Chapter 1

Introduction

Visions of future artificial intelligence systems have long included natural language interfaces. The ship's computer from Star Trek, the robots from the works of Heinlein, Asimov, and Dick, and the droids from Star Wars are all capable of human speech, and even those which are not capable of dialogue are able to receive orders verbally and respond in kind. The creation of an artificial intelligence like humans have been imagining for many years requires as a precondition the creation of a system for understanding and creating language. It is the latter part of this requirement which we attempt to address in this work.

1.1 Natural Language Generation Systems

Even without these lofty future goals, the increasing frequency of natural language interfaces for consumer products means that natural language generation (NLG) is becoming more and more important in industry. Consequently, a consistent and reliable method for creating a natural language generation system would be of value, especially if the system's computational requirements were small enough that the system could be embedded in consumer electronics. In this work, we propose a system based in statistical planning which serves as a general-purpose natural language generation system. As a part of this work, we propose a method for specifying a grammar based on tree-adjoining grammars using a modified UCT algorithm.

"Give me a sentence about a cat and a dog." Given such a *communicative goal*, most native English speakers can answer a question like this one quite easily. They can also usually provide several similar sentences, differing in details but all satisfying the general communicative goal, and they can usually do this with very little error. Natural language generation (NLG) develops techniques to extend similar capabilities to automated systems. Here, we consider the following restricted NLG problem: given a grammar, lexicon, and a communicative goal, output a valid English sentence that satisfies this goal.

1.2 Contribution

We propose an algorithm which is based in Monte-Carlo Tree Search, an increasingly popular method for probabilistic planning. Our algorithm specifically is based in UCT, the ”Upper Confidence bound applied to Trees” algorithm. This algorithm gives us the ability to efficiently search a large space (all possible natural language outputs) for one of many valid outputs. We compare our work to that of previous Natural Language Generation systems, including one which uses traditional AI Planning techniques. We find that our approach compares favorably in many areas with these state-of-the-art NLG systems.

Chapter 2

Background

2.1 Tree Adjoining Grammars

TAGs are tree-based grammars consisting of two sets of trees, called initial trees and auxiliary or adjoining trees. These two kinds of trees generally perform different roles semantically in addition to their differing syntactic roles. The former, initial trees, are usually for adding new semantic information to the sentence. They add new nodes to the sentence tree. In a simplified Context Free Grammar of English, initial trees contain rules like ”Verb Phrases contain a Verb and a Noun”, or ”VP -¿ V N”. A sentence can be made entirely of initial trees, but a sentence must contain at least one initial tree. An example of an initial tree is shown in Figure ¡SOMETHING¿.
(S (NP (D) (N (Cat))) (VP))
This tree has as its root the S node, and this defines how it can interact with other trees under a TAG. Since this is an initial tree, it can only interact with other trees by substitution. That is, this tree is a drop-in replacement for an S node with no children. This is how we get from our stub sentence (S) to a complete sentence.

Adjoining trees usually clarify a point in a sentence. In a simplified CFG of English, adjoining trees would contain rules like ”a noun can have an adjective placed in front of it,” or ”N -¿ A N”. An example of an adjoining tree is shown in Figure ¡SOMETHING¿.
(N (A (Red)) (N*))
This tree has as its root an N node. It also has a specially annotated N node elsewhere in the tree. These nodes define its interaction with other trees under a TAG. Adjoining trees interact with other trees only by ”adjoining”. In an adjoining action, you select the node to adjoin to, which must be of the same label as the root node of the adjoining tree. You remove that node from the other tree and put the adjoining tree in its place. Then you place that original node into the adjoining tree as a substitution for the foot node. For example, if we had the trees
(S (NP (D (the)) (N (cat))) (VP))
and we wanted to adjoin the example adjoining tree above, we would first create this intermediate tree:

(S (NP (D (the)) (N (A (red) (N*)))) (VP))

And then perform the substitution:

(S (NP (D (the)) (N (A (red) (N (cat))))) (VP))

Notice that this has the effect, in all cases, of making the tree deeper.

We use a variation of TAGs in our work, called a lexicalized TAG (LTAG), where each tree is associated with a lexical item called an anchor. All examples given above are examples of lexicalized trees. An example of an unlexicalized tree would be (NP (D) (N)), where there are no nodes containing lexical tokens.

2.2 Grammar Specification

A grammar, for our purposes, contains a set of trees, divided into two sets (initial and auxiliary). These trees need to be annotated with the entities in them. Entities are defined as any element anchored by precisely one node in the tree which can appear in a proposition representing the semantic content of the tree. These trees are uniquely named, and also contain an annotation (+) representing the lexicalized node.

2.3 Lexicon Specification

The lexicon that we use is a list of permissible word-tree pairings, annotated with the meaning of the pairing. For instance, if the grammar contained a tree named ”a.adj”, (N (A+) (N-foot*)) (an adjoining tree for prepending an adjective to a noun, whose foot node is an entity named ”foot”), then the lexicon might contain an entry ”a.adj: [’red’, ’red(foot)]”. That would mean that the overall LTAG that we are using contains the tree named ”a.adj”, lexicalized to ’red’, and that when that tree is applied to a sentence, the sentence’s meaning is adjusted to include red(name of the foot node).

2.4 Communicative Goal Specification

The communicative goal is just a list of propositions with dummy entity names. Matching entity names refer to the same entity; for instance, a communicative goal of 'red(d), dog(d)' would match a sentence with the semantic representation 'red(subj), dog(subj), cat(obj), chased(subj, obj)', like "The red dog chased the cat", for instance. As you can see, the communicative goal does not have to refer to any "central meaning" of the sentence as a human would select it, but rather to propositions which the sentence affirms.

Chapter 3

Related Work

3.1 Natural Language Generation Systems

Two broad categories of approaches have been used to attack the general NLG problem. One direction can be thought 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 [**?**, **?**]. 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 desired meaning is a valid output from a chart generation system. Another example of this idea is the HALogen/Nitrogen family of systems [**?**]. 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. [**?**, **?**].

A second line of attack formalizes NLG as an AI planning problem. SPUD [**?**], 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* [**?**]. The sentence planner takes in a sentence-sized chunk of information to be conveyed and enriches it in some way. This is then used by a *surface realization* module which encodes the enriched semantic representation into natural language. This chain is sometimes referred to as the "NLG Pipeline" [?]. Our approach is part of this broad category.

Another approach, called *integrated generation*, considers both sentence generation portions of the pipeline together. [?]. This is the approach taken in some modern generators like CRISP [?] and PCRISP [?]. In these generators, the input semantic requirements and grammar are encoded in PDDL [?], which an off-the-shelf planner such as Graphplan [?] 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.

3.2 Grammar Representation

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 (TAGs) are a common choice [?] [?]. 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". The first step finds an adjoining location 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, the target subtree is removed from the sentence tree, and placed 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. We use a variation of TAGs in our work, called a lexicalized TAG (LTAG), where each tree is associated with a lexical item called an anchor.

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 state-of-the-art deterministic planners can currently solve.

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 [**?**] 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.

3.3 NLG Applications

The work we describe here addresses the pure NLG problem without considering the surrounding context; in practice, such a system would be integrated into a larger system, such as one carrying out a dialog. However, we note that many dialog systems, such as NJFun [**?**], model dialog using reinforcement learning. While integrating our NLG approach with such a system is a direction for future work, the similarity of the formalism indicates it should be feasible. NLG has many applications, but one which is of particular interest is natural language interfaces, or dialog systems. Recently, such systems have generated a good deal of interest in mobile devices, though their origin goes back to GUS and similar systems developed at PARC in the 1970s [**?**]. These systems take input from a user in the form of natural-language speech or text, process that information in some way (i.e. running a query against a knowledge base as NJFun does [**?**]), then return a response to the user in natural English. This interaction proceeds by turns in much the same way as a natural dialog between humans. The dialog system is responsible for managing the state of the dialog. By the point that an output realizer is needed, the discourse planning step of the NLG pipeline has already been completed.

Chapter 4

Framework

We formulate NLG as a planning problem on a Markov decision process (MDP) [**?**]. 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, $T : S \times A \times S \to (0, 1)$ 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. $R : S \times A \to \mathbb{R}$ 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 [**?**] 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 [**?**], which is intractable in our case. But for our application, we do not need to find the optimal policy. Rather we just need to *plan* in an MDP to achieve a *given* communicative goal. Is it possible to do this without exploring the entire state-action space? Recent work answers this question affirmatively. New techniques such as sparse sampling [**?**] and UCT [**?**] show how to generate near-optimal plans in large MDPs with a time complexity that is independent of the state space size. Using such an approach with a suitable defined MDP (explained below) allows us to naturally handle probabilistic grammars as well as formulate NLG as a planning problem, unifying the distinct lines of attack described above. Further, the strong theoretical guarantees of UCT translate into fast generation in many cases, as we demonstrate in our experiments. As the state space size in the language generation MDP is very large, we use a variation of the UCT algorithm in our system, described below.

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 that UCT specifies is the method for determining the tree policy. The tree policy needed by UCT for a state s is the action a in that state which maximizes:

$$P(s,a) = Q(s,a) + c\sqrt{\frac{lnN(s)}{N(s,a)}} \qquad [4.1]$$

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 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, called Sentence Tree Realization with UCT (STRUCT). The states of the underlying MDP contain *partial sentences* along with their parse trees. The actions available at a state allow refinements to the partial sentences. In particular, the algorithm may choose an available nonterminal in the current parse tree for a substitution or adjoin operation from a (possibly probabilistic) LTAG. Given a nonterminal choice and a fixed probabilistic lexicalized TAG (PLTAG), a well defined probability distribution is induced over possible next states (possibly uniform if just an LTAG is used). Since trees in LTAGs 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

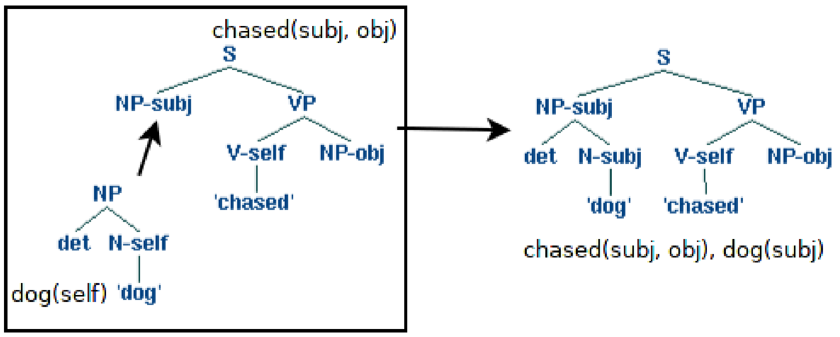


Figure 4.1 An example tree substitution operation in STRUCT.

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 [**?**], and can add auxiliary trees to partial sentences without breaking dependency links between nodes. In situations where the sentence is complete (no nonterminals without children), we add a dummy action to stop generation and emit the sentence. Note that a complete sentence does not necessarily imply a terminal state because adjoin operations can still be performed (e.g. “The dog ran” could be expanded to “The black dog ran quickly”).

In order to control the search space, we restrict the structure of the MDP so that while substitutions are available, only those operations are considered when determining the distribution over the next state, without any adjoins. We do this is in order to generate a complete and valid sentence quickly. This allows STRUCT to operate as an anytime algorithm, described further below.

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.

In order to interact with the communicative goal, each word in our grammars consists of two components, a grammar entry and a lexicon entry. A grammar entry has a name, a tree, and a set of annotations. These annotations contain data about how the syntactic data (the tree itself) interacts with the semantic data. A lexicon entry has a word and a list of meanings. These meanings are written as functions in first order logic, and can use as arguments the objects defined by the tree. This representation is similar to that used by CRISP [?]. In Figure 4 we shown an example of a substitution operation applying an LTAG production to a chosen nonterminal in a partial sentence, and the associated semantic information. The grammar is made up of paired grammar entries and lexicon entries. See Figure 4 for example usage.

While the production probabilities in the given PLTAG define the global structure of the state space, we use the reward function of the MDP to encode the specific communicative goal we are after. Since each state contains a partial sentence with its associated semantic information, we use this to evaluate the algorithm's progress towards the goal, and reward is allocated based on this progress. A key advantage of STRUCT over current techniques is our ability to specify a detailed reward function that captures complex communicative goals. 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", if we so choose. Of course, once any sentence is found that achieves the complete communicative goal, a large reward is given. Since the algorithm keeps track of the cumulative reward, such a sentence becomes a candidate solution. During the search, since we use finite depth lookahead and wish to propagate long range rewards to the root, we use a discount factor of 1.

With the MDP definition above, we use UCT to find a solution sentence (Algorithm 2). 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, after every action is selected and applied, we check to see if we are in a state in which the algorithm could terminate (i.e. the sentence has no nonterminals yet to be expanded). If so, we determine if this is the best possibly-terminal state we have seen so far. If so, we store it, and continue the generation process. If we reach a state from which we cannot continue generating, we begin again from the start state of the MDP. Because of the structure restriction above (substitution before adjoin), STRUCT also generates a valid sentence quickly. These modifications enable STRUCT to perform as an anytime algorithm, which if interrupted will return the highest-value complete and valid sentence.

After this point, any time that there are no substitutions available (all nonterminals have at least one child), we record the current sentence and its associated reward. Such a sentence is guaranteed to be both grammatical and complete. If generation is interrupted, we return the highest-value complete and valid sentence. In this way, our approach functions as an anytime algorithm.

We note that neither of these modifications cause any loss of generality. Our Anytime-UCT implementation will work on any suitably defined MDP. We implemented the system in Python 2.7. The pseudocode is shown in Algorithm 2.

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 different ordering of their adjectives.

4.0.1 AI Planning

Previously, those who interpret the sentence generation problem as planning have primarily used optimal planners to solve these planning problems. CRISP is a prime example of this.

In other domains, there has been recent interest in probabilistic planners. Monte Carlo Tree Search and, more specifically, the Upper Confidence bound applied to Trees (UCT), are exemplars of this subfield.

The UCT algorithm has already been shown to provide significant performance improvements for games played out using a minimax tree [**?**]. When compared with alpha-beta planning, Monte-Carlo planning, and Monte-Carlo planning with minimax value update, UCT was able to provide a lower failure rate with fewer iterations required. It was shown that the number of iteration required for UCT to have an error rate of 0 converged to the same value as the alpha-beta algorithm. However, if a smaller error rate was tolerated, the reduction in the number of iterations required could be significant.

The UCT algorithm can be used for natural language generation with some modifications. Chevelu et al. [**?**] used a variation of UCT, Monte Carlo Paraphrase Generation, in order to search through the possibilities afforded by a paraphrase table. The principal modification, which we have adopted, is to select the action which leads to the best reward, instead of the best average reward. This change is primarily motivated by the lack of adversary in the generation task, which was not the case in the game-playing initial conception of UCT.

4.0.2 UCT

UCT(Upper Confidence bound applied to Trees) [**?**] is a planning algorithm that takes advantage of Monte-Carlo sampling to prune large sections of the search space each iteration. Actions are sampled from each possible action at each state, and an action is chosen immediately based on the best average reward found. This algorithm can handle large search-spaces since it prunes a large percentage of the search space with each action. As it takes more samples per round, it increases the likelihood that it will prune only portions of the search space that do not contain the best output increases.

Our modifications of UCT in order to improve its use in the specific natural language generation task are show in Figure 2. Unlike in the game-playing task for which UCT was designed, we have no adversary in this generation task, and therefore we seek the best path that we have found so far, rather than the maximum average-value path. We have found that UCT, modified in this way, provides excellent optimal and near-optimal outputs. In addition, due to the way that we have structured our action definition, we use UCT as an anytime algorithm; we first generate the simplest and shortest valid sentence first, and increasingly improve the sentence over time until the sentence can no longer be improved.

Our choice of UCT is motivated by the high branching factor encountered in natural language generation when dealing with large grammars as well as the belief that optimal sentences to accomplish a communicative goal are not required in most discourse. In fact, generating optimal sentences may be more time-consuming than nearly-optimal plans which accomplish the goal nearly as well.

In order to further improve the runtime of our approach, we introduced parallel processing into the UCT generation. To do this, multiple processes are spawned, each running the generation steps and returning back the best action. Then, the best actions found from each process will be compared and the action resulting in the best reward will be executed. This allows for many more samples to be run in a system with multiple processors or capable of executing many threads at once, resulting in a better action being discovered in fixed time..

The process of generation can be split into four components:

4.0.2.1 Tree Policy

Starting from the current state, an action will repeatedly be chosen based on a balance of exploration and exploitation, until a state with an open action is hit, a leaf is hit, of the depth limit is reached. The balance between exploration and exploitation is maintained by probabilistically selecting actions based on their expected value and the number of samples taken from that state. More precisely:

$$P(a) = Q(s,a) + c\sqrt{\frac{lnn(s)}{n(s,a)}}$$

where s is the current state, a is the action from the current state, c is an exploration constant, $n(s)$ is the number of times state s has been encountered, $n(s,a)$ is the number of times action a was taken in state s, P(a) is the probability that action a is randomly selected, and $Q(s,a)$ is the expected value of state s after selecting action a.

4.0.2.2 State Expansion

If there are any unexplored actions for the current state, choose one of the unexplored actions at random. This will remove the chosen action from the open action list for that state.

4.0.2.3 Default Policy

Continue to select actions at random until either a terminal state or the depth limit is reached.

4.0.2.4 Reward Assignment

Calculate the reward function for the state that results after executing all of the chosen action and push the reward all the way back up to the root node representing the current state.

4.0.3 STRUCT

Here, we present STRUCT, 'Sentence Tree Realization using UCT', a statistical general-purpose natural language generator. STRUCT is our implementation of the algorithm shown in Figure 2, written in Python 2.

4.0.3.1 Grammar Choice

STRUCT supports Context Free Grammars (CFGs), Probabilistic Context Free Grammars (PCFGs), Tree Adjoining Grammars (TAGs), and Probabilistic Tree Adjoining Grammars (PTAGs). For the experiments below, we primarily focus on a restricted subset of TAGs and PTAGs, the Lexicalized Tree Adjoining Grammars (LTAGs, PLTAGs). LTAGs are TAGs in which each tree contains at least one word which will be present in the final sentence, an 'anchor word'. CRISP, mentioned above, also uses LTAGs, which makes comparison between the generators simpler.

LTAGs were chosen due to their interesting linguistic properties. 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, we have removed recursion from the domain of dependencies [**?**], and can add auxiliary trees to our partial sentences without breaking dependency links between nodes.

4.0.3.2 Action Definition

Actions in STRUCT are applications of a particular grammar rule to a specific node in the partial tree. In a context-free grammar and in the case of an initial tree in a TAG, these are determined by iterating through all leaf nonterminals and adding a possible action for each rule in the grammar applicable at this point. Adjoining trees in a TAG are determined by iterating through each node in the tree and adding a possible action for each adjoining tree applicable at that point.

4.0.3.3 Reward Function Definition

The reward function for a state must be defined exclusively in terms of the state in question. The reward function serves as a metric to rank the favorableness of a sentence. This reward function must be efficiently computable since it will be computed for every iteration in the UCT algorithm. It must also be able to provide a value for a partial sentence since, due to the depth limit, a complete sentence may not always be reached.

Algorithm 2 Pseudocode explanation of UCT

```
Require: Number of simulations $numTrials$, Depth of lookahead $maxDepth$
Ensure: Generated sentence tree
 1: $bestSentence \leftarrow$ nil
 2: while User has not interrupted generation do
 3:     $state \leftarrow$ empty sentence tree
 4:     while $state$ not terminal do
 5:         for $numTrials$ do
 6:             $testState \leftarrow state$
 7:             $currentDepth \leftarrow 0$
 8:             if $testState$ has unexplored actions then
 9:                 Apply one unexplored PLTAG production chosen uniformly at random to
                    $testState$
10:                 $currentDepth$++
11:             end if
12:             while $currentDepth < maxDepth$ do
13:                 Apply PLTAG production selected by tree policy (Equation 4.1)
14:                 $currentDepth$++
15:             end while
16:             calculate reward for $testState$
17:             associate reward with first action taken
18:         end for
19:         $state \leftarrow$ maximum reward $testState$
20:         if $state$ score $>$ $bestSentence$ score and $state$ has no nonterminal leaf nodes then
21:             $bestSentence \leftarrow state$
22:         end if
23:     end while
24: end while
25: return $bestSentence$
```

Chapter 5

Experiments

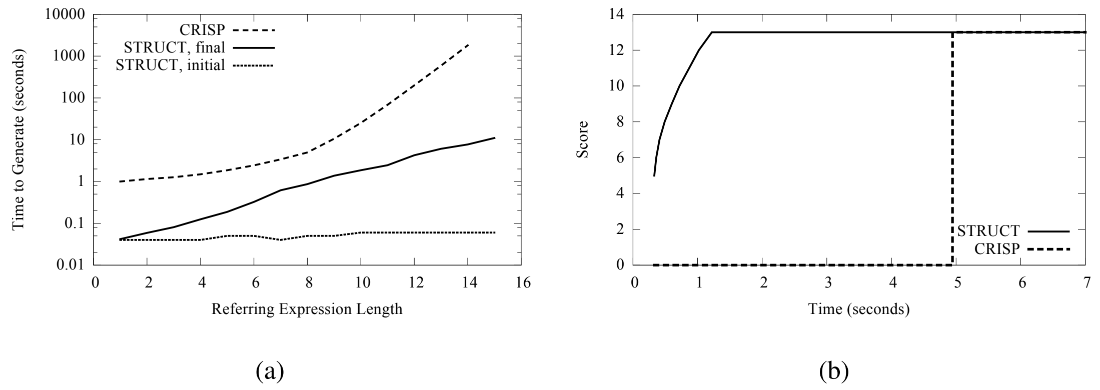


Figure 5.1 Experimental comparison between STRUCT and CRISP. (a) Generation time vs. length of referring expression (b) Score of best solution vs time.

In this section, we compare STRUCT to a state-of-the-art NLG system, CRISP [1] 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 communicating complex propositions, including multiple concurrent goals, negated goals, and nested subclauses. Finally, we evaluate the effect on STRUCT's performance of varying key parameters, including grammar size.

5.0.4 Comparison to CRISP

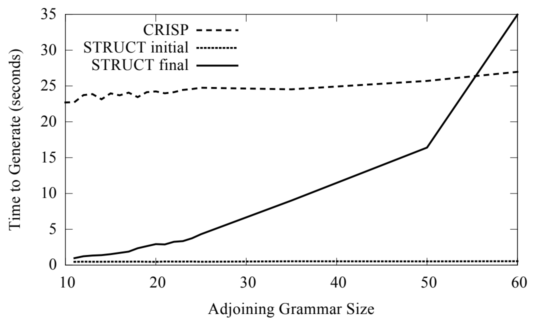
We first describe experiments comparing STRUCT to CRISP. We used a 2010 version of CRISP which uses a Java-based GraphPlan implementation. In these experiments, we use a deterministic grammar. Our reward function gives 50 points for using the correct communicative goal, and an additional $10 + \frac{20}{R}$ points for a correct reference to an argument, where R is the number of other entities that are possible referents for the argument. Because the reward signal is fine-grained, a myopic action selection strategy is sufficient for these experiments, and the d parameter is set to zero. The number of simulations for STRUCT varies between 20 to 150.

[1] We considered using the PCRISP system as a baseline [**?**]. However, we could not get it to work, and we did not receive a response to our queries, so we were unable to use it.

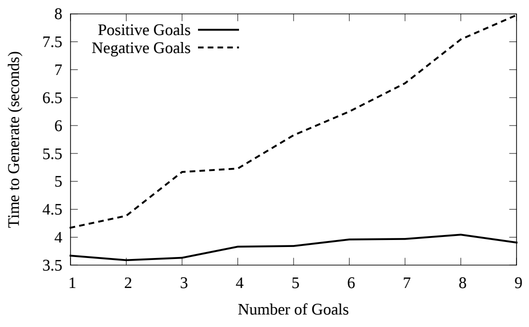
Referring Expressions. We first evaluate CRISP and STRUCT on their ability to generate referring expressions. We follow prior work ([**?**]) in our experiment design. We consider a series of sentence generation problems which require the planner to generate a sentence equivalent to "The Adj$_1$ Adj$_2$ ... Adj$_k$ dog chased the cat.", where the string of adjectives is a string that distinguishes one dog (whose identity is specified in the problem description) from all other entities in the world. The experiment has two parameters: j, the number of adjectives in the grammar, and k, the number of adjectives necessary to distinguish the entity in question from all other entities. We set $j = k$ and show the results in Figure 5.1 (a). We observe that CRISP was able to achieve sub-second or similar 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, performs much better and is able to generate much longer referring expressions without failing. Later experiments had successful referring expression generation of lengths as high as 25.

We can also observe the anytime nature of STRUCT from this experiment, shown in Figure 5.1 (b). Here we look at the length of the solution sentence generated as a function of time, for $k = 8$, a mid-range scenario which both generators are able to solve relatively quickly ($< 5s$). As expected, CRISP produces nothing until the end of its run, at which point it returns the solution. STRUCT, however, quickly produces a reasonable solution, "The dog chased the cat." 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.

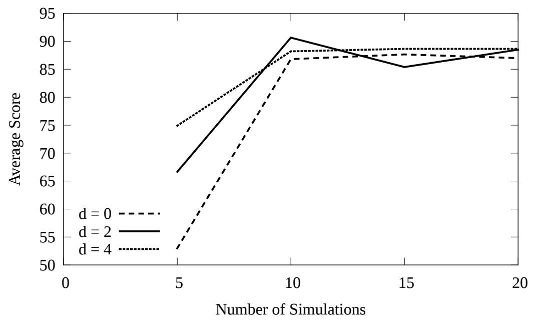
Grammar Size. We next evaluate STRUCT and CRISP's ability to handle larger grammars. This experiment is set up in the same way as the one above, with the exception of l "distracting" words, words which are not useful in the sentence to be generated. l is defined as $j - k$. In these experiments, we vary l between 0 and 50. Figure 5.2(a) shows the results of these experiments. We observe that CRISP using GraphPlan, as previously reported in [**?**], handles an increase in number of unused actions very well. Prior work reported a difference



(a)



(b)



(c)

Figure 5.2 (a) Effect of grammar size, (b) Effect of multiple and negated goals, (c) Effect of parameter variations on the STRUCT solution.

on the order of single milliseconds moving from $j = 1$ to $j = 10$. We report similar variations in CRISP runtime as j increases from 10 to 60: runtime increases by approximately 10% over that range.

STRUCT's performance with large grammars is similar to CRISP using the FF planner [**?**], also profiled in [**?**], which increased from 27 ms to 4.4 seconds over the interval from $j = 1$ to $j = 10$. 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 32 seconds. This is due almost entirely to the required increase in the value of n (number of samples) as the grammar size increases. At the low end, we can use $n = 20$, but at $l = 50$, we must use $n = 160$ in order to ensure perfect generation as soon as possible. Fortunately, as STRUCT is an anytime algorithm, valid sentences are available very early in the generation process, despite the size of the set of adjoining trees (the “STRUCT Initial” curve in Figure 5.2(a)). 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.

5.0.5 Evaluation of Complex Communicative 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 when generating a single sentence. In this experiment, we modify the problem from the previous section. In that section, the referred-to dog 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 dogs. 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 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. We find that, universally, these otherwise useless adjectives are included in the output of our generator, demonstrating that STRUCT is successfully balancing multiple communicative goals. As we show in figure 5.2(b) (the “Positive Goals” curve) , 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 given negated communicative goals. We again modify the problem used earlier by adding 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 just 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. We find that these adjectives which should have been selected immediately are omitted from the output, and that the sentence generated is the best possible under the constraints. This demonstrates that STRUCT is balancing these negated communicative goals with its positive goals. Figure 5.2(b) (the "Negative Goals" curve) shows the impact of negated goals on the time to generation. Since this experiment alters the grammar size, we see the time to final generation growing linearly with grammar size. The increased time to generate can be traced directly to this increase in grammar size.

5.0.6 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 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 5.2(c)).

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, because 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, especially since actions have far-reaching consequences for the meaning of the sentence at its conclusion. 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.

Chapter 6

Conclusion