Non-Projective Dependency Parsing in Expected Linear Time

Joakim Nivre
Uppsala University, Department of Linguistics and Philology, SE-75126 Uppsala
Växjö University, School of Mathematics and Systems Engineering, SE-35195 Växjö
E-mail: joakim.nivre@lingfil.uu.se

Abstract

We present a novel transition system for dependency parsing, which constructs arcs only between adjacent words but can parse arbitrary non-projective trees by swapping the order of words in the input. Adding the swapping operation changes the time complexity for deterministic parsing from linear to quadratic in the worst case, but empirical estimates based on treebank data show that the expected running time is in fact linear for the range of data attested in the corpora. Evaluation on data from five languages shows state-of-the-art accuracy, with especially good results for the labeled exact match score.

1 Introduction

Syntactic parsing using dependency structures has become a standard technique in natural language processing with many different parsing models, in particular data-driven models that can be trained on syntactically annotated corpora (Yamada and Matsumoto, 2003; Nivre et al., 2004; McDonald et al., 2005a; Attardi, 2006; Titov and Henderson, 2007). A hallmark of many of these models is that they can be implemented very efficiently. Thus, transition-based parsers normally run in linear or quadratic time, using greedy deterministic search or fixed-width beam search (Nivre et al., 2004; Attardi, 2006; Johansson and Nugues, 2007; Titov and Henderson, 2007), and graph-based models support exact inference in at most cubic time, which is efficient enough to make global discriminative training practically feasible (McDonald et al., 2005a; McDonald et al., 2005b).

However, one problem that still has not found a satisfactory solution in data-driven dependency parsing is the treatment of discontinuous syntactic constructions, usually modeled by non-projective dependency trees, as illustrated in Figure 1. In a projective dependency tree, the yield of every subtree is a contiguous substring of the sentence. This is not the case for the tree in Figure 1, where the subtrees rooted at node 2 (*hearing*) and node 4 (*scheduled*) both have discontinuous yields.

Allowing non-projective trees generally makes parsing *computationally* harder. Exact inference for parsing models that allow non-projective trees is NP hard, except under very restricted independence assumptions (Neuhaus and Bröker, 1997; McDonald and Pereira, 2006; McDonald and Satta, 2007). There is recent work on algorithms that can cope with important subsets of all non-projective trees in polynomial time (Kuhlmann and Satta, 2009; Gómez-Rodríguez et al., 2009), but the time complexity is at best $O(n^6)$, which can be problematic in practical applications. Even the best algorithms for deterministic parsing run in quadratic time, rather than linear (Nivre, 2008a), unless restricted to a subset of non-projective structures as in Attardi (2006) and Nivre (2007).

But allowing non-projective dependency trees also makes parsing *empirically* harder, because it requires that we model relations between non-adjacent structures over potentially unbounded distances, which often has a negative impact on parsing accuracy. On the other hand, it is hardly possible to ignore non-projective structures completely, given that 25% or more of the sentences in some languages cannot be given a linguistically adequate analysis without invoking non-projective structures (Nivre, 2006; Kuhlmann and Nivre, 2006; Havelka, 2007).

Current approaches to data-driven dependency parsing typically use one of two strategies to deal with non-projective trees (unless they ignore them completely). Either they employ a non-standard parsing algorithm that can combine non-adjacent substructures (McDonald et al., 2005b; Attardi, 2006; Nivre, 2007), or they try to recover non-

Proceedings of the 47th Annual Meeting of the ACL and the 4th IJCNLP of the AFNLP, pages 351–359,
Suntec, Singapore, 2-7 August 2009.

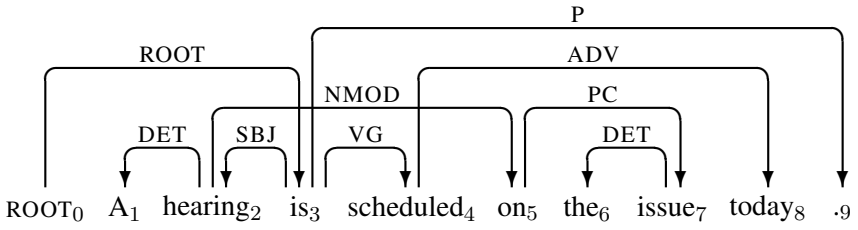

Figure 1: Dependency tree for an English sentence (non-projective).

projective dependencies by post-processing the output of a strictly projective parser (Nivre and Nilsson, 2005; Hall and Novák, 2005; McDonald and Pereira, 2006). In this paper, we will adopt a different strategy, suggested in recent work by Nivre (2008b) and Titov et al. (2009), and propose an algorithm that only combines adjacent substructures but derives non-projective trees by reordering the input words.

The rest of the paper is structured as follows. In Section 2, we define the formal representations needed and introduce the framework of transition-based dependency parsing. In Section 3, we first define a minimal transition system and explain how it can be used to perform projective dependency parsing in linear time; we then extend the system with a single transition for swapping the order of words in the input and demonstrate that the extended system can be used to parse unrestricted dependency trees with a time complexity that is quadratic in the worst case but still linear in the best case. In Section 4, we present experiments indicating that the expected running time of the new system on naturally occurring data is in fact linear and that the system achieves state-of-the-art parsing accuracy. We discuss related work in Section 5 and conclude in Section 6.

2 Background Notions

2.1 Dependency Graphs and Trees

Given a set L of dependency labels, a *dependency graph* for a sentence $x = w_1, \ldots, w_n$ is a directed graph $G = (V_x, A)$, where

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

The set V_x of *nodes* is the set of positive integers up to and including n, each corresponding to the linear position of a word in the sentence, plus an extra artificial root node 0. The set A of *arcs* is a set of triples (i, l, j), where i and j are nodes and l is a label. For a dependency graph $G = (V_x, A)$ to be well-formed, we in addition require that it is a *tree* rooted at the node 0, as illustrated in Figure 1.

2.2 Transition Systems

Following Nivre (2008a), we define a *transition system* for dependency parsing as a quadruple $S = (C, T, c_s, C_t)$, where

1. C is a set of *configurations*,
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 $x = w_1, \ldots, w_n$ to a configuration $c \in C$,
4. $C_t \subseteq C$ is a set of *terminal configurations*.

In this paper, we take the set C of configurations to be the set of all triples $c = (\Sigma, B, A)$ such that Σ and B are disjoint sublists of the nodes V_x of some sentence x, and A is a set of dependency arcs over V_x (and some label set L); we take the initial configuration for a sentence $x = w_1, \ldots, w_n$ to be $c_s(x) = ([0], [1, \ldots, n], \{\,\})$; and we take the set C_t of terminal configurations to be the set of all configurations of the form $c = ([0], [\,], A)$ (for any arc set A). The set T of transitions will be discussed in detail in Sections 3.1–3.2.

We will refer to the list Σ as the *stack* and the list B as the *buffer*, and we will use the variables σ and β for arbitrary sublists of Σ and B, respectively. For reasons of perspicuity, we will write Σ with its head (top) to the right and B with its head to the left. Thus, $c = ([\sigma|i], [j|\beta], A)$ is a configuration with the node i on top of the stack Σ and the node j as the first node in the buffer B.

Given a transition system $S = (C, T, c_s, C_t)$, a *transition sequence* for a sentence x 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 $t \in T$.

Transition		Condition
LEFT-ARC$_l$	$([\sigma\|i,j],B,A) \Rightarrow ([\sigma\|j],B,A\cup\{(j,l,i)\})$	$i \neq 0$
RIGHT-ARC$_l$	$([\sigma\|i,j],B,A) \Rightarrow ([\sigma\|i],B,A\cup\{(i,l,j)\})$	
SHIFT	$(\sigma,[i\|\beta],A) \Rightarrow ([\sigma\|i],\beta,A)$	
SWAP	$([\sigma\|i,j],\beta,A) \Rightarrow ([\sigma\|j],[i\|\beta],A)$	$0 < i < j$

Figure 2: Transitions for dependency parsing; $T_p = \{\text{LEFT-ARC}_l, \text{RIGHT-ARC}_l, \text{SHIFT}\}$; $T_u = T_p \cup \{\text{SWAP}\}$.

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

A transition system S is *sound* for a class $\mathbb{G}$ of dependency graphs iff, for every sentence x and transition sequence $C_{0,m}$ for x in S, $G_{c_m} \in \mathbb{G}$. S is *complete* for $\mathbb{G}$ iff, for every sentence x and dependency graph G for x in $\mathbb{G}$, there is a transition sequence $C_{0,m}$ for x in S such that $G_{c_m} = G$.

2.3 Deterministic Transition-Based Parsing

An *oracle* for a transition system S is a function $o : C \rightarrow T$. Ideally, o should always return the optimal transition t for a given configuration c, but all we require formally is that it respects the preconditions of transitions in T. That is, if $o(c) = t$ then t is permissible in c. Given an oracle o, deterministic transition-based parsing can be achieved by the following simple algorithm:

PARSE(o, x)
1 $c \leftarrow c_s(x)$
2 **while** $c \notin C_t$
3 **do** $t \leftarrow o(c)$; $c \leftarrow t(c)$
4 **return** G_c

Starting in the initial configuration $c_s(x)$, the parser repeatedly calls the oracle function o for the current configuration c and updates c according to the oracle transition t. The iteration stops when a terminal configuration is reached. It is easy to see that, provided that there is at least one transition sequence in S for every sentence, the parser constructs exactly one transition sequence $C_{0,m}$ for a sentence x and returns the parse defined by the terminal configuration c_m, i.e., $G_{c_m} = (V_x, A_{c_m})$. Assuming that the calls $o(c)$ and $t(c)$ can both be performed in constant time, 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.

When building practical parsing systems, the oracle can be approximated by a classifier trained on treebank data, a technique that has been used successfully in a number of systems (Yamada and Matsumoto, 2003; Nivre et al., 2004; Attardi, 2006). This is also the approach we will take in the experimental evaluation in Section 4.

3 Transitions for Dependency Parsing

Having defined the set of configurations, including initial and terminal configurations, we will now focus on the transition set T required for dependency parsing. The total set of transitions that will be considered is given in Figure 2, but we will start in Section 3.1 with the subset T_p (p for projective) consisting of the first three. In Section 3.2, we will add the fourth transition (SWAP) to get the full transition set T_u (u for unrestricted).

3.1 Projective Dependency Parsing

The minimal transition set T_p for projective dependency parsing contains three transitions:

1. LEFT-ARC$_l$ updates a configuration with i, j on top of the stack by adding (j, l, i) to A and replacing i, j on the stack by j alone. It is permissible as long as i is distinct from 0.
2. RIGHT-ARC$_l$ updates a configuration with i, j on top of the stack by adding (i, l, j) to A and replacing i, j on the stack by i alone.
3. SHIFT updates a configuration with i as the first node of the buffer by removing i from the buffer and pushing it onto the stack.

The system $S_p = (C, T_p, c_s, C_t)$ is sound and complete for the set of projective dependency trees (over some label set L) and has been used, in slightly different variants, by a number of transition-based dependency parsers (Yamada and Matsumoto, 2003; Nivre, 2004; Attardi, 2006;

Transition	Stack (Σ)	Buffer (B)	Added Arc
	$[\text{ROOT}_0]$	$[\text{A}_1, \ldots, .9]$	
SHIFT	$[\text{ROOT}_0, \text{A}_1]$	$[\text{hearing}_2, \ldots, .9]$	
SHIFT	$[\text{ROOT}_0, \text{A}_1, \text{hearing}_2]$	$[\text{is}_3, \ldots, .9]$	
LA_{DET}	$[\text{ROOT}_0, \text{hearing}_2]$	$[\text{is}_3, \ldots, .9]$	$(2, \text{DET}, 1)$
SHIFT	$[\text{ROOT}_0, \text{hearing}_2, \text{is}_3]$	$[\text{scheduled}_4, \ldots, .9]$	
SHIFT	$[\text{ROOT}_0, \ldots, \text{is}_3, \text{scheduled}_4]$	$[\text{on}_5, \ldots, .9]$	
SHIFT	$[\text{ROOT}_0, \ldots, \text{scheduled}_4, \text{on}_5]$	$[\text{the}_6, \ldots, .9]$	
SWAP	$[\text{ROOT}_0, \ldots, \text{is}_3, \text{on}_5]$	$[\text{scheduled}_4, \ldots, .9]$	
SWAP	$[\text{ROOT}_0, \text{hearing}_2, \text{on}_5]$	$[\text{is}_3, \ldots, .9]$	
SHIFT	$[\text{ROOT}_0, \ldots, \text{on}_5, \text{is}_3]$	$[\text{scheduled}_4, \ldots, .9]$	
SHIFT	$[\text{ROOT}_0, \ldots, \text{is}_3, \text{scheduled}_4]$	$[\text{the}_6, \ldots, .9]$	
SHIFT	$[\text{ROOT}_0, \ldots, \text{scheduled}_4, \text{the}_6]$	$[\text{issue}_7, \ldots, .9]$	
SWAP	$[\text{ROOT}_0, \ldots, \text{is}_3, \text{the}_6]$	$[\text{scheduled}_4, \ldots, .9]$	
SWAP	$[\text{ROOT}_0, \ldots, \text{on}_5, \text{the}_6]$	$[\text{is}_3, \ldots, .9]$	
SHIFT	$[\text{ROOT}_0, \ldots, \text{the}_6, \text{is}_3]$	$[\text{scheduled}_4, \ldots, .9]$	
SHIFT	$[\text{ROOT}_0, \ldots, \text{is}_3, \text{scheduled}_4]$	$[\text{issue}_7, \ldots, .9]$	
SHIFT	$[\text{ROOT}_0, \ldots, \text{scheduled}_4, \text{issue}_7]$	$[\text{today}_8, .9]$	
SWAP	$[\text{ROOT}_0, \ldots, \text{is}_3, \text{issue}_7]$	$[\text{scheduled}_4, \ldots, .9]$	
SWAP	$[\text{ROOT}_0, \ldots, \text{the}_6, \text{issue}_7]$	$[\text{is}_3, \ldots, .9]$	
LA_{DET}	$[\text{ROOT}_0, \ldots, \text{on}_5, \text{issue}_7]$	$[\text{is}_3, \ldots, .9]$	$(7, \text{DET}, 6)$
RA_{PC}	$[\text{ROOT}_0, \text{hearing}_2, \text{on}_5]$	$[\text{is}_3, \ldots, .9]$	$(5, \text{PC}, 7)$
RA_{NMOD}	$[\text{ROOT}_0, \text{hearing}_2]$	$[\text{is}_3, \ldots, .9]$	$(2, \text{NMOD}, 5)$
SHIFT	$[\text{ROOT}_0, \ldots, \text{hearing}_2, \text{is}_3]$	$[\text{scheduled}_4, \ldots, .9]$	
LA_{SBJ}	$[\text{ROOT}_0, \text{is}_3]$	$[\text{scheduled}_4, \ldots, .9]$	$(3, \text{SBJ}, 2)$
SHIFT	$[\text{ROOT}_0, \text{is}_3, \text{scheduled}_4]$	$[\text{today}_8, .9]$	
SHIFT	$[\text{ROOT}_0, \ldots, \text{scheduled}_4, \text{today}_8]$	$[.9]$	
RA_{ADV}	$[\text{ROOT}_0, \text{is}_3, \text{scheduled}_4]$	$[.9]$	$(4, \text{ADV}, 8)$
RA_{VG}	$[\text{ROOT}_0, \text{is}_3]$	$[.9]$	$(3, \text{VG}, 4)$
SHIFT	$[\text{ROOT}_0, \text{is}_3, .9]$	$[\,]$	
RA_{P}	$[\text{ROOT}_0, \text{is}_3]$	$[\,]$	$(3, \text{P}, 9)$
RA_{ROOT}	$[\text{ROOT}_0]$	$[\,]$	$(0, \text{ROOT}, 3)$

Figure 3: Transition sequence for parsing the sentence in Figure 1 (LA = LEFT-ARC, RA = REFT-ARC).

Nivre, 2008a). For proofs of soundness and completeness, see Nivre (2008a).

As noted in section 2, the worst-case time complexity of a deterministic transition-based parser is given by an upper bound on the length of transition sequences. In S_p, the number of transitions for a sentence $x = w_1, \ldots, w_n$ is always exactly $2n$, since a terminal configuration can only be reached after n SHIFT transitions (moving nodes $1, \ldots, n$ from B to Σ) and n applications of LEFT-ARC$_l$ or RIGHT-ARC$_l$ (removing the same nodes from Σ). Hence, the complexity of deterministic parsing is $O(n)$ in the worst case (as well as in the best case).

3.2 Unrestricted Dependency Parsing

We now consider what happens when we add the fourth transition from Figure 2 to get the extended transition set T_u. The SWAP transition updates a configuration with stack $[\sigma|i, j]$ by moving the node i back to the buffer. This has the effect that the order of the nodes i and j in the appended list $\Sigma + B$ is reversed compared to the original word order in the sentence. It is important to note that SWAP is only permissible when the two nodes on top of the stack are in the original word order, which prevents the same two nodes from being swapped more than once, and when the leftmost node i is distinct from the root node 0. Note also that SWAP moves the node i back to the buffer, so that LEFT-ARC$_l$, RIGHT-ARC$_l$ or SWAP can subsequently apply with the node j on top of the stack.

The fact that we can swap the order of nodes, implicitly representing subtrees, means that we can construct non-projective trees by applying

$$o(c) = \begin{cases} \text{LEFT-ARC}_l & \text{if } c = ([\sigma|i,j], B, A_c),\ (j,l,i) \in A \text{ and } A^i \subseteq A_c \\ \text{RIGHT-ARC}_l & \text{if } c = ([\sigma|i,j], B, A_c),\ (i,l,j) \in A \text{ and } A^j \subseteq A_c \\ \text{SWAP} & \text{if } c = ([\sigma|i,j], B, A_c) \text{ and } j <_G i \\ \text{SHIFT} & \text{otherwise} \end{cases}$$

Figure 4: Oracle function for $S_u = (C, T_u, c_s, C_t)$ with target tree $G = (V_x, A)$. We use the notation A^i to denote the subset of A that only contains the outgoing arcs of the node i.

LEFT-ARC$_l$ or RIGHT-ARC$_l$ to subtrees whose yields are not adjacent according to the original word order. This is illustrated in Figure 3, which shows the transition sequence needed to parse the example in Figure 1. For readability, we represent both the stack Σ and the buffer B as lists of tokens, indexed by position, rather than abstract nodes. The last column records the arc that is added to the arc set A in a given transition (if any).

Given the simplicity of the extension, it is rather remarkable that the system $S_u = (C, T_u, c_s, C_t)$ is sound and complete for the set of all dependency trees (over some label set L), including all non-projective trees. The soundness part is trivial, since any terminating transition sequence will have to move all the nodes $1, \ldots, n$ from B to Σ (using SHIFT) and then remove them from Σ (using LEFT-ARC$_l$ or RIGHT-ARC$_l$), which will produce a tree with root 0.

For completeness, we note first that projectivity is not a property of a dependency tree in itself, but of the tree in combination with a word order, and that a tree can always be made projective by reordering the nodes. For instance, let x be a sentence with dependency tree $G = (V_x, A)$, and let $<_G$ be the total order on V_x defined by an inorder traversal of G that respects the local ordering of a node and its children given by the original word order. Regardless of whether G is projective with respect to x, it must by necessity be projective with respect to $<_G$. We call $<_G$ the *projective order* corresponding to x and G and use it as our canonical way of finding a node order that makes the tree projective. By way of illustration, the projective order for the sentence and tree in Figure 1 is: $\text{A}_1 <_G \text{hearing}_2 <_G \text{on}_5 <_G \text{the}_6 <_G \text{issue}_7 <_G \text{is}_3 <_G \text{scheduled}_4 <_G \text{today}_8 <_G .9$.

If the words of a sentence x with dependency tree G are already in projective order, this means that G is projective with respect to x and that we can parse the sentence using only transitions in T_p, because nodes can be pushed onto the stack in projective order using only the SHIFT transition. If the words are *not* in projective order, we can use a combination of SHIFT and SWAP transitions to ensure that nodes are still pushed onto the stack in projective order. More precisely, if the next node in the projective order is the kth node in the buffer, we perform k SHIFT transitions, to get this node onto the stack, followed by $k-1$ SWAP transitions, to move the preceding $k-1$ nodes back to the buffer.[1] In this way, the parser can effectively sort the input nodes into projective order on the stack, repeatedly extracting the minimal element of $<_G$ from the buffer, and build a tree that is projective with respect to the sorted order. Since any input can be sorted using SHIFT and SWAP, and any projective tree can be built using SHIFT, LEFT-ARC$_l$ and RIGHT-ARC$_l$, the system S_u is complete for the set of all dependency trees.

In Figure 4, we define an oracle function o for the system S_u, which implements this "sort and parse" strategy and predicts the optimal transition t out of the current configuration c, given the target dependency tree $G = (V_x, A)$ and the projective order $<_G$. The oracle predicts LEFT-ARC$_l$ or RIGHT-ARC$_l$ if the two top nodes on the stack should be connected by an arc and if the dependent node of this arc is already connected to all its dependents; it predicts SWAP if the two top nodes are not in projective order; and it predicts SHIFT otherwise. This is the oracle that has been used to generate training data for classifiers in the experimental evaluation in Section 4.

Let us now consider the time complexity of the extended system $S_u = (C, T_u, c_s, C_t)$ and let us begin by observing that $2n$ is still a *lower* bound on the number of transitions required to reach a terminal configuration. A sequence of $2n$ transi-

[1] This can be seen in Figure 3, where transitions 4–8, 9–13, and 14–18 are the transitions needed to make sure that *on*$_5$, *the*$_6$ and *issue*$_7$ are processed on the stack before *is*$_3$ and *scheduled*$_4$.

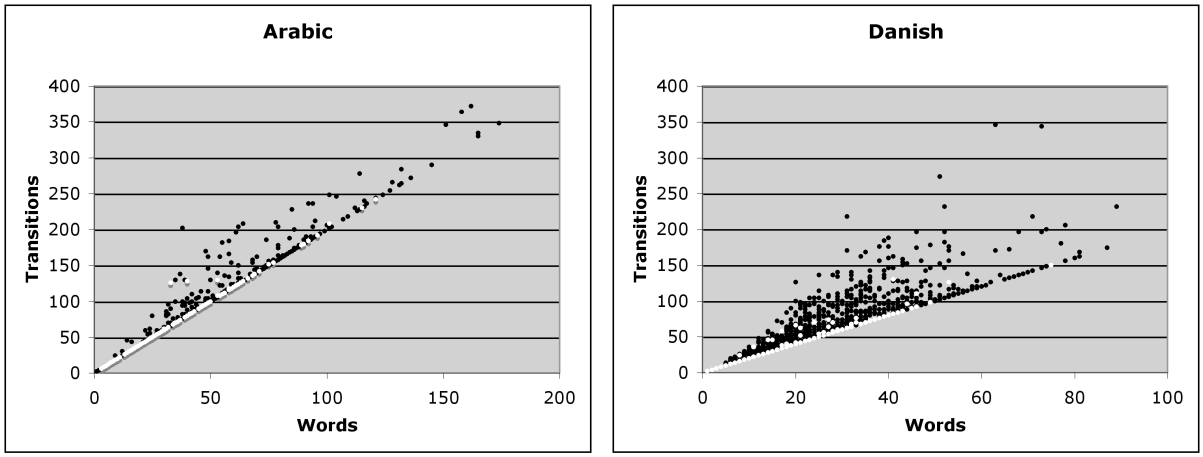

Figure 5: Abstract running time during training (black) and parsing (white) for Arabic (1460/146 sentences) and Danish (5190/322 sentences).

tions occurs when no SWAP transitions are performed, in which case the behavior of the system is identical to the simpler system S_p. This is important, because it means that the best-case complexity of the deterministic parser is still $O(n)$ and that the we can expect to observe the best case for all sentences with projective dependency trees.

The exact number of additional transitions needed to reach a terminal configuration is determined by the number of SWAP transitions. Since SWAP moves one node from Σ to B, there will be one additional SHIFT for every SWAP, which means that the total number of transitions is $2n + 2k$, where k is the number of SWAP transitions. Given the condition that SWAP can only apply in a configuration $c = ([\sigma|i, j], B, A)$ if $0 < i < j$, the number of SWAP transitions is bounded by $\frac{n(n-1)}{2}$, which means that $2n + n(n-1) = n + n^2$ is an upper bound on the number of transitions in a terminating sequence. Hence, the worst-case complexity of the deterministic parser is $O(n^2)$.

The running time of a deterministic transition-based parser using the system S_u is $O(n)$ in the best case and $O(n^2)$ in the worst case. But what about the average case? Empirical studies, based on data from a wide range of languages, have shown that dependency trees tend to be projective and that most non-projective trees only contain a small number of discontinuities (Nivre, 2006; Kuhlmann and Nivre, 2006; Havelka, 2007). This should mean that the expected number of swaps per sentence is small, and that the running time is linear on average for the range of inputs that occur in natural languages. This is a hypothesis that will be tested experimentally in the next section.

4 Experiments

Our experiments are based on five data sets from the CoNLL-X shared task: Arabic, Czech, Danish, Slovene, and Turkish (Buchholz and Marsi, 2006). These languages have been selected because the data come from genuine dependency treebanks, whereas all the other data sets are based on some kind of conversion from another type of representation, which could potentially distort the distribution of different types of structures in the data.

4.1 Running Time

In section 3.2, we hypothesized that the expected running time of a deterministic parser using the transition system S_u would be linear, rather than quadratic. To test this hypothesis, we examine how the number of transitions varies as a function of sentence length. We call this the *abstract running time*, since it abstracts over the actual time needed to compute each oracle prediction and transition, which is normally constant but dependent on the type of classifier used.

We first measured the abstract running time on the training sets, using the oracle to derive the transition sequence for every sentence, to see how many transitions are required in the ideal case. We then performed the same measurement on the test sets, using classifiers trained on the oracle transition sequences from the training sets (as described below in Section 4.2), to see whether the trained parsers deviate from the ideal case.

The result for Arabic and Danish can be seen

System	Arabic		Czech		Danish		Slovene		Turkish	
	AS	EM	AS	EM	AS	EM	AS	EM	AS	EM
S_u	67.1 (9.1)	**11.6**	**82.4** (**73.8**)	**35.3**	84.2 (22.5)	26.7	75.2 (23.0)	**29.9**	64.9 (**11.8**)	**21.5**
S_p	67.3 (**18.2**)	**11.6**	80.9 (3.7)	31.2	84.6 (0.0)	27.0	74.2 (3.4)	**29.9**	65.3 (6.6)	21.0
S_{pp}	67.2 (**18.2**)	**11.6**	82.1 (60.7)	34.0	84.7 (22.5)	28.9	74.8 (20.7)	26.9	65.5 (**11.8**)	20.7
Malt-06	66.7 (**18.2**)	11.0	78.4 (57.9)	27.4	84.8 (27.5)	26.7	70.3 (20.7)	19.7	65.7 (9.2)	19.3
MST-06	66.9 (0.0)	10.3	80.2 (61.7)	29.9	84.8 (**62.5**)	25.5	73.4 (26.4)	20.9	63.2 (**11.8**)	20.2
MST_{Malt}	**68.6** (9.4)	11.0	82.3 (69.2)	31.2	**86.7** (60.0)	**29.8**	**75.9** (**27.6**)	26.6	**66.3** (9.2)	18.6

Table 1: Labeled accuracy; AS = attachment score (non-projective arcs in brackets); EM = exact match.

in Figure 5, where black dots represent training sentences (parsed with the oracle) and white dots represent test sentences (parsed with a classifier). For Arabic there is a very clear linear relationship in both cases with very few outliers. Fitting the data with a linear function using the least squares method gives us $m = 2.06n$ ($R^2 = 0.97$) for the training data and $m = 2.02n$ ($R^2 = 0.98$) for the test data, where m is the number of transitions in parsing a sentence of length n. For Danish, there is clearly more variation, especially for the training data, but the least-squares approximation still explains most of the variance, with $m = 2.22n$ ($R^2 = 0.85$) for the training data and $m = 2.07n$ ($R^2 = 0.96$) for the test data. For both languages, we thus see that the classifier-based parsers have a lower mean number of transitions and less variance than the oracle parsers. And in both cases, the expected number of transitions is only marginally greater than the $2n$ of the strictly projective transition system S_p.

We have chosen to display results for Arabic and Danish because they are the two extremes in our sample. Arabic has the smallest variance and the smallest linear coefficients, and Danish has the largest variance and the largest coefficients. The remaining three languages all lie somewhere in the middle, with Czech being closer to Arabic and Slovene closer to Danish. Together, the evidence from all five languages strongly corroborates the hypothesis that the expected running time for the system S_u is linear in sentence length for naturally occurring data.

4.2 Parsing Accuracy

In order to assess the parsing accuracy that can be achieved with the new transition system, we trained a deterministic parser using the new transition system S_u for each of the five languages. For comparison, we also trained two parsers using S_p, one that is strictly projective and one that uses the pseudo-projective parsing technique to recover non-projective dependencies in a post-processing step (Nivre and Nilsson, 2005). We will refer to the latter system as S_{pp}. All systems use SVM classifiers with a polynomial kernel to approximate the oracle function, with features and parameters taken from Nivre et al. (2006), which was the best performing transition-based system in the CoNLL-X shared task.[2]

Table 1 shows the labeled parsing accuracy of the parsers measured in two ways: *attachment score* (AS) is the percentage of *tokens* with the correct head and dependency label; *exact match* (EM) is the percentage of *sentences* with a completely correct labeled dependency tree. The score in brackets is the attachment score for the (small) subset of tokens that are connected to their head by a non-projective arc in the gold standard parse. For comparison, the table also includes results for the two best performing systems in the original CoNLL-X shared task, Malt-06 (Nivre et al., 2006) and MST-06 (McDonald et al., 2006), as well as the integrated system MST_{Malt}, which is a graph-based parser guided by the predictions of a transition-based parser and currently has the best reported results on the CoNLL-X data sets (Nivre and McDonald, 2008).

Looking first at the overall attachment score, we see that S_u gives a substantial improvement over S_p (and outperforms S_{pp}) for Czech and Slovene, where the scores achieved are rivaled only by the combo system MST_{Malt}. For these languages, there is no statistical difference between S_u and MST_{Malt}, which are both significantly better than all the other parsers, except S_{pp} for Czech (McNemar's test, $\alpha = .05$). This is accompanied by an improvement on non-projective arcs, where

[2]Complete information about experimental settings can be found at http://stp.lingfil.uu.se/∼nivre/exp/.

S_u outperforms all other systems for Czech and is second only to the two MST parsers (MST-06 and MST_{Malt}) for Slovene. It is worth noting that the percentage of non-projective arcs is higher for Czech (1.9%) and Slovene (1.9%) than for any of the other languages.

For the other three languages, S_u has a drop in overall attachment score compared to S_p, but none of these differences is statistically significant. In fact, the only significant differences in attachment score here are the positive differences between MST_{Malt} and all other systems for Arabic and Danish, and the negative difference between MST-06 and all other systems for Turkish. The attachment scores for non-projective arcs are generally very low for these languages, except for the two MST parsers on Danish, but S_u performs at least as well as S_{pp} on Danish and Turkish. (The results for Arabic are not very meaningful, given that there are only eleven non-projective arcs in the entire test set, of which the (pseudo-)projective parsers found two and S_u one, while MST_{Malt} and MST-06 found none at all.)

Considering the exact match scores, finally, it is very interesting to see that S_u almost consistently outperforms all other parsers, including the combo system MST_{Malt}, and sometimes by a fairly wide margin (Czech, Slovene). The difference is statistically significant with respect to all other systems except MST_{Malt} for Slovene, all except MST_{Malt} and S_{pp} for Czech, and with respect to MST_{Malt} for Turkish. For Arabic and Danish, there are no significant differences in the exact match scores. We conclude that S_u may increase the probability of finding a completely correct analysis, which is sometimes reflected also in the overall attachment score, and we conjecture that the strength of the positive effect is dependent on the frequency of non-projective arcs in the language.

5 Related Work

Processing non-projective trees by swapping the order of words has recently been proposed by both Nivre (2008b) and Titov et al. (2009), but these systems cannot handle unrestricted non-projective trees. It is worth pointing out that, although the system described in Nivre (2008b) uses four transitions bearing the same names as the transitions of S_u, the two systems are not equivalent. In particular, the system of Nivre (2008b) is sound but not complete for the class of all dependency trees.

There are also affinities to the system of Attardi (2006), which combines non-adjacent nodes on the stack instead of swapping nodes and is equivalent to a restricted version of our system, where no more than two consecutive SWAP transitions are permitted. This restriction preserves linear worst-case complexity at the expense of completeness. Finally, the algorithm first described by Covington (2001) and used for data-driven parsing by Nivre (2007), is complete but has quadratic complexity even in the best case.

6 Conclusion

We have presented a novel transition system for dependency parsing that can handle unrestricted non-projective trees. The system reuses standard techniques for building projective trees by combining adjacent nodes (representing subtrees with adjacent yields), but adds a simple mechanism for swapping the order of nodes on the stack, which gives a system that is sound and complete for the set of all dependency trees over a given label set but behaves exactly like the standard system for the subset of projective trees. As a result, the time complexity of deterministic parsing is $O(n^2)$ in the worst case, which is rare, but $O(n)$ in the best case, which is common, and experimental results on data from five languages support the conclusion that expected running time is linear in the length of the sentence. Experimental results also show that parsing accuracy is competitive, especially for languages like Czech and Slovene where non-projective dependency structures are common, and especially with respect to the exact match score, where it has the best reported results for four out of five languages. Finally, the simplicity of the system makes it very easy to implement.

Future research will include an in-depth error analysis to find out why the system works better for some languages than others and why the exact match score improves even when the attachment score goes down. In addition, we want to explore alternative oracle functions, which try to minimize the number of swaps by allowing the stack to be temporarily "unsorted".

Acknowledgments

Thanks to Johan Hall and Jens Nilsson for help with implementation and evaluation, and to Marco Kuhlmann and three anonymous reviewers for useful comments.

References

Giuseppe Attardi. 2006. Experiments with a multilanguage non-projective dependency parser. In *Proceedings of CoNLL*, pages 166–170.

Sabine Buchholz and Erwin Marsi. 2006. CoNLL-X shared task on multilingual dependency parsing. In *Proceedings of CoNLL*, pages 149–164.

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

Carlos Gómez-Rodríguez, David Weir, and John Carroll. 2009. Parsing mildly non-projective dependency structures. In *Proceedings of EACL*, pages 291–299.

Keith Hall and Vaclav Novák. 2005. Corrective modeling for non-projective dependency parsing. In *Proceedings of IWPT*, pages 42–52.

Jiri Havelka. 2007. Beyond projectivity: Multilingual evaluation of constraints and measures on non-projective structures. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 608–615.

Richard Johansson and Pierre Nugues. 2007. Incremental dependency parsing using online learning. In *Proceedings of the Shared Task of EMNLP-CoNLL*, pages 1134–1138.

Marco Kuhlmann and Joakim Nivre. 2006. Mildly non-projective dependency structures. In *Proceedings of the COLING/ACL Main Conference Poster Sessions*, pages 507–514.

Marco Kuhlmann and Giorgio Satta. 2009. Treebank grammar techniques for non-projective dependency parsing. In *Proceedings of EACL*, pages 478–486.

Ryan McDonald and Fernando Pereira. 2006. Online learning of approximate dependency parsing algorithms. In *Proceedings of EACL*, pages 81–88.

Ryan McDonald and Giorgio Satta. 2007. On the complexity of non-projective data-driven dependency parsing. In *Proceedings of IWPT*, pages 122–131.

Ryan McDonald, Koby Crammer, and Fernando Pereira. 2005a. Online large-margin training of dependency parsers. In *Proceedings of ACL*, pages 91–98.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005b. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of HLT/EMNLP*, pages 523–530.

Ryan McDonald, Kevin Lerman, and Fernando Pereira. 2006. Multilingual dependency analysis with a two-stage discriminative parser. In *Proceedings of CoNLL*, pages 216–220.

Peter Neuhaus and Norbert Bröker. 1997. The complexity of recognition of linguistically adequate dependency grammars. In *Proceedings of ACL/EACL*, pages 337–343.

Joakim Nivre and Ryan McDonald. 2008. Integrating graph-based and transition-based dependency parsers. In *Proceedings of ACL*, pages 950–958.

Joakim Nivre and Jens Nilsson. 2005. Pseudo-projective dependency parsing. In *Proceedings of ACL*, pages 99–106.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2004. Memory-based dependency parsing. In *Proceedings of CoNLL*, pages 49–56.

Joakim Nivre, Johan Hall, Jens Nilsson, Gülsen Eryiğit, and Svetoslav Marinov. 2006. Labeled pseudo-projective dependency parsing with support vector machines. In *Proceedings of CoNLL*, pages 221–225.

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together (ACL)*, pages 50–57.

Joakim Nivre. 2006. Constraints on non-projective dependency graphs. In *Proceedings of EACL*, pages 73–80.

Joakim Nivre. 2007. Incremental non-projective dependency parsing. In *Proceedings of NAACL HLT*, pages 396–403.

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

Joakim Nivre. 2008b. Sorting out dependency parsing. In *Proceedings of the 6th International Conference on Natural Language Processing (GoTAL)*, pages 16–27.

Ivan Titov and James Henderson. 2007. A latent variable model for generative dependency parsing. In *Proceedings of IWPT*, pages 144–155.

Ivan Titov, James Henderson, Paola Merlo, and Gabriele Musillo. 2009. Online graph planarization for synchronous parsing of semantic and syntactic dependencies. In *Proceedings of IJCAI*.

Hiroyasu Yamada and Yuji Matsumoto. 2003. Statistical dependency analysis with support vector machines. In *Proceedings of IWPT*, pages 195–206.