, which we call **arc-standard** and **arc-eager**, respectively, adopting the terminology of Abney and Johnson (1991).

4.1 Arc-Standard Parsing

The transition set T for the arc-standard, stack-based parser is defined in Figure 3 and contains three types of transitions:

1. Transitions LEFT-ARC$_l$ (for any dependency label l) add a dependency arc (j, l, i) to A, where i is the node on top of the stack σ and j is the first node in the buffer β. In addition, they pop the stack σ. They have as a precondition that the token i is not the artificial root node 0 and does not already have a head.
2. Transitions RIGHT-ARC$_l^s$ (for any dependency label l) add a dependency arc (i, l, j) to A, where i is the node on top of the stack σ and j is the first node in the buffer β. In addition, they pop the stack σ and replace j by i at the head of β. They have as a precondition that the token j does not already have a head.[3]
3. The transition SHIFT removes the first node i in the buffer β and pushes it on top of the stack σ.

3 The superscript s is used to distinguish these transitions from the non-equivalent RIGHT-ARC$_l^e$ transitions in the arc-eager system.

Transition	Configuration			
	([0],	[1, . . . , 9],	∅	)
SHIFT ⟹	([0, 1],	[2, . . . , 9],	∅	)
LEFT-ARC$_{\text{NMOD}}$ ⟹	([0],	[2, . . . , 9],	$A_1 = \{(2, \text{NMOD}, 1)\}$	)
SHIFT ⟹	([0, 2],	[3, . . . , 9],	A_1	)
LEFT-ARC$_{\text{SBJ}}$ ⟹	([0],	[3, . . . , 9],	$A_2 = A_1 \cup \{(3, \text{SBJ}, 2)\}$	)
SHIFT ⟹	([0, 3],	[4, . . . , 9],	A_2	)
SHIFT ⟹	([0, 3, 4],	[5, . . . , 9],	A_2	)
LEFT-ARC$_{\text{NMOD}}$ ⟹	([0, 3],	[5, . . . , 9],	$A_3 = A_2 \cup \{(5, \text{NMOD}, 4)\}$	)
SHIFT ⟹	([0, 3, 5],	[6, . . . , 9],	A_3	)
SHIFT ⟹	([0, . . . 6],	[7, 8, 9],	A_3	)
SHIFT ⟹	([0, . . . , 7],	[8, 9],	A_3	)
LEFT-ARC$_{\text{NMOD}}$ ⟹	([0, . . . 6],	[8, 9],	$A_4 = A_3 \cup \{(8, \text{NMOD}, 7)\}$	)
RIGHT-ARC$^s_{\text{PMOD}}$ ⟹	([0, 3, 5],	[6, 9],	$A_5 = A_4 \cup \{(6, \text{PMOD}, 8)\}$	)
RIGHT-ARC$^s_{\text{NMOD}}$ ⟹	([0, 3],	[5, 9],	$A_6 = A_5 \cup \{(5, \text{NMOD}, 6)\}$	)
RIGHT-ARC$^s_{\text{OBJ}}$ ⟹	([0],	[3, 9],	$A_7 = A_6 \cup \{(3, \text{OBJ}, 5)\}$	)
SHIFT ⟹	([0, 3],	[9],	A_7	)
RIGHT-ARC$^s_{\text{P}}$ ⟹	([0],	[3],	$A_8 = A_7 \cup \{(3, \text{P}, 9)\}$	)
RIGHT-ARC$^s_{\text{ROOT}}$ ⟹	([ ],	[0],	$A_9 = A_8 \cup \{(0, \text{ROOT}, 3)\}$	)
SHIFT ⟹	([0],	[ ],	A_9	)

Figure 4
Arc-standard transition sequence for the English sentence in Figure 2.

The arc-standard parser is the closest correspondent to the familiar shift-reduce parser for context-free grammars (Aho, Sethi, and Ullman 1986). The LEFT-ARC$_l$ and RIGHT-ARC^s_l transitions correspond to reduce actions, replacing a head-dependent structure with its head, whereas the SHIFT transition is exactly the same as the shift action. One peculiarity of the transitions, as defined here, is that the "reduce" transitions apply to one node on the stack and one node in the buffer, rather than two nodes on the stack. The reason for this formulation is to facilitate comparison with the arc-eager parser described in the next section and to simplify the definition of terminal configurations. By way of example, Figure 4 shows the transition sequence needed to parse the English sentence in Figure 2.

Theorem 1
The arc-standard, stack-based algorithm is correct for the class of projective dependency forests.

Proof 1
To show the soundness of the algorithm, we show that the dependency graph defined by the initial configuration, $G_{c_s(x)} = (V_x, \emptyset)$, is a projective dependency forest, and that every transition preserves this property. We consider each of the relevant conditions in turn, keeping in mind that the only transitions that modify the graph are LEFT-ARC$_l$ and RIGHT-ARC^s_l.

1. ROOT: The node 0 is a root in $G_{c_s(x)}$, and adding an arc of the form $(i, l, 0)$ is prevented by an explicit precondition of LEFT-ARC$_l$.

2. SINGLE-HEAD: Every node $i \in V_x$ has in-degree 0 in $G_{c_s(x)}$, and both LEFT-ARC$_l$ and RIGHT-ARC$_l^s$ have as a precondition that the dependent of the new arc has in-degree 0.

3. ACYCLICITY: $G_{c_s(x)}$ is acyclic, and adding an arc (i, l, j) will create a cycle only if there is a directed path from j to i. But this would require that a previous transition had added an arc of the form (k, l', i) (for some k, l'), in which case i would no longer be in σ or β.

4. PROJECTIVITY: $G_{c_s(x)}$ is projective, and adding an arc (i, l, j) will make the graph non-projective only if there is a node k such that $i < k < j$ or $j < k < i$ and neither $i \rightarrow^* k$ nor $j \rightarrow^* k$. Let $C_{0,m}$ be a configuration sequence for $x = (w_0, w_1, \ldots, w_n)$ and let $\Pi(p, i, j)$ (for $0 < p < m$, $0 \leq i < j \leq n$) be the claim that, for every k such that $i < k < j$, $i \rightarrow^* k$ or $j \rightarrow^* k$ in G_{c_p}. To prove that no arc can be non-projective, we need to prove that, if $c_p \in C_{0,m}$ and $c_p = (\sigma|i, j|\beta, A_{c_p})$, then $\Pi(p, i, j)$. (If $c_p = (\sigma|i, j|\beta, A_{c_p})$ and $\Pi(p, i, j)$, then $\Pi(p', i, j)$ for all p' such that $p < p'$, since in c_p every node k such that $i < k < j$ must already have a head.) We prove this by induction over the number $\Delta(p)$ of transitions leading to c_p from the first configuration $c_{p-\Delta(p)} \in C_{0,m}$ such that $c_{p-\Delta(p)} = (\sigma|i, \beta, A_{c_{p-\Delta(p)}})$ (i.e., the first configuration where i is on the top of the stack).

   Basis: If $\Delta(p) = 0$, then i and j are adjacent and $\Pi(p, i, j)$ holds vacuously.

   Inductive step: Assume that $\Pi(p, i, j)$ holds if $\Delta(p) \leq q$ (for some $q > 0$) and that $\Delta(p) = q + 1$. Now consider the transition t_p that results in configuration c_p. There are three cases:

   Case 1: If t_p = RIGHT-ARC$_l^s$ (for some l), then there is a node k such that $j < k$, $(j, l, k) \in A_{c_p}$, and $c_{p-1} = (\sigma|i|j, k|\beta, A_{c_p} - \{(j, l, k)\})$. This entails that there is an earlier configuration c_{p-r} ($2 \leq r \leq \Delta(p)$) such that $c_{p-r} = (\sigma|i, j|\beta, A_{c_{p-r}})$. Because $\Delta(p - r) = \Delta(p) - r \leq q$, we can use the inductive hypothesis to infer $\Pi(p - r, i, j)$ and hence $\Pi(p, i, j)$.

   Case 2: If t_p = LEFT-ARC$_l$ (for some l), then there is a node k such that $i < k < j$, $(j, l, k) \in A_{c_p}$, and $c_{p-1} = (\sigma|i|k, j|\beta, A_{c_p} - \{(j, l, k)\})$. Because $\Delta(p - 1) \leq q$, we can use the inductive hypothesis to infer $\Pi(p - 1, k, j)$ and, from this, $\Pi(p, k, j)$. Moreover, because there has to be an earlier configuration c_{p-r} ($r < \Delta(p)$) such that $c_{p-r} = (\sigma|i, k|\beta, A_{c_{p-r}})$ and $\Delta(p - r) \leq q$, we can use the inductive hypothesis again to infer $\Pi(p - r, i, k)$ and $\Pi(p, i, k)$. $\Pi(p, i, k)$, $\Pi(p, k, j)$ and $(j, l, k) \in A_{c_p}$ together entail $\Pi(p, i, j)$.

   Case 3: If the transition t_p = SHIFT, then it must have been preceded by a RIGHT-ARC$_l^s$ transition (for some l), because otherwise i and j would be adjacent. This means that there is a node k such that $i < k < j$, $(i, l, k) \in A_{c_p}$, and $c_{p-2} = (\sigma|i, k|j|\beta, A_{c_p} - \{(i, l, k)\})$. Because $\Delta(p - 2) \leq q$, we can again use the inductive hypothesis to infer $\Pi(p - 2, i, k)$ and $\Pi(p, i, k)$. Furthermore, it must be the case that either k and j are adjacent or there is an earlier configuration c_{p-r}

($r < \Delta(p)$) such that $c_{p-r} = (\sigma|k,j|\beta, A_{c_{p-r}})$; in both cases it follows that $\Pi(p,k,j)$ (in the latter through the inductive hypothesis via $\Pi(p-r,k,j)$). As before, $\Pi(p,i,k)$, $\Pi(p,k,j)$ and $(i,l,k) \in A_{c_p}$ together entail $\Pi(p,i,j)$.

For completeness, we need to show that for any sentence x and projective dependency forest $G_x = (V_x, A_x)$ for x, there is a transition sequence $C_{0,m}$ such that $G_{c_m} = G_x$. We prove this by induction on the length $|x|$ of $x = (w_0, w_1, \ldots, w_n)$.

Basis: If $|x| = 1$, then the only projective dependency forest for x is $G = (\{0\}, \emptyset)$ and $G_{c_m} = G_x$ for $C_{0,m} = (c_s(x))$.

Inductive step: Assume that the claim holds if $|x| \leq p$ (for some $p > 1$) and assume that $|x| = p + 1$ and $G_x = (V_x, A_x)$ ($V_x = \{0, 1, \ldots, p\}$). Consider the subgraph $G_{x'} = (V_x - \{p\}, A^{-p})$, where $A^{-p} = A_x - \{(i,l,j)|i = p \vee j = p\}$, that is, the graph $G_{x'}$ is exactly like G_x except that the node p and all the arcs going into or out of this node are missing. It is obvious that, if G_x is a projective dependency forest for the sentence $x = (w_0, w_1, \ldots, w_p)$, then $G_{x'}$ is a projective dependency forest for the sentence $x' = (w_0, w_1, \ldots, w_{p-1})$, and that, because $|x'| = p$, there is a transition sequence $C_{0,q}$ such that $G_{c_q} = G_{x'}$ (in virtue of the inductive hypothesis). The terminal configuration of $G_{0,q}$ must have the form $c_q = (\sigma_{c_q}, [\,], A^{-p})$, where $i \in \sigma_{c_q}$ if and only if i is a root in $G_{x'}$ (else i would have been removed in a LEFT-ARC$_l$ or RIGHT-ARC$_l^s$ transition). It follows that, in G_x, i is either a root or a dependent of p. In the latter case, any j such that $j \in \sigma_{c_q}$ and $i < j$ must also be a dependent of p (else G_x would not be projective, given that i and j are both roots in $G_{x'}$). Moreover, if p has a head k in G_x, then k must be the topmost node in σ_{c_q} that is not a dependent of p (anything else would again be inconsistent with the assumption that G_x is projective). Therefore, we can construct a transition sequence $C_{0,m}$ such that $G_{c_m} = G_x$, by starting in $c_0 = c_s(x)$ and applying exactly the same q transitions as in $C_{0,q}$, followed by as many LEFT-ARC$_l$ transitions as there are left dependents of p in G_x, followed by a RIGHT-ARC$_l^s$ transition if and only if p has a head in G_x, followed by a SHIFT transition (moving the head of p back to the stack and emptying the buffer). ■

Theorem 2
The worst-case time complexity of the arc-standard, stack-based algorithm is $O(n)$, where n is the length of the input sentence.

Proof 2
Assuming that the oracle and transition functions can be computed in some constant time, the worst-case running time is bounded by the maximum number of transitions in a transition sequence $C_{0,m}$ for a sentence $x = (w_0, w_1, \ldots, w_n)$. Since a SHIFT transition decreases the length of the buffer β by 1, no other transition increases the length of β, and any configuration where $\beta = [\,]$ is terminal, the number of SHIFT transitions in $C_{0,m}$ is bounded by n. Moreover, since both LEFT-ARC$_l$ and RIGHT-ARC$_l^s$ decrease the height of the stack by 1, only SHIFT increases the height of the stack by 1, and the initial height of the stack is 1, the combined number of instances of LEFT-ARC$_l$ and RIGHT-ARC$_l^s$ in $C_{0,m}$ is also bounded by n. Hence, the worst case time complexity is $O(n)$. ■

Remark 1
The assumption that the oracle function can be computed in constant time will be discussed at the end of Section 6.1, where we approximate oracles with treebank-induced classifiers in order to experimentally evaluate the different algorithms. The assumption that every transition can be performed in constant time can be justified by noting that the only operations involved are those of adding an arc to the graph, removing the first element from the buffer, and pushing or popping the stack.

Theorem 3
The worst-case space complexity of the arc-standard, stack-based algorithm is $O(n)$, where n is the length of the input sentence.

Proof 3
Given the deterministic parsing algorithm, only one configuration $c = (\sigma, \beta, A)$ needs to be stored at any given time. Assuming that a single node can be stored in some constant space, the space needed to store σ and β, respectively, is bounded by the number of nodes. The same holds for A, given that a single arc can be stored in constant space, because the number of arcs in a dependency forest is bounded by the number of nodes. Hence, the worst-case space complexity is $O(n)$. ■

4.2 Arc-Eager Parsing

The transition set T for the arc-eager, stack-based parser is defined in Figure 5 and contains four types of transitions:

1. Transitions LEFT-ARC$_l$ (for any dependency label l) add a dependency arc (j, l, i) to A, where i is the node on top of the stack σ and j is the first node in the buffer β. In addition, they pop the stack σ. They have as a precondition that the token i is not the artificial root node 0 and does not already have a head.

Transitions

LEFT-ARC$_l$	$(\sigma\|i, j\|\beta, A) \Rightarrow (\sigma, j\|\beta, A \cup \{(j, l, i)\})$
RIGHT-ARC$_l^e$	$(\sigma\|i, j\|\beta, A) \Rightarrow (\sigma\|i\|j, \beta, A \cup \{(i, l, j)\})$
REDUCE	$(\sigma\|i, \beta, A) \Rightarrow (\sigma, \beta, A)$
SHIFT	$(\sigma, i\|\beta, A) \Rightarrow (\sigma\|i, \beta, A)$

Preconditions

LEFT-ARC$_l$	$\neg[i = 0]$ $\neg\exists k \exists l'[(k, l', i) \in A]$
RIGHT-ARC$_l^e$	$\neg\exists k \exists l'[(k, l', j) \in A]$
REDUCE	$\exists k \exists l[(k, l, i) \in A]$

Figure 5
Transitions for the arc-eager, stack-based parsing algorithm.

2. Transitions RIGHT-ARC$_l^e$ (for any dependency label l) add a dependency arc (i, l, j) to A, where i is the node on top of the stack σ and j is the first node in the buffer β. In addition, they remove the first node j in the buffer β and push it on top of the stack σ. They have as a precondition that the token j does not already have a head.
3. The transition REDUCE pops the stack β and is subject to the precondition that the top token has a head.
4. The transition SHIFT removes the first node i in the buffer β and pushes it on top of the stack σ.

The arc-eager parser differs from the arc-standard one by attaching right dependents (using RIGHT-ARC$_l^e$ transitions) as soon as possible, that is, before the right dependent has found all its right dependents. As a consequence, the RIGHT-ARC$_l^e$ transitions cannot replace the head-dependent structure with the head, as in the arc-standard system, but must store both the head and the dependent on the stack for further processing. The dependent can be popped from the stack at a later time through the REDUCE transition, which completes the reduction of this structure. The arc-eager system is illustrated in Figure 6, which shows the transition sequence needed to parse the English sentence in Figure 2 with the same output as the arc-standard sequence in Figure 4.

Theorem 4
The arc-eager, stack-based algorithm is correct for the class of projective dependency forests.

Transition	**Configuration**			
	([0],	[1, . . . , 9],	$\emptyset$	)
SHIFT $\Longrightarrow$	([0, 1],	[2, . . . , 9],	$\emptyset$	)
LEFT-ARC$_{\text{NMOD}}$ $\Longrightarrow$	([0],	[2, . . . , 9],	$A_1 = \{(2, \text{NMOD}, 1)\}$	)
SHIFT $\Longrightarrow$	([0, 2],	[3, . . . , 9],	A_1	)
LEFT-ARC$_{\text{SBJ}}$ $\Longrightarrow$	([0],	[3, . . . , 9],	$A_2 = A_1 \cup \{(3, \text{SBJ}, 2)\}$	)
RIGHT-ARC$^e_{\text{ROOT}}$ $\Longrightarrow$	([0, 3],	[4, . . . , 9],	$A_3 = A_2 \cup \{(0, \text{ROOT}, 3)\}$	)
SHIFT $\Longrightarrow$	([0, 3, 4],	[5, . . . , 9],	A_3	)
LEFT-ARC$_{\text{NMOD}}$ $\Longrightarrow$	([0, 3],	[5, . . . , 9],	$A_4 = A_3 \cup \{(5, \text{NMOD}, 4)\}$	)
RIGHT-ARC$^e_{\text{OBJ}}$ $\Longrightarrow$	([0, 3, 5],	[6, . . . , 9],	$A_5 = A_4 \cup \{(3, \text{OBJ}, 5)\}$	)
RIGHT-ARC$^e_{\text{NMOD}}$ $\Longrightarrow$	([0, . . . , 6],	[7, 8, 9],	$A_6 = A_5 \cup \{(5, \text{NMOD}, 6)\}$	)
SHIFT $\Longrightarrow$	([0, . . . , 7],	[8, 9],	A_6	)
LEFT-ARC$_{\text{NMOD}}$ $\Longrightarrow$	([0, . . . 6],	[8, 9],	$A_7 = A_6 \cup \{(8, \text{NMOD}, 7)\}$	)
RIGHT-ARC$^e_{\text{PMOD}}$ $\Longrightarrow$	([0, . . . , 8],	[9],	$A_8 = A_7 \cup \{(6, \text{PMOD}, 8)\}$	)
REDUCE $\Longrightarrow$	([0, . . . , 6],	[9],	A_8	)
REDUCE $\Longrightarrow$	([0, 3, 5],	[9],	A_8	)
REDUCE $\Longrightarrow$	([0, 3],	[9],	A_8	)
RIGHT-ARC$^e_{\text{P}}$ $\Longrightarrow$	([0, 3, 9],	[ ],	$A_9 = A_8 \cup \{(3, \text{P}, 9)\}$	)

Figure 6
Arc-eager transition sequence for the English sentence in Figure 2.

Proof 4
To show the soundness of the algorithm, we show that the dependency graph defined by the initial configuration, $G_{c_0(x)} = (V_x, \emptyset)$, is a projective dependency forest, and that every transition preserves this property. We consider each of the relevant conditions in turn, keeping in mind that the only transitions that modify the graph are LEFT-ARC$_l$ and RIGHT-ARC$_l^e$.

1. ROOT: Same as Proof 1.
2. SINGLE-HEAD: Same as Proof 1 (substitute RIGHT-ARC$_l^e$ for RIGHT-ARC$_l^s$).
3. ACYCLICITY: $G_{c_s(x)}$ is acyclic, and adding an arc (i, l, j) will create a cycle only if there is a directed path from j to i. In the case of LEFT-ARC$_l$, this would require the existence of a configuration $c_p = (\sigma|j, i|\beta, A_{c_p})$ such that $(k, l', i) \in A_{c_p}$ (for some k and l'), which is impossible because any transition adding an arc (k, l', i) has as a consequence that i is no longer in the buffer. In the case of RIGHT-ARC$_l^e$, this would require a configuration $c_p = (\sigma|i, j|\beta, A_{c_p})$ such that, given the arcs in A_{c_p}, there is a directed path from j to i. Such a path would have to involve at least one arc (k, l', k') such that $k' \leq i < k$, which would entail that i is no longer in σ. (If $k' = i$, then i would be popped in the LEFT-ARC$_{l'}$ transition adding the arc; if $k' < i$, then i would have to be popped before the arc could be added.)
4. PROJECTIVITY: To prove that, if $c_p \in C_{0,m}$ and $c_p = (\sigma|i, j|\beta, A_{c_p})$, then $\Pi(p, i, j)$, we use essentially the same technique as in Proof 1, only with different cases in the inductive step because of the different transitions. As before, we let $\Delta(p)$ be the number of transitions that it takes to reach c_p from the first configuration that has i on top of the stack.

   Basis: If $\Delta(p) = 0$, then i and j are adjacent, which entails $\Pi(p, i, j)$.

   Inductive step: We assume that $\Pi(p, i, j)$ holds if $\Delta(p) \leq q$ (for some $q > 0$) and that $\Delta(p) = q + 1$, and we concentrate on the transition t_p that results in configuration c_p. For the arc-eager algorithm, there are only two cases to consider, because if t_p = RIGHT-ARC$_l^e$ (for some l) or t_p = SHIFT then $\Delta(p) = 0$, which contradicts our assumption that $\Delta(p) > q > 0$. (This follows because the arc-eager algorithm, unlike its arc-standard counterpart, does not allow nodes to be moved back from the stack to the buffer.)

   Case 1: If t_p = LEFT-ARC$_l$ (for some l), then there is a node k such that $i < k < j$, $(j, l, k) \in A_{c_p}$, and $c_{p-1} = (\sigma|i|k, j|\beta, A_{c_p} - \{(j, l, k)\})$. Because $\Delta(p - 1) \leq q$, we can use the inductive hypothesis to infer $\Pi(p - 1, k, j)$ and, from this, $\Pi(p, k, j)$. Moreover, because there has to be an earlier configuration c_{p-r} ($r < \Delta(p)$) such that $c_{p-r} = (\sigma|i, k|\beta, A_{c_{p-r}})$ and $\Delta(p - r) \leq q$, we can use the inductive hypothesis again to infer $\Pi(p - r, i, k)$ and $\Pi(p, i, k)$. $\Pi(p, i, k)$, $\Pi(p, k, j)$ and $(j, l, k) \in A_{c_p}$ together entail $\Pi(p, i, j)$.

   Case 2: If the transition t_p = REDUCE, then there is a node k such that $i < k < j$, $(i, l, k) \in A_{c_p}$, and $c_{p-1} = (\sigma|i|k, j|\beta, A_{c_p})$. Because $\Delta(p - 1) \leq q$, we can again use the inductive hypothesis to infer $\Pi(p - 1, k, j)$ and $\Pi(p, k, j)$. Moreover,

there must be an earlier configuration c_{p-r} ($r < \Delta(p)$) such that $c_{p-r} = (\sigma|i,k|\beta, A_{c_{p-r}})$ and $\Delta(p-r) \leq q$, which entails $\Pi(p-r,i,k)$ and $\Pi(p,i,k)$. As before, $\Pi(p,i,k)$, $\Pi(p,k,j)$ and $(i,l,k) \in A_{c_p}$ together entail $\Pi(p,i,j)$.

For completeness, we need to show that for any sentence x and projective dependency forest $G_x = (V_x, A_x)$ for x, there is a transition sequence $C_{0,m}$ such that $G_{c_m} = G_x$. Using the same idea as in Proof 1, we prove this by induction on the length $|x|$ of $x = (w_0, w_1, \ldots, w_n)$.

Basis: If $|x| = 1$, then the only projective dependency forest for x is $G = (\{0\}, \emptyset)$ and $G_{c_m} = G_x$ for $C_{0,m} = (c_s(x))$.

Inductive step: Assume that the claim holds if $|x| \leq p$ (for some $p > 1$) and assume that $|x| = p + 1$ and $G_x = (V_x, A_x)$ ($V_x = \{0, 1, \ldots, p\}$). As in Proof 1, we may now assume that there exists a transition sequence $C_{0,q}$ for the sentence $x' = (w_0, w_1, w_{p-1})$ and subgraph $G_{x'} = (V_x - \{p\}, A^{-p})$, where the terminal configuration has the form $c_q = (\sigma_{c_q}, [\,], A^{-p})$. For the arc-eager algorithm, if i is a root in $G_{x'}$ then $i \in \sigma_{c_q}$; but if $i \in \sigma_{c_q}$ then i is *either* a root *or* has a head j such that $j < i$ in $G_{x'}$. (This is because i may have been pushed onto the stack in a RIGHT-ARC$_l^e$ transition and may or may not have been popped in a later REDUCE transition.) Apart from the possibility of unreduced right dependents, we can use the same reasoning as in Proof 1 to show that, for any $i \in \sigma_{c_q}$ that is a root in $G_{x'}$, if i is a dependent of p in G_x then any j such that $j \in \sigma_{c_q}$, $i < j$ and j is a root in $G_{x'}$ must also be a dependent of p in G_x (or else G_x would fail to be projective). Moreover, if p has a head k in G_x, then k must be in σ_{c_q} and any j such that $j \in \sigma_{c_q}$ and $k < j$ must either be a dependent of p in G_x or must have a head to the left in both $G_{x'}$ and G_x (anything else would again be inconsistent with the assumption that G_x is projective). Therefore, we can construct a transition sequence $C_{0,m}$ such that $G_{c_m} = G_x$, by starting in $c_0 = c_s(x)$ and applying exactly the same q transitions as in $C_{0,q}$, followed by as many LEFT-ARC$_l$ transitions as there are left dependents of p in G_x, interleaving REDUCE transitions whenever the node on top of the stack already has a head, followed by a RIGHT-ARC$_l^e$ transition if p has a head in G_x and a SHIFT transition otherwise (in both cases moving p to the stack and emptying the buffer). ■

Theorem 5
The worst-case time complexity of the arc-eager, stack-based algorithm is $O(n)$, where n is the length of the input sentence.

Proof 5
The proof is essentially the same as Proof 2, except that both SHIFT and RIGHT-ARC$_l^e$ decrease the length of β and increase the height of σ, while both REDUCE and LEFT-ARC$_l$ decrease the height of σ. Hence, the combined number of SHIFT and RIGHT-ARC$_l^e$ transitions, as well as the combined number of REDUCE and LEFT-ARC$_l$ transitions, are bounded by n. ■

Theorem 6
The worst-case space complexity of the arc-eager, stack-based algorithm is $O(n)$, where n is the length of the input sentence.

Proof 6
Same as Proof 3. ■

5. List-Based Algorithms

The list-based algorithms make use of two lists to store partially processed tokens, that is, tokens that have been removed from the input buffer but which are still considered as potential candidates for dependency links, either as heads or as dependents. A parser configuration is therefore defined as a quadruple, consisting of two lists, an input buffer, and a set of dependency arcs.

Definition 11
A **list-based** configuration for a sentence $x = (w_0, w_1, \ldots, w_n)$ is a quadruple $c = (\lambda_1, \lambda_2, \beta, A)$, where

1. λ_1 is a list of tokens $i_1 \leq k_1$ (for some $k_1 \leq n$),
2. λ_2 is a list of tokens $i_2 \leq k_2$ (for some k_2, $k_1 < k_2 \leq n$),
3. β is a buffer of tokens $j > k$,
4. A is a set of dependency arcs such that $G = (\{0, 1, \ldots, n\}, A)$ is a dependency graph for x.

The list λ_1 has its head to the right and stores nodes in descending order, and the list λ_2 has its head to the left and stores nodes in ascending order. Thus, $\lambda_1|i$ represents a list with head i and tail λ_1, whereas $j|\lambda_2$ represents a list with head j and tail λ_2.[4] We use square brackets for enumerated lists as before, and we write $\lambda_1.\lambda_2$ for the concatenation of λ_1 and λ_2, so that, for example, $[0, 1].[2, 3, 4] = [0, 1, 2, 3, 4]$. The notational conventions for the buffer β and the set A of dependency arcs are the same as before.

Definition 12
A **list-based** transition system is a quadruple $S = (C, T, c_s, C_t)$, where

1. C is the set of all list-based configurations,
2. $c_s(x = (w_0, w_1, \ldots w_n)) = ([0], [\,], [1, \ldots, n], \emptyset)$,
3. T is a set of transitions, each of which is a function $t : C \rightarrow C$,
4. $C_t = \{c \in C | c = (\lambda_1, \lambda_2, [\,], A)\}$.

A list-based parse of a sentence $x = (w_0, w_1, \ldots, w_n)$ starts with the artificial root node 0 as the sole element of λ_1, an empty list λ_2, all the nodes corresponding to real words in the buffer β, and an empty set A of dependency arcs; it ends as soon as the buffer β is empty. Thus, the only difference compared to the stack-based systems is that we have two lists instead of a single stack. Otherwise, both initialization and termination are

4 The operator | is taken to be left-associative for λ_1 and right-associative for λ_2.

Transitions

LEFT-ARC$_l^n$	$(\lambda_1 \mid i, \lambda_2, j \mid \beta, A) \Rightarrow (\lambda_1, i \mid \lambda_2, j \mid \beta, A \cup \{(j, l, i)\})$
RIGHT-ARC$_l^n$	$(\lambda_1 \mid i, \lambda_2, j \mid \beta, A) \Rightarrow (\lambda_1, i \mid \lambda_2, j \mid \beta, A \cup \{(i, l, j)\})$
NO-ARC^n	$(\lambda_1 \mid i, \lambda_2, \beta, A) \Rightarrow (\lambda_1, i \mid \lambda_2, \beta, A)$
SHIFT$^\lambda$	$(\lambda_1, \lambda_2, i \mid \beta, A) \Rightarrow (\lambda_1 . \lambda_2 \mid i, [\,], \beta, A)$

Preconditions

LEFT-ARC$_l^n$	$\neg[i = 0]$ $\neg\exists k \exists l'[(k, l', i) \in A]$ $\neg[i \rightarrow^* j]_A$
RIGHT-ARC$_l^n$	$\neg\exists k \exists l'[(k, l', j) \in A]$ $\neg[j \rightarrow^* i]_A$

Figure 7
Transitions for the non-projective, list-based parsing algorithm.

essentially the same. The transitions used by list-based parsers are again composed of two types of actions: adding (labeled) arcs to A and manipulating the lists λ_1 and λ_2, and the input buffer β. By combining such actions in different ways, we can construct transition systems with different properties. We will now define two such systems, which we call **non-projective** and **projective**, respectively, after the classes of dependency graphs that they can handle.

A clarification may be in order concerning the use of *lists* instead of *stacks* for this family of algorithms. In fact, most of the transitions to be defined subsequently make no essential use of this added flexibility and could equally well have been formalized using two stacks instead. However, we will sometimes need to append two lists into one, and this would not be a constant-time operation using standard stack operations. We therefore prefer to define these structures as lists, even though they will mostly be used as stacks.

5.1 Non-Projective Parsing

The transition set T for the non-projective, list-based parser is defined in Figure 7 and contains four types of transitions:

1. Transitions LEFT-ARC$_l^n$ (for any dependency label l) add a dependency arc (j, l, i) to A, where i is the head of the list λ_1 and j is the first node in the buffer β. In addition, they move i from the list λ_1 to the list λ_2. They have as a precondition that the token i is not the artificial root node and does not already have a head. In addition, there must not be a path from i to j in the graph $G = (\{0, 1, \ldots, n\}, A)$.[5]

5 We use the notation $[i \longrightarrow^* j]_A$ to signify that there is a path connecting i and j in $G = (\{0, 1, \ldots, n\}, A)$.

2. Transitions RIGHT-ARC$_l^n$ (for any dependency label l) add a dependency arc (i, l, j) to A, where i is the head of the list λ_1 and j is the first node in the buffer β. In addition, they move i from the list λ_1 to the list λ_2. They have as a precondition that the token j does not already have a head and that there is no path from j to i in $G = (\{0, 1, \ldots, n\}, A)$.

3. The transition NO-ARC removes the head i of the list λ_1 and inserts it at the head of the list λ_2.

4. The transition SHIFT removes the first node i in the buffer β and inserts it at the head of a list obtained by concatenating λ_1 and λ_2. This list becomes the new λ_1, whereas λ_2 is empty in the resulting configuration.

The non-projective, list-based parser essentially builds a dependency graph by considering every pair of nodes (i, j) $(i < j)$ and deciding whether to add a dependency arc between them (in either direction), although the SHIFT transition allows it to skip certain pairs. More precisely, if i is the head of λ_1 and j is the first node in the buffer β when a SHIFT transition is performed, then all pairs (k, j) such that $k < i$ are ignored. The fact that both the head and the dependent are kept in either λ_2 or β makes it possible to construct non-projective dependency graphs, because the NO-ARC^n transition allows a node to be passed from λ_1 to λ_2 even if it does not (yet) have a head. However, an arc can only be added between two nodes i and j if the dependent end of the arc is not the artificial root 0 and does not already have a head, which would violate ROOT and SINGLE-HEAD, respectively, and if there is no path connecting the dependent to the head, which would cause a violation of ACYCLICITY. As an illustration, Figure 8 shows the transition sequence needed to parse the Czech sentence in Figure 1, which has a non-projective dependency graph.

Theorem 7
The non-projective, list-based algorithm is correct for the class of dependency forests.

Proof 7
To show the soundness of the algorithm, we simply observe that the dependency graph defined by the initial configuration, $G_{c_0(x)} = (\{0, 1, \ldots, n\}, \emptyset)$, satisfies ROOT, SINGLE-HEAD, and ACYCLICITY, and that none of the four transitions may lead to a violation of these constraints. (The transitions SHIFT$^\lambda$ and NO-ARC^n do not modify the graph at all, and LEFT-ARC$_l^n$ and RIGHT-ARC$_l^n$ have explicit preconditions to prevent this.)

For *completeness*, we need to show that for any sentence x and dependency forest $G_x = (V_x, A_x)$ for x, there is a transition sequence $C_{0,m}$ such that $G_{c_m} = G_x$. Using the same idea as in Proof 1, we prove this by induction on the length $|x|$ of $x = (w_0, w_1, \ldots, w_n)$.

> **Basis:** If $|x| = 1$, then the only dependency forest for x is $G = (\{0\}, \emptyset)$ and $G_{c_m} = G_x$ for $C_{0,m} = (c_s(x))$.
>
> **Inductive step:** Assume that the claim holds if $|x| \leq p$ (for some $p > 1$) and assume that $|x| = p + 1$ and $G_x = (V_x, A_x)$ $(V_x = \{0, 1, \ldots, p\})$. As in Proof 1, we may now assume that there exists a transition sequence $C_{0,q}$ for the sentence $x' = (w_0, w_1, w_{p-1})$ and subgraph $G_{x'} = (V_x - \{p\}, A^{-p})$, but the terminal configuration now has the form $c_q = (\lambda_{c_q}, [\,], [\,], A^{-p})$, where $\lambda_{c_q} = [0, 1, \ldots, p-1]$. In order to construct a transition sequence $C_{0,m}$ such that $G_{c_m} = G_x$ we instead start from the configuration

Transition	Configuration				
	(	$[0]$,	$[\,]$,	$[1,\ldots,8]$,	$\emptyset$)
$\textsc{Shift}^{\lambda} \Longrightarrow$	(	$[0,1]$,	$[\,]$,	$[2,\ldots,8]$,	$\emptyset$)
$\textsc{Right-Arc}^{n}_{\text{Atr}} \Longrightarrow$	(	$[0]$,	$[1]$,	$[2,\ldots,8]$,	$A_1 = \{(1,\text{Atr},2)\}$)
$\textsc{Shift}^{\lambda} \Longrightarrow$	(	$[0,1,2]$,	$[\,]$,	$[3,\ldots,8]$,	A_1)
$\textsc{No-Arc}^{n} \Longrightarrow$	(	$[0,1]$,	$[2]$,	$[3,\ldots,8]$,	A_1)
$\textsc{No-Arc}^{n} \Longrightarrow$	(	$[0]$,	$[1,2]$,	$[3,\ldots,8]$,	A_1)
$\textsc{Right-Arc}^{n}_{\text{Pred}} \Longrightarrow$	(	$[\,]$,	$[0,1,2]$,	$[3,\ldots,8]$,	$A_2 = A_1 \cup \{(0,\text{Pred},3)\}$)
$\textsc{Shift}^{\lambda} \Longrightarrow$	(	$[0,\ldots,3]$,	$[\,]$,	$[4,\ldots,8]$,	A_2)
$\textsc{Shift}^{\lambda} \Longrightarrow$	(	$[0,\ldots,4]$,	$[\,]$,	$[5,\ldots,8]$,	A_2)
$\textsc{Left-Arc}^{n}_{\text{AuxZ}} \Longrightarrow$	(	$[0,\ldots,3]$,	$[4]$,	$[5,\ldots,8]$,	$A_3 = A_2 \cup \{(5,\text{AuxZ},4)\}$)
$\textsc{Right-Arc}^{n}_{\text{Sb}} \Longrightarrow$	(	$[0,1,2]$,	$[3,4]$,	$[5,\ldots,8]$,	$A_4 = A_3 \cup \{(3,\text{Sb},5)\}$)
$\textsc{No-Arc}^{n} \Longrightarrow$	(	$[0,1]$,	$[2,3,4]$,	$[5,\ldots,8]$,	A_4)
$\textsc{Left-Arc}^{n}_{\text{AuxP}} \Longrightarrow$	(	$[0]$,	$[1,\ldots,4]$,	$[5,\ldots,8]$,	$A_5 = A_4 \cup \{(5,\text{AuxP},1)\}$)
$\textsc{Shift}^{\lambda} \Longrightarrow$	(	$[0,\ldots,5]$,	$[\,]$,	$[6,7,8]$,	A_5)
$\textsc{No-Arc}^{n} \Longrightarrow$	(	$[0,\ldots,4]$,	$[5]$,	$[6,7,8]$,	A_5)
$\textsc{No-Arc}^{n} \Longrightarrow$	(	$[0,\ldots,3]$,	$[4,5]$,	$[6,7,8]$,	A_5)
$\textsc{Right-Arc}^{n}_{\text{AuxP}} \Longrightarrow$	(	$[0,1,2]$,	$[3,4,5]$,	$[6,7,8]$,	$A_6 = A_5 \cup \{(3,\text{AuxP},6)\}$)
$\textsc{Shift}^{\lambda} \Longrightarrow$	(	$[0,\ldots,6]$,	$[\,]$,	$[7,8]$,	A_6)
$\textsc{Right-Arc}^{n}_{\text{Adv}} \Longrightarrow$	(	$[0,\ldots,5]$,	$[6]$,	$[7,8]$,	$A_7 = A_6 \cup \{(6,\text{Adv},7)\}$)
$\textsc{Shift}^{\lambda} \Longrightarrow$	(	$[0,\ldots,7]$,	$[\,]$,	$[8]$,	A_7)
$\textsc{No-Arc}^{n} \Longrightarrow$	(	$[0,\ldots,6]$,	$[7]$,	$[8]$,	A_7)
$\textsc{No-Arc}^{n} \Longrightarrow$	(	$[0,\ldots,5]$,	$[6,7]$,	$[8]$,	A_7)
$\textsc{No-Arc}^{n} \Longrightarrow$	(	$[0,\ldots,4]$,	$[5,6,7]$,	$[8]$,	A_7)
$\textsc{No-Arc}^{n} \Longrightarrow$	(	$[0,\ldots,3]$,	$[4,\ldots,7]$,	$[8]$,	A_7)
$\textsc{No-Arc}^{n} \Longrightarrow$	(	$[0,1,2]$,	$[3,\ldots,7]$,	$[8]$,	A_7)
$\textsc{No-Arc}^{n} \Longrightarrow$	(	$[0,1]$,	$[2,\ldots,7]$,	$[8]$,	A_7)
$\textsc{No-Arc}^{n} \Longrightarrow$	(	$[0]$,	$[1,\ldots,7]$,	$[8]$,	A_7)
$\textsc{Right-Arc}^{n}_{\text{AuxK}} \Longrightarrow$	(	$[\,]$,	$[0,\ldots,7]$,	$[8]$,	$A_8 = A_7 \cup \{(0,\text{AuxK},8)\}$)
$\textsc{Shift}^{\lambda} \Longrightarrow$	(	$[0,\ldots,8]$,	$[\,]$,	$[\,]$,	A_8)

Figure 8
Non-projective transition sequence for the Czech sentence in Figure 1.

$c_0 = c_s(x)$ and apply exactly the same q transitions, reaching the configuration $c_q = (\lambda_{c_q}, [\,], [p], A^{-p})$. We then perform exactly p transitions, in each case choosing $\textsc{Left-Arc}^{n}_{l}$ if the token i at the head of λ_1 is a dependent of p in G_x (with label l), $\textsc{Right-Arc}^{n}_{l'}$ if i is the head of p (with label l') and $\textsc{No-Arc}^{n}$ otherwise. One final $\textsc{Shift}^{\lambda}$ transition takes us to the terminal configuration $c_m = (\lambda_{c_q}|p, [\,], [\,], A_x)$. ■

Theorem 8
The worst-case time complexity of the non-projective, list-based algorithm is $O(n^2)$, where n is the length of the input sentence.

Proof 8
Assuming that the oracle and transition functions can be performed in some constant time, the worst-case running time is bounded by the maximum number of transitions

in a transition sequence $C_{0,m}$ for a sentence $x = (w_0, w_1, \ldots, w_n)$. As for the stack-based algorithms, there can be at most n SHIFT$^{\lambda}$ transitions in $C_{0,m}$. Moreover, because each of the three other transitions presupposes that λ_1 is non-empty and decreases its length by 1, there can be at most i such transitions between the $i-1$th and the ith SHIFT transition. It follows that the total number of transitions in $C_{0,m}$ is bounded by $\sum_{i=1}^{n} i+1$, which is $O(n^2)$. ■

Remark 2
The assumption that transitions can be performed in constant time can be justified by the same kind of considerations as for the stack-based algorithms (cf. Remark 1). The only complication is the SHIFT$^{\lambda}$ transition, which involves appending the two lists λ_1 and λ_2, but this can be handled with an appropriate choice of data structures. A more serious complication is the need to check the preconditions of LEFT-ARC$_l^n$ and RIGHT-ARC$_l^n$, but if we assume that it is the responsibility of the oracle to ensure that the preconditions of any predicted transition are satisfied, we can postpone the discussion of this problem until the end of Section 6.1.

Theorem 9
The worst-case space complexity of the non-projective, list-based algorithm is $O(n)$, where n is the length of the input sentence.

Proof 9
Given the deterministic parsing algorithm, only one configuration $c = (\lambda_1, \lambda_2, \beta, A)$ needs to be stored at any given time. Assuming that a single node can be stored in some constant space, the space needed to store λ_1, λ_2, and β, respectively, is bounded by the number of nodes. The same holds for A, given that a single arc can be stored in constant space, because the number of arcs in a dependency forest is bounded by the number of nodes. Hence, the worst-case space complexity is $O(n)$. ■

5.2 Projective Parsing

The transition set T for the projective, list-based parser is defined in Figure 9 and contains four types of transitions:

1. Transitions LEFT-ARC$_l^p$ (for any dependency label l) add a dependency arc (j, l, i) to A, where i is the head of the list λ_1 and j is the first node in the buffer β. In addition, they remove i from the list λ_1 and empty λ_2. They have as a precondition that the token i is not the artificial root node and does not already have a head.
2. Transitions RIGHT-ARC$_l^p$ (for any dependency label l) add a dependency arc (i, l, j) to A, where i is the head of the list λ_1 and j is the first node in the buffer β. In addition, they move j from the buffer β and empty the list λ_2. They have as a precondition that the token j does not already have a head.
3. The transition NO-ARC^p removes the head i of the list λ_1 and inserts it at the head of the list λ_2. It has as a precondition that the node i already has a head.
4. The transition SHIFT$^{\lambda}$ removes the first node i in the buffer β and inserts it at the head of a list obtained by concatenating λ_1 and λ_2. This list becomes the new λ_1, while λ_2 is empty in the resulting configuration.

Transitions

LEFT-ARC$_l^p$	$(\lambda_1 \mid i, \lambda_2, j \mid \beta, A) \Rightarrow (\lambda_1, [\,], j \mid \beta, A \cup \{(j, l, i)\})$
RIGHT-ARC$_l^p$	$(\lambda_1 \mid i, \lambda_2, j \mid \beta, A) \Rightarrow (\lambda_1 \mid i \mid j, [\,], \beta, A \cup \{(i, l, j)\})$
NO-ARC^p	$(\lambda_1 \mid i, \lambda_2, \beta, A) \Rightarrow (\lambda_1, i \mid \lambda_2, \beta, A)$
SHIFT$^\lambda$	$(\lambda_1, \lambda_2, i \mid \beta, A) \Rightarrow (\lambda_1 . \lambda_2 \mid i, [\,], \beta, A)$

Preconditions

LEFT-ARC$_l^p$	$\neg[i = 0]$ $\neg\exists k \exists l'[(k, l', i) \in A]$
RIGHT-ARC$_l^p$	$\neg\exists k \exists l'[(k, l', j) \in A]$
NO-ARC^p	$\exists k \exists l[(k, l, i) \in A]$

Figure 9
Transitions for the projective, list-based parsing algorithm.

The projective, list-based parser uses the same basic strategy as its non-projective counterpart, but skips any pair (i, j) that could give rise to a non-projective dependency arc. The essential differences are the following:

1. While LEFT-ARC$_l^n$ stores the dependent i in the list λ_2, allowing it to have dependents to the right of j, LEFT-ARC$_l^p$ deletes it and in addition empties λ_2 because any dependency arc linking i, or any node between i and j, to a node succeeding j would violate PROJECTIVITY.
2. While RIGHT-ARC$_l^n$ allows the dependent j to seek dependents to the left of i, by simply moving i from λ_1 to λ_2, RIGHT-ARC$_l^p$ essentially incorporates a SHIFT$^\lambda$ transition by moving j to $\lambda_1 \mid i$, because any dependency arc linking j to a node preceding i would violate PROJECTIVITY. In addition, it does not move any nodes from λ_2 to λ_1, since these nodes can no longer be linked to any node succeeding j without violating PROJECTIVITY.
3. While NO-ARC^n is permissible as long as λ_1 is not empty, NO-ARC^p requires that the node i already has a head because any dependency arc spanning a root node would violate PROJECTIVITY (regardless of which arcs are added later).

The fact that the projective algorithm skips many node pairs that are considered by the non-projective algorithm makes it more efficient in practice, although the worst-case time complexity remains the same. Figure 10 shows the transition sequence needed to parse the English sentence in Figure 2 with the same output as the stack-based sequences in Figures 4 and 6.

Theorem 10
The projective, list-based algorithm is correct for the class of projective dependency forests.

Transition	Configuration				
	([0],	[ ],	[1, . . . , 9],	∅	)
SHIFT$^{\lambda}$ ⟹	([0, 1],	[ ],	[2, . . . , 9],	∅	)
LEFT-ARC$^{p}_{\text{NMOD}}$ ⟹	([0],	[ ],	[2, . . . , 9],	$A_1 = \{(2, \text{NMOD}, 1)\}$	)
SHIFT$^{\lambda}$ ⟹	([0, 2],	[ ],	[3, . . . , 9],	A_1	)
LEFT-ARC$^{p}_{\text{SBJ}}$ ⟹	([0],	[ ],	[3, . . . , 9],	$A_2 = A_1 \cup \{(3, \text{SBJ}, 2)\}$	)
RIGHT-ARC$^{p}_{\text{ROOT}}$ ⟹	([0, 3],	[ ],	[4, . . . , 9],	$A_3 = A_2 \cup \{(0, \text{ROOT}, 3)\}$	)
SHIFT$^{\lambda}$ ⟹	([0, 3, 4],	[ ],	[5, . . . , 9],	A_3	)
LEFT-ARC$^{p}_{\text{NMOD}}$ ⟹	([0, 3],	[ ],	[5, . . . , 9],	$A_4 = A_3 \cup \{(5, \text{NMOD}, 4)\}$	)
RIGHT-ARC$^{p}_{\text{OBJ}}$ ⟹	([0, 3, 5],	[ ],	[6, . . . , 9],	$A_5 = A_4 \cup \{(3, \text{OBJ}, 5)\}$	)
RIGHT-ARC$^{p}_{\text{NMOD}}$ ⟹	([0, . . . , 6],	[ ],	[7, 8, 9],	$A_6 = A_5 \cup \{(5, \text{NMOD}, 6)\}$	)
SHIFT$^{\lambda}$ ⟹	([0, . . . , 7],	[ ],	[8, 9],	A_6	)
LEFT-ARC$_{\text{NMOD}}$ ⟹	([0, . . . 6],	[ ],	[8, 9],	$A_7 = A_6 \cup \{(8, \text{NMOD}, 7)\}$	)
RIGHT-ARC$^{p}_{\text{PMOD}}$ ⟹	([0, . . . , 8],	[ ],	[9],	$A_8 = A_7 \cup \{(6, \text{PMOD}, 8)\}$	)
NO-ARC^{p} ⟹	([0, . . . , 6],	[8],	[9],	A_8	)
NO-ARC^{p} ⟹	([0, 3, 5],	[6, 8],	[9],	A_8	)
NO-ARC^{p} ⟹	([0, 3],	[5, 6, 8],	[9],	A_8	)
RIGHT-ARC$^{p}_{\text{P}}$ ⟹	([0, 3, 9],	[ ],	[ ],	$A_9 = A_8 \cup \{(3, \text{P}, 9)\}$	)

Figure 10
Projective transition sequence for the English sentence in Figure 2.

Proof 10
To show the soundness of the algorithm, we show that the dependency graph defined by the initial configuration, $G_{c_0(x)} = (V, \emptyset)$, is a projective dependency forest, and that every transition preserves this property. We consider each of the relevant conditions in turn, keeping in mind that the only transitions that modify the graph are LEFT-ARC$^{p}_{l}$ and RIGHT-ARC$^{p}_{l}$.

1. ROOT: Same as Proof 1 (substitute LEFT-ARC$^{p}_{l}$ for LEFT-ARC$_{l}$).

2. SINGLE-HEAD: Same as Proof 1 (substitute LEFT-ARC$^{p}_{l}$ and RIGHT-ARC$^{p}_{l}$ for LEFT-ARC$_{l}$ and RIGHT-ARC$^{s}_{l}$, respectively).

3. ACYCLICITY: $G_{c_s(x)}$ is acyclic, and adding an arc (i, l, j) will create a cycle only if there is a directed path from j to i. In the case of LEFT-ARC$^{p}_{l}$, this would require the existence of a configuration $c_p = (\lambda_1|j, \lambda_2, i|\beta, A_{c_p})$ such that $(k, l', i) \in A_{c_p}$ (for some $k < i$ and l'), which is impossible because any transition adding an arc (k, l', i) has as a consequence that i is no longer in the buffer. In the case of RIGHT-ARC$^{p}_{l}$, this would require a configuration $c_p = (\lambda_1|i, \lambda_2, j|\beta, A_{c_p})$ such that, given the arcs in A_{c_p}, there is a directed path from j to i. Such a path would have to involve at least one arc (k, l', k') such that $k' \leq i < k$, which would entail that i is no longer in λ_1 or λ_2. (If $k' = i$, then i would be removed from λ_1—and not added to λ_2—in the LEFT-ARC$^{p}_{l'}$ transition adding the arc; if $k' < i$, then i would have to be moved to λ_2 before the arc can be added and removed as this list is emptied in the LEFT-ARC$^{p}_{l'}$ transition.)

4. PROJECTIVITY: $G_{c_s(x)}$ is projective, and adding an arc (i, l, j) will make the graph non-projective only if there is a node k such that $i < k < j$ or $j < k < i$ and neither $i \rightarrow^* k$ nor $j \rightarrow^* k$. Let $C_{0,m}$ be a configuration sequence for $x = (w_0, w_1, \ldots, w_n)$ and let $\Pi(p, i, j)$ (for $0 < p < m$, $0 \leq i < j \leq n$) be the claim that, for every k such that $i < k < j$, $i \rightarrow^* k$ or $j \rightarrow^* k$ in G_{c_p}. To prove that no arc can be non-projective, we need to prove that, if $c_p \in C_{0,m}$ and $c_p = (\lambda_1|i, \lambda_2, j|\beta, A_{c_p})$, then $\Pi(p, i, j)$. (If $c_p = (\lambda_1|i, \lambda_2, j|\beta, A_{c_p})$ and $\Pi(p, i, j)$, then $\Pi(p', i, j)$ for all p' such that $p < p'$, because in c_p every node k such that $i < k < j$ must already have a head.) We prove this by induction over the number $\Delta(p)$ of transitions leading to c_p from the first configuration $c_{p-\Delta(p)} \in C_{0,m}$ such that $c_{p-\Delta(p)} = (\lambda_1, \lambda_2, j|\beta, A_{c_{p-\Delta(p)}})$ (i.e., the first configuration where j is the first node in the buffer).

   Basis: If $\Delta(p) = 0$, then i and j are adjacent and $\Pi(p, i, j)$ holds vacuously.

   Inductive step: Assume that $\Pi(p, i, j)$ holds if $\Delta(p) \leq q$ (for some $q > 0$) and that $\Delta(p) = q + 1$. Now consider the transition t_p that results in configuration c_p. For the projective, list-based algorithm, there are only two cases to consider, because if t_p = RIGHT-ARC$_l^p$ (for some l) or t_p = SHIFT then $\Delta(p) = 0$, which contradicts our assumption that $\Delta(p) > q > 0$. (This follows because there is no transition that moves a node back to the buffer.)

      Case 1: If t_p = LEFT-ARC$_l^p$ (for some l), then there is a node k such that $i < k < j$, $(j, l, k) \in A_{c_p}$, $c_{p-1} = (\lambda_1|i|k, \lambda_2, j|\beta, A_{c_p} - \{(j, l, k)\})$, and $c_p = (\lambda_1|i, [\,], j|\beta, A_{c_p})$. Because $\Delta(p-1) \leq q$, we can use the inductive hypothesis to infer $\Pi(p-1, k, j)$ and, from this, $\Pi(p, k, j)$. Moreover, because there has to be an earlier configuration c_{p-r} ($r < \Delta(p)$) such that $c_{p-r} = (\lambda_1|i, \lambda_{2'}, k|\beta, A_{c_{p-r}})$ and $\Delta(p-r) \leq q$, we can use the inductive hypothesis again to infer $\Pi(p-r, i, k)$ and $\Pi(p, i, k)$. $\Pi(p, i, k)$, $\Pi(p, k, j)$, and $(j, l, k) \in A_{c_p}$ together entail $\Pi(p, i, j)$.

      Case 2: If the transition t_p = NO-ARC^p, then there is a node k such that $i < k < j$, $(i, l, k) \in A_{c_p}$, $c_{p-1} = (\lambda_1|i|k, \lambda_2, j|\beta, A_{c_p})$, and $c_p = (\lambda_1|i, k|\lambda_2, j|\beta, A_{c_p})$. Because $\Delta(p-1) \leq q$, we can again use the inductive hypothesis to infer $\Pi(p-1, k, j)$ and $\Pi(p, k, j)$. Moreover, there must be an earlier configuration c_{p-r} ($r < \Delta(p)$) such that $c_{p-r} = (\lambda_1|i, \lambda_{2'}, k|\beta, A_{c_{p-r}})$ and $\Delta(p-r) \leq q$, which entails $\Pi(p-r, i, k)$ and $\Pi(p, i, k)$. As before, $\Pi(p, i, k)$, $\Pi(p, k, j)$, and $(i, l, k) \in A_{c_p}$ together entail $\Pi(p, i, j)$.

For completeness, we need to show that for any sentence x and dependency forest $G_x = (V_x, A_x)$ for x, there is a transition sequence $C_{0,m}$ such that $G_{c_m} = G_x$. The proof is by induction on the length $|x|$ and is essentially the same as Proof 7 up to the point where we assume the existence of a transition sequence $C_{0,q}$ for the sentence $x' = (w_0, w_1, w_{p-1})$ and subgraph $G_{x'} = (V_x - \{p\}, A^{-p})$, where the terminal configuration still has the form $c_q = (\lambda_{c_q}, [\,], [\,], A^{-p})$, but where it can no longer be assumed that $\lambda_{c_q} = [0, 1, \ldots, p-1]$. If i is a root in $G_{x'}$ then $i \in \lambda_{c_q}$; but if $i \in \lambda_{c_q}$ then i is *either* a root *or* has a head j such

that $j < i$ in $G_{x'}$. (This is because a RIGHT-ARC$_l^p$ transition leaves the dependent in λ_1 while a LEFT-ARC$_l^p$ removes it.) Moreover, for any $i \in \lambda_{c_q}$ that is a root in $G_{x'}$, if i is a dependent of p in G_x then any j such that $j \in \lambda_{c_q}$, $i < j$ and j is a root in $G_{x'}$ must also be a dependent of p in G_x (else G_x would fail to be projective). Finally, if p has a head k in G_x, then k must be in λ_{c_q} and any j such that $j \in \lambda_{c_q}$ and $k < j$ must either be a dependent of p in G_x or must have a head to the left in both $G_{x'}$ and G_x (anything else would again be inconsistent with the assumption that G_x is projective). Therefore, we can construct a transition sequence $C_{0,m}$ such that $G_{c_m} = G_x$, by starting in $c_0 = c_s(x)$ and applying exactly the same q transitions as in $C_{0,q}$, followed by as many LEFT-ARC$_l^p$ transitions as there are left dependents of p in G_x, interleaving NO-ARC^p transitions whenever the node at the head of λ_1 already has a head, followed by a RIGHT-ARC$_l^p$ transition if p has a head in G_x. One final SHIFT^n transition takes us to the terminal configuration $c_m = (\lambda_{c_m}, [\,], [\,], A_x)$. ■

Theorem 11
The worst-case time complexity of the projective, list-based algorithm is $O(n^2)$, where n is the length of the input sentence.

Proof 11
Same as Proof 8. ■

Theorem 12
The worst-case space complexity of the projective, list-based algorithm is $O(n)$, where n is the length of the input sentence.

Proof 12
Same as Proof 9. ■

6. Experimental Evaluation

We have defined four different transition systems for incremental dependency parsing, proven their correctness for different classes of dependency graphs, and analyzed their time and space complexity under the assumption that there exists a constant-time oracle for predicting the next transition. In this section, we present an experimental evaluation of the accuracy and efficiency that can be achieved with these systems in deterministic data-driven parsing, that is, when the oracle is approximated by a classifier trained on treebank data. The purpose of the evaluation is to compare the performance of the four algorithms under realistic conditions, thereby complementing the purely formal analysis presented so far. The purpose is not to produce state-of-the-art results for all algorithms on the data sets used, which would require extensive experimentation and optimization going well beyond the limits of this study.

6.1 Experimental Setup

The data sets used are taken from the CoNLL-X shared task on multilingual dependency parsing (Buchholz and Marsi 2006). We have used all the available data sets, taken

Table 1
Data sets. Tok = number of tokens (×1000); Sen = number of sentences (×1000); T/S = tokens per sentence (mean); Lem = lemmatization present; CPoS = number of coarse-grained part-of-speech tags; PoS = number of (fine-grained) part-of-speech tags; MSF = number of morphosyntactic features (split into atoms); Dep = number of dependency types; NPT = proportion of non-projective dependencies/tokens (%); NPS = proportion of non-projective dependency graphs/sentences (%).

Language	Tok	Sen	T/S	Lem	CPoS	PoS	MSF	Dep	NPT	NPS
Arabic	54	1.5	37.2	yes	14	19	19	27	0.4	11.2
Bulgarian	190	14.4	14.8	no	11	53	50	18	0.4	5.4
Chinese	337	57.0	5.9	no	22	303	0	82	0.0	0.0
Czech	1,249	72.7	17.2	yes	12	63	61	78	1.9	23.2
Danish	94	5.2	18.2	no	10	24	47	52	1.0	15.6
Dutch	195	13.3	14.6	yes	13	302	81	26	5.4	36.4
German	700	39.2	17.8	no	52	52	0	46	2.3	27.8
Japanese	151	17.0	8.9	no	20	77	0	7	1.1	5.3
Portuguese	207	9.1	22.8	yes	15	21	146	55	1.3	18.9
Slovene	29	1.5	18.7	yes	11	28	51	25	1.9	22.2
Spanish	89	3.3	27.0	yes	15	38	33	21	0.1	1.7
Swedish	191	11.0	17.3	no	37	37	0	56	1.0	9.8
Turkish	58	5.0	11.5	yes	14	30	82	25	1.5	11.6

from treebanks of thirteen different languages with considerable typological variation. Table 1 gives an overview of the training data available for each language.

For data sets that include a non-negligible proportion of non-projective dependency graphs, it can be expected that the non-projective list-based algorithm will achieve higher accuracy than the strictly projective algorithms. In order to make the comparison more fair, we therefore also evaluate pseudo-projective versions of the latter algorithms, making use of graph transformations in pre- and post-processing to recover non-projective dependency arcs, following Nivre and Nilsson (2005). For each language, seven different parsers were therefore trained as follows:

1. For the non-projective list-based algorithm, one parser was trained without preprocessing the training data.
2. For the three projective algorithms, two parsers were trained after preprocessing the training data as follows:
   (a) For the strictly projective parser, non-projective dependency graphs in the training data were transformed by lifting non-projective arcs to the nearest permissible ancestor of the real head. This corresponds to the *Baseline* condition in Nivre and Nilsson (2005).
   (b) For the pseudo-projective parser, non-projective dependency graphs in the training data were transformed by lifting non-projective arcs to the nearest permissible ancestor of the real head, and augmenting the arc label with the label of the real head. The output of this parser was post-processed by lowering dependency arcs with augmented labels using a top-down, left-to-right, breadth-first search for the first descendant of the head that matches the augmented arc label. This corresponds to the *Head* condition in Nivre and Nilsson (2005).

Table 2
Feature models. Rows represent tokens defined relative to the current configuration ($\mathcal{L}[i]$ = ith element of list/stack $\mathcal{L}$ of length n; $hd(x)$ = head of x; $ld(x)$ = leftmost dependent of x; $rd(x)$ = rightmost dependent of x). Columns represent attributes of tokens (Form = word form; Lem = lemma; CPoS = coarse part-of-speech; FPoS = fine part-of-speech; Feats = morphosyntactic features; Dep = dependency label). Filled cells represent features used by one or more algorithms (All = all algorithms; S = arc-standard, stack-based; E = arc-eager, stack-based; N = non-projective, list-based; P = projective, list-based).

	Attributes					
Tokens	**Form**	**Lem**	**CPoS**	**FPoS**	**Feats**	**Dep**
$\beta[0]$	All	All	All	All	All	N
$\beta[1]$	All			All		
$\beta[2]$				All		
$\beta[3]$				All		
$ld(\beta[0])$						All
$rd(\beta[0])$						S
$\sigma[0]$	SE	SE	SE	SE	SE	E
$\sigma[1]$				SE		
$hd(\sigma[0])$	E					
$ld(\sigma[0])$						SE
$rd(\sigma[0])$						SE
$\lambda_1[0]$	NP	NP	NP	NP	NP	NP
$\lambda_1[1]$				NP		
$hd(\lambda_1[0])$	NP					
$ld(\lambda_1[0])$						NP
$rd(\lambda_1[0])$						NP
$\lambda_2[0]$				N		
$\lambda_2[n]$				N		

All parsers were trained using the freely available MaltParser system,[6] which provides implementations of all the algorithms described in Sections 4 and 5. MaltParser also incorporates the LIBSVM library for support vector machines (Chang and Lin 2001), which was used to train classifiers for predicting the next transition. Training data for the classifiers were generated by parsing each sentence in the training set using the gold-standard dependency graph as an oracle. For each transition $t(c)$ in the oracle parse, a training instance $(\Phi(c), t)$ was created, where $\Phi(c)$ is a feature vector representation of the parser configuration c. Because the purpose of the experiments was not to optimize parsing accuracy as such, no work was done on feature selection for the different algorithms and languages. Instead, all parsers use a variant of the simple feature model used for parsing English and Swedish in Nivre (2006b), with minor modifications to suit the different algorithms.

Table 2 shows the feature sets used for different parsing algorithms.[7] Each row represents a node defined relative to the current parser configuration, where nodes

6 Available at `http://w3.msi.vxu.se/users/nivre/research/MaltParser.html`.

7 For each of the three projective algorithms, the strictly projective and the pseudo-projective variants used exactly the same set of features, although the set of values for the dependency label features were different because of the augmented label set introduced by the pseudo-projective technique.

defined relative to the stack σ are only relevant for stack-based algorithms, whereas nodes defined relative to the lists λ_1 and λ_2 are only relevant for list-based algorithms. We use the notation $\mathcal{L}[i]$, for arbitrary lists or stacks, to denote the ith element of $\mathcal{L}$, with $\mathcal{L}[0]$ for the first element (top element of a stack) and $\mathcal{L}[n]$ for the last element. Nodes defined relative to the partially-built dependency graph make use of the operators *hd*, *ld*, and *rd*, which return, respectively, the head, the leftmost dependent, and the rightmost dependent of a node in the dependency graph G_c defined by the current configuration c, if such a node exists, and a null value otherwise. The columns in Table 2 represent attributes of nodes (tokens) in the input (word form, lemma, coarse part-of-speech, fine part-of-speech, morphosyntactic features) or in the partially-built dependency graph (dependency label), which can be used to define features. Each cell in the table thus represents a feature $f_{ij} = a_j(n_i)$, defined by selecting the attribute a_j in the jth column from the node n_i characterized in the ith row. For example, the feature f_{11} is the word form of the first input node (token) in the buffer β. The symbols occurring in filled cells indicate for which parsing algorithms the feature is active, where *S* stands for arc-standard stack-based, *E* for arc-eager stack-based, *N* for non-projective list-based, and *P* for projective list-based. Features that are used for some but not all algorithms are typically not meaningful for all algorithms. For example, a right dependent of the first node in the buffer β can only exist (at decision time) when using the arc-standard stack-based algorithm. Hence, this feature is inactive for all other algorithms.

The SVM classifiers were trained with a quadratic kernel $K(x_i, x_j) = (\gamma x_i^T x_j + r)^2$ and LIBSVM's built-in one-versus-one strategy for multi-class classification, converting symbolic features to numerical ones using the standard technique of binarization. The parameter settings were $\gamma = 0.2$ and $r = 0$ for the kernel parameters, $C = 0.5$ for the penalty parameter, and $\epsilon = 1.0$ for the termination criterion. These settings were extrapolated from many previous experiments under similar conditions, using cross-validation or held-out subsets of the training data for tuning, but in these experiments they were kept fixed for all parsers and languages. In order to reduce training times, the set of training instances derived from a given training set was split into smaller sets, for which separate multi-class classifiers were trained, using FPoS(β[0]), that is, the (fine-grained) part of speech of the first node in the buffer, as the defining feature for the split.

The seven different parsers for each language were evaluated by running them on the dedicated test set from the CoNLL-X shared task, which consists of approximately 5,000 tokens for all languages. Because the dependency graphs in the gold standard are always trees, each output graph was converted, if necessary, from a forest to a tree by attaching every root node i ($i > 0$) to the special root node 0 with a default label ROOT. Parsing accuracy was measured by the **labeled attachment score** (LAS), that is, the percentage of tokens that are assigned the correct head and dependency label, as well as the **unlabeled attachment score** (UAS), that is, the percentage of tokens with the correct head, and the **label accuracy** (LA), that is, the percentage of tokens with the correct dependency label. All scores were computed with the scoring software from the CoNLL-X shared task, *eval.pl*, with default settings. This means that punctuation tokens are excluded in all scores. In addition to parsing accuracy, we evaluated efficiency by measuring the learning time and parsing time in seconds for each data set.

Before turning to the results of the evaluation, we need to fulfill the promise from Remarks 1 and 2 to discuss the way in which treebank-induced classifiers approximate oracles and to what extent they satisfy the condition of constant-time operation that was assumed in all the results on time complexity in Sections 4 and 5. When predicting the next transition at run-time, there are two different computations that take

place: the first is the classifier returning a transition t as the output class for an input feature vector $\Phi(c)$, and the second is a check whether the preconditions of t are satisfied in c. If the preconditions are satisfied, the transition t is performed; otherwise a default transition (with no preconditions) is performed instead.[8] (The default transition is SHIFT for the stack-based algorithms and NO-ARC for the list-based algorithms.) The time required to compute the classification t of $\Phi(c)$ depends on properties of the classifier, such as the number of support vectors and the number of classes for a multi-class SVM classifier, but is independent of the length of the input and can therefore be regarded as a constant as far as the time complexity of the parsing algorithm is concerned.[9] The check of preconditions is a trivial constant-time operation in all cases except one, namely the need to check whether there is a path between two nodes for the LEFT-ARC$_l^n$ and RIGHT-ARC$_l^n$ transitions of the non-projective list-based algorithm. Maintaining the information needed for this check and updating it with each addition of a new arc to the graph is equivalent to the union-find operations for disjoint set data structures. Using the techniques of path compression and union by rank, the amortized time per operation is $O(\alpha(n))$ per operation, where n is the number of elements (nodes in this case) and $\alpha(n)$ is the inverse of the Ackermann function, which means that $\alpha(n)$ is less than 5 for all remotely practical values of n and is effectively a small constant (Cormen, Leiserson, and Rivest 1990). With this proviso, all the complexity results from Sections 4 and 5 can be regarded as valid also for the classifier-based implementation of deterministic, incremental dependency parsing.

6.2 Parsing Accuracy

Table 3 shows the parsing accuracy obtained for each of the 7 parsers on each of the 13 languages, as well as the average over all languages, with the top score in each row set in boldface. For comparison, we also include the results of the two top scoring systems in the CoNLL-X shared task, those of McDonald, Lerman, and Pereira (2006) and Nivre et al. (2006). Starting with the LAS, we see that the multilingual average is very similar across the seven parsers, with a difference of only 0.58 percentage points between the best and the worst result, obtained with the non-projective and the strictly projective version of the list-based parser, respectively. However, given the large amount of data, some of the differences are nevertheless statistically significant (according to McNemar's test, $\alpha = .05$). Broadly speaking, the group consisting of the non-projective, list-based parser and the three pseudo-projective parsers significantly outperforms the group consisting of the three projective parsers, whereas there are no significant differences within the two groups.[10] This shows that the capacity to capture non-projective dependencies does make a significant difference, even though such dependencies are infrequent in most languages.

The best result is about one percentage point below the top scores from the original CoNLL-X shared task, but it must be remembered that the results in this article have

8 A more sophisticated strategy would be to back off to the second best choice of the oracle, assuming that the oracle provides a ranking of all the possible transitions. On the whole, however, classifiers very rarely predict transitions that are not legal in the current configuration.

9 The role of this constant in determining the overall running time is similar to that of a grammar constant in grammar-based parsing.

10 The only exception to this generalization is the pseudo-projective, list-based parser, which is significantly worse than the non-projective, list-based parser, but not significantly better than the projective, arc-standard, stack-based parser.

Table 3
Parsing accuracy for 7 parsers on 13 languages, measured by labeled attachment score (LAS), unlabeled attachment score (UAS) and label accuracy (LA). NP-L = non-projective list-based; P-L = projective list-based; PP-L = pseudo-projective list-based; P-E = projective arc-eager stack-based; PP-E = pseudo-projective arc-eager stack-based; P-S = projective arc-standard stack-based; PP-S = pseudo-projective arc-standard stack-based; McD = McDonald, Lerman and Pereira (2006); Niv = Nivre et al. (2006).

Labeled Attachment Score (LAS)

Language	NP-L	P-L	PP-L	P-E	PP-E	P-S	PP-S	McD	Niv
Arabic	63.25	63.19	63.13	64.93	64.95	65.79	**66.05**	66.91	66.71
Bulgarian	**87.79**	87.75	87.39	87.75	87.41	86.42	86.71	87.57	87.41
Chinese	85.77	85.96	85.96	85.96	85.96	**86.00**	**86.00**	85.90	86.92
Czech	78.12	76.24	78.04	76.34	77.46	78.18	**80.12**	80.18	78.42
Danish	**84.59**	84.15	84.35	84.25	84.45	84.17	84.15	84.79	84.77
Dutch	**77.41**	74.71	76.95	74.79	76.89	73.27	74.79	79.19	78.59
German	84.42	84.21	84.38	84.23	84.46	**84.58**	**84.58**	87.34	85.82
Japanese	**90.97**	90.57	90.53	90.83	90.89	90.59	90.63	90.71	91.65
Portuguese	**86.70**	85.91	86.20	85.83	86.12	85.39	86.09	86.82	87.60
Slovene	70.06	69.88	70.12	69.50	70.22	**72.00**	71.88	73.44	70.30
Spanish	**80.18**	79.80	79.60	79.84	79.60	78.70	78.42	82.25	81.29
Swedish	**83.03**	82.63	82.41	82.63	82.41	82.12	81.54	82.55	84.58
Turkish	64.69	64.49	64.37	64.37	64.43	64.67	**64.73**	63.19	65.68
Average	**79.77**	79.19	79.49	79.33	79.63	79.38	79.67	80.83	80.75

Unlabeled Attachment Score (UAS)

Language	NP-L	P-L	PP-L	P-E	PP-E	P-S	PP-S	McD	Niv
Arabic	76.43	76.19	76.49	76.31	76.25	77.76	**77.98**	79.34	77.52
Bulgarian	91.68	**91.92**	91.62	**91.92**	91.64	90.80	91.10	92.04	91.72
Chinese	89.42	90.24	90.24	90.24	90.24	**90.42**	**90.42**	91.07	90.54
Czech	84.88	82.82	84.58	82.58	83.66	84.50	**86.44**	87.30	84.80
Danish	**89.76**	89.40	89.42	89.52	89.52	89.26	89.38	90.58	89.80
Dutch	**80.25**	77.29	79.59	77.25	79.47	75.95	78.03	83.57	81.35
German	**87.80**	87.10	87.38	87.14	87.42	87.46	87.44	90.38	88.76
Japanese	92.64	92.60	92.56	**92.80**	92.72	92.50	92.54	92.84	93.10
Portuguese	**90.56**	89.82	90.14	89.74	90.08	89.00	89.64	91.36	91.22
Slovene	79.06	78.78	79.06	78.42	78.98	80.72	**80.76**	83.17	78.72
Spanish	83.39	83.75	83.65	**83.79**	83.65	82.35	82.13	86.05	84.67
Swedish	**89.54**	89.30	89.03	89.30	89.03	88.81	88.39	88.93	89.50
Turkish	75.12	75.24	75.02	75.32	74.81	**75.94**	75.76	74.67	75.82
Average	**85.43**	84.96	85.29	84.95	85.19	85.04	85.39	87.02	85.96

Label Accuracy (LA)

Language	NP-L	P-L	PP-L	P-E	PP-E	P-S	PP-S	McD	Niv
Arabic	75.93	76.15	76.09	78.46	78.04	**79.30**	79.10	79.50	80.34
Bulgarian	**90.68**	90.68	90.40	90.68	90.42	89.53	89.81	90.70	90.44
Chinese	88.01	87.95	87.95	87.95	87.95	**88.33**	**88.33**	88.23	89.01
Czech	84.54	84.54	84.42	84.80	84.66	86.00	**86.58**	86.72	85.40
Danish	88.90	88.60	88.76	88.62	88.82	**89.00**	88.98	89.22	89.16
Dutch	82.43	**82.59**	82.13	82.57	82.15	80.71	80.13	83.89	83.69
German	89.74	90.08	89.92	90.06	89.94	**90.79**	90.36	92.11	91.03
Japanese	93.54	93.14	93.14	93.54	**93.56**	93.12	93.18	93.74	93.34
Portuguese	**90.56**	90.54	90.34	90.46	90.26	90.42	90.26	90.46	91.54
Slovene	78.88	79.70	79.40	79.32	79.48	**81.61**	81.37	82.51	80.54
Spanish	**89.46**	89.04	88.66	89.06	88.66	88.30	88.12	90.40	90.06
Swedish	**85.50**	85.40	84.92	85.40	84.92	84.66	84.01	85.58	87.39
Turkish	**77.79**	77.32	77.02	77.00	77.39	77.02	77.20	77.45	78.49
Average	85.84	85.83	85.63	85.99	85.87	**86.06**	85.96	86.96	87.03

been obtained without optimization of feature representations or learning algorithm parameters. The net effect of this can be seen in the result for the pseudo-projective version of the arc-eager, stack-based parser, which is identical to the system used by Nivre et al. (2006), except for the lack of optimization, and which suffers a loss of 1.12 percentage points overall.

The results for UAS show basically the same pattern as the LAS results, but with even less variation between the parsers. Nevertheless, there is still a statistically significant margin between the non-projective, list-based parser and the three pseudo-projective parsers, on the one hand, and the strictly projective parsers, on the other.[11] For **label accuracy** (LA), finally, the most noteworthy result is that the strictly projective parsers consistently outperform their pseudo-projective counterparts, although the difference is statistically significant only for the projective, list-based parser. This can be explained by the fact that the pseudo-projective parsing technique increases the number of distinct dependency labels, using labels to distinguish not only between different syntactic functions but also between "lifted" and "unlifted" arcs. It is therefore understandable that the pseudo-projective parsers suffer a drop in pure labeling accuracy.

Despite the very similar performance of all parsers on average over all languages, there are interesting differences for individual languages and groups of languages. These differences concern the impact of non-projective, pseudo-projective, and strictly projective parsing, on the one hand, and the effect of adopting an arc-eager or an arc-standard parsing strategy for the stack-based parsers, on the other. Before we turn to the evaluation of efficiency, we will try to analyze some of these differences in a little more detail, starting with the different techniques for capturing non-projective dependencies.

First of all, we may observe that the non-projective, list-based parser outperforms its strictly projective counterpart for all languages except Chinese. The result for Chinese is expected, given that it is the only data set that does not contain any non-projective dependencies, but the difference in accuracy is very slight (0.19 percentage points). Thus, it seems that the non-projective parser can also be used without loss in accuracy for languages with very few non-projective structures. The relative improvement in accuracy for the non-projective parser appears to be roughly linear in the percentage of non-projective dependencies found in the data set, with a highly significant correlation (Pearson's $r = 0.815$, $p = 0.0007$). The only language that clearly diverges from this trend is German, where the relative improvement is much smaller than expected.

If we compare the non-projective, list-based parser to the strictly projective stack-based parsers, we see essentially the same pattern but with a little more variation. For the arc-eager, stack-based parser, the only anomaly is the result for Arabic, which is significantly higher than the result for the non-projective parser, but this seems to be due to a particularly bad performance of the list-based parsers as a group for this language.[12] For the arc-standard, stack-based parser, the data is considerably more noisy, which is related to the fact that the arc-standard parser in itself has a higher

11 The exception this time is the pseudo-projective, arc-eager parser, which has a statistically significant difference up to the non-projective parser but a non-significant difference down to the projective, arc-standard parser.

12 A possible explanation for this result is the extremely high average sentence length for Arabic, which leads to a greater increase in the number of potential arcs considered for the list-based parsers than for the stack-based parsers.

variance than the other parsers, an observation that we will return to later on. Still, the correlation between relative improvement in accuracy and percentage of non-projective dependencies is significant for both the arc-eager parser ($r = 0.766$, $p = 0.001$) and the arc-standard parser ($r = 0.571$, $p = 0.02$), although clearly not as strong as for the list-based parser. It therefore seems reasonable to conclude that the non-projective parser in general can be expected to outperform a strictly projective parser with a margin that is directly related to the proportion of non-projective dependencies in the data.

Having compared the non-projective, list-based parser to the strictly projective parsers, we will now scrutinize the results obtained when coupling the projective parsers with the pseudo-projective parsing technique, as an alternative method for capturing non-projective dependencies. The overall pattern is that pseudo-projective parsing improves the accuracy of a projective parser for languages with more than 1% of non-projective dependencies, as seen from the results for Czech, Dutch, German, and Portuguese. For these languages, the pseudo-projective parser is never outperformed by its strictly projective counterpart, and usually does considerably better, although the improvements for German are again smaller than expected. For Slovene and Turkish, we find improvement only for two out of three parsers, despite a relatively high share of non-projective dependencies (1.9% for Slovene, 1.5% for Turkish). Given that Slovene and Turkish have the smallest training data sets of all languages, this is consistent with previous studies showing that pseudo-projective parsing is sensitive to data sparseness (Nilsson, Nivre, and Hall 2007). For languages with a lower percentage of non-projective dependencies, the pseudo-projective technique seems to hurt performance more often than not, possibly as a result of decreasing the labeling accuracy, as noted previously. It is worth noting that Chinese is a special case in this respect. Because there are no non-projective dependencies in this data set, the projectivized training data set will be identical to the original one, which means that the pseudo-projective parser will behave exactly as the projective one.

Comparing non-projective parsing to pseudo-projective parsing, it seems clear that both can improve parsing accuracy in the presence of significant amounts of non-projective dependencies, but the former appears to be more stable in that it seldom or never hurts performance, whereas the latter can be expected to have a negative effect on accuracy when the amount of training data or non-projective dependencies (or both) is not high enough. Moreover, the non-projective parser tends to outperform the best pseudo-projective parsers, both on average and for individual languages. In fact, the pseudo-projective technique outperforms the non-projective parser only in combination with the arc-standard, stack-based parsing algorithm, and this seems to be due more to the arc-standard parsing strategy than to the pseudo-projective technique as such. The relevant question here is therefore why arc-standard parsing seems to work particularly well for some languages, with or without pseudo-projective parsing.

Going through the results for individual languages, it is clear that the arc-standard algorithm has a higher variance than the other algorithms. For Bulgarian, Dutch, and Spanish, the accuracy is considerably lower than for the other algorithms, in most cases by more than one percentage point. But for Arabic, Czech, and Slovene, we find exactly the opposite pattern, with the arc-standard parsers sometimes outperforming the other parsers by more than two percentage points. For the remaining languages, the arc-standard algorithm performs on a par with the other algorithms.[13] In order to

13 The arc-standard algorithm achieves the highest score also for Chinese, German, and Turkish, but in these cases only by a small margin.

explain this pattern we need to consider the way in which properties of the algorithms interact with properties of different languages and the way they have been annotated syntactically.

First of all, it is important to note that the two list-based algorithms and the arc-eager variant of the stack-based algorithm are all arc-eager in the sense that an arc (i, l, j) is always added at the earliest possible moment, that is, in the first configuration where i and j are the target tokens. For the arc-standard stack-based parser, this is still true for left dependents (i.e., arcs (i, l, j) such that $j < i$) but not for right dependents, where an arc (i, l, j) $(i < j)$ should be added only at the point where all arcs of the form (j, l', k) have already been added (i.e., when the dependent j has already found all its dependents). This explains why the results for the two list-based parsers and the arc-eager stack-based parser are so well correlated, but it does not explain why the arc-standard strategy works better for some languages but not for others.

The arc-eager strategy has an advantage in that a right dependent j can be attached to its head i at any time without having to decide whether j itself should have a right dependent. By contrast, with the arc-standard strategy it is necessary to decide not only whether j is a right dependent of i but also whether it should be added now or later, which means that two types of errors are possible even when the decision to attach j to i is correct. Attaching too early means that right dependents can never be attached to j; postponing the attachment too long means that j will never be added to i. None of these errors can occur with the arc-eager strategy, which therefore can be expected to work better for data sets where this kind of "ambiguity" is commonly found. In order for this to be the case, there must first of all be a significant proportion of left-headed structures in the data. Thus, we find that in all the data sets for which the arc-standard parsers do badly, the percentage of left-headed dependencies is in the 50–75% range. However, it must also be pointed out that the highest percentage of all is found in Arabic (82.9%), which means that a high proportion of left-headed structures may be a necessary but not sufficient condition for the arc-eager strategy to work better than the arc-standard strategy. We conjecture that an additional necessary condition is an annotation style that favors more deeply embedded structures, giving rise to chains of left-headed structures where each node is dependent on the preceding one, which increases the number of points at which an incorrect decision can be made by an arc-standard parser. However, we have not yet fully verified the extent to which this condition holds for all the data sets where the arc-eager parsers outperform their arc-standard counterparts.

Although the arc-eager strategy has an advantage in that the decisions involved in attaching a right dependent are simpler, it has the disadvantage that it has to commit early. This may either lead the parser to add an arc (i, l, j) $(i < j)$ when it is not correct to do so, or fail to add the same arc in a situation where it should have been added, in both cases because the information available at an early point makes the wrong decision look probable. In the first case, the arc-standard parser may still get the analysis right, if it also seems probable that j should have a right dependent (in which case it will postpone the attachment); in the second case, it may get a second chance to add the arc if it in fact adds a right dependent to j at a later point. It is not so easy to predict what type of structures and annotation will favor the arc-standard parser in this way, but it is likely that having many right dependents attached to (or near) the root could cause problems for the arc-eager algorithms, since these dependencies determine the global structure and often span long distances, which makes it harder to make correct decisions early in the parsing process. This is consistent with earlier studies showing that parsers using the arc-eager, stack-based algorithm tend to predict dependents of the root with

lower precision than other algorithms.[14] Interestingly, the three languages for which the arc-standard parser has the highest improvement (Arabic, Czech, Slovene) have a very similar annotation, based on the Prague school tradition of dependency grammar, which not only allows multiple dependents of the root but also uses several different labels for these dependents, which means that they will be analyzed correctly only if a RIGHT-ARC transition is performed with the right label at exactly the right point in time. This is in contrast to annotation schemes that use a default label ROOT, for dependents of the root, where such dependents can often be correctly recovered in post-processing by attaching all remaining roots to the special root node with the default label. We can see the effect of this by comparing the two stack-based parsers (in their pseudo-projective versions) with respect to precision and recall for the dependency type PRED (predicate), which is the most important label for dependents of the root in the data sets for Arabic, Czech, and Slovene. While the arc-standard parser has 78.02% precision and 70.22% recall, averaged over the three languages, the corresponding figures for the arc-eager parser are as low as 68.93% and 65.93%, respectively, which represents a drop of almost ten percentage points in precision and almost five percentage points in recall.

Summarizing the results of the accuracy evaluation, we have seen that all four algorithms can be used for deterministic, classifier-based parsing with competitive accuracy. The results presented are close to the state of the art without any optimization of feature representations and learning algorithm parameters. Comparing different algorithms, we have seen that the capacity to capture non-projective dependencies makes a significant difference in general, but with language-specific effects that depend primarily on the frequency of non-projective constructions. We have also seen that the non-projective list-based algorithm is more stable and predictable in this respect, compared to the use of pseudo-projective parsing in combination with an essentially projective parsing algorithm. Finally, we have observed quite strong language-specific effects for the difference between arc-standard and arc-eager parsing for the stack-based algorithms, effects that can be tied to differences in linguistic structure and annotation style between different data sets, although a much more detailed error analysis is needed before we can draw precise conclusions about the relative merits of different parsing algorithms for different languages and syntactic representations.

6.3 Efficiency

Before we consider the evaluation of efficiency in both learning and parsing, it is worth pointing out that the results will be heavily dependent on the choice of support vector machines for classification, and cannot be directly generalized to the use of deterministic incremental parsing algorithms together with other kinds of classifiers. However, because support vector machines constitute the state of the art in classifier-based parsing, it is still worth examining how learning and parsing times vary with the parsing algorithm while parameters of learning and classification are kept constant.

Table 4 gives the results of the efficiency evaluation. Looking first at learning times, it is obvious that learning time depends primarily on the number of training instances, which is why we can observe a difference of several orders of magnitude in learning time between the biggest training set (Czech) and the smallest training set (Slovene)

14 This is shown by Nivre and Scholz (2004) in comparison to the iterative, arc-standard algorithm of Yamada and Matsumoto (2003) and by McDonald and Nivre (2007) in comparison to the spanning tree algorithm of McDonald, Lerman, and Pereira (2006).

Table 4
Learning and parsing time for seven parsers on six languages, measured in seconds.
NP-L = non-projective list-based; P-L = projective list-based; PP-L = pseudo-projective list-based; P-E = projective arc-eager stack-based; PP-E = pseudo-projective arc-eager stack-based; P-S = projective arc-standard stack-based; PP-S = pseudo-projective arc-standard stack-based.

			Learning Time				
Language	**NP-L**	**P-L**	**PP-L**	**P-E**	**PP-E**	**P-S**	**PP-S**
Arabic	1,814	614	603	650	647	1,639	1,636
Bulgarian	6,796	2,918	2,926	2,919	2,939	3,321	3,391
Chinese	17,034	13,019	13,019	13,029	13,029	13,705	13,705
Czech	546,880	250,560	248,511	279,586	280,069	407,673	406,857
Danish	2,964	1,248	1,260	1,246	1,262	643	647
Dutch	7,701	3,039	2,966	3,055	2,965	7,000	6,812
German	48,699	16,874	17,600	16,899	17,601	24,402	24,705
Japanese	211	191	188	203	208	199	199
Portuguese	25,621	8,433	8,336	8,436	8,335	7,724	7,731
Slovene	167	78	90	93	99	86	90
Spanish	1,999	562	566	565	565	960	959
Swedish	2,410	942	1,020	945	1,022	1,350	1,402
Turkish	720	498	519	504	516	515	527
Average	105,713	46,849	46,616	51,695	51,876	74,798	74,691
			Parsing Time				
Language	**NP-L**	**P-L**	**PP-L**	**P-E**	**PP-E**	**P-S**	**PP-S**
Arabic	213	103	131	108	135	196	243
Bulgarian	139	93	102	93	103	135	147
Chinese	1,008	855	855	855	855	803	803
Czech	5,244	3,043	5,889	3,460	6,701	3,874	7,437
Danish	109	66	83	66	83	82	106
Dutch	349	209	362	211	363	253	405
German	781	456	947	455	945	494	1,004
Japanese	10	8	8	9	10	7	7
Portuguese	670	298	494	298	493	437	717
Slovene	69	44	62	47	65	43	64
Spanish	133	67	75	67	75	80	91
Swedish	286	202	391	201	391	242	456
Turkish	218	162	398	162	403	153	380
Average	1,240	712	1,361	782	1,496	897	1,688

for a given parsing algorithm. Broadly speaking, for any given parsing algorithm, the ranking of languages with respect to learning time follows the ranking with respect to training set size, with a few noticeable exceptions. Thus, learning times are shorter than expected, relative to other languages, for Swedish and Japanese, but longer than expected for Arabic and (except in the case of the arc-standard parsers) for Danish.

However, the number of training instances for the SVM learner depends not only on the number of tokens in the training set, but also on the number of transitions required to parse a sentence of length n. This explains why the non-projective list-based algorithm, with its quadratic complexity, consistently has longer learning times than the linear stack-based algorithms. However, it can also be noted that the projective, list-based algorithm, despite having the same worst-case complexity as the non-projective

algorithm, in practice behaves much more like the arc-eager stack-based algorithm and in fact has a slightly lower learning time than the latter on average. The arc-standard stack-based algorithm, finally, again shows much more variation than the other algorithms. On average, it is slower to train than the arc-eager algorithm, and sometimes very substantially so, but for a few languages (Danish, Japanese, Portuguese, Slovene) it is actually faster (and considerably so for Danish). This again shows that learning time depends on other properties of the training sets than sheer size, and that some data sets may be more easily separable for the SVM learner with one parsing algorithm than with another.

It is noteworthy that there are no consistent differences in learning time between the strictly projective parsers and their pseudo-projective counterparts, despite the fact that the pseudo-projective technique increases the number of distinct classes (because of its augmented arc labels), which in turn increases the number of binary classifiers that need to be trained in order to perform multi-class classification with the one-versus-one method. The number of classifiers is $\frac{m(m-1)}{2}$, where m is the number of classes, and the pseudo-projective technique with the encoding scheme used here can theoretically lead to a quadratic increase in the number of classes. The fact that this has no noticeable effect on efficiency indicates that learning time is dominated by other factors, in particular the number of training instances.

Turning to parsing efficiency, we may first note that parsing time is also dependent on the size of the training set, through a dependence on the number of support vectors, which tend to grow with the size of the training set. Thus, for any given algorithm, there is a strong tendency that parsing times for different languages follow the same order as training set sizes. The notable exceptions are Arabic, Turkish, and Chinese, which have higher parsing times than expected (relative to other languages), and Japanese, where parsing is surprisingly fast. Because these deviations are the same for all algorithms, it seems likely that they are related to specific properties of these data sets. It is also worth noting that for Arabic and Japanese the deviations are consistent across learning and parsing (slower than expected for Arabic, faster than expected for Japanese), whereas for Chinese there is no consistent trend (faster than expected in learning, slower than expected in parsing).

Comparing algorithms, we see that the non-projective list-based algorithm is slower than the strictly projective stack-based algorithms, which can be expected from the difference in time complexity. But we also see that the projective list-based algorithm, despite having the same worst-case complexity as the non-projective algorithm, in practice behaves like the linear-time algorithms and is in fact slightly faster on average than the arc-eager stack-based algorithm, which in turn outperforms the arc-standard stack-based algorithm. This is consistent with the results from oracle parsing reported in Nivre (2006a), which show that, with the constraint of projectivity, the relation between sentence length and number of transitions for the list-based parser can be regarded as linear in practice. Comparing the arc-eager and the arc-standard variants of the stack-based algorithm, we find the same kind of pattern as for learning time in that the arc-eager parser is faster for all except a small set of languages: Chinese, Japanese, Slovene, and Turkish. Only two of these, Japanese and Slovene, are languages for which learning is also faster with the stack-based algorithm, which again shows that there is no straightforward correspondence between learning time and parsing time.

Perhaps the most interesting result of all, as far as efficiency is concerned, is to be found in the often dramatic differences in parsing time between the strictly projective parsers and their pseudo-projective counterparts. Although we did not see any clear effect of the increased number of classes, hence classifiers, on learning time earlier, it is

quite clear that there is a noticeable effect on parsing time, with the pseudo-projective parsers always being substantially slower. In fact, in some cases the pseudo-projective parsers are also slower than the non-projective list-based parser, despite the difference in time complexity that exists at least for the stack-based parsers. This result holds on average over all languages and for five out of thirteen of the individual languages and shows that the advantage of linear-time parsing complexity (for the stack-based parsers) can be outweighed by the disadvantage of a more complex classification problem in pseudo-projective parsing. In other words, the larger constant associated with a larger cohort of SVM classifiers for the pseudo-projective parser can be more important than the better asymptotic complexity of the linear-time algorithm in the range of sentence lengths typically found in natural language. Looking more closely at the variation in sentence length across languages, we find that the pseudo-projective parsers are faster than the non-projective parser for all data sets with an average sentence length above 18. For data sets with shorter sentences, the non-projective parser is more efficient in all except three cases: Bulgarian, Chinese, and Japanese. For Chinese this is easily explained by the absence of non-projective dependencies, making the performance of the pseudo-projective parsers identical to their strictly projective counterparts. For the other two languages, the low number of distinct dependency labels for Japanese and the low percentage of non-projective dependencies for Bulgarian are factors that mitigate the effect of enlarging the set of dependency labels in pseudo-projective parsing. We conclude that the relative efficiency of non-projective and pseudo-projective parsing depends on several factors, of which sentence length appears to be the most important, but where the number of distinct dependency labels and the percentage of non-projective dependencies also play a role.

7. Related Work

Data-driven dependency parsing using supervised machine learning was pioneered by Eisner (1996), who showed how traditional chart parsing techniques could be adapted for dependency parsing to give efficient parsing with exact inference over a probabilistic model where the score of a dependency tree is the sum of the scores of individual arcs. This approach has been further developed in particular by Ryan McDonald and his colleagues (McDonald, Crammer, and Pereira 2005; McDonald et al. 2005; McDonald and Pereira 2006) and is now known as **spanning tree parsing**, because the problem of finding the most probable tree under this type of model is equivalent to finding an optimum spanning tree in a dense graph containing all possible dependency arcs. If we assume that the score of an individual arc is independent of all other arcs, this problem can be solved efficiently for arbitrary non-projective dependency trees using the Chu-Liu-Edmonds algorithm, as shown by McDonald et al. (2005). Spanning tree algorithms have so far primarily been combined with online learning methods such as MIRA (McDonald, Crammer, and Pereira 2005).

The approach of **deterministic classifier-based parsing** was first proposed for Japanese by Kudo and Matsumoto (2002) and for English by Yamada and Matsumoto (2003). In contrast to spanning tree parsing, this can be characterized as a greedy inference strategy, trying to construct a globally optimal dependency graph by making a sequence of locally optimal decisions. The first strictly incremental parser of this kind was described in Nivre (2003) and used for classifier-based parsing of Swedish by Nivre, Hall, and Nilsson (2004) and English by Nivre and Scholz (2004). Altogether it has now been applied to 19 different languages (Nivre et al. 2006, 2007; Hall et al. 2007). Most algorithms in this tradition are restricted to projective dependency graphs, but it is

possible to recover non-projective dependencies using **pseudo-projective parsing** (Nivre and Nilsson 2005). More recently, algorithms for non-projective classifier-based parsing have been proposed by Attardi (2006) and Nivre (2006a). The strictly deterministic parsing strategy has been relaxed in favor of *n*-best parsing by Johansson and Nugues (2006), among others. The dominant learning method in this tradition is support vector machines (Kudo and Matsumoto 2002; Yamada and Matsumoto 2003; Nivre et al. 2006) but memory-based learning has also been used (Nivre, Hall, and Nilsson 2004; Nivre and Scholz 2004; Attardi 2006).

Of the algorithms described in this article, the arc-eager stack-based algorithm is essentially the algorithm proposed for unlabeled dependency parsing in Nivre (2003), extended to labeled dependency parsing in Nivre, Hall, and Nilsson (2004), and most fully described in Nivre (2006b). The major difference is that the parser is now initialized with the special root node on the stack, whereas earlier formulations had an empty stack at initialization.[15] The arc-standard stack-based algorithm is briefly described in Nivre (2004) but can also be seen as an incremental version of the algorithm of Yamada and Matsumoto (2003), where incrementality is achieved by only allowing one left-to-right pass over the input, whereas Yamada and Matsumoto perform several iterations in order to construct the dependency graph bottom-up, breadth-first as it were. The list-based algorithms are both inspired by the work of Covington (2001), although the formulations are not equivalent. They have previously been explored for deterministic classifier-based parsing in Nivre (2006a, 2007). A more orthodox implementation of Covington's algorithms for data-driven dependency parsing is found in Marinov (2007).

8. Conclusion

In this article, we have introduced a formal framework for deterministic incremental dependency parsing, where parsing algorithms can be defined in terms of transition systems that are deterministic only together with an oracle for predicting the next transition. We have used this framework to analyze four different algorithms, proving the correctness of each algorithm relative to a relevant class of dependency graphs, and giving complexity results for each algorithm.

To complement the formal analysis, we have performed an experimental evaluation of accuracy and efficiency, using SVM classifiers to approximate oracles, and using data from 13 languages. The comparison shows that although strictly projective dependency parsing is most efficient both in learning and in parsing, the capacity to produce non-projective dependency graphs leads to better accuracy unless it can be assumed that all structures are strictly projective. The evaluation also shows that using the non-projective, list-based parsing algorithm gives a more stable improvement in this respect than applying the pseudo-projective parsing technique to a strictly projective parsing algorithm. Moreover, despite its quadratic time complexity, the non-projective parser is often as efficient as the pseudo-projective parsers in practice, because the extended set of dependency labels used in pseudo-projective parsing slows down classification. This demonstrates the importance of complementing the theoretical analysis of complexity with practical running time experiments.

Although the non-projective, list-based algorithm can be said to give the best trade-off between accuracy and efficiency when results are averaged over all languages in the sample, we have also observed important language-specific effects. In particular, the

15 The current version was first used in the CoNLL-X shared task (Nivre et al. 2006).

arc-eager strategy inherent not only in the arc-eager, stack-based algorithm but also in both versions of the list-based algorithm appears to be suboptimal for some languages and syntactic representations. In such cases, using the arc-standard parsing strategy, with or without pseudo-projective parsing, may lead to significantly higher accuracy. More research is needed to determine exactly which properties of linguistic structures and their syntactic analysis give rise to these effects.

On the whole, however, the four algorithms investigated in this article give very similar performance both in terms of accuracy and efficiency, and several previous studies have shown that both the stack-based and the list-based algorithms can achieve state-of-the-art accuracy together with properly trained classifiers (Nivre et al. 2006; Nivre 2007; Hall et al. 2007).

Acknowledgments

I want to thank my students Johan Hall and Jens Nilsson for fruitful collaboration and for their contributions to the MaltParser system, which was used for all experiments. I also want to thank Sabine Buchholz, Matthias Buch-Kromann, Walter Daelemans, Gülşen Eryiğit, Jason Eisner, Jan Hajič, Sandra Kübler, Marco Kuhlmann, Yuji Matsumoto, Ryan McDonald, Kemal Oflazer, Kenji Sagae, Noah A. Smith, and Deniz Yuret for useful discussions on topics relevant to this article. I am grateful to three anonymous reviewers for many helpful suggestions that helped improve the final version of the article. The work has been partially supported by the Swedish Research Council.

References

Abney, Steven and Mark Johnson. 1991. Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research*, 20:233–250.

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

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

Böhmová, Alena, Jan Hajič, Eva Hajičová, and Barbora Hladká. 2003. The Prague Dependency Treebank: A three-level annotation scenario. In Anne Abeillé, editor, *Treebanks: Building and Using Parsed Corpora*. Kluwer Academic Publishers, Dordrecht, pages 103–127.

Buchholz, Sabine and Erwin Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning*, pages 149–164, New York.

Chang, Chih-Chung and Chih-Jen Lin, 2001. *LIBSVM: A Library for Support Vector Machines*. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

Cheng, Yuchang, Masayuki Asahara, and Yuji Matsumoto. 2005. Machine learning-based dependency analyzer for Chinese. In *Proceedings of International Conference on Chinese Computing (ICCC)*, pages 66–73, Singapore.

Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

Covington, Michael A. 2001. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102, Athens, GA.

Eisner, Jason M. 1996. Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING)*, pages 340–345, Copenhagen.

Hajič, Jan, Barbora Vidova Hladka, Jarmila Panevová, Eva Hajičová, Petr Sgall, and Petr Pajas. 2001. Prague Dependency Treebank 1.0. LDC, 2001T10.

Hall, J., J. Nilsson, J. Nivre, G. Eryiğit, B. Megyesi, M. Nilsson, and M. Saers. 2007. Single malt or blended? A study in multilingual parser optimization. In *Proceedings of the CoNLL shared task of EMNLP-CoNLL 2007*, pages 933–939, Prague.

Hall, Johan, Joakim Nivre, and Jens Nilsson. 2006. Discriminative classifiers for deterministic dependency parsing. In *Proceedings of the COLING/ACL 2006 Main Conference Poster Sessions*, pages 316–323, Sydney.

Hudson, Richard A. 1990. *English Word Grammar*. Blackwell, Oxford.
Johansson, Richard and Pierre Nugues. 2006. Investigating multilingual dependency parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, pages 206–210, New York.
Kalt, Tom. 2004. Induction of greedy controllers for deterministic treebank parsers. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 17–24, Barcelona.
Kudo, Taku and Yuji Matsumoto. 2002. Japanese dependency analysis using cascaded chunking. In *Proceedings of the Sixth Workshop on Computational Language Learning (CoNLL)*, pages 63–69, Taipei.
Marcus, Mitchell P. 1980. *A Theory of Syntactic Recognition for Natural Language*. MIT Press, Cambridge, MA.
Marcus, Mitchell P., Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19:313–330.
Marcus, Mitchell P., Beatrice Santorini, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. 1994. The Penn Treebank: Annotating predicate-argument structure. In *Proceedings of the ARPA Human Language Technology Workshop*, pages 114–119, Plainsboro, NJ.
Marinov, S. 2007. Covington variations. In *Proceedings of the CoNLL Shared Task of EMNLP-CoNLL 2007*, pages 1144–1148, Prague.
McDonald, Ryan, Koby Crammer, and Fernando Pereira. 2005. Online large-margin training of dependency parsers. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 91–98, Ann Arbor, MI.
McDonald, Ryan, Kevin Lerman, and Fernando Pereira. 2006. Multilingual dependency analysis with a two-stage discriminative parser. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL)*, pages 216–220.
McDonald, Ryan and Joakim Nivre. 2007. Characterizing the errors of data-driven dependency parsing models. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 122–131, Prague.
McDonald, Ryan and Fernando Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *Proceedings of the 11th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 81–88, Trento.
McDonald, Ryan, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of the Human Language Technology Conference and the Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP)*, pages 523–530, Vancouver.
Mel'čuk, Igor. 1988. *Dependency Syntax: Theory and Practice*. State University of New York Press, New York.
Nilsson, Jens, Joakim Nivre, and Johan Hall. 2007. Generalizing tree transformations for inductive dependency parsing. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 968–975, Prague.
Nivre, Joakim. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*, pages 149–160, Nancy.
Nivre, Joakim. 2004. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together (ACL)*, pages 50–57, Barcelona.
Nivre, Joakim. 2006a. Constraints on non-projective dependency graphs. In *Proceedings of the 11th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 73–80, Trento.
Nivre, Joakim. 2006b. *Inductive Dependency Parsing*. Springer, Dordrecht.
Nivre, Joakim. 2007. Incremental non-projective dependency parsing. In *Proceedings of Human Language Technologies: The Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL HLT)*, pages 396–403, Rochester, NY.
Nivre, Joakim, Johan Hall, and Jens Nilsson. 2004. Memory-based dependency parsing. In *Proceedings of the 8th Conference on Computational Natural Language Learning (CoNLL)*, pages 49–56, Boston, MA.
Nivre, Joakim, Johan Hall, Jens Nilsson, Atanas Chanev, Gülşen Eryiğit, Sandra Kübler, Svetoslav Marinov, and

Erwin Marsi. 2007. MaltParser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13:95–135.

Nivre, Joakim, Johan Hall, Jens Nilsson, Gülsen Eryiğit, and Svetoslav Marinov. 2006. Labeled pseudo-projective dependency parsing with support vector machines. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL)*, pages 221–225, New York, NY.

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

Nivre, Joakim and Mario Scholz. 2004. Deterministic dependency parsing of English text. In *Proceedings of the 20th International Conference on Computational Linguistics (COLING)*, pages 64–70, Geneva.

Ratnaparkhi, Adwait. 1997. A linear observed time statistical parser based on maximum entropy models. In *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1–10, Providence, RI.

Ratnaparkhi, Adwait. 1999. Learning to parse natural language with maximum entropy models. *Machine Learning*, 34:151–175.

Sagae, Kenji and Alon Lavie. 2005. A classifier-based parser with linear run-time complexity. In *Proceedings of the 9th International Workshop on Parsing Technologies (IWPT)*, pages 125–132, Vancouver.

Sagae, Kenji and Alon Lavie. 2006. A best-first probabilistic shift-reduce parser. In *Proceedings of the COLING/ACL 2006 Main Conference Poster Sessions*, pages 691–698, Sydney.

Sgall, Petr, Eva Hajičová, and Jarmila Panevová. 1986. *The Meaning of the Sentence in Its Pragmatic Aspects*. Reidel, Dordrecht.

Shieber, Stuart M. 1983. Sentence disambiguation by a shift-reduce parsing technique. In *Proceedings of the 21st Conference on Association for Computational Linguistics (ACL)*, pages 113–118, Cambridge, MA.

Shieber, Stuart M., Yves Schabes, and Fernando C. N. Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24:3–36.

Tesnière, Lucien. 1959. *Éléments* de syntaxe structurale. Editions Klincksieck, Paris.

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