GOAL-DIRECTED LANGUAGE GENERATION WITH MULTIPLE BOOLEAN OPERATORS

by

ALEXANDER RAMBASEK

Submitted in partial fulfillment of the requirements

for the degree of Master of Science

Department of Computer and Data Sciences

CASE WESTERN RESERVE UNIVERSITY

January, 2024

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis of

Alexander Rambasek

candidate for the degree of **Master of Science***.

Committee Chair

Dr. Soumya Ray

Committee Member

Dr. Harold Connamacher

Committee Member

Dr. Andy Podgurski

Date of Defense

January 24, 2024

*We also certify that written approval has been obtained

for any proprietary material contained therein.

Contents

List of Figures

List of Abbreviations

CFG Context-Free Grammar

CSG Context-Sensitive Grammar

FOL First-Order Logic

LTAG Lexicalized Tree-Adjoining Grammar

MLP Multi-Layer Perceptron

NLG Natural Language Generation

NLP Natural Language Processing

PLTAG Probabilistic Lexicalized Tree-Adjoining Grammar

RE Referring Expression

REG Referring Expression Generation

S2S Sequence-to-Sequence Model

STRUCT Sentence Tree Realization with UCT

TAG Tree-Adjoining Grammar

UCB Upper Confidence Bound

UCT Upper Confidence Bounds applied to Trees

Goal-Directed Language Generation with Multiple Boolean Operators

Abstract

by

ALEXANDER RAMBASEK

We consider the problem of goal-directed natural language generation, where the aim is to produce sentences whose semantic meaning is as close as possible to the semantics of a communicative goal. A previous approach, Sentence Tree Realization with Upper Confidence Trees (STRUCT) could generate sentences with conjunctive semantics. In this thesis, we develop an extension, Disjunctive/Implicative STRUCT (diSTRUCT) to handle non-conjunctive semantics such as disjunction and implication. To do this we reformulate the reward component of STRUCT to explicitly model the semantic fidelity between truth tables, and introduce a new syntactic reward to encourage the generation of trees lexicalized with words corresponding to Boolean operators. Distributional heuristics are proposed to guide the search towards syntactically-promising sentences. We also create new semantically annotated Lexicalized Tree Adjoining Grammar (LTAG) trees to enable generation and a new dataset to evaluate our approach. We show that our approach is able to generate non-conjunctive goals and investigate the computational efficiency of the approach compared to only conjunctive goals.

Chapter 1

Introduction

In today's world, there is a gargantuan amount of data. A myriad of databases contain everything from photos and videos to social media posts, transaction ledgers, electronic health records, and so much more. Data is constantly being dumped by smartphones, industrial equipment, IoT devices, and the like. Consequently, it can be difficult if not impossible to manually extract useful information from big data. Instead, natural language generation (NLG) systems can be used to synthesize the information stored in structured data into text. The outputs of these systems can then be used to make informed decisions, such as what drug to prescribe or where to get Chinese takeout.

Natural language generation describes any system that produces natural language as an output. In goal-directed NLG, there is specific information that we want the system to convey via language: this is our communicative goal. It is also important to precisely express the goal. The output should contain neither too much or too little information to get the point across. This goal is commonly in reference to a knowledge-base of facts about the world. Exactly how the semantics of worlds and goals should be represented is a matter of open research. Different paradigms include logical approaches as well as statistical ones.

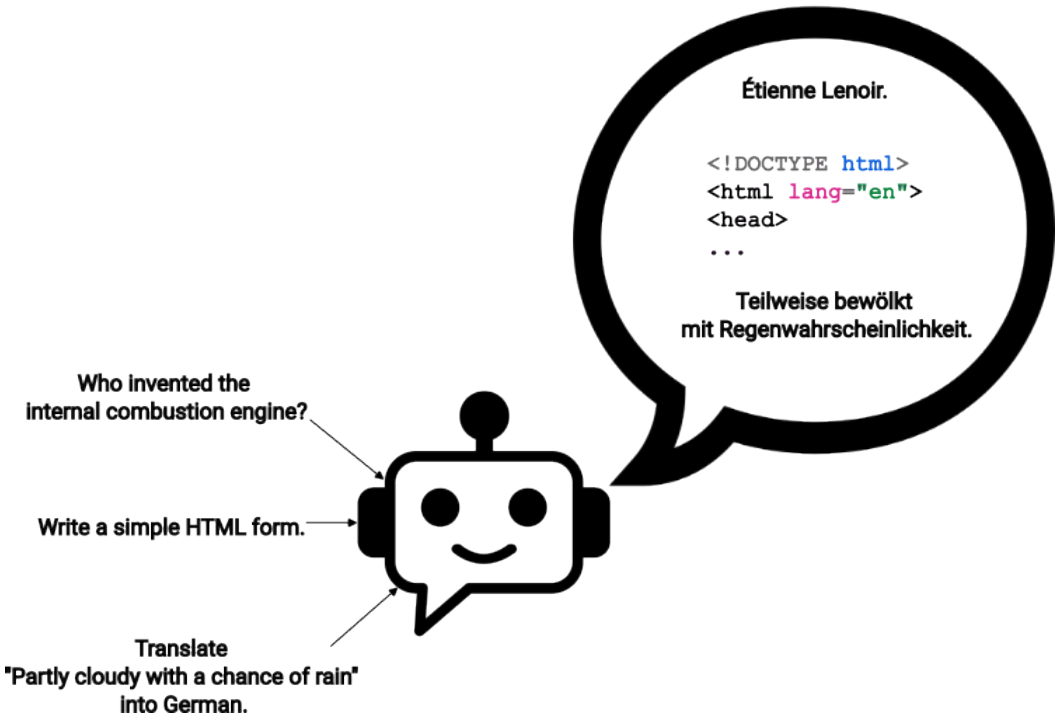

Figure 1.1: Basic Markov Decision Process
NLG systems can be leveraged to tackle many generation tasks, such as factual question answering, code completion, language translation, and image captioning.

To approach goal-directed NLG, we build on the work of the existing STRUCT (Sentence Tree Realization with UCT) system [25] [31] [6] from our lab. At a high level, STRUCT treats language generation as a planning problem and is a Markov decision process (MDP). States are partially-formed sentences, and actions are the addition of words to sentences that transform one state into another. The reward function calculates the similarity between the current partial sentence and the goal. Feedback from the reward function informs how we explore the search tree of possible actions.

As a baseline, we utilize the logical sentence-level generation system STRUCT. We offer a number of improvements over the existing STRUCT system. We address a key limitation of STRUCT by expanding its capabilities to handle worlds and goals that do not consist solely of conjunctive semantics, which are unit clauses

separated by logical "AND." We generalize the reward function of STRUCT to align the FOL goal semantics with the partial sentence semantics, as opposed to checking fulfilled relations. To do this, we leverage the Prolog logical inference engine [41] as well as FOL theorem provers. To combat the added time complexity of such an approach, we investigate how to adapt the existing method of pruning bindings to work with our new version, as well as using heuristics to inform the UCT search. Furthermore, we address the issue of semantically valid but improbable utterances by designing a syntactic reward to guide generation toward the goal. Finally, we make contributions to the semantic annotations of STRUCT's grammar.

We provide an overview of the current work in NLG and the relevance to our work in Chapter 2. In Chapter 3, we formally introduce the STRUCT system and its successors: S-STRUCT and HS-STRUCT. The motivation, methods, and design of the new system, diSTRUCT, is discussed in Chapter 4. The performance of diSTRUCT on our newly minted dataset for non-conjunctive goals, as well as a comparison of diSTRUCT to its predecessors on conjunctive goals, is presented in Chapter 5. Finally, in Chapter 6, we re-examine our contributions, and discuss potential future work in NLG, with an emphasis on the recent success and adoption of ChatGPT.

Chapter 2

Background and Related Work

2.1 NLG Pipeline

The NLG pipeline [11, 33] details the components necessary to convert raw inputs into natural language. Traditionally, NLG systems have modeled this pipeline explicitly, and significant research is available for all the pieces. However, it should be noted that the current trend—modeling NLG end-to-end with neural encoder-decoder architectures—omits the explicit representation of the pipeline and instead allows the model to learn whatever operations are necessary to get the desired outcome. These systems have achieved state-of-the-art performance on a plethora of NLG tasks, but a limitation is that the "black box" approach of these systems hinders examination of the inner workings, creating an obstacle when researchers try to resolve shortcomings such as hallucinations [28] and information repetition [15].

2.1.1 Content Determination

During *content determination*, parts of the input data relevant to the communicative goal are identified. If the input data is a collection of articles or a knowledge

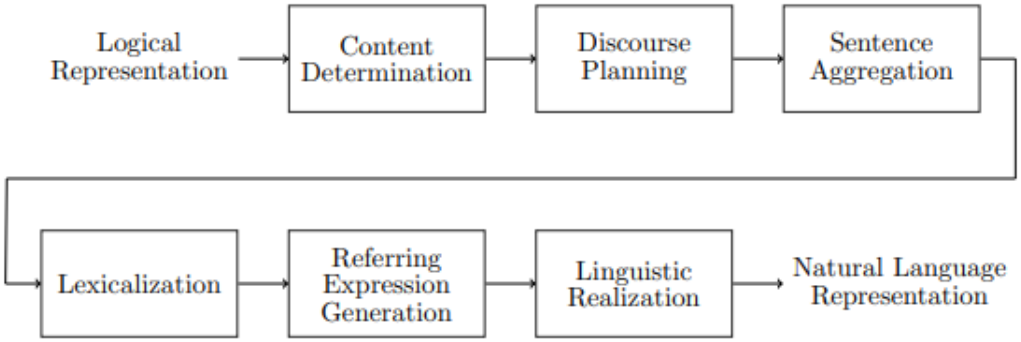

Figure 2.1: The NLG pipeline.

base, then the desired information to communicate is a small subset of the available information. The criticality of this step widely varies based on the specifics of the NLG task, and may be omitted entirely. In the template approach of [42], a degree of content determination is done implicitly as templates may only permit certain kinds of information to be captured. The work in [37] models content determination over an input text as a sequence-labeling task. A selector model masks tokens in the input text that are not related to the desired content, and the quality of the output text decoded from the encoding vectors of the unmasked tokens is used as a surrogate to evaluate the selector. The pipeline is thus able to generate novel text by attending to the specific tokens selected in the input text.

2.1.2 Document Structuring

Closely related to content determination, the results are divided up into sentences and paragraphs in *document structuring*. Different NLG systems model document structuring in very different ways. The simplest involves choosing in what order a list of generated sentences will appear in the output text [22]. A more sophisticated approach involves a discourse plan, by which facts from a knowledge base are divided up into the sentences that they should appear in; this information is then used downstream for the actual realization of these sentences via neural net-

Source Sentence: The sri lankan government on Wednesday announced the closure of government schools with immediate effect as a military campaign against tamil separatists escalated in the north of the country.

Selected : sri lankan, closure, schools
Text: sri lanka closes schools .

Selected : sri lankan, Wednesday, closure, schools
Text: sri lanka closes schools on Wednesday.

Selected : sri lankan, closure, schools, military campaign
Text: sri lanka shuts down schools amid war fears.

Selected : sri lankan, announced, closure, schools
Text: sri lanka declares closure of schools.

Figure 2.2: Content Determination in (Shen et al., 2019)
An example of results from a latent variable approach to controllable content determination. The decoder model is trained to produce sentences from the selected tokens while remaining faithful to the original text.

works [27]. For instance, consider the following fragment of a knowledge base:

name(*z*53, "Sammy Sosa")

baseball_player(*z*53)

born_in(*z*53, "Dominican Republic", 1968-11-12)

paternal_surname(*z*43, "Peralta")

position(*z*43, "right field")

career_start(*z*53, "Texas Rangers", 1989)

career_duration(*z*53, 19yr)

is_retired(*z*53, TRUE)

home_runs(*z*53, 609)

allstar_awards(*z*53, 7)

Figure 2.3: A toy knowledge base.

The highlights represent one possible grouping of the facts into sentences.

Downstream, the realized text might read: "Sammy Peralta Sosa is a Dominican former professional baseball right fielder. He made his debut for the Texas Rangers in 1989, and his career spans 19 years. He is a distinguished hitter, boasting 609 career home runs and 7 All-Star MVP awards." Some systems plan and generate synchronously, wherein latent variables are conditioned on when both templating and generating tokens. This can be realized in a hidden semi-Markov model, since it defines a joint distribution over inputs and latent segmentations [42].

2.1.3 Lexicalization

Lexicalization is the specific realization of input symbols as words in the generated text. An example is the lexicalization of a cat. A few possible lexicalizations might be "kitty," "kitten," "pussycat," "feline," "my pet cat," or "Mittens." Each different lexicalization imbues a different connotation into the text. The specific lexicalization to use depends on the context of previously generated text, and therefore is usually handled implicitly when decoding. The best example of an explicit lexicalization scheme is the "copy mechanism" [35]. Here, a "switch variable" is learned, which decides whether a token from the input text should be left unchanged or generated by a S2S model. This is especially useful to realize a text plan once it has been linearized. For instance, the linearization

$$\textbf{Marquesita} \rightarrow \text{country } [\textbf{Mexico}] \rightarrow [\textbf{Yucatán}].$$
$$\textbf{Dessert} \leftarrow \text{course } [\textbf{Marquesita}] \rightarrow \text{ingredient } [\textbf{Caramelized Milk}]].$$

is a rough outline of two sentences: the division of content between the sentences and the content ordering is correct (assuming this is what we want to say). Furthermore, the relational triplets contain words that should probably appear in the realization. The model will need to figure out when to copy the word from the plan and when other words need to be added. Such a lexicalization system

would be expected to yield something similar to: "**Marquesita** is a food found in the **Mexico** region **Yucátan**. The **Dessert Marquesita** requires **Caramelized Milk** as an ingredient."

2.1.4 Referring Expression Generation

After choosing what to say and where to say it, the next step is *microplanning* how to say it, such as via the generation of *referring expressions*. A referring expression is a noun phrase that uniquely identifies an entity in the world. Let us refer back to the example knowledge base in Figure 2.3, and suppose we added the facts

$$baseball\_player(z54, \text{“Reggie Jackson”})$$
$$position(z54, \text{“right field”})$$
$$home\_runs(z54, 563)$$

to the world. Now, there are two professional baseball right-fielders. So these two facts alone are no longer sufficient to uniquely identify Sammy Sosa. We would need to also mention something else, like the fact that Sammy Sosa hit 609 home runs.

Referring expression generation is one of the most widely studied topics in NLP Generating an arbitrary referring expression is trivial with sufficient information about the world, but a more challenging and interesting problem is the generation of a minimal referring expression: any referring expression with as few descriptors as possible. To achieve this, researchers have modeled REG as a search problem and proposed iterative algorithms [8]. However, this approach has been criticized in recent years, as the computational cost of computing REs is very high, and the results need not align with how humans generally describe things in discourse [21]. Imagine if in a baseball periodical, the referring expression "Dominican right fielder who debuted with the Texas Rangers" was used in place of "Sammy Sosa"

every time he was referred to besides in name. This would quickly devolve into borderline illegible writing. Instead, more recent work is concerned with referring expressions that are both natural and varied enough to avoid repetitive prose [29].

2.1.5 Surface Realization

The generated phrases are post-processed in *surface realization* to form a syntactically-valid and coherent text, taking into account any stylistic constraints. Surface realization can include the generation of any number of a broad array of elements, such as determiners, functional prepositions, morphologies, word inflections, and punctuation. Like many of the previous steps of the pipeline, systems often omit explicit surface realization and instead rely on the decoding process to produce clear and concise text.

2.2 Syntax

Natural language generation systems must concern themselves with both syntax and semantics. Syntax refers to the rules that govern how valid utterances in natural language can be formed. Semantics refers to the actual meaning of the utterances themselves. Syntax is often modeled as a formal grammar: a set of rules for rewriting strings. Formal grammars are comprised of production rules of the form

$$\mathcal{X} \rightarrow \mathcal{Y}_1\mathcal{Y}_2...\mathcal{Y}_n.$$

Here, $\mathcal{X}$ is a nonterminal symbol, and the production rule above allows for $\mathcal{X}$ to be replaced with the symbols $\mathcal{Y}_1\mathcal{Y}_2...\mathcal{Y}_n$ wherever it appears. Every grammar starts with a single nonterminal $\mathcal{S}$ (the "start symbol" or "sentence symbol"), from which all sentences are derived. For instance, a basic rule that exists in English might be described as "a noun phrase followed by a verb phrase is a valid sentence." Another

is "a verb followed by a noun phrase is a valid verb phrase." Here is a formalization of both:

$$\mathcal{S} \to \langle \text{NP} \rangle \langle \text{VP} \rangle$$

$$\langle \text{VP} \rangle \to \langle \text{V} \rangle \langle \text{NP} \rangle.$$

To generate a valid sentence in the grammar, production rules must continuously be applied until the string contains only terminal symbols: symbols that cannot be expanded further. In natural languages, these correspond to words.

$$\langle \text{N} \rangle \to \text{aardvark} | \text{abacus} | ... | \text{zygote}$$

$$\langle \text{V} \rangle \to \text{abandon} | \text{abate} | ... | \text{zoom}.$$

This is a general form for all formal grammars. However, there are different kinds of formal grammars, each with certain restrictions on the structure and use of rewrite rules. The Chomsky hierarchy [2] groups formal grammars based on their expressivity. The expressivity of a formal grammar is a notion of the variety of utterances that are valid strings in the language modelled by the grammar. The higher the expressivity, the more sophisticated the utterances that can exist in the language. In increasing order of expressivity, the four levels are regular, context-free, context-sensitive, and recursively enumerable. At the lowest level, a regular language is any language that can be recognized by a finite-state machine, while a recursively enumerable language is defined in terms of a Turing machine. We would like to strike a balance between complexity and simplicity such that our formal language of choice can represent any structure in natural language, while being as simple as possible. This is because learning a grammar for language is done statistically with corpora of text, and more sophistication requires

more training data. It is generally accepted that a *mildly context-sensitive* grammar is sufficient to adequately describe natural language syntax [18], lying in complexity between context-free grammars (CFG) and context-sensitive grammars (CSG). That is, natural language is mostly context-free but does allow for cross serial dependencies that cannot be captured by a context-free formalism. One specific type of mildly context-sensitive grammar is a *tree-adjoining grammar* (TAG) [17].

2.2.1 Tree Adjoining Grammars

In addition to being unable to model natural language, general CFGs, in which rewriting symbols with other symbols is the only operation, cannot be *lexicalized*. To lexicalize a grammar is to convert it into an equivalent grammar in which every action also induces the addition of a terminal symbol, or anchor. The result is that, at any point during parsing or generation, we can reason about the current *partial sentence*. As it turns out, a tree-adjoining grammar (TAG) is one such formalism that is both mildly context-sensitive and permits lexicalization. Instead of having rules for rewriting strings, a TAG has rules for rewriting the nodes of a tree as other trees. This operation is called *substitution*. In substitution, a node of a tree is replaced with another entire tree whose root node label matches the label at the substitution site. TAGs are equipped with another operation, *adjunction*, which permits trees to be embedded into the centers of trees. That is, adjunction operated similarly to substitution except that adjunction sites are in the middle of trees while substitution sites are leaf nodes. Trees that can be substituted and trees that can be adjoined are called initial trees and auxiliary trees, respectively. A TAG in which every tree has at least one lexical anchor is called a Lexicalized Tree Adjoining Grammar (LTAG). Figure 2.4 [6] shows how these LTAG trees are combined via substitution and adjunction to form utterances.

It is no surprise that phrases and sentences can have many different LTAG

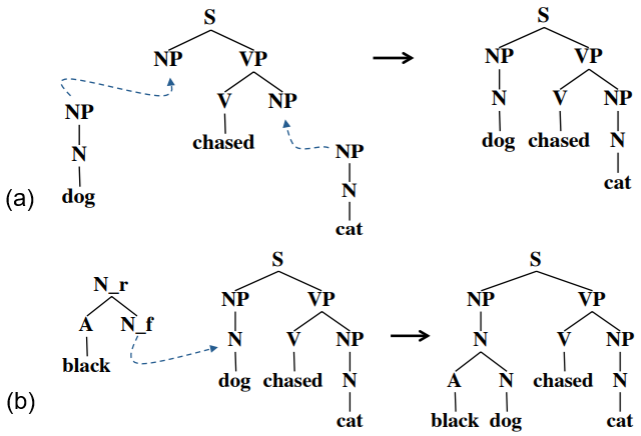

Figure 2.4: TAG Substitution and Adjunction
(a) The initial tree representing "dog" is substituted into the leaf NP node, changing "⟨NP⟩ chased cat" to "dog chased cat." (b) The auxiliary tree for "black" has a root label of N, so it can be adjoined at the N node in the "dog" subtree. The auxiliary tree's root takes the place of the old subtree root, and the "dog" subtree is substituted back into the foot node of the auxiliary tree.

parses. After all, there is semantic information encoded into words that lets humans know how a sentence should be mentally "parsed." A classic example is "time flies like an arrow" and "fruit flies like a banana." In the first sentence, the prepositional phrase "like an arrow" modifies the clause "time flies." But in the second example, "fruit flies" is a compound noun, and "like" is a transitive verb with object "a banana." Human beings can recognize this distinction automatically: fruit can't fly, and "fruit fly" is a well-known concept. But general TAG parsers are not privy to this, and will parse the second sentence like the first. Probabilistic Lexicalized Tree Adjoining Grammars (PLTAGs) aim to address this by formalizing the concept of the likelihood of a specific parse, informed by human-annotated parses on a text corpus. PLTAGs are the natural extension of n-gram statistics to trees. That is, for every tree τ and every lexicalization $\tau(x)$ of τ, the probability of $\tau(x)$ being substituted onto node α of a lexicalized tree $\tau'(x')$ is

$$\sum_{\text{derived tree} \in \text{corpus}} \sum_{\gamma \in \text{derived tree}} \frac{\mathbb{1}(\gamma = \tau'(x') \text{ and } \tau(x) \text{ subst. onto } \gamma)}{\mathbb{1}(\gamma = \tau'(x'))},$$

and a similar formulation for adjunction. However, there are sparsity concerns with PLTAGs [5]: there will be plenty of specific substitutions/adjoins with specific lexicalizations that will appear in novel sentences, but did not appear in the training corpora. For this reason STRUCT does not use PLTAGs.

2.2.2 XTAG Project

The XTAG project from the XTAG Research Group of the University of Pennsylvania [1], specifically their wide-coverage LTAG grammar for English, is a major reason why this research is possible. The XTAG grammar has a total of 1004 trees, and captures linguistic structures such as relative clauses and wh-movement. A large part of this work involved hand-annotating XTAG tree nodes with FOL semantics so that, when the XTAG parser creates derivation trees from text corpora, the associated semantics of the sentence would also be automatically generated. As a result, ground-truth semantics of the goal trees are available for evaluation. The manual annotation of XTAG trees with semantics has been done before [16]; unfortunately, this work has been lost to time. As will be discussed later, there is a critical need for an updated grammar and parser that takes advantage of recent developments in neural NLG. There are many sentences that are parsed improperly or not parsed at all, and first-order semantics are not capable of expressing all semantic structures found in natural language.

2.3 Semantics

Natural language generation systems must define both a syntax and semantics, explicitly or otherwise. The benefit of the syntax of TAGs is that the formalism makes no assumptions about the semantics of language. Indeed, any semantic representation can be used to annotate the nodes of TAG trees. When substituting

and adjoining trees, semantic labelings can be combined with custom compositional rules, in effect creating a "semantic tree" that is parsed synchronously with the syntax tree. This is called a Synchronous Tree Adjoining Grammar [38]. In this work we use first-order logic abstracted with lambda calculus, resulting in a compositional first-order logic, but the theory of this work is extendable to more powerful semantic representations, like Neo-Davidsonian semantics [30], frame semantics [12], and more broadly, distributional semantics.

2.3.1 First-Order Logic

First-order logic is best understood as an extension of propositional logic. Propositional logic deals with propositions, symbols that are either true or false, and logical connectives between propositions. In practice the allowed logical connectives are $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$ with the English interpretations "not", "and," "or," "implies," and "if and only if." The set of well-formed strings in propositional logic is recursively defined as any proposition or any logical connective between two well-formed strings. For example, let P be the proposition "it is raining" and Q "it is cloudy." Then the following are well-formed strings:

$$\neg P$$

$$(P \wedge Q) \vee (\neg P \wedge \neg Q)$$

$$P \leftrightarrow \neg(\neg P)$$

$$Q \wedge \neg Q.$$

Every statement in propositional logic evaluates either to true ($\top$) or false ($\bot$) under an assignment of truth values to propositions. The assignment "$P = \top, Q = \bot$" gives "$\neg P = \bot, (P \wedge Q) \vee (\neg P \wedge \neg Q) = \bot$." Notice how, regardless of the assignment,

the third statement is always true and the fourth statement always false. These types of statements are called tautologies and contradictions, respectively.

Propositional logic also comes equipped with inference rules. Inference rules allow for the derivation of new formulae from known ones. Modus ponens is the simplest:

$$\frac{P \to Q, P}{\therefore Q}.$$

That is, if P implies Q, and it is known that $P = \top$, modus ponens can be used to derive $Q = \top$.

The set of well-formed strings under the formal grammar of propositional logic defines a formal language, and the inference rules define a deductive system. Together, a formal language and a deductive system make a formal system. First-order logic is another such formal system. Like propositional logic, well-formed strings can be built by combining two well-formed strings with a Boolean connective. However, first-order logic generalizes the propositions of propositional logic into predicate functions. Predicate functions have a number of variables, determined by their arity. When world entities are bound to the variables, the predicate expression evaluates to $\top$ or $\bot$. The truth value changes based on the bound entities. For example, the predicate function likes(x, y) accepts two arguments. It may be the case that likes(Alice,Bob) = $\top$ but likes(Bob,Alice) = likes(Carol,Bob) = $\bot$. Furthermore, first-order logic allows for the quantifiers "$\forall$" and "$\exists$," read as "for all" and "there exists" The FOL statement

$$\forall x \, \exists y \, \text{loves}(x, y)$$

reads "For every x, there exists a y such that loves(x, y)" or more simply, "everybody loves somebody." First-order logic is called as such to distinguish it from higher-order logics such as second-order logic, which permits predicates to be

quantified over and accept other predicates as arguments.

As it turns out, to make first-order logic suitable for use with STRUCT, it must be augmented with lambda calculus. This is because otherwise, first-order logic expressions could not be composed together in any more sophisticated ways than joining them with a Boolean connective. The specific piece of lambda calculus necessary for this purpose is the lambda extraction. A lambda extraction is a function definition of the form

$$(\lambda x.M)$$

that takes as input x and returns the body M. Suppose that a TAG tree is rooted with "because" and the two subtrees have semantics matching "it is raining" and "it is cloudy." It is not possible to introduce a predicate "because" that accepts both sentences as arguments without transitioning into second-order logic. Instead, the lambda extraction

$$\lambda xy.(x \rightarrow y)$$

allows for the formation of the sentence "it is raining because it is cloudy" with the appropriate first-order semantics. This is useful to STRUCT because TAG trees have to reason over FOL expressions propagated upwards by subtrees.

Neo-Davidsonian Semantics

In first-order logic, there is no notion of time. First-order logic only permits one to discuss named entities and predicates defined over those entities. There is not a way to discuss events or anything about events, including where and when they happened. Consider the sentence

After the game, Mary took a walk in the park.

The four nouns in this sentence correspond to entities in FOL: "game," "Mary," "walk" and "park." A natural first step in building a FOL expression representing

this sentence is to declare four entities, characterize their role in the sentence, and model the main clause.

$$\text{person}(z1, Mary) \wedge game(z2) \wedge walk(z3) \wedge park(z4) \wedge took(z1, z3)$$

But, there is an issue. There is not an obvious way to model in FOL the fact that Mary's walk was located "in the park" and happened chronologically "after the game." As it turns out, there is no way to do this directly in FOL without modifying the logic or switching to a higher-order logic. The modifications "in the park" and "after the game" qualify an event, not an entity, and FOL does not allow predicates or quantification over anything except non-logical objects (variables). Furthermore, it is preferable to remain in first-order logic instead of a higher-order logic in reasoning systems because FOL is complete: any valid formula can be proved without additional inference rules. However, re-casting events into event variables circumvents this issue; this is precisely what is dubbed neo-Davidsonian semantics [30], a variation of Donald Davidson's original event semantics from 1967 [10].

In Davidson's original work, he argues that transitive verbs implicitly introduce an "event" variable that is important to the meaning. Consider the following sentence:

John bakes a cake.

"Bakes," in this context, is a transitive verb; there is a "baker" (John) and a thing that is baked (the cake). The semantics would reasonably read as

$$\text{BAKES}(\text{John}, \text{the cake}).$$

However, Davidson points out that representations like above are a dead end in

that no more can be specified about the cake or the circumstances surrounding John's baking of it. If the next sentence reads "John baked it hastily in an oven in his basement at midnight," it is not clear what "it" refers to in the continuation. Thus, Davidson proposes an alternate logical form.

$$\exists e\,[(BAKES)(\text{John, the cake, } e)].$$

The continuation is now obvious.

$$\begin{aligned}\exists e\,[&\text{BAKES(John, the cake, } e) \wedge \text{ASPECT}(e, \text{hastily})\\ \wedge\,&\text{INSTRUMENT}(e, \text{oven}) \wedge \text{LOCATION}(e, \text{basement})\\ \wedge\,&\text{TIME}(e, \text{midnight})].\end{aligned} \tag{2.1}$$

Linguists began to realize that, even though Davidson intended event variables to be linked to action verbs, any predicate may have a "hidden" Davidsonian event argument associated with it. Such an observation gave birth to the neo-Davidsonian paradigm [30]. The event is thus separated from the predicate and stands alone in neo-Davidsonian semantics:

$$\begin{aligned}\exists e\,[&\text{BAKES}(e) \wedge \text{AGENT}(e, \text{John})\\ \wedge\,&\text{PATIENT}(e, \text{John}) \wedge \text{ASPECT}(e, \text{hastily})\\ \wedge\,&\text{INSTRUMENT}(e, \text{oven}) \wedge \text{LOCATION}(e, \text{basement})\\ \wedge\,&\text{TIME}(e, \text{midnight})].\end{aligned} \tag{2.2}$$

Under the neo-Davidsonian paradigm, all verbs are predicates with arity 1 ranging over events, and the verb's arguments are introduced via roles like "agent" and "experiencer." This allows logicians to decompose verbs into more granular

elements such as cause and effect, which further permits the introduction of a temporal ordering on events.

Neo-Davidsonian semantics offer NLG systems more linguistic power, but for the sake of simplicity STRUCT is considered under the existing framework of FOL semantics. As will be shown, STRUCT's architecture is semantically agnostic, so the methods in this paper are generalizable to any compositional semantics.

2.3.2 Distributional Semantics

"You shall know a word by the company it keeps." This quote is attributed to linguist John Rupert Firth [13], and captures the motivation for distributional semantics. Distributional semantics is concerned with the co-occurrence statistics of words and phrases in natural language. From these statistics, high-dimensional embeddings can be learned for utterances, allowing for the direct quantification of the meaning of language. A good embedding will be smooth; i.e., small changes in the embedding space correspond to small changes in meaning. For instance, one would expect the vector for "happy" to be reasonably close to the vector for "joyous" and far away from the vectors for "carburetor" and "Arizona Diamondbacks." With a set of learned embeddings, models can autocomplete missing words or phrases from text. This technique is the main theory behind Google's word2vec [26], where log-linear classifiers are trained to predict words based on context. Often, a desired property of these embedding spaces is compositionality. A word that illustrates a concept can usually be broken down into smaller words (e.g., "mercenary" could be subdivided into "soldier" and "contractor"). An embedding space that exhibits additive compositionality might allow for assertions like $\overrightarrow{king} - \overrightarrow{man} + \overrightarrow{woman} = \overrightarrow{queen}$. Such constraints can be enforced during training to guide the space towards this more interpretable form [36]. Note that, under this regime, the similarity between a candidate utterance and a communicative goal

can be found by simply taking a dot product; this has been explored in previous work on STRUCT [6].

2.4 NLG Strategies

The pipeline illustrated in Section 2.1 is merely an abstraction of common themes; a reification of this pipeline frequently involves merging or even omitting steps. NLG systems generally adhere to a planning approach, a statistical approach, or a combination of both. Most modern NLG is done with neural networks, but researchers look to the NLG-as-planning literature to address some of the pitfalls of neural NLG.

2.4.1 NLG as Planning

In this paper, NLG is treated as a planning problem. A planning problem is defined by states, actions that transfer the agent from one state to another, and a desired "goal" state. Here, states are partial utterances, and actions modify these utterances by adding words. The actions of the planning problem are constrained by the grammar of language. The goal state is the target utterance. There is a rich literature of different formulations of the language planning problem, but the system most similar to the work done in this paper is SPUD (Sentence Planner Using Descriptions) [39]. SPUD is a question-and-answer system designed to be a librarian. That is, SPUD queries the knowledge base of the catalog of a library to answer questions about the library's books. For instance, the system might be asked "Do you have the books for Syntax 551 and Pragmatics 590?" The main idea of the creators of SPUD is to treat the realization of sentences as closely tied to referring expression generation. SPUD parses the question to determine the entities (books) and relations (things describing books) that are important to the speaker.

The system maintains two knowledge bases: information known to the system, and information known to the speaker. This allows for a single session of multiple questions and answers that do not convey any redundant information to the user. To distinguish entities for the speaker, SPUD generates minimal referring expressions for all necessary entities. Then, in a greedy fashion, SPUD substitutes and adjoins the TAG trees rooted with these entities to try and realize the communicative goal. There are many limitations of this system. The most onerous is the fact that many trees and semantic annotations are hand-tailored to be amenable specifically to the library-related task. Also, the allowed questions are very rudimentary in nature, only pertaining to distinguishing books from others and talking about whether a book has been lent. The CRISP system [20, 5] expands on SPUD by integrating an off-the-shelf graph planner as well as probabilistic LTAG trees to improve search speed and quality.

2.4.2 Neural NLG

The Transformer [40] network is a real *tour de force*, having captivated the computational natural language world and forming the backbone of popular generation systems like ChatGPT [9] and Google Bard. Inspired in part by the limitations of Recurrent Neural Networks (RNNs), the Transformer boasts faster training times and improved performance in the presence of long-range dependencies. The architecture is based on the multi-head attention mechanism [4]. Simply put, attention layers allow individual tokens to amplify signals from other parts of the input that are relevant to it.

Chapter 3

Conjunctive Goal-Directed NLG Using STRUCT

Sentence Tree Realization using UCT (STRUCT) is an NLG system that generates at the sentence level. Given a world of entities and relationships between entities expressed in first-order logic, STRUCT is tasked with generating a sentence that satisfies a communicative goal, while remaining logically consistent with the world. A communicative goal is some subset of the world that we want to talk about, precisely. STRUCT does this by iteratively modifying a semantic expression by introducing entities, relationships, and qualifiers on relationships. The performance of the system is obtained by comparing the semantic expression generated by STRUCT to that of the communicative goal.

STRUCT models NLG as a Markov decision process (MDP). The states S are all syntactically-valid XTAG trees with associated first-order logic (FOL) semantics. From any given state, the available actions A are all the ways that any trees in the grammar can be validly substituted into or adjoined onto the state. The transition probabilities T associated with actions capture the plausibility of the utterance of the resulting state. One possible way of doing this is to compute conditional

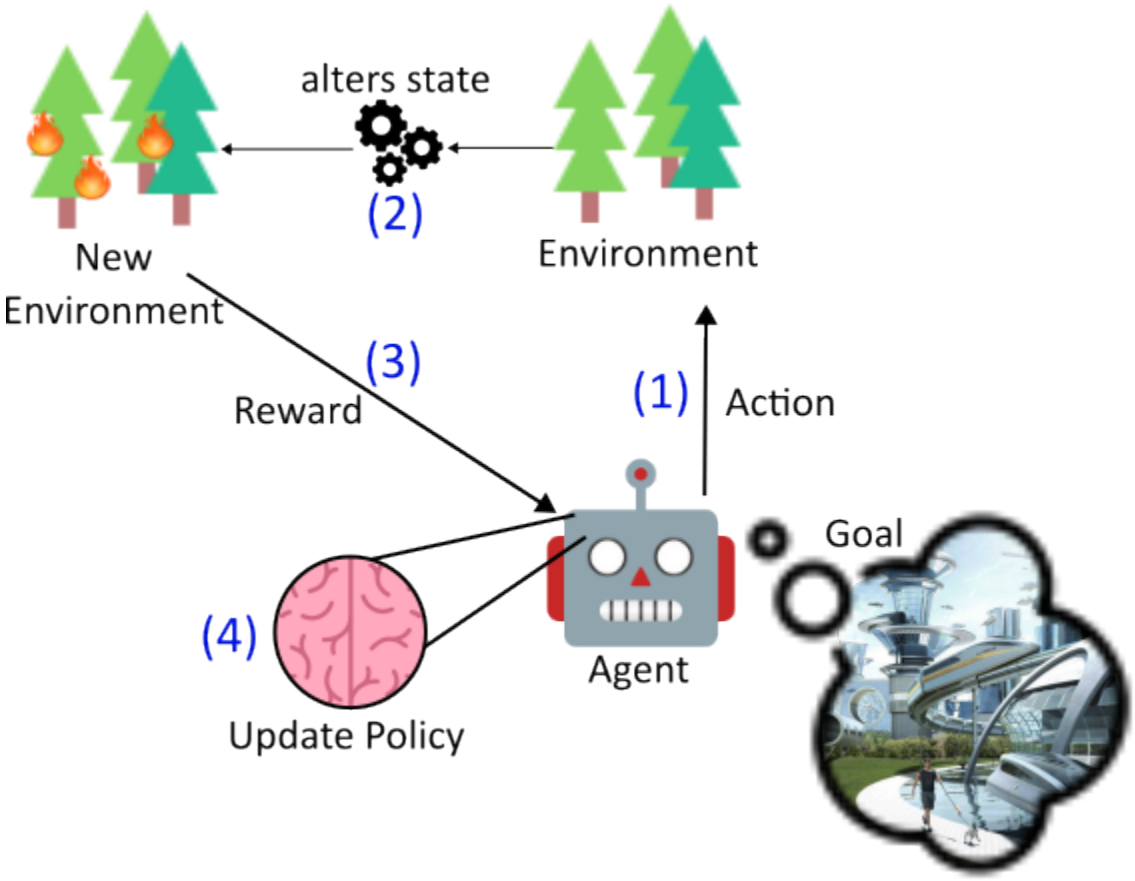

Figure 3.1: Natural Language Generation
(1) The agent selects and performs an action that is available to it in the current state. (2) The action results in a new state. (3) The agent receives feedback from the environment in the form of a reward or penalty. (4) The agent updates its policy based on the reward to perform better.

probabilities of lexicalized trees over text corpora. The reward function R maps actions from any state to a real-valued number that describes the "goodness" of the resulting state. Finally, the discount factor $\gamma \in [0,1]$ is used to scale future rewards, realizing an agent's preference for immediate rewards above future ones. In this domain, a stable environment and lack of time preference means that it is sensible to set $\gamma = 1$.

The search begins with the initial empty state ϵ. From this starting state, STRUCT can substitute any initial tree from the grammar. So, the next state could

be something like "(S (NP) (VP))," intuitively meaning "the sentence has a noun phrase followed by a verb phrase." Since there are no words in our state yet, there can be no associated FOL semantics. Another action could be the substitution of a noun tree anchored with "dog" to form "(S (NP (N dog)) (VP)," which parses as the incomplete sentence "dog" and FOL semantics $\exists\, x\, \text{dog}(x)$. From this state, the possibility for more actions opens up. Two such actions would be the addition of the determiner "the" and the adjective "brown," both adjunctions. This produces "(S (NP (D the) (NP (N (A brown) (N dog)))) (VP))," with semantics $\exists\, x\, \text{dog}(x) \wedge \text{brown}(x)$. This process continues until STRUCT can obtain no better expected future reward by performing an action, or the search is cut off prematurely. STRUCT is an "anytime algorithm," meaning that whenever the algorithm is terminated, STRUCT can return a syntactically valid answer.

3.1 UCT Algorithm

STRUCT uses a Monte Carlo tree search (MCTS) to plan in the MDP. MCTS is a stochastic method for heuristically exploring the decision process. MCTS is necessary because the number of possible states is enormous, and STRUCT could never explore all of them, even for very simple goals. Furthermore, a single goal means that exploring many of them is unnecessary. There are four steps in MCTS:

1. Begin at the root of the tree (the current state), and continue expanding child nodes until a leaf node (a child with no explored children) is reached. This step can be biased to expand towards the most promising nodes.

2. From the leaf node, apply one or more valid actions, and choose a child node from one of them.

3. Complete a random policy *rollout* from the child node until a terminal node

(or a specified depth) is reached. The rollout could be uniform random, or a more informed procedure.

4. Back up the reward from the rollout to update nodes on the path back to the root.

Algorithm 3.1: Monte Carlo Tree Search

Input: s_0 root state, d depth limit, γ discount
Output: best action from root

$\tau_0 \longleftarrow \texttt{BuildTree}(s_0)$
repeat
  $s \longleftarrow \tau_0.state$
  $r \longleftarrow 0$
  $\tau \longleftarrow null$
  while $\texttt{Nonterminal}(s)$ **do**
    $a \longleftarrow \texttt{SelectAction}(s)$
    $\hat{s} \longleftarrow \texttt{Transition}(s, a)$
    $r \longleftarrow r + \mathcal{R}(s, a)$
    if $\tau.children[a][\hat{s}] = null$ **then**
      $\tau.children[a][\hat{s}] \longleftarrow \texttt{BuildTree}(\hat{s})$
      break
    $\tau \longleftarrow \tau.children[a][\hat{s}]$
    $s \longleftarrow \hat{s}$
  $r \longleftarrow r + \gamma \cdot \texttt{PolicyRollout}(s, d)$
  while $\tau \neq null$ **do**
    $\tau.reward \longleftarrow \tau.reward + r$
    $\tau.count \longleftarrow \tau.count + 1$
    $\tau \longleftarrow \tau.parent$
until $\texttt{Timeout}()$
return $\arg\max_{a \in \mathcal{A}} \frac{\sum_{\tau' \in \tau.children[a][\cdot]} \tau'.reward}{\sum_{\tau' \in \tau.children[a][\cdot]} \tau'.count}$

The important part to notice is that the "select action" step of MCTS can be parameterized in many different ways. The trick is to balance exploration and exploitation: we want to explore enough of the search tree to find a good reward, but also exploit the best reward we've found thus far. For instance, it might be worthwhile to to explore an action with smaller reward but more unexplored states

to take advantage of the fact that we could encounter an even better reward than we've seen up to this point. However, after a sufficient amount of exploration, the likelihood that we will find a better reward diminishes, so it is better to take advantage of our best policy. UCT solves what is known as a *multi-armed bandit* problem. The motivation is that you are in a casino, standing in front of k slot machines. Every time you play a slot machine, you receive a reward sampled from that machine's unknown probability distribution. The goal is to obtain as great a reward as possible with a fixed number of slot pulls. Notice that, if we knew the machine with the highest expected reward, we would always pull that machine. But we can only estimate the expected reward of each machine by repeated pulls. If we calculate the cumulative expected reward from pulling some machines a total of k times and subtract that from the expected reward of pulling the best machine k times, we get a quantity known as regret. Regret minimization is one way to measure the goodness of a bandit algorithm.

To formalize this notion, STRUCT uses an upper confidence bound (UCB) [3] to select actions. That is, with high probability, we can know that the true expected payoff of an action is less than the upper bound. The upper bound is computed as

$$\hat{y}_i + \beta \sqrt{\frac{2\ln(n)}{n_i}}$$

where $\hat{y}_i$ is the average reward of machine i thus far, n_i is the number of times machine i was played previously, n is the total number of machine plays thus far, and β is a constant. As it turns out, always playing the machine that maximizes an expression of this form results in the smallest expected regret of any policy under uncertainty [3].

The Upper Confidence Bounds applied to Trees (UCT) algorithm [19] is a modified MCTS that uses UCB to select actions.

Algorithm 3.2: UCT

Input: State *state*, Timeout T

while $time < T$ **do**
  if *state has unexplored actions* **then**
    $action \leftarrow$ `GetRandOpenAction(state)`
  else
    initialize *policy*
    $actions \longleftarrow$ `GetValidActions(`*state*`)`
    for *action in actions* **do**
      $policy[action] \leftarrow avgRewards[s,a] + c\sqrt{\frac{\ln(stateVisits[s])}{stateActionVisits[s,a]}}$
    $action \leftarrow \arg\max(policy)$
  $nextState, reward \leftarrow$ `SimulateAction(`*state*$,$*action*`)`
  $q \leftarrow reward + \gamma \cdot$ `Explore(`*nextState*$,1$`)`
  `UpdateValue(`*state*$,$*action*$,q,0$`)`
return bestAction(*state*,0) `// The best action from` *state* `at depth 0`

3.2 STRUCT Algorithm

STRUCT leverages the UCT algorithm to explore actions over the tree grammar, with a goal of realizing English text whose meaning most closely matches the semantics of the communicative goal. This section lays out the foundations of the algorithm and provides intuition on certain design choices. It should be noted that previous research has yielded many variations of STRUCT with mixed results. The most impactful variations are touched upon briefly, but for the sake of brevity, focus stays on the essential building blocks. Furthermore, STRUCT (prior to this work) has always made the assumption that the semantics of the world, goal, and grammar are purely conjunctive (predicates connected with logical AND). Emphasis is placed on which components take advantage of this assumption to simplify the search procedure and reduce runtime.

3.2.1 Actions

Every action at STRUCT's disposal is either a substitution of an initial tree, or an adjunction of an auxiliary tree as discussed in Section 2.2.1. Through these actions, STRUCT incrementally alters the meaning of the current sentence. The procedure for action selection is illustrated in Algorithm 3.3.

Algorithm 3.3: getAction

```
   Input: Grammar R, State state, NumTrials N, Lookahead D
 1 for N do
 2     testState ← state
 3     if testState has unexplored actions then
 4         action ← pick with open action policy
 5     else
 6         action ← pick with tree policy
 7     testState ← applyAction(action, testState)
 8     depth ← 1
 9     while depth < D do
10         action ← sample PLTAG tree from R
11         testState ← applyAction(test, State)
12         depth ← depth + 1
13         reward ← calcRewardBindings(testState)
14     associate reward with first action taken
15 action ← action with max associated reward
```

At every iteration, a candidate action is selected from the set of all valid ("open") actions for the current state. Initially, unexplored actions are sampled at random. The weights of the multinomial distribution for random action selection can be weighted by the probabilities associated with the PLTAG trees. For previously explored actions, the tree policy (Algorithm 3.1) is used, balancing the preferences of exploring less-seen actions, and exploiting the best actions thus far.

3.2.2 World Pruning

The communicative goal for STRUCT is only a subset of its total knowledge, taken from the world. The world lists all entities and relations that are known to STRUCT before generation. STRUCT tailors its generations based on their satisfiability and specificity with respect to the world: utterances should not be contradicted by the world, nor be overly ambiguous. As the size of the world increases, it is more computationally expensive to generate referring expressions for entities. To alleviate this, it is natural to consider ways to limit the attention of STRUCT to only parts of the world.

The entities in the goal are a subset of all of the entities in the world. The world may contain lots of information that is either only indirectly related to the goal, or not related at all. Consider the toy goal "The dog chased the cat." The goal references a specific named entity, but omits many other modifiers. This goal makes no mention of the fact that, say, this particular dog is a 9-year-old golden retriever named Chelsea. But assuming the world contains mention of multiple dogs, some or all of this information may be necessary to uniquely refer to Chelsea. Alternatively, a separate fact in the goal asserting that "the mitochondria is the powerhouse of the cell" is superfluous to the task at hand. The latter can simply be excised from the world entirely, while the former is taken into account during referring expression generation.

3.2.3 Grammar Pruning

STRUCT's grammar—the set of all available annotated XTAG trees— needs to be as large as necessary to provide expressivity in generating a myriad of utterances. However, the "ideal" derivation tree for a communicative goal contains only a handful of these trees. It would not be sound to tell STRUCT exactly the trees and lexicalizations necessary to represent the goal, as this information would not

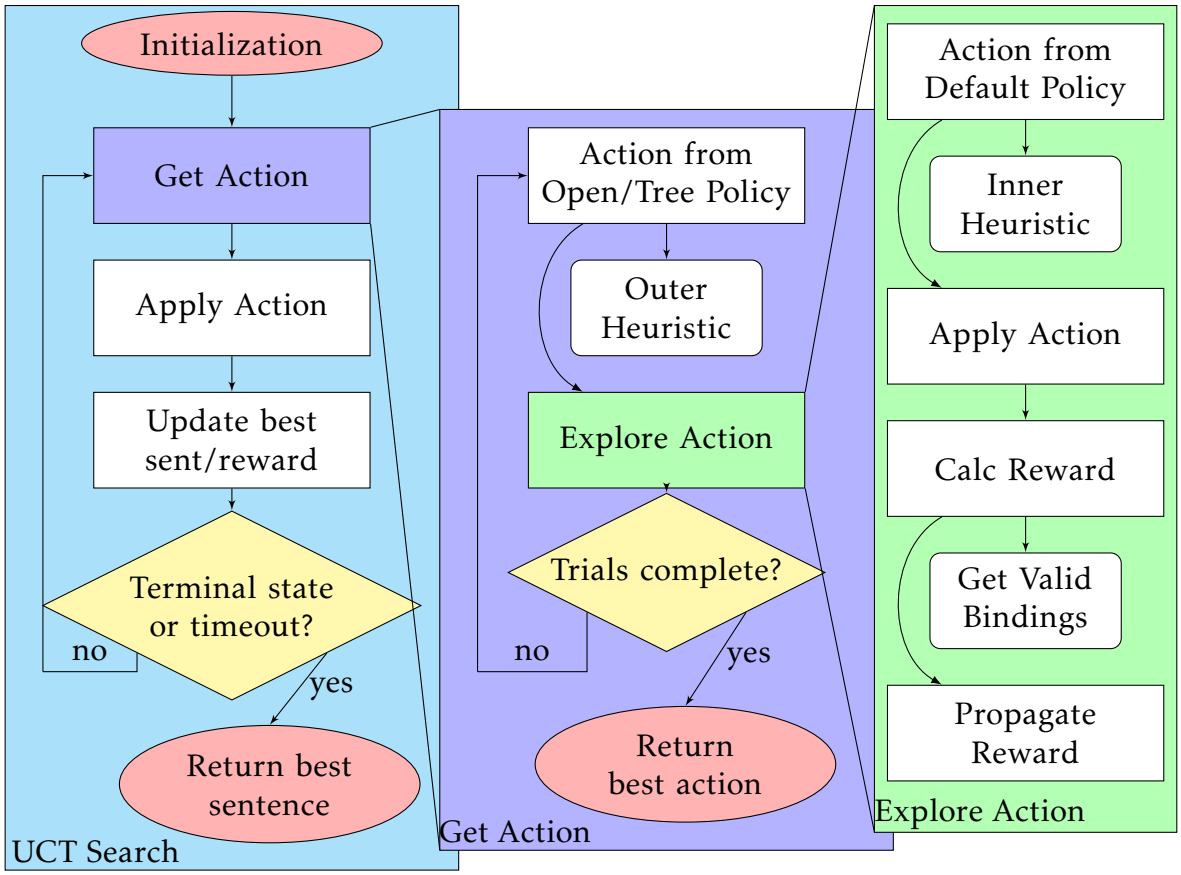

Figure 3.2: STRUCT flow of execution.

be available outside of testing. But grammar pruning, similar to world pruning, is a means of eliminating trees that are not essential to generation. Algorithm 3.4 details the process for pruning the grammar.

Algorithm 3.4: pruneGrammar

Input: Grammar R, World W, Goal G

$G' \leftarrow \emptyset$
$visited \leftarrow \emptyset$
$queue \leftarrow G$.entities
while $|queue| > 0$ **do**
    $e \leftarrow queue$.dequeue()
    $neighborhood \leftarrow$ {relations of G in which e is an argument}
    $G' \leftarrow G' \cup neighborhood$
    $neighbors \leftarrow neighborhood$.entities
    $queue += neighbors - visited$
    $visited$.append(e)
$R' \leftarrow \emptyset$
for $tree \in R$ **do**
    if *tree fulfills semantic constraints* ***or*** $tree.relations \subseteq G'.relations$ **then**
        $R' \leftarrow R' \cup \{tree\}$
return R'

First, STRUCT generates what is known as the closure of the goal. The closure contains every relationship that involves at least one goal entity. Adding relations in this manner can introduce new entities into the closure, so the procedure is iterated until there are no more relations left to add. Thus, the closure contains every predicate that can be used to distinguish a goal entity. Then, the semantics of every tree is compared with the semantics of the closure, and trees that do not add relevant information are discarded. There are many additional trees that should not be pruned from the grammar. These include trees that combine the semantic representation of other trees in different ways, such as a tree anchored with a comma used to connect two clauses together.

3.2.4 Reward

In order to reach the goal, STRUCT must explore many intermediate states. To inform the search, STRUCT needs a reward signal that represents how well the current state represents the goal. But, the utterances produced by tree substitu-

tion and adjunction do not directly model the entities in the goal. That is to say, for instance, a noun phrase tree introducing the phrase "the dog" into the state contributes semantics like $\exists x\, dog(x)$, which does not reference a named entity. World entities must be bound to the free variables for direct comparison with the goal. Then, the reward can be calculated as in Algorithm 3.5 by counting the number of predicates shared between the expressions, minding entities, order, and arity.

Algorithm 3.5: calcReward

Input: Partial Sentence S, World W, Goal G

$score \leftarrow 0$
$bindings \leftarrow getValidBindings(S, W)$
if $|bindings| > 0$ **then**
    $binding \leftarrow bindings[0]$
    $S \leftarrow$ apply $binding$ to S
    $score \mathrel{+}= C_1|G.relations \cap S.relations|$
    $score \mathrel{-}= C_2|G.conds - S.conds|$
    $score \mathrel{-}= C_3|G.entities \ominus S.entities|$
    $score \mathrel{/}= C_4|bindings|$
    $score \mathrel{-}= C_5|S.sentence|$
return $score$

A number of penalties are assessed to the reward. The reward is divided by the total number of world bindings. The motivation is that the more bindings there are, the more ambiguous the state is. STRUCT should be as specific as the world allows. Also, the reward is reduced by the length of this state's utterance. This is based on a heuristic similar to Occam's razor in that, between otherwise semantically equivalent sentences, STRUCT should prefer the shortest one. It should be noted that sentence length need not correlate with other desirable properties of utterances, like coherence and style. Such considerations would be handled further along in the NLG pipeline.

3.2.5 Pruning and Caching Bindings

Before this reward is calculated, STRUCT must determine all the ways the partial semantics can be bound with world entities. This is a combinatorial problem: there are $\binom{N}{K}$ bindings of N world entities into K free variables. Fortunately, there are ways of reducing the computational strain, the most straightforward of which is caching bindings to avoid repeated calculations when different tree states produce the same semantics. In the setting of conjunctive-only bindings, another modification can be made by realizing that, if a specific binding is invalidated for one state, then it can never become valid again in a future state, as future states will only be more specific.

Algorithm 3.6: getValidBindings

Input: Partial Sentence S, World W

```
validBindings ← emptyset
k ← min(|s.entities|,|W.entities|)
for worldEntities in {length k permutations of W.entities} do
    bindings ← mapping of {S.entities → worldEntities}
    S' ← applyBinding(binding, S)
    if S' is consistent with W then
        validBindings.add(binding)
return validBindings
```

With a more general FOL semantics, the set of all bindings cannot be computed efficiently because first-order logic satisfiability is undecidable.

Distributional Heuristics

The size of the search space and the difficulty of binding checking are major hurdles in scaling up STRUCT. It is natural to look to circumvent these issues by means of heuristics, where a degree of accuracy is sacrificed for efficiency. The introduction of distributional semantics is a potential solution. Rather than solely reasoning over FOL representations of meaning, STRUCT could work with word

vectors. Under this new regime, several things change. Entity relations, previously modeled via predicates, are now represented with the word vectors of the predicates themselves. If the word vectors obey some form of compositionality, then distributional meanings of the states and the goal itself can be constructed by composing the vectors of the individual relations. Now, the binding problem reduces to finding the world entity vectors that are most similar to the vectors of the free variables in the current state. A suitable choice of data structure (e.g., a k-d tree) allows for efficient lookup of these vectors. The reward, then, is obtained via cosine similarity between the state's vector and the goal's.

Chapter 4

STRUCT with First-Order Logic

A major shortcoming of the current STRUCT system is that it is not suited to handle non-conjunctive Boolean connectives. That is to say, STRUCT assumed that all goals and facts in the world are lists of predicate functions connected by logical "and." This assumption is built into the reward and bindings calculations discussed in Sections 3.2.4 and 3.2.5, respectively, and allows for many of the shortcuts and performance increases STRUCT enjoys. This section details Disjunctive/Implicative STRUCT (diSTRUCT), a re-factoring of STRUCT to properly handle non-conjunctive semantics like negation ("not"), disjunction ("or"), implication ("if") and double implication ("if and only if"). A new reward function and bindings cache compatible with this new regime are proposed and evaluated. Also, additional enhancements are made to the system with the goal of increasing expressivity and practicality of use.

It is worth commenting on what is meant, specifically, by "goals with non-conjunctive semantics." In this work, such goals represent uncertainty, either in an agent's knowledge about the world, or in a broader, epistemological sense. That is to say, an agent might be aware of the existence of a particular dog $d1$, but is uncertain of its exact breed, and only knows enough to say with certainty that it is

either a golden retriever, or a dachshund. In this case, such a factoid might appear as $\exists\, d1\ \text{dog}(d1) \wedge (\text{golden\_retriever}(d1) \vee \text{dachshund}(d1))$. This would in principle require the world to be represented by uncertain facts [7]. However, reasoning within such first-order probabilistic worlds can be undecidable in the worst case. Furthermore, the absence of a co-reference resolution pipeline makes it difficult to perform inference to deduce new information about the world during search. For example, if a dog is mentioned in two different sentences, it can be difficult to determine if they are in fact the same dog, as human speakers use context clues not directly captured in utterances. And without access to some global knowledge base, who is to say whether rules introduced in text apply to only a specific dog or to all dogs across all texts? Instead, this work concerns itself with the generation of sentences that are in agreement with a particular logical form.

4.1 Non-conjunctive Semantics

diSTRUCT, unlike STRUCT, handles the introduction of logical OR and logical NOT in the world and goals. Notice that once diSTRUCT can handle these operators, it can handle any logical formulae since these operators are functionally complete. It follows that the same is possible with only logical AND and logical NOT, but the introduction of logical NOT (as well as implication) allows for more natural-sounding utterances. It should be noted that logical OR and XOR are indistinguishable in conversation without context clues (e.g., "The dog barked or chased the cat" may be inclusive, but "Bob is dead or alive" is certainly not). As this is outside of the scope of this work, "or" is assumed to be "inclusive or" wherever it appears, and exclusion is possible but less concise (e.g., "Bob is dead or alive, and Bob is not both dead and alive").

4.1.1 Logical OR

Logical "or" in the world is interpreted to represent uncertainty. That is to say, if the fact "The dog or the pig chased the cat" is asserted to be true, then there are three possible models:

1. The dog chased the cat.
2. The pig chased the cat.
3. The dog and the pig both chased the cat.

However, diSTRUCT is penalized for generating sentences that are not logically entailed by the world. In this example, none of the possible models are entailed by the world, forcing diSTRUCT to only generate sentences that are as strong as the world permits. That is to say, if the world is only strong enough to assert that either the dog chased the cat or the pig chased the cat, allowing the system to generate "The dog chased the cat" is bad because this need not be true of the world.

4.1.2 Logical NOT and the CWA

STRUCT's adherence to the closed-world assumption (CWA) means that logical negation should follow naturally: there is no difference in logic between a negated literal in the world and an absent literal. In practice, however, STRUCT has no mechanism to explicitly interpret or generate negated utterances. Furthermore, STRUCT would be forced to generate prohibitively large sentences to satisfy a negated goal. Consider a world of N dogs, of which $N - \epsilon$ have bushy tails for $\epsilon << N$, with a goal involving the unique identification of one of the dogs without a bushy tail. For a sufficiently large N, the number of distractors could be huge (e.g., "the dog with a brown wavy coat that is longer than 2 inches and shorter than 3 inches and blue eyes and floppy ears and..."). The ability to instead say "the dog

without a bushy tail" ($\exists\, x\, \neg$has_bushy_tail(x)) allows diSTRUCT to only concern itself with disambiguation among the ϵ entities without bushy tails.

Recall also the core algorithm of STRUCT's binding checking: traversing the entity-relation hypergraph of the goal and world to determine which entities satisfy which predicate functions. However, this approach breaks down with the introduction of logical NOT. For starters, adapting this approach to handle negation means explicitly encoding negated predicates as relations in and of themselves on the hypergraph. That is to say, "dog" and "not_dog" are both their own relations. To find entities that aren't dogs, simply figure out which entities enter into the "not_dog" relationship with other entities. The problem is that, typically, the attributes that an entity does have will be dwarfed by the allowed attributed in the world that it *could* have. Consequently, for every world entity, there is a combinatoric explosion of negated relations added to the graph that makes computation infeasible. As discussed later, this is a major reason that this approach was substituted with the resolution engine of Prolog.

4.1.3 Implication

Logical implication can be built from the operators already discussed:

$$x \rightarrow y \equiv \neg x \vee y.$$

Despite this logical equivalence, it has its own place in natural language. For instance, a sentence like "if the dog chased the cat, the dog is tired" ($p \implies q$), is logically equivalent to "the dog didn't chase the cat, or the dog is tired" ($\neg p \vee q$) and "it is not true that the dog chased the cat and the dog isn't tired" ($\neg(p \wedge \neg q)$), but the latter two representations are difficult to decipher. It is thus preferable to guide search towards the first representation.

4.2 Bindings in diSTRUCT

The ability to determine what world entities can be bound to free variables in a statement such that the resulting proposition is entailed by the world is crucial to the functioning of STRUCT. Goal bindings allow for semantic comparison between an expression and the goal, and world bindings reveal the ambiguity of a statement. The Prolog resolution engine is leveraged to find entities satisfying one or more predicate functions, which may be negated.

In its current form, STRUCT is able to generate sentences like "The dog which chased the grey cat has long fur." An example of a potential generation path is:

The dog.

The dog has fur.

The dog has long fur.

The dog which chased the cat has long fur.

The dog which chased the grey cat has long fur.

At the beginning, STRUCT arrives at the partial sentence "The dog," introducing accompanying semantics $\exists\, x\, \text{dog}(x)$. The binding checker reports that there are many world entities which may be bound to the free variable x (i.e., the world contains many dogs). STRUCT receives a positive reward for satisfying the "dog" predicate in the world, and some negative reward for the number of bindings. The next partial sentence, "The dog has fur," now satisfies both the "dog" and "fur" predicates in the goal, while having a smaller number of bindings: there cannot be more dogs with fur than there are dogs. There are fewer dogs with long fur, fewer yet that have chased cats, and fewer yet that have chased grey cats. At every iteration in the search, the number of candidates for x is non-increasing. Suppose then, at the beginning of the search, it is determined that some world entity $x^{'}$ is not a dog, making the binding $x \rightarrow x^{'}$ inconsistent. Because the set of consistent

bindings either shrinks or stays the same, $x \to x'$ will never again be consistent with the world. This is intuitively pleasing: a non-dog can't be a dog, or a furry dog, or a furry dog that chases cats. STRUCT capitalizes on this observation by never again checking any bindings that have been invalidated before. As it turns out, such a strategy falls apart with non-conjunctive semantics. Consider a new communicative goal:

The dog or the pig chased the cat.

Starting with the same initial partial sentence, the set of bindings consists of all dogs in the world, which excludes x'. Supposing that the next partial sentence is "The dog or the pig," the semantics are $\exists\, x\, \text{dog}(x) \vee \text{pig}(x)$, and the valid bindings for x are all dogs and pigs in the world. If x' is indeed a pig, then $x \to x'$ is once again a valid binding. However, it was noted that STRUCT will never again reconsider this binding. Thus, the introduction of disjunction disallows this pruning strategy. A similar argument applies to negation. This presents an issue: the binding problem is combinatoric ($\binom{n}{r}$ ways to bind n world entities to r free variables), and this binding pruning strategy was a major defense against combinatoric explosion. Unfortunately, the undecidability of first-order logic dictates that there is no algorithm more efficient than checking bindings brute-force. There are several recourses, which are explored in diSTRUCT:

1. Sample a random subset of the possible bindings as a heuristic estimate.
2. Recursively use previously-computed bindings when building up new bindings.
3. Use good search heuristics to avoid excessive binding calculation.

Idea (1) appears in the reward pseudocode in Section 4.3. Idea (2) forms the basis for the caching of Prolog's bindings, and (3) is explored in the context of distributional heuristics.

4.2.1 Prolog and Undecidability

As mentioned, no efficient method exists for the general bindings problem as a consequence of the undecidability of first-order logic. However, Prolog operates over Horn clauses, a decidable fragment of first-order logic. This means that Prolog is by itself inadequate to solve this problem. Indeed, consider a simple disjunction of two unnegated literals:

$$p \vee q$$

This cannot be written as a Horn clause, which may only contain a single unnegated literal. But previously, the Prolog rule

$$dog(X) \models brown(X); black(X).$$

was provided, containing two unnegated literals in the statement body. How can this be? The answer is that calling the semicolon a "disjunction" is misleading. The above statement was already said to be equivalent to

$$dog(X) \models brown(X).$$
$$dog(X) \models black(X).$$

When Prolog is asked to check if $d1$ is a dog, it first encounters the rule $dog(X) \models brown(X)$. It then attempts to resolve $brown(d1)$. If $brown(d1)$ is true, Prolog can stop and return true. If not, it moves on to the rule $dog(X) \models brown(X)$ and does the same thing. This is exactly what the semicolon tells Prolog to do. The subtle point to notice is that this procedure is indifferent to the "disjunction" and will proceed the same if both $brown(d1)$ and $black(d1)$ are asserted separately, like they

would be in a conjunction. The result is that Prolog returns bindings which entail $p \wedge q$, even when the world is only strong enough to entail $p \vee q$. Since these "conjunctive" bindings are a superset of the "disjunctive" bindings (any binding that makes $p \wedge q$ true must also make $p \vee q$ true), explicit theorem proving via a truth table is used to validate the Prolog world bindings before the reward calculation.

4.3 Truth Table Reward

Once bindings are computed, a reward signal is necessary to give STRUCT feedback on how close the current state is to the communicative goal. The new algorithm is built around the previous algorithm, but makes some crucial changes. It is no longer enough to simply compare the number, arity, and arguments of the predicates that appear in the current state and in the goal, because there are many different FOL statements that are indistinguishable under this metric. Instead, the logical extension is to compare the truth tables of the goal and the current state, computed over the union of their entities. The larger the norm is of the difference of the truth tables, the less similar they are. This is used to scale the old reward. In addition, a "syntactic penalty" is assessed that penalizes STRUCT for generations that don't match the syntax that is desired in the goal. More concretely, if the goal is "If the dog chased the cat, then the dog is tired," another logically-equivalent way of saying this is "The dog didn't chase the cat or the dog is tired," but it is preferred that STRUCT maintain the explicit implication. This way, generation control can be implemented by changing the structure of the goal logic.

To create truth tables for the goal semantics and the semantics of a candidate utterance, all literals are replaced with variables, such that two literals are assigned the same variable if the name, arity, and arguments of the literal all match. The distance between the two truth tables is the (L0) norm of the logical XOR of the

result columns. This effectively counts the number of models that differ between the tables.

The reward is scaled down by the difference between the number of world bindings and the number of bindings found in the reward calculation. Originally this was a simple penalty on the number of world bindings, but diSTRUCT should not be penalized for being ambiguous when the goal itself is also ambiguous. Since there can be very many bindings, a controllable hyperparameter is introduced to only sample some of the bindings as a heuristic estimate. This is the "k" that appears in line 14.

Algorithm 4.1: calcReward2

Input: Partial Sentence S, World W, Goal G, max bindings k

1 $score \leftarrow 0$
2 $goal\_bindings \leftarrow getPrologBindings(S, G)$
3 **for** $gb \in goal\_bindings$ **do**
4 $\quad temp \leftarrow 0$
5 $\quad S^{'} \leftarrow$ apply gb to S
6 $\quad temp \mathrel{+}= C_1|G.relations \cap S^{'}.relations|$
7 $\quad temp \mathrel{-}= C_2|G.conds - S^{'}.conds|$
8 $\quad temp \mathrel{-}= C_3|G.entities \ominus S^{'}.entities|$
9 $\quad temp \mathrel{-}= C_4 \cdot |truthTableDistance(S^{'}, G)|$
10 $\quad score \leftarrow \max(score, temp)$
11 $score \mathrel{-}= C_5|S.sentence|$
12 $score \mathrel{-}= C_6 \cdot |syntacticPenalty(S, G)|$
13 $world\_bindings \leftarrow getPrologBindings(S, W, k)$
14 $score \mathrel{/}= \min(max(1, |\#world\_bindings - \#getPrologBindings(G, W)|), k)$
15 **return** $score$

Algorithm 4.2: truthTableDistance

Input: Semantic State $S1$, Semantic State $S2$

$S1, S2 \leftarrow$ replace predicates in S1 and S2 with variables
$numVars \leftarrow [0] * \#(\{S1.relations\} \cup \{S2.relations\})$
$T1 \leftarrow [0] * numVars$
$T2 \leftarrow [0] * numVars$
$i \leftarrow 0$
for *case* **in** $\bigtimes_{i=1}^{n} [TRUE, FALSE]$ **do**
    $T1[i] \leftarrow$ evaluate S1 at case
    $T2[i] \leftarrow$ evaluate S2 at case
return $\|T1 \oplus T2\|$ \\Element-wise XOR

An important detail about the new reward algorithm is that a maximum is taken over all goal bindings, instead of just applying the first one. This is because, in the old formulation, there was only ever one binding consistent with goal entities, but the uncertainty introduced by disjunction means there can be several. Unfortunately, the simplifying assumptions made to avoid explicit theorem proving (which is not tractable) means that the Prolog bindings are a superset of the actual valid goal bindings. Valid goal bindings will always have a better reward, so the maximum will pick them out.

Algorithm 4.3: syntacticPenalty

Input: Semantic State $S1$, Semantic State $S2$

$ops1 \leftarrow [0, 0, 0]$
$ops2 \leftarrow [0, 0, 0]$
$i \leftarrow 0$
for *op* **in** $[\vee, \neg, \implies]$ **do**
    $ops1[i] \leftarrow \#(x \text{ for } x \text{ in } S1.operators \text{ if } x = op)$
    $ops2[i] \leftarrow \#(x \text{ for } x \text{ in } S2.operators \text{ if } x = op)$
    $i \leftarrow i + 1$
return $\sum |ops1 - ops2|$

The syntactic penalty is motivated by the idea that there are many semantically-equivalent ways to say the same sentence, but only some are ideal. Specifically, diSTRUCT is encouraged to produce a sentence whose semantic form is closely

aligned with the goal's semantics. To accomplish this, a penalty is added for every logical operator that appears in the goal and not in the semantic state, and vice versa.

Algorithm 4.4: getPrologBindings

Input: Semantic State S, max bindings k

1 $relations \leftarrow getRelations(S)$
2 $query \leftarrow prologFormatting(relations)$
3 $bindings \leftarrow RESOLUTION(query)$
4 **return** $randomSample(bindings, \max(k, |bindings|))$

In the first line of *getPrologBindings*, the relations are extracted from the partial sentence. These relations are any predicates that appear in S, including whether or not each is negated. Keep in mind that at this stage of generation, the variables in S are unbound, so they do not reference any labeled entities. Since the disjunctions are not present in the relation set, the resulting bindings may not all be valid. This is OK; Prolog is not capable of finding all satisfiable bindings of an arbitrary CNF formula anyways, and this is dealt with in the main body of the reward. The relations are then combined into a single Prolog query with proper formatting. In this step, negated relations must be moved to the end of the expression. This is because Prolog uses *negation as failure*, which means that $\neg p$ is asserted to be true if an exhaustive search for p fails. Consider the FOL expression $\neg\text{fluffy}(x) \wedge \text{chased}(x, y)$. If Prolog is queried with "$\backslash + \text{fluffy}(X), \text{chased}(X, Y)$.", it will begin by trying to prove $\text{fluffy}(X)$. Assuming there is at least one fluffy thing in the world, $\text{fluffy}(X)$ resolves as true with X bound to something that is fluffy. Therefore, $\backslash + \text{fluffy}(X)$ resolves as false. Since Prolog has nowhere else to back the search up to, the query simply fails. On the other hand, if $\text{chased}(X, Y)$ has already resolved as true, Prolog can then figure out for which X it is not true that $\text{fluffy}(X)$. This is a consequence of the CWA: Prolog is evaluating the expressions against a "mini world model" that only includes facts known to the interpreter.

4.4 Caching Bindings

Caching bindings is an important part of keeping computation costs low. At its simplest, the signature of a particular FOL expression and the bindings can be cached so that if it is ever encountered again, further querying is avoided. An addition modification to caching is that it can be done recursively, so that the bindings of individual predicates (negated and otherwise) are used as building blocks to create bindings for larger expressions that contain them. A potential issue that arises is when the number of bindings for an expression is so large that computing all of them is intractable. In cases like this, a solution is to instead compute bindings for the negation of the original expression, and adjust the penalty appropriately.

An interesting problem arises with such a bottom-up procedure for caching bindings. As mentioned, the number of world bindings for some semantics can be arbitrarily large, especially with negated predicates (e.g., "How many non-dogs didn't chase non-cats?"). As a heuristic, the bindings calculation can be cut off prematurely. But if those bindings are involved in set operations to produce new bindings, the error due to missing bindings can compound. To circumvent this, one could prevent truncated bindings from being used again in downstream binding calculations, and instead simply re-query. However, if binding sizes are sufficiently large to cause this to happen, it is unlikely that a very precise set of world bindings is useful to the reward.

Algorithm 4.5: getCachedBindings

Input: Semantic State S, max bindings k

```
1  if S in cache then
2      return cache[S]
3  else if S is a unit clause then
4      cache[S] ← getPrologBindings(S, T, K)
5      return cache[S]
6  else if S is a negated clause then
7      bindings ← getCachedBindings(S, T, k)
8      res ← bindings^C
9      cache[S] ← res
10     return res
11 else if S is a conjunctive expression then
12     res_l ← getCachedBindings(S.first, k)
13     res_r ← getCachedBindings(S.second, k)
14     cache[S] ← intersect res_l and res_r
15     return cache[S]
16 else
17     \\S is a disjunctive expression
18     res_l ← getCachedBindings(S.first, k)
19     res_r ← getCachedBindings(S.second, k)
20     cache[S] ← union res_l and res_r
21     return cache[S]
```

If there is no cache entry, Algorithm 4.5 first checks if the expression is a unit clause (a single predicate). This is the base case, and the original bindings method is called. With a negated expression, the method recurses on the term and then takes the complement of the returned bindings with all possible world bindings. Note that it does not matter whether negation has been distributed here. For binary expressions, logical conjunctions and negations correspond to intersections and unions, respectively, of the cached bindings of the sub-terms, if they exist. This routine sacrifices a degree of overhead to reduce reliance on Prolog; it is likely that the speed increases, if they exist, are variable on the structure of the input.

4.5 Caching Truth Tables

The crux of the reward calculation is the truth table distance. Since truth tables do not change when renaming variables, 2 sets of expressions that are equivalent under a bijection of predicate functions have identical truth tables (the most trivial example is any unit clause having the same 2-element truth table). In a similar manner to the bindings cache, truth tables can be cached in a bottom-up fashion. By the time expressions reach the reward function, the only logical operators they contain are $\{\wedge, \vee, \neg\}$. It follows that every expression X can be written as $Q \wedge R$ or $Q \vee R$, where Q and R are themselves expressions (for unit clauses, $X \equiv X \wedge \text{TRUE} \equiv X \vee \text{FALSE}$).

4.6 Distributional Search Heuristics

The search space for STRUCT/diSTRUCT is very large, and it is important that the system has a notion of the potential viability of an action, even before that action is explored. This is realized both in the "inner heuristic" and "outer heuristic" of STRUCT. The outer heuristic selects the next action/state to consider, while the inner heuristic guides exploratory actions taken from that state.

In previous work [31], a linear perceptron was trained based on previous runs of the system. After a run of the system, the weights of the perceptron were tuned to be more favorable to the most successful actions in every intermediate state. Doing this achieved a small but noticeable improvement in reward and time to generation. However, there are a few issues with this approach. First, there is a real potential for overfitting to the types of actions that work well for the specific trees encountered in the dataset. Also, there is no guarantee that an action favorable in one state on a path to the goal is also favorable when the goal becomes something else. To this end, a pretrained large language model (LLM) [34] is leveraged to

estimate the *perplexity* of a given sentence, providing a sense of how plausible a given sentence is, syntactically.

More formally, let $X = (x_0, x_1, ..., x_t)$ be a tokenized sequence. Perplexity is the exponentiated average negative log-likelihood of X:

$$PP(X) = \exp\left\{-\frac{1}{t}\sum_{i}^{t}\log p_\theta(x_i|x_{<i})\right\}$$

where $p_\theta(x_i|x_{<i})$ is the model's parameterization of the likelihood of the i^{th} token conditioned on the preceding tokens in the context window. The larger the perplexity, the less likely it is that the given sequence was sampled from p. The assumption is that p accurately models the distribution of the English language, so that perplexity is a surrogate for the sensibility of the sentence.

Chapter 5

Experimental Results

STRUCT and diSTRUCT were both evaluated on a dataset of five buckets, each bucket consisting of roughly 100 samples. The buckets are sentences that include conjunctions, disjunctions, negations, implications, and mixed Boolean operators, respectively. Evaluation is based both on obtained reward, generation time, sentence quality, and semantic fidelity of generations.

5.1 Dataset Creation

Every data point is a single sentence. Sentences are either generated by hand or sourced from text corpora such as the Brown corpus [14] and the WSJ section of the Penn TreeBank corpus [24]. Collected sentences are then tagged with part of speech tags and fed to an LTAG parser [1]. The LTAG parser produces two trees: a parse tree and a derivation tree. The derivation tree is built from the XTAG trees that are substituted and adjoined according to the rules of the grammar to produce the final parse tree, where the internal nodes are parts of speech and the leaf nodes are words. Since the XTAG trees are annotated with FOL semantics extended with lambda calculus, α-conversion and β-reduction are performed in tandem with the tree operations, resulting in a FOL expression associated with the parse tree.

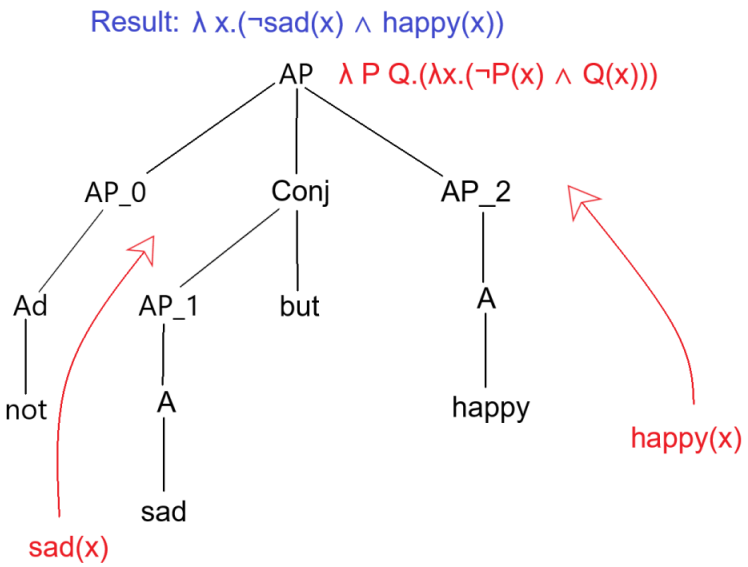

Figure 5.1: An example of the semantic annotation for the "betaARBax1CONJax2" tree from XTAG, anchored with "but+not." This tree can then be adjoined as an adjective phrase itself, resulting in something like "The dog is not sad, but happy" ($\exists x\ dog(x) \wedge \neg sad(x) \wedge happy(x)$).

The FOL semantics must be inspected to make sure they are syntactically valid, as there is no such guarantee. This is because it is difficult to map out the exact functions necessary to make the semantics valid, when a single tree can appear in many different contexts. (The need for a flexible, automatic, and content-aware semantic annotation of TAG trees is discussed as an area of future work.) Also, FOL is only sufficient to express certain fragments of natural language, and a number of linguistic constructs are out of reach. The semantics of each sentence is a goal, and the amalgamation of all goals into a single knowledge base is the world.

Previous work on STRUCT relied on hand-annotations of XTAG trees with compositional FOL semantics. In this work, the semantics for trees lexicalized with "not," "or", "nor," and "if-then," are necessary for the system to function properly. The semantic coverage is additionally expanded to a broader variety of trees with similar effects, such as "unless," "provided (that)", and "but-not" (e.g., "The dog chased the cat but not the pig"). It became necessary to support the in-

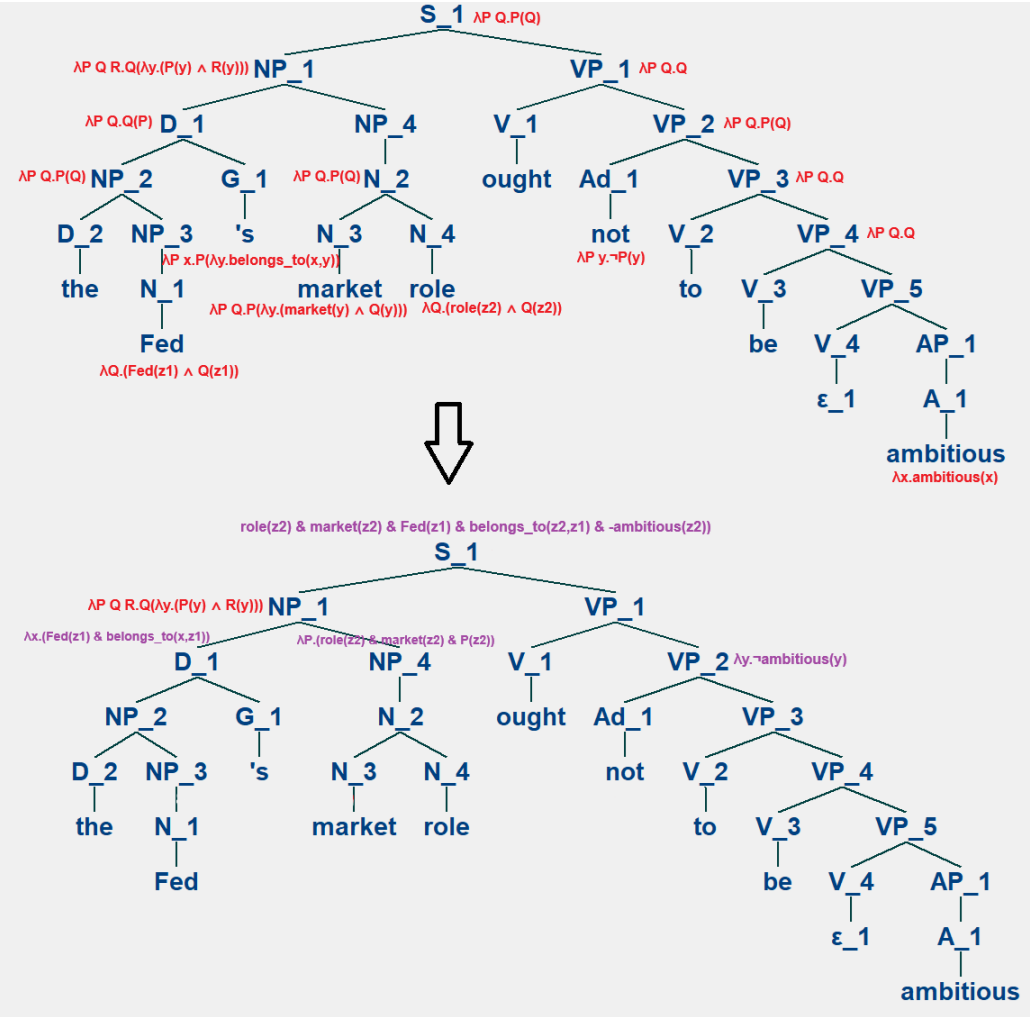

Figure 5.2: An example parse tree for the sentence "The Fed's market role ought not to be ambitious." Original semantic annotations are in red. Semantics propagate upwards towards the root. Nodes without labelings leave the semantics unchanged. Semantics created from β-reductions are shown in purple. The semantics of the entire sentence are at the root.

clusion of multi-lexicalized trees, or trees having multiple lexical anchors, to allow for the inclusion of the necessary linguistic constructs. Furthermore, it is necessary to change the semantics of trees based on the specific lexicalization, whereas before there was only one semantic annotation per tree, regardless of lexicalization. For instance, a sentential adjunct tree (betaPss in XTAG) anchored with "if" has semantics like $p \implies q$, while the same tree lexicalized with "unless" introduces semantics like $\neg q \implies p$.

5.2 Baseline STRUCT/diSTRUCT

The baseline version of both STRUCT and diSTRUCT are initialized with an exploration breadth (n) of 60 and a depth (d) of 4. That is to say, 60 rollouts are performed in each iteration, and every rollout explores 4 states deep. Both systems are run on the entire dataset. Every trial has a cutoff time of 60 seconds, so the search is cut off prematurely and the current best state is returned. This is a soft cutoff, so the program permits STRUCT/diSTRUCT to finish the iteration it is currently in before termination. Because diSTRUCT's bindings for certain expressions (especially with negation) can be computationally intractable, Prolog is capped at a maximum of 50 bindings for each query.

The quality of generations is assessed with the search score normalized by the best possible reward, as well as a ROUGE (Recall-Oriented Understudy for Gisting Evaluation) score [23]. The normalized reward score is a good indicator of whether STRUCT/diSTRUCT converges to a sentence whose reward is close to that of the goal. However, since both systems have their own reward functions, it is difficult to directly compare the quality of the generations with these scores. Instead, the ROUGE score gives an objective measure of similarity to the goal sentence by calculating co-occurrence of n-grams:

$$\text{ROUGE-N} = \frac{\sum_{gram_n \in S} Count_{match}(gram_n)}{\sum_{gram_n \in S} Count(gram_n)}$$

ROUGE scores work by comparing machine generated-text to expert reference summaries written by humans.

Another metric used to measure semantic overlap of two sentences is via the distance between their embeddings in a high-dimensional vector space. Sentence-BERT [32] uses siamese BERT models to learn semantically-meaningful vector representations of sentences. The cosine similarity between the embeddings of the goal sentence and the generated sentence gives another notion of semantic relatedness. Unfortunately this cannot be used during the search, as STRUCT/diSTRUCT cannot have access to the target sentence (they could just parrot it back as the output).

Time to maximum reward for both systems is shown in Figure 5.3. In the conjunctive, disjunctive, and negative tests, both systems quickly converge to a maximum, although diSTRUCT takes longer before terminating. This is a consequence of STRUCT having a more "lenient" surface-level reward function, while it is less straightforward to ensure semantic fidelity between generations and the world/goal.

For the implication and mixed experiments, the tail of the reward curve is more elongated, indicating that diSTRUCT had more trouble finding a good sentence. This can likely be explained away by the fact that both types of sentences will on average be more complicated than the others because they contain both disjunction and negation. Furthermore, disjunction and negation together provide the opportunity for bindings computations to slow down the system: negative bindings will usually be more numerous than otherwise, and disjoining them creates multiple possible models of the world. So, while considerable time can be shaved off simply by stopping diSTRUCT prematurely when it is within a certain distance

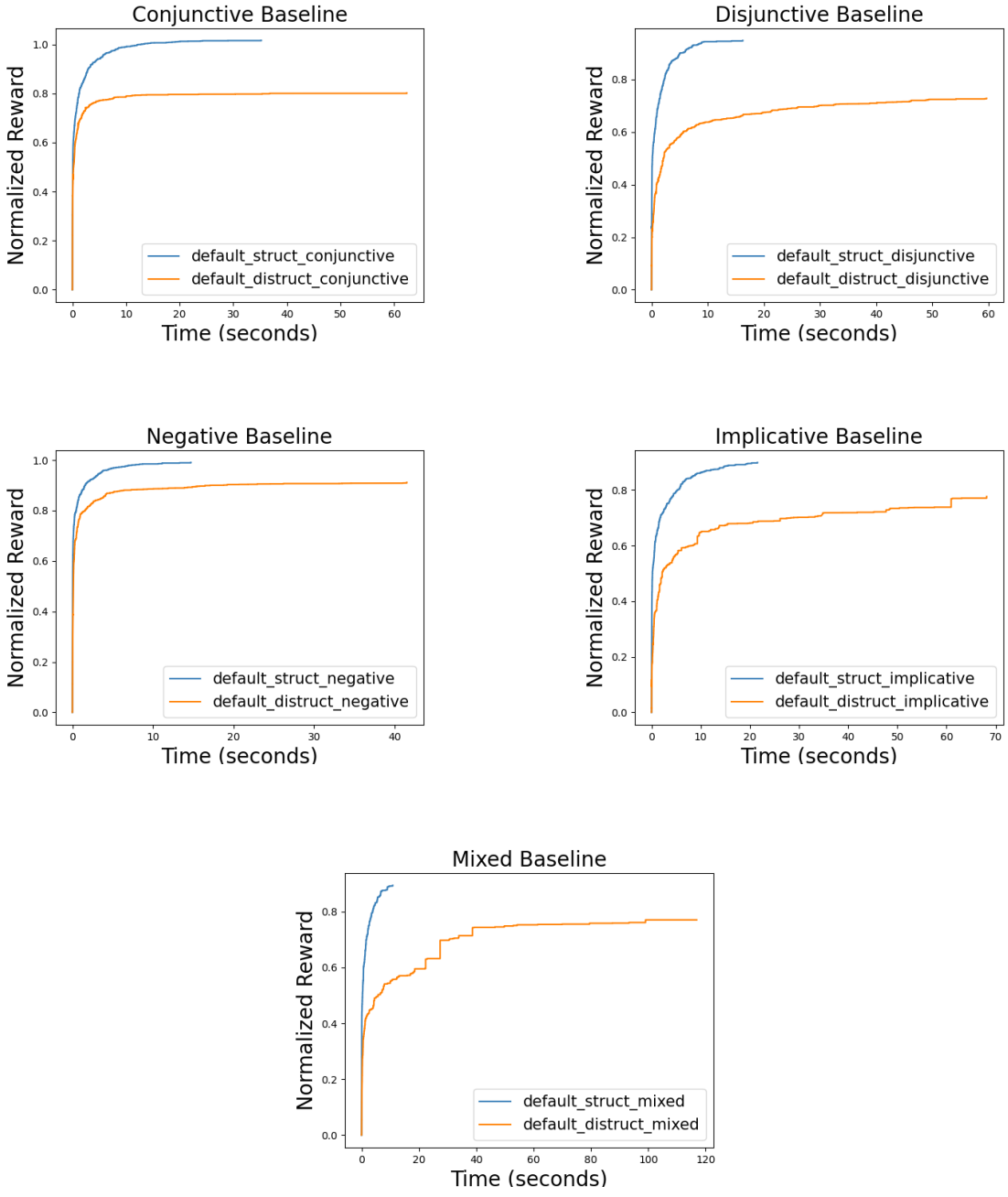

Figure 5.3: Baseline comparisons of time to maximum reward for STRUCT and diSTRUCT.

Conjunctive	Average Time to Highest Reward	Average Normalized Reward	Average ROUGE-1 Score	Average Normalized Truth Table Distance	Average Embedding Similarity
Vanilla STRUCT	4.4298	0.9118	0.7588	0.1178	0.8788
STRUCT	4.733	0.9145	0.7526	0.0158	0.8854
diSTRUCT	13.9282	0.8768	0.7518	0.0024	0.8731
Disjunctive	Average Time to Highest Reward	Average Normalized Reward	Average ROUGE-1 Score	Average Normalized Truth Table Distance	Average Embedding Similarity
Vanilla STRUCT	6.039	0.9252	0.7118	0.0888	0.8532
STRUCT	5.259	0.8977	0.7262	0.0384	0.865
diSTRUCT	24.9736	0.8121	0.6918	0.0043	0.8497
Negative	Average Time to Highest Reward	Average Normalized Reward	Average ROUGE-1 Score	Average Normalized Truth Table Distance	Average Embedding Similarity
Vanilla STRUCT	1.8695	0.9744	0.6872	0.8614	0.7395
STRUCT	1.8977	0.9535	0.768	0.0824	0.8822
diSTRUCT	6.8037	0.9344	0.7308	0.0366	0.8501
Implicative	Average Time to Highest Reward	Average Normalized Reward	Average ROUGE-1 Score	Average Normalized Truth Table Distance	Average Embedding Similarity
Vanilla STRUCT	6.5905	0.9097	0.6668	0.4827	0.8256
STRUCT	4.344	0.8463	0.6395	0.053	0.8029
diSTRUCT	21.6234	0.7824	0.587	0.0233	0.7507
Mixed	Average Time to Highest Reward	Average Normalized Reward	Average ROUGE-1 Score	Average Normalized Truth Table Distance	Average Embedding Similarity
Vanilla STRUCT	4.6489	0.8919	0.6553	0.2012	0.794
STRUCT	3.719	0.8854	0.661	0.0391	0.7805
diSTRUCT	20.8742	0.7917	0.5713	0.0088	0.7305

Figure 5.4: Average results for the baseline versions of each system, divided by logical operator. Time is given in seconds. "Vanilla STRUCT" is STRUCT without considering negation to be a part of a predicate's signature.

of the best possible reward, the Implicative and Mixed experiments show that further compute optimizations are in order.

diSTRUCT is more logically faithful to the communicative goal than STRUCT, as demonstrated by the decreased average truth table distance. ROUGE-1 scores and embedding similarity scores do not vary that much between systems and across datasets. It is not expected that ROUGE-1 scores should increase in diSTRUCT, because the semantic fidelity of the sentence does not depend on its well-formedness. But, it is nice to see that it does not dramatically decrease, indicating the quality of generations is high relative to STRUCT. Embedding similarity as measured by SentenceBERT also does not see much change, and actually decreases slightly from baseline in most of the experiments. It is difficult to discern exactly

why this happens, as the embedding space is not readily interpretable. However, the embedding space is ultimately only a surrogate for semantic similarity, and logically-equivalent utterances will not always be close.

Despite not being designed for it, baseline STRUCT is still able to generate some of the sentences with disjunctive semantics. The reasons for this are twofold:

- STRUCT was modified to interpret negation as part of the function signature. That is to say, $chased(x1, x2)$ and $\neg chased(x1, x2)$ are not counted as the same function, despite name and arity. Consequently, STRUCT can handle its own with negated semantics, even though there's no guarantee that what it says aligns with the world. The version of STRUCT without this modification is referred to as "Vanilla STRUCT."

- Grammar pruning is done to avoid an excessively large action space. When STRUCT has to choose between using "and" and "or," it can get lucky and choose the correct one. Ideally, diSTRUCT should thrive in a more logically-diverse action space, but a limitation is the added search time associated with such a space.

It is worth noting that, due to STRUCT's design, there is nothing stopping it from making false utterances if asked to do so. On the contrary, diSTRUCT's reward function forces it to be logically consistent with the world. Since experiments are conducted with subsets of the world that are always satisfiable, this phenomenon is observed less.

While diSTRUCT can beat STRUCT in terms of truth table distance, it is not as consistent at doing so as one might hope. A possible explanation is that, along the best path to the goal, there are states whose truth table-based rewards can be very bad. A simple example is any negated expression: if the goal is $-P$, then P is almost certainly an intermediate state (transformed into the goal after the adjunction of

Model Sentence	STRUCT Generation	diSTRUCT Generation
If he is arrogant, he may not be good.	he good because he arrogant	he not arrogant because he good
A flashlight or electric lantern should be available.	electric lantern and flashlight available	electric lantern available or flashlight available
Unless Ken arrives, the meeting starts without him.	meeting starts without him and Ken arrives	unless Ken arrives meeting starts without him

"not"). But the rewards of P and $-P$ are inversely correlated because they have exactly the opposite bindings of each other. Unfortunately, the simple solution to this problem—increasing the search depth—exacerbates the existing issue of long search times. The negation-aware STRUCT doesn't stumble into this pitfall as it doesn't compute bindings for negated expressions (and consequently cannot be sure that what it is saying is allowed).

Chapter 6

Conclusion and Future Work

At the beginning of this work, the building blocks of the STRUCT system are described and their strengths and weaknesses discussed. Specifically, simplifying assumptions used to prune bindings are no longer sound once the world is introduced to non-conjunctive semantics. diSTRUCT, unlike its predecessor, computes a semantic fidelity metric by means of truth tables to address the fact that comparing function names and arities is no longer sufficient. To address the increased time complexity of the system, caching algorithms are proposed for both bindings and truth tables. Distributional heuristics via a Transformer neural network are used to guide the search towards syntactically-promising sentences, to avoid wasting time searching states that are unlikely to be useful. System is done on a novel dataset tagged with an expanded semantic grammar, separated by prevalence of logical operator, and hand-checked for coherence.

Unfortunately, technical limitations and time constraints hindered some aspects of the project. Finding negated bindings does not scale well with world size, requiring binding limits and other creative heuristics to avoid overflow. Had the extent of this problem been known prior to implementation, different approaches for state representation would have been explored more thoroughly. Furthermore,

the TAG approach detailed in this work is not popular in the literature, as most authors have moved away from structured generation and towards neural generation, especially with Transformer-based models. The XTAG project at UPenn concluded in the early 2000s, and no additional releases of XTAG or XTAG-based tools have come out since then. Hand-annotating trees with semantics is laborious, not scalable, and prone to human error as it is difficult to predict how semantic annotations will behave in all scenarios.

While STRUCT/diSTRUCT achieve promising results, their scope is still quite narrow. This is the case, in part, because:

- The system cannot handle universal quantification
- Despite supporting it, the system lacks a "plug-and-play" semantic framework and is currently dependent on FOL
- The system does not have a bona-fide co-reference resolution pipeline or access to real-world knowledge bases
- The system cannot infer new facts about the world

These are all potential areas of future research, resulting in a system that seamlessly combines inference and generation over multi-modal knowledge bases.

Bibliography

[1] "A Lexicalized Tree Adjoining Grammar for English". In: *CoRR* cs.CL/9809024 (1998). URL: https://arxiv.org/abs/cs/9809024.

[2] Nicholas Allott, Terje Lohndal, and Rey Georges. "Synoptic Introduction". In: May 2021, pp. 1–17. ISBN: 9781119598701. DOI: 10.1002/9781119598732.ch1.

[3] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. "Finite-time Analysis of the Multiarmed Bandit Problem". In: *Machine Learning* 47.2 (May 2002), pp. 235–256. ISSN: 1573-0565. DOI: 10.1023/A:1013689704352. URL: https://doi.org/10.1023/A:1013689704352.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL].

[5] Daniel Bauer and Alexander Koller. "Sentence Generation as Planning with Probabilistic LTAG". In: *Proceedings of the 10th International Workshop on Tree Adjoining Grammar and Related Frameworks (TAG+10)*. Ed. by Srinivas Bangalore, Robert Frank, and Maribel Romero. Yale University: Linguistic Department, Yale University, June 2010, pp. 127–134. URL: https://aclanthology.org/W10-4416.

[6] Connor Baumler and Soumya Ray. "Hybrid Semantics for Goal-Directed Natural Language Generation". In: *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Smaranda Muresan, Preslav Nakov, and Aline Villavicencio. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 1936–1946. DOI: 10.18653/v1/2022.acl-long.136. URL: https://aclanthology.org/2022.acl-long.136.

[7] Vaishak Belle. "Statistical relational learning and neuro-symbolic AI: what does first-order logic offer?" In: *arXiv preprint arXiv:2306.13660* (2023).

[8] Bernd Bohnet and Robert Dale. "Viewing Referring Expression Generation as Search". In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence*. IJCAI'05. Edinburgh, Scotland: Morgan Kaufmann Publishers Inc., 2005, pp. 1004–1009.

[9] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: *arXiv preprint arXiv:2005.14165* (2020).

[10] Donald Davidson. "105The Logical Form of Action Sentences". In: *Essays on Actions and Events*. Oxford University Press, Sept. 2001. ISBN: 9780199246274. DOI: 10.1093/0199246270.003.0006. eprint: https://academic.oup.com/book/0/chapter/144434592/chapter-ag-pdf/45014039/book_3354_section_144434592.ag.pdf. URL: https://doi.org/10.1093/0199246270.003.0006.

[11] Juliette Faille, Albert Gatt, and Claire Gardent. "The Natural Language Pipeline, Neural Text Generation and Explainability". In: *2nd Workshop on Interactive Natural Language Technology for Explainable Artificial Intelligence*. Dublin, Ireland: Association for Computational Linguistics, Nov. 2020, pp. 16–21. URL: https://aclanthology.org/2020.nl4xai-1.5.

[12] Charles J. Fillmore and Collin F. Baker. "Frame semantics for text understanding". In: 2001. URL: https://api.semanticscholar.org/CorpusID:53006467.

[13] J. R. Firth. "A synopsis of linguistic theory 1930-55." In: 1952-59 (1957), pp. 1–32.

[14] W. Nelson Francis. "A Standard Corpus of Edited Present-Day American English". In: *College English* 26.4 (1965), pp. 267–273. ISSN: 00100994. URL: http://www.jstor.org/stable/373638 (visited on 01/13/2024).

[15] Zihao Fu et al. *A Theoretical Analysis of the Repetition Problem in Text Generation*. 2021. arXiv: 2012.14660 [cs.CL].

[16] Claire Gardent and Laura Kallmeyer. "Semantic Construction in Feature-Based TAG". In: *Proceedings of the Tenth Conference on European Chapter of the Association for Computational Linguistics - Volume 1*. EACL '03. Budapest, Hungary: Association for Computational Linguistics, 2003, pp. 123–130. ISBN: 1333567890. DOI: 10.3115/1067807.1067825. URL: https://doi.org/10.3115/1067807.1067825.

[17] A.K. Joshi, S.R. Kosaraju, and H.M. Yamada. "String adjunct grammars: I. Local and distributed adjunction". In: *Information and Control* 21.2 (1972), pp. 93–116. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(72)90051-4.

[18] Aravind K. Joshi. "Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions?" In: *Natural Language Parsing: Psychological, Computational, and Theoretical Perspectives*. Ed. by David R. Dowty, Lauri Karttunen, and Arnold M.Editors Zwicky. Studies in Natural Language Processing. Cambridge University Press, 1985, pp. 206–250. DOI: 10.1017/CBO9780511597855.007.

[19] Levente Kocsis and Csaba Szepesvári. "Bandit Based Monte-Carlo Planning". In: vol. 2006. Sept. 2006, pp. 282–293. ISBN: 978-3-540-45375-8. DOI: 10.1007/11871842_29.

[20] Alexander Koller and Matthew Stone. "Sentence generation as a planning problem". In: *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*. Ed. by Annie Zaenen and Antal van den Bosch. Prague, Czech Republic: Association for Computational Linguistics, June 2007, pp. 336–343. URL: https://aclanthology.org/P07-1043.

[21] Emiel Krahmer and Kees van Deemter. "Computational Generation of Referring Expressions: A Survey". In: *Computational Linguistics* 38.1 (Mar. 2012), pp. 173–218. ISSN: 0891-2017. DOI: 10.1162/COLI_a_00088. eprint: https://direct.mit.edu/coli/article-pdf/38/1/173/1810368/coli_a_00088.pdf. URL: https://doi.org/10.1162/COLI%5C_a%5C_00088.

[22] Mirella Lapata. "Probabilistic Text Structuring: Experiments with Sentence Ordering". In: *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*. Sapporo, Japan: Association for Computational Linguistics, July 2003, pp. 545–552. DOI: 10.3115/1075096.1075165. URL: https://aclanthology.org/P03-1069.

[23] Chin-Yew Lin. "ROUGE: A Package for Automatic Evaluation of Summaries". In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. URL: https://aclanthology.org/W04-1013.

[24] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. "Building a Large Annotated Corpus of English: The Penn Treebank". In: *Comput. Linguist.* 19.2 (June 1993), pp. 313–330. ISSN: 0891-2017.

[25] Nathan McKinley and Soumya Ray. "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)*. Ed. by Kristina Toutanova and Hua Wu. Baltimore, Maryland: Association for Computational Linguistics, June 2014, pp. 552–561. DOI: 10.3115/v1/P14-1052. URL: https://aclanthology.org/P14-1052.

[26] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[27] Amit Moryossef, Yoav Goldberg, and Ido Dagan. "Step-by-Step: Separating Planning from Realization in Neural Data-to-Text Generation". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 2267–2277. DOI: 10.18653/v1/N19-1236. URL: https://aclanthology.org/N19-1236.

[28] Niels Mündler et al. *Self-contradictory Hallucinations of Large Language Models: Evaluation, Detection and Mitigation*. 2023. arXiv: 2305.15852 [cs.CL].

[29] Nikolaos Panagiaris, Emma Hart, and Dimitra Gkatzia. "Generating unambiguous and diverse referring expressions". In: *Computer Speech Language* 68 (2021), p. 101184. ISSN: 0885-2308. DOI: https://doi.org/10.1016/j.csl.2020.101184. URL: https://www.sciencedirect.com/science/article/pii/S0885230820301170.

[30] Terence Parsons. *Events in the Semantics of English: A Study in Subatomic Semantics*. MIT Press, 1990.

[31] Jonathan Pfeil and Soumya Ray. "Scaling a Natural Language Generation System". In: *Proceedings of the 54th Annual Meeting of the Association for*

Computational Linguistics (Volume 1: Long Papers). Ed. by Katrin Erk and Noah A. Smith. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1148–1157. DOI: 10.18653/v1/P16-1109. URL: https://aclanthology.org/P16-1109.

[32] Nils Reimers and Iryna Gurevych. "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks". In: *CoRR* abs/1908.10084 (2019). arXiv: 1908.10084. URL: http://arxiv.org/abs/1908.10084.

[33] EHUD REITER and ROBERT DALE. "Building applied natural language generation systems". In: *Natural Language Engineering* 3.1 (1997), pp. 57–87. DOI: 10.1017/S1351324997001502.

[34] Victor Sanh et al. *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. 2020. arXiv: 1910.01108 [cs.CL].

[35] Abigail See, Peter J. Liu, and Christopher D. Manning. "Get To The Point: Summarization with Pointer-Generator Networks". In: *CoRR* abs/1704.04368 (2017). arXiv: 1704.04368. URL: http://arxiv.org/abs/1704.04368.

[36] Yeon Seonwoo et al. "Additive Compositionality of Word Vectors". In: Jan. 2019, pp. 387–396. DOI: 10.18653/v1/D19-5551.

[37] Xiaoyu Shen et al. "Select and Attend: Towards Controllable Content Selection in Text Generation". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 579–590. DOI: 10.18653/v1/D19-1054. URL: https://aclanthology.org/D19-1054.

[38] Stuart M. Shieber and Yves Schabes. "Synchronous Tree-Adjoining Grammars". In: *Proceedings of the 13th Conference on Computational Linguistics - Volume 3*. COLING '90. Helsinki, Finland: Association for Computational

Linguistics, 1990, pp. 253–258. ISBN: 9529020287. DOI: 10.3115/991146.991191. URL: https://doi.org/10.3115/991146.991191.

[39] Matthew Stone and Christine Doran. "Sentence Planning as Description Using Tree Adjoining Grammar". In: *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics*. ACL '98/EACL '98. Madrid, Spain: Association for Computational Linguistics, 1997, pp. 198–205. DOI: 10.3115/976909.979643. URL: https://doi.org/10.3115/976909.979643.

[40] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL].

[41] Jan Wielemaker. "An overview of the SWI-Prolog Programming Environment". In: *Proceedings of the 13th International Workshop on Logic Programming Environments*. Ed. by Fred Mesnard and Alexander Serebenik. CW 371. Heverlee, Belgium: Katholieke Universiteit Leuven, Dec. 2003, pp. 1–16.

[42] Sam Wiseman, Stuart Shieber, and Alexander Rush. "Learning Neural Templates for Text Generation". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 3174–3187. DOI: 10.18653/v1/D18-1356. URL: https://aclanthology.org/D18-1356.