Experiences with Planning for Natural Language Generation

Alexander Koller and **Ronald Petrick**
School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, Scotland, UK
{a.koller,r.petrick}@ed.ac.uk

Abstract

We investigate the application of modern planning techniques to domains arising from problems in natural language generation (NLG). In particular, we consider two novel NLG-inspired planning problems, the sentence generation domain and the GIVE ("Generating Instructions in Virtual Environment") domain, and investigate the efficiency of FF and SGPLAN in these domains. We also compare our results against an ad-hoc implementation of GraphPlan in Java. Our results are mixed. While modern planners are able to quickly solve many moderately-sized instances of our problems, the overall planning time is dominated by the grounding step that these planners perform (rather than search). This has a pronounced effect on our domains which require relatively short plans but have large universes. We share our experiences and offer these domains as challenges for the planning community.

Introduction

Natural language generation (NLG; Reiter and Dale 2000) is one of the major subfields of natural language processing, concerned with computing natural language sentences or texts that convey a given piece of information to the user. Intuitively, this task can be viewed as a problem involving actions, beliefs, and goals: an agent communicating with another agent tries to change the mental state of the hearer by applying actions which correspond to the utterance of words or sentences. This characterisation of NLG suggests obvious parallels to automated planning—a view which has a long tradition in NLG (Appelt 1985; Young and Moore 1994). More recently, there has been a resurgence of interest in applying modern planning techniques to NLG (Steedman and Petrick 2007; Koller and Stone 2007; Benotti 2008). While efficiency has not always been the main focus of NLG, problems in NLG are often more complex than their counterparts in other areas of natural language processing (such as parsing (Koller and Striegnitz 2002)), giving rise to computationally challenging domains. These new approaches combine an interest in planning as a modelling tool for natural language, with a hope of improved efficiency by exploiting modern approaches and algorithms.

The focus of this paper is twofold. We begin by presenting two recent planning domains that arise in the context of natural language generation: the sentence generation domain and the GIVE domain. In the sentence generation task, the goal is to generate a single sentence that expresses a given meaning. Koller and Stone (2007) cast this domain as a planning problem, where the plan encodes the necessary sentence and the actions correspond to uttering individual words. In the GIVE domain ("Generating Instructions in Virtual Environments"), we describe a new shared task that was recently posed as a challenge for the NLG community (Koller et al. 2007). GIVE uses planning as one module of an NLG system that generates natural-language instructions to guide a user performing a given task in a virtual environment.

We then discuss some of our experiences using off-the-shelf planners in our two domains. In particular, we explore the efficiency of FF (Hoffmann and Nebel 2001) and SGPLAN (Hsu et al. 2006)—planners that have been successful on traditional benchmarks, and in the International Planning Competition (IPC)—on a range of problem instances from our planning domains. We also compare these results against an ad-hoc Java implementation of GraphPlan (Blum and Furst 1997). Our findings are mixed. On the one hand, it turns out that modern planners handle some of the search problems that arise in NLG quite easily (although large problem instances still remain a challenge). On the other hand, these same planners spend tremendous amounts of time on preprocessing. For instance, FF spends 90% of its runtime in the sentence generation domain on grounding out literals and actions, and ends up slower than our version of GraphPlan that avoids grounding. For domains like ours, which are dominated by the number of actions and the universe size, rather than the combinatorics of the search problem, this observation suggests that the overall runtime of a planner could be improved by grounding more selectively. We therefore offer these domains as challenges for the planning community.

Planning in NLG

We begin by presenting our two planning domains: the sentence generation domain and the instruction giving domain.

Sentence generation as planning

Sentence generation is the problem of computing, from a grammar and a semantic representation, a single sentence that expresses this piece of meaning. This problem is traditionally split into several steps (Reiter and Dale 2000).

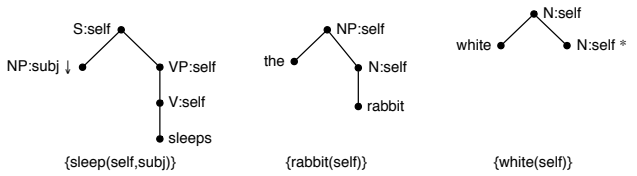

Figure 1: The example grammar.

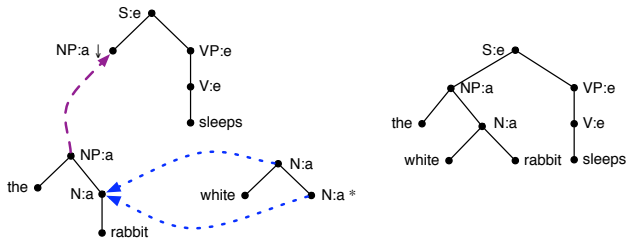

Figure 2: Derivation of "The white rabbit sleeps."

In a first step, called *sentence planning*, the semantic representation is first enriched with more information; for instance, *referring expressions*, which refer to individuals that we want to talk about, are determined at this point. In a second step, called *surface realization*, this enriched representation is then translated into a natural-language sentence, using the grammar.

In practice, the process of determining referring expressions typically interacts with the realization step and so it turns out to be beneficial to perform both of these steps together. This was the goal of the SPUD system (Stone et al. 2003), which performed this task using a top-down generation algorithm based on tree-adjoining grammars (Joshi and Schabes 1997) whose lexical entries were equipped with semantic and pragmatic information. Unfortunately, SPUD suffered from having to explore a huge search space, and had to resort to a non-optimal greedy search strategy to retain reasonable efficiency. To improve on the efficiency of SPUD, Koller and Stone (2007) translate the sentence generation problem into a planning problem and use a planning algorithm for generation.

We illustrate this process on a simplified example. Consider a knowledge base containing the individuals e, r_1 and r_2, and a set of attributes encoding the fact that r_1 and r_2 are rabbits, r_1 is white and r_2 is brown, and e is an event in which r_1 sleeps. Say that we want to express the information $\{\mathsf{sleep}(e, r_1)\}$ using the tree-adjoining grammar shown in Figure 1. This grammar consists of *elementary trees* (i.e., the disjoint trees in the figure), each of which contributes certain *semantic content*. We can instantiate these trees by substituting individuals for *semantic roles*, such as self and subj, and then combine the tree instances as shown in Figure 2 to obtain the sentence "The white rabbit sleeps".

We compute this grammatical derivation in a top-down manner, starting with the elementary tree for "sleeps". This tree satisfies the need to convey the semantic information, but introduces a need to generate a noun phrase (NP) for the subject; this NP must refer uniquely to the target referent r_1. In a second step, we substitute the tree for "the rabbit" into the open NP leaf, which makes the derivation grammatically complete. Since there are two different individuals that could be described as "the rabbit"—technically, r_2 is still a *distractor* (i.e., based on the description "the rabbit", the hearer might erroneously think that we're talking about r_2 and not r_1)—we are still not finished. To complete the derivation, the tree for "white" is added to the existing structure by an *adjunction* operation, making the derivation syntactically and semantically complete.

```
(:action add-sleeps
   :parameters (?u - node
                ?xself - individual
                ?xsubj - individual)
   :precondition (and (subst S ?u)
      (referent ?u ?xself)
      (sleep ?xself ?xsubj))
   :effect (and (not (subst S ?u))
      (expressed sleep ?xself ?xsubj)
      (subst NP (subj ?u))
      (referent (subj ?u) ?xsubj)
      (forall (?y - individual)
         (when (not (= ?y ?xself))
            (distractor (subj ?u) ?y)))))

(:action add-rabbit
   :parameters (?u - node
                ?xself - individual)
   :precondition (and (subst NP ?u)
      (referent ?u ?xself)
      (rabbit ?xself))
   :effect (and (not (subst NP ?u))
      (canadjoin N ?u)
      (forall (?y - individual)
         (when (not (rabbit ?y))
           (not (distractor ?u ?y))))))

(:action add-white
   :parameters (?u - node
                ?xself - individual)
   :precondition (and (canadjoin N ?u)
      (referent ?u ?xself)
      (rabbit ?xself))
   :effect (forall (?y - individual)
      (when (not (white ?y))
         (not (distractor ?u ?y)))))
```

Figure 3: PDDL actions for generating the sentence "The white rabbit sleeps."

The process described above has clear parallels to planning: we manipulate a state by applying actions in order to achieve a goal. We can make this connection even more precise by translating the SPUD problem into a planning problem. For instance, Figure 3 shows the corresponding PDDL actions for the above generation task, where each action corresponds to an operation that adds a single elementary tree to the derivation. In each case, the first parameter of the action is a node name in the derivation tree, and the remaining parameters stand for the individuals to which the semantic roles will be instantiated. The syntactic precon-

ditions and effects are encoded using open and canadjoin literals; the status of each referring expression is tracked using distractor literals. Notice that the action effects contain terms of the form $\mathsf{subj}(u)$, which construct new node names. In order to meet the syntactic requirements of PDDL, these terms can be eliminated by estimating an upper bound n for the plan length, making n copies of each action, ensuring that copy i can only be applied in step i, and replacing the term $\mathsf{subj}(u)$ in an action copy by the constant subj_i.

Once the appropriate actions are defined, we can solve the generation problem as an ordinary planning problem. For instance, the following plan solves our previous example:

1. $\mathsf{sleeps}(\mathsf{root}, r_1)$
2. $\mathsf{rabbit}(\mathsf{subj}(\mathsf{root}), r_1)$
3. $\mathsf{white}(\mathsf{subj}(\mathsf{root}), r_1)$

Using this plan, the grammatical derivation in Figure 2, and therefore the generated sentence, can be systematically reconstructed. Thus, we can solve the sentence generation problem via the detour through planning and bring current search heuristics for planning to bear on generation.

Planning in instruction giving

The object of the GIVE Challenge ("Generating Instructions in Virtual Environments"; Koller et al. 2007) is to build an NLG system which is able to produce natural-language instructions which will guide a human user in performing some task in a virtual environment. From an NLG perspective, GIVE makes for an interesting challenge because it is a theory-neutral task that exercises all components of an NLG system, and emphasizes the study of communication in a (simulated) physical environment. It also has the advantage that the user and the NLG system can be in physically different places, as long as the 3D client and the NLG system are connected over a network. This makes it possible to evaluate GIVE NLG systems on a large scale over the Internet. The first GIVE evaluation will take place in late 2008; currently eight research teams from five countries are working on developing systems to participate in the challenge.[1]

A map of an example GIVE world is shown in Figure 4. In this world, the user's task is to pick up a trophy in the top left room. The trophy is hidden in a safe behind a picture, so the user must first move the picture out of the way and open the safe by pushing a sequence of buttons (the small square boxes on the walls) in the correct order. In order to get to all these buttons, the user must also open the door in the centre of the map and deactivate the alarm tile by pushing further buttons. To simplify both the planning and the NLG task, the world is discretized into tiles of equal size; the user can turn by 90 degree steps in either direction, and can move from the centre of one tile to the centre of the next. Figure 5 shows some of the available actions in PDDL syntax.

The plans generated in the GIVE domain are often non-trivial. For instance, in the example above the shortest plan consists of 108 action steps; the first few steps are as follows:

[1]See the GIVE website for more details about this project: `homepages.inf.ed.ac.uk/v1akolle/proj/give/`.

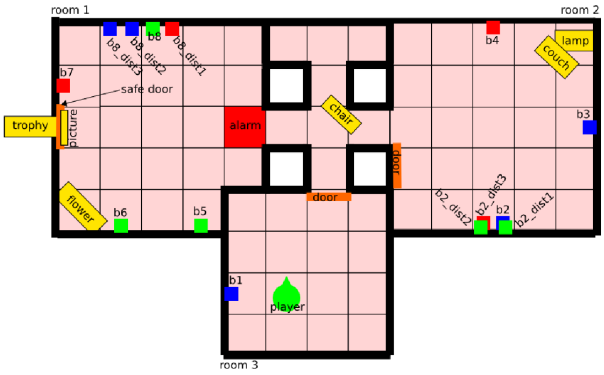

Figure 4: Map of the GIVE development world.

1. $\mathsf{turn\text{-}left}(\mathsf{north}, \mathsf{west})$
2. $\mathsf{move}(\mathsf{pos\_5\_2}, \mathsf{pos\_4\_2}, \mathsf{west})$
3. $\mathsf{manipulate\text{-}b1\text{-}off\text{-}on}(\mathsf{pos\_5\_2})$
4. $\mathsf{turn\text{-}right}(\mathsf{west}, \mathsf{north})$

Most of the actions in this plan are "move" actions. This makes the bulk of the GIVE problem very similar to the Gridworld problem, which also involves finding a route through a world with discrete positions—but with the additional need to press buttons in the right order, reason about many more objects in the world, and navigate somewhat more complicated room shapes.

The GIVE task comes with a number of interesting research challenges both for the NLG and the planning communities. On the NLG side, one major problem is that the natural language instruction sequences can't simply be obtained by verbalising each action instance in a plan in turn. For instance, the NLG system could generate an instruction sequence starting with "turn left; walk forward; press the button; turn right; walk forward; walk forward; turn right; walk forward; walk forward; turn left; walk forward". However, this would be clumsy and boring, and it is to be expected that frustrated users will quickly cancel an interaction with such a system, resulting in a bad evaluation score. Having just pressed the button, it would be much better to say "turn around and walk through the door"; i.e., multiple action instances should be merged into a single instruction. Conversely, it may also be necessary to express a single planning step with several instructions. Having just entered the top left room, it may be easier for the user to understand the instruction "walk to the centre of the room; turn right; now press the green button in front of you" than the instruction "press the green button on the wall to your right". To plan the referring expression "the green button in front of you" at a time when the button is not yet in front of the user, the NLG system must keep track of the hypothetical changes in what objects are visible to the user. Thus the acts of referring and instructing, and the modules for planning and NLG, must be tightly integrated.

From a planning perspective, GIVE imposes very strict runtime requirements on the planner and its associated modules: planning must happen in real time and the system

```
(:action move
   :parameters (?from - position
                ?to - position
                ?ori - orientation)
   :precondition (and (player-pos ?from)
      (adjacent ?from ?to ?ori)
      (player-orient ?ori)
      (not-blocked ?from ?to)
      (not-alarmed ?to))
   :effect (and (not (player-pos ?from))
      (player-pos ?to)))

(:action turn-left
   :parameters (?ori - orientation
                ?newOri - orientation)
   :precondition (and (player-orient ?ori)
      (next-orient-left ?ori ?newOri))
   :effect (and (not (player-orient ?ori))
      (player-orient ?newOri)))

(:action turn-right
   :parameters (?ori - orientation
                ?newOri - orientation)
   :precondition (and (player-orient ?ori)
      (next-orient-right ?ori ?newOri))
   :effect (and (not (player-orient ?ori))
      (player-orient ?newOri)))

(:action manipulate-b1-off-on
   :parameters (?pos - position)
   :precondition (and (state b1 off)
      (player-pos ?pos)
      (position b1 ?pos))
   :effect (and (not (state b1 off))
      (state b1 on)
      (not (state d1 closed))
      (state d1 open)
      (not (blocked pos_6_5 pos_6_4))
      (not (blocked pos_6_4 pos_6_5))))
```

Figure 5: PDDL actions for the GIVE domain.

must respond to a user in a timely fashion. If the system takes too much time deliberating over an instruction to give, rather than actually giving this instruction, the user may have walked or turned away, thus making the instruction invalid. Furthermore, plan execution monitoring also plays an important role in the GIVE problem. At a high level, the system needs to monitor a user's actions and compare them against the generated instruction set to determine if the user has correctly followed directions or not. In the case of the latter, new instructions may have to be generated. In practice, the situation can be quite complicated since the mental state of the user is not known and so the system must observe the user's actions in real time. For example, a user directed to "turn around and walk through the door" may not necessarily perform these actions to the letter, i.e., immediately turning 180 degrees and proceeding directly to the door. Instead, the user might take a roundabout route through the room, eventually exiting out the door. Although the user's actions do not match the generated instructions exactly, they meet the intended goal. The system must be able to identify such "equivalent" plans and not immediately generate new instructions as soon as the user's actions have gone off course. Furthermore, a user can communicate certain intentions to the system, both through action and inaction. For instance, the system should infer that a user has failed to follow instructions if the user exits a room when given a directive to "walk to the centre of the room". The system should also make a similar conclusion if a user simply does nothing when given the instruction.

Experiments

We now describe the results of three simple experiments designed to evaluate the suitability of particular planners (FF, SGPLAN, and an ad-hoc implementation of GraphPlan) in our two NLG domains. As a first impression, our results indicate that planning is a promising tool for both domains. In the GIVE domain, SGPLAN 5.2.2 computes a domain plan from the initial state to the goal in 0.3 seconds, which is fast enough for moderately-sized problems instances in the application.[2] In the sentence generation domain, FF dramatically outperforms the best previously known algorithm for the same problem (a reimplementation of (Stone et al. 2003)), although the latter is a greedy search algorithm with a heuristic that is hand-tailored to the domain.

We also observe that although FF manages the search for a plan very efficiently, it spends comparatively large amounts of time computing instantiations of predicates and actions, most of which are then never used during the search. As a result, FF's "grounding time" dominates its overall planning time, leading to some conflicting results. Our experiments below seek to improve our understanding of this situation.

Experiment 1: Sentence generation

In the first experiment, we generate a series of sentence generation problems which require the planner to compute a plan representing the sentence "Mary likes the $\text{Adj}_1 \ldots \text{Adj}_n$ rabbit." Each problem instance assumes a certain number m of rabbits that were distinguished by $n \leq m$ different properties, such that all n properties are required to distinguish the target referent from all other rabbits. The n properties are realized as n different adjectives, in any order. This setup allows us to control the plan length (a plan with n properties will have length $n + 4$) and the universe size (the universe will contain $m + 1$ individuals in addition to the differently-typed individuals used to encode the grammar).

The results of this experiment are shown in Figure 6. The input parameters (m, n) are plotted in lexicographic order on the horizontal axis and the runtime is shown in milliseconds on the vertical axis. These results reveal a number of interesting insights. First, FF significantly outperforms SGPLAN in this domain.[3] Second, FF's runtime is dominated

[2] All runtimes were measured on a Pentium 4 CPU running at 3 GHz. Java programs were allowed to "warm up", i.e. the planner was run five times and the first four measurements discarded to ensure that the JVM had just-in-time compiled all relevant bytecode.

[3] Experiments with SGPLAN use a pre-release version of SGPLAN, kindly provided by Chih-Wei Hsu. The release version of

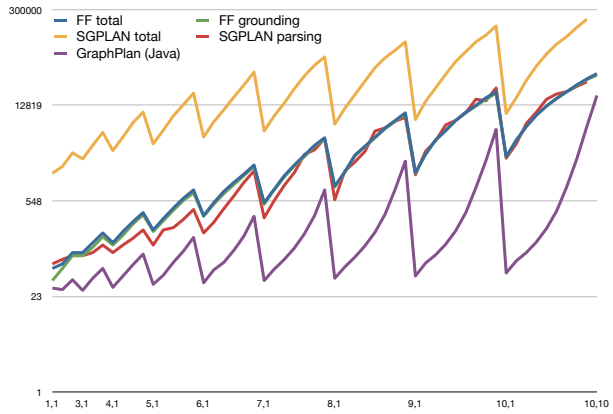

Figure 6: Runtimes (in ms) in the sentence generation domain. The horizontal axis represents parameters (m, n) from $(1, 1)$ to $(10, 10)$ in lexicographical order.

by its initial grounding step, in which it computes the ground instances of all predicates and actions used in the planning problem to avoid unnecessary instantiations during search. In particular, the ratio of grounding time to total runtime is generally above 85%, and rises to above 99% at $m = 11$, which is still a small universe in this application.[4]

In fact, the time spent by FF on grounding is such that it is consistently outperformed by an ad-hoc Java implementation of GraphPlan which only computes instances of predicates and actions as they are discovered (i.e., while the planning graph is being built)—despite the fact that FF is consistently much faster as far as pure search time is concerned. Looking at the difference from another angle, FF's performance is much more sensitive to the domain size: if we fix $n = 1$, FF takes 60 ms to compute a plan at $m = 1$, but 2.4 seconds (for the same plan) at $m = 10$; our GraphPlan implementation takes 30 ms at $m = 1$ and still only requires 50 ms at $m = 10$. Conversely, GraphPlan's runtime grows much faster with the plan size (i.e., with growing values of n for a fixed m). Larger (but still realistically-sized) instances of the sentence generation problem are still problematic for the planners we tested.

Experiment 2: Minimal GIVE worlds

In the second experiment, we evaluate the performance of the planners on problems arising in the GIVE domain. We construct a series of test worlds, similar to the one illustrated in Figure 7. These worlds consist of a $2n$ by h grid of positions, such that there are buttons at positions $(2i - 1, 1)$ and $(2i, h)$ for $1 \leq i \leq n$. The player starts in position $(1, 1)$ and must press all the buttons in order to successfully complete the game. The world is generated as a GIVE world description, and then automatically converted into a planning problem (as in Figure 5) by the GIVE software.

SGPLAN 5.2.2 had a bug, causing it to crash on some instances.

[4]The "grounding" time reported here is what FF reports as "time spent: instantiating action templates".

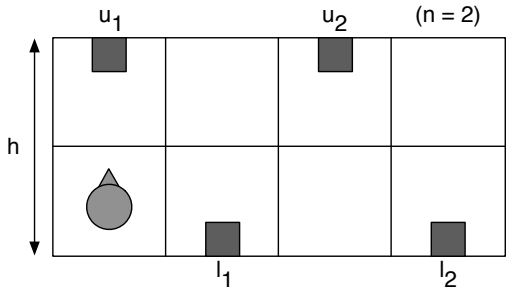

Figure 7: Minimal GIVE world.

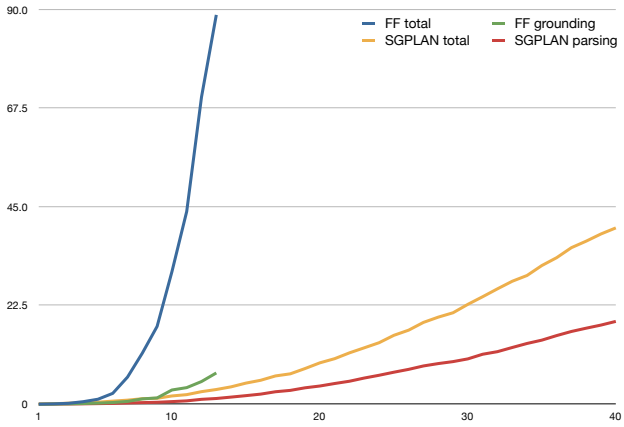

Figure 8: Runtimes (in seconds) of FF and SGPLAN on the minimal GIVE worlds for $h = 20$. The horizontal axis is n.

Results for the $h = 20$ case, with n ranging from 1 to 40, are shown in Figure 8. The most obvious result is that FF is unable to solve any problems beyond $n = 13$ on our experimentation machine within the memory limit of 1 GB. SGPLAN, on the other hand, solves instances beyond $n = 40$ without major problems. The time spent on grounding is not a major factor in either planner, probably because the planners need more time to actually compute the plan—for instance, the optimal plan for the problem $n = 40$ has a length of about 1600 steps.

Experiment 3: GIVE worlds with extra positions

In the final experiment, we vary the structure of the GIVE world in order to judge the effect of universe size on the planning problem in this domain. Starting with the ordinary GIVE world described in Experiment 2, we add another w by h empty "junk" positions to the right of the minimal world (see Figure 9). These new positions are not actually needed in any plan, but approximate the situation in the actual GIVE domain, where most grid positions are never used. We leave the initial state and goal untouched. As before, we generate a GIVE world description and then convert it into a planning problem.

Results for the $h = 20$, $n = 5$ case (with w ranging from 1 to 70) are shown in Figure 10. As in Experiment 2, FF

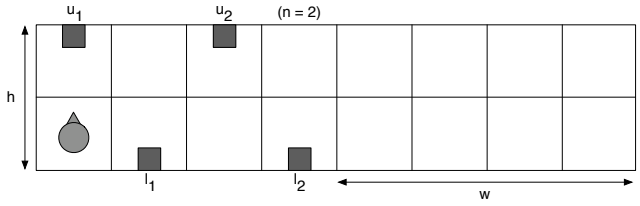

Figure 9: GIVE world with extra "junk" positions.

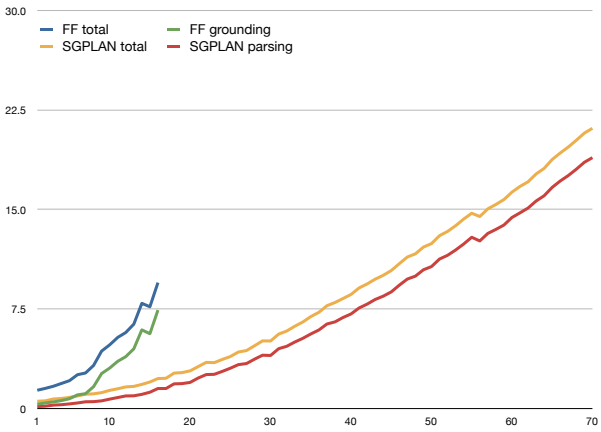

Figure 10: Runtimes (in seconds) of FF and SGPLAN on the GIVE worlds with junk positions for $h = 20$ and $n = 5$. The horizontal axis is w.

again runs out of memory, this time at $w = 17$. SGPLAN happily solves inputs beyond $w = 70$. However, unlike in Experiment 2, both planners now spend a substantial proportion of their time on grounding. In SGPLAN, this translates to a "parsing time" (which we assume includes grounding) which grows from 180 ms to 21.7 seconds as w grows from 1 to 75. The rest of the runtime (which also includes the search time) only grows from 400 ms to 2.3 seconds. This difference is particularly dramatic given that the actual optimal plan in each case is an identical plan of about 100 steps. The planning times for these instances are also concerning since times over a couple seconds negatively affect the overall response time of the system, which must react in real time to user actions.

Discussion

The positive conclusion we can draw from the simple experiments reported above is that modern planners are fairly good at controlling the search for many of the moderately-sized NLG problems we looked at. This is particularly true for FF in the sentence generation domain, which generates nontrivial 14-word sentences in about two seconds. There is still room for improvement, however; special-purpose algorithms generate much larger referring expressions in milliseconds (Areces, Koller, and Striegnitz 2008). Although search efficiency is not the main concern in these domains, it nevertheless becomes a factor in scenarios like Experiment 2, where the world is almost completely unconstrained and (unsurprisingly) the search becomes much slower.

The most restrictive bottleneck in the two domains we investigated is the initial grounding step that many modern planners perform. While this may be a good strategy for traditional planning benchmarks and IPC domains—where plan length often dominates universe size, and an initial investment into grounding pays off in saved instantiations during the search—this approach is less effective in the domains we've considered. As our experiments have shown, there are natural planning domains in which relatively short plans must be computed in large universes. For instance, in the case of the sentence generation domain, it is not unrealistic for the universe to consist of thousands of domain individuals and tens of thousands of actions, some of which require three domain individuals as parameters. The common strategy of splitting the universe over several types of individuals will not be very effective here.

The recent trend in planning research has (rightfully) focused on the development of algorithms that control search in sophisticated ways, resulting in a host of planners that are more powerful and more successful than their predecessors. However, the common strategy of grounding out sets of predicates and actions in implementations of these algorithms has a very pronounced effect on domains such as those described above. Since our domains are not that unusual in their structure and composition, we hope that the lessons learnt from our experiences can help improve the performance of current systems. We believe that such improvements are necessary if planning is to become a more mature technology that can offer tools to a wider community of users. At the end of the day, real-world users will care about the *total* runtime of a planner, and this is more than just the search time.

By and large, our experiences with the planning community from the point of view of a "customer" (one of the authors is not a planning researcher) have been relatively pleasant. Thanks to the planning competitions, it is easy to identify and download a fast implementation, at least for Linux. However, in the course of our experiments we found (and reported) bugs in both SGPLAN and FF. We also discovered that deciding between the range of available planners is not always straightforward. As our experiments have shown, even planners as closely related as FF and SGPLAN can differ significantly in their performance on different domains.

Conclusion

In this paper, we introduced two novel planning domains arising from problems in natural language generation: the sentence generation domain and the GIVE navigation domain. We also reported on the results of a number of experiments in which we applied off-the-shelf planners to these domains. Our main results were mixed. While modern planners do a pretty good job of controlling the complexity of search, they also suffer from practical problems that limit their performance in unexpected ways. In particular, in both domains the grounding step performed by both planners dominates the time it takes to perform the search itself. The NLG community's recent interest in planning also

presents a valuable opportunity for planning researchers: provided some of the challenges we have highlighted can be addressed, projects like GIVE (and other NLG-inspired problem domains) offer a constructive platform for the planning community to showcase their techniques to a wider audience—and to improve the quality of their tools for real-world planning tasks.

References

Appelt, D. 1985. *Planning English Sentences*. Cambridge England: Cambridge University Press.

Areces, C.; Koller, A.; and Striegnitz, K. 2008. Referring expressions as formulas of description logic. In *Proceedings of the 5th International Natural Language Generation Conference*.

Benotti, L. 2008. Accommodation through tacit sensing. In *LONDIAL 2007 Workshop on the Semantics and Pragmatics of Dialogue*.

Blum, A., and Furst, M. 1997. Fast planning through graph analysis. *Artificial Intelligence* 90.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14.

Hsu, C. W.; Wah, B. W.; Huang, R.; and Chen, Y. X. 2006. New features in SGPlan for handling soft constraints and goal preferences in pddl 3.0. In *Proc. Fifth International Planning Competition, ICAPS-2006*.

Joshi, A., and Schabes, Y. 1997. Tree-Adjoining Grammars. In Rozenberg, G., and Salomaa, A., eds., *Handbook of Formal Languages*. Berlin: Springer-Verlag. chapter 2.

Koller, A., and Stone, M. 2007. Sentence generation as planning. In *Proc. of the 45th ACL*.

Koller, A., and Striegnitz, K. 2002. Generation as dependency parsing. In *Proc. 40th ACL*.

Koller, A.; Moore, J.; di Eugenio, B.; Lester, J.; Stoia, L.; Byron, D.; Oberlander, J.; and Striegnitz, K. 2007. Shared task proposal: Instruction giving in virtual worlds. In White, M., and Dale, R., eds., *Working group reports of the Workshop on Shared Tasks and Comparative Evaluation in Natural Language Generation*.

Reiter, E., and Dale, R. 2000. *Building Natural Language Generation Systems*. Cambridge University Press.

Steedman, M., and Petrick, R. P. A. 2007. Planning dialog actions. In *Proceedings of the 8th SIGdial Workshop on Discourse and Dialogue (SIGdial 2007)*, 265–272.

Stone, M.; Doran, C.; Webber, B.; Bleam, T.; and Palmer, M. 2003. Microplanning with communicative intentions: The SPUD system. *Computational Intelligence* 19(4).

Young, R. M., and Moore, J. D. 1994. DPOCL: a principled approach to discourse planning. In *Proceedings of the Seventh International Workshop on Natural Language Generation*.