UCT for Tactical Assault Planning in Real-Time Strategy Games

Radha-Krishna Balla and **Alan Fern**
School of EECS, Oregon State University
Corvallis, OR, 97331, USA
{balla, afern}@eecs.oregonstate.edu

Abstract

We consider the problem of tactical assault planning in real-time strategy games where a team of friendly agents must launch an assault on an enemy. This problem offers many challenges including a highly dynamic and uncertain environment, multiple agents, durative actions, numeric attributes, and different optimization objectives. While the dynamics of this problem are quite complex, it is often possible to provide or learn a coarse simulation-based model of a tactical domain, which makes Monte-Carlo planning an attractive approach. In this paper, we investigate the use of UCT, a recent Monte-Carlo planning algorithm for this problem. UCT has recently shown impressive successes in the area of games, particularly Go, but has not yet been considered in the context of multi-agent tactical planning. We discuss the challenges of adapting UCT to our domain and an implementation which allows for the optimization of user specified objective functions. We present an evaluation of our approach on a range of tactical assault problems with different objectives in the RTS game Wargus. The results indicate that our planner is able to generate superior plans compared to several baselines and a human player.

1 Introduction

Real-time strategy (RTS) games involve multiple teams acting in a real-time environment with the goal of gaining military or territorial superiority over one another. To achieve this goal, a player typically must address two key RTS sub-problems, resource production and tactical planning. In resource production, the player must produce (or gather) various raw materials, buildings, civilian and military units, to improve their economic and military power. In tactical planning, a player uses military units to gain territory and defeat enemy units. A game usually involves an initial period where players rapidly build their economy via resource production, followed by a period where those resources are exploited for offensive military assaults and defense. Thus, one of the keys to overall success is to form effective tactical assault plans, in order to most effectively exploit limited resources to optimize a battle objective.

In this paper, we focus on automated planning for the RTS tactical assault problem. In particular, the goal is to develop an action selection mechanism that can control groups of military units to conduct effective offensive assaults on a specified set of enemy forces. This type of assault is common after a player has built up forces and gathered information about where enemy troops are located. Here the effectiveness of an assault is measured by an objective function, perhaps specified by a user, which might ask the planner to minimize the time required to defeat the enemy or to destroy the enemy while maximizing the remaining health of friendly units at the end of the battle. Such a mechanism would be useful as a component for computer RTS opponents and as an interface option to human players, where a player need only specify the tactical assault objective rather than figure out how to best achieve it and then manually orchestrate the many low-level actions.

In addition to the practical utility of such a mechanism, RTS tactical assault problems are interesting from an AI planning perspective as they encompass a number of challenging issues. First, our tactical battle formulation involves temporal actions with numeric effects. Second, the problems typically involve the concurrent control of multiple military units. Third, performing well requires some amount of spatial-temporal reasoning. Fourth, due to the highly dynamic environment and inaccurate action models, partly due to the unpredictable enemy response, an online planning mechanism is required that can quickly respond to changing goals and unexpected situations. Finally, an effective planner should be able to deal with a variety of objective functions that measure the goodness of an assault.

The combination of the above challenges makes most state-of-the-art planners inapplicable to RTS tactical assault problems. Furthermore, there has been little work on specialized model-based planning mechanisms for this problem, with most games utilizing static script-based mechanisms. One exception, which has shown considerable promise, is the use of Monte-Carlo planning for tactical problem [Chung *et al.*, 2005; Sailer *et al.*, 2007]. While these approaches can be more flexible and successful than scripting, they are still constrained by the fact that they rely on domain-specific human knowledge, either in the form of a set

of human provided plans or a state evaluation function. It is often difficult to provide this knowledge, particularly when the set of run-time goals can change dynamically.

In this work, we take a step toward planning more flexible behavior, where the designer need not specify a set of plans or an evaluation function. Rather, we need only to provide the system with a set of simple abstract actions (e.g. join unit groups, group attack, etc.) which can be composed together to arrive at an exponentially large set of potential assault plans. In order to deal with this increased flexibility we draw on a recent Monte-Carlo planning technique, UCT [Kocsis and Szepesvari, 2006], which has shown impressive success in a variety of domains, most notably the game of Go [Gelly and Wang, 2006; Gelly and Silver, 2007]. UCT's ability to deal with the large state-space of Go and implicitly carry out the necessary spatial reasoning, makes it an interesting possibility for RTS tactical planning. However, there are a number of fundamental differences between the RTS and Go domains, which makes its applicability unclear. The main contribution of this paper is to describe an abstract problem formulation of tactical assault planning for which UCT is shown to be very effective compared to a number of baselines across a range of tactical assault scenarios. This is a significant step toward arriving at a full model-based planning solution to the RTS tactical problem.

2 The RTS Tactical Assault Domain

In general, the tactical part of RTS games involves both planning about defensive and offensive troop movements and positioning. The ultimate goal is generally to completely destroy all enemy troops, which is typically achieved via a series of well timed assaults while maintaining an adequate defensive posture. In this paper, we focus exclusively on solving RTS tactical assault problems, where the input is a set of friendly and enemy units along with an optimization objective. The planner must then control the friendly troops in order to best optimize the objective. The troops may be spread over multiple locations on the map and are often organized into groups. Typical assault objectives might be to destroy the selected enemy troops as quickly as possible or to destroy the enemy while losing as little health as possible. Note that our focus on the assault problem ignores other aspects of the full RTS tactical problem such as developing a strong defensive stance and selecting the best sequence of assaults to launch. Thus, we view our planner as just one component to be called by a human or high-level planner.

Successful tactical assault planning involves reasoning about the best order of attacks on the enemy groups and the size of the friendly groups with which to launch each of the attacks, considering the attrition and time taken for each of the individual battles. This presents an interesting and challenging planning problem where we need to deal with a large state space involving durative actions that must be executed concurrently. To help manage this complexity, we focus on an abstract version of the tactical assault problem, where we reason about proximity-based groups of units instead of individual units. This abstraction is very much in line with how typical assaults are fought in RTS games, where tactics are controlled at a group level. Thus, the abstract state space used by our planner is in terms of properties of the sets of enemy and friendly groups, such as health and location. The primary abstract actions we consider are joining of groups and attacking an enemy group. The micromanagement of individual agents in the groups under each abstract action is left to the default AI of the game engine.

3 Related Work

Monte Carlo sampling techniques have been used successfully to produce action strategies in board games like bridge, poker, Scrabble and Go. The main difference between these games and our domain is that all these games are turn-based with instantaneous effects, whereas actions in the RTS domain are simultaneous and durative. [Chung *et al.,* 2005] used a form of Monte Carlo simulation for RTS tactical planning with considerable success. At each planning epoch the approach performed limited look-ahead to select an action by Monte Carlo simulation of random action sequences followed by the application of an evaluation function. Unfortunately this is highly reliant on the availability of a quality evaluation function, which makes the approach more challenging to bring to a new domain and less adaptable to new goal conditions.

Another Monte Carlo approach for RTS tactical problems [Sailer *et al.*, 2007] assume a fixed set of strategies, and at each step use simulation to estimate the values of various combinations of enemy and friendly strategies. These results are used to compute a Nash-policy in order to select a strategy for execution. A weakness of this approach is its restriction to only consider strategies in the predefined set, which would need to be constructed on a per-domain basis. Comparably, our approach does not require either a strategy set or an evaluation function, but rather only that a set of abstract actions are provided along with the ability to simulate their effects. However, unlike their approach, our planner assumes that the enemy is purely reactive to our assault, whereas their approach reasons about the offensive capacity of the enemy, though restricted to the provided set of strategies. This is not a fundamental restriction for our planner as it can easily incorporate offensive actions of the enemy into the Monte Carlo process, likely at a computation cost.

Recent work has also focused on model-based planning for the resource-production aspect of RTS games [Chan *et al.*, 2007]. While that work provides mechanisms for real-time planning with temporal, concurrent actions and numeric state properties, it is quite specialized to resource production and it is not clear how to apply it to tactical problems. Recent work, has also applied reinforcement learning to the problem of controlling individual agents in tactical battles between two groups of units [Wilson *et al.*, 2008], which could be leveraged by our method in place of the default AI.

4 UCT for Tactical Assault Planning

In this section, we describe our overall planning architecture, the UCT algorithm, and our application of UCT to tac-

tical assault planning by detailing our search space formulation.

Planning Architecture. RTS games are highly dynamic due to the stochastic aspects of the game environment, along with the unpredictability of the opponent's actions and incoming goals. For this reason, we utilize an online planning approach rather than computing an offline plan and then attempting to follow it. As explained earlier, in order to reduce complexity, our planner reasons at an abstract level about groups of units, rather than about individuals. In our current implementation, we compute these groups at each decision epoch based on unit proximity via simple agglomerative clustering. However, it is straightforward to incorporate any other grouping scheme, e.g. as computed by a higher level planner.

Given a set of unit groups at the current decision epoch, our planner then utilizes the Monte Carlo planning algorithm UCT, described in the next section, to assign abstract group actions to all of the groups, which are then executed in the game until the next decision epoch is triggered. In our current implementation a decision epoch is triggered whenever any of the groups becomes idle after completing the currently assigned action. It is straightforward to incorporate additional trigger conditions for decision epochs into our approach, e.g. when an unexpected enemy group is encountered. The online planning loop repeats until reaching an end state, which for tactical assault problems is when either all of the friendly or enemy units have been destroyed.

We have instrumented our RTS engine Stratagus to support two types of abstract group actions, which the planner can select among. First, the action *Join*(*G*), where *G* is a set of groups, causes all of the groups in *G* to move toward their centroid location and to form into a larger joint group. This action is useful for the common situation where we want to explicitly join groups before launching a joint attack so that the units among these groups arrive at the enemy at the same time. Such joint attacks can be much more effective than having the individual groups attack independently, which generally results in groups reaching the enemy at different times. The second abstract action, *Attack*(*f*,*e*), where *f* is a friendly group and *e* is an enemy group, causes *f* to move toward and attack *e*. Currently the actions of individual friendly agents during an attack are controlled by the default Stratagus AI, though in concept it is straightforward to utilize more advanced controllers, e.g. controllers learned via reinforcement learning [Wilson *et al.*, 2008].

The UCT Algorithm. UCT is a Monte Carlo planning algorithm first proposed by [Kocsis and Szepesvari, 2006], which extends recent algorithms for bandit problems to sequential decision problems while retaining the strong theoretical performance guarantees. At each decision epoch, we use UCT to build a sparse tree over the state-space with the current state as the root, edges corresponding to actions, and leaf nodes corresponding to terminal states. Each node in the resulting tree stores value estimates for each of the available actions, which are used to select the next action to be executed. UCT is distinct in the way that it constructs the tree and estimates action values. Unlike standard mini-max search or sparse sampling [Kearns *et al*., 2001], which typically build depth bounded trees and apply evaluation functions at the leaves, UCT does not impose a depth bound and does not require an evaluation function. Rather, UCT incrementally constructs a tree and updates action values by carrying out a sequence of Monte Carlo rollouts of entire game sequences starting from the root to a terminal state. The key idea behind UCT is to intelligently bias the rollout trajectories toward ones that appear more promising based on previous trajectories, while maintaining sufficient exploration. In this way, the most promising parts of the tree are grown first, while still guaranteeing that an optimal decision will be made given enough rollouts.

It remains to describe how UCT conducts each rollout trajectory given the current tree (initially just the root node) and how the tree is updated in response. Each node *s* in the tree stores the number of times the node has been visited in previous rollouts $n(s)$, the number of times each action *a* has been explored in *s* in previous rollouts $n(s,a)$, and a current action value estimate for each action $Q(s,a)$. Each rollout begins at the root and actions are selected via the following process. If the current state contains actions that have not yet been explored in previous rollouts, then a random unexplored action is selected. Otherwise if all actions in the current node s have been explored previously then we select the action that maximizes the upper confidence bound given by,

$$Q^{+}(s,a) = Q(s,a) + c \times \sqrt{\frac{log\ n(s)}{n(s,a)}}$$

where *c* is a domain dependent constant. After selecting an action, it is simulated and the resulting state is added to the tree if it is not already present. This action selection mechanism is based on the UCB bandit algorithm [Auer *et al.*, 2002] and attempts to balance exploration and exploitation. The first term rewards actions whose action values are currently promising, while the second term adds an exploration reward to actions that have not been explored much and goes to zero as an action is explored more frequently.

In practice the value of the constant *c* has a large impact on performance. In our application, this is particularly true since unlike the case of board games such as Go where the action values are always in the range of [0,1], in our applica-

Figure 1. Screenshot of a typical Wargus game scenario

tions the action values can be quite large and have a wide variance across different tactical scenarios. Thus, we found it difficult to find a single constant that provided robust performance. For this reason, we use a variation of UCT where we let $c = Q(s,a)$, to ensure that the exploration term is on the same scale as the action values. While the theoretical implications of this choice are not clear, the practical improvement in our experience is significant.

Finally, after the trajectory reaches a terminal state the reward R for that trajectory is calculated based on the current objective function. The reward is used to update the action value function of each state along the generated trajectory. In particular, for any state action pair (s, a) on the trajectories we perform the following updates,

$$n(s,a) \leftarrow n(s,a) + 1; \quad n(s) \leftarrow n(s) + 1$$
$$Q(s,a) \leftarrow Q(s,a) + \frac{1}{n(s,a)}[R - Q(s,a)]$$

Search Space Formulation. UCT is most naturally applied to domains that involve sequential, non-durative actions, as in most board games. However, in our domain, actions have variable durations and must be executed concurrently. We now describe a search space that allows for UCT to search over these aspects of our domain.

Each abstract state in our search space is described by: 1) the current set of friendly and enemy groups and their properties including group hit points (i.e. health) and mean location, 2) the current action being taken by each friendly group, and 3) the current game cycle/time. Following [Chung *et al.*, 2005] the hit points $HP(G)$ of a group G is a measure of the overall health of a group and is recalculated each time new groups are formed based on the hit points of the joining groups using the formula $HP(G) = (\sum \sqrt{HP_i})^2$, where HP_i is the hit points for the i'th joining group. This formula better reflects the effective hit point power of a group compared to summing the hit points of joining groups. For example, a group of 2 units with 50 hit points each is more useful in battle than 1 unit with 100 hit points.

Given these search nodes, we must now describe the arcs of our search space. At each search node with at least one idle friendly group (i.e. no assigned action), the available arcs correspond to assigning a single idle group an action, which can be either to attack a specified enemy group, or to join another friendly group in the current search node. Note that in the case of join, if the group being joined to is currently assigned to join yet another group, then the join action is applied to all of the groups. From this we see that a search node with n idle friendly groups and m enemy groups has $O(n^2 + n \times m)$ possible successors, each corresponding to an action assignment to an idle group.

It is important to note that these "assignment search arcs" do not increment the game cycle of the next search node and do not change the properties of any groups. Rather they should be viewed as book keeping search steps that modify the "internal state" of the groups to keep track of the action that they have been assigned. The game cycles are incremented and actions are simulated only from search states with no idle groups, where the only choice is to move to a search node that results from simulating the game according to the current action selections until one of the groups becomes idle. The resulting successor state will reflect the updated position and hit points of the groups.

Note that under this search space multiple search steps are required to assign activities to multiple idle groups. An alternative formulation would have been to allow for single search steps to jointly assign actions to all idle groups in a node. This would exponentially increase the number of arcs out of the nodes, but decreased the depth required to reach a final state since multiple search steps would no longer be necessary to assign joint actions. We chose to use the former search space since it appears to be better matched to the UCT approach. Intuitively, this is because our search space contains many search nodes on route to a joint action assignment, each representing a partial assignment, which allows UCT to collect quality statistics about each of the encountered partial assignments. Accordingly the rollouts can be biased toward partial assignments that appear more promising. Rather, the later search space that has an arc for each joint action is unable to gather any such statistics and UCT will be forced to try each one independently. Thus our search space allows for UCT to much more effectively exploit previous rollouts when searching for joint actions.

Monte Carlo Simulation. A key component of our approach is to simulate the effect of abstract group actions on the abstract state in order to generate UCT's rollout trajectories. This involves estimating the times for actions to complete and how they alter the positions and hit points of existing friendly and enemy groups. Full details of the simulation process are in the full report [Balla, 2009] and are omitted here due to space constraints. In concept the simulation process is straightforward, but involves careful book keep of the concurrent activity going on. For example, to simulate multiple groups attacking another group one must keep track of the arrival times of each attacking group and account for the additional offensive power that becomes available. For predicting the rate of hit point reductions during encounters and the movement times, we utilized simple numeric models, which were hand-tuned based on examples of game play. In general, it would be beneficial to incorporate machine learning techniques to continually monitor and improve the accuracy of such models.

5 Experiments and Results

We present experiments in the game of Wargus, which is run on top of the open-source Stratagus RTS engine.

Experimental Setup. We created 12 game scenarios for evaluation that differ in the number of enemy and friendly units, their groupings, and the placement of the groups across the 128x128 tile map. Figure 1 shows a screen shot of the action during one of these scenarios. The upper-left corner depicts an abstract view of the full map showing the locations of 2 friendly and 4 enemy groups. The main part of the figure shows a zoomed in area of the map where an encounter between enemy and friendly groups is about to take place. In order to simplify the simulation of actions in

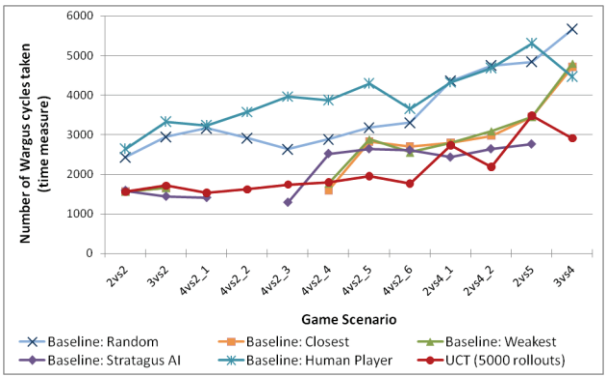

Figure-2: Time results for UCT(t) and baselines.

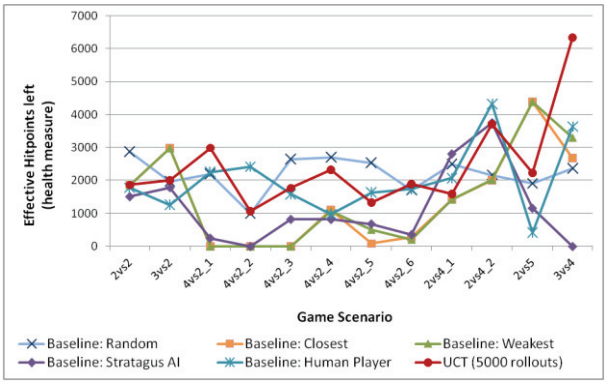

Figure-3: Hit point results for UCT(t) and baselines.

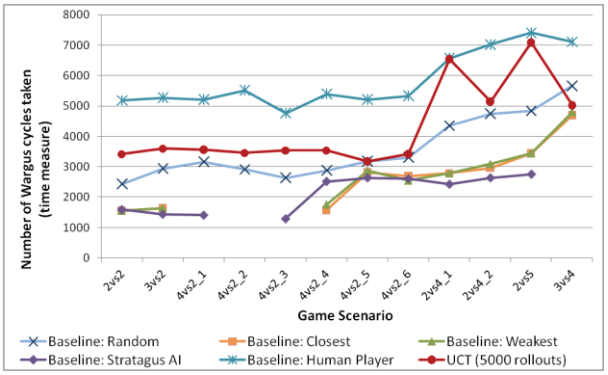

Figure-4: Time results for UCT(hp) and baselines.

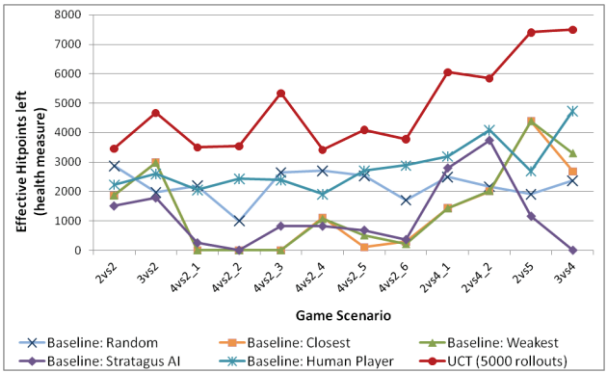

Figure-5: Hit point results for UCT(hp) and baselines.

this initial investigation, we have restricted all of the scenarios to utilize a single type of unit known as a footman. All of our scenarios are designed so that there is a winning strategy, though the level of intelligence required to win varies across the scenarios. The scenarios vary the number of enemy and friendly units from 10 to 20 per side and the number of initial groups from 2 to 5. Details of the scenarios are available in the full report [Balla, 2009].

Planners. Our experiments consider two version of UCT: 1) UCT(t), which attempts to minimize the time, as measured by number of game cycles, to destroy the enemy, and 2) UCT(hp), which attempts to maximize the effective hit points of the friendly units remaining after defeating the enemy. The only difference between these two versions of UCT is the value of the reward returned at each terminal node at the end of each rollout, which is equal to the objective under consideration. Unless otherwise noted, we used 5000 rollout trajectories for UCT.

We compare against 5 baselines: 1) Random, which selects random join and attack actions for idle groups, 2) Attack-Closest, which causes any idle group to attack the closest enemy, 3) Attack-Weakest, which causes an idle group to attack the weakest enemy group and in the case of ties to select the closest of those, 4) Stratagus-AI, which controls the friendly units with the default Stratagus AI, and 5) the performance achieved by an experienced human player.

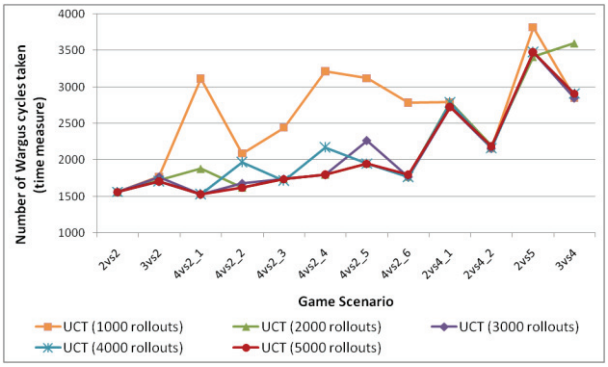

Figure-6: Time results for UCT(t) with varying rollouts.

Results. We ran the planners on all benchmarks and measured the time (game cycles) to defeat the enemy and the hit points of the friendly forces at the end of the game. For the Random baseline and UCT the results are averaged over 5 runs. Figures 2 and 3 give the results for UCT(t) and the baselines for the time and hit point metrics respectively. The x-axis labels give a description of the scenarios in terms of the number of enemy and friendly groups. For example, 4vs2_1 is the first scenario that involves 4 friendly groups and 2 enemy groups on the initial map. When a planner does not win a game, the hit points are recorded as 0 and no point plotted for the time metric in Figure 2.

We notice first that UCT(t) is the only planner besides the human to win all of the scenarios. By utilizing a model-based approach our planner is able to avoid many of the fatal mistakes of the other planners. Furthermore, Figure 2 shows that UCT(t) is always among the top performers as measured by completion time, which is the objective being optimized by UCT(t). In Figure 3, we see that UCT(t) is also often among the top performers in terms of effective hit points, though in some cases it is significantly outperformed by one or more of the baselines, which should be expected since UCT(t) is not trying optimize hit points. The human player has great difficulty trying to optimize this objective. The primary reason for this is the difficulty in quickly controlling the units using the Stratagus user interface.

Figures 4 and 5 are similar to the previous two figures but plot results for UCT(hp) rather than UCT(t). We see from Figure 5, that UCT(hp) outperforms all other planners in terms of effective hit points in all but one of the scenarios and again it is the only planner besides the human that wins all of the scenarios. From Figure 3, we further see that UCT(t), which did not attempt to optimize hit points, did not perform nearly as well in terms of hit points. This indicates that our UCT planner is clearly sensitive to the optimization objective given to it. UCT(hp) performs poorly in terms of completion time, which should be expected since the best way to optimize hit points is to take time to initially form a large group and then to attack the enemy groups sequentially. However, we see that UCT(hp) is still able to significantly improve on the completion time compared to the human player. Overall, for both metrics our planner has advantages compared to the other baselines and the human.

We now compare the performance of UCT with respect to the number of rollout trajectories. We ran variations of the UCT(t) planner on all scenarios where we increased the number of rollouts from 1000 through 5000 in steps of 1000. Figure 6 shows the results for the time metric. It can be observed that limiting to only 1000 rollouts per decision results in significantly worse performance in most of the scenarios. Increasing the number of rollouts improves the performance and reaches that with 5000 rollouts in all but the few scenarios. Increasing the number of rollouts beyond 5000 for UCT(t) did not produce significant improvements.

Our current prototype is not yet fast enough for true real-time performance in the larger scenarios when using 5000 rollouts per decision epoch. The most expensive decision epoch is the first one, since the number of groups is maximal resulting in long rollout trajectories and more complex simulations. However, later decisions are typically much faster since the number of groups decreases as the assault proceeds. In the worst case for our most complex scenario, the first decision took approximately 20 seconds for 5000 rollouts, while the later stages took a maximum of 9 seconds, but usually much faster on average. Our current implementation has not yet been optimized for computation time and there are significant engineering opportunities that we believe will yield real-time performance. This is the case in Go, for example, where the difference between a highly optimized UCT and a prototype can be orders of magnitude.

6 Summary and Future Work

To the best of our knowledge there is no domain-independent planner that can handle all of the features of tactical planning. Furthermore, prior Monte Carlo methods required significant human knowledge. Our main contribution is to show that UCT, which requires no such human knowledge, is a promising approach for assault planning. Across a set of 12 scenarios in the game of Wargus, our planner is a top performer compared to a variety of baselines and a human player. Furthermore it was the only planner to find winning strategies in all of the scenarios. In the future, we plan to optimize our implementation to arrive at truly real-time performance. We also plan to integrate machine learning techniques to learn improved simulation models, making it easier to evaluate our planner in more complex scenarios involving multiple unit types and more sophisticated adversaries. Finally, we plan to integrate our planner into an overall architecture for full RTS game play.

Acknowledgements

This work was supported by NSF grant IIS-0546867 and DARPA contract FA8750-05-2-0249.

References

[Auer *et al.*, 2002] P. Auer, N. Cesa-Bianchi, P. Fischer. Finite-time Analysis of the Multiarmed Bandit Problem, Machine Learning, 2002

[Balla, 2009] Radha-Krishna Balla. UCT for Tactical Assault Battles in Real-Time Strategy Games. M.S. Thesis, Oregon State Univ., 2009

[Buro and Furtak, 2004] Michael Buro, Timothy M. Furtak. RTS Games and Real-Time AI Research. Proc. Behavior Representation in Modeling and Simulation, 2004

[Chan *et al.*, 2007] Hei Chan, Alan Fern, Soumya Ray, Nick Wilson and Chris Ventura. Online Planning for Resource Production in Real-Time Strategy Games. ICAPS, 2007

[Chung *et al.*, 2005] Michael Chung, Michael Buro, Jonathan Schaeffer. Monte Carlo Planning in RTS Games. IEEE Symposium on Computational Intelligence and Games, 2005

[Gelly and Silver, 2007] Sylvian Gelly, David Silver. Combining Online and Offline Knowledge in UCT. ICML, 2007.

[Gelly and Wang, 2006] Sylvian Gelly, Yizao Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go. NIPS, 2006.

[Kocsis and Szepesvari, 2006] Levente Kocsis, Csaba Szepesvari. Bandit Based Monte-Carlo Planning. ECML, 2006.

[Kovarsky and Buro, 2005] Alexander Kovarsky, Michael Buro. Heuristic Search Applied to Abstract Combat Games. Proc. Canadian Conference on Artificial Intelligence, 2005

[Kearns *et al*., 2001] Michael Kearns, Y. Mansour, and A. Ng. A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes. *Machine Learning,* 49 (2-3).

[Sailer *et al.*, 2007] Frantisek Sailer, Michael Buro, and Marc Lanctot. Adversarial Planning Through Strategy Simulation. Proc. IEEE Symposium on Computational Intelligence and Games, 2007

[Wilson *et al.*, 2008] Aaron Wilson, Alan Fern, Soumya Ray and Prasad Tadepalli. Learning and Transferring Roles in Multi-Agent Reinforcement Learning. Proc. AAAI-08 Workshop on Transfer Learning for Complex Tasks, 2008