AN ABSTRACT OF THE THESIS OF

Radha-Krishna Balla for the degree of Master of Science in Computer Science presented on February 19, 2009.

Title: UCT for Tactical Assault Battles in Real-Time Strategy Games.

Abstract approved:

______________________________________________________
Alan Fern

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 thesis, 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.
UCT for Tactical Assault Battles in Real-Time Strategy Games

by
Radha-Krishna Balla

A THESIS

submitted to
Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented February 19, 2009
Commencement June 2009

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

__________________________
Radha-Krishna Balla, Author
ACKNOWLEDGEMENTS

I would like to thank my academic advisor Dr. Alan Fern for his invaluable guidance, insight and support throughout my work here and for being flexible with me. I would also like to thank Dr. Prasad Tadepalli and Dr. Weng-Keen Wong for the excellent courses that they have taught me and for being part of my Defense committee.

Next I would like to thank my fellow graduate students, especially Balaji Reddy, Janardhan Rao and Kannan for helping me with my work. Last but not least, I would like to thank my parents and sister for their constant support and encouragement, and to Oregon for being such a beautiful home away from home.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 The RTS Tactical Assault Domain</td>
<td>4</td>
</tr>
<tr>
<td>3 Related Work</td>
<td>7</td>
</tr>
<tr>
<td>4 UCT for Tactical Assault Planning</td>
<td>10</td>
</tr>
<tr>
<td>4.1 Planning Architecture</td>
<td>10</td>
</tr>
<tr>
<td>4.2 The UCT Algorithm</td>
<td>11</td>
</tr>
<tr>
<td>4.3 Search Space Formulation</td>
<td>15</td>
</tr>
<tr>
<td>4.4 Domain-specific Challenges</td>
<td>17</td>
</tr>
<tr>
<td>4.4.1. State space abstraction</td>
<td>17</td>
</tr>
<tr>
<td>4.4.2. Concurrency of actions</td>
<td>18</td>
</tr>
<tr>
<td>4.5 Monte Carlo Simulation</td>
<td>20</td>
</tr>
<tr>
<td>5 Experiments and Results</td>
<td>24</td>
</tr>
<tr>
<td>5.1 Experimental Setup</td>
<td>24</td>
</tr>
<tr>
<td>5.2 Planners</td>
<td>26</td>
</tr>
<tr>
<td>5.3 Results and Analysis</td>
<td>27</td>
</tr>
<tr>
<td>6 Summary and Future Work</td>
<td>33</td>
</tr>
<tr>
<td>Bibliography</td>
<td>35</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Before the start of a battle</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Middle of the battle</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>End of the battle</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Pseudo code for building the UCT tree at a state</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Pseudo code for Monte-Carlo simulations for a composite attack action.</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Time results for UCT(t) and baselines.</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Hit point results for UCT(t) and baselines.</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Time results for UCT(hp) and baselines.</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Hit point results for UCT(hp) and baselines.</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>Time results for UCT(t) with varying rollouts.</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Details of the different game scenarios</td>
<td>25</td>
</tr>
</tbody>
</table>
UCT for Tactical Assault Battles in Real-Time Strategy Games

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 thesis, 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 the various 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 maximize 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. Some of the primary challenges are listed below:

- Our tactical battle formulation involves temporal actions with numeric effects.
• The problems typically involve the concurrent control of multiple military units.

• Performing well requires some amount of spatial-temporal reasoning.

• 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.

• 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 commercial games utilizing static script-based mechanisms, which only mimic intelligent behavior. One exception, which has shown considerable promise, is the use of Monte-Carlo planning for tactical problem [3], [8]. 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 [7], which has shown impressive success in a variety of domains, most notably the game of Go (see [5] and [4]). 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 thesis 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.

The remainder of this thesis is organized as follows. Chapter 2 describes the RTS domain with special emphasis on the tactical assault problem, which is the main problem that we attempted to solve in this work. Chapter 3 gives a literature survey of the relevant work that has been done in this area. In Chapter 4 we describe the UCT algorithm and Monte-Carlo simulations, along with details about how they were implemented in our domain. Chapter 5 presents the various experiments that were conducted and provides an analysis of the results of our planner in comparison to the various baseline planners and a human player. We conclude in Chapter 6, along with a discussion about possible future work in this area.
2 The RTS Tactical Assault Domain

In general, the tactical part of RTS games\(^1\) 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 thesis, 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.

---

\(^1\) Refer to [20] for a brief discussion of Real-Time Strategy games in general.
For the current work, we used the game of Wargus [19] running on the open source engine Stratagus [18]. Figure 2.1, Figure 2.2 and Figure 2.3 show the screenshots of various stages of a typical battle scenario in a game of Wargus. In each of the figures, the upper left corner of the screen shows a mini-map of the real-time rendering of the entire game. The game statistics are presented on the top bar, and the details of selected units are presented under the mini-map in the left hand portion. The main portion of the screen shows the zoomed in view of a particular area in the map; this is where the player gives instructions to the various units in the game. Figure 2.1 shows a group of 8 footmen\(^2\) advancing towards an enemy group of 5 footmen to attack, Figure 2.2 shows a time point somewhere in the middle of the battle and finally Figure 2.3 shows the screen where the battle is completed, with the former group winning over the latter. It can be observed from the mini-map in all the 3 screenshots, that there is simultaneously another battle going on in a different location of the map.

As can be seen from the above figures, 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

\(^2\)Footmen are a type of military units in the game of Wargus.
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 micro-management of individual agents in the groups under each abstract action is left to the default AI of the game engine.
3 Related Work

The primary characteristics of the tactical assault domain are large state space, durative actions with an extent of stochasticity, simultaneous moves and real-time execution of the actions. In recent years there has been significant research to create planners that deal with some or all of the above mentioned aspects. We cover the related work in this chapter and also explain the motivation for designing a planner based on Monte Carlo simulations that is guided by the UCT algorithm.

Monte Carlo sampling techniques have been used successfully to produce action strategies in board games with imperfect information like backgammon [13], bridge [14], poker [15] and Scrabble [16], and in two-player perfect information games like Go ([17], [5] and [4]). The primary 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.

Michael Chung et al., [3] have 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. This process is repeated over a number of simulations, to get the best-looking plan among them. 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. Our work has some commonality with this work in that we use Monte Carlo simulations in a similar fashion to estimate the results of actions, but we have made use of an algorithm called UCT [7] which utilizes an effective method for balancing exploration of new actions and exploitation of promising actions by constructing a search tree. Complete rollouts of the game are simulated in building the search tree and value functions are computed which are propagated up the tree, and these values are used to guide the exploration-exploitation tradeoff in subsequent rollouts.
Further details about the algorithm and its implementation are given in detail in the subsequent chapters.

Frantisek Sailer *et al.*, [8] have also used Monte Carlo planning to deal with a domain that is more similar to that of ours where they deal with tactical battles which involve both the opposing teams attacking each other. They assume the availability of a fixed set of strategies (a bundled set of low-level actions achieving a sub-goal) and at each step use Monte Carlo simulation to estimate the values of various combinations of the enemy and friendly strategies. These results are used to compute a Nash-equilibrium policy from which the best strategy is selected 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. This will involve considerable time and resources of an expert player. Comparably, our approach does not require either a strategy set or an evaluation function, but rather only that a set of abstract actions are to be 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 simulation process, perhaps at a computation cost.

Recent work (Hei Chan *et al.*, [2]) has also focused on model-based planning for the resource-production aspect of RTS games. They use a means-ends analysis to obtain good sequential plans followed by a rescheduling of actions to achieve concurrency and a bounded search over sub-goals to improve the makespan. While that work provides mechanisms for real-time planning with temporal, concurrent actions and numeric state properties, it is quite specialized to the resource production domain which has deterministic actions that have well-defined preconditions and effects, and it is not clear how to apply the approach to tactical problems where the actions are more stochastic in nature.
Recent work (Aaron Wilson et al., [9]), has also applied reinforcement learning to the problem of controlling individual agents in tactical battles between two groups of units. That work is complementary to ours in that the learned controllers could be used to replace the default AI, which we currently use for controlling individuals.
4 UCT for Tactical Assault Planning

In this section, we first describe the overall architecture of our planner. Next, we describe the UCT planning algorithm in terms of general search spaces, and proceed to describe how UCT is applied to tactical assault planning by detailing our search space formulation. The challenges faced in customizing the concepts of UCT to the RTS domain are described next. Finally, we describe the Monte-Carlo simulations and the way they were carried out in our domain.

4.1 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. An online planner would be capable of finding good plans for any situation (game state), and then would be able to come up with new plans as the game state changes; in contrast, an offline planner would analyze the game at the initial state and come up with a series of actions to be executed until it reaches the goal state, ignoring any stochasticity during the plan execution.

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 concurrently 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:

1) \textit{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. Larger-size groups have not only the advantage of defeating an enemy group successfully, but doing so in a short time while losing minimum health.

2) \textit{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 [9].

4.2 The UCT Algorithm

UCT is a Monte Carlo planning algorithm first proposed by [7], 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 [6], 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 \( n(s) \). Each edge \( a \) (connected to the node \( s \)) in the tree stores the number of times that action has been explored in \( s \) in previous rollouts \( n(s,a) \), and a current action value estimate \( Q(s,a) \). 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,

\[
Q^+(s, a) = Q(s, a) + c \times \sqrt{\frac{\log n(s)}{n(s,a)}} \quad (4.1)
\]

where \( c \) is a domain dependent constant. After selecting an action, its effects are 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 [1] and attempts to balance exploration and exploitation. The first term in the above formula, 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 applications 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. Based on these updated action value estimates $Q^+(s, a)$, the action that maximizes this action value is chosen given by,

$$ \pi(s) = \arg \max_a Q^+(s, a) \quad \text{(4.2)} $$

where $\pi(s)$ denotes the policy that is followed to choose the best action $a$ from state $s$.

Finally, after the trajectory reaches a terminal state the reward $R$ for that trajectory is calculated based on the current objective function. As the reward $R$ is calculated only at the end state of a game (simulated), this objective function\(^3\) can be a simple evaluation, unlike an evaluation in the middle of a game which would have required a complex evaluation function to be designed by experts. 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,

$$ n(s, a) \leftarrow n(s, a) + 1 $$
$$ n(s) \leftarrow n(s) + 1 $$
$$ Q(s, a) \leftarrow Q(s, a) + \frac{1}{n(s, a)} [R - Q(s, a)] \quad \text{(4.3)} $$

\(^3\)The 2 types of objective functions used in our domain, are explained in Section 5.2.
Pseudo-code:

At each interesting time point in the game:

\[ \text{build}\_\text{UCT}\_\text{tree}(\text{current state}); \]
\[ \text{choose argmax action(s) based on the UCT policy; // as given by Formula (4.2)} \]
\[ \text{execute the aggregated actions in the actual game;} \]
\[ \text{wait until one of the actions get executed;} \]

\[ \text{build}\_\text{UCT}\_\text{tree}(\text{state}) : \]
\[ \text{for each UCT pass do} \]
\[ \text{run UCT}\_\text{rollout}(\text{state}); \]

\[ \text{UCT}\_\text{rollout}(\text{state}): \text{recursive algorithm} \]
\[ \text{if leaf node reached then} \]
\[ \text{estimate final reward; // based on the objective function} \]
\[ \text{propagate reward up the tree and update value functions; // as given by Formulae (4.3) and (4.1)(4.1)(4.1) \]
\[ \text{return;} \]
\[ \text{populate possible actions; // as given by Formula (4.5)} \]
\[ \text{if all actions explored at least once then} \]
\[ \text{choose the action with best value function; // as given by Formula (4.2)} \]
\[ \text{else if there exists unexplored action} \]
\[ \text{choose an action based on random sampling;} \]
\[ \text{run Monte-Carlo simulation to get next state based on current state and action; // as described in Section 4.5} \]
\[ \text{call UCT}\_\text{rollout}(\text{next state}); \]

Figure 4.1 Pseudo code for building the UCT tree at a state
Given enough trajectories and an appropriate choice of \( c \), the action values are guaranteed to converge to ground truth. The complete logic used in building the UCT tree at any game state, is summarized in Figure 4.1.

### 4.3 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:

- the current set of friendly and enemy groups and their properties including group hit points (i.e. health) and mean location,
- the current action being taken by each friendly group, and
- the current game cycle/time.

Following [3], 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. From this we see that a search node with \( n \) idle friendly groups and \( m \) enemy groups, the number of action choices would be,

\[
\text{Number of action choices} = \binom{n_{\text{friendly}}}{2} + n_{\text{friendly}} \times n_{\text{enemy}} \quad \text{(4.5)}
\]

each corresponding to an action assignment to an idle group. 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. Similarly in the case of attack, if multiple attack actions correspond to different friendly groups assigned to a single enemy group, then the actions are aggregated together according to the scenario.\(^4\)

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,

\(^4\) The logic used in the aggregation of the join and attack actions, is explained in detail under Section 4.4.2.
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.

### 4.4 Domain-specific Challenges

To maintain the flow of the explanation, we first describe the challenges faced in customizing the UCT algorithm to our domain, before venturing into the details of how the Monte-Carlo simulations are carried out. These concepts would be necessary to be able to fully appreciate the logic used in carrying out the simulations. To keep things simple, in the present work we have dealt with only a single type of units – *footmen*.

Tactical battles in real-time strategy games involve a large number of individual units spread over the map, battling against a comparable-sized opponent army. The different military units may belong to different types (e.g., *Footman, Archer* etc.) and sometimes there may be multiple opponent teams involved. All this amounts to a significant state-space, which needs to be abstracted to some level for the planning problem to be tractable. In addition to that, the actions in this domain are durative (take time for action to complete once started) and simultaneous moves are allowed, which means that different units can decide to do different actions concurrently. As the game is played in real-time, the planning algorithm cannot afford to take significant time, since the opponent can take advantage of any time duration that the current player stays idle.

The above factors pose significant challenges to planning in our domain. In the following subsections, we describe how some of these challenges have been tackled.

#### 4.4.1. State space abstraction

To prevent explosion of state-space due to the large number of units present in the game, we consider grouping of similar units based on proximity. In practice, this idea of
grouping of units is logical, because most of the actions in real-time strategy games are carried out by groups of similar units, to multiply the effect and complete the corresponding action in shorter duration. Also, the movement of units on the map, is given in terms of tiles (square blocks) instead of exact map coordinates, to reduce the dimensionality without sacrificing much in terms of functionality.

In our implementation, we have set a limit of 5 tiles for the proximity criterion for grouping, which would mean that any pair of units within 5 tiles distance to each other (in any direction), would be added to the same group. Thus there would be clusters of friendly and enemy units spread all over the map and we would be dealing with them only at a group level for any join/attack actions.

### 4.4.2. Concurrency of actions

As explained at the starting of this section, it is important to deal with concurrent actions so that no more than one friendly group is idle at any single point of time. In the action selection step of building the UCT tree, we consider only a single action – either a join action between two friendly groups or a friendly group deciding to attack an enemy group. To enable concurrency, we proceed with the rollout without actually advancing the state, by choosing a next action among the remaining friendly groups which have not yet been assigned any action. In the UCT tree, this would still be represented as a new state but without any change in the properties of the individual groups except the book keeping operation that some of them have been assigned to a particular action. This process of rollout continues until all friendly groups have been assigned to some action. Once it is observed that there is no idle group at the current state, the actual Monte Carlo simulation will be run, and the next state is estimated and the global clock advanced.

Even within the Monte Carlo simulations, not all actions are simulated until the end result is reached. Among the concurrently executing actions, the action that gets completed first is simulated, and the rest of the actions undergo partial simulations, which give the result of running the respective actions until the time when the fastest action gets
completed. This would ensure that only one group remains idle at any time point, and another UCT tree can be built freshly to decide the next course of action.

We also do an aggregation of the concurrent actions, so that two or more groups involved in similar actions can be joined together, so as to be able to execute the actions more effectively and in less time. The logic for aggregating join and attack actions is explained below.

**Case-i:** Aggregation of Join actions: This aggregation is quite simple to handle. Each pair of actions which have a common group are identified, and the two actions are clubbed to form a single action involving a join action over multiple (two or more) groups, such that the different groups would join at their centroid.

**Case-ii:** Aggregation of Attack actions: This aggregation can be little complex based on the scenario. When there are multiple attack actions involving the same enemy group, it might be advisable to join the forces of the different friendly groups and attack at once to be more effective. But this may not always be feasible because, if one of the friendly groups is already attacking the enemy group, it will not be able retreat and join the other groups. This can also be the case when the friendly group is too close to the enemy group to be able to withdraw from the attack. This dilemma would lead to 2 sub-cases.

**Case-ii (a):** None of the friendly groups are currently attacking: In this case, the friendly groups can join together and launch a combined offensive. Thus the individual attack actions are changed to a single join action involving all the friendly groups. The attack action is ignored, and if the UCT algorithm is designed well, it is expected to pick the corresponding attack action as the follow-up action.

**Case-ii (b):** At least one of the friendly groups is already attacking the common enemy group: In this case, since there is no turning back of the attacking group, the rest of the friendly groups have to rush as early as possible to the attack site, so that they can lend support to the currently involved friendly group(s) in defeating the enemy group.

---

5 It must be noted that, since we are dealing with only footmen, the speed of all units will be same. Hence, if a unit tries to run away in the middle of a battle, there will most likely be an opposing unit pursuing it at the same speed. Therefore it could never escape a battle, once it is started.
There would not be sufficient time to first join the other groups and then attack, because by then the currently executing action might have completed. Hence as part of the aggregation, all the attack actions are combined to form a composite attack action, with the various friendly groups ordered based on their distance from the enemy group. Based on this ordering, the simulation logic will take care of producing the intermediate states. The details of the Monte Carlo simulations are explained in the next section.

### 4.5 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. 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. The details of how the Monte-Carlo simulations are carried out in our domain are explained below.

Once an action has been selected at a node based on the UCT logic, Monte-Carlo simulations are carried out to predict the resultant state. The logic for the simulations is based on the actual game play of Wargus, and the various parameters are obtained by observing the result of different types of individual join and attack actions and their results.

\[
Time_{join} = Time_{to\text{-move}}
\]
\[ Time_{attack} = Time_{to-move} + Time_{to-attack} \]

\[ Time_{to-move} = f(distance\_to\_be\_covered, speed\_of\_units) \]

\[ Time_{to-attack} = f(HP_{friendly - group}, HP_{enemy - group}) \]

\[ HP_{friendly} (join) = (\Sigma \sqrt{HP_{friendly - i}(initial)})^2 \]

\[ HP_{friendly} (attack) = HP_{friendly (initial)} - \frac{HP_{enemy (initial)}}{2} \quad ------ (4.6) \]

In the above formulae, the \( Time_{join} \) and \( Time_{attack} \) denote the times taken to complete a join and attack actions respectively, \( Time_{to-move} \) denotes the time taken to move from the current location to the destination, and \( Time_{to-attack} \) denote the time taken to finish an attack once the 2 attacking groups are adjacent to each other. \( HP_{friendly} \) and \( HP_{enemy} \) denote the effective hit points (health) of the friendly and enemy groups respectively. The term within the brackets following the \( HP \) indicates the state of the group under consideration; the term \( (initial) \) indicates the state before a join/attack action is carried out, whereas the terms \( (join) \) or \( (attack) \) indicate the state after a join or attack action have been completed.

The parameters in estimating the result of join actions are obtained easily, since these actions are more or less deterministic. The time taken for the action can easily be calculated based on the speed of the units involved and the distance to be covered; and the strength of the resultant group would be calculated as in formula (4.4). The simulation for attack actions is little more complex, since the result of a battle between two groups of units has some extent of randomness associated with it. Hence the parameters for estimating the time taken and the hit points left at the end of a battle are obtained by observing a number of individual battles with varying group sizes.
The result (win/loss) of an attack action involving a friendly group and an enemy group is simulated based on the difference in the effective hit points of both the groups. If the sign of the effective hit points left is positive, it indicates a win and a loss otherwise. But as described in case-ii(b) of section 4.4.2, there is a chance of multiple friendly groups being pitted against a single enemy group as part of an aggregated attack action. In this case, the simulation has to be done in stages because the various friendly groups are likely to be at varying distances from the enemy group, hence by the time one group reaches the enemy, there would be other group(s) already attacking it. Hence there is a need of being able to simulate partial battles as well as complete battles, as part of the Monte Carlo simulations. The pseudo code for the logic followed in such scenarios is given in Figure 4.2.

To get the partial loss of hit points in a battle, we do a linear interpolation of the battle result based on the time available for battle. It should also be noted that, during the UCT tree construction, in order to discourage losing battles, we give a high negative reward for leaf nodes that result in an ultimate defeat in the game.
**Pseudo-code:**

1: order the various friendly groups (involved in the composite attack action) based on increasing distance from the enemy group.

2: calculate the time $t_1$ taken by the first friendly group to reach the enemy group, and advance the coordinates of all friendly groups for time $t_1$

3: loop for each friendly group:

4: calculate the time $t_{(i+1)}$ taken by the $(i+1)^{th}$ friendly group to reach the enemy group

5: do a partial simulation of a battle between the currently attacking friendly group and the enemy group for this time $t_{(i+1)}$

6: if the partial battle results in a win or draw (for the friendly group) then

7: stop the simulation and mark the attack action as finished.

8: compute the time taken and update the reduced hit points for the attacking friendly group.

9: else if the partial battle results in a loss (for the friendly group) then

10: update the reduced hit points for the enemy group and continue.

11: else if the partial battle results in an incomplete result then

12: update the reduced hit points for both the attacking friendly group and the enemy group

13: merge the attacking friendly group with the $(i+1)^{th}$ friendly group (ready for battle in the next iteration)

14: continue the loop until it results in a complete destruction of the enemy group or until all friendly groups (that are part of the attack action) have been eliminated.

Figure 4.2 Pseudo code for Monte-Carlo simulations for a composite attack action.
5 Experiments and Results

In this chapter we first present the experimental setup and a brief description about the different scenarios that we tested. All experiments were conducted in the game of Wargus, which is run on top of the open-source Stratagus RTS engine. Next we describe the different baseline planners that were used to test our scenarios, the results from which are used to compare against those from our planner (optimized for different objective functions). Finally we present the results and their analysis.

5.1 Experimental Setup

We created 16 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 2.1, Figure 2.2 and Figure 2.3 show screen shots of various stages of an attack action during one of these scenarios. In each of the screenshot, the upper-left corner depicts an abstract view of the full map showing the locations of 2 friendly and 2 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 taking place. In order to simplify the simulation of actions in this initial investigation, we have restricted all of the scenarios to utilize a single type of unit known as a footman.

The scenarios vary the number of friendly and enemy units from 10 to 20 per side and the number of initial groups (based on proximity) from 2 to 5. Table 5.1 gives the details about the various scenarios that are used for conducting the experiments. The naming convention followed for each scenario name is <number-of-friendly-groups>vs<number-of-enemy-groups>, optionally followed by an index to differentiate multiple scenarios with the same combination of friendly and enemy groups. Even with the same number of friendly and enemy groups, variations in their positions on the map will require different kind of strategies to be employed for winning. It can be observed from the composition of the friendly and enemy groups (columns 4 and 6), that 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.

<table>
<thead>
<tr>
<th>#</th>
<th>Scenario Name</th>
<th># of friendly groups</th>
<th>Friendly groups composition</th>
<th># of enemy groups</th>
<th>Enemy groups composition</th>
<th># of possible ‘Join’ actions</th>
<th># of possible ‘Attack’ actions</th>
<th>Total # of possible actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2vs2</td>
<td>2</td>
<td>{6,6}</td>
<td>2</td>
<td>{5,5}</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3vs2</td>
<td>3</td>
<td>{6,2,4}</td>
<td>2</td>
<td>{5,5}</td>
<td>3</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>4vs2_1</td>
<td>4</td>
<td>{2,4,2,4}</td>
<td>2</td>
<td>{5,5}</td>
<td>6</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>4vs2_2</td>
<td>4</td>
<td>{2,4,2,4}</td>
<td>2</td>
<td>{5,5}</td>
<td>6</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>4vs2_3</td>
<td>4</td>
<td>{2,4,2,4}</td>
<td>2</td>
<td>{5,5}</td>
<td>6</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>4vs2_4</td>
<td>4</td>
<td>{2,4,2,4}</td>
<td>2</td>
<td>{5,5}</td>
<td>6</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>4vs2_5</td>
<td>4</td>
<td>{2,4,2,4}</td>
<td>2</td>
<td>{5,5}</td>
<td>6</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>4vs2_6</td>
<td>4</td>
<td>{2,4,2,4}</td>
<td>2</td>
<td>{5,5}</td>
<td>6</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>4vs2_7</td>
<td>4</td>
<td>{3,3,6,4}</td>
<td>2</td>
<td>{5,9}</td>
<td>6</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>4vs2_8</td>
<td>4</td>
<td>{3,3,3,6}</td>
<td>2</td>
<td>{5,8}</td>
<td>6</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>11</td>
<td>2vs4_1</td>
<td>2</td>
<td>{9,9}</td>
<td>4</td>
<td>{4,5,5,4}</td>
<td>1</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>12</td>
<td>2vs4_2</td>
<td>2</td>
<td>{9,9}</td>
<td>4</td>
<td>{5,5,5}</td>
<td>1</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>13</td>
<td>2vs4_3</td>
<td>2</td>
<td>{9,9}</td>
<td>4</td>
<td>{5,5,5}</td>
<td>1</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>14</td>
<td>2vs5_1</td>
<td>2</td>
<td>{9,9}</td>
<td>5</td>
<td>{5,5,5,5}</td>
<td>1</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>15</td>
<td>2vs5_2</td>
<td>2</td>
<td>{10,10}</td>
<td>5</td>
<td>{5,5,5,5}</td>
<td>1</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>16</td>
<td>3vs4</td>
<td>3</td>
<td>{12,4,4}</td>
<td>4</td>
<td>{5,5,5,5}</td>
<td>3</td>
<td>12</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 5.1 Details of the different game scenarios

The column for the number of possible ‘Join’ actions in the Table 5.1, is populated based on the number of choices available to select 2 friendly groups out of the total number of available friendly groups, which is \( \binom{n_{\text{friendly}}}{2} \). The column for the number of possible ‘Attack’ actions, is populated based on the number of choices available to select any one of the friendly groups to attack one of the enemy groups, given by \( n_{\text{friendly}} \cdot n_{\text{enemy}} \). Accordingly, the total number of action choices is the sum of the 2 types of action choices, as given by the formula (4.5).
Note that, these choices vary as the game progresses since some groups may join together to form bigger groups, or some groups get eliminated as a result of battles. Also note that, as a result of the UCT algorithm, an aggregation of actions is carried out to facilitate concurrency, as explained earlier in Section 4.3. The scenario files, our Stratagus agent interface and the complete source code of our online-planner are available publicly at [21].

5.2 Planners

Our experiments consider two version of the UCT planner:

1) UCT(t) - which attempts to minimize the time, as measured by number of game cycles, to destroy the enemy.

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. Note that this objective that is used for evaluating a win may vary from scenario to scenario. For example, in scenarios when multiple enemy armies are attacking, the main criterion might be winning in the shortest time so as to be available for other battles; whereas in other scenarios like offensive campaigns, winning the battle with the maximum units left is more preferred; and towards the end of the game none of this might be a concern and just winning the battle would suffice.

We compare against 5 baseline planners:

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 group.

3) **Attack-Weakest**: which causes an idle group to attack the weakest enemy group and in the case of ties to select the closest among those.
4) **Stratagus-AI:** which controls the friendly units with the default Stratagus AI. For this planner, the game attributes have been modified to give infinite sight radius to all friendly units, so that they attack the enemy units according to the built-in AI of the game engine.

5) **Human:** the performance achieved by an experienced human player.

For all the first 4 planners above, actions are assigned (according to the type of the planner) whenever a group becomes idle. Unless otherwise noted, we used 5000 rollout trajectories for the UCT planner.

### 5.3 Results and Analysis

We ran all of the planners on all 16 benchmarks and measured both the time (game cycles) required to defeat the enemy, as well as the effective 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 to account randomness. Figure 5.1 and Figure 5.2 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 friendly and enemy groups. For example, 4vs2_1 is the first scenario that involves 4 friendly groups and 2 enemy groups on the initial map. In scenarios, where a planner does not win a game the hit points are recorded as 0 in Figure 5.2 and there is no point plotted for the time metric in Figure 5.1. Hence, some breaks can be observed in the plots of Figure 5.1.
Figure 5.1 Time results for UCT(t) and baselines.

Figure 5.2 Hit point results for UCT(t) and baselines.
Figure 5.3 Time results for UCT(hp) and baselines.

Figure 5.4 Hit point results for UCT(hp) and baselines.
From the first 4 graphs above, we notice first that UCT(t) is the only planner besides the human to win all of the scenarios. This indicates that by utilizing a model-based approach our planner is able to avoid many of the mistakes made by the other planners which result in defeat. Furthermore, we see from Figure 5.1 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 5.2, 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. Notice that the human player has great difficulty trying to optimize the time objective. The primary reason for this is the difficulty in quickly controlling the units using the Stratagus user interface, since the game requires assigning multiple actions at different locations of the map as and when groups finish their currently executing actions, and these assignments very often are required to be made in parallel. This shows the potential usefulness for incorporating an automated planner such as ours into the user interface.
Figure 5.3 and Figure 5.4 are similar to the previous two figures but plot results for UCT(hp) rather than UCT(t) \( i.e., \) with maximizing the hit points at game completion as the optimization objective. We see from Figure 5.4, 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 5.2, 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. The performance of UCT(hp) is quite poor in terms of completion time, which should be expected. In particular, the best way to optimize hit points is to take time initially to form a very large group and then to attack the enemy groups sequentially. However, again in this case, we see that UCT(hp) is able to significantly improve on the completion time compared to the human player. Overall we see that for both metrics our UCT 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 5.5 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 the optimum with 5000 rollouts. Increasing the number of rollouts beyond 5000 for UCT(t) did not produce significant improvements. This shows that 5000 rollouts are apparently sufficient for the scale of problems that we are dealing with, which are reflective of what might arise in an actual game.

In terms of computation time, our current prototype implementation 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 rich features of tactical planning. Furthermore, prior Monte Carlo methods required significant human knowledge to design a good evaluation function. Our main contribution is to show that UCT, which requires no such human knowledge, is a very promising approach for assault planning in real-time strategy games. Across a set of 16 scenarios in the game of Wargus and for two objective functions, our UCT-based planner is a top performer compared to a variety of baselines and a human player. A significant advantage of using UCT is the ability to produce different plans that optimize on different winning criterion, which can be quite useful in a domain like RTS games, where different stages in the game demand different types of tactics to be used. Furthermore it was the only planner among our set to find winning strategies in all of the scenarios, apart from the expert human player. Our planner could potentially be used as an important component of an overall planner for RTS games.

In the future, the online planner could be extended in several ways to increase the performance and to produce better results.

- Mainly on the engineering side, we believe that the data structures that are used in the implementation could be optimized to arrive at truly real-time performance.

- Machine learning techniques can be used so as to learn improved simulation models of both friendly and enemy movements and their interactions. This will make it easier to evaluate our planner in more complex scenarios that involve multiple types of units and more sophisticated adversaries (e.g. humans). This will also allow our planner to be customized more easily to different domains.

- We have worked on a simplified version of the tactical battles where the enemy units are mainly reactive and the primary task is to decide ways of
attacking them effectively (tactical assault). Since UCT is an online planning framework, it should be able to deal with proactive enemy units as well by considering a wider range of interesting time points in the game.

- Finally, our planner can be integrated into an overall AI architecture for complete RTS game play.

The complete source code along with some basic documentation is available as open source [21].
Bibliography


