# A DECISION THEORETIC APPROACH TO NATURAL LANGUAGE GENERATION

by

Nathan McKinley

Submitted in partial fulfillment of the requirements
For the degree of Master of Science

Thesis Advisor: Dr. Soumya Ray

Electrical Engineering and Computer Science
CASE WESTERN RESERVE UNIVERSITY

2013

## ACKNOWLEDGMENTS

I thank my advisor, Professor Soumya Ray, for his assistance through the entire process of developing this idea and creating this thesis. When I began this program I knew very little of the process of research, but Professor Ray guided me through creating my first prototype of this system and my first paper for submission to a conference. From there, we ended up here, with the submission of my thesis in order to obtain the degree of Master of Science. I am very thankful to him for all his help, edits, and guidance.

I also thank Professor Wyatt Newman for suggesting that I attempt this, my thesis committee for reviewing my work, and (some other people).

# TABLE OF CONTENTS

Page

# LIST OF FIGURES

Appendix
Figure | Page

# A Decision Theoretic Approach to Natural Language Generation


## Abstract

by

NATHAN MCKINLEY

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 suitably defined reward function that reflects the communicative goal. We then use probabilistic planning to solve the MDP and generate a sentence that, with high probability, accomplishes 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 is anytime and can handle complex communicative goals, including negated goals.

# Chapter 1

# Introduction

Artificial Intelligence (A.I.) systems are becoming increasingly prevalent in modern consumer products. Google's "Google Now" system determines what information its user wants to see before they ask for it. Apple's "Siri" acts as an artificial personal assistant, attempting to respond to queries stated in natural language. Android (which includes Google Now) and iOS (which includes Siri) have 85% market penetration between them, and there are over one billion smartphones active.

Visions of future A.I. systems have long included natural language interfaces. The ship's computer from Star Trek, the robots from the works of Heinlein and Asimov, 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.

Consequently, a crucial subcomponent of artificial intelligence is Natural Language Processing (NLP). NLP studies the task of interacting with humans using languages which are inherently complex and ambiguous (e.g. English). In order to successfully communicate with people, a system which does natural language processing will need to accept language as input and translate it into a format that computers can work with. Such a system will also need to be able to translate from an internal meaning representation to natural language. See Figure 1.1 for a visual explanation of this process.

For example, consider a robotic concierge system which takes phone calls from users. A system like this has been shown to be practical[1], so it serves as a good example of an NLG interaction. One possible interaction might be the following:

```
Computer:  How may I help you today?
User    :  I am looking for a place to eat dinner.
```

Once the user has finished speaking, the computer system has a series of electrical signals on a phone line which it will need to decode into human speech. This is the first step in the block diagram of Figure 1.1. Current-generation speech recognition software is highly reliable for most accents, so let us assume that this step proceeds without error. At this point, the computer system will need to attempt to understand the natural language input of ”I am looking for a place to eat dinner”. There are many approaches to this problem, but the previously-mentioned concierge system was able to get by with a simple keyword-searching approach. Such an approach might look for the words ”I”, ”place”, ”eat”, and ”dinner”, and determine that the speaker desires a listing of restaurants that are open for dinner. This is the ”NLP Parsing” step of Figure 1.1. The system would then do some internal processing (e.g. database queries), which makes up the ”Processing” step. The system may determine that it needs to respond to the user, perhaps to ask a clarifying question to narrow the user’s search. This determination will be made in the ”Response Generation” block. At this point, the system will have an internal representation of the type of question it needs to ask which we call a ”communicative goal”. That goal might look something like the following:

```
question type: clarifying
question seeks: preferences
question regards: dinner type.
question language: English
```

At this point, the computer system needs to translate that into a sentence in its output language. The process of going from this internal representation to text like ”What type of food

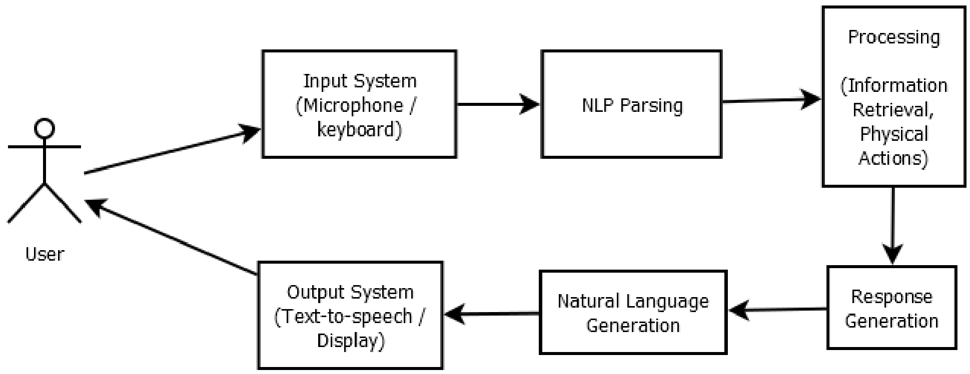



Figure 1.1 A diagram showing the flow of information through a normal interaction with a user in a generalized NLP system

do you typically enjoy for dinner?” is called Natural Language Generation, and comprises the fifth block of the NLP process. Finally, this generated text is conveyed to the user by encoding it into the data that can travel over a telephone, using a text-to-speech program.

As this example shows, even without the aforementioned lofty future goals, the increasing frequency of natural language interfaces for consumer products shows that natural language generation (NLG) is becoming more and more important in the world. 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 thesis, we consider the following restricted NLG problem: given a grammar, lexicon, and a communicative goal, output a valid English sentence that satisfies this goal. We call this problem ”restricted”, because it assumes a level of knowledge about the context in which the generation will occur. In principle, the most general NLG problem is to produce a valid sentence in an arbitrary language, using the entire grammar and lexicon available in that language, to satisfy an arbitrary communicative goal. In practice, we rarely require this level of generality; it would be unnecessary for the concierge program described above to be intimately familiar with the finer points of Klingon grammar. In practice, therefore, our restricted problem is a reasonable one.

Previous work on this problem has taken two broad approaches. On one hand we have the classical planning approaches, which treat the problem of NLG as an AI planning problem. These systems have the advantage of being capable of finding acceptable output in all circumstances where a perfect answer exists, but the disadvantage of being unable to handle a probabilistic grammar in a structured way. They also struggle with completing a subset of the communicative goals, when full completion is not possible. On the other hand, there are statistical planners which work by examining the most probable combinations of words and determining if any of those match the meaning they are attempting to generate. These have the advantage of completing execution very quickly, and of taking advantage of Zipf's Law, which states broadly that a very small subset of a language is used a very large proportion of the time. They have the disadvantage of requiring an inordinate amount of memory to be able to search for the less-likely phrases and sentences which will occasionally need to be generated. They also scale poorly with large grammar sizes.

We propose an algorithm which unifies these two approaches, and has, to some extent, the advantages of both. This algorithm gives us the ability to efficiently search a large space (all possible natural language outputs) for one of many valid outputs. We support probability in a structured way and allow for generation using a large grammar. We support partial completion of the communicative goal, but strive for full completion when it is possible. We allow for generation to stop at any time and will return a partially-complete result very quickly.

We do this, broadly, by using probabilistic planning rather than classical planning. Our algorithm is based in Monte-Carlo Tree Search, an increasingly popular method for probabilistic planning.

We believe that if this algorithm were developed further, it could be useful as the final step of a dialog system, or useful in generation situations where flexibility of the generation system is crucial. At present, it is already useful as the output stage of a simple dialog system. An efficient implementation would be able to take advantage of multiprocessing and therefore run very well on a massively multiprocessing system, enhancing performance substantially.

In chapter 2, we provide background information on NLG and probabilistic planning. In chapter 3, we discuss related work, including historical natural language generation frameworks. We also discuss the closest related system to ours, called CRISP. In chapter 4, we present our framework for natural language generation. In chapter 5, we present our experimental evaluation of our implementation, which shows that it performs comparably to the current state-of-the-art in the field. In chapter 6, we conclude by describing the potential applications and future work using this method of generation.

# Chapter 2

# Background

In this chapter, we provide some background on the algorithms and underlying formalism which we build on in this work. We explain Markov Decision Processes, classical AI planning, probabilistic planning, natural language grammars, and Monte Carlo Tree Search, each of which is an important component in our algorithm.

## 2.1 Markov Decision Processes

A Markov Decision Process (MDP) [2] is a tuple $(S, A, T, R, \gamma)$ 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, $\gamma$ 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 [3] 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.

## 2.2 Planning

Planning is the problem of creating a policy of action selection which fulfills some goal. Depending on the environment, actions may be drawn from some distribution dependent on the state of the world in which planning is being done. This state may be observable, partially observable, or invisible to the agent doing the planning. The actions that the agent takes usually transform the state of the world in some way, and the goal is defined in terms of the state of the world.

### 2.2.1 Classical Planning

Classical planning is a planning problem with six conditions. In a classical planning problem, there is a single initial state, which is fully known. Actions are deterministic, can only be taken by the single agent which is present in the world, and take no time. These three conditions combine to mean that the world does not change without the agent causing the change. The goal is one or more states which are reachable from the initial state, and Finally, actions are sequential.

Classical planning has been studied extensively, and is widely considered to be the simplest possible planning problem. These problems are usually solved by two broad categories of planning system: forward state-space planners and backward planners. Forward state space planners plan by taking actions from the initial state and examining the state that results from the action. Since actions are deterministic, it is straightforward to see that such a planner would be guaranteed to find the goal state by trying every possible combination of actions. Backward planners plan by working backwards from the goal: they maintain a frontier and iteratively check which states in the state space can reach the states in the frontier.

One popular backward planning algorithm is Graphplan[4]. Graphplan works by creating a ”planning graph” which contains two types of nodes and three types of edges. The two node types are either representing facts in the world or representing actions which can be taken in the world. The three edge types are between a fact node and an action node, between an action node and a fact node, and between two nodes of the same type. Edges from a fact node to an action node represent preconditions (that action can be taken if that fact is either true or false), and edges from an action node to a fact node represent effects (that fact becomes true or false when the action is taken). Edges from a node to a node of the same type represent mutual incompatibility; two facts which cannot be true at the same time or two actions which cannot be taken simultaneously (due to altering facts which are required preconditions, for instance).

Graphplan constructs this graph one level at a time, starting from the goal and working towards a state in which all the facts which are true in the initial state are true in the graph. It then works backwards by picking actions that will reach the goal state from the initial state. Graphplan is, therefore, not a state-space planner, since its graph does not have a representation for states specifically.

### 2.2.2 Probabilistic Planning

Probabilistic planning is an alternative to Classical Planning when the preconditions required to solve the easier problem do not hold. Probabilistic planning is done on an MDP, where actions are not required to be deterministic and where we attempt to maximize a reward function rather than reach a particular state. PP does require full observability of the state (this was irrelevant during the classical planning problem, since exploration with deterministic actions is trivial). PP does not require that the initial state be given or known, which means that it is often described as determining a policy given a current state rather than an ordered list of actions to take.

Determining the optimal policy at *every* state using the TD strategy described in the above MDP discussion is polynomial in the size of the state-action space [5], which is often intractable. But for many applications, we do not need to find the optimal policy in all states; rather we just need to *plan* in an MDP from an initial state to achieve a single communicative goal. New techniques such as sparse sampling [6] and UCT [7] show how to generate near-optimal plans in large MDPs with a time complexity that is independent of the state space size.

A recently popular PP algorithm is the Upper Confidence bound applied to Trees (UCT)[7]. 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:

$$P(s,a) = Q(s,a) + c\sqrt{\frac{lnN(s)}{N(s,a)}} \qquad [2.1]$$

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.

## 2.3 Natural Language Grammars

A grammar is a set of rules which define strings that are contained within a language. Many artificial languages, especially programming languages like C or Java, have grammars which can be expressed concisely and without ambiguity. Some constructed languages, like Lojban, also have this property.

Natural languages (e.g. English), however, are well-known to have grammars which are difficult to represent with any single given formalism. Many attempts have been made to create a grammar which can sufficiently define natural language, the XTAG project chiefly among them, but even those projects fail to sufficiently embody the constraints that spoken and written English put on word orderings and meaning.

There are many possible formalisms which can contain the rules that make up a grammar. Most of these formalisms involve a form of ”rewriting”, which is to say, replacing a nonterminal token in a string with a specific set of terminals and nonterminals.

### 2.3.1 Context-Free Grammars

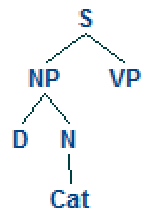
One of the simplest grammars which can convey relationships between terminals and nonterminals is the ”Context Free Grammar” (CFG). A CFG is made up of rules, each of which specifies a possible rewrite of a nonterminal into one or more terminals or nonterminals. Once there are no further nonterminals to be rewritten, the final state is a series of terminals which is in the language defined by the grammar. This language is defined as the set of all possible generated sentences.

CFGs are so named because the rules in them are unable to consider the ”context” for their rewriting. No rules may condition on the presence or placement of terminals or nonterminals other than the single nonterminal to be rewritten. This is, of course, insufficiently expressive to be a grammar for English (or any other natural language).

One attempt to make CFGs more realistic for parsing or generating natural languages was to introduce a probabilistic component. In a Probabilistic Context Free Grammar (PCFG), CFG rules are tagged with probabilities, usually representing the frequency of their appearance in an observed corpus. These probabilities are required to sum to 1 for each nonterminal to be rewritten, so that there is a defined probability distribution over the options for each nonterminal.

This grammar type does appropriately convey the crucial truth that that not all rules are equal. However, it does so in a naive way, unable to express these probabilities in terms of

Figure 2.1 An example of an initial tree




the placement of the nonterminal in a sentence, and consequently also unable to successfully express English.

### 2.3.2 Tree Adjoining Grammars

A Tree Adjoining Grammar (TAG) takes a different approach, differing substantially from CFGs and PCFGs. TAGs are tree-based grammars consisting of two sets of trees, called initial trees and adjoining trees (sometimes “auxiliary trees”). These two kinds of trees tend to 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 TAG 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 2.1.

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 TAG 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 2.2.

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

Figure 2.2 An example of an adjoining tree

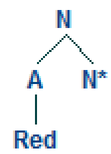
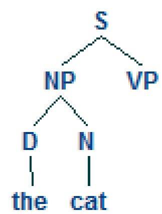



Figure 2.3 Partial tree




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 tree in Figure 2.3 and we wanted to adjoin the example adjoining tree in Figure 2.2, we would first create the intermediate tree in Figure 2.4, and then perform the substitution and get the tree in Figure 2.5. Notice that this has the effect, in all cases, of making the tree deeper.

And then perform the substitution:

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

Figure 2.4 Intermediate tree

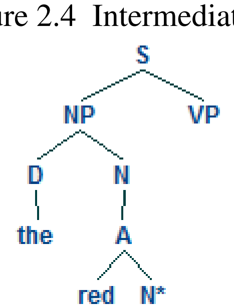
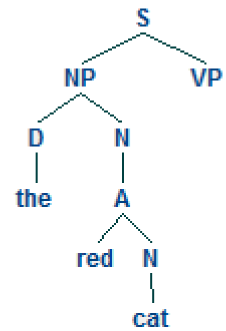


Figure 2.5 Final tree




lexicalized trees. An example of an unlexicalized tree would be (NP (D) (N)), where there are no nodes containing lexical tokens.

As with CFGs, an attempt to make TAGs more tractable for generation or parsing was to introduce probabilities. Each tree rule in a TAG must be annotated with a probability, and there will be a probability distribution for each nonterminal at the frontier of the tree. These probabilistic TAGs, or PTAGs, can be useful if a corpus of text is available.

The XTAG project has had some success using LTAGs to model English grammar, so we focus mostly on LTAGs in our work.

# Chapter 3

# Related Work

## 3.1 Approaches to NLG

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 [8, 9]. 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 [10]. 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. [11, 12].

A second line of attack formalizes NLG as an AI planning problem. SPUD [13], 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* [14]. 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" [15]. Our approach is part of this broad category.

Another approach, called *integrated generation*, considers both sentence generation portions of the pipeline together. [16]. This is the approach taken in some modern generators like CRISP [16] and PCRISP [17]. In these generators, the input semantic requirements and grammar are encoded in PDDL [18], which an off-the-shelf planner such as Graphplan [19] 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 [16] [17]. 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 [17] 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 [20] 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 [1], 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 [21]. 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 [1]), 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

## 4.1 NLG as planning in an MDP

We formulate NLG as a planning problem on a Markov decision process (MDP) 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 in chapter 2.2.

## 4.2 STRUCT

### 4.2.1 Overview

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 and semantic annotations. 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 choose any nonterminal for an 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

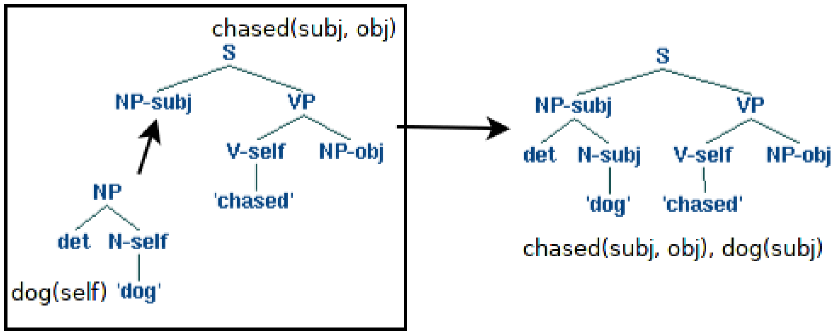



Figure 4.1 An example tree substitution operation in STRUCT.

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 [22], 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.

### 4.2.2 Grammar Description

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,

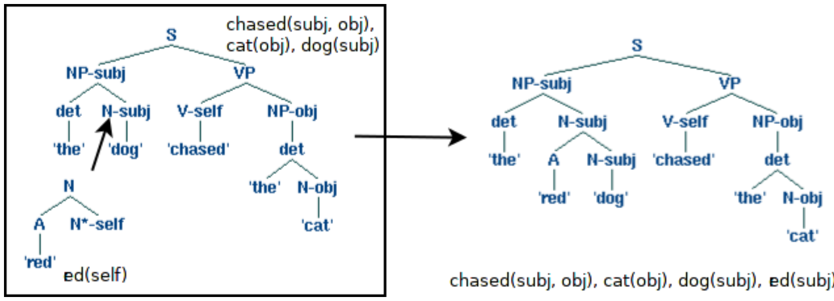



Figure 4.2 An example tree adjoining operation in STRUCT.

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 [16]. In Figure 4.2.1 we show an example of a substitution operation applying an LTAG production to a chosen nonterminal in a partial sentence, and the associated semantic information. Figure **??** similarly shows an example of an adjoining operation.

## 4.3 Specifications of Inputs

### 4.3.1 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. See Appendix A for an example of a combined grammar / lexicon.

### 4.3.2 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). Appendix A contains an example of a combined grammar / lexicon.

### 4.3.3 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 shown here, 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. Appendix B contains an example of a communicative goal and a number of sentences which satisfy it.

### 4.3.4 Reward Function

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.

### 4.3.5 Execution as an Anytime Algorithm

With the MDP definition above, we use UCT to find a solution sentence (Algorithm 4.3). 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 4.3.

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.

### 4.3.6 Algorithm Details

UCT(Upper Confidence bound applied to Trees) [7] 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 4.3. 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

Figure 4.3 The STRUCT Algorithm.

**Require:** Number of simulations $numTrials$, Depth of lookahead $maxDepth$

**Ensure:** Generated sentence tree

```
bestSentence ← nil
while User has not interrupted generation do
    state ← empty sentence tree
    while state not terminal do
        for numTrials do
            testState ← state
            currentDepth ← 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 2.1)
                currentDepth++
            end while
            calculate reward for testState
            associate reward with first action taken
        end for
        state ← maximum reward testState
        if state score > bestSentence score and state has no nonterminal leaf nodes then
            bestSentence ← state
        end if
    end while
end while
return bestSentence
```

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.

The process of generation can be split into four components:

1. Tree Policy
2. State Expansion
3. Default Policy
4. Reward Assignment

#### 4.3.6.1 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 $a$ is an action, $s$ is the current state, $c$ is the algorithm's 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, and $Q(s, a)$ is the expected value of state $s$ after selecting action $a$.

#### 4.3.6.2 State Expansion

If there are any unexplored actions for the current state, choose one of the unexplored actions according to an arbitrary heuristic. Heuristic choice is described below.

#### 4.3.6.3 Default Policy

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

#### 4.3.6.4 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.

### 4.3.7 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 [22], and can add auxiliary trees to our partial sentences without breaking dependency links between nodes.

### 4.3.8 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.

### 4.3.9 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.

We found that there are several possible reward functions which will return satisfactory results on a variety of experiments. The reward function which we found to be nearly universally useful considers a goal as well as a grounded world. It examines all possible mappings of entities present in a given sentence's annotation to entities present in the world, then returns a high value if there are any mappings which are both possible (contain no statements which are not present in the grounded world) and fulfill the goal (contain the goal statement).

This reward function happens to perform somewhat slowly since it includes a subsumption problem as a step within it. It performs in $O(N!)$ time, where N is the number of entities present in the sentence. We refer to this algorithm as ”perfect evaluation”.

Since on occasion we will not need the power of a perfectly correct algorithm, it's worthwhile to consider alternate algorithms. One such algorithm assumes that the given sentence is correct and that all entity references are appropriate, then searches for statements in the sentence which contradict this assumption. This algorithm obviously runs much faster but has limitations. One such limitation is that such an approach cannot perform an explicit mapping of entities in the sentence to entities in the world, which means that some sentences which do not precisely match the communicative goal will have a positive result.

We refer to STRUCT using our perfectly correct algorithm as STRUCT$_a$, and STRUCT using our more efficient algorithm as STRUCT$_b$. Our experiments will show that STRUCT$_b$ has better performance than STRUCT$_a$, but that STRUCT$_a$ generates better sentences in some domains.

### 4.3.10 Grammar Pruning

English is, of course, a very large language. In most communicative goals, it is not necessary to consider every one of the possible configurations of words and meanings that make up the set of possible expressions. If this was necessary, it would be completely intractable to generate even the simplest sentences.

The approach that we took to reduce the complexity of generation is to ensure that generation takes place using exclusively the relevant words. This is done by selecting the entities in the world which are needed for the goal, then iteratively selecting all meanings transitively related to those entities. Once we have selected all these meanings, we will remove all words from the grammar which have a meaning not contained in this list.

In practice, this means that we often need a very small proportion of the overall language, speeding generation significantly.

# Chapter 5

# Experiments

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.

We will be comparing CRISP to two different versions of STRUCT. As mentioned in the previous chapter, there are two different reward functions which we have written and found to be useful in this domain. We compare to both such functions in order to demonstrate the performance tradeoffs of a system based on a reward function.

## 5.1 Comparison to CRISP

We begin by describing 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. 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. In most cases, a small $n$, under 100, is

[1] We considered using the PCRISP system as a baseline [17]. However, we could not get the system to compile, and we did not receive a response to our queries, so we were unable to use it.

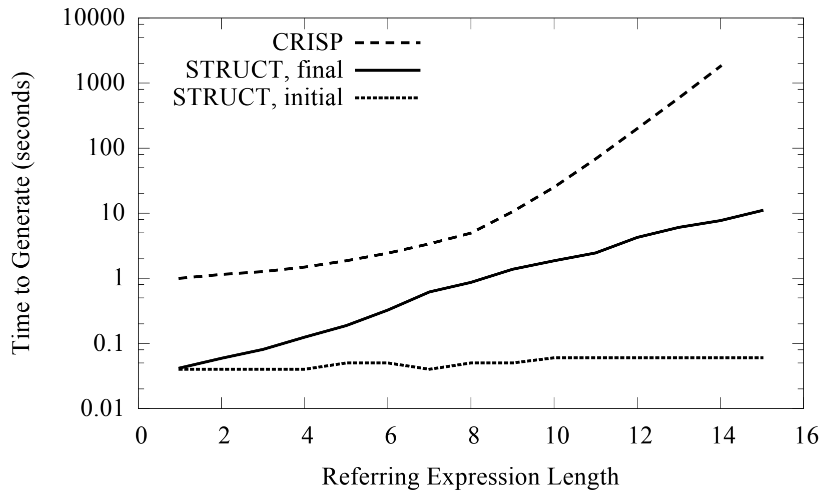



Figure 5.1  Experimental comparison between STRUCT and CRISP: Generation time vs. length of referring expression

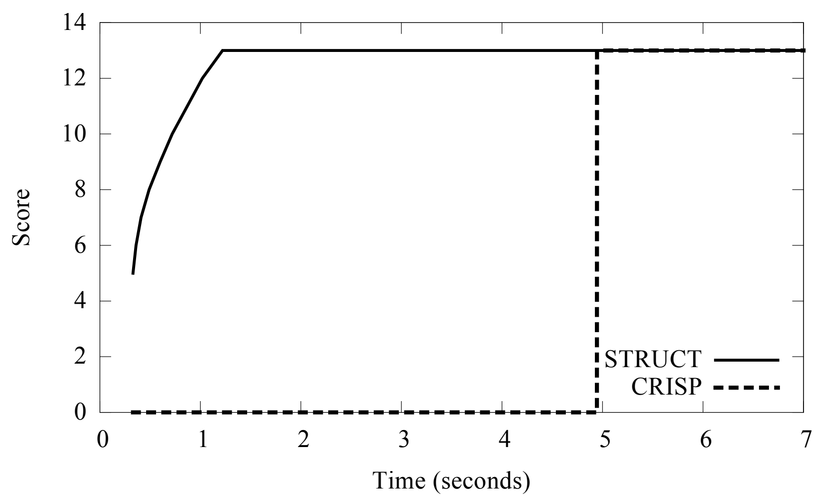



Figure 5.2  Experimental comparison between STRUCT and CRISP: Score of best solution vs time.

sufficient to guarantee generation success. The exploration constant $c$ in Equation 2.1 is irrelevant when $n \leq |\mathbf{A}|$, since it applies only to actions selected after all open actions have already been tried once.

### 5.1.1 Comparison to CRISP: Referring Expressions

We first evaluate CRISP and STRUCT on their ability to generate referring expressions. We follow prior work ([14]) in our initial experiment design. We consider a series of sentence generation problems which require the planner to generate a sentence like "The $\text{Adj}_1$ $\text{Adj}_2$ ... $\text{Adj}_k$ 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. 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. $\text{STRUCT}_b$, 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. $\text{STRUCT}_a$ performs similarly to CRISP asymptotically.

We can also observe the anytime nature of STRUCT from this experiment, shown in Figure 5.2. 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 (both versions) 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.

### 5.1.2 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, 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

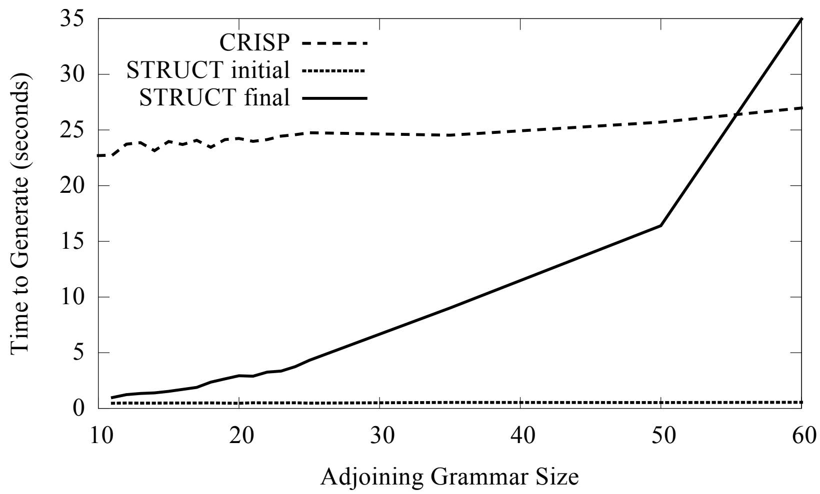



Figure 5.3 Effect of grammar size

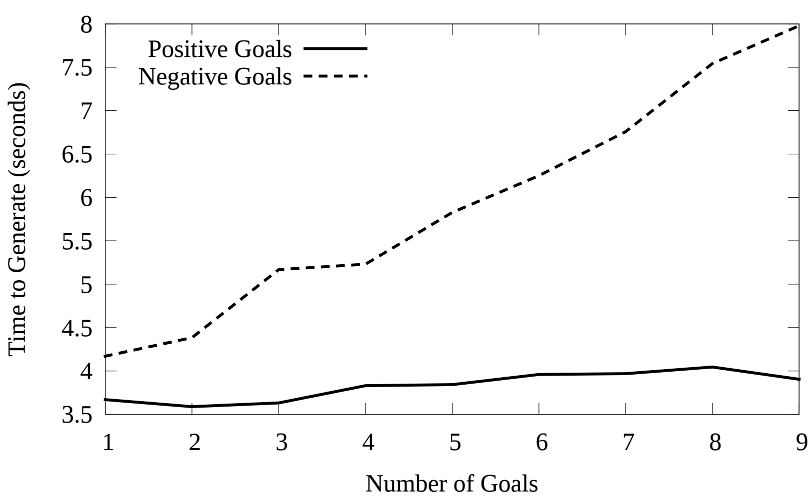



Figure 5.4 Effect of multiple and negated goals

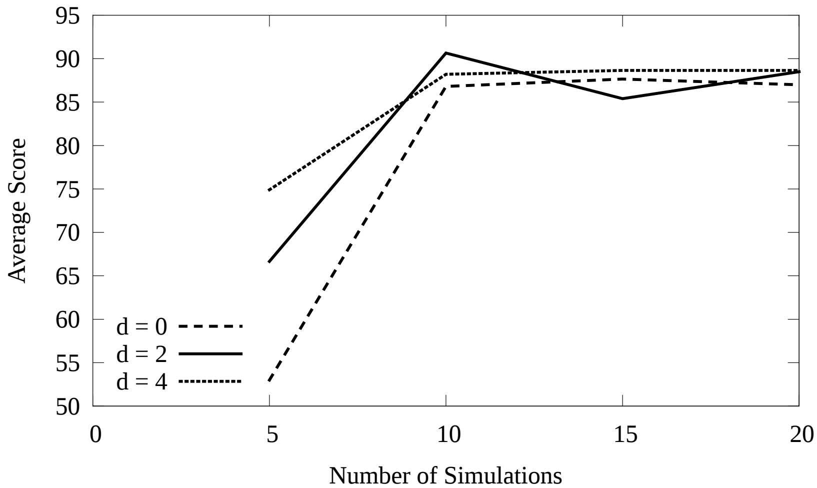



Figure 5.5 Effect of parameter variations on the STRUCT solution.

experiments, we vary $l$ between 0 and 50. Figure 5.1.1 shows the results of these experiments. We observe that CRISP using GraphPlan, as previously reported in [14], handles an increase in number of unused actions very well. Prior work reported a difference 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.

#### 5.1.2.1 Absent Pruning

STRUCT's performance with large grammars is similar to CRISP using the FF planner [23], also profiled in [14], 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 5.1.1). 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.

#### 5.1.2.2 With Pruning

STRUCT's performance with large grammars improves dramatically if we allow for pruning (described in Chapter 4). This experiment involving distracting words is a perfect example of a case where pruning will perform optimally. When we apply pruning we find that STRUCT is able to completely ignore the effect of additionaly distracting words. Experiments showed roughly constant times for generation for $j = 1$ through $j = 5000$. Although pruning is $O(n)$ in grammar size, repeated experiments failed to show any significant distinction in runtime on very large grammars.

## 5.2 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.

### 5.2.1 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 5.1.1 (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.

### 5.2.2 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 our target can now be referred to unambiguously using only one adjective, our generator should just select one of these new adjectives (this has been experimentally confirmed). 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 5.1.1 (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. This is a case where pruning does not help us in reducing the grammar size; we cannot optimistically prune out words that we do not plan to use. Doing so might reduce the ability of STRUCT to produce a sentence which partially fulfills its goals.

### 5.2.3 Nested subclauses

Here, we evaluate STRUCT$_a$'s ability to generate sentences with nested subclauses. An example of such a sentence is "The dog which ate the treat chased the cat". This is a difficult sentence to generate for several reasons. The first, and clearest, is that there are words in the sentence which do not help to increase the score assigned to the partial sentence. Notably, we must adjoin the word ”which” to ”the dog” during the portion of generation where the sentence reads ”the dog chased the cat”. This decision requires us to do planning deeper than one level in the future, which massively increases the number of simulations STRUCT requires in order to get the best possible result. The second reason that this sentence is a challenge for our generation algorithm is that that adjoinment (”the dog” $\rightarrow$ ”the dog which”) corresponds to a tree where the verb that will be a child of ”which” (in this case, ”ate”) does not have its argument as a child. See Figure **??** for a visual explanation of this. Consequently, we need to introduce indirection. We add a node to our tree which represents the implied subject of the verb ”ate”. This is the approach taken by XTAG [24] in their attempt to create a TAG which appropriately represents the English language. See Figure **??** for a visual explanation.

Despite these troubles, STRUCT is capable of generating these sentences. As we can see in Figure **??**, STRUCT's time to generate increases with the number of nested clauses. To the best of our knowledge, CRISP is not able to generate sentences of this form, and consequently we present our results without baselines. We present results only for STRUCT$_a$ here, since STRUCT$_b$ is not capable of generating sentences using indirection.

### 5.2.4 Conjunctions

Here, we evaluate STRUCT$_b$'s ability to generate sentences including conjunctions. We introduce the conjunction ”and”, which allows for the root nonterminal of a new sentence ('S') to be adjoined to any other sentence. We then provide STRUCT with multiple goals. Given sufficient depth for the search ($d = 3$ was determined to be sufficient, as our reward signal is fine-grained), STRUCT will produce two sentences joined by the conjunction ”and”. In this, too, we follow prior work in our experiment design [14] [2].

[2] CRISP was shown to be able to do this in [14], but the grammar that allowed this was never published, and neither was documentation describing how to re-create it. Since our inquiries by email were never answered, we were unable to replicate this experiment and present STRUCT's results with no baselines

As we can see in Figure **??**, STRUCT successfully generates results for conjunctions of up to five sentences. This is not a hard upper bound, but generation times begin to be impractically large at that point, and further experimentation would be unnecessary. Fortunately, human language tends toward shorter discourse units than these unwieldy (but technically grammatical) sentences.

STRUCT increases in generation time both as the number of sentences increases and as the number of objects per sentences increases. We show results for STRUCT$_a$ here, as our output should contain only simple sentences without nesting, and because STRUCT$_b$ is exponential in number of entities in the sentence, which will cause impractically large generation times for this experiment.

### 5.2.5 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 5.1.1).

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

# APPENDIX
# Example Grammars

## A.1 Basic Experiment

Figure A.1 Grammar for the basic experiment

```
"grammar" :
{
    "i.nvn":    "(S(NP-subj)(VP(V+-self)(NP-obj)))",
    "i.np":     "(NP(D)(N+-self))",
    "i.d":      "(D+-self)",
    "i.cv":     "(V+-self)",
    "a.ad":     "(N(A+)(N*-self))",
    "a.sub":    "(N(N*-self)(PP(P+-clause)(VP(V-clauseverb|
        self,clauseobj)(NP-clauseobj))))"
}
```

Figure A.2 Lexicon for the basic experiment

```
"lexicon":
{
    "i.nvn": [{"word" : "chased", "meaning": "chased(subj, obj
        )"}, {"word":"ate", "meaning":"ate(subj, obj)"}],
    "i.d":  [{"word":"the","meaning": None}, {"word":"a", "
        meaning":None}],
    "i.np":   [{"word":"cat", "meaning":"cat(self)"}, {"word
        ":"dog", "meaning":"dog(self)"},
    {"word":"treat", "meaning":"treat(self)"}],
    "a.ad":  [],
    "i.cv": [{"word":"chased", "meaning":"chased(self)"}],
    "a.sub": [{"word":"which", "meaning":None}]
}
```

Figure A.3 World and Goal for the basic experiment

```
{
    "world": ["chased(d1, c)", "dog(d1)", "cat(c)"],
    "goal": "chased(d1, c)"
}
```

# LIST OF REFERENCES


[1] D. Litman, S. Singh, M. Kearns, and M. Walker, "Njfun: a reinforcement learning spoken dialogue system," in *Proceedings of the 2000 ANLP/NAACL Workshop on Conversational systems-Volume 3*, pp. 17–20, Association for Computational Linguistics, 2000.

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

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

[4] A. L. Blum and M. L. Furst, "Fast planning through planning graph analysis," *Artificial intelligence*, vol. 90, no. 1, pp. 281–300, 1997.

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

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

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

[8] S. M. Shieber, "A uniform architecture for parsing and generation," in *Proceedings of the 12th conference on Computational linguistics - Volume 2*, COLING '88, (Stroudsburg, PA, USA), p. 614619, Association for Computational Linguistics, 1988.

[9] M. Kay, "Chart generation," in *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, ACL '96, (Stroudsburg, PA, USA), p. 200204, Association for Computational Linguistics, 1996.

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

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

[12] W. Lu, H. Ng, and W. Lee, "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*, pp. 400–409, Association for Computational Linguistics, 2009.

[13] M. Stone, C. Doran, B. Webber, T. Bleam, and M. Palmer, "Microplanning with communicative intentions: The spud system," *Computational Intelligence*, vol. 19, no. 4, pp. 311–381, 2003.

[14] A. Koller and R. P. A. Petrick, "Experiences with planning for natural language generation," *Computational Intelligence*, vol. 27, no. 1, p. 2340, 2011.

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

[16] A. Koller and M. Stone, "Sentence generation as a planning problem," in *ANNUAL MEETING-ASSOCIATION FOR COMPUTATIONAL LINGUISTICS*, vol. 45, p. 336, 2007.

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

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

[19] A. Blum and M. Furst, "Fast planning through planning graph analysis," *Artificial intelligence*, vol. 90, no. 1, pp. 281–300, 1997.

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

[21] D. G. Bobrow, R. M. Kaplan, M. Kay, D. A. Norman, H. Thompson, and T. Winograd, "Gus, a frame-driven dialog system," *Artificial intelligence*, vol. 8, no. 2, pp. 155–173, 1977.

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

[23] J. Hoffmann and B. Nebel, "The FF planning system: fast plan generation through heuristic search," *J. Artif. Int. Res.*, vol. 14, p. 253302, May 2001.

[24] X. R. Group, "A lexicalized tree adjoining grammar for english," Tech. Rep. IRCS-01-03, IRCS, University of Pennsylvania, 2001.