Generation as Dependency Parsing

Alexander Koller and Kristina Striegnitz
Dept. of Computational Linguistics, Saarland University
{koller|kris}@coli.uni-sb.de

Abstract

Natural-Language Generation from flat semantics is an NP-complete problem. This makes it necessary to develop algorithms that run with reasonable efficiency in practice despite the high worst-case complexity. We show how to convert TAG generation problems into dependency parsing problems, which is useful because optimizations in recent dependency parsers based on constraint programming tackle exactly the combinatorics that make generation hard. Indeed, initial experiments display promising runtimes.

1 Introduction

Existing algorithms for realization from a flat input semantics all have runtimes which are exponential in the worst case. Several different approaches to improving the runtime in practice have been suggested in the literature – e.g. heuristics (Brew, 1992) and factorizations into smaller exponential subproblems (Kay, 1996; Carroll et al., 1999). While these solutions achieve some measure of success in making realization efficient, the contrast in efficiency to parsing is striking both in theory and in practice.

The problematic runtimes of generation algorithms are explained by the fact that realization is an NP-complete problem even using just context-free grammars, as Brew (1992) showed in the context of shake-and-bake generation. The first contribution of our paper is a proof of a stronger NP-completeness result: If we allow semantic indices in the grammar, realization is NP-complete even if we fix a single grammar. Our alternative proof shows clearly that the combinatorics in generation come from essentially the same sources as in parsing for free word order languages. It has been noted in the literature that this problem, too, becomes NP-complete very easily (Barton et al., 1987).

The main point of this paper is to show how to encode generation with a variant of tree-adjoining grammars (TAG) as a parsing problem with dependency grammars (DG). The particular variant of DG we use, Topological Dependency Grammar (TDG) (Duchier, 2002; Duchier and Debusmann, 2001), was developed specifically with efficient parsing for free word order languages in mind. The mere existence of this encoding proves TDG's parsing problem NP-complete as well, a result which has been conjectured but never formally shown so far. But it turns out that the complexities that arise in generation problems in practice seem to be precisely of the sort that the TDG parser can handle well. Initial experiments with generating from the XTAG grammar (XTAG Research Group, 2001) suggest that our generation system is competitive with state-of-the-art chart generators, and indeed seems to run in polynomial time in practice.

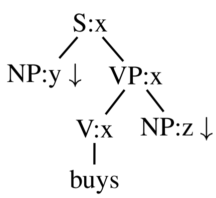
Next to the attractive runtime behaviour, our approach to realization is interesting because it may provide us with a different angle from which to look for tractable fragments of the general realization problem. As we will show, the computation that takes place in our system is very different from that in a chart generator, and may be more efficient in some cases by taking into account global information to guide local choices.

Plan of the Paper. We will define the problem we want to tackle in Section 2, and then show that it is NP-complete (Section 3). In Section 4, we sketch the dependency grammar formalism we use. Section 5 is the heart of the paper: We show how to encode TAG generation as TDG parsing, and discuss some examples and runtimes. We compare our approach to some others in Section 6, and conclude and discuss future research in Section 7.

2 The Realization Problem

In this paper, we deal with the subtask of natural language generation known as *surface realization*: given a grammar and a semantic representation, the problem is to find a sentence which is grammatical according to the grammar and expresses the content of the semantic representation.

We represent the semantic input as a multiset (bag) of ground atoms of predicate logic, such as {*buy(e,a,b), name(a,mary) car(b)*}. To encode syntactic information, we use a tree-adjoining grammar without feature structures (Joshi and Schabes, 1997). Following Stone and Doran (1997) and Kay (1996), we enhance this TAG grammar with a syntax-semantics interface in which nonterminal nodes of the elementary trees are equipped with index variables, which can be bound to individuals in the semantic input. We assume that the root node, all substitution nodes, and all nodes that admit adjunction carry such index variables. We also assign a semantics to every elementary tree, so that lexical entries are pairs of the form (φ, T), where φ is a multiset of semantic atoms, and T is an initial or auxiliary tree, e.g.

({*buy(x,y,z)*}, )

When the lexicon is accessed, x, y, z get bound to terms occurring in the semantic input, e.g. e, a, b in our example. Since we furthermore assume that every index variable that appears in T also appears in φ, this means that all indices occurring in T get bound at this stage.

The semantics of a complex tree is the multiset union of the semantics of the elementary trees involved. Now we say that the *realization problem* of a grammar G is to decide for a given input semantics S and an index i whether there is a derivation tree which is grammatical according to G, is assigned the semantics S, and has a root node with index i.

3 NP-Completeness of Realization

This definition is the simplest conceivable formalization of problems occurring in surface realization as a decision problem: It does not even require us to compute a single actual realization, just to check whether one exists. Every practical generation system generating from flat semantics will have to address this problem in one form or another.

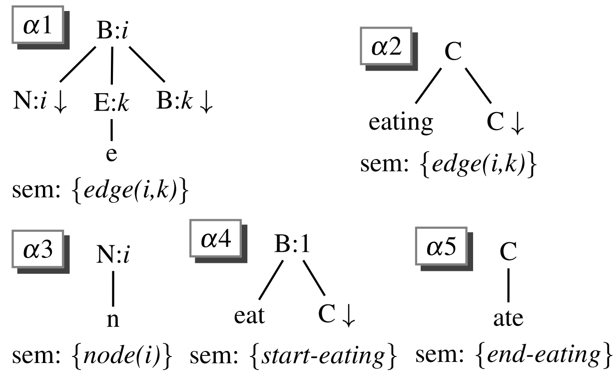
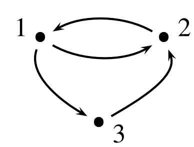


Figure 1: The grammar G_{ham}.

Now we show that this problem is NP-complete. A similar result was proved in the context of shake-and-bake generation by Brew (1992), but he needed to use the grammar in his encoding, which leaves the possibility open that for every single grammar G, there might be a realization algorithm tailored specifically to G which still runs in polynomial time. Our result is stronger in that we define a single grammar G_{ham} whose realization problem is NP-complete in the above sense. Furthermore, we find that our proof brings out the sources of the complexity more clearly. G_{ham} does not permit adjunction, hence the result also holds for context-free grammars with indices.



It is clear that the problem is in NP: We can simply guess the elementary trees we need and how to combine them, and then check in polynomial time whether they verbalize the semantics. The NP-hardness proof is by reducing the well-known HAMILTONIAN-PATH problem to the realization problem. HAMILTONIAN-PATH is the problem of deciding whether a directed graph has a cycle that visits each node exactly once, e.g. (1,3,2,1) in the graph shown above.

We will now construct an LTAG grammar G_{ham} such that every graph $G = (V, E)$ can be encoded as a semantic input S for the realization problem of G_{ham}, which can be verbalized if and only if G has a Hamiltonian cycle. S is defined as follows:

$$\begin{aligned} S &= \{node(i) \mid i \in V\} \\ &\cup \{edge(i,k) \mid (i,k) \in E\} \\ &\cup \{\textit{start-eating}, \textit{end-eating}\}. \end{aligned}$$

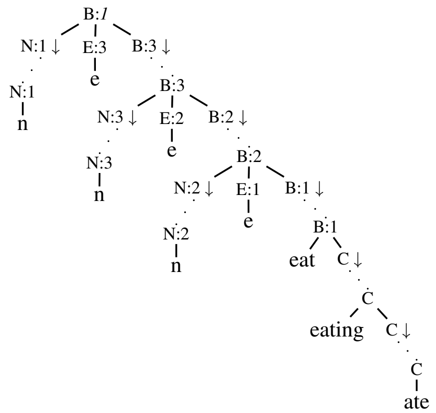


Figure 2: A derivation with G_{ham} corresponding to a Hamiltonian cycle.

The grammar G_{ham} is given in Fig. 1; the start symbol is B, and we want the root to have index 1. The tree α_1 models an edge transition from node i to the node k by consuming the semantic encodings of this edge and (by way of a substitution of α_3) of the node i. The second substitution node of α_1 can be filled either by another α_1, in which way a path through the graph is modelled, or by an α_4, in which case we switch to an "edge eating mode". In this mode, we can arbitrarily consume edges using α_2, and close the tree with α_5 when we're done. This is illustrated in Fig. 2, the tree corresponding to the cycle in the example graph above.

The Hamiltonian cycle of the graph, if one exists, is represented in the indices of the B nodes. The list of these indices is a path in the graph, as the α_1 trees model edge transitions; it is a cycle because it starts in 1 and ends in 1; and it visits each node exactly once, for we use exactly one α_1 tree for each $node$ literal. The edges which weren't used in the cycle can be consumed in the edge eating mode.

The main source for the combinatorics of the realization problem is thus the interaction of lexical ambiguity and the completely free order in the flat semantics. Once we have chosen between α_1 and α_2 in the realization of each $edge$ literal, we have determined which edges should be part of the prospective Hamiltonian cycle, and checking whether it really is one can be done in linear time. If, on the other hand, the order of the input placed restrictions on the structure of the derivation tree, we would again have information that told us when to switch into the edge eating mode, i.e. which edges should be part of the cycle. A third source of combinatorics which does not become so clear in this encoding is the configuration of the elementary trees. Even when we have committed to the lexical entries, it is conceivable that only one particular way of plugging them into each other is grammatical.

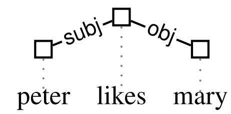


Figure 3: TDG parse tree for "Peter likes Mary."

4 Topological Dependency Grammar

These factors are exactly the same that make dependency parsing for free word order languages difficult, and it seems worthwhile to see whether optimized parsers for dependency grammars can also contribute to making generation efficient. We now sketch a dependency formalism which has an efficient parser and then discuss some of the important properties of this parser. In the next section, we will see how to employ the parser for generation.

4.1 The Grammar Formalism

The parse trees of topological dependency grammar (TDG) (Duchier and Debusmann, 2001; Duchier, 2002) are trees whose nodes correspond one-to-one to the words of the sentence, and whose edges are labelled, e.g. with syntactic relations (see Fig. 3). The trees are unordered, i.e. there is no intrinsic order among the children of a node. Word order in TDG is initially completely free, but there is a separate mechanism to specify constraints on linear precedence. Since completely free order is what we want for the realization problem, we do not need these mechanisms and do not go into them here.

The lexicon assigns to each word a set of lexical entries; in a parse tree, one of these lexical entries has to be picked for each node. The lexical entry specifies what labels are allowed on the incoming edge (the node's *labels*) and the outgoing edges (the node's *valency*). Here are some examples:

word	labels	valency
likes	$\emptyset$	$\{\mathsf{subj}, \mathsf{obj}, \mathsf{adv}*\}$
Peter	$\{\mathsf{subj}, \mathsf{obj}\}$	$\emptyset$
Mary	$\{\mathsf{subj}, \mathsf{obj}\}$	$\emptyset$

The lexical entry for "likes" specifies that the corresponding node does not accept any incoming edges (and hence must be the root), must have precisely one subject and one object edge going out, and can have arbitrarily many outgoing edges with label `adv` (indicated by $*$). The nodes for "Peter" and "Mary" both require their incoming edge to be labelled with either `subj` or `obj` and neither require nor allow any outgoing edges.

A well-formed dependency tree for an input sentence is simply a tree with the appropriate nodes, whose edges obey the labels and valency restrictions specified by the lexical entries. So, the tree in Fig. 3 is well-formed according to our lexicon.

4.2 TDG Parsing

The parsing problem of TDG can be seen as a search problem: For each node, we must choose a lexical entry and the correct mother-daughter relations it participates in. One strength of the TDG approach is that it is amenable to strong syntactic inferences that tackle specifically the three sources of complexity mentioned above.

The parsing algorithm (Duchier, 2002) is stated in the framework of *constraint programming* (Koller and Niehren, 2000), a general approach to coping with combinatorial problems. Before it explores all choices that are possible in a certain state of the search tree *(distribution)*, it first tries to eliminate some of the choices which definitely cannot lead to a solution by simple inferences *(propagations)*. "Simple" means that propagations take only polynomial time; the combinatorics is in the distribution steps alone. That is, it can still happen that a search tree of exponential size has to be explored, but the time spent on propagation in each of its node is only polynomial. Strong propagation can reduce the size of the search tree, and it may even make the whole algorithm run in polynomial time in practice.

The TDG parser translates the parsing problem into constraints over (variables denoting) finite sets of integers, as implemented efficiently in the Mozart programming system (Oz Development Team, 1999). This translation is complete: Solutions of the set constraint can be translated back to correct dependency trees. But for efficiency, the parser uses additional propagators tailored to the specific inferences of the dependency problem. For instance, in the "Peter likes Mary" example above, one such propagator could contribute the information that neither the "Peter" nor the "Mary" node can be an `adv` child of "likes", because neither can accept an `adv` edge. Once the choice has been made that "Peter" is the `subj` child of "likes", a propagator can contribute that "Mary" must be its `obj` child, as it is the only possible candidate for the (obligatory) `obj` child.

Finally, lexical ambiguity is handled by *selection constraints*. These constraints restrict which lexical entry should be picked for a node. When all possible lexical entries have some information in common (e.g., that there must be an outgoing `subj` edge), this information is automatically lifted to the node and can be used by the other propagators. Thus it is sometimes even possible to finish parsing without committing to single lexical entries for some nodes.

5 Generation as Dependency Parsing

We will now show how TDG parsing can be used to enumerate all sentences expressing a given input semantics, thereby solving the realization problem introduced in Section 2. We first define the encoding. Then we give an example and discuss some runtime results. Finally, we consider a particular restriction of our encoding and ways of overcoming it.

5.1 The Encoding

Let G be a grammar as described in Section 2; i.e. lexical entries are of the form (φ, T), where φ is a flat semantics and T is a TAG elementary tree whose nodes are decorated with semantic indices. We make the following simplifying assumptions. First, we assume that the nodes of the elementary trees of G are not labelled with feature structures. Next, we assume that whenever we can adjoin an auxiliary tree at a node, we can adjoin arbitrarily many trees at this node. The idea of multiple adjunction is not new (Schabes and Shieber, 1994), but it is simplified here because we disregard complex adjunction constraints. We will discuss these two restrictions in the conclusion. Finally, we assume that every lexical semantics φ has precisely one member; this restriction will be lifted in Section 5.4.

Now let's say we want to find the realizations of the input semantics $S = \{\varphi_1, \ldots, \varphi_n\}$, using the grammar G. The input "sentence" of the parsing

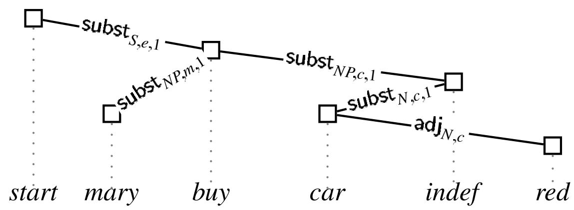


Figure 4: Dependency tree for "Mary buys a red car."

problem we construct is the sequence $\{start\} \cup S$, where $start$ is a special start symbol. The parse tree will correspond very closely to a TAG derivation tree, its nodes standing for the instantiated elementary trees that are used in the derivation.

To this end, we use two types of edge labels – substitution and adjunction labels. An edge with a substitution label $\mathsf{subst}_{A,i,p}$ from the node α to the node β (both of which stand for elementary trees) indicates that β should be plugged into the p-th substitution node in α that has label A and index i. We write $\mathsf{subst}(A)$ for the maximum number of occurrences of A as the label of substitution nodes in any elementary tree of G; this is the maximum value that p can take.

An edge with an adjunction label $\mathsf{adj}_{A,i}$ from α to β specifies that β is adjoined at some node within α carrying label A and index i and admitting adjunction. It does not matter for our purposes to which node in α β is adjoined exactly; the choice cannot affect grammaticality because there is no feature unification involved.

The dependency grammar encodes how an elementary tree can be used in a TAG derivation by restricting the labels of the incoming and outgoing edges via labels and valency requirements in the lexicon. Let's say that T is an elementary tree of G which has been matched with the input atom φ_r, instantiating its index variables. Let A be the label and i the index of the root of T. If T is an auxiliary tree, it accepts incoming adjunction edges for A and i, i.e. it gets the labels value $\{\mathsf{adj}_{A,i}\}$. If T is an initial tree, it will accept arbitrary incoming substitution edges for A and i, i.e. its labels value is

$$\{\mathsf{subst}_{A,i,p} \mid 1 \leq p \leq \mathsf{subst}(A)\}$$

In either case, T will require precisely one outgoing substitution edge for each of its substitution nodes, and it will allow arbitrary numbers of outgoing adjunction edges for each node where we can adjoin. That is, the valency value is as follows:

$$\begin{aligned}&\{\mathsf{subst}_{A,i,p} \mid \text{ex. substitution node } N \text{ in } T \text{ s.t. } A \\ &\qquad \text{is label, } i \text{ is index of } N\text{, and } N \text{ is} \\ &\qquad p\text{th substitution node for } A{:}i \text{ in } T\} \\ \cup\ &\{\mathsf{adj}_{A,i}* \mid \text{ex. node with label } A\text{, index } i \\ &\qquad \text{in } T \text{ which admits adjunction}\}\end{aligned}$$

We obtain the set of all lexicon entries for the atom φ_r by encoding all TAG lexicon entries which match φ_r as just specified. The start symbol, $start$, gets a special lexicon entry: Its labels entry is the empty set (i.e. it must be the root of the tree), and its valency entry is the set $\{\mathsf{subst}_{S,k,1}\}$, where k is the semantic index with which generation should start.

5.2 An Example

Now let us go through an example to make these definitions a bit clearer. Let's say we want to verbalize the semantics

$$\{name(m, mary), buy(e, m, c), \\ car(c), indef(c), red(c)\}$$

The LTAG grammar we use contains the elementary trees which are used in the tree in Fig. 5, along with the obvious semantics; we want to generate a sentence starting with the main event e. The encoding produces the following dependency grammar; the entries in the "atom" column are to be read as abbreviations of the actual atoms in the input semantics.

atom	labels	valency
start	$\emptyset$	$\{\mathsf{subst}_{S,e,1}\}$
buy	$\{\mathsf{subst}_{S,e,1}\}$	$\{\mathsf{subst}_{NP,c,1}, \mathsf{subst}_{NP,m,1}, \mathsf{adj}_{VP,e}*, \mathsf{adj}_{V,e}*\}$
mary	$\{\mathsf{subst}_{NP,m,1}, \mathsf{subst}_{NP,m,2}\}$	$\{\mathsf{adj}_{NP,1}*, \mathsf{adj}_{PN,m}*\}$
indef	$\{\mathsf{subst}_{NP,c,1}, \mathsf{subst}_{NP,c,2}\}$	$\{\mathsf{adj}_{NP,c}*\}$
car	$\{\mathsf{subst}_{N,c,1}\}$	$\{\mathsf{adj}_{N,c}*\}$
red	$\{\mathsf{adj}_{N,c}\}$	$\emptyset$

If we parse the "sentence"

start mary buy car indef red

with this grammar, leaving the word order completely open, we obtain precisely one parse tree, shown in Fig. 4. Reading this parse as a TAG derivation tree, we can reconstruct the derived tree in Fig. 5, which indeed produces the string "Mary buys a red car".

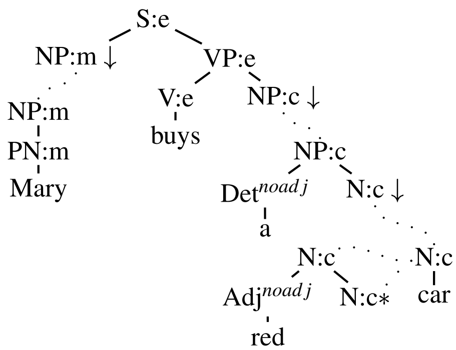


Figure 5: Derived tree for "Mary buys a red car."

5.3 Implementation and Experiments

The overall realization algorithm we propose encodes the input problem as a DG parsing problem and then runs the parser described in Section 4.2, which is freely available over the Web, as a black box. Because the information lifted to the nodes by the selection constraints may be strong enough to compute the parse tree without ever committing to unique lexical entries, the complete parse may still contain some lexical ambiguity. This is no problem, however, because the absence of features guarantees that every combination of choices will be grammatical. Similarly, a node can have multiple children over adjunction edges with the same label, and there may be more than one node in the upper elementary tree to which the lower tree could be adjoined. Again, all remaining combinations are guaranteed to be grammatical.

In order to get an idea of the performance of our realization algorithm in comparison to the state of the art, we have tried generating the following sentences, which are examples from (Carroll et al., 1999):

(1) The manager in that office interviewed a new consultant from Germany.

(2) Our manager organized an unusual additional weekly departmental conference.

We have converted the XTAG grammar (XTAG Research Group, 2001) into our grammar format, automatically adding indices to the nodes of the elementary trees, removing features, simplifying adjunction constraints, and adding artificial lexical semantics that consists of the words at the lexical anchors and the indices used in the respective trees. XTAG typically assigns quite a few elementary trees to one lemma, and the same lexical semantics can often be verbalized by more than hundred elementary trees in the converted grammar. It turns out that the dependency parser scales very nicely to this degree of lexical ambiguity: The sentence (1) is generated in 470 milliseconds (as opposed to Carroll et al.'s 1.8 seconds), whereas we generate (2) in about 170 milliseconds (as opposed to 4.3 seconds).[1] Although these numbers are by no means a serious evaluation of our system's performance, they do present a first proof of concept for our approach.

The most encouraging aspect of these results is that despite the increased lexical ambiguity, the parser gets by without ever making any wrong choices, which means that it runs in polynomial time, on all examples we have tried. This is possible because on the one hand, the selection constraint automatically compresses the many different elementary trees that XTAG assigns to one lemma into very few classes. On the other hand, the propagation that rules out impossible edges is so strong that the free input order does not make the configuration problem much harder in practice. Finally, our treatment of modification allows us to multiply out the possible permutations in a postprocessing step, after the parser has done the hard work. A particularly striking example is (2), where the parser gives us a single solution, which multiplies out to $312 = 13 \cdot 4!$ different realizations. (The 13 basic realizations correspond to different syntactic frames for the main verb in the XTAG grammar, e.g. for topicalized or passive constructions.)

5.4 More Complex Semantics

So far, we have only considered TAG grammars in which each elementary tree is assigned a semantics that contains precisely one atom. However, there are cases where an elementary tree either has an empty semantics, or a semantics that contains multiple atoms. The first case can be avoided by exploiting TAG's extended domain of locality, see e.g. (Gardent and Thater, 2001).

The simplest possible way for dealing with the second case is to preprocess the input into several

[1] A newer version of Carroll et al.'s system generates (1) in 420 milliseconds (Copestake, p.c.). Our times were measured on a 700 MHz Pentium-III PC.

different parsing problems. In a first step, we collect all possible instantiations of LTAG lexical entries matching subsets of the semantics. Then we construct all partitions of the input semantics in which each block in the partition is covered by a lexical entry, and build a parsing problem in which each block is one symbol in the input to the parser.

This seems to work quite well in practice, as there are usually not many possible partitions. In the worst case, however, this approach produces an exponential number of parsing problems. Indeed, using a variant of the grammar from Section 3, it is easy to show that the problem of deciding whether there is a partition whose parsing problem can be solved is NP-complete as well. An alternative approach is to push the partitioning process into the parser as well. We expect this will not hurt the runtime all that much, but the exact effect remains to be seen.

6 Comparison to Other Approaches

The perspective on realization that our system takes is quite different from previous approaches. In this section, we relate it to chart generation (Kay, 1996; Carroll et al., 1999) and to another constraint-based approach (Gardent and Thater, 2001).

In chart based approaches to realization, the main idea is to minimize the necessary computation by reusing partial results that have been computed before. In the setting of fixed word order parsing, this brings an immense increase in efficiency. In generation, however, the NP-completeness manifests itself in charts of worst-case exponential size. In addition, it can happen that substructures are built which are not used in the final realization, especially when processing modifications.

By contrast, our system configures nodes into a dependency tree. It solves a search problem, made up by choices for mother-daughter relations in the tree. Propagation, which runs in polynomial time, has access to global information (illustrated in Section 4.2) and can thus rule out impossible mother-daughter relations efficiently; every propagation step that takes place actually contributes to zooming in on the possible realizations. Our system can show exponential runtimes when the distributions span a search tree of exponential size.

Gardent and Thater (2001) also propose a constraint based approach to generation working with a variant of TAG. However, the performance of their system decreases rapidly as the input gets larger even when when working with a toy grammar. The main difference between their approach and ours seems to be that their algorithm tries to construct a *derived* tree, while ours builds a *derivation* tree. Our parser only has to deal with information that is essential to solve the combinatorial problem, and not e.g. with the internal structure of the elementary trees. The reconstruction of the derived tree, which is cheap once the derivation tree has been computed, is delegated to a post-processing step. Working with derived trees, Gardent and Thater (2001) cannot ignore any information and have to keep track of the relationships between nodes at points where they are not relevant.

7 Conclusion

Generation from flat semantics is an NP-complete problem. In this paper, we have first given an alternative proof for this fact, which works even for a fixed grammar and makes the connection to the complexity of free word order parsing clearly visible. Then we have shown how to translate the realization problem of TAG into parsing problems of topological dependency grammar, and argued how the optimizations in the dependency parser – which were originally developed for free word order parsing – help reduce the runtime for the generation system. This reduction shows in passing that the parsing problem for TDG is NP-complete as well, which has been conjectured, but never proved.

The NP-completeness result for the realization problem explains immediately why all existing complete generation algorithms have exponential runtimes in the worst case. As our proof shows, the main sources of the combinatorics are the interaction of lexical ambiguity and tree configuration with the completely unordered nature of the input. Modification is important and deserves careful treatment (and indeed, our system deals very gracefully with it), but it is not as intrinsically important as some of the literature suggests; our proof gets by without modification. If we allow the grammar to be part of the input, we can even modify the proof to show NP-hardness of the case where semantic atoms can be verbalized more often than they appear in the input, and of the case where they can be verbalized less often. The case where every atom can be used arbitrarily often remains open.

By using techniques from constraint programming, the dependency parser seems to cope rather well with the combinatorics of generation. Propagators can rule out impossible local structures on the grounds of global information, and selection constraints greatly alleviate the proliferation of lexical ambiguity in large TAG grammars by making shared information available without having to commit to specific lexical entries. Initial experiments with the XTAG grammar indicate that we can generate practical examples in polynomial time, and may be competitive with state-of-the-art realization systems in terms of raw runtime.

In the future, it will first of all be necessary to lift the restrictions we have placed on the TAG grammar: So far, the nodes of the elementary trees are only equipped with nonterminal labels and indices, not with general feature structures, and we allow only a restricted form of adjunction constraints. It should be possible to either encode these constructions directly in the dependency grammar (which allows user-defined features too), or filter out wrong realizations in a post-processing step. The effect of such extensions on the runtime remains to be seen.

Finally, we expect that despite the general NP-completeness, there are restricted generation problems which can be solved in polynomial time, but still contain all problems that actually arise for natural language. The results of this paper open up a new perspective from which such restrictions can be sought, especially considering that all the natural-language examples we tried are indeed processed in polynomial time. Such a polynomial realization algorithm would be the ideal starting point for algorithms that compute not just any, but the *best* possible realization – a problem which e.g. Bangalore and Rambow (2000) approximate using stochastic methods.

Acknowledgments. We are grateful to Tilman Becker, Chris Brew, Ann Copestake, Ralph Debusmann, Gerald Penn, Stefan Thater, and our reviewers for helpful comments and discussions.

References

Srinivas Bangalore and Owen Rambow. 2000. Using tags, a tree model, and a language model for generation. In *Proc. of the TAG+5 Workshop*, Paris.

G. Edward Barton, Robert C. Berwick, and Eric Sven Ristad. 1987. *Computational Complexity and Natural Language*. MIT Press, Cambridge, Mass.

Chris Brew. 1992. Letting the cat out of the bag: Generation for Shake-and-Bake MT. In *Proceedings of COLING-92*, pages 610–616, Nantes.

John Carroll, Ann Copestake, Dan Flickinger, and Victor Poznanski. 1999. An efficient chart generator for (semi-)lexicalist grammars. In *Proceedings of the 7th European Workshop on NLG*, pages 86–95, Toulouse.

Denys Duchier and Ralph Debusmann. 2001. Topological dependency trees: A constraint-based account of linear precedence. In *Proceedings of the 39th ACL*, Toulouse, France.

Denys Duchier. 2002. Configuration of labeled trees under lexicalized constraints and principles. *Journal of Language and Computation*. To appear.

Claire Gardent and Stefan Thater. 2001. Generating with a grammar based on tree descriptions: A constraint-based approach. In *Proceedings of the 39th ACL*, Toulouse.

Aravind Joshi and Yves Schabes. 1997. Tree-Adjoining Grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, chapter 2, pages 69–123. Springer-Verlag, Berlin.

Martin Kay. 1996. Chart generation. In *Proceedings of the 34th Annual Meeting of the ACL*, pages 200–204, Santa Cruz.

Alexander Koller and Joachim Niehren. 2000. Constraint programming in computational linguistics. To appear in Proceedings of LLC8, CSLI Press.

Oz Development Team. 1999. The Mozart Programming System web pages. `http://www.mozart-oz.org/`.

Yves Schabes and Stuart Shieber. 1994. An alternative conception of tree-adjoining derivation. *Computational Linguistics*, 20(1):91–124.

Matthew Stone and Christy Doran. 1997. Sentence planning as description using tree-adjoining grammar. In *Proceedings of the 35th ACL*, pages 198–205.

XTAG Research Group. 2001. A lexicalized tree adjoining grammar for english. Technical Report IRCS-01-03, IRCS, University of Pennsylvania.