

# AN EFFICIENT ALGORITHM FOR PROJECTIVE DEPENDENCY PARSING

**Joakim Nivre**

Växjö University

School of Mathematics and Systems Engineering

SE-35195 VÄXJÖ

Joakim.Nivre@msi.vxu.se

## Abstract

This paper presents a deterministic parsing algorithm for projective dependency grammar. The running time of the algorithm is linear in the length of the input string, and the dependency graph produced is guaranteed to be projective and acyclic. The algorithm has been experimentally evaluated in parsing unrestricted Swedish text, achieving an accuracy above 85% with a very simple grammar.

## 1 Introduction

Despite a long and venerable tradition in linguistics, dependency grammar has until quite recently played a fairly marginal role in natural language parsing. However, dependency-based representations have turned out to be useful in statistical approaches to parsing and disambiguation (see, e.g., Collins [4, 5, 6], Eisner [15, 16, 17], Samuelsson [25]) and they also appear well suited for languages with less rigid word order constraints (Covington [9, 10], Collins et al. [7]).

Several different parsing techniques have been used with dependency grammar. The most common approach is probably to use some version of the dynamic programming algorithms familiar from context-free parsing, with or without statistical disambiguation (Eisner [15, 16, 17], Barbero et al. [2], Courtin and Genthial [8], Samuelsson [25]). Another school proposes that dependency parsing should be cast as a constraint satisfaction problem and solved using constraint programming (Maruyama [22], Menzel and Schröder [24], Duchier [12, 13, 14]).

Here I will instead pursue a deterministic approach to dependency parsing, similar to shift-reduce parsing for context-free grammar. In the past, deterministic parsing of natural language has mostly been motivated by psycholinguistic concerns, as in the well-known Parsifal system of Marcus [21]. However, deterministic parsing also has the more direct practical advantage of providing very efficient disambiguation. If the disambiguation can be performed with high accuracy and robustness, deterministic parsing becomes an interesting alternative to more traditional algorithms for natural language parsing. There are potential applications of natural language parsing, for example in information retrieval, where it may not always be necessary to construct a complete dependency structure for a sentence, as long as dependency relations can be identified with good enough accuracy, but where efficiency can be vital because of the large amount of data to be processed. It may also be possible to improve parse accuracy by adding customized post-processing in order to correct typical errors introduced by the parser. In this way, deterministic dependency parsing can be seen as a compromise between so-called deep

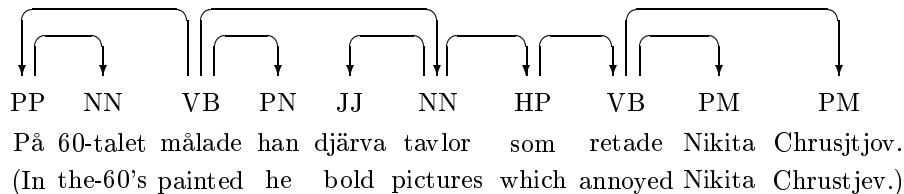


Figure 1: Dependency graph for Swedish sentence

and shallow processing. It is a kind of deep processing in that the goal is to build a complete syntactic analysis for the input string, not just identify basic constituents as in partial parsing. But it resembles shallow processing in being robust, efficient and deterministic.

In this paper, I present an algorithm that produces projective dependency graphs deterministically in linear time. Preliminary experiments indicate that parsing accuracy above 85% for unrestricted text is attainable with a very simple grammar. In section 2, I introduce the necessary concepts from dependency grammar and the specific form of grammar rules required by the parser. In section 3, I define the algorithm and prove that its worst case running time is linear in the size of the input; I also prove that the dependency graphs produced by the algorithm are projective and acyclic. In section 4, I present preliminary results concerning the parsing accuracy when applied to unrestricted Swedish text, using a simple hand-crafted grammar. In section 5, I conclude with some suggestions for further research.

## 2 Projective Dependency Grammar

The linguistic tradition of dependency grammar comprises a large and fairly diverse family of grammatical theories and formalisms that share certain basic assumptions about syntactic structure, in particular the assumption that syntactic structure consists of *lexical nodes* linked by binary relations called *dependencies* (see, e.g., Tesnière [30], Sgall et al. [26], Mel'čuk [23], Hudson [20]). Thus, the common formal property of dependency structures, as compared to the representations based on constituency (or phrase structure), is the lack of phrasal nodes.

In a dependency structure, every lexical node is dependent on at most one other lexical node, usually called its *head* or *regent*, which means that the structure can be represented as a directed graph, with nodes representing lexical elements and arcs representing dependency relations. Figure 1 shows a dependency graph for a simple Swedish sentence, where each word of the sentence is labeled with its part of speech.

Over and above these minimal assumptions, different varieties of dependency grammar impose different conditions on dependency graphs. (In fact, not even the constraint that each lexical node has at most one head is adopted universally; cf. Hudson [20].) The following three constraints are common in the literature and will all be adopted in the sequel:

1. Dependency graphs should be acyclic.
2. Dependency graphs should be connected.
3. Dependency graphs should be projective.

Adopting the first and second of these conditions is tantamount to the assumption that the dependency graph forms a rooted tree, with a single lexical node being the transitive head of

all others. For example, the dependency graph in Figure 1 is a tree with the finite verb *mâlade* (painted) as the root node.

While the conditions of acyclicity and connectedness are adopted in most versions of dependency grammar, the projectivity constraint is more controversial and can in fact be seen as a major branch point within this tradition. However, most researchers seem to agree that, even if the assumption of projectivity is questionable from a theoretical point of view, it is nevertheless a reasonable approximation in practical parsing systems. First of all, it must be noted that projectivity is not a property of the dependency graph in itself but only in relation to the linear ordering of tokens in the surface string. Several different definitions of projectivity can be found in the literature, but they are all roughly equivalent (see, e.g., Melčuk [23], Hudson [20], Sleator and Temperley [27, 28]). I will follow Hudson [20] and define projectivity in terms of an extended notion of adjacency:

1. A dependency graph is *projective* iff every dependent node is *graph adjacent* to its head.
2. Two nodes  $n$  and  $n'$  are *graph adjacent* iff every node  $n''$  occurring between  $n$  and  $n'$  in the surface string is dominated by  $n$  or  $n'$  in the graph.

Note that projectivity does not imply that the graph is connected; nor does it imply that the graph is acyclic (although it does exclude cycles of length greater than 2).

We are now in a position to define what we mean by a well-formed dependency graph:

- A string of words  $W$  is represented as a list of *tokens*, where each token  $n = (i, w)$  is a pair consisting of a position  $i$  and a word form  $w$ ; the functional expressions  $\text{POS}(n) = i$  and  $\text{LEX}(n) = w$  can be used to extract the position and word form of a token. We let  $<$  denote the complete and strict ordering of tokens in  $W$ , i.e.  $n < n'$  iff  $\text{POS}(n) < \text{POS}(n')$ .
- A dependency graph for  $W$  is a directed graph  $D = (N_W, A)$ , where the set of nodes  $N_W$  is the set of tokens in  $W$ , and the arc relation  $A$  is a binary, irreflexive relation on  $N_W$ . We write  $n \rightarrow n'$  to say that there is an arc from  $n$  to  $n'$ , i.e.  $(n, n') \in A$ ; we use  $\rightarrow^*$  to denote the reflexive and transitive closure of the arc relation  $A$ ; and we use  $\leftrightarrow$  and  $\leftrightarrow^*$  for the corresponding undirected relations, i.e.  $n \leftrightarrow n'$  iff  $n \rightarrow n'$  or  $n' \rightarrow n$ .
- A dependency graph  $D = (N_W, A)$  is well-formed iff the following conditions are satisfied:

**Single head**  $(\forall n n' n'') (n \rightarrow n' \wedge n'' \rightarrow n') \Rightarrow n = n''$

**Acyclic**  $(\forall n n') \neg(n \rightarrow n' \wedge n' \rightarrow^* n)$

**Connected**  $(\forall n n') n \leftrightarrow^* n'$

**Projective**  $(\forall n n' n'') (n \leftrightarrow n' \wedge n < n'' < n') \Rightarrow (n \rightarrow^* n'' \vee n' \rightarrow^* n'')$

Finally, let us note that all dependency graphs considered in this paper are unlabeled, in the sense that they have no labels on the edges representing different kinds of dependency relations (such as *subject*, *object*, *adverbial*, etc.).

Most formalizations of dependency grammar use rules that specify whole configurations of dependents for a given head, using some notion of *valence frames* (Hays [19], Gaifman [18], Carroll and Charniak [3], Sleator and Temperley [27, 28], Barbero et al. [2], Eisner [17], Debusmann [11]). Here I will instead use a much simpler formalism, where only binary relations

between heads and dependents can be specified. More precisely, a grammar  $G$  is a pair  $(T, R)$ , where  $T$  is a (terminal) vocabulary (set of word forms) and  $R$  is a set of rules of the following form (where  $w, w' \in T$ ):

$$\begin{array}{l} w \leftarrow w' \\ w \rightarrow w' \end{array}$$

Given a string of words with nodes  $n$  and  $n'$  such that  $\text{LEX}(n) = w$ ,  $\text{LEX}(n') = w'$  and  $n < n'$ , the first rule says that  $n'$  can be the head of  $n$ , while the second rule says that  $n$  can be the head of  $n'$ . Rules of the first kind are referred to as right-headed, rules of the second kind as left-headed. The grammar rules used here are very similar to Covington's [9] notion of  $D$ -rules, except that the latter are undirected, and I will therefore call them *directed  $D$ -rules*. Directed  $D$ -rules are also similar to – but simpler than – the “dependency relations” of Courtin and Genthal [8].

Finally, it is worth pointing out that, even though the parsing algorithm defined in the next section presupposes directed  $D$ -rules, this does not exclude its use with more traditional dependency grammars, since it is usually straightforward to extract directed  $D$ -rules from these grammars. For example, given a grammar in the formalism of Hays [19], we can construct a set of directed  $D$ -rules in the following way:

- For every rule  $X(X_{-m} \cdots X_{-1} * X_1 \cdots X_n)$  in the original grammar, introduce rules  $w_l \leftarrow w$  and  $w \rightarrow w_r$  for every  $w \in X$ ,  $w_l \in X_{-m} \cup \cdots \cup X_{-1}$ ,  $w_r \in X_1 \cup \cdots \cup X_n$ .

However, using the parsing algorithm defined in the next section with the directed  $D$ -rules extracted in this way will not result in a parser for the original grammar, since the extracted rules normally impose much weaker constraints on the dependency structure than the original rules.

### 3 Parsing Algorithm

The parsing algorithm presented in this section is in many ways similar to the basic shift-reduce algorithm for context-free grammars (Aho et al. [1]), although the parse actions are different given that we are using directed  $D$ -rules instead of context-free grammar rules. For example, since there are no nonterminal symbols in the grammar, the stack will never contain anything but input tokens (graph nodes), and a reduce action simply amounts to popping the topmost element from the stack.

Parser configurations are represented by triples  $\langle S, I, A \rangle$ , where  $S$  is the stack (represented as a list),  $I$  is the list of (remaining) input tokens, and  $A$  is the (current) arc relation for the dependency graph. (The set of nodes  $N_W$  in the dependency graph is given by the input string, as defined in the preceding section, and need not be represented explicitly in the configuration.) Given an input string  $W$ , the parser is initialized to  $\langle \text{nil}, W, \emptyset \rangle$  and terminates when it reaches a configuration  $\langle S, \text{nil}, A \rangle$  (for any list  $S$  and set of arcs  $A$ ). The input string  $W$  is *accepted* if the dependency graph  $D = (N_W, A)$  given at termination is well-formed; otherwise  $W$  is *rejected*. The behavior of the parser is defined by the transitions defined in Figure 2 (where  $n$  and  $n'$  are arbitrary graph nodes):

<b>Initialization</b>	$\langle \text{nil}, W, \emptyset \rangle$	
<b>Termination</b>	$\langle S, \text{nil}, A \rangle$	
<b>Left-Arc</b>	$\langle n S, n' I, A \rangle \rightarrow \langle S, n' I, A \cup \{(n', n)\} \rangle$	$\text{LEX}(n) \leftarrow \text{LEX}(n') \in R$ $\neg \exists n''(n'', n) \in A$
<b>Right-Arc</b>	$\langle n S, n' I, A \rangle \rightarrow \langle n' n S, I, A \cup \{(n, n')\} \rangle$	$\text{LEX}(n) \rightarrow \text{LEX}(n') \in R$ $\neg \exists n''(n'', n') \in A$
<b>Reduce</b>	$\langle n S, I, A \rangle \rightarrow \langle S, I, A \rangle$	$\exists n'(n', n) \in A$
<b>Shift</b>	$\langle S, n I, A \rangle \rightarrow \langle n S, I, A \rangle$	

Figure 2: Parser transitions

- The transition **Left-Arc** adds an arc  $n' \rightarrow n$  from the next input token  $n'$  to the node  $n$  on top of the stack and reduces (pops)  $n$  from the stack, provided that the grammar contains the rule  $\text{LEX}(n) \leftarrow \text{LEX}(n')$  and provided that the graph does not contain an arc  $n'' \rightarrow n$ . The reason that the dependent node is immediately reduced is to eliminate the possibility of adding an arc  $n \rightarrow n'$  (should the grammar contain the rule  $\text{LEX}(n) \rightarrow \text{LEX}(n')$ ), which would create a cycle in the graph.
- The transition **Right-Arc** adds an arc  $n \rightarrow n'$  from the node  $n$  on top of the stack to the next input token  $n'$ , licensed by an appropriate grammar rule, and shifts (pushes)  $n'$  onto the stack. The reason that the dependent node is immediately shifted is the same as in the preceding case, i.e. to prevent the creation of cycles. (Since  $n'$  must be reduced before  $n$  can become topmost again, no further arc linking these nodes can ever be added.)
- The transition **Reduce** simply reduces (pops) the node  $n$  on top of the stack, provided that this node has a head. This transition is needed for cases where a single head has multiple dependents on the right, in which case the closer dependent nodes must be reduced before arcs can be added to the more distant ones. The condition that the reduced node has a head is necessary to ensure that the dependency graph is projective, since otherwise there could be ungoverned nodes between a head and its dependent.
- The transition **Shift**, finally, simply shifts (pushes) the next input token  $n$  onto the stack. This transition is needed for cases where a right dependent has its own left dependents, in which case these dependents have to be reduced (by **Left-Arc** transitions) before the arc can be added from the head to the right dependent. Moreover, the **Shift** transition, which has no condition associated with it except that the input list is non-empty, is needed to guarantee termination.

The transitions **Left-Arc** and **Reduce** have in common that they reduce the stack size by 1 without affecting the length of the input list. I will therefore call these transitions POP-transitions in the following. In a similar fashion, **Right-Arc** and **Shift** have in common that

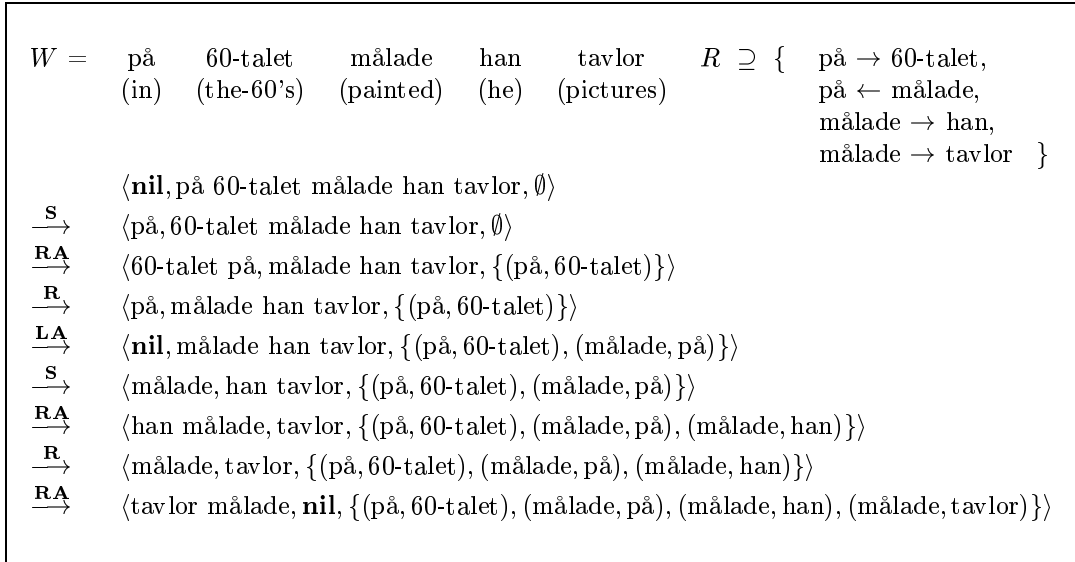


Figure 3: Parse of Swedish sentence

they reduce the length of the input list by 1 and increases the size of the stack by 1 and will therefore be called PUSH-transitions.

As it stands, this transition system is nondeterministic, since several transitions often apply to the same configuration, and in order to get a deterministic parser we need to impose some scheduling policy on top of the system. The simplest way to do this is to use a constant priority ordering of transitions **Left-Arc** > **Right-Arc** > **Reduce** > **Shift** (where  $a > b$  means that  $a$  has higher priority than  $b$ ). Figure 3 shows how a reduced variant of the Swedish sentence in Figure 1 can be parsed using this priority ordering (with the obvious transition labels **LA** = **Left-Arc**, **RA** = **Right-Arc**, **R** = **Reduce**, **S** = **Shift**).

Deciding how to resolve these conflicts is crucial from the point of view of parsing accuracy, as we will see in the next section, but it does not affect the basic properties of the parser with respect to running time and well-formedness of the dependency graph produced. For the remainder of this section, I will therefore simply assume that there exists some unspecified determinization of the parser (perhaps randomized) such that exactly one of the permissible transitions is chosen for every non-terminal configuration.

- **Proposition 1:** Given an input string  $W$  of length  $n$ , the parser terminates after at most  $2n$  transitions.
- **Proof of Proposition 1:** We first show that a transition sequence starting from an initial configuration  $\langle \mathbf{nil}, W, \emptyset \rangle$  can contain at most  $n$  PUSH-transitions and that such a sequence must be terminating:
  1. PUSH-transitions have as a precondition that the current input list is non-empty and decreases its length by 1. Since there are no transitions that increase the length of the input list, the maximum number of PUSH-transitions is  $n$ . For the same reason, a transition sequence containing  $n$  PUSH-transitions must end in a configuration  $\langle S, \mathbf{nil}, A \rangle$ , which is terminal.

We then show that a transition sequence containing  $n$  PUSH-transitions can contain at most  $n$  POP-transitions:

2. POP-transitions have as a precondition that the stack  $S$  is non-empty and have as an effect that the size of  $S$  is decreased by 1. Since the initial configuration has an empty  $S$ , and since the only transitions that increase the size of  $S$  are PUSH-transitions of which there can be no more than  $n$  instances, it follows that the maximum number of POP-transitions is also  $n$ .

Given that the number of POP-transitions is bounded by the number of PUSH-transitions, and given that at least one PUSH-transition (**Shift**) is applicable to every non-terminal configuration, we conclude that, for every initial configuration  $\langle \mathbf{nil}, W, \emptyset \rangle$  with an input list  $W$  of length  $n$ , there exists a transition sequence containing exactly  $n$  PUSH-transitions and at most  $n$  POP-transitions leading to a terminal configuration  $\langle S, \mathbf{nil}, A \rangle$ .  $\square$

The practical significance of Proposition 1 is that the parser is guaranteed to be both efficient and robust. As long as each transition can be performed in constant time, which only requires a reasonable choice of data structures for lookup of grammar rules and graph arcs, the worst case running time of the algorithm will be linear in the length of the input. Moreover, even if the parser does not succeed in building a well-formed dependency graph, it is always guaranteed to terminate.

- **Proposition 2:** The dependency graph  $G = (N_W, A)$  given at parser termination is projective and acyclic.
- **Proof of Proposition 2:** For projectivity we need to show that, for every input string  $W$ , if  $\langle \mathbf{nil}, W, \emptyset \rangle \rightarrow^* \langle S, \mathbf{nil}, A \rangle$  then the graph  $(N_W, A)$  is projective, which means that, for every pair  $(n, n') \in A$ ,  $n$  and  $n'$  are graph adjacent in  $G$ . Assume  $\langle \mathbf{nil}, W, \emptyset \rangle \rightarrow^* \langle S, \mathbf{nil}, A \rangle$  and assume  $(n, n') \in A$ . Since graph adjacency is a symmetric relation, we can without loss of generality assume that  $n < n'$ . Then we have  $\langle \mathbf{nil}, W, \emptyset \rangle \rightarrow^* \langle S', n|n_1|\dots|n_k|n'|I, A' \rangle \rightarrow \langle n|S', n_1|\dots|n_k|n'|I, A'' \rangle \rightarrow^* \langle n|S', n'|I, A''' \rangle \rightarrow^* \langle S, \mathbf{nil}, A \rangle$ . What we need to show is that all of  $n_1, \dots, n_k$  are dominated by  $n$  or  $n'$  in  $A$ , but since no arcs involving  $n_1, \dots, n_k$  can be in  $A''$  or  $A - A'''$ , it is sufficient to show that this holds in  $A'' - A'''$ . We first observe that the reduction of  $k$  nodes requires exactly  $2k$  transitions, since every reduction requires a **Shift** followed by a **Left-Arc**, or a **Right-Arc** followed by a **Reduce** (although the two transitions in each pair need not be adjacent in the sequence). Let  $\Phi(k)$  be the claim that if  $\langle n|S', n_1|\dots|n_k|n'|I, A'' \rangle \rightarrow^{2k} \langle n|S', n'|I, A''' \rangle$  then all of  $n_1, \dots, n_k$  are dominated by  $n$  or  $n'$  in  $A'''$ , parameterized on the number  $k$  of nodes between  $n$  and  $n'$ . We now use an inductive proof to show that  $\Phi(k)$  holds for all  $k \geq 0$ :

1. **Basis:** If  $k = 0$  then  $n$  and  $n'$  are string adjacent and  $\Phi(k)$  holds vacuously.
2. **Induction:** Assume  $\Phi(k)$  ( $k \geq 0$ ) and assume that  $\langle n|S', n_1|\dots|n_k|n'|I, A'' \rangle \rightarrow^{2(k+1)} \langle n|S', n'|I, A''' \rangle$ . We begin by noting that the first transition in this sequence must be a PUSH-transition, because otherwise  $n$  would be reduced and would not be on the stack in the final configuration. Hence, we need to consider two cases:
  - (a) If the first transition is **Right-Arc**, then  $n_1$  must be reduced with a **Reduce** transition and there exists some  $m \geq 0$  such that:

$$\begin{aligned}
& \langle n|S', n_1| \cdots |n_k|n'|I, A'' \rangle && \rightarrow \\
& \langle n_1|n|S', n_2| \cdots |n_k|n'|I, A'' \cup \{(n, n_1)\} \rangle && \rightarrow^{2m} \\
& \langle n_1|n|S', n_{2+m}| \cdots |n_k|n'|I, A'''' \rangle && \rightarrow \\
& \langle n|S', n_{2+m}| \cdots |n_k|n'|I, A'''' \rangle && \rightarrow^{2(k+1)-(2m+2)} \\
& \langle n|S', n'|I, A'''' \rangle
\end{aligned}$$

Since  $2(k+1) - (2m+2) \leq 2k$ , we can use the inductive hypothesis to infer that  $n$  and  $n'$  are adjacent.

- (b) If the first transition is **Shift**, then  $n_1$  must be reduced with a **Left-Arc** transition and there exists some  $m \geq 0$  such that:

$$\begin{aligned}
& \langle n|S', n_1| \cdots |n_k|n'|I, A'' \rangle && \rightarrow \\
& \langle n_1|n|S', n_2| \cdots |n_k|n'|I, A'' \rangle && \rightarrow^{2m} \\
& \langle n_1|n|S', n_{2+m}| \cdots |n_k|n'|I, A'''' \rangle && \rightarrow \\
& \langle n|S', n_{2+m}| \cdots |n_k|n'|I, A'''' \cup \{(n_{2+m}, n_1)\} \rangle && \rightarrow^{2(k+1)-(2m+2)} \\
& \langle n|S', n'|I, A'''' \rangle
\end{aligned}$$

Again, it follows from the inductive hypothesis that  $n$  and  $n'$  are adjacent.

The proof that  $(N_W, A)$  is free of cycles is analogous to the proof of projectivity, except that we consider the claim  $\Psi(k)$  that if  $\langle n|S', n_1| \cdots |n_k|n'|I, A'' \rangle \rightarrow^{2k} \langle n|S', n'|I, A'''' \rangle$  then there is no path from  $n$  to  $n'$  or from  $n'$  to  $n$ , again parameterized on the number  $k$  of nodes between  $n$  and  $n'$ . Here the base case follows from the definition of **Left-Arc** and **Right-Arc**, which prevents cycles of length 2.  $\square$

In virtue of Proposition 2, we know that even if a string  $W$  is rejected by the parser, the resulting dependency graph  $(N_W, A)$  is projective and acyclic, although not necessarily connected. In fact, the graph will consist of a number of connected components, each of which forms a well-formed dependency graph for a substring of  $W$ . This is important from the point of view of robustness, since each of these components represent a partial analysis of the input string.

## 4 Empirical Evaluation

In order to estimate the parsing accuracy that can be expected with the algorithm described in the preceding section and a grammar consisting of directed  $D$ -rules, a small experiment was performed using data from the Stockholm-Umeå Corpus of written Swedish (SUC [29]), which is a balanced corpus consisting of fictional and non-fictional texts, organized in the same way as the Brown corpus of American English. The experiment is based on a random sample consisting of about 4000 words, made up of 16 connected segments, and containing 257 sentences in total, which have been manually annotated with dependency graphs by the author.

The Stockholm-Umeå Corpus is annotated for parts of speech (and manually corrected), and the tagged sentences were used as input to the parser. This means that the vocabulary of the grammar consists of word-tag pairs, although most of the grammar rules used only take parts of speech into account. The grammar used in the experiment is hand-crafted and contains a total of 126 rules, divided into 90 left-headed rules (of the form  $w \rightarrow w'$ ) and 36 right-headed rules (of the form  $w \leftarrow w'$ ).

Parser	Mean	Std
Baseline	80.0	13.2
S/R	87.8	11.0
S/RA	89.0	10.6

Table 1: Attachment scores (mean and standard deviation)

Parsing accuracy was measured by the *attachment score* used by Eisner [15] and Collins et al. [7], which is computed as the proportion of words in a sentence that is assigned the correct head (or no head if the word is a root). The overall attachment score was then calculated as the mean attachment score over all sentences in the sample.

As mentioned in the previous section, different scheduling policies for the parser transitions yield different deterministic parsers. In this experiment, three different versions of the parser were compared:

- The baseline parser uses the constant priority ordering of transitions defined in section 3 **Left-Arc** > **Right-Arc** > **Reduce** > **Shift** (cf. also Figure 3).
- The second parser, called S/R, retains the ordering **Left-Arc** > **Right-Arc** > **Reduce**, **Shift** but uses the following simple rule to resolve **Shift/Reduce** conflicts:

If the node on top of the stack can be a transitive head of the next input token (according to the grammar) then **Shift**; otherwise **Reduce**.

- The third parser, called S/RA, in addition uses a simple lookahead for resolving **Shift/Right-Arc** conflicts, preferring a **Shift**-transition over a **Right-Arc**-transition in configurations where the token on top of the stack is a verb and the next token could be a post-modifier of this verb but could also be a pre-modifier of a following token, as in the following example (cf. Figure 1):

PN	<b>VB</b>	<b>AB</b>	<b>JJ</b>	NN
han	<b>målar</b>	<b>extremt</b>	<b>djärva</b>	tavlor
(he)	<b>(paints)</b>	<b>(extremely)</b>	<b>(bold)</b>	(pictures)

Making a **Shift**-transition in these cases is equivalent to a general preference for the pre-modifier interpretation.

Table 1 shows the mean attachment score and standard deviation obtained for the three different parsers. We can see that the parsing accuracy improves with the more complex scheduling policies, all differences being statistically significant despite the relatively small sample (paired *t*-test,  $\alpha = .05$ ). There are no directly comparable results for Swedish text, but Eisner [15] reports an accuracy of 90% for probabilistic dependency parsing of English text, sampled from the *Wall Street Journal* section of the Penn Treebank. Moreover, if the correct part of speech tags are given with the input, accuracy increases to almost 93%. Collins et al. [7] report an accuracy of 91% for English text (the same corpus as in Eisner [15]) and 80% accuracy for Czech text. Given that Swedish is intermediate between English and Czech with regard to inflectional richness and freedom of word order, the results seem rather promising, even with the 3% drop in accuracy that can be expected when a part of speech tagger is used to preprocess the input

(Eisner [15]). However, it should also be remembered that the empirical basis for evaluation is still very small. With a 95% confidence interval, the accuracy of the best parser can be estimated to lie in the range 85–93%, so a conservative conclusion is that a parsing accuracy above 85% is achievable.

## 5 Conclusion

The parsing algorithm presented in this paper has two attractive properties. It is robust, in the sense that it produces a projective and acyclic dependency graph for any input string, and it is efficient, producing these graphs in linear time. The crucial question for further research is whether the algorithm can achieve good enough accuracy to be useful in practical parsing systems. So far, it has been used with deterministic scheduling policies and simple directed *D*-rules to achieve reasonable accuracy when parsing unrestricted Swedish text. Topics to be investigated in the future include both alternative scheduling policies, possibly involving stochastic models, and alternative grammar formalisms, encoding stronger constraints on well-formed dependency structures.

## References

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles Techniques, and Tools*. Addison Wesley, 1986.
- [2] Cristina Barbero, Leonardo Lesmo, Vincenzo Lombardo, and Paola Merlo. Integration of syntactic and lexical information in a hierarchical dependency grammar. In Sylvain Kahane and Alain Polguère, editors, *Proceedings of the Workshop on Processing of Dependency-Based Grammars*, pages 58–67, Université de Montréal, Quebec, Canada, August 1998.
- [3] Glenn Carroll and Eugene Charniak. Two experiments on learning probabilistic dependency grammars from corpora. Technical Report TR-92, Department of Computer Science, Brown University, 1992.
- [4] Michael Collins. A new statistical parser based on bigram lexical dependencies. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 184–191, Santa Cruz, CA, 1996.
- [5] Michael Collins. Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pages 16–23, Madrid, Spain, 1997.
- [6] Michael Collins. *Head-Driven Statistical Models for Natural Language Parsing*. PhD thesis, University of Pennsylvania, 1999.
- [7] Michael Collins, Jan Hajič, Eric Brill, Lance Ramshaw, and Christoph Tillmann. A Statistical Parser of Czech. In *Proceedings of 37th ACL Conference*, pages 505–512, University of Maryland, College Park, USA, 1999.

- [8] Jacques Courtin and Damien Genthial. Parsing with dependency relations and robust parsing. In Sylvain Kahane and Alain Polguère, editors, *Proceedings of the Workshop on Processing of Dependency-Based Grammars*, pages 95–101, Université de Montréal, Quebec, Canada, August 1998.
- [9] Michael A. Covington. A dependency parser for variable-word-order languages. Technical Report AI-1990-01, University of Georgia, Athens, GA, 1990.
- [10] Michael A. Covington. Discontinuous dependency parsing of free and fixed word order: Work in progress. Technical Report AI-1994-02, University of Georgia, Athens, GA, 1994.
- [11] Ralph Debusmann. A declarative grammar formalism for dependency grammar. Master’s thesis, Computational Linguistics, Universität des Saarlandes, November 2001.
- [12] Denys Duchier. Axiomatizing dependency parsing using set constraints. In *Sixth Meeting on Mathematics of Language*, pages 115–126, Orlando, Florida, July 1999.
- [13] Denys Duchier. Lexicalized syntax and topology for non-projective dependency grammar. In *Joint Conference on Formal Grammars and Mathematics of Language FGMOL’01*, Helsinki, August 2001.
- [14] Denys Duchier. Configuration of labeled trees under lexicalized constraints and principles. *Journal of Language and Computation*, 1, 2002.
- [15] Jason M. Eisner. An empirical comparison of probability models for dependency grammar. Technical Report IRCS-96-11, Institute for Research in Cognitive Science, University of Pennsylvania, 1996.
- [16] Jason M. Eisner. Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of COLING-96*, Copenhagen, 1996.
- [17] Jason M. Eisner. Bilexical grammars and their cubic-time parsing algorithms. In Harry Bunt and Anton Nijholt, editors, *Advances in Probabilistic and Other Parsing Technologies*. Kluwer, 2000.
- [18] Haim Gaifman. Dependency systems and phrase-structure systems. *Information and Control*, 8:304–337, 1965.
- [19] David G. Hays. Dependency theory: A formalism and some observations. *Language*, 40:511–525, 1964.
- [20] Richard A. Hudson. *English Word Grammar*. Blackwell, 1990.
- [21] Mitchell P. Marcus. *A Theory of Syntactic Recognition for Natural Language*. MIT Press, 1980.
- [22] Hiroshi Maruyama. Structural disambiguation with constraint propagation. In *Proceedings of the 28th ACL*, pages 31–38, Pittsburgh, PA, 1990.
- [23] Igor Mel’cuk. *Dependency Syntax: Theory and Practice*. State University of New York Press, 1988.

- [24] Wolfgang Menzel and Ingo Schröder. Decision procedures for dependency parsing using graded constraints. In Sylvain Kahane and Alain Polguère, editors, *Proceedings of the Workshop on Processing of Dependency-Based Grammars*, pages 78–87, Université de Montréal, Québec, Canada, August 1998.
- [25] Christer Samuelsson. A statistical theory of dependency syntax. In *Proceedings COLING-2000*. Morgan Kaufmann, 2000.
- [26] Petr Sgall, Eva Hajicova, and Jarmila Panevova. *The Meaning of the Sentence in Its Pragmatic Aspects*. Reidel, 1986.
- [27] Daniel Sleator and Davy Temperley. Parsing English with a link grammar. Technical Report CMU-CS-91-196, Carnegie Mellon University, Computer Science, 1991.
- [28] Daniel Sleator and Davy Temperley. Parsing English with a link grammar. In *Third International Workshop on Parsing Technologies*, 1993.
- [29] Stockholm Umeå Corpus. Version 1.0. Produced by Department of Linguistics, Umeå University and Department of Linguistics, Stockholm University. ISBN 91-7191-348-3., August 1997.
- [30] Lucien Tesnière. *Éléments de syntaxe structurale*. Editions Klincksieck, 1959.