Scaling a Natural Language Generation System

Jonathan Pfeil
Department of EECS
Case Western Reserve University
Cleveland, OH, USA
jonathan.pfeil@case.edu

Soumya Ray
Department of EECS
Case Western Reserve University
Cleveland, OH, USA
sray@case.edu

Abstract

A key goal in natural language generation (NLG) is to enable fast generation even with large vocabularies, grammars and worlds. In this work, we build upon a recently proposed NLG system, Sentence Tree Realization with UCT (STRUCT). We describe four enhancements to this system: (i) pruning the grammar based on the world and the communicative goal, (ii) intelligently caching and pruning the combinatorial space of semantic bindings, (iii) reusing the lookahead search tree at different search depths, and (iv) learning and using a search control heuristic. We evaluate the resulting system on three datasets of increasing size and complexity, the largest of which has a vocabulary of about 10K words, a grammar of about 32K lexicalized trees and a world with about 11K entities and 23K relations between them. Our results show that the system has a median generation time of 8.5s and finds the best sentence on average within 25s. These results are based on a sequential, interpreted implementation and are significantly better than the state of the art for planning-based NLG systems.

1 Introduction and Related Work

We consider the restricted natural language generation (NLG) problem (Reiter and Dale, 1997): given a grammar, lexicon, world and a communicative goal, output a valid sentence that satisfies this goal. Though restricted, this problem is still challenging when the NLG system has to deal with the large probabilistic grammars of natural language, large knowledge bases representing realistic worlds with many entities and relations between them, and complex communicative goals.

Prior work has approach NLG from two directions. One strategy is over-generation and ranking, in which an intermediate structure generates many candidate sentences which are then ranked according to how well they match the goal. This includes systems built on chart parsers (Shieber, 1988; Kay, 1996; White and Baldridge, 2003), systems that use forest architectures such as HALogen/Nitrogen, (Langkilde-Geary, 2002), systems that use tree conditional random fields (Lu et al., 2009), and newer systems that use recurrent neural networks (Wen et al., 2015b; Wen et al., 2015a). Another strategy formalizes NLG as a goal-directed planning problem to be solved using an automated planner. This plan is then semantically enriched, followed by *surface realization* to turn it into natural language. This is often viewed as a *pipeline* generation process (Reiter and Dale, 1997).

An alternative to pipeline generation is *integrated generation*, in which the sentence planning and surface realization tasks happen simultaneously (Reiter and Dale, 1997). CRISP (Koller and Stone, 2007) and PCRISP (Bauer and Koller, 2010) are two such systems. These generators encode semantic components and grammar actions in PDDL (Fox and Long, 2003), the input format for many off-the-shelf planners such as Graphplan (Blum and Furst, 1997). During the planning process a semantically annotated parse is generated alongside the sentence, preventing ungrammatical sentences and structures that cannot be realized. PCRISP builds upon the CRISP system by incorporating grammar probabilities as costs in an off-the-shelf metric planner (Bauer and Koller, 2010). Our work builds upon the Sentence Tree Realization with UCT (STRUCT) system (McKinley and Ray, 2014), described further in the next section. STRUCT performs integrated generation by formalizing the generation problem as planning in a Markov decision process (MDP), and using a probabilistic planner to solve it.

Results reported in previous work (McKinley and Ray, 2014) show that STRUCT is able to correctly generate sentences for a variety of communicative goals. Further, the system scaled better with grammar size (in terms of vocabulary) than CRISP. Nonetheless, these experiments were performed with toy grammars and worlds with artificial communicative goals written to test specific experimental variables in isolation. In this work, we consider the question: can we enable STRUCT to scale to realistic generation tasks? For example, we would like STRUCT to be able to generate any sentence from the Wall Street Journal (WSJ) corpus (Marcus et al., 1993). We describe four enhancements to the STRUCT system: (i) pruning the grammar based on the world and the communicative goal, (ii) intelligently caching and pruning the combinatorial space of semantic bindings, (iii) reusing the lookahead search tree at different search depths, and (iv) learning and using a search control heuristic. We call this enhanced version Scalable-STRUCT (S-STRUCT). In our experiments, we evaluate S-STRUCT on three datasets of increasing size and complexity derived from the WSJ corpus. Our results show that even with vocabularies, grammars and worlds containing tens of thousands of constituents, S-STRUCT has a median generation time of 8.5s and finds the best sentence on average within 25s, which is significantly better than the state of the art for planning-based NLG systems.

2 Background: LTAG and STRUCT

STRUCT uses an MDP (Puterman, 1994) to formalize the NLG process. The states of the MDP are semantically-annotated partial sentences. The actions of the MDP are defined by the rules of the grammar. STRUCT uses a probabilistic lexicalized tree adjoining grammar (PLTAG).

Tree Adjoining Grammars (TAGs) (Figure 1) consist of two sets of trees: initial trees and auxiliary (adjoining) trees. An initial tree can be applied to an existing sentence tree by replacing a leaf node whose label matches the initial tree's root label in an action called "substitution". Auxiliary trees have a special "foot" node whose label matches the label of its root, and uses this to encode recursive language structures. Given an existing sentence tree, an auxiliary tree can be applied in a three-step process called "adjunction". First, an adjunction site is selected from the sentence tree; that is, any node whose label matches that of the auxiliary tree's root and foot. Then, the subtree rooted by the adjunction site is removed from the sentence tree and substituted into the foot node of the auxiliary tree. Finally, the modified auxiliary tree is substituted back into the original adjunction location. LTAG is a variation of TAG in which each tree is associated with a lexical item known as an anchor (Joshi and Schabes, 1997). Semantics can be added to an LTAG by annotating each tree with compositional lambda semantics that are unified via β-reduction (Jurafsky and Martin, 2000). A PLTAG associates probabilities with every tree in the LTAG and includes probabilities for starting a derivation, probabilities for substituting into a specific node, and probabilities for adjoining at a node, or not adjoining.

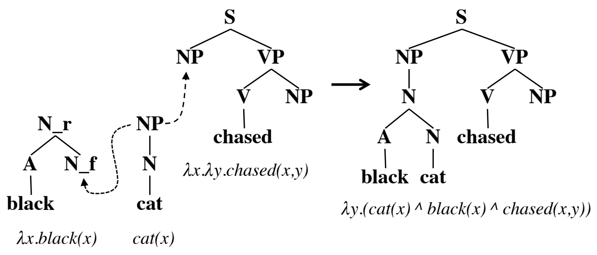


Figure 1: LTAG examples: initial tree (chased), substitution (cat), and adjunction (black)

The STRUCT reward function is a measure of progress towards the communicative goal as measured by the overlap with the semantics of a partial sentence. It gives positive reward to subgoals fulfilled and gives negative reward for unbound entities, unmet semantic constraints, sentence length, and ambiguous entities. Therefore, the best sentence for a given goal is the shortest unambiguous sentence which fulfills the communicative goal and all semantic constraints. The transition function of the STRUCT MDP assigns the total probability of selecting and applying an action in a state to transition to the next, given by the action's probability in the grammar. The final component of the MDP is the discount factor, which is set to 1. This is because with lexicalized actions, the state does not loop, and the algorithm may need to generate long sentences to match the communicative goal.

STRUCT uses a modified version of the probabilistic planner UCT (Kocsis and Szepesvári, 2006), which can generate near-optimal plans with a time complexity independent of the state space size. UCT's online planning happens in two steps: for each action available, a lookahead search tree is constructed to estimate the action's utility. Then, the best available action is taken and the procedure is repeated. If there are any unexplored actions, UCT will choose one according to an "open action policy" which samples PLTAGs without replacement. If no unexplored actions remain, an action a is chosen in state s according to the "tree policy" which maximizes Equation 1.

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

Here $Q(s,a)$ is the estimated value of a, computed as the sum of expected future rewards after (s,a). $N(s,a)$ and $N(s)$ are the visit counts for s and (s,a) respectively. c is a constant term controlling the exploration/exploitation trade off. After an action is chosen, the policy is rolled out to depth D by repeatedly sampling actions from the PLTAG, thereby creating the lookahead tree.

UCT was originally used in an adversarial environment, so it selects actions leading to the best average reward; however, language generation is not adversarial, so STRUCT chooses actions leading to the best overall reward instead.

Algorithm 1 S-STRUCT Algorithm

```
Require: Grammar $R$, World $W$, Goal $G$, num trials $N$, lookahead depth $D$, timeout $T$
 1: $^\dagger R \leftarrow pruneGrammar(R)$
 2: $state \leftarrow$ empty state
 3: $uctTree \leftarrow$ new search tree at $state$
 4: while $state$ not terminal and time < T do
 5:     $^\dagger uctTree \leftarrow getAction(uctTree, N, D)$
 6:     $state \leftarrow uctTree.state$
 7: end while
 8: return $extractBestSentence(uctTree)$
```

The modified STRUCT algorithm presented in this paper, which we call Scalable-STRUCT (S-STRUCT), is shown in Algorithm 1. If the changes described in the next section (lines marked with †) are removed, we recover the original STRUCT system.

3 Scaling the STRUCT system

In this section, we describe five enhancements to STRUCT that will allow it to scale to real world NLG tasks. Although the implementation details of these are specific to STRUCT, all but one (reuse of the UCT search tree) could theoretically be applied to any planning-based NLG system.

Algorithm 2 getAction (Algorithm 1, line 5)

```
Require: Search Tree $uctTree$, num trials $N$, lookahead depth $D$, grammar $R$
 1: for $N$ do
 2:     $node \leftarrow uctTree$
 3:     if $node.state$ has unexplored actions then
 4:         $^\dagger action \leftarrow$ pick with open action policy
 5:     else
 6:         $^\dagger action \leftarrow$ pick with tree policy
 7:     end if
 8:     $^\dagger node \leftarrow applyAction(node, action)$
 9:     $depth \leftarrow 1$
10:     while $depth < D$ do
11:         $action \leftarrow$ sample PLTAG from $R$
12:         $^\dagger node \leftarrow applyAction(node, action)$
13:         $reward \leftarrow calcReward(node.state)$
14:         propagate $reward$ up $uctTree$
15:         $depth \leftarrow depth + 1$
16:     end while
17: end for
18: $uctTree \leftarrow$ best child of $uctTree$
19: return $uctTree$
```

3.1 Grammar Pruning

It is clear that for a given communicative goal, only a small percentage of the lexicalized trees in the grammar will be helpful in generating a sentence. Since these trees correspond to actions, if we prune the grammar suitably, we reduce the number of actions our planner has to consider.

Algorithm 3 pruneGrammar (Algorithm 1, line 1)

```
Require: Grammar $R$, World $W$, Goal $G$
 1: $G' \leftarrow \emptyset$
 2: for $e \in G$.entities do
 3:     $G' \leftarrow G' \cup referringExpression(e, W)$
 4: end for
 5: $R' \leftarrow \emptyset$
 6: for $tree \in R$ do
 7:     if $tree$ fulfills semantic constraints or $tree$.relations $\subseteq G'$.relations then
 8:         $R' \leftarrow R' \cup \{tree\}$
 9:     end if
10: end for
11: return $R'$
```

There are four cases in which an action is relevant. First, the action could directly contribute to the goal semantics. Second, the action could satisfy a semantic constraint, such as mandatory determiner adjunction which would turn "cat" into "the cat" in Figure 1. Third, the action allows for additional beneficial actions later in the generation. An auxiliary tree anchored by "that", which introduces a relative clause, would not add any semantic content itself. However, it would add substitution locations that would let us go from "the cat" to "the cat that chased the rabbit" later in the generation process. Finally, the action could disambiguate entities in the communicative goal. In the most conservative approach, we cannot discard actions that introduce a relation sharing an entity with a goal entity (through any number of other relations), as it may be used in a referring expression (Jurafsky and Martin, 2000). However, we can optimize this by ensuring that we can find at least *one*, instead of *all*, referring expressions.

This grammar pruning is "lossless" in that, after pruning, the full communicative goal can still be reached, all semantic constraints can be met, and all entities can be disambiguated. However it is possible that the solution found will be longer than necessary. This can happen if we use two separate descriptors to disambiguate two entities where one would have sufficed. For example, we could generate the sentence "the black dog chased the red cat" where saying "the large dog chased the cat" would have sufficed (if "black", "red", and "large" were only included for disambiguation purposes).

We implement the pruning logic in the $pruneGrammar$ algorithm shown in Algorithm 3. First, an expanded goal G' is constructed by explicitly solving for a referring expression for each goal entity and adding it to the original goal. The algorithm is based on prior work (Bohnet and Dale, 2005) and uses an alternating greedy search, which chooses the relation that eliminates the most distractors, and a depth-first search to describe the entities. Then, we loop through the trees in the grammar and only keep those that can fulfill semantic constraints or can contribute to the goal. This includes trees introducing relative clauses.

3.2 Handling Semantic Bindings

As a part of the reward calculation in Algorithm 4, we must generate the valid bindings between the entities in the partial sentence and the entities in the world (line 2). We must have at least one

Algorithm 4 calcReward (Algorithm 2, line 13)

```
Require: Partial Sentence $S$, World $W$, Goal $G$
 1: $score \leftarrow 0$
 2: $^{\dagger}B \leftarrow getValidBindings(S, W)$
 3: if $|B| > 0$ then
 4:     $^{\dagger}m \leftarrow getValidBinding(S, G)$
 5:     $S \leftarrow$ apply $m$ to $S$
 6:     $score \mathrel{+}= C_1\,|G.\text{relations} \cap S.\text{relations}|$
 7:     $score \mathrel{-}= C_2\,|G.\text{conds} - S.\text{conds}|$
 8:     $score \mathrel{-}= C_3\,|G.\text{entities} \ominus S.\text{entities}|$
 9:     $score \mathrel{-}= C_4\,|S.\text{sentence}|$
10:     $score \mathrel{/}= C_5\,|B|$
11: end if
12: return $score$
```

valid binding, as this indicates that our partial sentence is factual (with respect to the world); however, more than one binding means that the sentence is ambiguous, so a penalty is applied. Unfortunately, computing the valid bindings is a combinatorial problem. If there are N world entities and K partial sentence entities, there are $\binom{N}{K}$ bindings between them that we must check for validity. This quickly becomes infeasible as the world size grows.

Algorithm 5 getValidBindings (Alg. 4, line 2)

```
Require: Partial Sentence $S$, World $W$
 1: $validBindings \leftarrow \emptyset$
 2: $queue \leftarrow prevBindings$ if exists else $[\emptyset]$
 3: while $|queue| > 0$ do
 4:     $b \leftarrow queue.\text{pop()}$
 5:     $S' \leftarrow$ apply binding $b$ to $S$
 6:     if $S', W$ consistent and $S'$.entities all bound then
 7:         $validBindings.\text{append}(b)$
 8:     else if $S', W$ consistent then
 9:         $freeS \leftarrow$ unbound $S'$.entities
10:         $freeW \leftarrow W$.entities not in $b$
11:         for $e_s, e_w \in freeS \times freeW$ do
12:             $queue.\text{push}(b \cup \{e_s \rightarrow e_w\})$
13:         end for
14:     end if
15: end while
16: return $validBindings$
```

Instead of trying every binding, we use the procedure shown in Algorithm 5 to greatly reduce the number of bindings we must check. Starting with an initially empty binding, we repeatedly add a single $\{sentenceEntity \rightarrow worldEntity\}$ pair (line 12). If a binding contains all partial sentence entities and the semantics are consistent with the world, the binding is valid (lines 6-7). If at any point, a binding yields partial sentence semantics that are *inconsistent* with the world, we no longer need to consider any bindings which it is a subset of (when condition on line 8 is false, no children expanded). The benefit of this bottom-up approach is that when an inconsistency is caused by adding a mapping of partial sentence entity e_1 and world entity e_2, *all* of the $\binom{N-1}{K-1}$ bindings containing $\{e_1 \rightarrow e_2\}$ are ruled out as well. This procedure is especially effective in worlds/goals with low ambiguity (such as real-world text).

We further note that many of the binding checks are repeated between action selections. Because our sentence semantics are conjunctive, entity specifications only get *more* specific with additional relations; therefore, bindings that were invalidated earlier in the search procedure can never again become valid. Thus, we can cache and reuse valid bindings from the previous partial sentence (line 2). For domains with very large worlds (where most relations have no bearing on the communicative goal), most of the possible bindings will be ruled out with the first few action applications, resulting in large computational savings.

3.3 Reusing the Search Tree

The STRUCT algorithm constructs a lookahead tree of depth D via policy rollout to estimate the value of each action. This tree is then discarded and the procedure repeated at the next state. But it may be that at the next state, many of the *useful* actions will already have been visited by prior iterations of the algorithm. For a lookahead depth D, some actions will have already been explored up to depth $D - 1$.

For example if we have generated the partial sentence "the cat chased the rabbit" and S-STRUCT looks ahead to find that a greater reward is possible by introducing the relative clause "the rabbit that ate", when we transition to "the rabbit that", we do not need to re-explore "ate" and can directly try actions that result in "that ate grass", "that ate carrots", etc. Note that if there are still unexplored actions at an earlier depth, these will still be explored as well (action rollouts such as "that drank water" in this example).

Reusing the search tree is especially effective given that the tree policy causes us to favor areas of the search space with high value. Therefore, when we transition to the state with highest value, it is likely that many useful actions have already been explored. Reusing the search tree is reflected in Algorithms 1-2 by passing $uctTree$ back and forth to/from $getAction$ instead of starting a new search tree at each step. In $applyAction$, when a state/action already in the tree is chosen, S-STRUCT transitions to the next state without having to recompute the state or its reward.

3.4 Learning and Using Search Control

During the search procedure, a large number of actions are explored but relatively few of them are helpful. Ideally, we would know which actions would lead to valuable states without actually having to expand and evaluate the resultant states, which is an expensive operation. From prior knowledge, we know that if we have a partial sentence of "the sky is", we should try actions resulting in "the sky is blue" before those resulting in "the sky is yellow". This prior knowledge can be estimated through *learned* heuristics from previous runs of the planner (Yoon et al., 2008). To do this, a set of previously completed plans can be treated as a training set: for each (state, action) pair considered, a feature vector $\Phi(s, a)$ is emitted, along with either the distance to the goal state or a binary indicator of whether or not the state is on the path to the goal. A perceptron (or similar model) $H(s, a)$ is trained on the $(\Phi(s, a), target)$ pairs. $H(s, a)$ can be incorporated into the planning process to help guide future searches.

We apply this idea to our S-STRUCT system by tracking the (state, action) pairs visited in previous runs of the STRUCT system where STRUCT obtained at least 90% of the reward of the known best sentence and emit a feature vector for each, containing: global tree frequency, tree probability (as defined in Section 4.1), and the word correlation of the action's anchor with the two words on either side of the action location. We define the global tree frequency as the number of times the tree appeared in the corpus normalized by the number of trees in the corpus; this is different than the tree probability as it does not take any context into account (such as the parent tree and substitution location). Upon search completion, the feature vectors are annotated with a binary indicator label of whether or not the (state, action) pair was on the path to the best sentence. This training set is then used to train a perceptron $H(s, a)$.

Table 1: Summary statistics for test data sets

Test Set	**Goals / Sentences**	**Vocab Size**	**Lex Trees / Actions**	**World Entities**	**World Relations**	**Avg. Goal Entities**	**Avg. Goal Relations**	**Max Depth**
Small	50	130	395	77	135	1.54	2.70	0
Medium	500	1165	3734	741	1418	1.48	2.83	1
Large	5000	9872	31966	10998	23097	2.20	4.62	6

We use $H(s,a)$ to inform both the open action policy (Algorithm 2, line 4) and the tree policy (Algorithm 2, line 6). In the open action policy , we choose open actions according to their heuristic values, instead of just their tree probabilities. In the tree policy, we incorporate $H(s,a)$ into the reward estimation by using Equation 2 in place of Equation 1 in Algorithm 2 (Chaslot et al., 2008a):

$$P(s,a) = Q(s,a) + \lambda H(s,a) + c\sqrt{\frac{lnN(s)}{N(s,a)}}. \quad (2)$$

Here, $H(s,a)$ is a value prediction from prior knowledge and λ is a parameter controlling the trade-off between prior knowledge and estimated value on this goal.

4 Empirical Evaluation

In this section, we evaluate three hypotheses: (1) S-STRUCT can handle real-world datasets, as they scale in terms of (a) grammar size, (b) world size, (c) entities/relations in the goal, (d) lookahead required to generate sentences, (2) S-STRUCT scales better than STRUCT to such datasets and (3) Each of the enhancements above provides a positive contribution to STRUCT's scalability in isolation.

4.1 Datasets

We collected data in the form of grammars, worlds and goals for our experiments, starting from the WSJ corpus of the Penn TreeBank (Marcus et al., 1993). We parsed this with an LTAG parser to generate the best parse and derivation tree (Sarkar, 2000; XTAG Research Group, 2001). The parser generated valid parses for 18,159 of the WSJ sentences. To pick the *best* parse for a given sentence, we choose the parse which minimizes the PARSEVAL bracket-crossing metric against the gold-standard (Abney et al., 1991). This ensures that the major structures of the parse tree are retained. We then pick the 31 most frequently occurring XTAG trees (giving us 74% coverage of the parsed sentences) and annotate them with compositional semantics. The final result of this process was a corpus of semantically annotated WSJ sentences along with their parse and derivation trees [1].

To show the scalability of the improved STRUCT system, we extracted 3 datasets of increasing size and complexity from the semantically annotated WSJ corpus. We nominally refer to these datasets as Small, Medium, and Large. Summary statistics of the data sets are shown in Table 1. For each test set, we take the grammar to be all possible lexicalizations of the unlexicalized trees given the anchors of the test set. We set the world as the union of all communicative goals in the test set. The PLTAG probabilities are derived from the entire parseable portion of the WSJ. Due to the data sparsity issues (Bauer and Koller, 2010), we use unlexicalized probabilities.

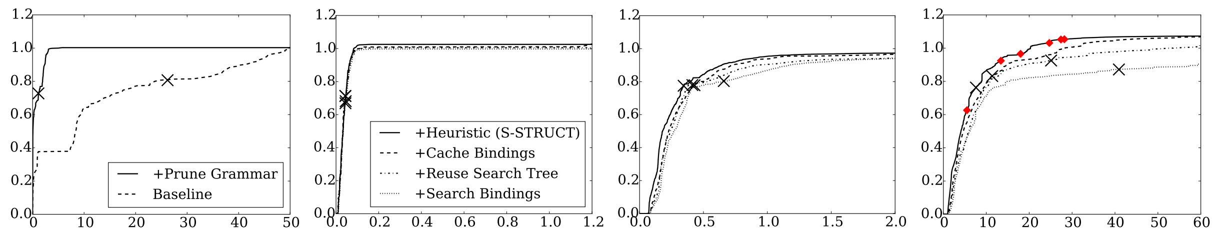
The reward function constants $\boldsymbol{C}$ were set to $[500, 100, 10, 10, 1]$. In the tree policy, c was set to 0.5. These are as in the original STRUCT system. λ was chosen as 100 after evaluating $\{0, 10, 100, 1000, 10000\}$ on a tuning set.

In addition to test sets, we extract an independent training set using 100 goals to learn the heuristic $H(s,a)$. We train a separate perceptron for each test set and incorporate this into the S-STRUCT algorithm as described in Section 3.4.

4.2 Results

For these experiments, S-STRUCT was implemented in Python 3.4. The experiments were run on a single core of a Intel(R) Xeon(R) CPU E5-2450 v2 processor clocked at 2.50GHz with access to 8GB of RAM. The times reported are from the start of the generation process instead of the start of the program execution to reduce variation caused by interpreter startup, input parsing, etc. In

[1] Not all of the covered trees were able to recursively derive their semantics, despite every constituent tree being semantically annotated. This is because β-reduction of the λ-semantics is not associative in many cases where the syntactic composition is associative, causing errors during semantic unification. Due to this and other issues, the number of usable parse trees/sentences was about 7500.



(a) Small Baseline (b) Small (c) Medium (d) Large

Figure 2: Avg. Best Normalized Reward (y-axis) vs. Time in Seconds (x-axis) for (a) Small Baseline, (b) Small, (c) Medium, (d) Large. Time when first grammatical sentence available marked as $\times$. Experiments are cumulative (a trial contains all improvements below it in the legend).

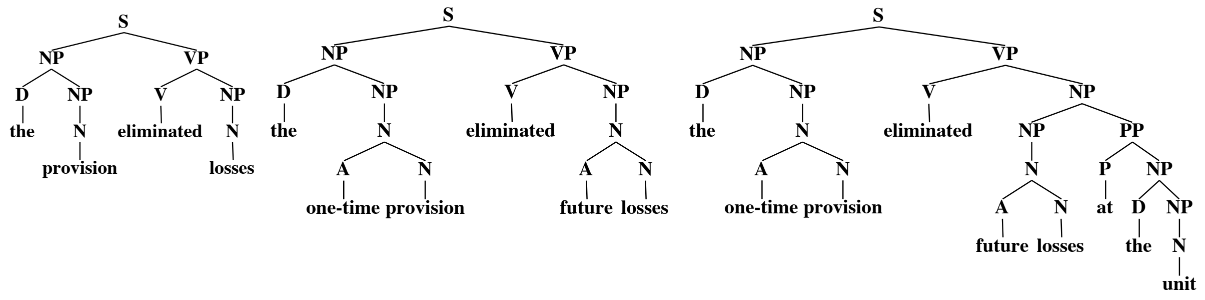

the provision eliminated losses
the one-time provision eliminated future losses
the one-time provision eliminated future losses at the unit

Figure 3: Best sentence available during S-STRUCT generation at 5.5 (s), 18.0 (s), and 28.2 (s)

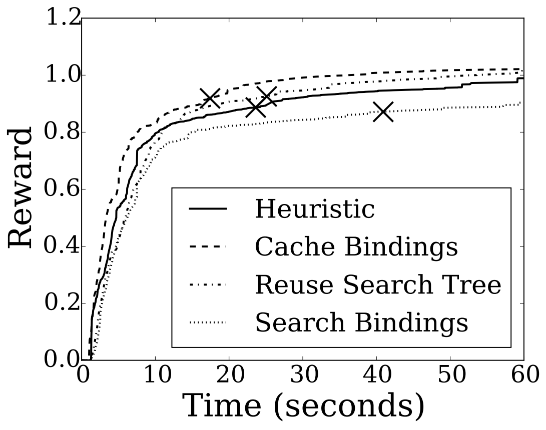
all experiments, we normalize the reward of a sentence by the reward of the actual parse tree, which we take to be the gold standard. Note that this means that in some cases, S-STRUCT can produce solutions with *better* than this value, e.g. if there are multiple ways to achieve the semantic goal.

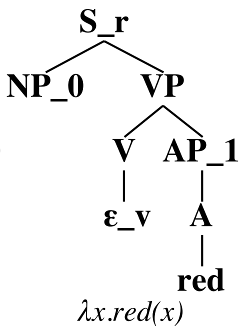
To investigate the first two hypotheses that S-STRUCT can handle the scale of real-world datasets and scales better than STRUCT, we plot the average best reward of all goals in the test set over time in Figure 2. The results show the cumulative effect of the enhancements; working up through the legend, each line represents “switching on” another option and includes the effects of all improvements listed below it. The addition of the heuristic represents the entire S-STRUCT system. On each line, $\times$ marks the time at which the first grammatically correct sentence was available.

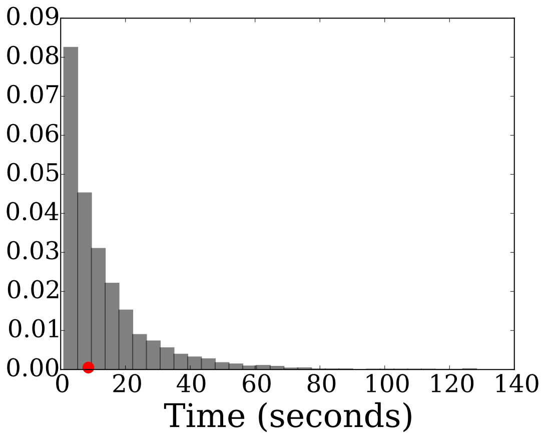
The Baseline shown in Figure 2a is the original STRUCT system proposed in (McKinley and Ray, 2014). Due to the large number of actions that must be considered, the Baseline experiment's average first sentence is not available until 26.20 seconds, even on the Small dataset. In previous work, the experiments for both STRUCT and CRISP were on toy examples, with grammars having 6 unlexicalized trees and typically < 100 lexicalized trees (McKinley and Ray, 2014; Koller and Stone, 2007). In these experiments, STRUCT was shown to perform better than or as well as CRISP. Even in our smallest domain, however, the baseline STRUCT system is impractically slow. Further, prior work on PCRISP used a grammar that was extracted from the WSJ Penn TreeBank, however it was restricted to the 416 sentences in Section 0 with <16 words. With PCRISP's extracted grammar, the most successful realization experiment yielded a sentence in only 62% of the trials, the remainder having timed out after five minutes (Bauer and Koller, 2010). Thus it is clear that these systems do not scale to real NLG tasks.

Adding the grammar pruning to the Baseline allows S-STRUCT to find the first grammatically correct sentence in 1.3 seconds, even if the reward is still sub-optimal. For data sets larger than Small, the Baseline and Prune Grammar experiments could not be completed, as they still enumerated all semantic bindings. For even the medium world, a sentence with 4 entities would have to consider 1.2×10^{10} bindings. Therefore, the cumulative experiments start with Prune Grammar and Search Bindings turned on.

Figures 2b, 2c and 2d show the results for each enhancement above on the corresponding dataset. We observe that the improved binding search further improves performance on the Small task. The Small test set does not require any lookahead, so it is expected that there would be no benefit to

(a)


(b)


(c)


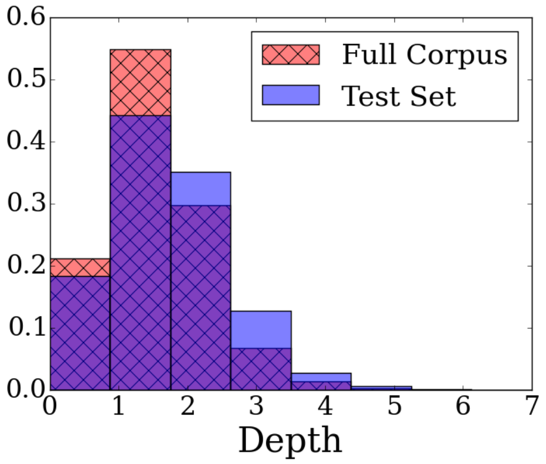
(d)


Figure 4: (a) Large Non-Cumulative Experiment (b) $\alpha nx0Ax1$ XTAG tree (c) Time to 90% Reward (d) Lookahead Required.

reusing the search tree, and little to no benefit from caching bindings or using a heuristic. In the Small domain, S-STRUCT is able to generate sentences very quickly; the first sentence is available by 44ms and the best sentence is available by 100ms. In the medium and large domains, the "Reuse Search Tree", "Cache Bindings", and "Heuristic" changes *do* improve upon the use of only "Search Bindings". The Medium domain is still extremely fast, with the first sentence available in 344ms and the best sentence available around 1s. The large domain slows down due to the larger lookahead required, the larger grammar, and the huge number of bindings that have to be considered. Even with this, S-STRUCT can generate a first sentence in 7.5s and the best sentence in 25s. In Figure 4c, we show a histogram of the generation time to 90% of the best reward. The median time is 8.55s (● symbol).

Additionally, histograms of the lookahead required for guaranteed optimal generation are shown for the entire parsable WSJ and our Large world in Figure 4d. The complexity of the entire WSJ does not exceed our Large world, thus we argue that our results are representative of S-STRUCT's performance on real-world tasks.

To investigate the third hypothesis that each improvement contributes positively to the scalability, the noncumulative impact of each improvement is shown in Figure 4a. All experiments still must have Prune Grammar and Search Bindings turned on in order to terminate. Therefore, we take this as a baseline to show that the other changes provide additional benefits. Looking at Figure 4a, we see that each of the changes improves the reward curve and the time to generate the first sentence.

4.3 Discussion, Limitations and Future Work

As an example of sentences available at a given time in the process, we annotate the Large Cumulative Heuristic Experiment with ♦ symbols for a specific trial of the Large dataset. Figure 3 shows the best sentence that was available at three different times. The first grammatically correct sentence was available 5.5 seconds into the generation process, reading "The provision eliminated losses". This sentence captured the major idea of the communicative goal, but missed some critical details. As the search procedure continued, S-STRUCT explored adjunction actions. By 18 seconds, additional semantic content was added to expand upon the details of the provision and losses. S-STRUCT settled on the best sentence it could find at 28.2 seconds, able to match the entire communicative goal with the sentence "The one-time provision eliminated future losses at the unit".

In domains with large lookaheads required, reusing the Search Tree has a large effect on both the best reward at a given time and on the time to generate the first sentence. This is because S-STRUCT has already explored some actions from depth 1 to $D-1$. Additionally, in domains with a large world, the Cache Binding improvement is significant. The learned heuristic, which achieves the best reward and the shortest time to a complete sentence, tries to make S-STRUCT choose better actions at each step instead of allowing STRUCT to explore actions faster; this means that there is less overlap between the improvement of the heuristic and other strategies, allowing the *total* improvement to be higher.

One strength of the heuristic is in helping S-STRUCT to avoid "bad" actions. For example, the XTAG tree $\alpha nx0Ax1$ shown in Figure 4b is an initial tree lexicalized by an adjective. This tree would be used to say something like "The dog is red." S-STRUCT may choose this as an initial action to fulfill a subgoal; however, if the goal was to say that a red dog chased a cat, S-STRUCT will be shoehorned into a substantially worse goal down the line, when it can no longer use an initial tree that adds the "chase" semantics. Although the rollout process helps, some sentences can share the same reward up to the lookahead and only diverge later. The heuristic can help by biasing the search against such troublesome scenarios.

All of the results discussed above are without parallelization and other engineering optimizations (such as writing S-STRUCT in C), as it would make for an unfair comparison with the original system. The core UCT procedure used by STRUCT and S-STRUCT could easily be parallelized, as the sampling shown in Algorithm 2 can be done independently. This has been done in other domains in which UCT is used (Computer Go), to achieve a speedup factor of 14.9 using 16 processor threads (Chaslot et al., 2008b). Therefore, we believe these optimizations would result in a constant factor speedup.

Currently, the STRUCT and S-STRUCT systems only focuses on the domain of single sentence generation, rather than discourse-level planning. Additionally, neither system handles non-semantic feature unification, such as constraints on number, tense, or gender. While these represent practical concerns for a production system, we argue that their presence will not affect the system's scalability, as there is already feature unification happening in the λ-semantics. In fact, we believe that additional features could *improve* the scalability, as many available actions will be ruled out at each state.

5 Conclusion

In this paper we have presented S-STRUCT, which enhances the STRUCT system to enable better scaling to real generation tasks. We show via experiments that this system can scale to large worlds and generate complete sentences in real-world datasets with a median time of 8.5s. To our knowledge, these results and the scale of these NLG experiments (in terms of grammar size, world size, and lookahead complexity) represents the state-of-the-art for planning-based NLG systems. We conjecture that the parallelization of S-STRUCT could achieve the response times necessary for real-time applications such as dialog. S-STRUCT is available through Github upon request.

References

S. Abney, S. Flickenger, C. Gdaniec, C. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. Klavans, M. Liberman, M. Marcus, S. Roukos, B. Santorini, and T. Strzalkowski. 1991. Procedure for quantitatively comparing the syntactic coverage of english grammars. In E. Black, editor, *Proceedings of the Workshop on Speech and Natural Language*, HLT '91, pages 306–311, Stroudsburg, PA, USA. Association for Computational Linguistics.

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

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

Bernd Bohnet and Robert Dale. 2005. Viewing referring expression generation as search. In *International Joint Conference on Artificial Intelligence*, pages 1004–1009.

Guillaume M. JB Chaslot, Mark H.M. Winands, H. Jaap van Den Herik, Jos W.H.M. Uiterwijk, and Bruno Bouzy. 2008a. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(03):343–357.

Guillaume M. JB Chaslot, Mark H.M. Winands, and H Jaap van Den Herik. 2008b. Parallel monte-carlo tree search. In *Computers and Games*, pages 60–71. Springer.

M. Fox and D. Long. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124.

Aravind K Joshi and Yves Schabes. 1997. Tree-adjoining grammars. In *Handbook of formal languages*, pages 69–123. Springer.

Daniel Jurafsky and James H. Martin. 2000. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition.

Martin Kay. 1996. Chart generation. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, ACL '96, pages 200–204, Stroudsburg, PA, USA. Association for Computational Linguistics.

Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg. Springer-Verlag.

Alexander Koller and Matthew Stone. 2007. Sentence generation as a planning problem. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 336–343, Prague, Czech Republic, June. Association for Computational Linguistics.

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

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

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of english: The Penn Treebank. *Computational linguistics*, 19(2):313–330.

Nathan McKinley and Soumya Ray. 2014. A decision-theoretic approach to natural language generation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 552–561, Baltimore, Maryland, June. Association for Computational Linguistics.

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

Ehud Reiter and Robert Dale. 1997. Building applied natural language generation systems. *Natural Language Engineering*, 3(1):57–87.

Anoop Sarkar. 2000. Practical experiments in parsing using tree adjoining grammars. In *Proceedings of TAG*, volume 5, pages 25–27.

Stuart M. Shieber. 1988. A uniform architecture for parsing and generation. In *Proceedings of the 12th conference on Computational linguistics - Volume 2*, COLING '88, pages 614–619, Stroudsburg, PA, USA. Association for Computational Linguistics.

Tsung-Hsien Wen, Milica Gasic, Dongho Kim, Nikola Mrksic, Pei-Hao Su, David Vandyke, and Steve Young. 2015a. Stochastic language generation in dialogue using recurrent neural networks with convolutional sentence reranking. In *Proceedings of the 16th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 275–284, Prague, Czech Republic, September. Association for Computational Linguistics.

Tsung-Hsien Wen, Milica Gasic, Nikola Mrkšić, Pei-Hao Su, David Vandyke, and Steve Young. 2015b. Semantically conditioned lstm-based natural language generation for spoken dialogue systems. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1711–1721, Lisbon, Portugal, September. Association for Computational Linguistics.

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

XTAG Research Group. 2001. A lexicalized tree adjoining grammar for english. Technical Report IRCS-01-03, IRCS, University of Pennsylvania.

Sungwook Yoon, Alan Fern, and Robert Givan. 2008. Learning control knowledge for forward search planning. *The Journal of Machine Learning Research*, 9:683–718.