Combinatory Categorial Grammar Learning for Plan Recognition in Domains with Type Trees

Pavan Kantharaju,[1] **Santiago Ontañón,**[2*] **Christopher W. Geib,**[3] **Mark Roberts**[4]
[1,2] Drexel University, Philadelphia, PA, 19104
[3] SIFT LLC, Minneapolis, MN, 55401
[4] US Naval Research Laboratory, Washington DC, 20375
[1,2]{pk398, so367}@drexel.edu, [3]cgeib@sift.net, [4]mark.roberts@nrl.navy.mil

Abstract

Combinatory Categorial Grammars (CCGs) have been used for plan recognition and planning in various domains such as robotics and video games. Prior work on CCG planning and plan recognition used hand-authored CCGs, which requires domain knowledge, and can be both time-consuming and error prone to construct. This paper focuses on automatically learning CCGs and presents two key contributions to the literature of CCG learning. First, we extend existing CCG learning algorithms for domains with predefined *type trees* via searching over type trees to find type generalizations. Second, we present the first comparison of learned CCGs with CCGs hand-authored by humans, showing tradeoffs of these algorithms. We apply this extended learning algorithm to Minecraft and Monroe, and demonstrate its performance against a hand-authored model for plan recognition tasks.

1 Introduction

Hierarchical approaches to planning are commonly used in applications (Ghallab, Nau, and Traverso 2016) and have gained recent traction in the planning community (e.g., (Alford, Bercher, and Aha 2015; Alford et al. 2016)). The spectrum of representations for hierarchical planning include Hierarchical Task Networks (Erol, Hendler, and Nau 1994a), Combinatory Categorial Grammars (CCGs) (Geib and Goldman 2001), Teleoreactive Logic Programs (Langley and Choi 2006), and Hierarchical Goal Networks (Shivashankar 2015). CCGs are particularly attractive because they are based on a natural language grammar (thus allowing for a more natural explanation (Fox, Long, and Magazzeni 2017)), they are more expressive than HTNs (Geib and Steedman 2007), and they provide a unified representation for plan synthesis and plan recognition (Geib 2016).

Encoding CCGs can be difficult, which limits their adoption. While many approaches for learning task network structures exist (e.g., (Hogg, Muñoz-Avila, and Kuter 2016; Gopalakrishnan, Muñoz-Avila, and Kuter 2016)), only two works (i.e., (Geib and Kantharaju 2018; Kantharaju, Ontañón, and Geib 2019a) have examined how to automatically learn CCGs. One aspect that is particularly challenging for CCG learning is generalizing models across different types in a type taxonomy. Figure 1 shows an example of a type tree from the computer game Minecraft. Instead of learning a general method for growing crops of different types (e.g. all seeds have the same basic process for growing a crop), current approaches will learn distinct methods for each type, which impacts performance for plan recognition.

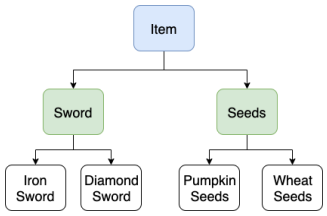

Figure 1: Example of Minecraft Type Tree

This paper generalizes CCG learning to work with type taxonomies. This is a non-trivial extension because it requires adding mechanisms to generalize plan traces. The contributions of the paper include:

- Extending a CCG learning algorithm, called Lex_{Greedy} (Kantharaju, Ontañón, and Geib 2019a) to generalize over type trees;
- Developing hand-authored CCG models for two benchmark domains, Minecraft and Monroe ; and
- Performing a comparison of the hand authored and learned models to draw insights about the tradeoffs of automatically learning CCG models.

Our results indicate Lex_{Greedy} outperforms several baselines in both Monroe and Minecraft, and provides a performance comparison to hand-authored models. Our approach offers benefits beyond CCGs in that the extension can potentially be applicable to HTN learning.

2 Background

We leverage a restricted form of CCGs, defined by Geib (2009), to perform plan recognition. We will introduce CCGs and their recognition with an example from Minecraft, a first-person sandbox video game with

*Currently at Google

a research interface from Microsoft Research (Johnson et al. 2016). The open world nature of Minecraft presents a challenge for AI planning and plan recognition because the set of possible plans to generate or recognize is open ended. Many of the high-level objectives that players typically pursue are usually made up of smaller objectives such as gathering resources or crafting items, providing a natural hierarchical structure for tasks.

Minecraft also has a natural type taxonomy (Guss et al. 2019). Figure 1 provides an example type tree for items, where *Iron Sword*, *Diamond Sword*, *Pumpkin Seeds*, and *Wheat Seeds* are subtypes of *Sword* and *Seeds*. These, in turn, are a subtype of *Item*. The same actions in Minecraft can be applied to objects with different types. For example, we can use objects with types *iron sword*, *diamond sword*, *pumpkin seeds* and *wheat seeds* in Minecraft. Without type trees, we need separate actions and hierarchies for each type. With type trees, we can specify a single action that uses an *item* and reuse it with specifications of *item* in the tree, motivating the need for type trees in Minecraft.

CCGs are a hierarchical grammar that is constructed to synthesize or recognize hierarchical plans. Learning a CCG is done with respect to a sequence of observed actions $\vec{\sigma}$. Consider a $\vec{\sigma}$ for a Minecraft agent $Player_1$ to obtain chicken meat, shown in Figure 3 (top). To obtain the meat, $Player_1$ moves toward a chicken, attacks the chicken, and gathers the resulting drop. Such actions are instances of *action types*, which are templates for constructing executable actions. The sequence $\vec{\sigma}$ contain three action types: **Move**(U_1), **Attack**(U_1, U_2), and **Gather**(U_1, G). An action can be *instantiated* by assigning objects in the domain to the variables of the action type (e.g. to get **Move**$(Player_1)$, we use **Move**(U_1) and assign the value $Player_1$ to variable U_1).

Each action type is associated with a finite set of CCG categories $\mathcal{C}$ that capture functions from states of the world to other states of the world. CCG categories come in two types: *Atomic* and *Complex*. Atomic categories are a set of indivisible labels A, B, C.. $\in \mathcal{C}$ that function as the non-terminals within the grammar; these labels map to action types as the terminals. Drawing parallels to hierarchical planning, they can be seen as tasks that will be decomposed into instances of action types. Complex categories, on the other hand, can be seen as curried functions (Curry 1977), that capture structure, abstraction, and ordering constraints within the grammar. Complex categories are built using two operators "/" and "\" and having the form Z/X or Z\X, where Z is a single (atomic or complex) category, and X is a set (possibly singleton) of atomic categories. Each operator defines a function that takes a set of *arguments* (the categories on the right hand side of the slash, X), and produces a *result* (the category on the left hand side of the slash, Z). The slash operators define ordering constraints for plans, indicating where other actions and tasks are 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. We will see an example of this shortly. For convenience, we also define the *root* of category G (atomic or complex) as the leftmost atomic category in G (e.g C would be the root of $((\text{C})\backslash\{\text{A}\})\backslash\{\text{B}\}$).

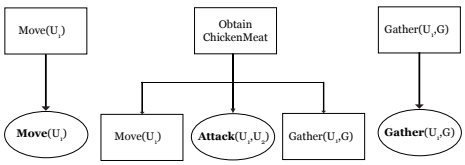

Figure 2: An example CCG (described formally in Figure 3) in graphical form. Ovals indicate action types and boxes indicate atomic categories

A problem domain is defined by a *plan lexicon* (referred to as CCG lexicon or CCG), $\Lambda = \langle \Sigma, \mathcal{C}, f \rangle$, where Σ is a finite set of action types, $\mathcal{C}$ is a finite set of atomic and complex categories (as defined above), 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 $\forall a \sum_{k=i}^{j} p(c_k|a) = 1$. This definition implies that action type can have multiple but finite number of categories associated with it in Λ. We refer the interested reader to Geib (2009) for details on these probabilities and their use in plan recognition.

In CCGs, *combinators* parse sequences of tokens. The two most common types of combinators are *application* and *composition*. The application combinator applies an atomic category to a complex category of arity one while composition combines two complex categories:

Rightward Application: $X/\{Y\} \quad Y \rightarrow X$

Leftward Application: $Y \quad X\backslash\{Y\} \rightarrow X$

Rightward Composition: $X/\{Y\} \quad Y/\{Z\} \rightarrow X/\{Z\}$

Intuitively, we can think of application and composition combinators as functional application and composition.

Example: Figure 3 (middle) shows one possible CCG relevant to $\vec{\sigma}$. Here, Σ = {**Move**, **Attack**, **Gather**} and $\mathcal{C}$ = {Move, ObtainChickenMeat, Attack, $((\text{ObtainChickenMeat})/\{\text{Gather}\})\backslash\{\text{Move}\}$} (stated without parameters for brevity). The action types **Move**(U_1), **Attack**(U_1, U_2), and **Gather**(U_1, G) each have typed variables representing entities U_1 and U_2, and dropped item G. Since each action type has a single category, $p(c|a) = 1$.

Figure 2 provides a graphical representation of the CCG in Figure 3 (types are removed for simplicity). Action types are indicated by ovals and categories are indicated by rectangles. The atomic categories "Move" and "Gather" can be viewed as tasks that can be completed by action types **Move** and **Gather**. The complex category "$((\text{ObtainChickenMeat})/\{\text{Gather}\})\backslash\{\text{Move}\}$" associated with the action type **Attack**, states that task "ObtainChickenMeat" can be completed by executing the following sequence of actions and tasks: $\langle$Move, **Attack**, Gather$\rangle$. Action, category pairs are similar to methods from hierarchical planning (Erol, Hendler, and Nau 1994b).

The CCG Plan Recognition Problem

We define the *CCG plan recognition problem* as $PR = (\vec{\sigma}, \Lambda, s_0)$, where $\vec{\sigma}$ is a sequence of observed actions, Λ is

$$\vec{\sigma} = [\,\textbf{Move}(Player_1), \textbf{Attack}(Player_1, Chicken_1), \textbf{Gather}(Player_1, ChickenMeat)\,]$$

$f(\textbf{Move}(U_1 - \textit{Entity})) \rightarrow \{\text{Move}(U_1 - \textit{Entity}) : 1\}$

$f(\textbf{Attack}(U_1 - \textit{Entity}, U_2 - \textit{Entity})) \rightarrow$

$\quad \{((\text{ObtainChickenMeat})/\{\text{Gather}(U_1 - \textit{Entity}, G - \textit{Drop})\})\backslash\{\text{Move}(U_1 - \textit{Entity})\} : 1\}$

$f(\textbf{Gather}(U_1 - \textit{Entity}, G - \textit{Drop})) \rightarrow \{\text{Gather}(U_1 - \textit{Entity}, G - \textit{Drop}) : 1\}$

1) $\quad \textbf{Move}(Player_1) \qquad \textbf{Attack}(Player_1, Chicken_1) \qquad \textbf{Gather}(Player_1, ChickenMeat)$

2) $\quad \text{Move}(U_1 = Player_1) \quad ((\text{ObtainChickenMeat})/\{\text{Gather}(U_1, G)\})\backslash\{\text{Move}(U_1)\} \quad \text{Gather}(U_1 = Player_1, G = ChickenMeat)$

3) $\quad (\text{ObtainChickenMeat})/\{\text{Gather}(U_1, G)\}$

4) $\quad \text{ObtainChickenMeat}$

Figure 3: An example plan $\vec{\sigma}$ (top), CCG (middle), and parse of three observed actions for obtaining chicken meat (bottom)

a CCG lexicon, and s_0 is the initial state of the world from which $\vec{\sigma}$ was executed. This work assumes each $\vec{\sigma}$ corresponds to a single task. The solution to the CCG plan recognition problem is a pair (E, T), where T is a conditional probability distribution over the set of top-level tasks that are being pursued, and E is a list of hypotheses that each define a unique possible plan which are instances of $\vec{\sigma}$. T is computed from E using weighted model counting.

Example: Figure 3 (bottom) shows the recognition of the observed sequence of actions $\vec{\sigma}$ (Figure 3 (middle)), which is a plan for obtaining chicken meat. First, a category is assigned to each action and its parameters (U_1, G) are bound based on the action (i.e. $U_1 = Player_1, G = ChickenMeat$) . This results in the categories shown in line 2 of Figure 3. The category $((\text{ObtainChickenMeat})/\{\text{Gather}(U_1, G)\})\backslash\{\text{Move}(U_1)\}$ requires a $\text{Move}(U_1)$ to its left. Binding U_1 to $Player_1$ unifies $\text{Move}(U_1 = Player_1)$ with $\text{Move}(U_1)$ and allows leftward application to produce $(\text{ObtainChickenMeat})/\{\text{Gather}(U_1, G)\}$ shown in line 3. This category expects a $\text{Gather}(U_1, G)$ to the right. Rightward application can be used to produce ObtainChickenMeat. Since there are no more category arguments or actions, parsing ends, hypothesizing an instance of the ObtainChickenMeat plan.

3 Supervised CCG Learning

This work focuses on extending the CCG learning algorithm Lex_{Greedy}. Lex_{Greedy} takes as input a labeled training dataset D and an initial lexicon Λ_{init}. Λ_{init} provides minimal prior knowledge about a domain by mapping each action type to a single atomic category in the lexicon. Intuitively, this lexicon indicates that each action is generated by a single, unique task. The labeled training dataset D is a set of sequences of observed actions (called plan trace) $\vec{\sigma}_i = \langle a_1, a_2, \ldots a_m \rangle$ paired with a single top-level task T_i (symbolic representation of an activity in a domain (Hogg, Kuter, and Muñoz-Avila 2010)): $D = \{\vec{\sigma}_i : T_i | i = 1 \ldots N\}$.

A CCG lexicon is learned by two, possibly interleaved, processes: *category generation* and *probability estimation*. Category generation entails hypothesizing complex CCG categories given the training data. In particular, category generation learns the set of CCG categories $\mathcal{C}$ from $\Lambda = \langle \Sigma, \mathcal{C}, f \rangle$. Probability estimation entails estimating the conditional probabilities $p(c_{i,j}|a_i)$ ($a_i \in \Lambda_\Sigma, c_{i,j} : p(c_{i,j}|a_i) \in \Lambda_{f(a_i)}$) in f. We summarize both algorithm's below and refer interested readers to the original works of Kantharaju, Ontañón, and Geib (2019a) for more details.

Lex_{Greedy} learns a mapping from plan traces to their corresponding top-level task, while abstracting common sequences of tasks. Prior to learning, preprocessing is applied on the plan traces in D to convert them into *category traces*, where each action is replaced with its action type's corresponding atomic category in Λ_{init}. Lex_{Greedy} works on this updated dataset $D_C = \{\vec{C}_i : T_i | i = 1 \ldots N\}$.

The category generation process first greedily learns a set of abstractions M, and then constructs complex categories given M. Each $m \in M$ contains a sequence of atomic categories $m_{\vec{C}}$ and an abstracted atomic category m_C. We note that atomic categories created by Lex_{Greedy} represents a sequence of action types, where some instantiation of the sequence is found in D. Specifically, category generation starts by finding the most common sequence of atomic categories $\vec{S}$ in the category traces whose frequency meets an *abstraction threshold* γ (percentage of the dataset), where $\vec{S}$ contains at least one category in $\Lambda_{init,\mathcal{C}}$. Next, for each training instance, category trace pair $\vec{\sigma}_i, \vec{C}_i$ ($1 \leq i \leq N$) relevant to $\vec{S}$, Lex_{Greedy} retrieves non-overlapping sequences of actions from $\vec{\sigma}_i$ that is an instantiation of the action types represented by the atomic categories in $\vec{S}$.

For each non-overlapping sequence, Lex_{Greedy} constructs m in the following way. First, the algorithm retrieves the objects from the actions and constructs an ordered set of objects consistent with the ordering of the actions. Given this, Lex_{Greedy} gets the types for each object, and construct an ordered set of typed variables where each type is given a variable name based on its location in the ordered set. Next, $m_{\vec{C}}$ is set as $\vec{S}$, and an atomic category m_C is constructed by constructing an identifier for $\vec{S}$ as its name and the ordered set as its parameters. Finally, m_C replaces $\vec{S}$ in $\vec{C}_i$.

This process is repeated until all abstractions are found or each category trace has exactly one category from $\Lambda_{init,\mathcal{C}}$. Lex_{Greedy} then constructs a set of abstractions for $\vec{C}_i : T_i \in D_C$, where m_C is the top-level task T_i and $m_{\vec{C}}$ is $\vec{C}_i$. Once all abstractions M are constructed, complex categories are construct for each $m \in M$ by first removing a category $c \in m_{\vec{C}}$ from $m_{\vec{C}}$ where $c \in \Lambda_{init,\mathcal{C}}$. Next, m_C is set as the root and the categories in $m_{\vec{C}}$ become arguments. The category is then assigned to the action type corresponding to c.

Probability (or parameter) estimation entails estimating the conditional probabilities $p(c_{i,j}|a_i)$ ($a_i \in \Lambda_\Sigma, c_{i,j} : p(c_{i,j}|a_i) \in \Lambda_{f(a_i)}$) in f. These probabilities are estimated based on the frequency of the category being constructed during category generation. Categories whose conditional probabilities are below a *pruning threshold* are removed.

4 CCG Learning With Type Trees

A limitation of Lex_{Greedy} is that it does not leverage a-priori type tree knowledge. Specifically, it assumed that each type symbol in the domain model did not have any specializations or generalizations. This limits application to domains such as Minecraft or Monroe. This section presents necessary changes to Lex_{Greedy} to leverage knowledge of type trees. We assume that these type trees capture the type/subtype relations in the domain as inheritance trees.

Our extension is aimed at the category generation process of CCG learning. Suppose we have two common sequences of atomic categories that abstract to two atomic categories t_1 and t_2 with the same name. In this case, we want to construct a single atomic category for both common sequences. However, the set of parameters (typed variables) for each category can have different types. Thus, we need to compute generalized types for each variable in t_1 and t_2. Figure 4 provides such an algorithm for finding an ordered set S of generalizations of types for variables in t_1 and t_2 given the type trees T from the domain. If this algorithm is successful, a single atomic category can be constructed for both sequences. However, if unsuccessful, two separate categories have to be constructed.

In Figure 4, $t_{1,i}$ and $t_{2,i}$ (where $1 \leq i \leq n$) correspond to a typed variable in atomic categories t_1 and t_2, and we note that the variables of $t_{1,i}$ and $t_{2,i}$ will be the same. First, several quick checks are made (lines 4-5) to ensure that both categories can be generalized. Next, the algorithm finds the least general generalization (LGG) for each type in the category. An LGG is defined as the first type in a type tree that is more general than both $t_{1,i}$ and $t_{2,i}$. If one is more general to the other, then the more general type is the LGG.

If the types of $t_{1,i}$ and $t_{2,i}$ are the same, then they are the LGG of each other and $t_{1,i}$ is added to S. If they are not the same, then the algorithm gets the trees $T_1, T_2 \in T$ which contain the types of $t_{1,i}$ and $t_{2,i}$. Next, the algorithm computes the path from the type of $t_{1,i}$ to the root of T_1 (P_1) and the path from the type of $t_{2,i}$ to the root of T_2 (P_2). Each path is a sequence of types from the starting type to the root type. *CategoryGeneralization* then finds the first common type ω from P_1 and P_2 using the $FirstCommon(P_1, P_2)$ function. $FirstCommon$ traverses the sequence of types in P_1 and checks to see if any exist in P_2. If no common element (i.e. $\omega = nil$) is found, this implies that $t_{1,i}$ and $t_{2,i}$ do not have an LGG, and the algorithm fails.

```
1:  procedure CategoryGeneralization(t1, t2, T)
2:      t1, t2 → Atomic Categories, T → Type Trees
3:      S = ∅, n = |args(t1)|
4:      if name(t1) ≠ name(t2) ∧ |args(t1)| ≠ |args(t2)| then
5:          return Failure
6:      for i = 1 ... n do
7:          if type(t1,i) == type(t2,i) then
8:              S ← append(S, t1,i)
9:          else
10:             T1 → Tree containing type(t1,i) from T
11:             T2 → Tree containing type(t2,i) from T
12:             P1 ← Path from type type(t1,i) to root of T1
13:             P2 ← Path from type type(t2,i) to root of T2
14:             ω ← FirstCommon(P1, P2)
15:             if ω ≠ nil then
16:                 S ← append(S, (var(t1,i) − ω))
17:             else
18:                 return Failure
19:     if S = ∅ then
20:         return Failure
21:     else
22:         return S
```

Figure 4: Generalization of Variables in Atomic Categories

Given this, the category generation process of Lex_{Greedy} is updated as follows. First, the process constructs a set of abstractions $m \in M$ as per the original Lex_{Greedy} algorithm described in Section 3. Next, the abstracted atomic category m_C and each atomic category in $m_{\vec{C}}$ is generalized via the *CategoryGeneralization* algorithm from Figure 4. Finally, the process constructs complex categories using this updated M as per the original Lex_{Greedy} algorithm.

We provide an quick example of *CategoryGeneralization*'s execution based on the type tree in Figure 1 on the two following categories:

$$Cat_1(W - \textit{Iron Sword}, X - \textit{Wheat Seeds})$$
$$Cat_1(W - \textit{Sword}, X - \textit{Pumpkin Seeds})$$

We note that all quick checks are passed (same category name and number of arguments). Next, *CategoryGeneralization* checks the types *Iron Sword* and *Sword*. Since both are not the same type, the algorithm checks for its LGG by computing $P_1 = \langle \textit{Iron Sword}, \textit{Sword}, \textit{Item} \rangle$ and $P_2 = \langle \textit{Sword}, \textit{Item} \rangle$. Here, the first common type is *Sword*, and S is updated to be $S = \{W - \textit{Sword}\}$. Next, *CategoryGeneralization* checks the types *Wheat Seeds* and *Pumpkin Seeds*. Similar to the previous pair of types, the algorithm checks an LGG by computing $P_1 = \langle \textit{Pumpkin Seeds}, \textit{Seeds}, \textit{Item} \rangle$ and $P_2 = \langle \textit{Wheat Seeds}, \textit{Seeds}, \textit{Item} \rangle$. Here, the first common type is *Seeds*, and S is then updated to be $S = \{W - \textit{Sword}, X - \textit{Seeds}\}$. Given this, Lex_{Greedy} can now construct the atomic category $Cat_1(W - \textit{Sword}, X - \textit{Seeds})$.

5 Experiments

We aim to answer the question *How well does CCG learning apply domains with type trees?* As such, our experi-

ments focus on evaluating the performance of CCG learning (specifically Lex_{Greedy}) for plan recognition in Minecraft and Monroe (Blaylock and Allen 2005). We also structurally compare the CCGs learned by Lex_{Greedy} with the hand-authored model to understand their differences. All experiments were run on a machine with 3.40GHz Intel i7-6700 CPU and 32 GB RAM. We use Monte-Carlo Tree Search (MCTS) for CCG plan recognition (Kantharaju, Ontañón, and Geib 2019b) as this approach has demonstrated success in RTS games from prior work. We first start by describing the Monroe and Minecraft datasets used in our experiments. Next, we define all tunable parameters for both Lex_{Greedy} and MCTS. Finally, we describe metrics used in our experiments, experiment setup, and analyze results.

The Monroe domain captures plans for disaster relief including, providing medical aid, plowing snow, and clearing debris from roads. We constructed a corpus of 1000 plans for various tasks in the domain using the publicly-available Monroe dataset generator.[1] This generator uses a modified version of SHOP2 (Nau et al. 2003) to generate a set of valid plans for a task, and randomly chooses a plan from the set to add to the corpus. We did not use the available dataset *Monroe 5000* as it contained plans for multiple tasks. This breaks the assumption that Lex_{Greedy} requires that each plan trace in its training corpus maps to a single top-level task.

We constructed a Minecraft dataset as follows. We build an interface over Project Malmö to extract a learning dataset. In this work, we assume full observability of the Minecraft environment. At a high-level, we have a single agent execute predefined tasks, and extract observed actions from the agent. The five tasks in our dataset are *obtain beef*, *obtain chicken meat*, *obtain potato*, *obtain pumpkin pie* and *obtain bread*. Specifically, the agent uses SHOP2 to generate symbolic plans. These plans are then fed into the Minecraft interface, and observable actions are executed in the environment. The set of observable actions are **gather**, **select**, **move**, **look at**, **attack**, **harvest** and **use**. A dataset for CCG learning is then constructed by extracting sequences of these observed actions and their corresponding task. This dataset assumes that each sequence of observed actions corresponds to a single task (no interleaving plans). Additionally, this dataset is noisy as there are various plans in the dataset that are missing observations. This is most likely due to the interface not registering those actions.

Both the Minecraft and Monroe datasets have an imbalance in the number of instances per tasks. In Minecraft, this is a result of the agent's policy for choosing tasks. Almost 42% of the dataset corresponds to the *obtain beef* task as the agent preferred this task over others. In Monroe, tasks are chosen using a random weighted distribution. The majority task *plow road* makes up approximately 24% of the total instances. To address this imbalance, we additionally construct a balanced dataset by oversampling the minority tasks.

For our experiments, we tune the number of iterations of MCTS to 15000. We believed that this was more than enough to recognize many of the tasks in the Minecraft domain. We also use an ϵ-greedy tree policy with $\epsilon = 0.4$, and a random playout policy. For Lex_{Greedy}, we set the abstraction threshold γ to be 50% of the dataset and the pruning threshold τ to be 0, which indicates no pruning.

We apply two metrics to evaluate performance of CCG learning. The first metric is *task recognition accuracy*:

$$Accuracy = 100 * \frac{Number\ of\ correctly\ predicted\ task}{Total\ Number\ of\ Testing\ Instances}$$

The second metric is the *convergence point* for successfully predicted tasks (Blaylock 2005). The convergence point is computed as the minimum percentage of actions in a plan trace required before plan recognition correctly predicts the task as the most likely, and continues to predict that task until the end of the plan trace. Early recognition in Minecraft is important as it allows one to act quickly to the needs of players or agents. Thus, this metric should be low.

Hand-Authored vs. Learned CCG

We evaluate Lex_{Greedy} against a hand-authored CCG, random recognizer, and biased recognizer. A random plan recognizer chooses a task uniformly at random from the set of possible tasks. A biased recognizer chooses the most frequent task in the dataset. For the unbalanced dataset, this is the majority task (*obtain beef* for Minecraft and *plow road* for Monroe). The biased recognizer is the same as the random recognizer for the balanced dataset, but we include it for completeness. For this evaluation, we disabled the probability estimation in Lex_{Greedy}. This resulted in a CCG where categories associated with each action type had a uniform probability distribution. We refer to this version of Lex_{Greedy} as Lex^*_{Greedy}. We also trained and tested Lex^*_{Greedy} on the entire dataset. Both were done to provide a fair comparison to the hand-authored CCG.

The hand-authored Minecraft and Monroe model was constructed and refined over a single day by a single researcher. The researcher has over 20 years of video game experience, with approximately 20-30 hours put into Minecraft. Explicitly, the researcher has done various activities in the game such as maintaining a farm of animals and crops, and built a house. The researcher however, has not played other crafting games. With respect to Monroe, the researcher has little prior knowledge of the plans in the domain. They also have significant knowledge of CCGs, and has hand-authored CCGs in the past.

The authoring process for both domains was as follows. For both domains, models were constructed using a text editor. In particular, the Minecraft domain model was constructed by first outlining plans for each executable task in the Minecraft dataset. Second, relevant CCG categories for each plan were then constructed. Third, the model was refined and tested against these plans until all tasks could be recognized. The Monroe model was constructed by adapting the SHOP2 model used to construct the dataset. As such, the results for this model will be better than one without knowledge of the SHOP2 model. However, the researcher only checked for syntactic errors, and didn't test recognition. Thus, we expect reduced performance for plan recognition due to errors. Over 100 executions of Lex^*_{Greedy} on the unbalanced datasets, Lex^*_{Greedy} took on average 2.94 seconds

[1] https://www.cs.rochester.edu/research/cisd/resources/monroe-plan/

	Minecraft		Monroe	
Measures/Metrics	Hand-Authored	Lex^*_{Greedy}	Hand-Authored	Lex^*_{Greedy}
# of complex categories with only right arguments	2	10	5	287
# of complex categories with only left arguments	5	4	43	3
# of complex categories with both right and left args. (i.e. hybrid categories)	11	16	34	146
Total # of categories in CCG	26	40	121	464
Total # of action types in CCG	7	7	28	28
Statistics for # of categories / action type (Avg/Std. Dev/Min/Max)	3.71 / 2.12 / 1 / 6	5.71 / 3.69 / 2 / 14	4.32 / 5.06 / 1 / 25	16.57 / 49.34 / 1 / 247

Table 1: Comparison of CCG Learned by Lex^*_{Greedy} to Hand-Authored CCG

	Minecraft				Monroe			
	Hand-authored	Lex^*_{Greedy}	Random Rec.	Biased Rec.	Hand-authored	Lex^*_{Greedy}	Random Rec.	Biased Rec.
Balanced	53.22%	64.06%	20.00%	20.00%	64.81%	28.16%	10.00%	10.00%
Unbalanced	53.95%	29.68%	20.00%	42.40%	74.8%	33.2%	10.00%	24.00%

Table 2: Recognition Accuracy for Hand-Authored vs. Learned CCG (higher=better)

	Minecraft		Monroe	
	Hand-authored	Lex^*_{Greedy}	Hand-authored	Lex^*_{Greedy}
Balanced	84.91%	70.28%	90.24%	75.77%
Unbalanced	85.00%	62.60%	90.74%	82.02%

Table 3: Convergence Point for Hand-Authored and Learned CCG (lower=better)

(with standard deviation of 0.07) to learn a CCG for the Minecraft domain, and 6.91 seconds (with standard deviation of 0.04) for the Monroe domain. Given that both models each took one day to construct, learning a model is *29388x* faster for Minecraft and *12504x* faster for Monroe. Thus, we see that automatically constructing CCGs from training data can reduce time needed to construct a CCG.

Table 1 provides several structural metrics for CCGs, their descriptions, and results on different CCGs. We found three different types of complex categories in the CCG, *left*, *right*, and *hybrid*, defined in rows 3-5 of Table 1. For the Minecraft domain, we see that many of the categories in the CCGs are hybrids. However, many of the hybrid categories for the learned CCG have more right arguments than left, especially categories for *obtain bread*. The hand-authored model on the other hand had hybrid categories with more left arguments. For the Monroe domain, many of the learned complex categories are right and hybrid. Interestingly, many of these hybrid categories contained more right arguments than left.

For both domains, we see that the number of categories generated during learning is higher than a hand-authored model, even having more categories per action type. This makes sense as the hand-authored CCGs were made by a researcher knowledgeable about CCGs. The number of categories per action type is the branching factor for CCG plan recognition search. As such, they strategically placed categories with specific actions to reduce complexity during plan recognition. For the Monroe domain, the number of categories per action type for the learned model is much higher than the hand-authored model. This is because two action types had a large number of categories associated with it, with one having 247 categories. The learned CCGs had more categories per action type than the hand authored model, but also took significantly less time to construct.

Next, we compare the performance of the learned CCG against a hand-authored model for plan recognition. Table 2 contains recognition accuracy for Lex^*_{Greedy} and the hand-authored CCG for both Minecraft and Monroe. For the Minecraft domain, Lex^*_{Greedy} achieved recognition accuracy of 64.06% on the balanced dataset, outperforming the hand-authored model with an accuracy of 53.22%. Both additionally outperformed the random and recognizer biased recognizer. The hand-authored model split recognition between *obtain beef* and *obtain chicken meat* almost equally while Lex^*_{Greedy} favored *obtain chicken meat*. This is a result of ambiguity between the tasks; both tasks use the same sequence of actions, differing only in the parameters of the actions. We believe the recognition accuracy difference was due to this discrepancy. We also believe that the failure of the hand-authored CCG to recognize bread added to this difference. This failure is due to the constraints placed on MCTS. Running the CCG on higher number of iterations yielded successful recognition for *obtain bread*.

For the Minecraft domain, Lex^*_{Greedy} achieved an accuracy of 29.68% compared to the hand-authored model's 53.95% on the unbalanced dataset. Both outperformed a random recognizer, but only the hand-authored outperformed a biased recognizer. Lex^*_{Greedy} had poor accuracy compared to the hand-authored model. This was most likely a result of recognizing *obtain potato* as *obtain bread*. Upon inspection of the learned CCG, we noticed that Lex^*_{Greedy} learned a CCG category for *obtain bread* that could recognize the sequence of actions for *obtain potato*. This means that the Minecraft dataset contained a plan for *obtain bread* that was similar to *obtain potato*. We also expected *obtain chicken meat* to be recognized as *obtain beef* as it is the most frequent task in the dataset. However, our results indicate the opposite. We noticed that for a majority of the instances for the two tasks, the two task's probabilities differed by about 0.005, implying that there was one or two additional hypothesis that yielded *obtain chicken meat* over *obtain beef*.

For the Monroe domain, we see that both the hand-authored model and Lex^*_{Greedy} outperformed the random and biased baselines. However, the hand-authored model outperformed Lex^*_{Greedy}. This is expected as Lex^*_{Greedy} learned an unoptimized CCG model. Recall from Table 1 that Lex^*_{Greedy} learned a CCG model that inadvertently increased the complexity for plan recognition. As such, given the number of iterations for plan recognition, Lex^*_{Greedy} had

lower accuracy. One interesting thing about results on the hand-authored model is that it failed to recognize *fix power line*. Recall that the hand-authored domain was only tested for syntactic correctness. The researcher who constructed the model made an error constructing a category for that task, thus MCTS could not recognize that task. However, Lex^*_{Greedy} learned categories for that task that were correct. Thus, the learned model recognize that task correctly. We get 74.05% (balanced) and 81.0% (unbalanced) recognition accuracy using a fixed hand-authored model.

Table 3 provides the convergence points for the hand-authored model and Lex^*_{Greedy} for Monroe and Minecraft. Overall, Lex^*_{Greedy} constructed CCGs that recognized accurate plans early for both domains. For Minecraft, Lex^*_{Greedy} constructed a CCG that recognized tasks such as *obtain beef* and *obtain pumpkin pie* almost halfway through the plans. For Monroe, the constructed CCGs recognized tasks such as *plow road* and *quell riot* prior to plan completion. On the other hand, the hand-authored model required almost 100% to recognize many of the tasks in both domains. For example, *obtain beef* and *obtain chicken meat* for Minecraft and *provide medical attention* and *quell riot* for Monroe required 100% of the plan. This makes sense as the hand-authored models was designed to contain more leftward complex CCG categories than rightward ones. These results also align with Table 1. Recall that Lex^*_{Greedy} learned complex categories that had more right arguments than left while the hand-authored model had the opposite. Thus, the hand-authored model should need more actions to recognize tasks as it delays recognition until later in a plan.

6 Related Work

There are two related bodies of work: Hierarchical Task Network (HTN) and Teleoreactive Logic Program (TLP) learning. HTNs are a modeling formalism for specifying the structure of plans in a hierarchical manner. Some approaches learn search control (i.e. method preconditions) for HTNs. In particular, Ilghami et al. (2002) and Ilghami et al. (2005) used candidate elimination (Mitchell 1977) to learn HTN method preconditions while HTN-Learner (Zhuo et al. 2009) used a MAX-SAT solver (Heras, Larrosa, and Oliveras 2008) to automatically construct method preconditions from partially-observable states. Our work is more related to learning task hierarchies, similar to HTN-Maker (Hogg, Muñoz-Avila, and Kuter 2008), HTN-MakerND (Hogg, Kuter, and Muñoz-Avila 2009), Q-Maker (Hogg, Kuter, and Muñoz-Avila 2010), HTN Learning System (Yang, Pan, and Pan 2007), and Word2HTN (Gopalakrishnan, Muñoz-Avila, and Kuter 2016).

Our work is closely related to Nguyen et al. (2017), who applies Word2Vec (Mikolov et al. 2013) to learn HTNs for Minecraft, and TLP learning. TLPs are a framework for encoding knowledge using ideas from logic programming, reactive control, and HTNs. TLPs can represent type trees using a concepts, which is a representation that can be used to encode state information in a hierarchical manner. TLPs have two types of concepts: primitive and non-primitive. Here, each type in the type tree would become a concept, where the leaves of the type tree would be primitive concepts while the internal nodes are non-primitive concepts. Nejati, Langley, and Konik (2006) and Li et al. (Li et al. 2009) both present techniques for learning the HTN methods for TLPs from data. However, the work was not explicitly applied to learning from type trees. One main difference between our work and those in HTN and TLP learning is that our work is applied to CCGs instead of HTNs or TLPs. Another difference is that our work is applied to the problem of plan recognition, while most work has been applied to planning.

7 Conclusion

This paper demonstrated learning CCGs on domains with type trees. Our first contribution was an extension of Lex_{Greedy} to generalize two tasks given type trees. Our second contribution is a comparison of hand-authored CCG models to ones learned by Lex_{Greedy}. Our results indicate that Lex_{Greedy} was able to automatically construct a CCG successfully to recognize plans in Minecraft and Monroe, outperforming a random recognizer and biased recognizer. Our results also indicate that learned CCGs allowed early recognition of plans compared to a hand-authored model.

As part of our future work, we would like to apply the learned CCGs generated by Lex_{Greedy} to hierarchical planning in Minecraft. Second, we want to integrate the MCTS CCG plan recognition algorithm directly into the Minecraft interface. Both of these would allow us to construct a full game-playing agent in Minecraft. Finally, we want to look at other domains with type trees.

8 Acknowledgements

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001119C0128. We like to thank the US Naval Research Laboratory for providing the Minecraft dataset.

References

Alford, R.; Bercher, P.; and Aha, D. W. 2015. Tight Bounds for HTN Planning with Task Insertion. In *Proc. of the 25th Int. Joint Conf. on Artificial Intelligence*, 1502–1508. {AAAI} Press.

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016. Hierarchical Planning: Relating Task and Goal Decomposition with Task Sharing. In *Proc. IJCAI*, 3022–3028.

Blaylock, N., and Allen, J. 2005. Generating Artificial Corpora for Plan Recognition. In *Proc. of the 10th International Conference on User Modeling*, UM'05, 179–188. Berlin, Heidelberg: Springer-Verlag.

Blaylock, N. 2005. *Towards tractable agent-based dialogue*. Ph.D. Dissertation, University of Rochester.

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

Erol, K.; Hendler, J.; and Nau, D. S. 1994a. HTN planning: Complexity and expressivity. In *Proc. of the 8th AAAI Conference on Artificial Intelligence*, 1123–1123.

Erol, K.; Hendler, J.; and Nau, D. S. 1994b. UMCP: A Sound and Complete Procedure for Hierarchical Task Network Planning. In *Proc. of the 2nd International Conference on Artificial Intelligence Planning Systems*, 249–254.

Fox, M.; Long, D.; and Magazzeni, D. 2017. Explainable Planning. *CoRR* abs/1709.1.

Geib, C. W., and Goldman, R. P. 2001. Probabilistic Plan Recognition for Hostile Agents. In *Proc. of the Fourteenth International Florida Artificial Intelligence Research Society Conference*, 580–584. AAAI Press.

Geib, C. W., and Kantharaju, P. 2018. Learning Combinatory Categorial Grammars for Plan Recognition. In *Proc. of the 32nd AAAI Conference on Artificial Intelligence*.

Geib, C. W., and Steedman, M. 2007. On Natural Language Processing and Plan Recognition. In *Proc. of the 20th International Joint Conferences on Artificial Intelligence*, 1612–1617.

Geib, C. W. 2009. Delaying commitment in plan recognition using combinatory categorial grammars. In *Proc. of the 21st International Joint Conference on Artificial Intelligence*, 1702–1707.

Geib, C. W. 2016. Lexicalized Reasoning About Actions. *Advances in Cognitive Systems* Volume 4:187–206.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge University Press.

Gopalakrishnan, S.; Muñoz-Avila, H.; and Kuter, U. 2016. Word2HTN:learning task hierarchies using statistical semantics and goal reasoning. In *Proc. of the IJCAI-16 Workshop on Goal Reasoning*.

Guss, W. H.; Houghton, B.; Topin, N.; Wang, P.; Codel, C.; Veloso, M.; and Salakhutdinov, R. 2019. MineRL: A Large-Scale Dataset of Minecraft Demonstrations. In *IJCAI 2019*.

Heras, F.; Larrosa, J.; and Oliveras, A. 2008. MiniMaxSAT: An efficient weighted Max-SAT solver. *Journal of Artificial Intelligence Research* 31:1–32.

Hogg, C.; Kuter, U.; and Muñoz-Avila, H. 2009. Learning Hierarchical Task Networks for Nondeterministic Planning Domains. In *Proc. of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, 1708–1714. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Hogg, C.; Kuter, U.; and Muñoz-Avila, H. 2010. Learning Methods to Generate Good Plans: Integrating HTN Learning and Reinforcement Learning. In *Proc. of the 24th AAAI Conference on Artificial Intelligence*, 1530–1535.

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, AAAI'08, 950–956. AAAI Press.

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2016. Learning hierarchical task models from input traces. *Computational Intelligence* 32(1):3–48.

Ilghami, O.; Nau, D. S.; Muñoz-Avila, H.; and Aha, D. W. 2002. CaMeL: Learning Method Preconditions for HTN Planning. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems*, 131–141.

Ilghami, O.; Muñoz-Avila, H.; Nau, D. S.; and Aha, D. W. 2005. Learning approximate preconditions for methods in hierarchical plans. In *Proceedings of the 22nd International Conference on Machine Learning*, 337–344. ACM.

Johnson, M.; Hofmann, K.; Hutton, T.; and Bignell, D. 2016. The Malmo Platform for Artificial Intelligence Experimentation. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, 4246–4247.

Kantharaju, P.; Ontañón, S.; and Geib, C. W. 2019a. Extracting CCGs for plan recognition in RTS games. In *Proc. of the Workshop on Knowledge Extraction in Games 2019*.

Kantharaju, P.; Ontañón, S.; and Geib, C. W. 2019b. Scaling up CCG-Based Plan Recognition via Monte-Carlo Tree Search. In *Proc. of the IEEE Conference on Games 2019*.

Langley, P., and Choi, D. 2006. Learning recursive control programs from problem solving. *Journal of Machine Learning Research* 7:493–518.

Li, N.; Stracuzzi, D. J.; Cleveland, G.; Langley, P.; Konik, T.; Shapiro, D.; Ali, K.; Molineaux, M.; and Aha, D. W. 2009. Learning hierarchical skills for game agents from video of human behavior. Technical report, Knexus Research Corporation.

Mikolov, T.; Chen, K.; Corrado, G.; and Dean, J. 2013. Efficient estimation of word representations in vector space. *Proceedings of the ICLR-13 Workshop Track*.

Mitchell, T. M. 1977. Version spaces: A candidate elimination approach to rule learning. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, 305–310. Morgan Kaufmann Publishers Inc.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20(1):379–404.

Nejati, N.; Langley, P.; and Konik, T. 2006. Learning hierarchical task networks by observation. In *Proc. 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 *Proceedings of the 2017 IEEE Conference on Computational Intelligence and Games*, 226–231.

Shivashankar, V. 2015. *Hierarchical goal networks: Formalisms and algorithms for planning and acting*. Ph.D. Dissertation, University of Maryland.

Yang, Q.; Pan, R.; and Pan, S. J. 2007. Learning recursive HTN-method structures for planning. *Proceedings of the ICAPS-07 Workshop on Planning and Learning*.

Zhuo, H. H.; Hu, D. H.; Hogg, C.; Yang, Q.; and Muñoz-Avila, H. 2009. Learning HTN method preconditions and action models from partial observations. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, 1804–1809. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.