A Decision-Theoretic Approach to Natural Language Generation

Paper ID: 537

Abstract

We study the problem of generating an English sentence given an underlying probabilistic grammar, a vocabulary and a communicative goal. We model the generation problem as a Markov decision process with a reward function that reflects the communicative goal. We then use probabilistic planning to solve the MDP and generate a sentence that, with high probability, satisfies the communicative goal. We show empirically that our approach can generate complex sentences with a speed that generally matches or surpasses the state of the art. Further, we show that our approach can handle large grammars and complex communicative goals, including negated goals.

Introduction

"Give me a sentence about a cat and a dog." Given such a *communicative goal*, most people can answer a question like this one very quickly. Further, they can easily provide multiple similar sentences, differing in details but all satisfying the general communicative goal, with no or very little error. Natural language generation (NLG) develops techniques to extend similar capabilities to automated systems. In this paper, we study the following restricted NLG problem: given a grammar, lexicon, and a communicative goal, output a valid English sentence that satisfies this goal.

This NLG problem is deceptively simple. A key source of difficulty is the nature of the grammar, which is generally large, probabilistic and ambiguous. Some NLG techniques use sampling strategies [Knight and Hatzivassiloglou, 1995] where a set of sentences is sampled from a data structure created from an underlying grammar and ranked according to how well they meet the communicative goal. Variations of this idea have explored data structures for efficient sampling (e.g. [Langkilde-Geary, 2002]). Such approaches naturally handle statistical grammars, but do not solve the generation problem in a goal-directed manner.

An alternative approach to the NLG problem is to view the task as a *planning* problem [Koller and Stone, 2007]. Here, the communicative goal is treated as a predicate to be satisfied, and the grammar and vocabulary are suitably encoded as logical operators. Then automated planning techniques are used to derive a plan. In some cases, the plan describes the sentence structure, with the *realization* of the actual sentence in a following step [Reiter and Dale, 2000]. More recent work [Koller and Petrick, 2011] has argued that realization should be incorporated into the planning step; in *integrated generation*, the solution plan contains the sentence to be output.

While the above approach is a nice formalization of NLG, restrictions on what current planning techniques can do limit its applicability. A key limitation is the logical nature of automated planning systems, which do not handle probabilistic grammars, or force ad-hoc approaches for doing so [Bauer and Koller, 2010]. A second limitation comes from restrictions on the goal: it may be difficult to ensure that some specific piece of information should *not* be communicated, or to specify preferences over communicative goals, or specify general conditions, like that the sentence should be readable by a sixth grader. A third limitation comes from the search process: without strong heuristics, most planners get bogged down when given communicative goals that require chaining together long sequences of operators [Koller and Petrick, 2011].

In our work, we also view NLG as a planning problem. However, we differ in that our underlying formalism for NLG is a *Markov decision process* (MDP). This setting allows us to address the limitations outlined above: it is naturally probabilistic, and handles probabilistic grammars; we are able to specify complex communicative goals and general criteria through a suitably-defined reward function; and, as we show in our experiments, recent developments in fast planning in large MDPs result in a generation system that can rapidly deal with very specific communicative goals even in large grammars. Further, our system has several other desirable properties: it is an anytime approach; with a probabilistic grammar, it can naturally be used to sample and generate multiple sentences satisfying the communicative goal; and it is parallelizable. Finally, the decision-theoretic setting allows for a precise tradeoff between exploration of the grammar and vocabulary to find a better solution and exploitation of the current most promising (partial) solution, instead of a heuristic search through the solution space as performed by standard planning approaches.

Related Work

Two broad lines of approaches have been used to attack the general NLG problem. One direction can be though of as "overgeneration and ranking." Here some (possibly probabilistic) structure is used to generate multiple candidate sentences, which are then ranked according to how well they satisfy the generation criteria. This includes work based on chart generation and parsing [Shieber, 1988; Kay, 1996]. These generators assign semantic meaning to each individual token, then use a set of rules to decide if two words can be combined. Any combination which contains a semantic representation equivalent to the input at the conclusion of the algorithm is a valid output from a chart generation system. Another example of this idea are the HALogen/Nitrogen systems [Langkilde-Geary, 2002]. HALogen uses a two-phase architecture where first, a "forest" data structure that compactly summarizes possible expressions is constructed.The structure allows for a more efficient and compact representation compared to lattice structures that had been previously used in statistical sentence generation approaches. Using dynamic programming, the highest ranked sentence from this structure is then output. Many other systems using similar ideas exist, e.g. [White and Baldridge, 2003; Lu, Ng, and Lee, 2009].

A second line of attack formalizes NLG as an AI planning problem. SPUD [Stone et al., 2003], a system for NLG through microplanning, considers NLG as a problem which requires realizing a deliberative process of goal-directed activity. Many such NLG-as-planning systems use a pipeline architecture, working from their communicative goal through a series of processing steps and concluding by outputting the final sentence in the desired natural language. This is usually done into two parts: discourse planning and sentence generation. In discourse planning, information to be conveyed is selected and split into sentence-sized chunks. These sentence-sized chunks are then sent to a *sentence generator*, which itself is usually split into two tasks, *sentence planning* and *surface realization* [Koller and Petrick, 2011]. The sentence planner takes in a sentence-sized chunk of information to be conveyed and enriches it in some way. This output is then used by a *surface realization* module which encodes the enriched semantic representation into the desired natural language. This chain is sometimes referred to as the "NLG Pipeline" [Reiter and Dale, 2000].

Another approach, called *integrated generation*, considers both sentence generation portions of the pipeline together. [Koller and Stone, 2007]. This is the approach taken in some modern generators like CRISP [Koller and Stone, 2007] and PCRISP [Bauer and Koller, 2010]. In these generators, the input semantic requirements and grammar are encoded in PDDL [Fox and Long, 2003], which an off-the-shelf planner such as Graphplan [Blum and Furst, 1997] uses to produce a list of applications of rules in the grammar. These generators generate parses for the sentence at the same time as the sentence, which keeps them from generating realizations that are grammatically incorrect, and keeps them from generating grammatical structures that cannot be realized properly. PCRISP extends CRISP by adding support for probabilistic grammars. However the planner in PCRISP's back end is still a standard PDDL planner, so PCRISP transforms the probabilities into costs so that a low likelihood transition has a high cost in terms of the plan metric.

In the NLG-as-planning framework, the choice of grammar representation is crucial in treating NLG as a planning problem; the grammar provides the actions that the planner will use to generate a sentence. Tree Adjoining Grammars are a common choice [Koller and Stone, 2007] [Bauer and Koller, 2010]. TAGs are tree-based grammars consisting of two sets of trees, called initial trees and auxiliary or adjoining trees. An entire initial tree can replace a leaf node in the sentence tree whose label matches the label of the root of the initial tree in a process called "substitution." Auxiliary trees, on the other hand, encode recursive structures of language. Auxiliary trees have, at a minimum, a root node and a foot node whose labels match. The foot node must be a leaf of the auxiliary tree. These trees are used in a three-step process called "adjoining". In the first step, we find an adjoining location. We do this by searching through our sentence to find any subtree with a root whose label matches the root node of the auxiliary tree. In the second step, we removed our target subtree from the sentence tree, and place it in the auxiliary tree as a direct replacement for the foot node. Finally, the modified auxiliary tree is placed back in the sentence tree in the original target location.

Though the NLG-as-planning approaches are elegant and appealing, a key drawback is the difficulty of handling probabilistic grammars, which are readily handled by the overgeneration and ranking strategies. Recent approaches such as PCRISP attempt to remedy this, but do so in a somewhat ad-hoc way, because they rely on deterministic planning to actually realize the output. In this work, we directly confront this by switching to a more expressive underlying formalism, a Markov decision process (MDP). We show in our experiments that this modification has other benefits as well, such as being anytime and an ability to handle complex communicative goals beyond those that deterministic planners can handle.

We note that though the application of MDPs to NLG appear not to have been explored, some preliminary work has explored the application of MDPs and the UCT algorithm [Chevelu et al., 2009] that we also use in our work to paraphrasing. Here the algorithm was used to search through a paraphrase table to find the best paraphrase solution.

The STRUCT System

We formulate NLG as a planning problem on a Markov decision process (MDP) [Puterman, 1994]. An MDP is a tuple (S, A, T, R, γ) where S is a set of states, A is a set of actions available to an agent, $T : S \times A \times S \rightarrow (0, 1)$ is a possibly stochastic function defining the probability $T(s, a, s')$ with which the environment transitions to s' when the agent does a in state s. $R : S \times A \rightarrow \mathbb{R}$ is a real-valued reward function that specifies the utility of performing action a in state s. Finally, γ is a discount factor that allows planning over infinite horizons to converge. In such an MDP, the agent selects actions at each state (a *policy*) to optimize the expected long-term discounted reward: $\pi^*(s) =$

$\arg\max_a E(\sum_t \gamma^t R(s_t, a_t)|s = s_0)$, where the expectation is taken with respect to the state transition distribution. When the MDP model (T and R) is known, various dynamic programming algorithms such as value iteration [Bellman, 1957] can be used to plan and act in an MDP. When the model is unknown, and the task is to formulate a policy, it can be solved in a model-free way (i.e. without estimating T and R) through temporal difference (TD) learning. The key idea in TD-learning is to take advantage of Monte Carlo sampling; since the agent visits states and transitions with a frequency governed by the unknown underlying T, simply keeping track of average rewards over time yields the expected values required to compute the optimal actions at each state.

Determining the optimal policy at *every* state using the above strategy is polynomial in the size of the state-action space [Brafman and Tennenholtz, 2003]. However, suppose we only want to plan in the MDP to go from a specific initial state to a goal state. Is it possible to do this without exploring the entire state-action space? Surprisingly, recent work answers this question affirmatively. New techniques such as sparse sampling [Kearns, Mansour, and Ng, 1999] and UCT [Kocsis and Szepesvari, 2006] show how to generate near-optimal plans in large MDPs with a time complexity that is independent of the state space size. We use a variation of the UCT algorithm in our system; we describe it here.

Online planning in MDPs generally follows two steps. From each state encountered, a lookahead tree is constructed and used to estimate the utility of each action in this state. Then, the best action is taken, the system transitions to the next state and the procedure is repeated. In order to build a lookahead tree, a "rollout policy" is used. This policy has two components: if it encounters a state already in the tree, it follows a "tree policy," discussed further below. If it encounters a new state, the policy reverts to a "default" policy that typically randomly samples an action. In all cases, any rewards received during the rollout search are backed up. Because this is a Monte Carlo estimate, typically, several simultaneous trials are run, and we keep track of the rewards received by each choice and use this to select the best action at the root.

The final detail we need to specify is how the tree policy is determined. In UCT, this is determined by choosing the action a in state s that maximizes:

$$P(s,a) = Q(s,a) + c\sqrt{\frac{lnN(s)}{N(s,a)}} \quad (1)$$

Here $Q(s,a)$ is the estimated value of a as observed in the tree search and $N(s)$ and $N(s,a)$ are visit counts for the state and state-action pair. Thus the second term is an exploration term that biases the algorithm towards visiting actions that have not been explored enough. c is a constant that trades off exploration and exploitation. This formulation essentially treats each action decision as a bandit problem; previous work shows that this approach can efficiently select near-optimal actions at each state.

We now describe our approach, that we call STRUCT (Sentence Tree Realization with UCT). The states of the underlying MDP contain *partial sentences.* The actions available to the algorithm consist of productions in a lexicalized TAG (LTAG) or probabilistic lexicalized TAG (PLTAG). We chose LTAGs due to their power and their common usage in NLG scenarios. Since trees are associated with individual words, they can be interpreted as adding a specific semantic meaning to the overall sentence while describing precisely the syntactic environment in which these words can occur, and can even define any arguments for the word in question, for instance, the location of the subject and object of a verb, or the presence of a required third argument. Further, since all recursive phenomena are encoded in auxiliary trees, it factors recursion from the domain of dependencies [Bauer, 2009], and can add auxiliary trees to partial sentences without breaking dependency links between nodes. The transition distribution in the underlying MDP is implicitly defined by the probabilities associated with a PLTAG, or uniform in the case of a standard LTAG.

Clearly, the action set can get very large for a large grammar or for a long sentence with many locations for adjoining. Still, this is the only way to ensure that we can generate all possible grammatical sentences. UCT deals very well with this large action set due to the pruning inherent in its iterated Monte Carlo sampling method. We can further compensate for this large action set by increasing the number of samples, if necessary. It should also be noted that most combinations of these actions are order-independent; for instance, two actions, each adjoining an adjective to the subject and object of the sentence, respectively. It is also notable that, occasionally, the order of some words in a sentence is not important. "A small white teapot" and "a white small teapot" convey the same semantic information despite the ordering of their adjectives.

When STRUCT uses Tree Adjoining Grammars, we require that all substitutions be performed first, before any adjoins can be done. We do this is in order to generate a complete and valid sentence quickly. After this point, any time that there are no substitutions available (all nonterminals have at least one child), we record the current sentence. If generation is interrupted, we return the most recent complete and valid sentence. In this way, our approach functions as an anytime algorithm.

A key advantage of STRUCT is our ability to specify a detailed reward function that captures complex communicative goals. The reward function depends on the current state and action and captures the "goodness" of the current partial sentence we have produced. Thus we can guide the planning algorithm at any granularity we choose. For example, we can give "progress" rewards for partial sentences that achieve some parts of the communicative goal; we can penalize the algorithm if it attempts to generate a sentence that communicates something we wish *not* to communicate; we can trade off the importance of multiple communicative goals in the same sentence; we can even set up the reward function to prefer global criteria such as "readability" at a specific grade level, if we so choose. Of course, once the sentence achieves the complete communicative goal, a large reward is given, so the algorithm returns this as the solution. Since we use finite depth lookahead and wish to propagate long range rewards

Algorithm 1 The STRUCT Algorithm.

Require: Number of simulations $numTrials$, Depth of lookahead $maxDepth$
Ensure: Generated sentence tree
$state \leftarrow$ empty sentence tree
while $state$ not terminal **do**
  for $numTrials$ **do**
    $testState \leftarrow state$
    $currentDepth \leftarrow 0$
    if $testState$ has unexplored actions **then**
      Apply one unexplored PLTAG production chosen uniformly at random to $testState$
      $currentDepth$++
    end if
    while $currentDepth < maxDepth$ **do**
      Apply PLTAG production selected by tree policy (Equation 1)
      $currentDepth$++
    end while
    calculate reward for $testState$
    associate reward with first action taken
  end for
  $state \leftarrow$ maximum reward $testState$
end while
return state

to the root, we use a discount factor of 1.

With the definition above, we use UCT to find a solution sentence. We modify the standard algorithm in two ways. First, in the action selection step, we select the action that leads to the best $P(s, a)$ over all simulations rather than the best *average* $P(s, a)$. We do this because the original formulation of UCT is designed to work in adversarial situations, in such cases, selecting the absolute best may be risky if the opponent can respond with something that can also lead to a very bad result. In our case, however, there is no opponent, so we can freely choose the action leading to best overall reward. Second, we parallelize the planning algorithm to run more quickly. Since UCT relies on simulations, it is very simple to parallelize: we simply fork each simulation into its own thread, and compute the best action once all simulations return. We implemented the system in Python 2. The pseudocode is shown in Algorithm 1.

Empirical Evaluation

In this section, we compare STRUCT to a state-of-the-art NLG system, CRISP, and evaluate three hypotheses: (i) STRUCT will be comparable in speed and generation quality to CRISP as it generates increasingly large referring expressions, (ii) STRUCT will be comparable in speed and generation quality to CRISP as the size of the grammar which they use increases, and (iii) STRUCT is capable of solving complex communicative goals, including multiple concurrent goals and negated goals. Finally, we evaluate the effect of varying key parameters on STRUCT's performance.

Comparison to CRISP

In order to maximize compatibility with the CRISP generation system (Koller 2007), we read from two data files which are used by the CRISP system: a grammar file and a world file. The grammar file contains a list of elementary trees and a lexicon. The elementary trees may be either initial or adjoining trees, and have semantic roles of arguments clearly marked, while the lexicon entries reference these elementary trees and lexicalize them. These lexicon entries also refer to the semantic meanings and semantic requirements of the trees which they represent. All this information is encoded in our grammar (for these experiments, we use a deterministic grammar, as CRISP does). World files also contain a list of communicative goals, which are encoded by a reward function in STRUCT. Our reward function gives 1000 points for using the correct communicative goal, an additional 300 points for a correct and unambiguous reference to an argument of that communicative goal or an additional $\frac{150}{n}$ points for a correct but ambiguous reference, where n is the number of other entities that are possible referents for the argument.

In these experiments, the reward signal is fine-grained, and we are able to inform the algorithm about progress to the end goal. As a result, we found that a myopic action selection strategy is sufficient for STRUCT, and the d parameter can be set to zero. We set the number of simulations to $|\mathbf{A}|$, where $\mathbf{A}$ is the set of actions available at each state. The exploration constant c in Equation 1 was set to 0.5.

Comparison to CRISP: Referring Expressions We first evaluate CRISP and STRUCT on their ability to generate referring expressions. This experiment is the same as in prior work [Koller and Petrick, 2011]. We consider a series of sentence generation problems which require the planner to generate a sentence equivalent to "Mary likes the Adj_1 Adj_2 ... Adj_n rabbit", where the string of adjectives is a string that distinguishes one rabbit (whose identity is specified in the problem description) from all other entities in the world.

We used a 2010 version of CRISP which uses a Java-based GraphPlan implementation. The experiment has two parameters: m, the number of adjectives in the grammar, and n, the number of adjectives necessary to distinguish the entity in question from all other entities. We ran experiments for m=30, and n ranging from 1 to 15. The results are shown in Figure 1 (a).

From the figure, we observe that CRISP was able to achieve sub-second times for all expressions of less than length 5, but its generation times increase exponentially past that point, exceeding 100 seconds for some plans at length 10. At length 15, CRISP failed to generate a referring expression; after 90 minutes the Java garbage collector terminated the process. STRUCT, though initially just a little slower than CRISP, scales much better: it was able to generate referring expressions of length 25 without failing.

The results of this experiment also demonstrate the anytime nature of STRUCT, as we show in figures 1 (b) and (c). Here we look at the score of the solution generated as a function of time, as measured by our reward function, for $n = 14$, the largest scenario CRISP is able to solve. As expected, CRISP produces nothing until the end, at which

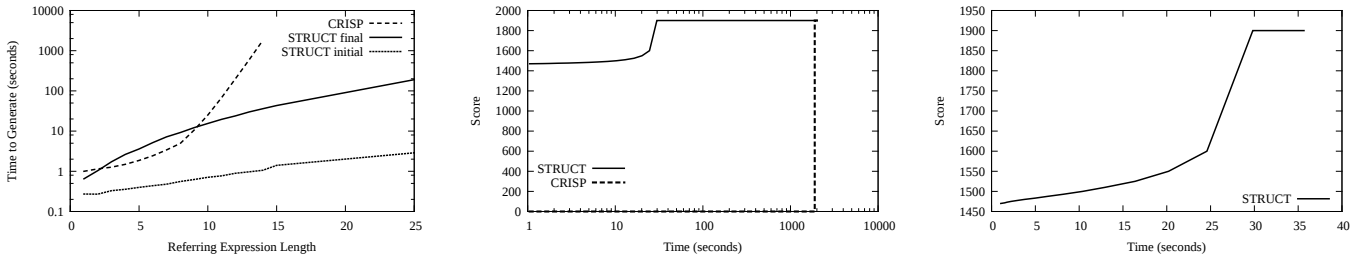

Figure 1: a) Generation of referring expressions of increasing length increases time to generation significantly. CRISP is able to generate expressions of length up to 14; STRUCT can continue further. b) STRUCT and CRISP generate 14-word referring expressions; CRISP completes generation all at once, while STRUCT has a response ready quickly and improves it until it is optimal.c) A closer look at STRUCT's improvement over time.

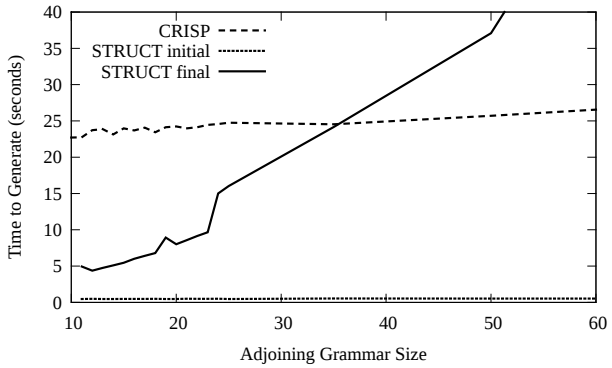

Figure 2: As the number of possible adjoining trees increases, STRUCT's generation of its final result slows. CRISP also slows, but less severely than STRUCT. STRUCT's first anytime result is available very quickly regardless of the size of the grammar.

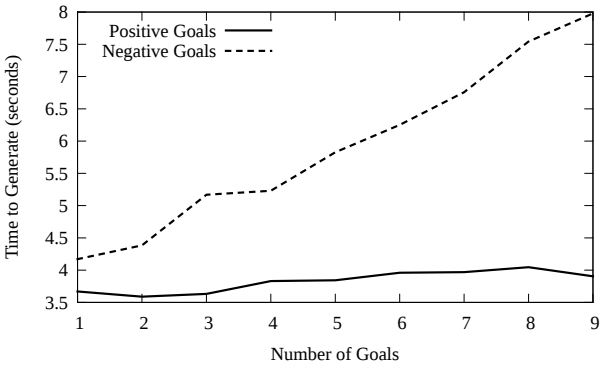

Figure 3: As additional positive goals are added to STRUCT, runtime does not increase substantially when all goals are satisfiable. As additional negated goals are added, so are additional grammar elements. The increase in time seen here as additional negated goals are added can be traced directly to those additional grammar elements.

point it returns the solution. STRUCT, however, quickly produces a reasonable solution, "Mary likes the rabbit." This is then improved upon by adjoining until the referring expression is unambiguous. If at any point the generation process was interrupted, STRUCT would be able to return a solution that at least partially solves the communicative goal.

Comparison to CRISP: Grammar Size We next evaluate STRUCT and CRISP's ability to handle larger grammars. This experiment is set up in the same way as the one above, but here, instead of keeping the grammar size m fixed and varying the referring expressions n, we keep the number of referring expressions fixed ($n = 10$) and vary the size of the grammar ($m = 10$ to 60). In these experiments, there are between 0 and 50 adjoining trees in the grammar which do not in any way help to accomplish the communicative goal. The results of this experiment are shown in Figure 2.

From this figure, we see that the CRISP GraphPlan system, as originally reported in [Koller and Petrick, 2011], handles an increase in number of unused actions very well. Prior work reported a difference on the order of single milliseconds moving from $m = 1$ to $m = 10$. We report similar variations in CRISP runtime as m increases from 10 to 60: runtime increases by approximately 10% over that range.

STRUCT's performance with large grammars is much more similar to the FF planner [Hoffmann and Nebel, 2001], also profiled in [Koller and Petrick, 2011], which increased from 27 ms to 4.4 seconds over the interval from $m = 1$ to $m = 10$. STRUCT's performance is less sensitive to larger grammars than this, but over the same interval where CRISP increases from 22 seconds of runtime to 27 seconds of runtime, STRUCT increases from 4 seconds to 100 seconds. Fortunately, as STRUCT is an anytime algorithm, valid sentences are available as early as 0.46 seconds, despite the size of the set of adjoining trees (the "STRUCT Initial" line in Figure 2. This value does not change substantially with increases in grammar size. However, the time to improve this solution does. An interesting question for future work is how to limit this increase in time complexity in STRUCT.

Complex Goals

In the next set of experiments, we illustrate that STRUCT can solve conjunctions of communicative goals as well as negated communicative goals.

Multiple Goals We next evaluate STRUCT's ability to accomplish multiple communicative goals by generating a single sentence. In this experiment, we deviate slightly from the problem proposed in the previous section. In that section, the referred-to rabbit was unique, and it was therefore possible to produce a referring expression which identified it unambiguously. In this experiment, we remove this condition by creating a situation in which the generator will be forced to ambiguously refer to several rabbits. We then add to the world a number of adjectives which are common to each of these possible referents. Since these adjectives do not further disambiguate their subject, our generator should not choose to use them in its output. We then encode these adjectives into communicative goals, so that they will be included in the output of the generator despite not assisting in the accomplishment of disambiguation.

As we show in figure 3, the presence of additional satisfiable semantic goals does not substantially affect the time required for generation. We are able to accomplish this task with the same very high frequency as the CRISP comparisons, as we use the same parameters.

Negated Goals We now evaluate STRUCT's ability to generate sentences despite negated goals. In this experiment, we again deviate slightly from the problem proposed in the Referring Expression comparison. In that experiment, the referred-to rabbit could be identified uniquely only through the use of many adjectives; the final instance of that problem required a list of 25 adjectives. In this experiment, we demonstrate our generator's ability to avoid using specified actions. We add to our lexicon several new adjectives, each applicable only to the target of our referring expression. Since this entity can now be referred to unambiguously using only one adjective, our generator should select one of these new adjectives. We then encode these adjectives into negated communicative goals, so that they will not be included in the output of the generator, despite allowing a much shorter referring expression.

Figure 3 shows the impact of negated goals on the time to generation. Since this experiment does alter the grammar size, we see the time to final generation growing linearly with grammar size.

Effect of Parameters

Finally, we study the effect of the number of simulations and lookahead depth on the performance of STRUCT. We design this experiment to require lookahead for a good solution by using a sparse reward function that penalizes a final sentence based on the number of adjectives it has. We also use a probabilistic LTAG that has multiple actions all relevant to reaching the goal, but that add differing numbers of adjectives to the sentence. We then run STRUCT on this problem with differing parameter values and report the score of the best solution found, as measured by our reward function (Figure 4).

From the figure, it is clear that as the number of simulations increase, the quality of the solution improves for all values of d. This is likely because increasing simulations means a better estimate of the utility of each action. Further, in this particular case, increasing the depth of lookahead also yields a benefit, becuase of the structure of our problem. This is especially true if the branching factor of the search space is large, which is common in NLG applications. Similarly, the deeper we allow the tree search to continue, the better the estimation of the future value of each action. This is especially true if actions have far-reaching consequences for the meaning of the sentence at its conclusion, which, again, is true in many NLG applications. It is interesting that even for low numbers of simulations $d = 4$ is able able to find a reasonably good solution. These behaviors are expected and verify that STRUCT does not display any pathologies with respect to its parameters.

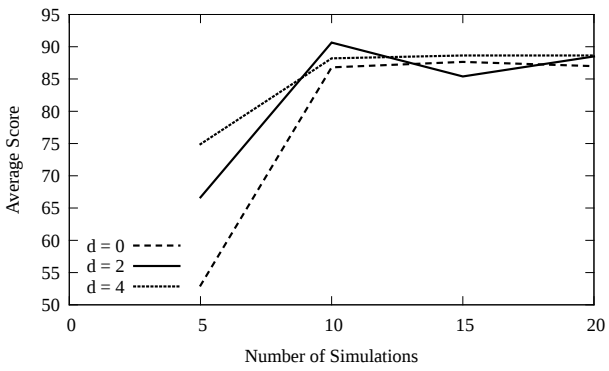

Figure 4: The PLTAG score as a function of number of simulations. Note that $d = 4$ is the best setting at $n = 5$, and remains better than $d = 0$ at $n = 10$.

Conclusion

We have proposed STRUCT, a general-purpose natural language generation system which is comparable to current state-of-the-art generators. STRUCT formalizes the generation problem as an MDP and applies the UCT algorithm, a fast online MDP planner, to solve it. Thus, STRUCT naturally handles proabilistic grammars. We demonstrate empirically that STRUCT is comparable to existing generation-as-planning systems in certain NLG tasks, and is also capable of handling other, more complex tasks such as negated communicative goals. Finally, the algorithm has other desirable properties: it is easy to parallelize and has anytime characteristics. In future work, we plan to explore ways to limit the time complexity of STRUCT as the size of the grammar grows. We also plan to explore ways to learn reward functions that can best guide the UCT planner for a given communicative goal.

References

Bauer, D., and Koller, A. 2010. Sentence generation as planning with probabilistic LTAG. *Proceedings of the 10th International Workshop on Tree Adjoining Grammar and Related Formalisms, New Haven, CT*.

Bauer, D. 2009. Statistical natural language generation as planning. *Proceedings of the 44th annual meeting on Association for Computational Linguistics*.

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press, Princeton, NJ.

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial intelligence* 90(1):281–300.

Brafman, R., and Tennenholtz, M. 2003. R-max-a general polynomial time algorithm for near-optimal reinforcement learning. *The Journal of Machine Learning Research* 3:213–231.

Chevelu, J.; Lavergne, T.; Lepage, Y.; and Moudenc, T. 2009. In *Introduction of a new paraphrase generation tool based on Monte-Carlo sampling*, 249–252. Association for Computational Linguistics.

Fox, M., and Long, D. 2003. Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)* 20:61–124.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: fast plan generation through heuristic search. *J. Artif. Int. Res.* 14(1):253302.

Kay, M. 1996. Chart generation. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, ACL '96, 200204. Stroudsburg, PA, USA: Association for Computational Linguistics.

Kearns, M.; Mansour, Y.; and Ng, A. 1999. A sparse sampling algorithm for near-optimal planning in large markov decision processes. In *International Joint Conference on Artificial Intelligence*, volume 16, 1324–1331. LAWRENCE ERLBAUM ASSOCIATES LTD.

Knight, K., and Hatzivassiloglou, V. 1995. Two-level, many-paths generation. In *Proceedings of the 33rd annual meeting on Association for Computational Linguistics*, 252–260. Association for Computational Linguistics.

Kocsis, L., and Szepesvari, C. 2006. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, 282293. Springer.

Koller, A., and Petrick, R. P. A. 2011. Experiences with planning for natural language generation. *Computational Intelligence* 27(1):2340.

Koller, A., and Stone, M. 2007. Sentence generation as a planning problem. In *ANNUAL MEETING-ASSOCIATION FOR COMPUTATIONAL LINGUISTICS*, volume 45, 336.

Langkilde-Geary, I. 2002. An empirical verification of coverage and correctness for a general-purpose sentence generator. In *Proceedings of the 12th International Natural Language Generation Workshop*, 17–24. Citeseer.

Lu, W.; Ng, H.; and Lee, W. 2009. Natural language generation with tree conditional random fields. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1-Volume 1*, 400–409. Association for Computational Linguistics.

Puterman, M. 1994. *Markov decision processes: Discrete stochastic dynamic programming*. John Wiley & Sons, Inc.

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

Shieber, S. M. 1988. A uniform architecture for parsing and generation. In *Proceedings of the 12th conference on Computational linguistics - Volume 2*, COLING '88, 614619. Stroudsburg, PA, USA: Association for Computational Linguistics.

Stone, M.; Doran, C.; Webber, B.; Bleam, T.; and Palmer, M. 2003. Microplanning with communicative intentions: The spud system. *Computational Intelligence* 19(4):311–381.

White, M., and Baldridge, J. 2003. Adapting chart realization to ccg. In *Proceedings of the 9th European Workshop on Natural Language Generation*, 119–126.