Extracting CCGs for Plan Recognition in RTS Games

Pavan Kantharaju and **Santiago Ontañón**[*]
Drexel University
3141 Chestnut St
Philadelphia, PA 19104
{pk398, so367}@drexel.edu

Christopher W. Geib
SIFT LLC.
319 1st Ave. South, Suite 400
Minneapolis, MN 55401
cgeib@sift.net

Abstract

Domain-configurable planning and plan recognition approaches such as Hierarchical Task Network and Combinatory Categorial Grammar-based (CCG) planning and plan recognition require a domain expert to handcraft a domain definition for each new domain in which we want to plan or recognize. This paper describes an approach to automatically extracting these definitions from plan traces acquired from Real-Time Strategy (RTS) game replays. Specifically, we present a greedy approach to learning CCGs from sets of plan trace, goal pairs that extends prior work on learning CCGs for plan recognition. We provide an empirical evaluation of our learning algorithm in the challenging domain of RTS games. Our results show that we are able to learn a CCG that represents larger sequences of actions, and use them for plan recognition. Our results also demonstrate how scaling the size of the plan traces affects the size of the learned representation, which paves the road for interesting future work.

Introduction

Domain-configurable planning and plan recognition have been used in various applications such as robotics and games. Each approach requires a domain definition that represents knowledge about the structures of plans and actions in an application domain. This knowledge can be represented using Planning Domain Definition Language (McDermott et al. 1998), Hierarchical Task Networks (Erol, Hendler, and Nau 1994), and Combinatory Categorial Grammars (CCG) (Steedman 2001). CCGs are a grammar formalism that has been shown to effectively capture phenomenon in real-world language (Geib and Steedman 2007) and more recently represent and recognize plans in the form of the ELEXIR framework (Geib 2009; Geib and Goldman 2011).

Past work on CCG-based planning and plan recognition used handcrafted domain definition, which can be time-consuming and error-prone to construct. Recent work in form of Lex_{Learn} successfully learned CCGs by enumerating all possible abstractions for a set of plan traces (Geib and Kantharaju 2018) given a set of templates. Lex_{Learn} was then later used to learn reactive behaviors for the Real-Time Strategy Game μRTS, and applied to the problem of adversarial planning (Kantharaju, Ontañón, and Geib 2018).

[*]Currently at Google

However, exhaustive enumeration may not scale well when learning from long plan traces.

This paper presents a greedy CCG learning algorithm called Lex_{Greedy} motivated by work on probabilistic HTN learning by Li et al. (2014) that improves the scalability of learning, allowing knowledge extraction from longer plan traces. Our learning algorithm employs a greedy approach to abstract common sequences of actions from a set of plan trace, goal pairs, and then estimates probabilities for each abstraction. We evaluate our approach for CCG-based plan recognition in the domain of RTS games using the AI RTS testbed μRTS. RTS games provide a challenge for both CCG learning and plan recognition as strategies employed by RTS game-playing agents can be long, and the learned CCG representation must be compact to handle these long plans.

This paper is structured as follows. First, we provide some related work in the area of CCG learning and hierarchical plan learning. Second, we provide a brief description of our application domain μRTS. Third, we provide background knowledge on CCGs, CCG-based plan recognition, and CCG learning. Fourth, we describe our greedy CCG learning algorithm Lex_{Greedy}. Fifth, we provide our experimental evaluation and analysis. Finally, we conclude with directions for future work.

Related Work

There are two major areas of related work: CCG learning for Natural Language Processing (NLP) and plan hierarchy learning. Zettlemoyer and Collins (2005) use supervised CCG learning to learn a mapping between sentences and their semantic representations. Thomforde and Steedman (2011) presents Chart Inference, an unsupervised learner for deriving structured CCG categories for unknown words using a partial parse chart. Bisk and Hockenmaier (2012) introduce an unsupervised learner to generate categories from part-of-speech tagged text that relies on minimal language-specific knowledge. Kwiatkowski et al. (2012) define a learning approach for learning sentence and semantic representation pairs from child utterances.

Our work differs from CCG learning for NLP in that the learned CCG represents plan knowledge about different strategies employed in RTS games instead of the syntactic and semantic structure of sentences. Specifically, this work learns plan hierarchies that abstract common sequences of actions into abstract plan structures like those in Hierarchical Task Network (HTN) learning. Nejati, Langley, and Konik (2006) learns teleoreactive logic programs (Langley and Choi 2006), a specialized class of HTNs, from expert traces. Hogg, Muñoz Avila, and Kuter (2008) present *HTN-Maker*, an HTN learning algorithm that learns HTN methods from analyzing the state of the world before and after a given sequence of actions. Hogg, Kuter, and Muñoz-Avila (2010) build off *HTN-Maker* and introduces *Q-Maker*, a learning algorithm that combines HTN-Maker with reinforcement learning. Zhuo, Muñoz-Avila, and Yang (2014) presents *HTNLearn* which builds an HTN from partially-observable plan traces.Gopalakrishnan, Muñoz-Avila, and Kuter (2016) introduce *Word2HTN*, which learns both tasks and methods using Word2Vec (Mikolov et al. 2013). Nguyen et al. (2017) present a technique for learning HTNs for Minecraft, a sandbox video game, using Word2Vec and Hierarchical Agglomerative Clustering.

The closest related work to our learning approach is that of Li et al. (2014), who successfully learns probabilistic HTNs using techniques from Probabilistic CFG learning. They abstract both looping constructs and frequent sequences of actions given a set of non-parameterized sequences of actions. Our work, on the other hand, learns a CCG representation from parameterized actions, but only abstracts common sequences of actions.

μRTS

μRTS[1] is a minimalistic Real-Time Strategy game designed to evaluate AI research in an RTS setting (Ontañón 2013). Figure 1 shows two scripted agents playing against each other in μRTS. Compared to complex commercial RTS games such as *StarCraft*, μRTS still maintains those properties of RTS games that make them complex from an AI point of view (i.e. durative and simultaneous actions, real-time combat, large branching factors, and full or partial observability). For the purposes of this paper, μRTS games are deterministic, and fully observable. μRTS has been used in previous work to evaluate RTS AI research, (Shleyfman, Komenda, and Domshlak 2014; Ontañón and Buro 2015) and has also been used in AI competitions.[2]

This work specifically focuses on learning different strategies employed by scripted game agents. In this work, we use a collection of scripted game agents to generate plan traces. One such game agent uses a strategy called *Rush*. Rush strategies consist of quickly constructing many units of a specific type, and having all those units attack the enemy. An example of a rush strategy from the commercial RTS game *Starcraft* is Zerg rush. Plan traces are constructed using *replay* data from gameplay of the scripted agents. A *replay* is a trace of a game represented as a sequence of state, action tuples containing the evolution of the game state as well as the actions performed by each player during the game. We extract sequences of actions from these replays to create plan traces. We provide a formal definition of plan traces and replays in the next section.

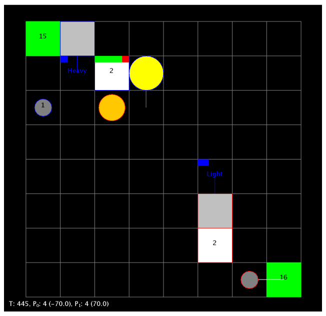

Figure 1: Screenshot of μRTS gameplay

Background

This section describes a restricted form of Combinatory Categorial Grammars (CCGs), using the definition of CCGs from Geib (2009), and defines *CCG-based plan recognition*, *plan traces* and *replays*. Each action in a domain is associated with a set of CCG categories $\mathcal{C}$, defined as follows:

Atomic categories: A set of category units A, B, C... $\in \mathcal{C}$.

Complex categories: Given a set of atomic categories $\mathcal{C}$, where Z $\in \mathcal{C}$ and $\{$W, X, Y...$\} \neq \emptyset$ and $\{$W, X, Y...$\} \in \mathcal{C}$, then Z/$\{$W, X, Y...$\} \in \mathcal{C}$ and $Z\backslash\{W, X, Y, ...\} \in \mathcal{C}$.

Intuitively, categories are functions that take other functions as arguments. Atomic categories are zero-arity functions, whereas complex categories are curried functions (Curry 1977), defined by two operators: "\" and "/". These operators each take a set of *arguments* (the categories on the right hand side of the slash, $\{$W, X, Y...$\}$), and produce the *result* (the category on the left hand side of the slash, Z). We define the *root* of some category G (atomic or complex) if it is the leftmost atomic category in G. For example, for the complex category $((\mathrm{C})\backslash\{\mathrm{A}\})\backslash\{\mathrm{B}\}$, the root would be C. The slash operators define ordering constraints for plans, indicating where other actions are to be found relative to an action. Those categories associated with the forward slash operator are after the action, and those associated with the backward slash are before it.

CCGs are lexicalized grammars. As such, we define a CCG by a *plan lexicon*, $\Lambda = \langle \Sigma, \mathcal{C}, f \rangle$, where Σ is a finite set of action types, $\mathcal{C}$ is a set of CCG categories, and f is a mapping function such that $\forall a \in \Sigma$,

$$f(a) \rightarrow \{c_i : p(c_i|a), ..., c_j : p(c_j|a)\}$$

where $c_i \ldots c_j \in \mathcal{C}$ and $\forall a \sum_{k=i}^{j} p(c_k|a) = 1$. Given a sequence of actions, these probabilities represent the likelihood of assigning a category to an action for plan recognition. For more details on these probabilities and how they are used for plan recognition, see Geib (2009).

Similar to the work by Geib and Kantharaju (2018), we assume that all complex categories are *leftward applicable* (all arguments with the backward slash operator are discharged before any forward ones), and we only consider

[1]`https://github.com/santiontanon/microrts`

[2]`https://sites.google.com/site/micrortsaicompetition/home`

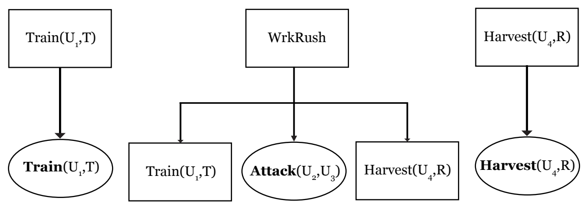

Figure 2: Hierarchical Representation of CCG

complex categories with atomic categories for arguments. We also extend the definitions of action types and atomic categories to a first-order representation by introducing *parameters* to represent domain objects and variables. A discussion of CCGs with parameterized actions and categories is presented by Geib (2016).

One major benefit of using CCGs is that the same representation can be used to plan and recognize goals. In this work, goals correspond to strategies employed by scripted agents. Below is an example CCG representation with parameterized actions and categories for executing and recognizing a *Worker Rush* strategy in μRTS. We let $\Sigma = \{$**Train**, **Attack**, **Harvest**$\}$ and $\mathcal{C} = \{\text{Train, Harvest, WrkRush}\}$:

$$f(\textbf{Train}(U_1, T)) \rightarrow \{\text{Train}(U_1, T)) : 1\}$$
$$f(\textbf{Attack}(U_2, U_3)) \rightarrow$$
$$\{((\text{WrkRush})/\{\text{Harvest}(U_4, R)\})\backslash\{\text{Train}(U_1, T)\} : 1\}$$
$$f(\textbf{Harvest}(U_4, R)) \rightarrow \{\text{Harvest}(U_4, R)) : 1\}$$

The action types **Train**(U_1, T), **Attack**(U_2, U_3), and **Harvest**(U_4, R) each have parameters representing different units U_1, U_2, U_3, U_4, unit type T, and resource R. Since each action has only a single category, $P(c_i|a) = 1$.

Figure 2 provides a hierarchical representation of the above plan lexicon. Action types are denoted by ovals and categories are denoted by rectangles. The atomic categories "Train" and "Harvest" breaks down "Train" and "Harvest" into the action types **Train** and **Harvest**. The complex category "((WrkRush)/{Harvest})\{Train}" associated with the action type **Attack**, breaks down the strategy "WrkRush" into the following sequence of atomic categories and actions: $\langle$Train, **Attack**, Harvest$\rangle$. We note that CCG categories are similar to *methods* from the Hierarchical Task Network literature (Erol, Hendler, and Nau 1994).

We now define a few terminology relevant to CCG-based plan recognition and learning. We define a *plan* π as a sequence of observed actions $a_1, a_2, \ldots a_m$ and a *partial plan* as a subsequence of these actions $a_1, a_2, \ldots a_k$, $1 \leq k \leq m$. A partial plan can be a plan. Next, we define the *CCG plan recognition problem* as $PR = (\pi', \Lambda, s_0)$, where π' is a partial plan, Λ is a CCG, and s_0 is the initial state of the world from where π' was executed. The solution to the plan recognition problem is a pair (G, π''), where G is the predicted goal of π', and π'' is the predicted sequence of actions such that $\pi' + \pi''$ results in G. We refer readers to Geib (2009) for a description on how CCGs are used for plan recognition.

A *plan trace* is a plan that is fed into a learning algorithm. Recall that actions in a plan trace are extracted from replay data. For a given two-player game, *replay data* is defined as a sequence of game state and action pairs seen over the course of a game session, $R = [(s_0, a_0^1, a_0^2), ..., (s_n, a_n^1, a_n^2)]$, where i is a game frame, s_i is the current game state and a_i^1 and a_i^2 are the actions done by player 1 and player 2. From this, we can create two plan traces: one for player 1 $(a_0^1, a_1^1, \ldots, a_n^1)$ and one for player 2 $(a_0^2, a_1^2, \ldots, a_n^2)$. We ignore states $s_0 \ldots s_n$ because we do not learn state information for our representation, and leave this to future work.

Greedy Learning of Combinatory Categorial Grammars

Prior work on CCG planning and plan recognition required hand-authored domain definitions representing the structure of plans in a domain. This process can be time consuming and error-prone for complex domains. Recent work by Geib and Kantharaju (2018) used Lex_{Learn} learn these CCGs. However, Lex_{Learn} was not able to scale to longer plan traces because it enumerated all possible abstractions for the set of plan traces, and relied on a breadth-first (BF) search based plan recognition algorithm. The complexity of BF plan recognition lies in the branching factor (average number of categories per action) and length of the plan. If the branching factor and plan length are high enough, plan recognition becomes intractable. In terms of CCGs, scaling means having a low average number of categories per action while still recognizing plans with high accuracy. The purpose of this work is to scale learning of CCGs to longer plan traces. This grammar can then be used for CCG planning and plan recognition.

We describe our greedy approach for generating action category pairs from plan traces, motivated by Li et al. (2014) that reduces the number of learned abstractions and does not rely on a plan recognizer. Our learning process is split into two processes: generating action category pairs (*GreedyGen*), and estimating action category pair conditional probabilities (*GreedyProbEst*). Algorithm 1 is the high level pseudocode for our learning algorithm, Lex_{Greedy}, where Λ refers to the learned CCG.

Lex_{Greedy} takes two inputs: an initial lexicon, Λ_{init}, and a set of training pairs, $D = \{(\pi_i, G_i) : i = 1...n\}$, where each π_i is a plan trace, $a_1, ..., a_m$, that achieves some top-level task, denoted by the atomic category G_i. For the domain of μRTS, G_i denotes strategies employed by game-playing agents. Lex_{Greedy} assumes that each each π_i results in its respective G_i, and the initial lexicon, Λ_{init}, contains a single atomic category for each action type. The atomic category's parameters are identical to those of its action type.

Greedy Category Generation

The task of *GreedyGen* is to hypothesize a set of complex categories that yield common sequences of actions from a set of plan traces $\pi_i \in D$. This is divided into two steps: hypothesizing abstractions (tasks), and generating complex categories given these tasks. We borrow the term *tasks* from the HTN literature, which is a symbolic representation of an activity in a domain (Hogg, Kuter, and Muñoz-Avila 2010). Algorithm 2 outlines *GreedyGen* in high-level pseudocode.

Algorithm 1 Greedy Learning Algorithm - Lex_{Greedy}

```
 1: procedure $Lex_{Greedy}(D, \Lambda_{init})$
 2:     $\Lambda$ = GreedyGen($D, \Lambda_{init}$)
 3:     GreedyProbEst($D, \Lambda$)
 4:     for all $a \in \Lambda_\Sigma$ do
 5:         for all $c \in \Lambda_{f(a)}$ do
 6:             if $p(c|a) < \tau$ then
 7:                 Remove $c$ from $\Lambda_{f(a)}$
 8:             end if
 9:         end for
10:         Normalize probability distribution of $\Lambda_{f(a)}$
11:         to satisfy the following constraint:
12:         $\sum_{c \in \Lambda_{f(a)}} p(c|a) = 1$
13:     end for
14:     Return $\Lambda$
15: end procedure
```

Algorithm 2 Greedy Category Generation Pseudocode

```
 1: procedure GREEDYGEN($D, \Lambda_{init}$)
 2:     Let $\Pi = \{\pi_i | i = 1 \dots n, \pi_i \in D\}$
 3:     Let $\Lambda = \Lambda_{init}$
 4:     Let $PT = \emptyset$
 5:     repeat
 6:         $\langle t, (\psi_0 \dots \psi_u) \rangle$ = CreateCommon($\Pi$)
 7:         $\langle \Pi', PT' \rangle$ = Update($\Pi, \langle t, \psi_0 \dots \psi_u \rangle$)
 8:         if $PT' \neq \emptyset$ then
 9:             $PT = PT \cup PT'$
10:             $\Pi = \Pi'$
11:         end if
12:     until $t = \emptyset \vee PT' = \emptyset$
13:     $\Lambda \leftarrow \Lambda_{init} \cup$ CreateCommonCategories($PT$)
14:     $\Lambda \leftarrow \Lambda \cup$ CreateGoalCategories($\Pi$)
15:     Return $\Lambda$
16: end procedure
```

This process takes, as input, a set of plan traces and their corresponding goals D, and initial lexicon Λ_{init}, and returns a lexicon containing the set of hypothesized categories Λ.

We define Π as a set of abstract traces which contain both actions and tasks generated during the learning process, and PT as the set of learned common abstractions. Initially, Π contains plan traces from D. *GreedyGen* iteratively hypothesizes new tasks that abstract some common sequence of actions and tasks $\psi_0 \dots \psi_u$ found in Π, and replaces them with the new task. Since $\psi_k \in \{\psi_0 \dots \psi_u\}$ can be a task, this makes the learned abstractions hierarchical. We ignore parameters of actions and tasks when searching for the most common sequence. If we considered their parameters, each action and task in the sequence and their parameters have to match exactly when searching for the most common sequence, which can reduce abstraction.

We look at each function in Algorithm 2 below. The function *CreateCommon* creates a new task t for the most common sequence of actions and tasks $\psi_0 \dots \psi_u$ in the set of abstract traces Π above a given tunable *abstraction threshold* γ. We define this threshold γ as the percentage of instances in D. If there are ties for the most common sequence, then the first encountered sequence is considered. We note that this sequence must contain at least one action. If no sequence contains at least one action, then *CreateCommon* returns $\emptyset$.

Next, *Update* updates each trace in Π with this newly-created task t, and parameterizes $\langle t, (\psi_0 \dots \psi_u) \rangle$. For each $\omega_i \in \Pi$, the function replaces each occurrence of $\psi_0 \dots \psi_u$ with t and creates a pair $\langle t', \psi'_0 \dots \psi'_u \rangle$, where t' is the parameterized task and $(\psi'_0 \dots \psi'_u)$ are parameterized actions and tasks. The parameters for task t' is defined by the set union of the parameters of $\psi'_0 \dots \psi'_u$. The parameters for any action is defined by their original parameters from π_i. *Update* replaces the common sequences in ω_i if and only if the result still contains at least one action. *Update* returns the revised traces Π' and the set of parameterized task, sequence pairs PT', and updates Π and PT if PT' is non-empty. If *Update* can not replace any actions in each abstract trace ($PT' = \emptyset$), the loop terminates.

Once the loop terminates, *GreedyGen* creates complex categories for the common sequences in PT (*CreateCommonCategories*) and the abstract traces in Π (*CreateGoalCategories*). Both functions create complex categories using the following template:

$$\rho_k \rightarrow \Upsilon(x)/\{\Upsilon(\rho_u)\}/\dots/\{\Upsilon(\rho_{k+1})\} \\ \backslash\{\Upsilon(\rho_0)\}\backslash\dots\backslash\{\Upsilon(\rho_{k-1})\}$$

where ρ_k (where $0 \leq k \leq u$) is the action whose action type will be assigned the complex category, x is a task, and Υ is a function that either creates an atomic category for a task or retrieves the atomic category in Λ_{init} for an action. In our experiments, ρ_k is the action closest to the middle of $\rho_0 \dots \rho_u$. If the action type of ρ_k is already assigned the complex category, the category is then ignored.

CreateCommonCategories creates complex categories for each pair $\langle t', \psi'_0 \dots \psi'_u \rangle \in PT$. We let $\rho_k = \psi'_k$, $x = t'$ and $\rho_i = \psi'_i$, where $\psi'_i, \psi'_k \in \psi'_0 \dots \psi'_u$ and ψ'_k is an action. *CreateGoalCategories* creates complex categories that yields each $\omega_i \in \Pi$. Here, we let $\Upsilon(x) = G_i$, $\rho_i = \psi_i$, and $\rho_k = \psi_k$, where $\omega_i = \psi_0 \dots \psi_u$ and ψ_k is an action.

We illustrate the greedy generation process through an example. Suppose we have an initial lexicon Λ_{init} as follows:

$$f(\textbf{Train}(U_1, T)) \rightarrow \{\text{Train}(U_1, T) : 1\}$$
$$f(\textbf{Attack}(U_2, U_3)) \rightarrow \{\text{Attack}(U_2, U_3) : 1\}$$
$$f(\textbf{Harvest}(U_4, R)) \rightarrow \{\text{Harvest}(U_4, R) : 1\}$$
$$f(\textbf{Return}(U_5, B_1)) \rightarrow \{\text{Return}(U_5, B) : 1\}$$

where U_1, U_2, U_3, U_4, U_5 are units, R is a resource, B is a base, T represents a unit type, $\Sigma = \{$**Train**, **Attack**, **Harvest**, **Return**$\}$, and $\mathcal{C} = \{$Train, Attack, Harvest, Return$\}$. Next, suppose we have the following two plan traces:

$$\pi_1 = \langle \textbf{Harvest}(U_1, R_1), \textbf{Return}(U_1, B_1), \textbf{Train}(U_3, H) \rangle$$
$$\pi_2 = \langle \textbf{Harvest}(U_2, R_2), \textbf{Return}(U_2, B_2), \textbf{Attack}(U_4, U_1) \rangle$$

and π_1 corresponds to strategy *HeavyRush* and π_2 corresponds to strategy *WorkerRush*. First, we create $\Pi = \{\omega_1 = \pi_1, \omega_2 = \pi_2\}$. Next, *CreateCommon* creates a task t_x for the most common sequence $\langle$**Harvest**, **Return**$\rangle$. Next, *Update* replaces this common sequence in Π,

$$\omega_1 = \langle t_x, \textbf{Train}(U_3, H)\rangle$$
$$\omega_2 = \langle t_x, \textbf{Attack}(U_4, U_1)\rangle$$

creates a set of parameterized task, sequence pairs and adds them to PT:

$$\langle \mathrm{t}'_x(U_1, R_1, B_1), (\textbf{Harvest}(U_1, R_1), \textbf{Return}(U_1, B_1))\rangle$$
$$\langle \mathrm{t}'_x(U_2, R_2, B_2), (\textbf{Harvest}(U_2, R_2), \textbf{Return}(U_2, B_2))\rangle$$

Since there are no more common sequences in Π, we exit the loop. Next, *CreateCommonCategories* creates a complex category for each pair in PT using the previously defined template:

$$\textbf{Harvest}(U_1, R_1) \rightarrow \Upsilon(\mathrm{t}'_x(U_1, R_1, B_1))/\{\mathrm{Return}(U_1, B_1)\}$$
$$\textbf{Harvest}(U_1, R_1) \rightarrow \Upsilon(\mathrm{t}'_x(U_2, R_2, B_2))/\{\mathrm{Return}(U_2, B_2)\}$$

where each action in the sequence is replaced with its initial category in Λ_{init} and $\rho_k = \textbf{Harvest}(U_1, R_1)$. Finally, *CreateGoalCategories* creates complex categories to yield both ω_1 and ω_2,

$$\textbf{Return}(U_1, B_1) \rightarrow \mathrm{HeavyRush}\backslash\{\Upsilon(\mathrm{t}'_x(U_1, R_1, B_1))\}$$
$$\textbf{Return}(U_1, B_1) \rightarrow \mathrm{WorkerRush}\backslash\{\Upsilon(\mathrm{t}'_x(U_2, R_2, B_2))\}$$

where $\rho_k = \textbf{Return}(U_1, B_1)$. These categories are added to the lexicon and the category generation process is complete.

Greedy Probability Estimation

Probability estimation is the process of estimating action type, category conditional probabilities $P(c|a)$, which are used during CCG plan recognition. Recall that the CCG plan recognition problem is defined as $PR = (\pi', \Lambda, s_0)$, where π' is a partial plan, Λ is a CCG, and s_0 is the initial state of the world from where π' was executed. CCG-based plan recognition assigns a single category from Λ to each action in π' and parses these categories using CCG combinators (Geib 2009). Geib and Kantharaju (2018) used stochastic gradient ascent to estimate the conditional probability as a normalized weighted frequency of assigning category c to action a during plan recognition, where a breadth-first plan recognizer was used to acquire these frequencies. We propose a greedy approach (*GreedyProbEst*) to estimate probabilities without a plan recognizer.

The *GreedyGen* process creates complex categories and assigns these categories to a subset of actions and their types in each plan trace $\pi_i \in \Pi$. *GreedyProbEst* assigns the remaining actions in each π_i that are not assigned a complex category during the *GreedyGen* process to atomic categories from Λ_{init}. This essentially simulates part of the plan recognition process (assignment of categories to actions in a plan) without actually executing a plan recognition algorithm. From this, we can infer the number of times a complex category c is assigned to some action type, denoted by $H_{c,\alpha}$ where α is the action type of action $a \in \pi_i$.

GreedyProbEst computes $H_{c,\alpha}$ as follows. Let Λ indicate the lexicon created by *GreedyGen*, and $\Lambda_{f(\alpha)}$ refer to the set of categories assigned to α. For each action type $\alpha \in \Lambda_\Sigma$, *GreedyProbEst* gets the set of categories $\mathcal{C}$ assigned to actions in Π with α. Next, for $c \in \Lambda_{f(\alpha)}$, $H_{c,\alpha}$ is computed,

$$H_{c,\alpha} = \begin{cases} \mathcal{F}(c, \mathcal{C}) & c \in \mathcal{C} \\ 1 & \text{otherwise} \end{cases}$$

where $\mathcal{F}(c, \mathcal{C})$ is the frequency of c in $\mathcal{C}$. *GreedyProbEst* then computes the probability $p(c|\alpha)$ for $\alpha \in \Lambda_\Sigma$ as:

$$p(c|\alpha) = \frac{H_{c,\alpha}}{\sum_{c' \in \Lambda_{f(\alpha)}} H_{c',\alpha}}$$

Lex_{Greedy} successfully learns a CCG upon completing the probability estimation process. However, the generated CCG may not be tractable for plan recognition. Thus, Lex_{Greedy} prunes categories from Λ with an estimated probability lower than a given *pruning threshold* τ (lines 4-12 in Algorithm 1). Intuitively, this represents the maximum number of allowable categories per action. Finally, Lex_{Greedy} re-normalizes the conditional probabilities (lines 10-12 in Algorithm 1) and the learning process is complete.

Experiments

The purpose of our experiments is to evaluate both the scalability of CCG learning and performance of the learned CCG for CCG-based plan recognition. To this end, we compare against the exhaustive CCG learning technique Lex_{Learn} by Geib and Kantharaju (2018). All experiments were run on a machine with 3.40GHz Intel i7-6700 CPU and 32 GB RAM. We start by describing the learning dataset used in our experiments. Next, we define the tunable parameters for both Lex_{Learn} and Lex_{Greedy}. Finally, for each experiment, we describe metrics and experiment setup, and analyze results.

We generated a learning dataset of plan traces paired with their corresponding goals using a replay dataset generated from gameplay sessions on the μRTS testbed. Recall that replay data is a sequence of game state/action pairs seen over the course of a game session. These actions are *player actions*, i.e., the set of all the *unit actions* issued to each unit controlled by a player at the current game state. For example, if three actions are done by a player's units, {**Attack**, **Harvest**, **Produce**}, the player action would be **Attack_Harvest_Produce**.

Our replay dataset was created by running a 5-iteration round-robin tournament using the following built-in scripted agents: *POLightRush*, *POHeavyRush*, *PORangedRush*, *POWorkerRush*, *EconomyMilitaryRush*, *EconomyRush*, *HeavyDefense*, *LightDefense*, *RangedDefense*, *WorkerDefense*, *WorkerRushPlusPlus*. Each agent played against each other as both player 1 and player 2 on the open maps of the CIG 2018 μRTS tournament.[3] We chose these agents over other agents such as *NaiveMCTS* because they execute a defined strategy which we could use as the goal for learning and plan recognition.

Using the replay dataset, we constructed our learning dataset as follows. For each replay in the replay dataset, we generated two plan trace/goal pairs: one for each agent in the

[3]`https://sites.google.com/site/micrortsaicompetition/rules`

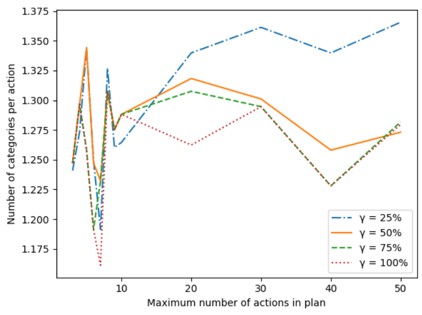

Figure 3: Scalability results for Lex_{Greedy} (pruning)

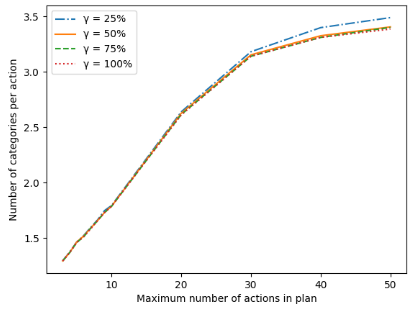

Figure 4: Scalability results for Lex_{Greedy} (No pruning)

replay. Each pair was constructed by parsing player actions done by an agent, and using the agent itself as the goal (i.e., the goal of plan recognition will be to identify which of the agents does a plan trace come from).

Next, we define the tunable parameters for each learning approach. Both Lex_{Greedy} and Lex_{Learn} have one common parameter: *pruning threshold* τ, which effectively limits the maximum number of categories per action. This parameter was set to allow for tractable plan recognition using the learned CCGs. After initial experimentation, we chose $\tau = 0.1$ for Lex_{Learn}, and $\tau = 0.01$ for Lex_{Greedy}. All other parameters for Lex_{Learn} were set according to the original paper (Geib and Kantharaju 2018). Lex_{Greedy}'s *abstraction threshold* γ was set based on the experiment. For Experiment 1, γ ranged from 25% to 100% in increments of 25 and for Experiment 2, γ was 75% and 100%.

Experiment 1: The first experiment focuses on analyzing the scalability of Lex_{Greedy} and Lex_{Learn}. We measure scalability by computing the average number of categories per action, defined as the total number of categories over the total number of action types in the CCG. We use this metric because it directly impacts the tractability of breadth-first plan recognition, and the compactness of the learned representations. An optimal CCG would have an average of 1.0.

For this experiment, we averaged our results over 5 runs. For each run, we randomly shuffled all plan trace/goal pairs in the learning dataset, and learned CCGs using both Lex_{Learn} and Lex_{Greedy}. The learning dataset was shuffled because Lex_{Learn} is an incremental algorithm, and its learned CCG depends on the ordering of the plan traces. Figure 3 shows the number of categories per action plotted against the maximum number of allowed actions per plan trace (ranging from 3 to 50 actions). Overall, we see that all values of γ (25-100%) almost followed the same pattern for plans with less than 10 actions. After 10 actions, $\gamma = 25\%$ diverged from the rest of the values and continued a positive trend while the others maintained a negative trend. This implies that longer plans benefit from γ than smaller plans.

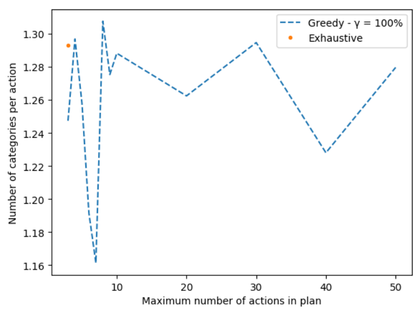

Figure 5: Comparison of Lex_{Learn} and Lex_{Greedy} against different plan trace lengths

However, there was no overall trend in the results. This is a result of pruning the number of categories per action after learning the CCG. To prove this, we ran Lex_{Greedy} where we set $\tau = 0$, preventing Lex_{Greedy} from pruning any categories. Figure 4 shows the results of this experiment. We see that all values of γ have similar average number of categories per action. We also notice a positive linear correlation between plan lengths and average number of categories per action, which is the expected result.

Figure 5 provides a comparison between the scalability of Lex_{Learn} (Exhaustive) and Lex_{Greedy} (Greedy), plotting plan length against number of categories per action. We only provided results for Lex_{Greedy} with a $\gamma = 100\%$ as results with other thresholds were relatively similar to it. We note that Lex_{Learn} was only able to execute for plans of length 3 because it ran out of memory for longer plans. Recall that Lex_{Learn}'s probability estimation technique uses breadth-first plan recognition. Lex_{Learn} will run out of memory if the average number of categories per action is high enough.

Lex_{Greedy} was able to successfully learn a CCG representation for plans with 50 actions with a low average number of categories per action, successfully demonstrating the scalability of our learning technique. We also notice that the average number of categories per action was close to 1, indicating that both Lex_{Learn} and Lex_{Greedy} learned a near optimal CCG. Despite this, there was a large variance in the number of categories assigned to each action where some actions had 1 category and others had significantly more. Therefore, as we will see in Experiment 2, this large variance will result in our CCG plan recognizer running out of memory for plans with more than 10 actions.

Experiment 2: The second experiment focuses on the per-

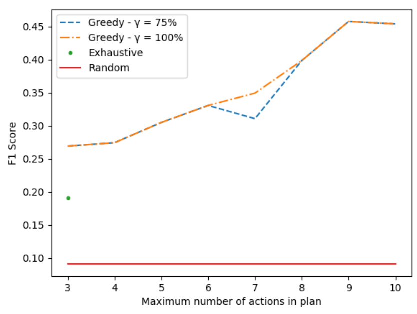

Figure 6: Effect of plan length on plan recognition - F1 Score

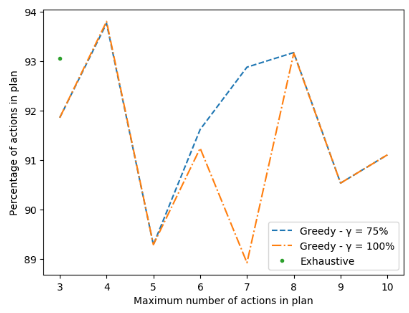

Figure 7: Effect of plan length on plan recognition - MTTR

formance of the learned CCGs for plan recognition. We use the ELEXIR framework developed by Geib (2009) for breadth-first plan recognition. Recall that the CCG plan recognition problem is defined as $PR = (\pi', \Lambda, s_0)$ and its solution as (G, π''). We set $s_0 = \emptyset$ as ELEXIR can recognize plans with an empty s_0, and focus on predicting G.

We use two metrics from the original Lex_{Learn} paper (Geib and Kantharaju 2018). The first metric, following Zhuo, Muñoz-Avila, and Yang (2014) is the F1 Score:

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

where precision and recall (Zettlemoyer and Collins 2005) adapted for the problem of plan recognition are:

$$Precision = \frac{\#\ correctly\ recognized}{\#\ of\ parsed\ plan\ traces}$$

$$Recall = \frac{\#\ correctly\ recognized}{\#\ of\ plan\ traces}$$

The second metric is Mean-Time-To-Recognition (MTTR) or the average percentage of actions in a plan required to recognize the goal. The formal definition of this can be found in Geib and Kantharaju (2018). A high-performing CCG would have a high F1 score and a low MTTR.

All metrics are averaged over 5 runs. For each round, we randomly shuffled all plan trace/goal pairs in the learning dataset, split the dataset into 80% training and 20% testing, trained Lex_{Learn} and Lex_{Greedy}, and recognized the plan traces in the testing dataset using the learned CCG.

Figure 6 provides average F1 scores for plan lengths ranging from 3 to 10 for Lex_{Greedy} (Greedy), Lex_{Learn} (Exhaustive), and random recognition (Random) (higher is better). Random recognition recognizes a plan by choosing a goal at random from the set of possible goals (recall goals are the agents). Similar to Experiment 1, Lex_{Learn} ran out of memory for plans with more than 3 actions, and both $\gamma = 0.75$ and $\gamma = 1.0$ didn't have any significant difference in performance. However, both Lex_{Learn} and Lex_{Greedy} were able to outperform random recognition with an F1 score of 0.0909. We note that pruning can have a significant impact on the F1 score as categories in the learned CCG that are needed for plan recognition can be removed. This may have resulted in Lex_{Greedy} outperforming Lex_{Learn} as the latter approach had a significantly higher pruning threshold. In terms of RTS games, pruning results in the reduction of strategy execution knowledge. While pruning helps scale the learned CCGs, it can prevent a game-playing agent from recognizing less-frequently used strategies.

Figure 7 shows average MTTR plotted against plan lengths from 3 to 10 for both Lex_{Greedy} and Lex_{Learn} (lower is better). We see that Lex_{Greedy} (MTTR of approximately 92%) was able to outperform Lex_{Learn} (MTTR of approximately 93%) for plans with 3 actions. Similar to the F1 scores, $\gamma = 75\%$ and $\gamma = 100\%$ do not have significantly different average MTTR over all plan lengths. The main takeaway from this is that both Lex_{Learn} and Lex_{Greedy} were both able to recognize plans prior to completion. This is important for agents playing RTS games because early recognition results in a higher chance of winning a game.

We restricted the plan lengths to a maximum of 10 actions because ELEXIR ran out of memory when recognizing plans greater than length 10. However, from Experiment 1, we see that Lex_{Greedy} was able to learn plans for length 50. This indicates that, while we are able to learn CCGs for longer plans, we can not yet use the learned representations for plan recognition. We believe that if we gave ELEXIR more memory or used a non-breadth-first plan recognizer, we may have recognized plans with more than 10 actions.

Conclusion and Future Work

This paper presented a greedy CCG learning algorithm called Lex_{Greedy} motivated by work on probabilistic HTN learning by Li et al. (2014) that improves the scalability of learning. We evaluated our learned representations on CCG-based plan recognition in the domain of RTS games using the AI RTS testbed μRTS, and evaluated the scalability of learning. Our results demonstrate Lex_{Greedy} can learn compact CCGs for long plans in μRTS, allowing us to automated the authoring of CCG domain definitions. However, the learned CCGs are still too large for plan recognition.

There are a few avenues for future work that build directly from this work. First, we would like to analyze the scaling from Lex_{Greedy} for planning in and playing μRTS. Second, we want to develop a plan recognizer that can recognize plans using the representations learned by Lex_{Greedy}. Third and finally, we would like to improve Lex_{Greedy} by making it an incremental learning algorithm.

References

Bisk, Y., and Hockenmaier, J. 2012. Simple robust grammar induction with combinatory categorial grammars. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, AAAI'12, 1643–1649. Palo Alto, California, USA: AAAI Press.

Curry, H. 1977. *Foundations of Mathematical Logic*. Dover Publications Inc.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. UMCP: A sound and complete procedure for hierarchical task network planning. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS 94)*, 249–254.

Geib, C. W., and Goldman, R. P. 2011. Recognizing plans with loops represented in a lexicalized grammar. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, AAAI'11, 958–963. Palo Alto, California, USA: AAAI Press.

Geib, C., and Kantharaju, P. 2018. earning Combinatory Categorial Grammars for Plan Recognition. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*.

Geib, C. W., and Steedman, M. 2007. On Natural Language Processing and Plan Recognition. In *Proceedings of the International Joint Conferences on Artificial Intelligence*.

Geib, C. W. 2009. Delaying commitment in plan recognition using combinatory categorial grammars. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, 1702–1707. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Geib, C. W. 2016. Lexicalized reasoning about actions. *Advances in Cognitive Systems* Volume 4:187–206.

Gopalakrishnan, S.; Muñoz-Avila, H.; and Kuter, U. 2016. Word2HTN:Learning Task Hierarchies Using Statistical Semantics and Goal Reasoning. In *Proceedings of the IJCAI 2016 Workshop on Goal Reasoning*.

Hogg, C.; Kuter, U.; and Muñoz-Avila, H. 2010. Learning Methods to Generate Good Plans: Integrating HTN Learning and Reinforcement Learning. In *Aaai*.

Hogg, C.; Muñoz Avila, H.; and Kuter, U. 2008. Htn-maker: Learning htns with minimal additional knowledge engineering required. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, 950–956. AAAI Press.

Kantharaju, P.; Ontañón, S.; and Geib, C. 2018. μCCG, a CCG-based Game-Playing Agent for microRTS. In *IEEE Conference on Computational Intelligence in Games (CIG 2018)*.

Kwiatkowski, T.; Goldwater, S.; Zettlemoyer, L.; and Steedman, M. 2012. A probabilistic model of syntactic and semantic acquisition from child-directed utterances and their meanings. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics*, EACL '12, 234–244. Stroudsburg, PA, USA: Association for Computational Linguistics.

Langley, P., and Choi, D. 2006. Learning Recursive Control Programs From Problem Solving. *Journal of Machine Learning Research* 7(Mar):493–518.

Li, N.; Cushing, W.; Kambhampati, S.; and Yoon, S. 2014. Learning probabilistic hierarchical task networks as probabilistic context-free grammars to capture user preferences. *ACM Trans. Intell. Syst. Technol.* 5(2):29:1–29:32.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Mikolov, T.; Chen, K.; Corrado, G.; and Dean, J. 2013. Efficient Estimation of Word Representations in Vector Space. *arXiv preprint arXiv:1301.3781*.

Nejati, N.; Langley, P.; and Konik, T. 2006. Learning hierarchical task networks by observation. In *Proceedings of the 23rd international conference on Machine learning*, 665–672. ACM.

Nguyen, C.; Reifsnyder, N.; Gopalakrishnan, S.; and Muñoz-Avila, H. 2017. Automated Learning of Hierarchical Task Networks for Controlling Minecraft Agents. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, 226–231.

Ontañón, S., and Buro, M. 2015. Adversarial hierarchical-task network planning for complex real-time games. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

Ontañón, S. 2013. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Proceedings of the AAAI Artificial Intelligence and Interactive Digital Entertainment conference (AIIDE 2013)*.

Shleyfman, A.; Komenda, A.; and Domshlak, C. 2014. On combinatorial actions and cmabs with linear side information. In *Proceedings of the Twenty-first European Conference on Artificial Intelligence*, 825–830. IOS Press.

Steedman, M. 2001. *The Syntactic Process*. Cambridge, MA, USA: MIT Press.

Thomforde, E., and Steedman, M. 2011. Semi-supervised ccg lexicon extension. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, 1246–1256. Stroudsburg, PA, USA: Association for Computational Linguistics.

Zettlemoyer, L. S., and Collins, M. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI '05, Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence, Edinburgh, Scotland, July 26-29, 2005*, UAI'05, 658–666. AUAI Press.

Zhuo, H. H.; Muñoz-Avila, H.; and Yang, Q. 2014. Learning hierarchical task network domains from partially observed plan traces. *Artificial Intelligence* 212:134 – 157.