μCCG, a CCG-based Game-Playing Agent for μRTS

Pavan Kantharaju and Santiago Ontañón
Drexel University
Philadelphia, Pennsylvania, USA
pk398@drexel.edu, so367@drexel.edu

Christopher W. Geib
SIFT LLC
Minneapolis, Minnesota, USA
cgeib@sift.net

***Abstract*—This paper presents a Combinatory Categorial Grammar-based game playing agent called μCCG for the Real-Time Strategy testbed μRTS. The key problem that μCCG tries to address is that of adversarial planning in the very large search space of RTS games. In order to address this problem, we present a new hierarchical adversarial planning algorithm based on Combinatory Categorial Grammars (CCGs). The grammar used by our planner is automatically learned from sequences of actions taken from game replay data. We provide an empirical analysis of our agent against agents from the CIG 2017 μRTS competition using competition rules. μCCG represents the first complete agent to use a learned formal grammar representation of plans to adversarially plan in RTS games.**

***Index Terms*—adversarial planning, RTS games, combinatory categorial grammars**

I. Introduction

Real-Time Strategy (RTS) games are particularly useful in AI research because they provide a way to test AI that solve real world problems in a controlled environment. Since the call to research by Michael Buro in 2003 [1], RTS games have been used to solve challenging real-time AI problems such as decision making under uncertainty, resource management, opponent modeling, and adversarial planning. This work focuses on the problem of adversarial planning in deterministic and fully-observable RTS games by using Combinatory Categorial Grammars [2]. Combinatory Categorial Grammars (CCGs) are a well known grammar formalism developed for Natural Language Processing. Recent work by Geib [3] and Geib and Goldman [4] used probabilistic Combinatory Categorial Grammars to represent plans in a number of domains for the problem of plan recognition [5].

This work focuses on the domain of RTS games. Game tree search does not apply well to RTS games due to the enormous search space that needs to be traversed. Ontañón *et al.* describe a scenario where a 128x128 size game map with 400 units result in 16384^{400} possible game states [6]. Approaches to deal with this complexity in the literature range from hard-coding manually defined scripts (as is common in the StarCraft AI competition [6], to using abstraction in the action or state space [7]–[9], portfolio approaches [10], [11], or search strategies that attempt to scale up to the large branching factors of RTS games [12]. A promising approach to addressing this problem was work by Ontañón and Buro [13] who used an adversarial Hierarchical Task Network [14] (AHTN) planner to generate sequences of actions for playing μRTS. By integrating HTN planning with adversarial search, the advantages of domain-configurable planning, in terms of reduction of the search space, can be brought to RTS games. However, the HTN definitions used by the AHTN had to be hand authored.

This paper outlines two contributions. First, we present an alternative hierarchical planning formulation based on CCGs in the form of a μRTS agent called μCCG. Second, we show that we can learn a CCG plan representation from sequences of actions collected from game replay data. This is done by using a known CCG lexicon learning algorithm Lex_{Learn} by Geib and Kantharaju [15] to learn common sequences of action triples used by different μRTS agents. We limited ourselves to sequences of action triples because we wanted to model reactive behavior in RTS gameplay.

This paper is structured as follows. In section II, we provide a brief background of CCGs. In section III, we provide a brief overview of Lex_{Learn}. In section IV, we provide a description of μCCG and the hierarchical adversarial planner. In section V, we provide an empirical analysis of our agent against agents from the CIG 2017 μRTS tournament. In section VI, we provide a brief description of related work. Finally, in section VII we provide concluding remarks and future work.

II. Combinatory Categorial Grammars

This section briefly describes plan CCGs following the definitions used in the work of Geib *et al.* [3], [4]. Each action in a planning domain is associated with a set of CCG categories that can be thought of as functions and are defined recursively. We define a set of CCG categories $\mathcal{C}$ as follows.

Atomic categories are defined as a finite set of base categories denoted by $\{A, B, C...\} \in \mathcal{C}$. **Complex categories** are defined as $Z/\{W, X, Y, ...\} \in \mathcal{C}$ and $Z\backslash\{W, X, Y, ...\} \in \mathcal{C}$ where $\mathcal{C}$ is a set of categories, $Z \in \mathcal{C}$, $\{W, X, Y, ...\} \neq \emptyset$, and $\{W, X, Y, ...\} \in \mathcal{C}$. Atomic categories can be thought of as a zero-arity function that transitions from any initial state to a state associated with the atomic category. Complex categories define curried functions from states to states [16] based on the two left associative operators "\" and "/". These operators each take a set of **arguments** (the categories on the right hand side of the slash, $\{W, X, Y...\}$), and produces

the state identified with the atomic category specified as its **result** (the category on the left hand side of the slash, Z). The slash also defines ordering constraints for plans, indicating where the category's arguments are to be found relative to the action. Forward slash categories find their arguments after the action, and backslash categories before it. We assume that all complex categories must be **leftward applicable** (all leftward arguments must be discharged before any rightward ones), and we only consider leftward applicable categories with atomic categories for arguments. A category R is the **root** or **root result** of a category G if it is the leftmost atomic result category in G. For example, for a complex category $(C\backslash\{A\})\backslash\{B\}$, the root would be C.

Using the above definition, a **plan lexicon** is a tuple, $\Lambda = \langle\Sigma, \mathcal{C}, f\rangle$, where Σ is a finite set of action types, $\mathcal{C}$ is a set of possible CCG categories, and f is a function such that $\forall a_i \in \Sigma : f(a_i) \rightarrow \{(c_{i,j} : \text{P}(c_{i,j}|a_i))\}$ such that $c_{i,j} \in \mathcal{C}$ and $\forall a_i, c_{i,j}, \sum_j \text{P}(c_{i,j}|a_i) = 1$. The function f maps each observable action, a_i, to a non-empty set of pairs $\{(c_{i,j} : \text{P}(c_{i,j}|a_i))\}$ each made up of a category, $c_{i,j}$, and the conditional probability that the action is assigned the category, $\text{P}(c_{i,j}|a_i)$. The definitions of actions and categories are extended to a first-order representation by introducing **parameters** for actions and atomic categories to represent domain objects and variables. The CCG learning algorithm Lex_{Learn} will learn a lexicon containing parameterized actions and categories, but the current version of the adversarial planner presented in this paper will not make use of any parameters during the planning process. We provide an explanation for this in Section IV. However, we still provide an example lexicon to illustrate the representation Lex_{Learn} generates:

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

Note that **Train**(U_1, T), **Attack**(U_2, U_3), and **Harvest**(U_4, R) each have two 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, $P(c_{i,j}|a_i) = 1$. A full discussion of a CCG lexicon with parameterized actions and categories can be found in Geib [17].

III. Lex_{Learn}

This section briefly describes Geib and Kantharaju's CCG learning algorithm Lex_{Learn} [15]. Interested readers are referred to the full paper for more information. Lex_{Learn} is a supervised domain-independent CCG lexicon learning algorithm that generates lexicons for the ELEXIR framework, a CCG-based plan recognition algorithm [3]. Lex_{Learn} was built on prior work by Zettlemoyer and Collins [18] on CCG lexicon learning for Natural Language Processing (NLP). This is the first work that applies the learned CCG lexicon to the problem of adversarial hierarchical planning.

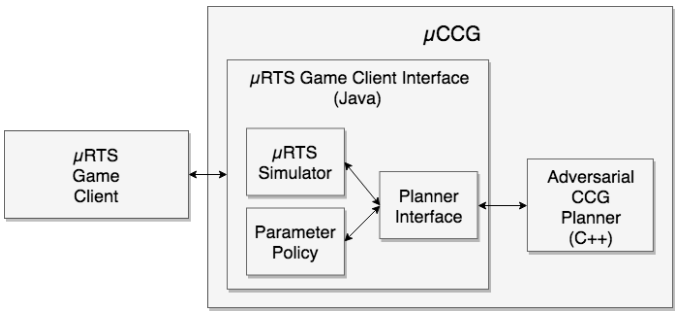

Fig. 1. μCCG Architecture Diagram

Lex_{Learn} takes as input an initial lexicon Λ_{init} and a training dataset, $\{(T_i, G_i) : i = 1...n\}$, where each T_i refers to a plan trace, a sequence of of observed actions that achieves a goal state G_i. Λ_{init} contains parameterized actions each paired with a single parameterized atomic category, and all actions in each T_i are contained in Λ_{init}. For each T_i, G_i pair in the training dataset, Lex_{Learn} incrementally updates Λ_{init} using two interleaved processes: category generation and parameter estimation.

Category generation is the process of generating new complex categories for actions in Λ_{init}. Given a plan trace T_i, the category generation process exhaustively enumerates the set of all possible categories for each action $a_j \in T_i$ using a set of predefined category templates. In the original work, pruning was used to limit the set of learned categories for actions that occurred more than once in T_i. However, in this work, we do not prune any generated complex categories for these actions. Λ_{init} is then updated by adding all constructed complex categories to action a_j.

The second process, parameter estimation, estimates the conditional probabilities $P(c_{j,k}|a_j)$ of all the action category pairs in the updated lexicon using stochastic gradient ascent [18]. Intuitively, each $P(c_{j,k}|a_j)$ represents a weighted frequency of how often the category $c_{j,k}$ is assigned to the action a_j during plan recognition. A full definition of the gradient used for gradient ascent can be found in the original paper [15].

IV. μCCG Agent

This section describes our μRTS game-playing agent, μCCG. Figure 1 provides an architecture diagram of our agent. There are two main components to μCCG as seen in the diagram: adversarial CCG planner and μRTS game client interface. The game client interface contains three subcomponents. The μRTS simulator is used to simulate a μRTS game state for planning, and the planner interface provides a bridge between the adversarial CCG planner and the μRTS environment. We describe the adversarial CCG planner and the parameter policy components from Figure 1 below.

At each game frame, the agent generates the best possible set of actions it can find for a given game state, given an allotted time. As per the CIG 2018 μRTS tournament rules, our agent is time constrained to 100ms per game frame. Our

procedure ACCG(s, T_+, T_-, t_+, t_-, d)
  if $\neg canIssueActions(s', +) \wedge d == MD$ **then**
    return
  end if
  $s' = simulateUntilNextChoicePoint(s)$
  if $GameEnd(s') \vee d \leq 0 \vee$ *both trees traversed* **then**
    return $(T_+, T_-, reward(s'))$
  end if
  if T_+ is traversed **then**
    Complete planning for min, ignore max
  end if
  if T_- is traversed **then**
    Complete planning for max, ignore min
  end if
  if $canIssueActions(s', +)$ **then**
    if $t_{-,c}$ is complex $\wedge$ root of $t_{+,c}$ is next **then**
      return $ACCG(\gamma(t_{+,a}, s'), T_+, T_-, t_+, t_-, d-1)$
    end if
    return $ACCGMax(s', T_+, T_-, t_+, t_-, d)$
  end if
  if $t_{-,c}$ is complex $\wedge$ root of $t_{-,c}$ is next **then**
    return $ACCG(\gamma(t_{-,a}, s'), T_+, T_-, t_+, t_-, d-1)$
  end if
  return $ACCGMin(s', T_+, T_-, t_+, t_-, d)$
end procedure
procedure ACCGMAX(s, T_+, T_-, t_+, t_-, d)
  if $t_{+,c}$ is atomic **then**
    $t = Update(T_+)$
    return $ACCG(\gamma(s, t_{+,a}), T_+, T_-, t, t_-, d-1)$
  end if
  $T_+^* = \bot, T_-^* = \bot, r^* = -\infty$
  if $t_+ == nil$ **then**
    $t_+ = T_+$
  end if
  $c_{next} = NextCat(t_{+,c})$
  $\mathcal{C} = AllDecompWithRoot(c_{next})$
  $\mathcal{C}'$ = N action, category pairs from $\mathcal{C}$ with highest UCB1 score
  for all $t \in \mathcal{C}'$ **do**
    $T_+' = AddToStack(t)$
    $(T_+', T_-', r') = ACCG(s, T_+', T_-, t, t_-, d)$
    if $r' > r^*$ **then**
      $T_+^* = T_+', T_-^* = T_-', r^* = r'$
    end if
  end for
  return (T_+^*, T_-^*, r^*)
end procedure

Fig. 2. Pseudocode for CCG Adversarial Planner

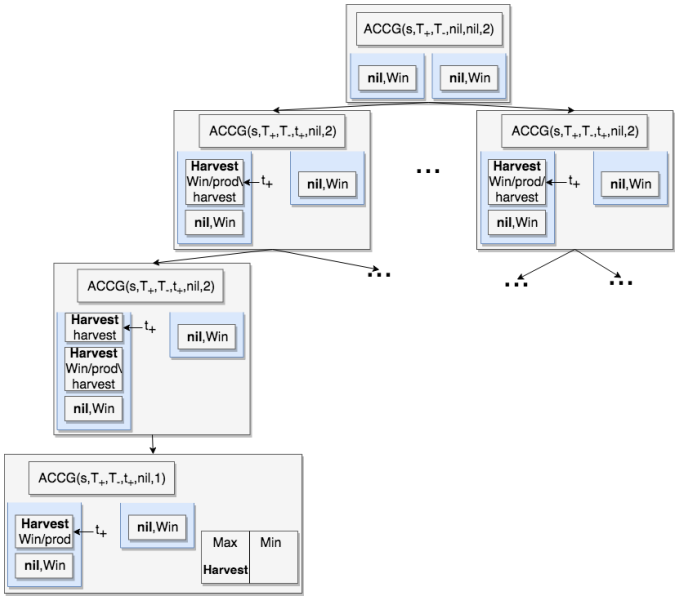

Fig. 3. Example Execution of Adversarial CCG Planner For Max Player

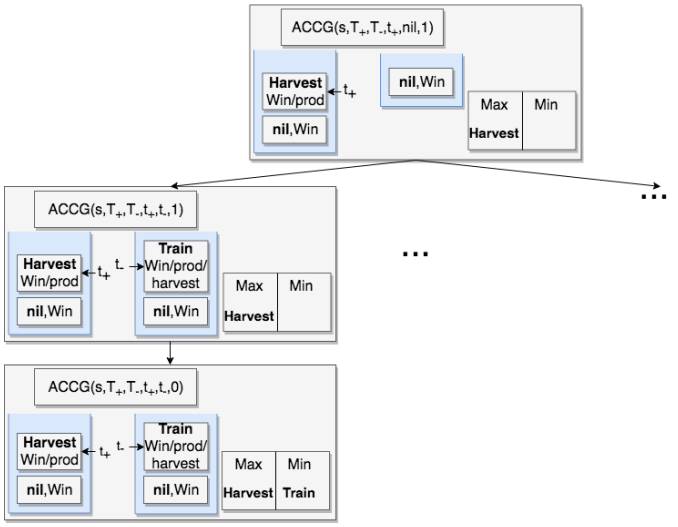

Fig. 4. Example Execution of Adversarial CCG Planner For Min Player

adversarial planner, motivated by Ontañón and Buro [13], uses a variant of minimax for RTS games.

Complex CCG categories represent a temporal relationship between states of a world. Given the complex category G/C\A, the sequence of actions resulting in state A must be completed and successful before executing the action(s) resulting in state G and state C. However, in the case of RTS games, actions can be executed in parallel in a given game state by multiple units. This requires us to relax the temporal restrictions defined by CCGs. For example, let A represent the state in which the agent executed the action **Train** for a base to train a worker unit and C represent the state in which the agent executed the **Attack** action for a heavy unit to attack an enemy unit. If both of these actions can be executed in a given game state, and **Train** fails to execute, that should not result in **Attack** failing as both actions are executed on different units. Therefore, during the planning process, if such a situation arises, instead of backtracking, the whole plan is still considered, and **Train** is replaced by an empty action.

In the current version of μCCG, our planner does not generate action parameters. Parameters for each action are generated using a hand-authored *Parameter Policy*. Although our planner can, in theory, generate action parameters, we don't currently use the planner to define parameters for actions because some of the parameters require information beyond what the state representation received by our planner currently contains, such as terrain and resources. For example, the attack command uses both unit and map layout, which the state

representation we currently provide to our planner ignores. Future work will look into augmenting the state representation with this information.

We define CCG decomposition as the process of expanding an atomic category c to a set of action category pairs whose root is the same as c. For example, if we have the category Win, a possible decomposition would be the action category pair (**Attack**, (Win/({harvest}))\{train}) (representing a plan where we first want to **Train**, then **Attack**, and then **Harvest**). It is possible the the number of decompositions for c might be very large (in this particular example, we might have a very large number of decompositions for how to Win in the learned lexicon). Thus, we select only a subset of N decompositions ($N = 5$ in our experiments). To select the subset, we currently use the Upper Confidence Bound (UCB1) [19] policy. In order to use UCB1, the planner keeps track of how many times each decomposition has been selected so far during planning, and the number of time c is decomposed (these counts are global for each player and are not reset throughout a whole game). We currently use the conditional probability $P(c_{j,k}|a_j)$ of an action category pair $(a_j, c_{j,k})$ from the lexicon as the reward value for UCB1. Notice that this probability never changes throughout the planning process, since it's only determined during category learning. Thus, the effect of UCB1 here is just to vary the subset of decompositions that are considered during planning, so that the planner does not always just consider the highest probability pairs. As part of our future work, we would like to use the actual achieved reward of each decomposition during gameplay instead, making our planner one step closer to a CCG-based MCTS planner, which would be the end goal.

Our planner generates plans using three actions from μRTS, **Attack**, **Harvest**, and **Return**, and a modified **Produce** action. The **Produce** action is modified to additionally include the unit being produced, such as **ProduceWorker** or **ProduceBarracks**, allowing the planner to determine build order instead of hard-coding it. This is done, since, as mentioned above, our planner does not currently generate action parameters. Thus, this encodes the unit type parameter to produce units without actually having any parameter. We assume the planner does not plan any movement actions. Initial experimentation resulted in our agent always choosing to move instead of any other action due to how frequent the movement action is in actual gameplay. Therefore, to prevent this, if **Attack**, **Harvest**, or **Return** need to move in order to succeed in a given game state, then a move action is issued.

Figure 2 provides pseudocode for our CCG adversarial planning algorithm. We provide pseudocode for only $ACCGMax$ in Figure 2, but $ACCGMin$ is a mirror definition. We refer to T_+ and T_- as the max and min player's **plan stack**. Action, category pairs are added to a plan stack based on when they are found during plan search. Figures 3 and 4 provide an example of plan stacks for both the max (left stack) and min (right stack) player. We define t_+ and t_- as pointers to the current decomposition in the stack, and d, MD, and s as current depth, maximum depth, and current game state.

There are four functions in the planner that interface with μRTS (seen in Figure 2). The first function $simulateUntilNextChoicePoint$ provides the μRTS framework with a game state s, and simulates until either player min or max can issue an action. However, given that the learned CCG lexicon cannot ensure that actions would be issued to all units in a given game state, it might be the case that even after exhausting plan search for the max player, max can still issue actions. This would result in the min player never getting a chance to perform search. To avoid this situation, we currently simulate the game state for one game frame before checking whether any player can issue an action to ensure that the game state advances and does not get stuck, and thus giving the planner a chance to search for actions for both players. This would not be an issue with hand-crafted lexicons that could be defined in such a way that this situation never arises, but our planner needs to be robust with respect to learned lexicons that do not necessarily ensure actions will be produced for all units in the game.

The next function $canIssueActions$ provides the μRTS framework with game state s and a player (either $+$, $-$, or ? referring to player max, min, or either) to determine if the player is ready to execute actions. If the player is ?, then player max is prioritized over player min. The next function γ applies a given action from a decomposition, $t_{+,a}$ or $t_{-,a}$, to a state s, returning the next game state. The final function $reward$ computes a reward using a reward function based on the given state.

The next four functions are specific to CCG adversarial planning. First, $NextCat$ takes a category c, and returns the leftmost atomic category that is not the root c_{left}. If c is atomic, then c is returned. For example, if c is Win/{prod}\harvest as seen in max's plan stack in Figure 3, the leftmost atomic category c_{left} would be harvest. Second, $AllDecompWithRoot$ decomposes c_{left} and returns the set of action category pairs that have c_{left} as a root ($\mathcal{C}$ in Figure 2). Third, $AddToStack$ pushes a given action category decomposition pair to a given stack.

The fourth and final function $Update$ propagates down the stack to find and return the next complex category to search. This function is called only when an action is generated by the planner. Until a complex category with at least one argument is found, the function pops off any atomic categories. If the function finds arguments from complex categories that have been fully reduced to a sequence of actions, it removes the argument and removes the category if the result of removal makes the category atomic.

Figures 3 and 4 provide a small example of the planner's execution tree where Figure 4 follows directly after Figure 3, so we first look at Figure 3. Each node in the execution tree represents a call to $ACCG$. At each game frame, the planner is given top-level goals which, in the case of the example, is Win for both the min and max player, and game state s. These top-level goals are added to the min and max plan stack T_- and T_+ as the action, category pair (**nil**,Win), and $ACCG(s, T_+, T_-, nil, nil, 2)$ is called.

Next, the planner simulates s using $simulateUntilNext-$

$ChoicePoint$ until at least one player can execute an action, and checks the available player using $canIssueActions$. In our example, this is the max player so the max player's function $ACCGMax$ is called. Since t_+ is nil, the planner sets this to the first action category pair in the stack (**nil**,Win). Win is already the leftmost atomic category, so $NextCat$ just returns Win. Next, using the function $AllDecompWithRoot$, the planner decomposes Win to two action category pairs, but we focus on one: (**Harvest**,Win/harvest\prod). Finally the planner adds the action category pair to max's plan stack, assigns the pointer t_+ to it and calls $ACCG(s, T_+, T_-, t_+, nil, 2)$.

Since the max player can still issue actions, the function $simulateUntilNextChoicePoint$ returns the next state and the planner calls $ACCGMax$ to decompose the complex category pointed to by t_+. In $ACCGMax$, $NextCat$ returns the leftmost atomic category harvest, and calls *AllDecompWithRoot* returning one decomposition, (**Harvest**, harvest). The planner adds this decomposition to the plan stack, points t_+ to it, and calls $ACCG(s, T_+, T_-, t_+, nil, 2)$. Next, $simulateUntilNextChoicePoint$ again returns the next game state, and calls $ACCGMax$ as the max player can still issue an action.

Calling *ACCGMax* again, the planner decomposes harvest to the action **Harvest**. The planner applies the action to the game state s, and calls *Update* to traverse down the stack. Since the planner already decomposed harvest, it pops the action category pair from the stack and its occurrence in the complex category Win/prod\harvest.

Next, as illustrated in Figure 4 the planner calls $ACCG(s, T_+, T_-, t_+, nil, 1)$. Note that the depth is only decremented when an action is issued by either player. After calling $simulateUntilNextChoicePoint$, planner detects that the max player can no longer issue actions, and the min player can. Similar to the max player, the planner calls $ACCGMin$ to decompose the min player's Win category into (**Train**,Win/prod/harvest), point t_- to it, and call $ACCG(s, T_+, T_-, t_+, t_-, 1)$.

Because this entire category is rightward-looking, the first leftmost category is actually the root, Win. This means that the next action to issue in the plan is **Train**. Therefore, instead of calling $ACCGMin$, the planner adds the action **Train** from the action category pair, and applies that action to the game state s. Finally, calling $ACCG(s, T_+, T_-, t_+, t_-, 0)$, completes planning.

Next, we look at the parameter policy, defined as follows. Any distance computation in our policy is computed using Euclidean distance. Given the **Attack** action, all produced offensive units are ordered to attack their closest enemy unit. We define offensive units as *Ranged*, *Heavy*, and *Light*. Worker units are not offensive because, from initial experiments, the agents would use the workers to attack instead of harvesting and building an army. Given the **Harvest** action, the agent finds the closest resource to a random base, and finds the closest worker to that resource. If the agent is within range of the resource, it harvests, and moves towards the resource if not. Given the **Return** action, the agent finds the closest base to a random worker. The agent then checks whether the worker is close to the base and moves if not.

The **Produce** action works differently. To prevent any resource contention when producing units, we only allow a single producing action to execute at a time. The planner dictates what units are constructed. However, in order to improve the game play strength of our agent for the competition setting, we impose some constraints on the planner output (this basically encodes our human domain knowledge of what a μRTS agent should do). The first restriction is that worker unit production is limited to a maximum of 2*(width of the map)/8 + 1 units to prevent over construction of workers in some maps. We wanted the agent to create at maximum three workers for the minimum map size of 8x8, and add two workers each time the game map quadrupled in size. We only used the width because most of the maps were squares. This limit will most definitely be changed for the competition. The second restriction is that we only allow a single worker unit to construct *Barracks* and *Base* to prevent the agent from constantly having to decide who should create these units. If the worker dies, another worker is chosen as the constructor.

V. Empirical Evaluation

The objective of our experiments is to test the effectiveness of a learned CCG lexicon and the adversarial CCG planner by evaluating it in the μRTS environment. In order to do this, we generated a dataset of plan traces from μRTS game replays using agents from last years's μRTS competition. Recall from Section III that a plan trace is defined as a sequence of observed actions. We then learned a CCG lexicon based on this dataset, and used it to play the game. We then evaluate game playing strength in the eight open maps that will be used for the 2018 μRTS competition, and compare against all the bots that participated in such competition.

We used the CCG lexicon learning algorithm Lex_{Learn} by Geib and Kantharaju [15] to generate a CCG lexicon for adversarial CCG planning. Lex_{Learn}'s parameters were tuned to the same values as Geib and Kantharaju's experiments. Recall from Section IV that the CCG adversarial planner only plans using four actions: **Attack**, **Harvest**, **Return**, and a modified **Produce** action. Below is the initial lexicon provided to Lex_{Learn}:

$$f(\textbf{Attack}) \rightarrow \{\text{attack} : 1\}$$
$$f(\textbf{Harvest}) \rightarrow \{\text{harvest} : 1\}$$
$$f(\textbf{Return}) \rightarrow \{\text{return} : 1\}$$
$$f(\textbf{ProduceBarracks}) \rightarrow \{\text{produce} : 1\}$$
$$f(\textbf{ProduceBase}) \rightarrow \{\text{produce} : 1\}$$
$$f(\textbf{ProduceWorker}) \rightarrow \{\text{produce} : 1\}$$
$$f(\textbf{ProduceLight}) \rightarrow \{\text{produce} : 1\}$$
$$f(\textbf{ProduceHeavy}) \rightarrow \{\text{produce} : 1\}$$
$$f(\textbf{ProduceRanged}) \rightarrow \{\text{produce} : 1\}$$

We note that **Attack**, **Harvest**, and **Return** have separate atomic categories, and each **Produce** action is given the same

TABLE I
TOURNAMENT RESULTS (WINS / LOSSES / TIES)

	POLightRush	POWorkerRush	RandomBiasedAI	NaiveMCTS	PVAIML_ED	StrategyTactics	μCCG	Total	Win Ratio
POLightRush	-	45-20-15	76-2-2	59-18-3	35-35-10	30-35-15	64-7-9	309-117-54	0.7000
POWorkerRush	20-45-15	-	71-0-9	32-39-9	35-35-10	31-38-11	64-10-6	253-167-60	0.5896
RandomBiasedAI	2-76-2	0-71-9	-	1-56-23	10-53-17	5-73-2	9-62-9	27-391-62	0.1208
NaiveMCTS	18-59-3	39-32-9	56-1-23	-	46-31-3	18-59-3	36-34-10	213-216-51	0.4969
PVAIML_ED	35-35-10	35-35-10	53-10-17	31-46-3	-	18-54-8	53-15-12	225-195-60	0.5313
StrategyTactics	35-30-15	38-31-11	73-5-2	59-18-3	54-18-8	-	70-1-9	329-103-48	0.7354
μCCG	7-64-9	10-64-6	62-9-9	34-36-10	15-53-12	1-70-9	-	129-296-55	0.3260

TABLE II
MAP RESULTS FOR μCCG (WINS / LOSSES / TIES)

Maps	POLightRush	POWorkerRush	RandomBiasedAI	NaiveMCTS	PVAIML_ED	StrategyTactics
FourBasesWorkers8x8	5-5-0	0-10-0	5-5-0	0-10-0	0-10-0	1-9-0
TwoBasesBarracks16x16	0-10-0	3-8-0	10-0-0	7-3-0	3-7-0	0-10-0
NoWhereToRun9x8	2-7-1	5-0-5	8-0-2	1-9-0	4-0-6	0-2-8
DoubleGame24x24	0-2-8	0-9-1	3-0-7	0-3-7	1-4-5	0-10-0
basesWorkers8x8A	0-10-0	0-10-0	6-4-0	1-9-0	7-2-1	0-10-0
basesWorkers16x16A	0-10-0	0-10-0	10-0-0	8-2-0	0-10-0	0-10-0
BWDistantResources32x32	0-10-0	1-9-0	10-0-0	7-0-3	0-10-0	0-9-1
(4)BloodBath.scmB	0-10-0	1-9-0	10-0-0	10-0-0	0-10-0	0-10-0

TABLE III
CIG 2018 μRTS TOURNAMENT MAPS AND GAME CYCLES

Maps	*Number of Game Cycles*
FourBasesWorkers8x8	3000
TwoBasesBarracks16x16	4000
NoWhereToRun9x8	3000
DoubleGame24x24	5000
basesWorkers8x8A	3000
basesWorkers16x16A	4000
BWDistantResources32x32	6000
(4)BloodBath.scmB	8000

atomic category. This relegated the decision of unit production to the adversarial planner. If each **Produce** action was given different atomic categories, then Lex_{Learn} would embed build information directly into the generated lexicon.

Our training dataset consists of plan traces derived from replay data of μRTS games. Specifically, we generated replay data by running a five-iteration Round Robin tournament on each of the open maps from the CIG 2018 μRTS tournament, shown in Table III with agents *POWorkerRush*, *POLightRush*, *PVAIML_ED*, and *StrategyTactics* [20], where each agent played as Player 1 and Player 2. Games on each map were limited to the number of game cycles stated in Table III. Next, we used the replay data to generate a set of 50,000 training instances, pruning any **Move** actions as we do not wish to learn any movement actions. While there were more training instances that could be generated (three action sequences with nine possible actions would require at minimum 729 instances to cover all possible permutations of three action sequences), we believe that 50,000 instances was enough for training. Each training instance corresponds to a 3-action behavior employed by the agents. The sequences of actions were limited to three to allow our agent to plan within the 100ms time limit as per the CIG 2018 tournament rules.

The adversarial CCG planner has three tunable parameters. The first parameter is the constant for UCB1, which was set to 20. The second parameter is the number of searched action category decompositions, N, which was set to 5. The third and final parameter is the maximum depth, which we set to 6 as that is the maximum number of actions that could be planned by both the max and min player in our adversarial planning search. These parameters were set to get results for the paper, but will be optimized for the competition.

We tested μCCG against six baseline agents: *RandomBiased*, *POWorkerRush*, *POLightRush*, *NaiveMCTS*, *StrategyTactics* [20], and *PVAIML_ED* using the eight open maps from the CIG 2018 μRTS tournament provided in Table III. We ran a five-iteration Round-Robin tournament where each agent played as both Player 1 and Player 2. Our experiments used all of the rules stated in the CIG 2018 tournament, except that we gave our agent a 30ms extra grace period per game frame to produce an action, since the purpose of these experiments was just to compare the agents.

Table I provides "Wins-Losses-Ties" and Win ratio (# wins + 0.5 $\times$ # ties) from our five-iteration Round-Robin tournament. Overall, our agent placed second-to-last in terms of Win ratio. Looking at Table II, which provides per-map results for μCCG, we were able to easily beat the RandomBiased agent, even winning more games than PVAIMIL_ED and NaiveMCTS. Additionally, we see that against the POWork-

erRush, RandomBiasedAI, and PVAIML_ED agents, μCCG didn't lose a single match on the NoWhereToRun9x8 map.

μCCG did come close to outperforming NaiveMCTS, with a win-loss difference of two. Table II indicates that for the first four maps, NaiveMCTS significantly outperformed our agent. However, for the last three maps μCCG outperformed NaiveMCTS. Specifically for the last two maps, μCCG never lost a single match against NaiveMCTS. Additionally, μCCG was able to win a few games against the two top performing agents in the CIG 2017 μRTS competition, StrategyTactics, and POLightRush. We believe that with an improved parameter policy, μCCG could potentially win more games against these agents.

We believe that μCCG may have won on the last two maps against NaiveMCTS and RandomBiased because the maps were relatively large with (4)BloodBath.scmB being the largest map in the set at 64x64. As the size of the map gets larger and the number of units increases, NaiveMCTS has to search a larger search space. We believe that this search space explosion resulted in NaiveMCTS losing games against μCCG. We believe that the ties between μCCG and NaiveMCTS in BWDistantResources32x32 may have been due to the game reaching the maximum number of cycles. μCCG may have destroyed most of NaiveMCTS' units, but as a result caused NaiveMCTS to start playing optimally because the state space decreased.

We believe most of our losses were due to a few factors related to the parameter policy and planning. First, we speculate that μCCG may have delayed constructing barracks and offensive units because we only allowed a single unit to construct units at a time. Thus, if μCCG was creating workers, it wouldn't be able to construct barracks or any offensive units. Second, we believe that not allowing workers to attack enemy units may have caused μCCG to lose on small maps. On small maps, μCCG would not have time to construct offensive units and a group of offensive workers would immediately overwhelm us. Third and finally, we believe that coupling the attack action with build order planning may have stopped offensive units from attacking. Offensive units could only attack if the attack action was administered by the agent. If the planner didn't generate an attack action, all offensive units would stop attacking (even mid-assault on the enemy).

Although μCCG still does not outperform state of the art bots, our experiments show that the idea of using an adversarial CCG planner with a learned CCG lexicon generated from a domain-independent CCG lexicon learning algorithm is viable for RTS games. As part of our future work before the 2018 competition, we would like to optimize our parameter policy, as well as the training set and planning algorithms to maximize game-play performance, which was not a priority at this point.

VI. Related Work

There are several areas of research that are closely related to our work: 1) RTS game-playing agents, 2) Adversarial Planning, and 3) Plan Learning. There is a plethora of work in the scientific literature on creating agents to play RTS games such as Starcraft and μRTS. Synnaeve and Bessière present BroodwarBotQ which uses a Bayesian model for unit control in Starcraft [21]. Uriarte presents Nova, a Starcraft agent that combines several techniques used to solve different AI problems [22]. Churchill and Buro present UAlbertaBot which optimizes build order planning using action abstractions and heuristics [23]. Other Starcraft agents include Skynet and Berkeley's Overmind [24].

There is also a large amount of prior research on adversarial planning, but we state a few here. Stanescu *et al.* present an approach to hierarchical adversarial search motivated by the chain of command employed by the military. Specifically, they employ game tree search on two layers of plan abstractions [25]. Willmott *et al.* [26] presents GoBI, an adversarial HTN planner for the game of Go that uses α-β search with a standard HTN planner. α uses the HTN planner to generate an action, and passes the game state to β to generate their action while attempting to force α to backtrack its search. In recent years, Ontañón and Buro used Hierarchical Task Networks (HTNs) [13] and minimax to adversarially plan against an opponent in RTS games. This work builds off this, but uses CCGs instead of HTNs for planning, and learns the plan representation instead of hand-authoring one.

The last ares of related research is plan learning. Hogg *et al.* present HTN-Maker which learn HTN methods from analyzing the state of the world before and after a given sequence of actions [27]. Zhuo *et al.* [28] present HTNLearn which builds an HTN from partially-observable plan traces. Nejati *et al.* [29] present a technique for learning a specialized class of HTNs from expert traces. Finally, Li *et al.* [30] present a learning algorithm that successfully learns probabilistic HTNs using techniques from probabilistic Context-Free Grammar induction. The two main differences of our work is that we learn a plan CCG representation and we learn this for an RTS domain.

VII. Conclusion

This paper presents initial work on a CCG-based game playing agent for μRTS called μCCG. This paper provides two main contributions. First, we presented an alternative hierarchical planning formulation based on CCGs. Second, we show that we can learn a CCG plan representation from sequences of actions collected from game replay data. We also provided initial results of μCCG against against other μRTS agents. Our results seem promising and demonstrate that μCCG can use a learned representation generated by a domain-independent learning algorithm to play against other agents. We are currently in the process of improving the agent for the CIG 2018 μRTS tournament, specifically the parameter policy.

There are a few directions for future work. First, we want to look into interweaving other RTS problems such as terrain analysis and resource management into the planner to improve the planning process. Second, we want to look into improving hierarchical learning of CCGs by incorporating RTS domain knowledge into the learning process as Lex_{Learn}

is a general CCG plan learning algorithm. Third, once our agent is able compete against other μRTS agents, we want to apply our CCG adversarial planner to the commercial RTS game, Starcraft. Fourth, we trained Lex_{Learn} on sequences of three actions due to the time constraint defined in the μRTS tournament rules, but plans can be larger than three actions. Thus, we want to learn from larger sequences of actions. Finally, for the current version of our planner, we relaxed the temporal restrictions of CCG lexicons in order to accommodate parallel actions. However, we would like to investigate this issued further and design a general framework to deal with concurrent actions in the context of CCGs.

REFERENCES

[1] M. Buro, "Real-time strategy gaines: A new ai research challenge," in *Proceedings International Joint Conference in Artifical Intelligence*, 2003, p. 15341535.

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

[3] C. W. Geib, "Delaying commitment in plan recognition using combinatory categorial grammars," in *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, ser. IJCAI'09. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 1702–1707. [Online]. Available: http://dl.acm.org/citation.cfm?id=1661445.1661719

[4] C. W. Geib and R. P. Goldman, "Recognizing plans with loops represented in a lexicalized grammar," in *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, ser. AAAI'11. Palo Alto, California, USA: AAAI Press, 2011, pp. 958–963. [Online]. Available: http://dl.acm.org/citation.cfm?id=2900423.2900576

[5] C. F. Schmidt, N. Sridharan, and J. L. Goodson, "The plan recognition problem: An intersection of psychology and artificial intelligence," *Artificial Intelligence*, vol. 11, no. 1-2, pp. 45–83, 1978.

[6] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A survey of real-time strategy game AI research and competition in StarCraft," *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG)*, vol. 5, pp. 1–19, 2013.

[7] R.-K. Balla and A. Fern, "UCT for tactical assault planning in real-time strategy games," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 2009, pp. 40–45.

[8] N. Justesen, B. Tillman, J. Togelius, and S. Risi, "Script-and cluster-based UCT for StarCraft," in *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*. IEEE, 2014, pp. 1–8.

[9] A. Uriarte and S. Ontañón, "Game-tree search over high-level game states in RTS games," in *Proceedings of the AAAI Artificial Intelligence and Interactive Digital Entertainment conference (AIIDE 2014)*, 2014.

[10] D. Churchill and M. Buro, "Portfolio greedy search and simulation for large-scale combat in StarCraft," in *IEEE Conference on Computational Intelligence in Games (CIG 2013)*, 2013, pp. 1–8.

[11] M. Chung, M. Buro, and J. Schaeffer, "Monte carlo planning in RTS games," in *Proceedings of the IEEE Conference on Computational Intelligence in Games (CIG 2005)*, 2005.

[12] S. Ontanón, "Combinatorial multi-armed bandits for real-time strategy games," *Journal of Artificial Intelligence Research*, vol. 58, pp. 665–702, 2017.

[13] S. Ontanón and M. Buro, "Adversarial hierarchical-task network planning for complex real-time games," in *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[14] K. Erol, J. Hendler, and D. S. Nau, "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)*, 1994, pp. 249–254.

[15] C. Geib and P. Kantharaju, "Learning combinatory categorial grammars for plan recognition," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, 2017.

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

[17] C. W. Geib, "Lexicalized reasoning about actions," *Advances in Cognitive Systems*, vol. Volume 4, pp. 187–206, 2016.

[18] L. S. Zettlemoyer and M. Collins, "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*, ser. UAI'05. AUAI Press, 2005, pp. 658–666.

[19] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[20] N. A. Barriga, M. Stanescu, and M. Buro, "Combining strategic learning and tactical search in real-time strategy games," in *Proceedings of the Thirteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-17)*, 2017, pp. 9–15.

[21] G. Synnaeve and P. Bessi?e, "A bayesian model for rts units control applied to starcraft," in *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, Aug 2011, pp. 190–196.

[22] A. Uriarte, "Multi-reactive planning for real-time strategy games," Master's thesis, Universitat Autònoma de Barcelona, 01 2011.

[23] D. Churchill and M. Buro, "Build order optimization in starcraft." in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2011, pp. 14–19.

[24] M. Buro and D. Churchill, "Real-time strategy game competitions," *AI Magazine*, vol. 33, no. 3, p. 106, 2012.

[25] M. Stanescu, N. Barriga, and M. Buro, "Hierarchical adversarial search applied to real-time strategy games," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2014.

[26] S. Willmott, J. Richardson, A. Bundy, and J. Levine, "An adversarial planning approach to go," in *International Conference on Computers and Games*. Springer, 1998, pp. 93–112.

[27] C. Hogg, H. Muñoz Avila, and U. Kuter, "Htn-maker: Learning htns with minimal additional knowledge engineering required," in *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, ser. AAAI'08. AAAI Press, 2008, pp. 950–956. [Online]. Available: http://dl.acm.org/citation.cfm?id=1620163.1620220

[28] H. H. Zhuo, H. Muñoz-Avila, and Q. Yang, "Learning hierarchical task network domains from partially observed plan traces," *Artificial Intelligence*, vol. 212, pp. 134 – 157, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370214000447

[29] N. Nejati, P. Langley, and T. Konik, "Learning hierarchical task networks by observation," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 665–672.

[30] N. Li, W. Cushing, S. Kambhampati, and S. Yoon, "Learning probabilistic hierarchical task networks as probabilistic context-free grammars to capture user preferences," *ACM Trans. Intell. Syst. Technol.*, vol. 5, no. 2, pp. 29:1–29:32, Apr. 2014. [Online]. Available: http://doi.acm.org/10.1145/2589481