Chapter 1

Introduction

With the increasing frequency of natural language interfaces for consumer products, natural language generation (NLG) is becoming more and more important in industry. Consequently, a consistent and reliable method for creating a natural language generation system would be of value, especially if the system's computational requirements were small enough that the system could be embedded in consumer electronics. In this work, we propose a system based in statistical planning which serves as a general-purpose natural language generation system. As a part of this work, we propose a method for specifying a grammar based on tree-adjoining grammars using a modified UCT algorithm.

1.1 Natural Language Generation Systems

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

Chapter 2

Background

2.1 Tree Adjoining Grammars

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

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

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

And then perform the substitution:

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

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

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

2.2 Grammar Specification

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

2.3 Lexicon Specification

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

2.4 Communicative Goal Specification

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

2.5 The UCT Algorithm

UCT(Upper Confidence bound applied to Trees) (Kocsis, Szepesvari 2006) is a planning algorithm that takes advantage of Monte-Carlo sampling to narrow down the search space during each iteration as an action is chosen. This algorithm provides many benefits in that it can handle large search-spaces, stop at any point and return back the best answer based on the Monte-Carlo simulations done till that point, and can have sub-optimal outputs. The last benefit may actually seem like a weakness, however, in natural language generation, it is beneficial to not have a guarantee of generating the best sentence each time. If it were the case that our planner was optimal, the output would always be the same given matching inputs which is not a realistic output.

The goal of the UCT algorithm is to select the optimal or near-optimal sequence of actions from the start to end state. This is done in an iterative manner with a single action chosen at each iteration. In this the algorithm, the simulations performed in the main loop are used to estimate the value of each action. These simulations can be broken down into 4 steps.

2.5.1 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 assigning each of the actions a probability based on its expected value as well as how often it has been executed for the number of times the current state has been reached. The exact formula used is the following:

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

where s is the current state, a is the action from the current state, c is the exploration constant that must be tuned empirically in practice, n(s) is the number of times state s has been encountered, and n(s,a) is the number of times action a was taken in state s.

2.5.2 State Expansion

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

2.5.3 Default Policy

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

2.5.4 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.

Once sufficient simulations have been run, the action with the largest value expectation is chosen. The expectation of the value of an action a_i is computed by taking the average of all of the values computed for each simulation that began with action a_i.

Figure **??** provides an illustration of the UCT algorithm. It shows a view of the state search tree at the end of an intermediate Monte-Carlo simulation. The root of this tree represents that current state and its child nodes are the possible actions that UCT will be choosing from. As the figure shows, the tree policy contains all of the node that have been expanded at least once. During the tree policy, we will traverse down the tree until we hit a leaf or a state containing an open actions. In this example, the traversal follows the blue line and stops at the white node on the left since it has an open action. Next, for the State Expansion step, an open action is chosen at random and is denoted with the star shape in the figure. Then, for the Default Policy, random actions are taken until the depth limit is reached or there are no more open actions. Each of the random actions taken are denoted with a diamond shape and the terminal node is denoted as a square in the figure. Once this happens, the reward for the square node is calculated(in this case 1) and propagated up to the nodes in the tree policy along the blue line. This process is repeated for the number of iterations specified when starting UCT and at each iteration, there is a possibility of adding one more state to the tree policy.

The modification made to UCT in this work deals with the available actions to choose from. In the default policy, instead of choosing an action at random from all possible actions, the next action is only chosen from the new actions that resulted from executing the previous action. The motivation for this change is that by limiting the number of actions allowed, the final value computed at the end of the action sequence will be a better representation of the value expectation for the first action in the sequence. This provides the added benefit that the algorithm is now selecting from a pruned search space allowing for faster execution.

Chapter 3

Related Work

3.1 Natural Language Generation Systems

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

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

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

3.2 Grammar Representation

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

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

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

3.3 NLG Applications

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

Chapter 4

Framework

Chapter 5

Experiments

Chapter 6

Conclusion