Combining Monte Carlo Tree Search and Apprenticeship Learning for Capture the Flag

Jayden Ivanovic, William L. Raffe, Fabio Zambetta, and Xiaodong Li
School of Computer Science and IT, RMIT University
Melbourne (Australia)

Email: s3331571@student.rmit.edu.au, william.raffe@rmit.edu.au, fabio.zambetta@rmit.edu.au, xiaodong.li@rmit.edu.au

Abstract—In this paper we introduce a novel approach to agent control in competitive video games which combines Monte Carlo Tree Search (MCTS) and Apprenticeship Learning (AL). More specifically, an opponent model created through AL is used during the expansion phase of the Upper Confidence Bounds for Trees (UCT) variant of MCTS. We show how this approach can be applied to a game of Capture the Flag (CTF), an environment which is both non-deterministic and partially observable. The performance gain of a controller utilizing an opponent model learned via AL when compared to a controller using just UCT is shown both with win/loss ratios and True Skill rankings. Additionally, we build on previous findings by providing evidence of a bias towards a particular style of play in the AI Sandbox CTF environment. We believe that the approach highlighted here can be extended to a wider range of games other than just CTF.

I. INTRODUCTION

Computer games players often argue that the AI presented to them as companions or opponents are ill-advised and short-sighted. An agent that could observe the player and later adapt its strategy to counter that player would offer a vast improvement in gameplay and potentially offer better entertainment value [1]. A player facing such an agent would be forced to adapt their strategies rather than sticking to repetitive and ultimately predictable ones that exploit the agent’s weaknesses. This would lead to a more engaging player experience with a game tailored towards the individual. Additionally, one can imagine simulations being created for tactics planning in regards to military and sports.

We propose to solve this opponent modeling problem through Apprenticeship Learning (AL) [2], an algorithm that learns from demonstrations given by a domain expert to perform like that expert. This can alternatively be seen as learning a policy that mimics the behavior of an opponent agent, whether that opponent is human or another controller. Once the opponent policy is learnt, it is integrated into the Upper Confidence Bound for Trees (UCT) [3] implementation of Monte Carlo Tree Search (MCTS) [4], producing a method that we refer to as UCT+AL. In this way, the opponent model created by AL is used to bias action selection during the expansion phase of the UCT search, therefore considering the opponent’s policy when searching for a winning policy.

This approach is intriguing as it is general and can simply be applied to other games without a large amount of domain specific knowledge. All that is needed are samples of gameplay of the opponent which is to be modeled. We argue that this is a natural approach to win competitive games in the real world. Take for example a sporting team that studies plays made by their opponent to plan a strategy before their encounter. In this way, an agent would be able to adapt its play style in order to counter that of its adversaries, thus offering the opponent a personalized challenge.

Although MCTS has seen much success in deterministic games such as GO [5], there is still a lot of work that needs to be done in regards to non-deterministic environments. Browne et al. [4] call for research in MCTS applying opponent modeling techniques in non-deterministic environments, a current area which they believe holds much promise in future works. Ponsen et al. [6] follow a similar approach to ours by applying Bayesian techniques to learn an opponent model for poker. From this they witnessed an improvement in performance but, unfortunately, this method is not trivial to transfer to other domains.

In this paper, the test bed domain is that of the capture the flag (CTF) module in the AI Sandbox [1], which provides a game state that is both non-deterministic and partially observable. Within this module, one writes a controller for a commander which orders a team of non-player characters (bots) to perform actions. In our previous work [7], we applied Reinforcement Learning (RL) to the AI Sandbox CTF environment. There we witnessed promising performance from the RL controller, yet it still struggled to be competitive against the scripted commanders in the test bed. The RL commander only performed well on a small number of maps and this is believed to have occurred due to the need to handcraft both the state space and the reward function. This leads to the present work in which the reward function is learned autonomously and separately for each opponent rather than being specified by a designer.

In summary, this paper contributes a hybridized implementation of MCTS and AL, demonstrating it as a general and extensible method for creating adaptive agents in a variety of games. To the best of our knowledge, ours is the first attempt to use AL to perform opponent modeling in any form of MCTS, be it in deterministic or non-deterministic games. As an aside, we also add evidence to the existence of a strong bias in strategies in the AI Sandbox CTF module, reinforcing our past findings.

The remainder of this paper is organised as follows: In section II we provide some background concepts necessary for detailing the UCT+AL approach provided in Section III. In Section IV we provide our key experimental results to then

1The AI Sandbox, created by AIGameDev.com: http://aisandbox.com/
conclude our paper in Section V by providing avenues for future work.

II. BACKGROUND

In this section we provide background knowledge on the foundational algorithms used in this paper. Namely, we discuss Apprenticeship Learning (using Least-Squares Temporal Difference Learning—\(\mu\)) and Monte Carlo Tree Search (using Upper Confidence Bounds for Trees). We also give a brief introduction to the AI Sandbox: Capture the Flag environment that was used as a test bed.

A. Apprenticeship Learning

The traditional Reinforcement Learning (RL) problem, as formulated by Sutton and Barto [8], operates on a Markov Decision Process (MDP) defined by a set of states \(S\), a set of actions \(A\), probabilities of transitioning from each state to another given an action \(P_a(s, s')\), and a reward function for transitioning from one state to another with a given action \(R_a(s, s')\). With this information, an RL algorithm attempts to find an optimal policy \(\pi^*\) that maximizes the reward while transitioning through the states of the MDP, typically done by iteratively taking actions, observing the reward, and updating the policy.

However, when learning to perform a new task, such as playing a board game, it is often easier and more intuitive to learn via watching demonstrations of play from an expert. In this case, the reward function \(R_a(s, s')\) that the expert is following is likely to be unknown; they are following a strategy that is known only to them. This then defines the Inverse Reinforcement Learning (IRL) [9] problem, whereby recorded trajectories of an expert’s actions in state sequences are given and the reward function must be approximated.

Abbeel and Ng [2] propose that a reward function learned via IRL can then be used to attain a policy that achieves performance similar to that of the expert. This process is known as Apprenticeship Learning (AL) and all that it requires are trajectories of the expert. In IRL, it is assumed that there exists a reward function that is expressed as:

\[
R^*(s) = w^* \phi(s)
\]

where \(\phi\) is a function that produces a vector of features given the state of the environment \(s\) and \(w^*\) specifies the relative weights (trade-offs) between features. For example, Abbeel and Ng applied AL to driving a car, and defined \(\phi\) to produce a vector of features including collisions, off-road, left lane, middle lane, right lane, etc. Different drivers are likely to make trade-offs between these features depending on their own driving style. Therefore, if learning a policy for a safe driver, obviously driving off-road will have a lower weight.

Once the reward function is known, a policy can be generated to approximate that of the expert. In this paper we use a variant of AL proposed by Klein et al. [10] that utilizes Least-Squares Temporal Difference (LSTD) [11] and is known as LSTD-\(\mu\). LSTD-\(\mu\) offers the advantages of being batch, off-policy, and model-free. By treating each component \(i\) of the feature expectation \(\mu^i(s)\) as a value function of an MDP, using Equation 2, LSTD-\(\mu\) performs as similar to the original AL [2].

\[
\mu^i_t(s) = E\left[ \sum_{t=0}^{\infty} \gamma^t \phi_i(s_t) | s_0 = s, \pi \right]
\]

Here, \(\pi\) is the current sub-optimal policy being evaluated, \(s_t\) is the current state at time \(t\), \(s_0\) is the starting state, and \(\gamma\) is a constant discount factor between [0, 1]. In LSTD-\(\mu\), a transition probability function of the MDP does not need to be modeled, which in itself would be quite a complex task in all but the most trivial domains. Instead, the Least Square Policy Iteration [12] algorithm is used to solve the MDP without running simulations of the game. Such AL has seen success in games such as Super Mario [13] and Unreal Tournament [14] where agents were shown to achieve performance similar to their human counter-part.

B. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is an anytime, heuristic-free, asymmetric algorithm which randomly samples actions in the decision space to build a search tree. For this work, we adopt the terminology set by [4] where Upper Confidence Bound for Trees (UCT) refers to MCTS using an Upper Confidence Bound (UCB1) tree selection policy.

The algorithm consists of four mains steps that are repeated for a given number of iterations or until some time constraint has been reached. These are selection, expansion, rollout, and backpropagation. Note that the rollout phase is sometimes referred to as the simulation phase, however here we use the term ‘simulation’ to refer to the entire MCTS process as all the earlier mentioned phases are simulated outside of our test bed CTF game, which is discussed further in Section III-A4.

During the selection phase, a policy is followed through the nodes in the tree that have already been expanded (i.e. nodes that have previously been visited). In UCT, during the selection phase, a child node is chosen greedily where the value of each previously expanded child node \(j\) is:

\[
\tilde{X}_j + 2C_p \sqrt{\frac{2 \ln n}{n_j}}
\]

where \(\tilde{X}_j\) is the mean reward experienced after following child node \(j\), \(n_j\) is the number of times the child node \(j\) has been visited, \(n\) is the total number of the times that the current (parent) node has been visited, and \(C_p > 0\) is a constant that controls exploration.

Once a node is reached with one or more unexpanded child nodes, one of those child nodes is chosen at random for expansion. From this newly expanded node, the rollout phase sees a random policy followed until the end of an episode or a depth limit is reached. Any reward that is encountered is then backpropagated to update the mean reward value of the newly expanded node and all of its parent nodes that were followed during the selection phase.

MCTS has gained much attention since its conception in 2006. The use of Monte Carlo evaluations in a tree-based
search was shown in [15]. This agent (Crazy Stone) was able to successfully win the 10th KGS (then known as the ‘Kiseido Go Server’) computer Go tournament on a 9 x 9 board. Kocsis and Szepesvari [3] applied UCB to tree-based search resulting in the UCT algorithm. They show that UCT is significantly more efficient compared to several alternative algorithms. Additionally, they highlight that the probability of selecting the optimal action can be made to converge to 1 as the number of samples grows to infinity. From this work, Gelly et al. [16] apply UCT to their own Go agent, MoGo, the first computer Go program to use UCT. Here, they show strong play on both 9x9 and 13x13 boards. Additionally, they begin to incorporate domain knowledge into the random rollouts to select more reasonable moves, as well as exploring parallelization and pruning of the tree.

Chaslot et al. [17] describe MCTS as a general framework for game AI, including domains such as classic board games, modern board games, and video games. For example, Go, Settlers of Catan, and general RTS games respectively. However, Chaslot et al. [18] also show MCTS to be applicable to general domains such as production management problems. Finnsson [19] highlights the strength of MCTS in a variety of problem domains via their agent CADIA-Player which successfully won the Third Annual General Game Playing (GGP) Competition given only the rules of the game. A small subset of the games in this competition were Bomberman, Pac Man, Connect Four, and Asteroids to name a few.

III. UCT+AL FOR CAPTURE THE FLAG

In this section we discuss how UCT is integrated into the AI Sandbox and then how the AL opponent model is integrated into the UCT algorithm, producing the UCT+AL algorithm for the CTF game. Firstly, the details of the state and action representations for a UCT commander in CTF are presented. Then we demonstrate how an opponent’s trajectory is expressed as state-action pairs and how this can be used in conjunction with AL to generate an opponent model.

Figure 2 shows how the various components of the UCT+AL solution work together. The AI Sandbox is used to log trajectories of the opponent in the CTF environment. Next, these trajectories are given to the LSTD-μ variation of the AL algorithm to generate a policy that is representative of the opponent (i.e. an opponent model). Next, this policy is used to weight the choice of actions taken by UCT during the expansion phase. Finally, we play the game against the original opponent using the policy generated by UCT.

C. Capture the Flag

Capture the Flag (CTF) is a highly competitive and popular game mode found in many first-person shooter games. In CTF there are two teams that both attempt to capture the opposing team’s flag whilst defending their own. A flag capture is the sequence of picking up the flag from its spawn location and returning it to the teams scoring location. During the course of the game, teams are likely to confront one another and engage in combat. If a player’s health is fully depleted in combat, they die and are removed from the game until the next respawn timer elapses. At the end of the allotted game time (five minutes in our framework), the team with the most flags captured is considered the winner.
A. UCT in CTF

In this section we first proceed to provide details of the main components in any MDP problem (actions, states, and reward function) to then discuss simulating games in UCT and the trade-offs in doing that.

1) State Representation: The CTF map is subdivided into areas as in Figure 3 and within these we record the number of friendly and enemy bots. A typical player of an RTS is likely to subdivide maps in a much similar way and send units to locations, rather than moving them using very fine-grained control.

For example, in Figure 3 we see seven bots each in the red and blue spawn areas. The string which corresponds to this state can be seen in Table I. The first character in each line of the state string represents the beginning of different information sets; ‘T’ denotes team data, ‘E’ for enemy data, and ‘F’ for features. The simple map representation captures much of the information that a commander needs to be successful. However, the last line is the feature vector that results from the feature function \( \phi \) that is used during AL, the structure of which can be seen in Table II.

2) Action Representation: The five actions available for a commander to issue to a bot through the AI Sandbox interface are ‘Attack’, ‘Charge’, ‘Move’, ‘Defend’, and ‘Null’. In ‘Attack’ mode, a bot will move slowly to a given location and attack any enemy it sees. ‘Charge’ works similarly but the bot will run faster and fire slower when it sees an enemy. The ‘Move’ command tells a bot to run to a given location and to not engage any enemies along the way. ‘Defend’ orders a bot to hold its current position with a higher rate of fire but a smaller field of vision. The ‘Null’ command simply instructs a bot to continue to carry out the previous command.

We allow our commander to issue these actions with the map areas shown in Figure 3 as the target position. In the case of the ‘Defend’ command we simply defend the current position; that is, the UCT algorithm directs a bot to a particular area and then instructs the bot to defend.

As there are multiple bots per team, the action that a commander takes at each game tick is a combination of these individual orders (primitive actions) to each bot. For example, Table III shows the actions taken by a commander for a number of time steps.

3) Reward: The reward that is backpropagated through the UCT game tree after a rollout has completed can take one of the following values: Win = 1.0, Draw = 0.5, or Loss = 0.0. Here, a win is given to the UCT commander if it has captured the most flags at the end of the rollout phase, a draw if both teams have the same number of flags, and a loss if this commander has a smaller number of captured flags.

4) Simulation Granularity: For the most part, we made use of an existing implementation of the UCT algorithm\(^2\). Instead, the main changes that were made were to how the UCT simulations were conducted. Due to technical constraints, the CTF game state itself was not suitable for use, even though it would have given the most accurate simulation. Instead, coarser game data needed in order to run simulations is extracted from the current CTF game state and the simulation is run externally. This was primarily data of each bot, the status of the flags, and information about the map. We then updated this information during the simulation by following the game rules in the AI Sandbox CTF module. Although this approach isn’t as accurate, a similar approach has proven to be successful in the award winning game Total War: Rome II\(^3\).

Each round of the game is of five minutes duration. It is important to note that this can be accelerated but at the cost of information granularity. Therefore, we had to balance the total number of steps allowed during the rollout phase, and the speed at which the game was played. As it stands, the game plays in pseudo real-time. Therefore, each time the UCT algorithm runs, the game effectively pauses until a decision is made as to the next action the commander will execute. Additionally, the commander does not run the UCT algorithm at every game tick, but rather, every 3 seconds. We did this in

\[^2\]The MCTS research hub: http://mcts.ai/code/python.html

\[^3\]Monte-Carlo Tree Search in TOTAL WAR: ROME II’s Campaign AI: http://aigamedev.com/open/coverage/mcts-rome-ii/
TABLE IV. AN EXAMPLE OF LOGGED TRAJECTORIES. THE FIRST LINE IS THE ELAPSED GAME TIME, THE SECOND LINE IS THE BOT ACTIONS THAT THE COMMANDER CONTROLLER HAS ISSUED, AND THE THIRD THROUGH FIFTH LINES ARE THE STATE INFORMATION FROM TABLE I.

<table>
<thead>
<tr>
<th>Elapsed Game Time</th>
<th>Bot Actions</th>
<th>State Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>251.90003138s</td>
<td>C:0:C:0:C:0</td>
<td>F:0:0:0:0:0:0:0</td>
</tr>
<tr>
<td>263.63347853s</td>
<td>C:1:C:0:C:0</td>
<td>F:0:1:0:0:0:0:0</td>
</tr>
<tr>
<td>265.100013826s</td>
<td>C:0:C:1:C:5</td>
<td>F:1:0:0:0:0:0:0</td>
</tr>
</tbody>
</table>

order to allow the commander to execute the selected action over a number of time steps and due to the fact that the game state is rarely likely to change drastically between a short series of game ticks. If the commander issued orders at very short time, the commander’s bots would have only begun executing their orders by the time the next UCT action was selected. Therefore, this approach allows consequences to be realized and the bots to actually perform their orders before the commander selects the next action to take.

B. Opponent Modeling via AL

To learn a policy for the opponent, we must provide the AL algorithm with a log of trajectories; that is, demonstrations of the opponent’s behavior. To generate this information, a game of CTF is run using an opponent controller. All of the experiments in this paper are run against scripted controllers that come with the AI Sandbox CTF distribution. In the context of the original AL description, these controllers are considered as the experts of domain.

A trajectory is built with the following information, taking an approach similar to that of Lee et al. [13]: the time since the beginning of the game, the action taken, the state the commander was in, and the current features of \( \phi \). The trajectories are processed to discard those which do not differ between time steps as we can consider that no new state has been observed and that the commander is still executing a single action.

An example of an opponent’s trajectories can be seen in Table IV. The first line is the elapsed game time. The second line (denoted by ‘C’) indicates the actions the commander controller has issued. The next three lines are the state information found in Table I. From the recorded trajectories, the state-action-state transitions \((s, a, s')\) required by LSTD-\( \mu \) are generated. At run-time, when the reward witnessed in the transition is required, we calculate it via the feature function \( \phi \) and weight vector \( w \) to give us a tuple of the form \((s, a, r, s')\).

These transitions are passed to the LSTD-\( \mu \) algorithm and from this a policy is generated that allows us to determine what moves the opponent will take in its current state. We then store the policy and repeat for the other opponent controllers provided with the CTF test bed. When the UCT+AL commander controller encounters a particular opponent, the corresponding opponent model is read in. This opponent model is used to weight the choice of actions the UCT+AL commander can take during the expansion phases of the tree search. This is in contrast to typical UCT where all non-expanded child nodes of the current node have an equal probability of being selected for expansion. Additionally, if it is known from the model that an opponent never plays a particular move, that branch can be pruned from the tree with a weight of 0.0.

IV. Experimental Results

The AI Sandbox has a large selection of maps available. Additionally, the number of bots on each team is configurable and can differ between maps. For our experiments however, we only use Map 50 (seen earlier in Figure 1) and four bots per team. We do this to focus our preliminary work on a more confined problem. This number of bots was chosen as it still allows the commanders to perform their strategies effectively and minimizes the branching factor which would only scale exponentially as additional bots are added. Despite this, the branching factor of the tree remains orders of magnitudes larger than Go.

There are four scripted commander controllers provided with the AI Sandbox. We use these controllers both as baselines and also as opponents for the UCT+AL controller to optimize for. These scripted controllers are:

- Greedy - Sends every bot to the enemy flag in an attempt to overwhelm them. Sacrifices all defense for a strong offense.
- Defender - One bot is sent to the opposition’s flag whilst the others remain around their flag defending it. Focuses on preventing the opposition from picking up the flag.
- Random - Every bot is sent to some arbitrary position on the map or within some distance of either team’s flag or goal. This behavior results in a team which both randomly patrols areas of the map and attempts to capture enemy flags.
- Balanced - This commander assigns one bot to attack, one to defend and the others to randomly patrol. It is somewhat of a combination of all the other commander types.

Each of these controllers is made to play 100 games against every other scripted controller. This is done to understand which controllers are competitive on this map and to establish their relative performance. For a more in-depth analysis of these scripted controllers across numerous maps, please refer to our previous work [7].

With the baselines established, we first experiment with UCT alone, without the AL opponent model. The UCT commander plays 30 games against each of the scripted commanders with the iterations of the simulation phase set to 1000. We then repeat this but where the only actions available to the UCT commander are ‘Attack’ and ‘Charge’. This is done to exploit the apparent bias in the CTF game towards greedy options that we found previously [7]. We then repeat this yet again but with the UCT+AL commander controller, though due to the added computation costs of AL we only perform
TABLE V. A CROSS COMPARISON OF THE SCRIPTED COMMANDERS ON MAP 50. EACH COMMANDER PLAYED AGAINST THE OTHER FOR 100 GAMES. THE RESULTS ARE FROM THE PERSPECTIVE OF THE FIRST COMMANDER IN THE ROW. THE FOUR COMMANDERS ARE: GRE = GREEDY, DEF = DEFENDER, RAND = RANDOM, BAL = BALANCED.

<table>
<thead>
<tr>
<th>Versus</th>
<th>Wins</th>
<th>Draws</th>
<th>Losses</th>
<th>Flags Captured</th>
<th>Flags Conceded</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gre vs Def</td>
<td>90</td>
<td>4</td>
<td>6</td>
<td>346</td>
<td>99</td>
</tr>
<tr>
<td>Gre vs Ran</td>
<td>99</td>
<td>1</td>
<td>0</td>
<td>555</td>
<td>38</td>
</tr>
<tr>
<td>Gre vs Bal</td>
<td>95</td>
<td>3</td>
<td>2</td>
<td>447</td>
<td>78</td>
</tr>
<tr>
<td>Ran vs Bal</td>
<td>23</td>
<td>23</td>
<td>54</td>
<td>78</td>
<td>153</td>
</tr>
<tr>
<td>Ran vs Def</td>
<td>3</td>
<td>4</td>
<td>93</td>
<td>51</td>
<td>346</td>
</tr>
<tr>
<td>Def vs Bal</td>
<td>90</td>
<td>9</td>
<td>1</td>
<td>328</td>
<td>46</td>
</tr>
</tbody>
</table>

this against the two strongest scripted commanders, namely the Greedy and Defender commanders.

Finally, we rank all the variations of UCT and the baseline commanders according to TrueSkill [20]. Due to the earlier mentioned experiments there are many games played. Therefore, the rating given by TrueSkill is a strong estimate of the relative performance of the commanders. Browne et al. [4] suggest that ELO is a suitable metric for comparing the performance of different MCTS variations. However, TrueSkill has superseded ELO as it addresses issues of games where more than two players exist in a team based setting, can be used in a variety of settings, and provides an accurate estimate of a player’s rating after only a few games [20].

A. Scripted Commander Results

Table V shows the results of running our baseline experiment. Immediately one can see the strength of the Greedy commander who wins consistently against the others and has a much greater number of flags captured. The Defender commander is the second most competitive and wins consistently against the Balanced and Random commanders. Lastly, the Balanced commander outperforms the Random commander. From these results we can see that both the Greedy and Defender commander, who utilize two contrasting but extreme strategies, are the most competent on this particular map. This also highlights the fact that being aggressive is the most optimal form of play in this particular CTF environment.

B. UCT, Reduced UCT, and UCT+AL Results

Firstly, we see from Figure 4 that UCT performs well against the Random commander by the large number of games won. When competing against the Balanced commander performance is poor but the UCT commander still wins some games. On the other hand, it fails to do well against Greedy or Defender. We believe the reason for this is that, once again, the extreme strategies are more optimal and that given the large state space, one cannot learn to take a counter strategy via Monte Carlo steps in the rollout phase in such a short simulation time.

When the action set available to the UCT commander is reduced to simply ‘Attack’ and ‘Charge’, the results in Figure 5 are produced. Here we witness a large improvement in performance compared to Figure 4. The Random commander and Balanced commander are both defeated easily a number of times. Greedy and Defender still remain strong competitors, winning the majority of games. However, this variation performs substantially better than before. These results further demonstrate the bias which exists in the environment for the greedier actions.

Figure 6 shows the results of the UCT+AL commander against the top performing scripted commanders; the Greedy and Defender controllers. Although it is not as strong as UCT with the reduced action set, it still achieves better performance than UCT with the full set of actions. To clarify, the UCT+AL variant was given the full action set. Therefore, we believe the results are quite positive in this respect. If we compare Figure 6 to Figure 4, standard UCT has a substantially higher percentage of losses. With the opponent model, our variation is able to perform much better than standard UCT and starts to approach performance of UCT with the reduced action set.

C. TrueSkill Rankings

Table VI shows a leader-board of each UCT variation and the scripted commanders as ranked by TrueSkill. The greater the difference between the two players $\mu$ values, the greater the chance there is of the player with the highest $\mu$ value to win the game. From the original description of TrueSkill [20], if one player’s $\mu$ is greater than another by 4.16 then there is a greater than 75.6% chance of the first player winning a match.

Therefore, UCT+AL and Reduced UCT (i.e. with a reduced
D. A Note on Rollout Iterations

In all of the results presented above, the number of iterations in the rollout phase of UCT was set to 1000. It is worth noting that we repeated these experiments with both 100 and 500 iterations as well. However, the results of these experiments (not shown here) indicated that in most cases there was an insignificant difference between the iteration settings. However, in some cases, allowing for 1000 rollout iterations actually performed worse when compared to 100 and 500 iterations and thus the results presented here are a worst case of all the experiments we conducted.

This is a surprising result as one would expect that the greater number of iterations would allow the agent to traverse to a greater depth in the tree. While we do not know for certain the reason to this result, it may be that the simulation that is run externally to the CTF game is too coarse and subsequent iterations in the rollout phase either provide no benefit or causes the search to fluctuate between optimal and sub-optimal policies. The extra time spent in the simulation is used only to realize branches in the tree which do not lead to a fruitful outcome. A key factor in this is the large branching factor which prevents exploration of the entire state space and in some games we may explore branches which are non-optimal.

In the UCT+AL case, the lack of difference between the iteration settings may be due to the opponent model and its influence over the action which is selected. Regardless, of the reasons, this result is fortuitously beneficial for many real-time games as the number of iterations needed for the rollout phases, and thus the time needed for each round of UCT optimization, can be significantly reduced.

V. Conclusion and Future Work

In this paper we demonstrated a novel approach for generating opponent models for MCTS using AL. We analyzed UCT, UCT with a reduced action set, and the algorithm presented here (UCT+AL), discussing the performance of each and the issues they face within the CTF environment. We showed that the learned opponent model can be used to improve on vanilla MCTS and partly reduce the impact of having a large branching factor by having knowledge of an opponent’s next move(s).

Standard UCT performed quite competitively against the Random and Balanced commanders. However, it struggled against the Greedy and Defender commanders. By reducing the total number of actions available to the agent to just the most aggressive actions, UCT’s performance improved substantially and became the strongest variant tested. Although the Greedy and Defender commander were still a challenge, we improved upon our previous work in [7] that used Reinforcement Learning.

The UCT+AL algorithm developed here can be used to improve on the performance witnessed by standard UCT. UCT+AL performed better than the standard UCT using the full set of actions. It is intuitive to hypothesize that UCT+AL performs better than UCT due to it having knowledge available in regards to the opponent’s next move. If one knows a number of the opponent’s next moves and can simulate a set of them, even if they can’t simulate the entire game rollout, they can still realize a more likely game scenario than through random sampling. However, when standard UCT used the reduced action set, it outperformed UCT+AL, albeit by a small margin. The similar performance here may be due to UCT+AL being
able to ignore the less optimal actions and choosing more appropriate ones from the full action set.

Finally, the results in this paper and our last [7] strongly suggest the presence of a bias towards the aggressive actions of the Greedy commander in the AI Sandbox CTF module. This may not be a fault of the AI Sandbox but instead the CTF game mode in general, as defending won’t earn points, only capturing the enemy’s flag will. Therefore, an agent could exhibit excellent play and defend for the majority of the game, only to make a mistake towards the end and have the opponent capture a single flag and win the game. We believe that UCT with the reduced action set would be able to overcome such strategies given sufficient exploration time and a stronger simulation.

VI. FUTURE WORK

The main focus of our future work is to investigate and further develop the UCT+AL approach to other games. The AI Sandbox CTF game provides an extensive test bed for modern gameplay but this bring additional challenges such as being partially observable, non-deterministic, producing a massive branching factor, and containing a strong bias towards certain actions. Considering these challenges, the UCT+AL solution performed relatively well, which hints at its potential strength in environments where such issues aren’t prevalent.

Additionally, earlier we discussed how we employed a rough approximation of the CTF game for the simulation of actions in UCT. We noted that even though this approximation was coarse, the commander was still able to perform adequately and quite impressively in a number of games. It would be intriguing to compare the effect of such a coarse simulation on the performance of both UCT and UCT+AL. For example, a game such as GO can allow for both exact simulations and coarse simulations and to compare the benefits and disadvantages of both. In doing so, we hope to identify properties and techniques that can be used where simulating the game in its entirety is too expensive. Alternatively, one may also look at how the large branching factor of UCT in a complex environment can be pruned more aggressively in order to select actions in real-time.

REFERENCES