Storage in Deterministic Dependency Parsing

Joakim Nivre

1. Introduction

There are two naive strategies that we can use when faced with the ubiquitous phenomenon of ambiguity in natural language processing: brute-force enumeration and deterministic disambiguation. In brute-force enumeration, we simply enumerate all the fully specified representations that are compatible with a given input, e.g., all the possible parse trees for an input string given some grammar. This often leads to serious efficiency problems with respect to both time and space. In deterministic disambiguation, we find some more or less principled way of resolving each ambiguity as it arises, which means that we only derive a single fully specified representation for each input. While this avoids the efficiency problems associated with brute-force enumeration, it can instead lead to erroneous decisions because of early commitment, a problem that is often aggravated by error propagation.

More sophisticated techniques that have been proposed for dealing with this problem usually have in common that they avoid early commitment by efficiently computing and representing a set of possible interpretations without enumerating them. A typical example of this is the technique known as *storage* in computational semantics, often referred to as *Cooper storage* after the pioneering work of Robin Cooper (Cooper 1975), which is a mechanism for postponing the resolution of scope ambiguities and for efficiently computing and representing the unresolved interpretation. Another example is the use of dynamic programming and packed parse forests in syntactic parsing, which allows the derivation and storage of an exponentially large number of parse trees in polynomial time and space. However, it is worth noting that, although this makes parsing tractable without early commitment, it is much less efficient than deterministic disambiguation, which can often be realized in linear time and space.

In this paper, I discuss how techniques inspired by storage in computational semantics can be used to improve a technique for dependency-based syntactic parsing that is essentially an instance of naive deterministic disam-

biguation. The ultimate goal is to be able to improve parsing accuracy without compromising efficiency by using storage-like mechanisms to postpone certain disambiguation decisions, e.g., about the syntactic function of topicalized constituents. This paper is limited to a discussion of possible techniques for reaching this goal. The experimental evaluation of different methods is left for future research.

The paper is structured as follows. In section 2, I introduce deterministic dependency parsing, supported by treebank-induced classifiers. In section 3, I exemplify the problems that arise from the early commitment inherent in deterministic disambiguation, using some examples from Swedish syntax. In section 4, I then discuss two different ways of introducing storage techniques into deterministic dependency parsing. In section 5, finally, I conclude with some suggestions for future research.

2. Deterministic Dependency Parsing

Deterministic dependency parsing has recently emerged as not only one of the most efficient but also one of the most accurate methods for dependency parsing, provided that the deterministic parser is guided by a classifier trained on treebank data. This method was pioneered by Kudo and Matsumoto (2002) for Japanese and Yamada and Matsumoto (2003) for English and has later been developed and applied to a wide range of languages by Nivre, Hall, and Nilsson (2004), Cheng, Asahara, and Matsumoto (2004), and Attardi (2006), among others. In the CoNLL-X shared task on multilingual dependency parsing, the deterministic classifier-based approach was represented by one of the two top performing systems (Nivre et al. 2006). In this section, I begin by defining the representations used in dependency parsing, move on to discuss deterministic parsing algorithms, and finally explain the role of treebank-induced classifiers for accurate deterministic parsing.

2.1. Dependency Graphs

In dependency parsing, the syntactic analysis of a sentence is represented by a dependency graph, which we define as a labeled directed graph, the nodes of which are indices corresponding to the tokens of a sentence. Formally:

- Given a set R of dependency types (arc labels), a *dependency graph* for a sentence $x = (w_1, \ldots, w_n)$ is a labeled directed graph $G = (V, E, L)$, where:

  1. $V = \{0, 1, 2, \ldots n\}$
  2. $E \subseteq V \times V$
  3. $L : E \rightarrow R$

- A dependency graph G is *well-formed* if and only if:

  1. The node 0 is a root (ROOT).
  2. G is (weakly) connected (CONNECTEDNESS).
  3. Every node has at most one head, i.e., if $i \rightarrow j$ then there is no node k such that $k \neq i$ and $k \rightarrow j$ (SINGLE-HEAD).

The set V of *nodes* (or *vertices*) is the set of non-negative integers up to and including n. This means that every token index i of the sentence is a node ($1 \leq i \leq n$) and that there is a special node 0, which does not correspond to any token of the sentence and which will always be a root of the dependency graph (the only root in a well-formed dependency graph).

The set E of *arcs* (or *edges*) is a set of ordered pairs (i, j), where i and j are nodes. Since arcs are used to represent dependency relations, we will say that i is the *head* and j is the *dependent* of the arc (i, j). As usual, we will use the notation $i \rightarrow j$ to mean that there is an arc connecting i and j (i.e., $(i, j) \in E$) and we will use the notation $i \rightarrow^* j$ for the reflexive and transitive closure of the arc relation E (i.e., $i \rightarrow^* j$ if and only if $i = j$ or there is a path of arcs connecting i to j).

The function L assigns a dependency type (arc label) $r \in R$ to every arc $e \in E$. We will use the notation $i \xrightarrow{r} j$ to mean that there is an arc labeled r connecting i to j (i.e., $(i, j) \in E$ and $L((i, j)) = r$).

Figure 1 shows a Swedish sentence with a well-formed dependency graph. Since a well-formed dependency graph is always a tree rooted at 0 (Nivre 2006), I will from now on refer to well-formed dependency graphs as *dependency trees*.

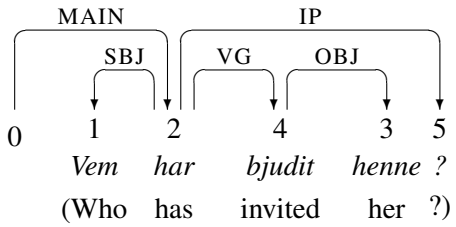

Figure 1. Dependency structure for Swedish sentence (subject interrogative)

2.2. Parsing Algorithm

The task for a disambiguating dependency parser is to derive the correct dependency tree G for a given sentence $x = (w_1, \ldots, w_n)$. A deterministic parser solves this problem by deriving a single analysis for each sentence. Most of the algorithms proposed for deterministic dependency parsing have similarities with shift-reduce parsing for constituency-based representations. Thus, the algorithm of Yamada and Matsumoto (2003), which performs unlabeled dependency parsing only, have only three elementary parsing actions, one *shift* action and two *reduce* actions (one for head-initial structures, one for head-final structures). The algorithm of Nivre (2003) can be seen as a variation of this approach, where the *reduce* action for head-initial structures is split into one action for adding the dependency arc and a separate action for popping the dependent from the stack. This allows arc-eager parsing, which reduces the amount of nondeterminism in the parsing process and makes parsing strictly incremental (Nivre 2004). This algorithm can be characterized as follows:

- Given a set R of dependency types (arc labels) and a sentence $x = (w_1, \ldots, w_n)$, a parser configuration for x is a triple $c = (\sigma, \tau, G)$, where:
  1. σ is a stack of nodes i ($0 \leq i \leq j$ for some $j \leq n$).
  2. τ is a sequence of nodes $(k, k+1, k+2, \ldots n)$ ($k > j$).
  3. G is a dependency graph for x.
- The parser is initialized to $((0), (1, 2, \ldots, n), G_0)$, where $G_0 = (\{0, 1, 2, \ldots, n\}, \emptyset, \emptyset)$.

- The parser terminates in any configuration of the form $(\sigma, (), G)$, outputting a dependency graph G', which differs from G only in that, for every root i in G such that $i \neq 0$, there is an arc $0 \rightarrow i$ in G'.
- The following transitions (parser actions) are possible in a non-terminal configuration satisfying the constraints stated for each transition:

  1. $(\sigma|i, j|\tau, G) \Longrightarrow (\sigma, j|\tau, G[j \xrightarrow{r} i])$ $\quad \neg\exists k : k \rightarrow i \text{ in } G$
  2. $(\sigma|i, j|\tau, G) \Longrightarrow (\sigma|i|j, \tau, G[i \xrightarrow{r} j])$ $\quad i \neq 0, \neg\exists k : k \rightarrow j \text{ in } G$
  3. $(\sigma|i, \tau, G) \Longrightarrow (\sigma, \tau, G)$ $\quad \exists k : k \rightarrow i \text{ in } G$
  4. $(\sigma, i|\tau, G) \Longrightarrow (\sigma|i, \tau, G)$

  where $\sigma|i$ is a stack with top element i, $j|\tau$ is a list with head j and tail τ, and $G[i \rightarrow j]$ is the graph that differs from G only be the addition of the arc (i, j).

The idea is that the sequence τ represents the remaining input tokens in a left-to-right pass over the input sentence x; the stack σ contains partially processed nodes that are still candidates for dependency arcs, either as heads or dependents, and the graph G is the partially constructed dependency graph. Provided that every transition can be performed in constant time, the algorithm has time complexity $O(n)$, where n is the number of words in the input sentence (Nivre 2003). Moreover, the algorithm guarantees that the graph G given at termination is acyclic and satisfies ROOT and SINGLE-HEAD, which means that it can be transformed to a dependency tree by adding arcs from the special root 0 to any other roots in the graph (Nivre 2006). One limitation of this parsing algorithm is that only derives strictly projective dependency trees (i.e., trees where the projection of every head is continuous), whereas dependency parsing in general needs to allow non-projective trees as well. However, this limitation can be overcome by special pre- and post-processing techniques for recovering non-projective dependencies, so-called *pseudo-projective parsing* (Nivre and Nilsson 2005).

2.3. Classifier-Based Parsing

Deterministic parsing algorithms need an *oracle* for predicting the next parser action at nondeterministic choice points. In *classifier-based* parsing, such oracles are approximated with *classifiers* trained on data derived from treebanks. This is essentially a form of *history-based parsing*, where features of

the parsing history are used to predict the next parser action (Black et al. 1992; Magerman 1995; Ratnaparkhi 1997).

Let $\Phi(\sigma, \tau, G)$ be a feature vector representation of a parser configuration with stack σ, input sequence τ and dependency graph G. Training data for the classifier can be generated by running the parser on a sample of treebank data, using the gold standard dependency graph as an oracle to predict the next parser action and constructing one training instance $(\Phi(\sigma, \tau, G), t)$ for each occurrence of a transition t in a parser configuration represented by $\Phi(\sigma, \tau, G)$. The features in $\Phi(\sigma, \tau, G)$ can be arbitrary features of the input x and the partially built graph G but usually consist mainly of linguistic attributes of input tokens, including their dependency types according to G.

The history-based classifier can be trained with any of the available supervised methods for function approximation, such as memory-based learning or decision trees, but the best performance has so far been achieved with support vector machines (Vapnik 1995), which has been used for parsing in a wide range of experiments (Kudo and Matsumoto 2002; Yamada and Matsumoto 2003; Sagae and Lavie 2005; Nivre et al. 2006).

3. The Problem

In the previous section I have sketched how it is possible to achieve highly accurate dependency parsing, despite the naive approach of deterministic disambiguation, using history-based classifiers trained on large data sets derived from treebanks. Nevertheless, it is clear that some of the errors performed by deterministic incremental parsers are due to the fact that the parser sometimes has to commit to an analysis before all the relevant evidence can be taken into account.

A typical example is the analysis of topicalized constituents in Swedish (and other verb-second languages), exemplified by *wh*-questions in figure 1 and figure 2. When processing these sentences incrementally, the parser has to make an early decision about whether the topicalized constituent (*Vem*) should be linked to the finite (auxiliary) verb (*har*) by the subject relation (as in figure 1) or not at all (as in figure 2). To some extent, this problem can be alleviated by the lookahead available in the input string, but there is never any guarantee that this will lead to the correct decision, despite the fact that the necessary information is available in the rest of the sentence.[1]

Examples such as these abound in natural language syntax. Let us consider

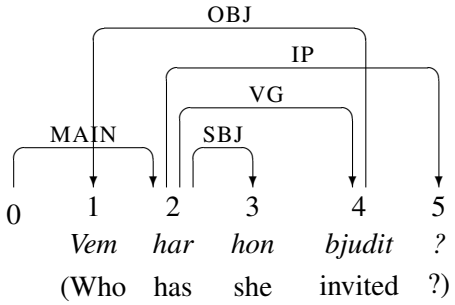

Figure 2. Dependency structure for Swedish sentence (object interrogative)

just one more case from Swedish syntax:

(1) a. *Kommer han, eller hon?*
Comes he, or she?
'Does he come, or she?'
b. *Kommer han, går hon.*
Comes he, goes she.
'If he comes, she goes.'

When parsing (1a) with the algorithm described above, the first action should be to add an arc labeled MAIN from the root 0 to the token 1, since the finite verb *Kommer* is the head of the main clause. In (1b), by contrast, no such arc should be added, because the finite verb *Kommer* in this case is the head of a (conditional) subordinate clause and should therefore be a dependent of the main clause verb *går*.

So, the question I want to pose is what we can do to improve the parser's ability to make the correct decision in cases like these, while still maintaining the attractive time and space complexity of deterministic parsing.

4. Two Possible Solutions

I will now outline two methods for postponing difficult decisions of the kind exemplified in the preceding section, both inspired by the notion of *storage*.

4.1. Multi-Pass Parsing

The most straightforward application of the storage principle is to simply store the tokens that we do not (yet) know where to attach on the stack. Without any other modification to the parsing algorithm, this would mean that the topicalized constituents in figure 1 and figure 2 would remain unattached throughout the parsing process and would at termination be attached to the special root node 0. So, in order to arrive at the desired analysis for each sentence, we would instead have to do a second pass over the input, where we attach tokens that have been stored on the stack, using all the evidence available in the partial dependency graph. This requires only two minor modifications to the algorithm, namely that we allow the parser to be initialized with an arbitrary (acyclic) dependency graph obeying ROOT and SINGLE-HEAD (rather than G_0), and that we output the graph G given at termination (rather than the graph G' obtained by attaching all roots in $\{1,\ldots,n\}$ to 0).

Given that one and the same sentence may contain several problematic constructions, we can either decide to have a single additional pass, where we deal with all the problematic phenomena together, or we can allow multiple passes, fixing only one type of construction in each pass. In either case, however, we want to train different classifiers for the different passes, in order to exploit the fact that the amount of information available in both input and output increases from earlier to later passes.

More precisely, if we want to divide the parsing problem into m passes, we need a partitioning function f on the arc relation and labeling function of an arbitrary dependency tree, such that if $f(E,L) = ((E_1,L_1)\ldots,(E_m,L_m))$ then $\bigcup_{i=1}^{m} E_i = E$, $E_i \cap E_j = \emptyset$ (for any i,j), L_i is a total function on E_i (for any i), and E_i contains exactly the dependency arcs that should be added in pass m. If we use G_j to denote the graph $(V, \bigcup_{i=1}^{j} E_i, \bigcup_{i=1}^{j} L_i)$, then we train the classifier for pass i using G_{i-1} as input and G_i as output. It is worth noting that the training scheme for the original parsing algorithm is just the special case of this where $m = 1$.

The basic idea in multi-pass parsing, i.e., that some decisions should be postponed until others have been made, is also the motivation for the iterative approach used by Kudo and Matsumoto (2002) and Yamada and Matsumoto (2003). However, one difference is that they use the same classifier for all passes and do not impose any upper bound on the number of passes but terminate only when no new arc has been added in the previous iteration. This means that the worst-case time complexity of this algorithm is $O(n^2)$ (even

though the worst case seldom occurs in practice). By contrast, if we can define linguistically motivated criteria for the partitioning of dependency arcs, such that the number of passes is always bounded by a constant *m*, which is independent of the length of the input sentence, then we can preserve the linear time complexity of the original algorithm. Interestingly, a proposal along these lines has already been made by Arnola (1998) (for a different parsing algorithm), who hypothesizes that the partitioning (in our terminology) can be defined in terms of dependency types (arc labels), so that there will be one pass for each distinct dependency type, and that there is an optimal sequence in which dependency types should be processed in order to maximize parsing accuracy while maintaining linear-time parsing.

4.2. Generalized Pseudo-Projective Parsing

Another way of implementing the storage idea is to add temporary arcs with temporary labels for problematic constructions. For example, we may decide to always attach topicalized constituents to the lowest node that dominates all its potential heads. In terms of the previous examples, this would mean that we always attach the topicalized *Vem* to the finite (auxiliary) verb *har*, but with a temporary arc label signifying that the real head of *Vem* is in the subtree dominated by *har*. In a post-processing phase, we can then perform a top-down search for the true head, again using all the information available in the rest of the derived dependency tree.

This approach can be seen as a generalization of the idea behind *pseudo-projective* parsing (Nivre and Nilsson 2005). As a matter of fact, since the dependency tree in figure 2 is non-projective, it can only be derived by first attaching *Vem* to *har*, with a temporary label signifying that the true head is a descendant of *har* with the dependency type VG. In a post-processing phase, we then perform a top-down breadth-first search in order to find out that the true head is the main verb *bjudit*. What I am now proposing is to use the same technique also for the tree in figure 1, which is strictly projective but where the true head and dependency type of *Vem* is difficult to determine before we have derived the rest of the structure.

In order to realize this idea we have to implement an instance of the following scheme:

1. Transform the training data for the parser so that problematic constructions are analyzed with temporary heads and dependency labels.
2. Train the parser on the transformed training data.
3. Parse new sentences into the transformed representation.
4. Perform an inverse transformation to the parser output, guided by the information in temporary dependency labels.

In order to preserve the linear time complexity of parsing, we need to ensure that both pre- and post-processing can be performed in linear time as well. Since the transformation of a single arc can usually be performed in time proportional to the total number of arcs, which is linear in string length, this requires either that all arcs can be processed together or that there is (again) a constant upper bound on the number of arcs that need to be processed for an arbitrary sentence.

The use of pre- and post-processing in deterministic dependency parsing has previously been studied not only in connection with pseudo-projective parsing but also as a general technique for improving accuracy under the heading of *tree transformations* (Nilsson, Nivre, and Hall 2006; Nilsson 2007). The generalized pseudo-projective parsing approach also has affinities with the notion of *temporary landing sites* proposed by Buch-Kromann (2006) in the context of parsing with Discontinuous Grammar.

5. Conclusion

In this paper I have proposed two ways of incorporating a notion of storage into deterministic, classifier-based dependency parsing, with the aim of improving accuracy without sacrificing efficiency. The most pressing need for future research is obviously an experimental evaluation of the proposed techniques. For the multi-pass approach it is also important to try to find linguistically motivated principles for partitioning dependency graphs. For generalized pseudo-projective parsing it is crucial to achieve a harmonious integration with pseudo-projective parsing in the original sense, so that non-projective dependencies can still be recovered. Needless to say, it is much too early to say whether any of these techniques will be as successful in dependency parsing as the original notion of storage has been in computational semantics.

Notes

1. Because of the composite tense form, the word order uniquely identifies the subject and the object in the example sentences.

References

Arnola, Harri
1998 On parsing binary dependency structure deterministically in linear time. Sylvain Kahane, and Alain Polguère (eds.), *Proceedings of the Workshop on Processing of Dependency-Based Grammars (ACL-COLING)*. 68–77.

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

Black, Ezra, Frederick Jelinek, John D. Lafferty, David M. Magerman, Robert L. Mercer, and Salim Roukos
1992 Towards history-based grammars: Using richer models for probabilistic parsing. *Proceedings of the 5th DARPA Speech and Natural Language Workshop*. 31–37.

Buch-Kromann, Matthias
2006 Discontinous Grammar. a model of human parsing and language acquisitiion. Ph.D. diss., Copenhagen Business School.

Cheng, Yuchang, Masayuki Asahara, and Yuji Matsumoto
2004 Deterministic dependency structure analyzer for Chinese. *Proceedings of the First International Joint Conference on Natural Language Processing (IJCNLP)*. 500–508.

Cooper, Robin
1975 Montague's semantic theory and transformational syntax. Ph.D. diss., University of Massachusetts at Amherst.

Kudo, Taku, and Yuji Matsumoto
2002 Japanese dependency analysis using cascaded chunking. *Proceedings of the Sixth Workshop on Computational Language Learning (CoNLL)*. 63–69.

Magerman, David M.
1995 Statistical decision-tree models for parsing. *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics (ACL)*. 276–283.

Nilsson, Jens
2007 Tree Transformations in Inductive Dependency Parsing. Licentiate Thesis, Växjö University.

Nilsson, Jens, Joakim Nivre, and Johan Hall
2006 Graph transformations in data-driven dependency parsing. *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics (COLING-ACL)*. 257–264.

Nivre, Joakim
2003 An efficient algorithm for projective dependency parsing. Gertjan Van Noord (ed.), *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*. 149–160.
2004 Incrementality in deterministic dependency parsing. Frank Keller, Stephen Clark, Matthew Crocker, and Mark Steedman (eds.), *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together (ACL)*. 50–57.
2006 *Inductive Dependency Parsing*. Springer.
Nivre, Joakim, Johan Hall, and Jens Nilsson
2004 Memory-based dependency parsing. Hwee Tou Ng, and Ellen Riloff (eds.), *Proceedings of the 8th Conference on Computational Natural Language Learning (CoNLL)*. 49–56.
Nivre, Joakim, Johan Hall, Jens Nilsson, Gülsen Eryiğit, and Svetoslav Marinov
2006 Labeled pseudo-projective dependency parsing with support vector machines. *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL)*. 221–225.
Nivre, Joakim, and Jens Nilsson
2005 Pseudo-projective dependency parsing. *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*. 99–106.
Ratnaparkhi, Adwait
1997 A linear observed time statistical parser based on maximum entropy models. *Proceedings of the Second Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1–10.
Sagae, Kenji, and Alon Lavie
2005 A classifier-based parser with linear run-time complexity. *Proceedings of the 9th International Workshop on Parsing Technologies (IWPT)*. 125–132.
Vapnik, Vladimir N.
1995 *The Nature of Statistical Learning Theory*. Springer.
Yamada, Hiroyasu, and Yuji Matsumoto
2003 Statistical dependency analysis with support vector machines. Gertjan Van Noord (ed.), *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT)*. 195–206.