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

Appendix
Figure

A Decision Theoretic Approach to Natural Language Generation

Abstract

by

NATHAN MCKINLEY

We study the problem of generating a single sentence which satisfies a communicative goal, given a grammar which specifies the language. We model the generation process as a Markov Decision Process (MDP), using a reward function which reflects that communicative goal. We use probabilistic planning to solve the MDP and generate a sentence which satisfies the communicative goal. We compare our approach to a current state-of-the-art generation system, and find that our system can generally match the state-of-the-art in both performance and generation quality, while offering generation capabilities that exceed those usually found in planning-based generators.

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 45% market penetration between them, and there were 470 smartphones sold in 2011, with expectations of increase in 2012 and 2013 [2].

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. A system for understanding and creating language is a fundamental part of an artificial intelligence like humans have been imagining for many years.

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 [3], so it serves as a good example of an

interaction with a system capable of generating language. 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". One very simple possibility is a straightforward 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
```

The computer system next 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 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.

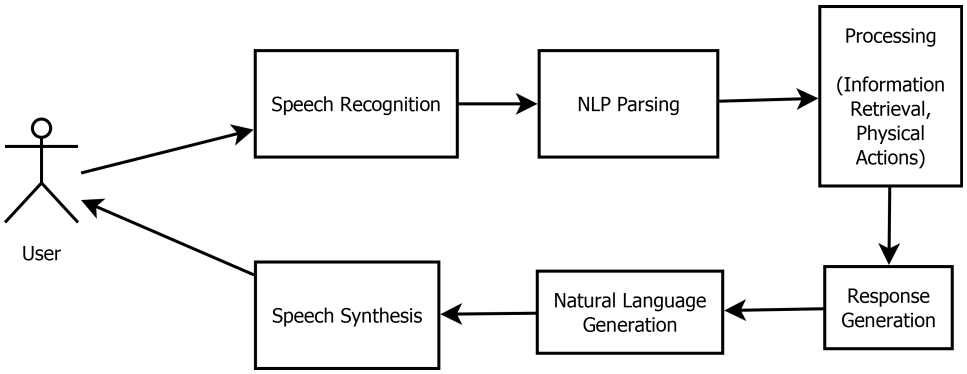

Figure 1.1 The flow of information through a normal interaction with a user in a generalized NLP system

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; for instance, it would be unnecessary for the concierge program described above to be intimately familiar with highly technical jargon. We therefore start with this restricted subproblem, and plan to extend it in future work.

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 techniques which are not goal-directed. These techniques 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. This algorithm performs "simulations", examining different paths that generation could take and iteratively refining the search based on how "good" the outcomes of previous simulations were. This "goodness" is measured by a reward function which returns a numerical estimation of the value of the generation.

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) [4] 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 [5] 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

Given a model of the world, an initial state and a goal state, planning is the problem of finding a sequence of actions that gets you from the initial state to the goal state. 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 unit 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. These problems are usually solved by two broad categories of planning system: state-space planners and plan-space 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.

Plan-space planners work differently. For state-space planners, every node in the planning graph represents a state of the environment, reached through some list of ordered actions. For plan-space planners, each node in the planning graph represents a partial plan, not necessarily

ordered. These partial plans have preconditions and effects; the plan is complete and correct when a node's effects are a superset of the goal state and the same node's preconditions are a subset of the initial state. Plan-space planners begin with a node which has the entire goal state as its effects and its preconditions, then iteratively refine the plan so that the preconditions become a subset of the initial state.

This iterative refinement is done in two ways: resolving open goals and resolving threats. Open goals are preconditions of the plan which we have not yet resolved, and threats are actions which are part of the plan whose effects cancel out the preconditions of other actions which are part of the plan. We resolve open goals by finding actions whose effects contain the preconditions we seek to establish, then adding them to the plan. We resolve threats by introducing ordering constraints; the action which will eliminate a necessary precondition should happen either before that precondition is established or after the action which required it.

Plan-space planners can be faster than state-space planners at determining that a plan will be impossible if some open goals cannot be resolved, but they can also be much slower if continuing to resolve threats causes the plan to balloon in size. In some domains, one failure mode is substantially more likely than the other, and so a choice between the two types of planners might be motivated by domain knowledge.

One popular plan-space planning algorithm is Graphplan[6]. 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. This may fail, and in that case the graph will be extended another level.

2.2.2 Probabilistic Planning and the UCT algorithm

Probabilistic planning is an alternative to Classical Planning when the preconditions required to solve the easier problem do not hold. Probabilistic planning (PP) is done on an MDP, where actions are not required to be deterministic and where we attempt to maximize a reward function. PP does require full observability of the state (this was irrelevant during the classical planning problem, since exploration with deterministic actions is trivial). PP is often described as determining a policy given a current state, rather than generating 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 [7], 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 [8] and UCT [9] 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)[9]. 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:

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

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" where the best action is determined by iteratively exploring the branches of the action tree that previous experiments have shown to be best. 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 [10], 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 [11]. 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 another natural language)[12].

One attempt to make CFGs more realistic for parsing or generating natural languages was to introduce a probabilistic component. This does not make CFGs any more expressive, but it does help to cut down on the large number of potential parses for any given sentence (English being a highly ambiguous language) by providing some direction as to which constructions should be considered first. In a Probabilistic Context Free Grammar (PCFG) [13], 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 naïve way, unable to express these probabilities in terms of 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 $\rightarrow$ 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.

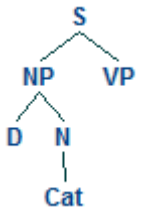

Figure 2.1 An example of an initial tree in a lexicalized tree adjoining grammar

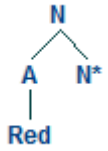

Figure 2.2 An example of an adjoining tree in a lexicalized tree adjoining grammar

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 $\rightarrow$ 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 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.

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

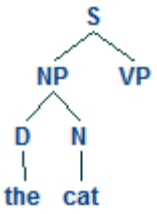

Figure 2.3 Partial tree

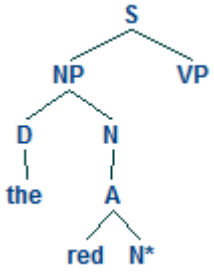

Figure 2.4 Intermediate tree

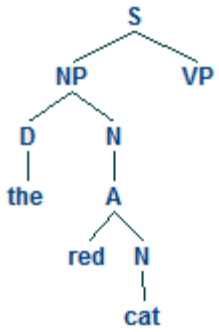

Figure 2.5 Final tree

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

In this section, we discuss previous approaches to NLG, including the two main diverging approaches. Broadly, these are the approaches based in statistical generation and classical planning, respectively. Our work takes an approach which we feel unifies these two differing branches, one of statistical planning. We discuss these two branches in terms of their exemplar systems, Halogen [14] and CRISP [15], respectively. We also discuss prior work in dialog systems, which represent an application of NLG to real-world problems.

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 [16, 17]. 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 [14]. 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. [18, 19].

A second line of attack formalizes NLG as an AI planning problem. SPUD [20], 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* [21]. 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" [22]. Our approach is part of this broad category.

A subcategory of this approach, called *integrated generation*, considers both sentence generation portions of the pipeline together. [15]. This is the approach taken in some modern generators like CRISP [15] and PCRISP [23]. In these generators, the input semantic requirements and grammar are encoded in PDDL [24], which an off-the-shelf classical planner such as Graphplan [25] 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.

3.1.1 Halogen / Nitrogen

The exemplar system for the statistical generation approach is the Halogen / Nitrogen family of systems, based on the idea of ranking thousands of sentences without comprehending their meaning using a special data structure called a ”word lattice”. A word lattice can be thought of as a DAG where each edge is labelled with a single word. The DAG is constructed such that any path from a specially labelled start node to a specially labelled end node is a valid English sentence (at least on the local level). Sentence log likelihoods can be calculated efficiently using this data structure, and the many sentences which can be generated given syntactic unknowns can be ranked efficiently.

Systems like these take a semantic input like that shown in the appendix figure A.4. This input has the advantage of being relatively easy for humans to read, but relatively difficult for machines to create. However, these systems are very efficient and able to produce desirable outputs in many cases. Due to their structure, they are only truly able to consider local dependencies of very short length. In fact, the initial version of Halogen worked on a bigram model and generated sentences like ”I only hires men who is good pilots.” [1]. Obviously, the linguistic model of combining words and choosing the most likely ordering will not always generate sentences which accomplish the desired linguistic goal, and requires eliminating all sentences which do not accomplish the linguistic goal before generation even begins. Later work on these generators included the scoring of the top sentences by more expensive and reliable means, which improved generation quality substantially at the cost of very little time.

Fundamentally, these systems have the advantage of fast and linguistically plausible generation. They take advantage of Zipf’s Law, which causes them to be fairly reliable under certain conditions. When those conditions are not met, however, the generators perform badly; with no real understanding of their output, they cannot adequately correct for bad entries in their word lattice.

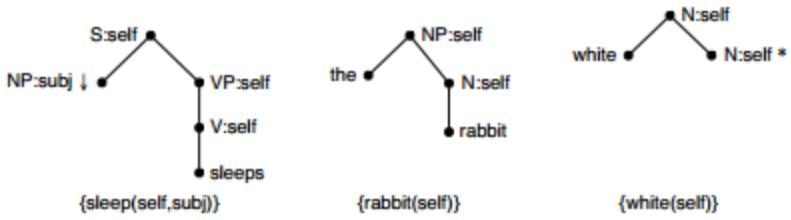

Figure 3.1 An example CRISP grammar. Note the semantic annotations beneath each tree.

3.1.2 CRISP / PCRISP

CRISP is the exemplar system of the latter approach to NLG. CRISP treats the problem of generation as a classical planning problem; the goal state is any state in which the sentence currently being generated matches the desired meaning. Actions add new words to the sentence, and consequently modify CRISP's representation of the meaning of the sentence. This means that CRISP needs to keep a representation of the way in which every entry in the grammar it uses changes the meaning of any sentence in which it is placed. CRISP uses LTAGs as its grammar formalism, which means that these meanings will be phrased as components of tree-rewrite rules. For example, see Figure 3.1, originally published in [21]. This can be problematic for some generation problems, but works well for simple sentences.

CRISP uses graphplan to search for a solution to the planning problem which describes its generation problem. This has the advantage of running very quickly on some types of sentences. It also has the advantage of ignoring irrelevant pieces of the grammar automatically, reducing the number of states that need to be considered substantially. It is also a complete search; if a solution to the generation problem exists using the grammar and lexicon CRISP is provided, CRISP will never fail to find it.

This is also a downside to CRISP; it needs to consider many states for each generation and consequently gets very slow if many steps are needed to generate the sentence. Our experiments show an exponential increase in time to generation as the length of the sentence increases. This makes sense, given that graphplan is an exponential time algorithm in the

length of the plan, and the length of CRISP's plan corresponds directly to the length of the generated sentence.

PCRISP [23] is a later improvement to CRISP which attempts to address the failure of CRISP to consider probabilities in its generation. PCRISP's core advancement is to perform its planning by searching the most probable sentences first. This combines, in some sense, the Halogen / Nitrogen model of generation with the classical planning model. PCRISP accomplishes this by introducing costs into the classical planning model. CRISP considers all adjoins and substitutions to be equal, and will always find the plan which takes the smallest number of actions to complete (as is the nature of graphplan). PCRISP, instead, assigns actions a cost of $\frac{1}{P(s,a)}$, which means that actions which are probable will be investigated first. In the experiments performed in [26], Bauer found that this decreased the incidence of timeouts and increased the speed of generation, but caused an increased instance of generation failures.

PCRISP is the system most like the algorithm we propose in this thesis. However, we feel that using a true probabilistic planning algorithm and working in an MDP rather than including probability in what is normally a stochastic planning algorithm gives us some advantages.

3.2 Dialog Systems

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 [3], 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 [27]. 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 [3]), 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

The STRUCT System

In this section, we introduce our algorithm for natural language generation, entitled "Sentence Tree Realization using UCT" (STRUCT). We begin by describing how we formalize the problem as an MDP. We then explain the modifications we have made to UCT in order to improve its applicability in NLG. We discuss the inputs to STRUCT and explain how those can be specified. We present STRUCT's pseudocode, and describe our method of creating reward functions for our MDPs. We conclude by detailing enhancements to STRUCT external to the MDP planning problem.

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.

As discussed earlier, an MDP is made up of a set of states, a set of actions, a transition function, a reward function, and a discount factor. In our transformation of the NLG problem into an MDP, we must define each of these in such a way that a plan in the MDP becomes a sentence in a natural language. Our set of states, therefore, will be partial sentences. Each

partial sentence possible in the language defined by the generation problem's grammar is a state in our MDP. There are, therefore, an infinite number of states, since TAG adjoins can be repeated indefinitely. Our set of actions will correspond to a single TAG substitution or adjoin at a particular valid location in the tree. Since we are using LTAGs in this work, this corresponds to adding a single word to the partial sentence. This definition makes our transition function intuitive; it is simply the mapping between a partial sentence / action pair and the partial sentence which results from the TAG adjoin / substitution. Our reward function, intuitively, describes how close we are to the goal semantic meaning. The construction of this function is nontrivial and will be described in detail later. We use a discount factor of 1; this is motivated by our willingness to tolerate lengthy sentences in order to ensure we get a generated result that matches our goal.

4.2 Modifications to UCT

We made three modifications to UCT in order to increase its usability for NLG tasks. This is motivated by the difference between normal use of UCT and the NLG application's needs. UCT is normally used to search for sparse rewards in a tree of finite depth. Our reward functions, which we will describe later, provide feedback after every action, and our tree is of potentially infinite depth. Consequently, we have implemented a depth limit on our tree search. We perform a number of actions in the default policy stage of UCT equal to a parameter d. We then evaluate our reward function at the state we reach and push that reward up to the root as usual.

Second, UCT is normally used in an environment where the transition function is nondeterministic. UCT is often used in an adversarial environment, like chess or Go, and consequently searches for the best average case from a given state. 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.

Third, UCT normally ends the tree policy phase by selecting an action from the unexplored actions list randomly. Since we can reasonably expect that in the domain of NLG, there could easily be a better ordering of those unexplored actions which can be known before selecting them. For instance, some words might be substantially more common in practice than others, so we should search trees which use those words before other trees. STRUCT allows for a heuristic to order the nodes it searches.

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.

4.3 STRUCT

4.3.1 Inputs to STRUCT

STRUCT takes four inputs and generates a single sentence. These inputs are a grammar, a lexicon, a worldfile, and a communicative goal.

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.

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.

A worldfile is a list of all statements which are true in the world surrounding our generation. Matching entity names refer to the same entity. Since we use the grounding assumption to simplify our search, we need to include all statements which are true.

The communicative goal is just a list of propositions with dummy entity names. These proposition names follow the same rules as those in worldfiles. 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.

4.3.2 The STRUCT Algorithm

We now describe our approach, called Sentence Tree Realization with UCT (STRUCT). We begin with the state representing the empty tree. The empty tree contains no semantic meaning whatsoever, and is made up of a single nonterminal, "S". Recall that the actions which can be taken from every state are tree adjoin / tree substitution actions described in the LTAG / PLTAG which defines the language we search over. Since trees in LTAGs are associated with individual words, actions 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. These adjoin operations can also define arguments for the word in question. For instance, the location of the subject and object of a verb relative to the verb itself. In this way, the tree shown in Figure **??** defines the "subject" and "object" of the verb specified. Further, since all recursive phenomena are encoded in auxiliary trees, we factor recursion from the domain of dependencies [26], and can add auxiliary trees to partial sentences without breaking

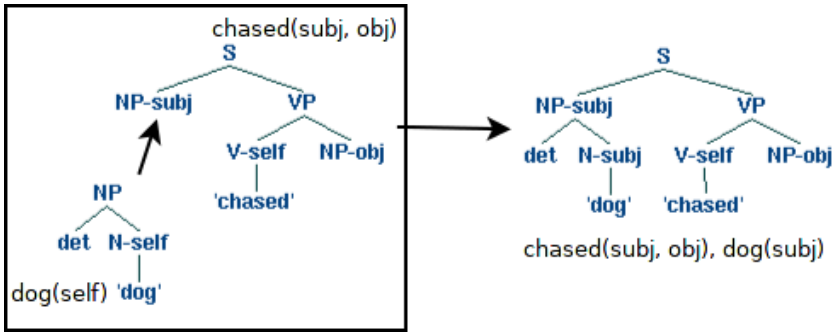

Figure 4.1 An example tree substitution operation in STRUCT.

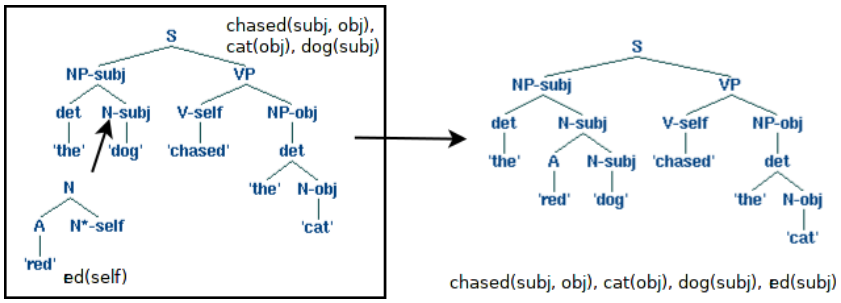

Figure 4.2 An example tree adjoining operation in STRUCT.

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. This also allows us to generate a complete and valid sentence quickly. This allows STRUCT to operate as an anytime algorithm, described further below.

From this point, we simply use our modified version of UCT as described above to plan in the MDP from the initial state specified above. See Figure 4.3 for the general algorithm.

Figure 4.3 The STRUCT Algorithm.

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

Ensure: Generated sentence tree

$bestSentence \leftarrow$ nil
while User has not interrupted generation **do**
  $state \leftarrow$ empty sentence tree
  while $state$ not terminal **do**
    for $numTrials$ **do**
      $testState \leftarrow state$
      $currentDepth \leftarrow 0$
      if $testState$ has unexplored actions **then**
        Apply one unexplored PLTAG production chosen uniformly at random to $testState$
        $currentDepth$++
      end if
      while $currentDepth < maxDepth$ **do**
        Apply PLTAG production selected by tree policy (Equation 2.1)
        $currentDepth$++
      end while
      calculate reward for $testState$
      associate reward with first action taken
    end for
    $state \leftarrow$ maximum reward $testState$
    if $state$ score $>$ $bestSentence$ score **and** $state$ has no nonterminal leaf nodes **then**
      $bestSentence \leftarrow state$
    end if
  end while
end while
return $bestSentence$

4.4 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. There are, of course, many possible reward functions which fulfill all of these requirements.

Since the reward function is specific to the individual generation problem and guides the search for the correct meaning in the given world, we designed a method for automatically constructing a reward function from these inputs.

We found two reward functions which will return satisfactory results on a variety of experiments, but have complementary properties where they differ. The first reward function which we found to be 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 runs in $O(N!)$ time, where N is the number of entities present in the sentence. The pseudocode for this algorithm is present in 4.4

Of course, if we can remove the permutation from this algorithm, we can get much better performance. We know that in most cases, STRUCT won't need to consider combinations of entity mapping and can get by with simply considering whether an entity, by itself, could satisfy the goal. To this end, we present STRUCT$_b$, in Figure 4.5.

Our experiments will show that STRUCT$_b$ has better performance than STRUCT$_a$, but that STRUCT$_a$ generates better sentences in some domains. This is due to the occasions on which permutation considerations apply; for instance, we'll need those considerations if we wish to

Figure 4.4 The STRUCT$_a$ reward function.

Require: State s, World w, Goal g

Ensure: Numerical reward

$bestReward \leftarrow$ -INF
$countValidPermutations \leftarrow 0$
for permutation p of entities in the world **do**
  $currentReward \leftarrow 0.0$
  if all $prop$ in s, mutated by p, are in w **then**
    $currentReward \leftarrow 10.0$
    $countValidPermutations++$
  end if
  if g is satisfied by s mutated by p **then**
    $currentReward += 50.0$
  end if
  for entity e in p **do**
    if e is uniquely described by s, mutated by p **then**
      $currentReward += 1.0$
    end if
  end for
  if $currentReward > bestReward$ **then**
    $bestReward \leftarrow currentReward$
  end if
end for
if $countValidPermutations == 0$ **then**
  return -INF
end if
return $bestReward/countValidPermutations$

Figure 4.5 The STRUCT$_b$ reward function.

Require: State s, World w, Goal g

Ensure: Numerical reward

```
1:  maxReward ← 0
2:  for entity e in the world do
3:      currentReward ← 0
4:      possibleEntities ← 0
5:      for proposition p in s do
6:          for entity e_p in p do
7:              if p could be true if e is e_p then
8:                  possibleEntities++
9:              end if
10:             if g could be true if e is e_p then
11:                 currentReward+ = 5
12:             end if
13:         end for
14:     end for
15:     if possibleEntities == 0 then
16:         currentReward ← −INF
17:     else
18:         currentReward = currentReward/possibleEntities
19:     end if
20:     if currentReward > maxReward then
21:         maxReward = currentReward
22:     end if
23: end for
24: return maxReward
```

generate sentences like “The dog which chased the black cat ran after the white cat,” because we’ll need to care about which cat the dog chased and which it ran after.

4.5 Enhancements External to the MDP

The algorithm as described above performs well, but there are enhancements external to UCT which can increase performance and provide additional positive qualities.

4.5.1 Execution as an Anytime Algorithm

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

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.

4.5.2 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 [23]. 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.

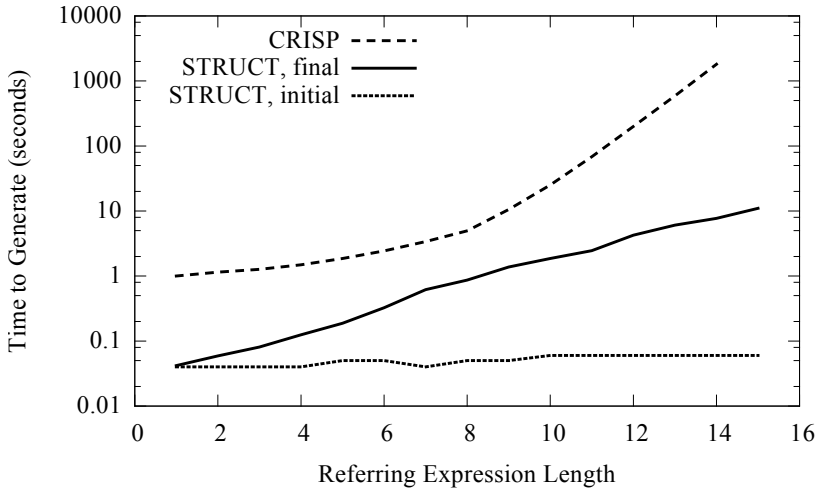

Figure 5.1 Experimental comparison between STRUCT and CRISP: Generation time vs. length of referring expression

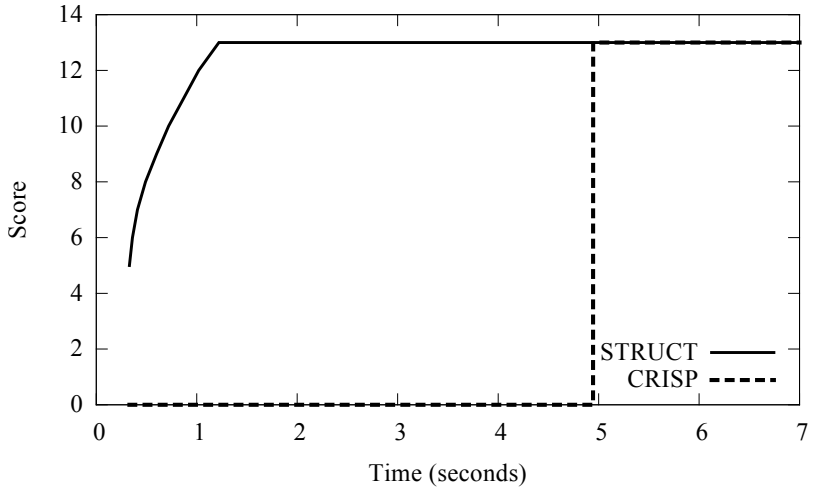

Figure 5.2 Experimental comparison between STRUCT and CRISP: Score of best solution vs time.

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 Referring Expressions

We first evaluate CRISP and STRUCT on their ability to generate referring expressions. We follow prior work ([21]) in our initial experiment design. We consider a series of sentence generation problems which require the planner to generate a sentence like "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. 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_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. 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 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

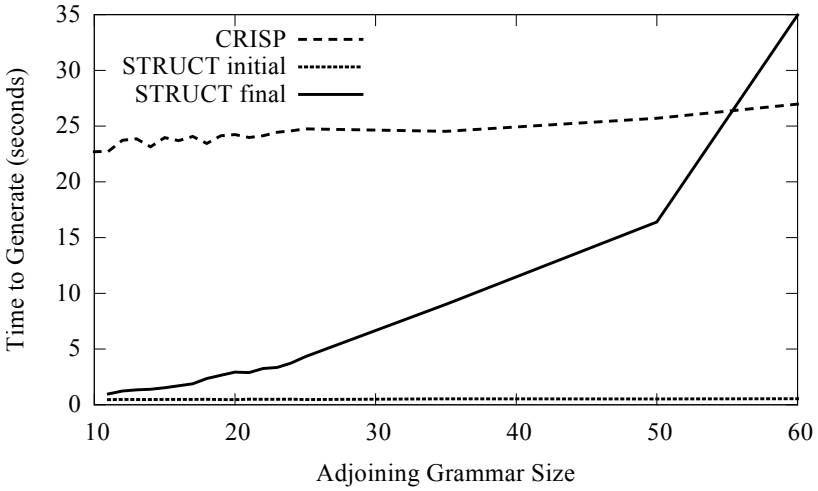

Figure 5.3 Effect of grammar size

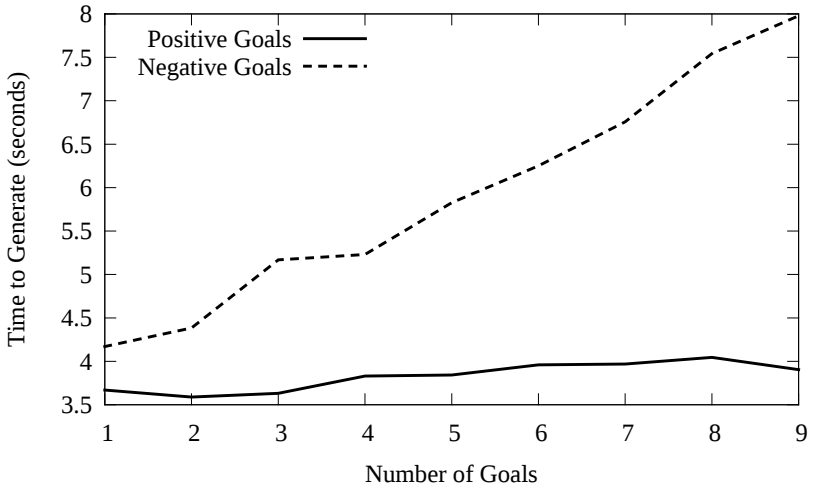

Figure 5.4 Effect of multiple and negated goals

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 [21], 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 [28], also profiled in [21], 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, even 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 [29] in their attempt to create a TAG which appropriately represents the English language. See Figure **??** for a visual explanation.

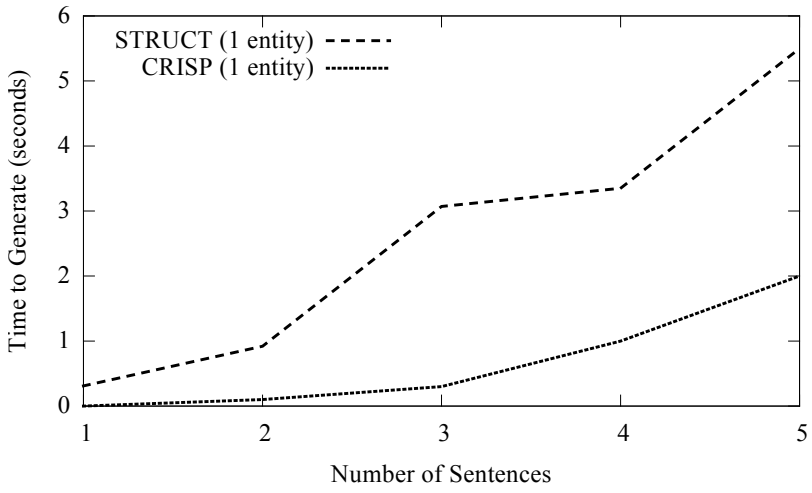

Figure 5.5 Time to generate sentences with conjunctions with one entitiy ("The man sat and the girl sat and ....")

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". Here, we follow prior work in our experiment design [21].

As we can see in Figures 5.2.4, 5.2.4, and 5.2.4, 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.

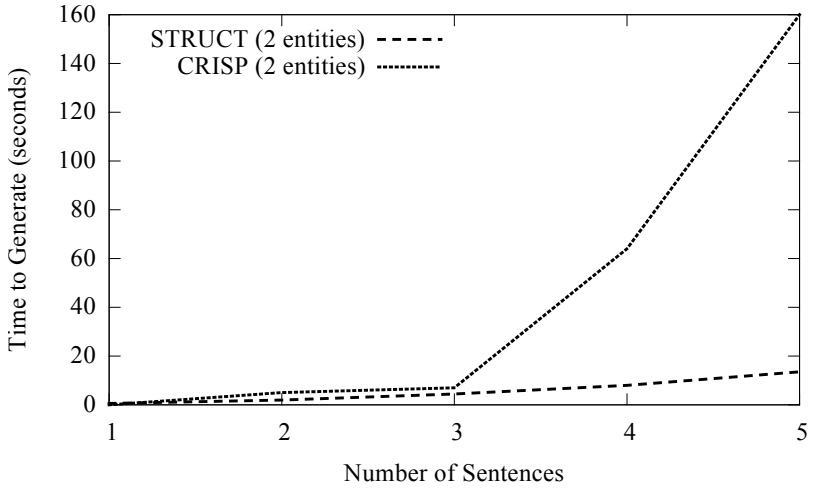

Figure 5.6 Time to generate sentences with conjunctions with two entities (”The dog chased the cat and ...”)

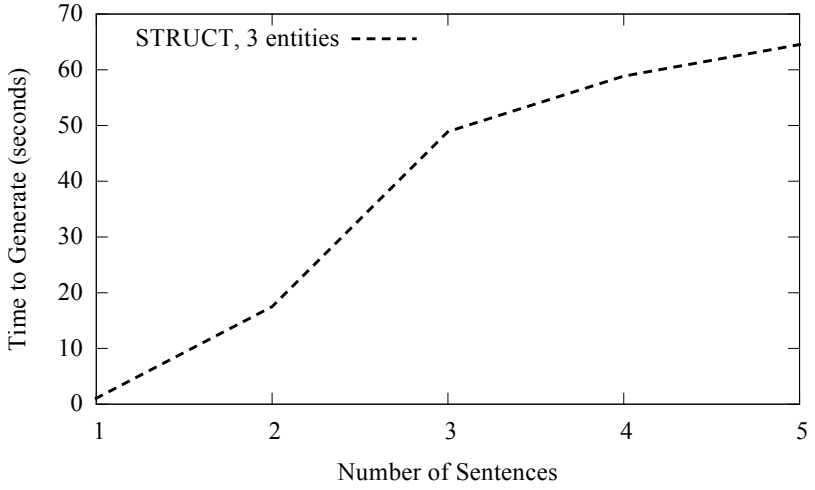

Figure 5.7 Time to generate sentences with conjunctions with three entities (”The man gave the girl the book and ....”)

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.

We also compare our results to those presented in [21] for CRISP with the FF Planner. Koller et al attempted to generate sentences with three entities and failed to find a result within their 4 GB memory limit. As you can see, CRISP generates a result slightly faster than STRUCT when we are working with a single entity, but works much much slower for two entities and cannot generate results for a third entity. According to Koller's findings, this is because the search space grows by a factor of the universe size with the addition of another entity [21].

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 5.2.5).

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.

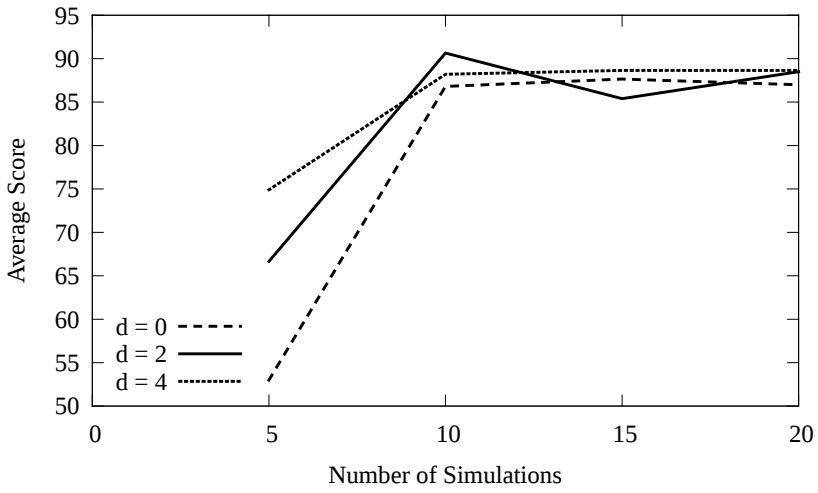

Figure 5.8 Effect of parameter variations on the STRUCT solution.

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

In conclusion, we will compare STRUCT to established NLG methods, discuss other strengths and weaknesses of STRUCT, and describe directions for future work.

6.1 Comparison to Competing NLG Methods

As discussed in Chapter 3, there are two other popular methods of solving the NLG problem. In the first, the NLG problem is treated as a classical planning problem and solved using an off-the-shelf planner. In the second, the NLG problem is solved using an overgeneration strategy which reduces to a dynamic programming approach. See chapter 3 for a more detailed explanation of these methods.

We believe that we have produced an algorithm which unifies these two approaches. We have fused the probabilistic reasoning and domain knowledge use from the probabilistic ranking method with the planning problem conversion and structured approach to semantics from the classical planning approach. We have the advantage of partial goal satisfaction from "overgenerate and rank" and the advantage of explicit semantic meaning specification and output from classical planning.

We have avoided the all-or-nothing weakness of classical planning, where a classical planner cannot emit a sentence which does not optimally satisfy the goal. Yet, STRUCT allows generation of complex goals, which would be intractable with a probabilistic ranking generator. We have avoided the requirement that we specify a large percentage of our output as input, which is a problem with probabilistic ranking.

In short, we have constructed an algorithm with many of the strengths of both classical planning and probabilistic ranking, and few of the weaknesses of either.

6.2 STRUCT: Strengths and Weaknesses

STRUCT's prime strength is its ability to partially satisfy goals in cases where perfectly correct generation is impossible, either because of conflicting goals or because of a grammar which does not allow for all goals. This allows STRUCT to be used in situations even without perfect knowledge of the domain, since STRUCT can recover from some of the issues that come from unknown domains (like a grammar which is too small or insufficient knowledge of the world), and continue attempting to generate close-to-optimal output where other generators would either fail after churning on the problem for a long time, or emit something nonsensical.

Another important strength of STRUCT is its anytime nature; something which we have not seen done before. STRUCT is capable of creating an approximate solution very quickly (usually less than 0.5 seconds), and that approximate solution can be emitted if the user desires a response very quickly. The solution will be iteratively improved until it reaches a sentence which fulfills all communicative goals given.

One important weakness of STRUCT is the rapid increase in generation time past certain limits. STRUCT has trouble generating referring expressions of length 10 or more, and has trouble generating sentences which adjoin more than about 5 verbs.

STRUCT also is limited by its grammar. It is difficult to produce an LTAG grammar for English, and any such grammar will, by nature, overgenerate (the language specified by the grammar will contain some constructs not acceptable in English). It is best to build a grammar which undergenerates (the language specified by the grammar is a subset of acceptable English), but that can be difficult without knowledge of which constructs exactly will be necessary.

6.3 Future Work

Due to some peculiarities of the python interpreter, we were unable to efficiently parallelize UCT during the search phase of generation. This would be relatively straightforward using a worker pool solution, and future work could easily shrink the amount of time spent in that search drastically.

The dialog system which we have implemented is quite rudimentary (simpler even than NJFun [3]), and could be improved to use more intelligent parsing techniques than simple keyword searching.

In conclusion, we have presented an algorithm which performs Natural Language Generation in a well-principled way, unifying two popular schools of thought regarding NLG. Experimental evaluation shows that it performs as well as the state-of-the-art in the field, and its nature allows for a substantially greater set of use cases than either of the popular schools it synthesizes.

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)"
}
```

Figure A.4 Example input for Halogen; appears in [1]

```
(A / Ihave the quality of beingl
:DOMAIN (P / [procurel
:AGENT (A2 / [American[)
:PATIENT (G / [gun, arml))
:RANGE (E / [easy, effortless[))
```

LIST OF REFERENCES

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

[2] L. Goasduff and C. Pettey, "Gartner says worldwide smartphone sales soared in fourth quarter of 2011 with 47 percent growth," *Gartner, Inc*, vol. 15, 2012.

[3] 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.

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

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

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

[7] 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.

[8] 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.

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

[10] J. W. Cowan, *The complete Lojban language*. Logical Language Group, 1997.

[11] E. Klein, "Context free grammars," 2012.

[12] D. Jurafsky, J. H. Martin, A. Kehler, K. Vander Linden, and N. Ward, *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*, vol. 2. MIT Press, 2000.

[13] F. Jelinek, J. D. Lafferty, and R. L. Mercer, *Basic methods of probabilistic context free grammars*. Springer, 1992.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.

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