Adjutant Bot: An Evaluation of Unit Micromanagement Tactics

Nicholas Bowen
Department of EECS
University of Central Florida
Orlando, Florida USA
Email: nicholas.bowen@knights.ucf.edu

Jonathan Todd
Department of EECS
University of Central Florida
Orlando, Florida USA
Email: jonathan.todd@knights.ucf.edu

Gita Sukthankar
Department of EECS
University of Central Florida
Orlando, Florida USA
Email: gitars@eecs.ucf.edu

***Abstract*—Constructing an effective real-time strategy bot requires multiple interlocking elements including a well-designed architecture, efficient build order, and good strategic and tactical decision-making. However even when the bot's high-level strategy and resource allocation is sound, poor battlefield tactics can result in unnecessary losses. This paper focuses on the problem of avoiding troop loss by identifying good tactical groupings. Banding separated units together using UCT (Upper Confidence bounds applied to Trees) along with a learned reward model outperforms grouping heuristics at winning battles while preserving resources. This paper describes our findings in the context of the *Adjutant* bot design which won the best Newcomer honor at CIG 2012 and is the basis for our 2013 entry.**

I. Introduction

An oft-quoted military proverb "for want of a nail the shoe was lost" illustrates how small choices can have large consequences [1]. The morale of the story is that cumulative small losses lead inexorably to resource deficits that can ultimately result in larger defeats due to troop shortages. Hence avoiding superfluous small losses can be as important to designing a real-time strategy bot as rapidly constructing units or anticipating the opponent's strategy. One common cause of unnecessary losses in real-time strategy games is poor spatial allocation of units; separated units can be eliminated more easily by enemies and contribute less toward winning the battle.

In this paper, we examine the impact of different tactical grouping approaches on post-battle resource value and demonstrate an application of the UCT (Upper Confidence bounds applied to Trees) algorithm [2] for identifying good tactical groupings in real-time. Due to the high variability of RTS games, we do not attempt to learn any specific action plans offline. Instead, our bot formulates decisions on the fly so that we are able to adapt to actions of the opponent as they happen. For turn-based games like cards or Go [3], it is obvious when an agent must decide on its next action. However in continuous RTS games like StarCraft, it is not feasible to re-evaluate our current actions at every frame of game execution. Instead, we identify key points in the game where troops are under-utilized and focus the state exploration on these time steps. The reward model used by the UCT rollouts is learned during an offline data collection phase. We demonstrate that the tactical groupings discovered with UCT result in higher post-battle resource values, measured in terms of the health and the construction costs of the surviving troops, than commonly used allocation heuristics such as attacking the closest or most isolated group of enemies.

The next section provides a high-level description of our domain, bot development for real-time strategy games. Section III provides an overview of related work in the area of StarCraft and RTS bot development. The design of the *Adjutant* bot is described in Section IV. Section V describes our approach for learning grouping tactics, and Section VI presents a comprehensive evaluation of the unit micromanagement and overall bot performance.

II. Background

This paper describes the design of the *Adjutant* bot, which is designed to compete against other opponents in the real-time strategy (RTS) game StarCraft: Brood War. Players in RTS games operate in a simultaneous, rather than turn-based fashion, which poses specialized metacontrol challenges for AI systems, since there is a definite opportunity cost incurred by delaying action in favor of thinking. This time restriction favors the development of lightweight bot architectures capable of displaying bounded rationality [4] on complex problems. The cost of adding additional components to the reasoning process needs to be weighed carefully against the potential benefits of optimizing, rather than satisficing, decision-making.

The aim of most RTS games is to gather resources in order to construct buildings and military units which are then used to defeat the opponent's military units. Typically, RTS games have multiple types of specialized units such as Gatherer Units, Worker Units, and Military Units. Gatherer Units collect the resources needed to create other units and additional buildings. Worker Units are generally the units that physically construct additional buildings, while the Military Units are used for offensive and defensive strategies. By fabricating additional buildings and conducting research, the player can progress to more advanced technologies. These technological advancements can be represented on a tech-tree to show the prerequisites needed to achieve a particular technology. Early decisions on building and resource gathering limit the potential military strategies available to the player.

In general, opponents can begin a game with one of following three strategies: Gathering, Economical, or Rushing. In a Gathering Strategy, the player will attempt to gather as many resources as possible while producing military units because these resources and units will be needed throughout the game.

978-1-4673-5311-3/13/$31.00 ©2013 IEEE

In an Economical Strategy a player will forgo producing military units in an attempt to create more buildings and progress down the tech-tree faster. Finally, in a Rush Strategy the player will attempt to create a small military force and send everything they have at the opponent early in the game in an effort to take them off-guard. Each of these strategies can work, but only if the situation is right for it. The game is ultimately won when one player defeats their opponent by destroying all their units/buildings or by completing some predefined goal first. Uncertainty is introduced into the decision-making process by shrouding the map in a fog of war which prevents the player from seeing objects and events not within sight range of a friendly unit.

Our research centers on bot development for StarCraft: Brood War, a science-fiction themed RTS which was originally created by Blizzard Entertainment. Since its release, StarCraft has become a very popular title and one of the most competitive RTS games, with many international competitions such as the World Cyber Games, Electronic Sports World Cup, and Blizzard's BlizzCon challenge. In addition to the human vs. human competitions, there are bot vs. bot and also man-machine competitions that take place annually at AI conferences. StarCraft contains three races (Protoss, Terran, and Zerg) that are significantly different in terms of units, tech-trees and game playing styles. *Adjutant* was created for playing the Terran faction and finished in 4th place (out of 10 bots) in the CIG 2012 StarCraft RTS competition and won the best Newcomer award for the highest placing first-time entry [5].

III. Related Work

Each StarCraft bot must make a set of interdependent decisions regarding resource gathering, building/unit fabrication, map exploration, opponent modeling, strategic planning, and tactical maneuvering. In this paper, we specifically analyze the problem of avoiding material loss through improved unit micromanagement since it can be evaluated in a decoupled fashion from the rest of the bot's design. Our proposed "waste not, want not" battle approach is applicable to bots playing any high-level strategy and can be implemented without substantial changes to other elements of a bot's architecture.

However, there are several other important elements to designing an effective StarCraft bot. Since early construction decisions constrain all military decisions, identifying build orders for rapidly constructing the desired units from available resources is critical to a bot's success. Churchill and Buro developed a set of heuristics and abstractions for approximating the build order planning problem, some of which are used in the 2nd place *UAlbertaBot* bot [6]. It can be equally valuable to predict what the opponent is building since certain units are especially good at countering the abilities of other units. The 7th place finisher *BroodwarBotQ* makes extensive use of opponent modeling; the authors of that bot demonstrated an unsupervised Bayesian approach for tech-tree prediction [7] and also a similar technique for predicting the opponent's general opening strategy [8]. Even with perfect knowledge of the opponent's intentions, coupling opponent modeling with planning can be challenging. Weber et al. illustrate how reactive planning can be used to respond to unexpected game events using a Goal-Driven Autonomy paradigm [9].

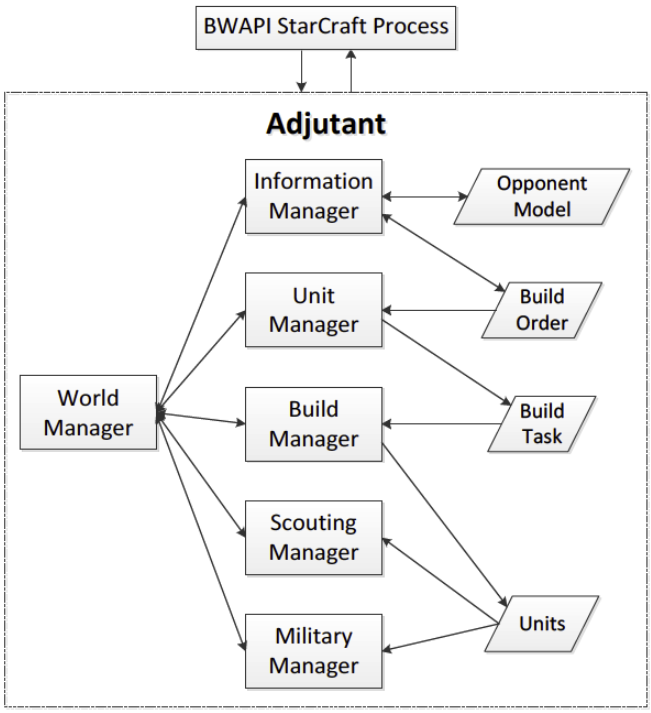

Fig. 1. Architecture of *Adjutant* bot, CIG 2012 best Newcomer

At the tactical level, kiting, utilizing a unit's mobility and firing range to inflict damage without taking damage in return, is a specialized technique for avoiding unnecessary losses. The NOVA bot, that took 5th place in the CIG competition, uses influence maps to kite enemy units [10]. Grouping is more generally applicable than kiting, which cannot always be performed, depending on the opposing units and the terrain. Reinforcement learning algorithms, such as Q-learning and SARSA, have been applied to learning kiting policies for a single unit but it is unclear whether it is possible to scale this single-unit approach to large StarCraft battles [11].

Our work is based on Balla and Fern's approach to learning tactical decision-making for the Wargus RTS [12]. Rather than trying to reduce risk by retreating as is done in kiting, idle units have the option of joining with other groups or attacking directly. Our work extends on the previous Wargus work by introducing different types of units in the test scenarios. This provides another layer of complexity that we confront by using offline learning to learn a reward model for confrontations between different unit types. Although we use a group abstraction for decision-making, confrontations are tracked and simulated at the unit level within UCT rollouts. Furthermore, our test cases include computer opponents that are controlled by the built-in AI unlike the simplified stationary bots used in previous work.

IV. Bot Design

Figure 1 shows the architecture of our *Adjutant* bot which utilizes the preexisting Broodwar API and can be downloaded at http://code.google.com/p/adjutantbot/.

Adjutant employs several modules to handle different aspects of the decision-making process: World Manager, In-

formation Manager, Unit Manager, Build Manager, Scouting Manager, Military Manager. The output of these modules is an opponent model, build order, the current build task, and unit tactics. The general philosophy behind *Adjutant*'s design is as follows. First, a predetermined strategy is used to decide on unit creation and building construction. As needed, adjustments are made in response to the opponent's playing style using a set of reactive plans. Our bot scouts aggressively throughout the game and also utilizes a Terran-specific ability to reveal a specified area of the map.

We maintain an opponent forces model that helps determine what composition of units our agent should produce. *Adjutant* is pre-loaded with an empirically-determined matrix that outlines one-to-one unit counters and the associated strength. For instance, the Terran unit Firebat is proficient at fighting the Zerg unit Zergling, so in the matrix, the Zergling row would list Firebat with a high countering strength. Although a one-to-one relationship is an imperfect model since it does not represent the synergies between different allied units, this simplified structure is used in the *Adjutant* agent when deciding which type of unit to create.

We examine the known enemy forces that our forces have encountered and choose to produce the unit types that best counter the enemy units. Some basic planning techniques are used to prioritize a given set of actions and a goal; the bot then follows a sequence of actions in an attempt to reach the goal state from the current state. This plan affects the strategies employed and the type of units created throughout the game.

The aspect of timing when to make offensive maneuvers plays a key role in the game of StarCraft. At the strategic level, we follow the assumption that our agent should only attack when we believe our army outnumbers the opponent's army. To get an estimate of the value of each army, we sum the total mineral and gas cost of each of our units and use the known enemy forces model to estimate their current value. If our agent's army value is significantly higher than the opponents, then we launch an offensive strike towards the enemy's base. This method not only aids in attacking, but also helps when knowing when to retreat. For example, if our army value has fallen too low compared to the enemy, our units will retreat in order to build up new forces.

The Military Manager handles all military unit activities, including unit micromanagement which is the focus of this paper. Based on the information received from the World Manager and Information Manager, the Military Manager determines what strategy to perform throughout the game. Depending on the situation, this component will perform either an attack or a counter-offensive against the enemy units. When enemy units are in close proximity to our military units, low-level logic is employed to target specific units to attack or retreat if necessary. Military decisions are updated every frame to help determine the best up to the minute action to take.

V. Learning Grouping Tactics

Based on our initial development experiences, we concluded that the grouping tactics encoded in the low-level logic of the Military Manager had an impact on unit attrition rates. Since these could be modified without substantial changes to the other bot components, we experimented with learning this aspect of the tactical model using UCT.

A. UCT

UCT (Upper Confidence bounds applied to Trees) is an algorithm that was first suggested in [2] as a more efficient alternative in Monte Carlo planning. The idea requires the problem to be represented by a set of states with associated actions to take. By guiding the sampling of possible actions towards the most promising end node, UCT can discover a high-reward action for a given state. In our case, we use UCT to help decide between the possible join and attack actions to assign to idle unit groups. Whenever we reach a decision point, we generate a tree using the UCT algorithm starting with the root node and game state.

UCT typically requires a large amount of rollouts, or action trajectories, to learn the best policy. Because of the incurred overhead cost, it is usually either performed offline with a large set of rollouts, but rollouts can be conducted with simpler simulation models for real-time performance [13]. UCT is designed to select the action with the highest known reward much of the time, but also awards an exploration bonus based on the number of node and action visits.

B. Representation

The basic idea behind our implementation is to combine UCT with offline training and simulation to estimate the best way to maneuver units in a real-time strategy game. The dynamic nature of the game genre and the volatile nature of the opponent make this an interesting problem. UCT implementations have primarily been applied to programs that have discrete states and actions. Adapting UCT to work in a real-time environment requires several alterations to the canonical operation. In our implementation, the information about the current game state is stored separately from the node in the UCT tree. The transitions between each node are completed using one of actions: `Join(G)` and `Attack(f,e)`, as suggested in [12]. The `Join(G)` action takes as input a set of groups, G, and causes all groups in G to move towards the centroid of their current locations. Once they are within a certain threshold of their target location, all groups are merged to form one large group. This is useful in RTS games because there is typically an advantage awarded to the group with more units in a confrontation. This action ensures that all groups are close together before launching an attack. The `Attack(f,e)` action requires one friendly group, f, and one enemy group, e, as input parameters. When executed, the friendly group will move towards and attack the enemy group until one of the groups no longer has any units left. Figure 2 shows an example game tree.

The edges between each node in the tree represent the different individual join and attack actions that can be taken. This means that it is likely that several edges will need to be traversed before all groups have been assigned actions. While this potentially leads to several paths through the tree, representing the same set of actions, it also avoids the exponential number of action combinations needed for each node otherwise. Furthermore, the guiding aspect of the UCT algorithm should cause rollouts to quickly converge beyond duplicate action sequences.

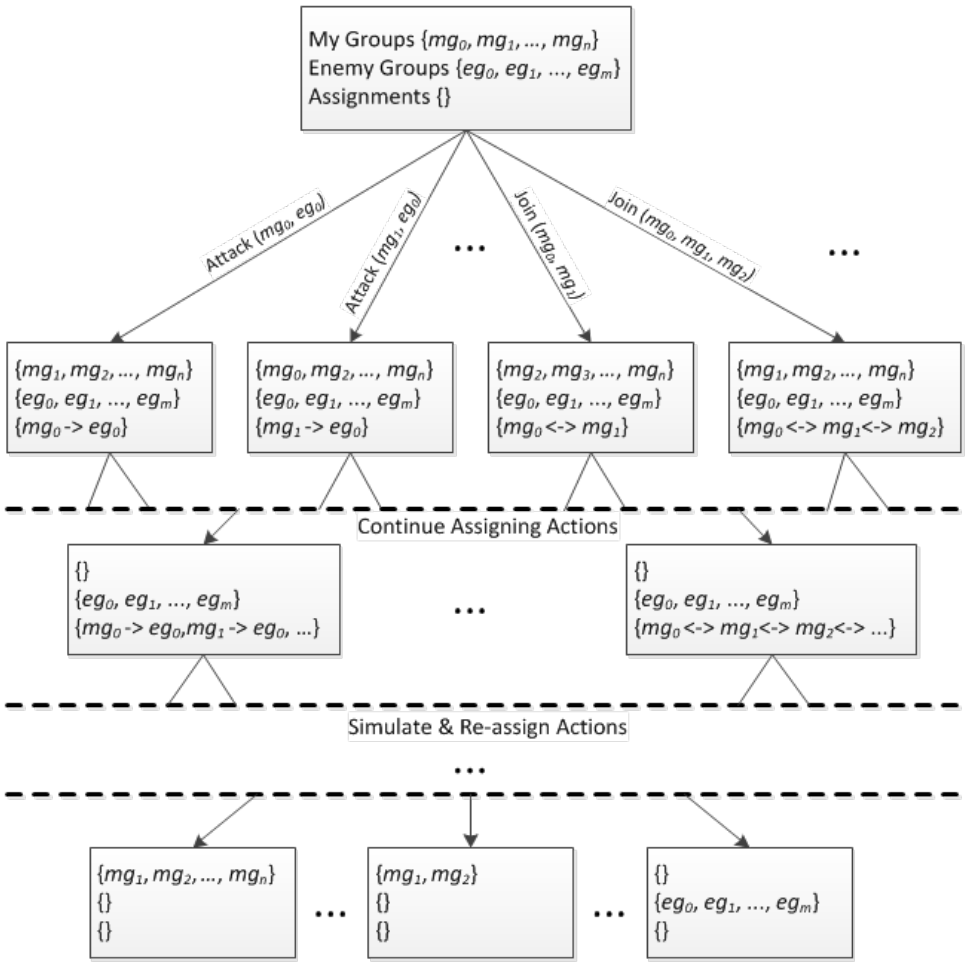

Fig. 2. An example UCT game tree. The root game state contains the current list of friendly and enemy groups. Attack and/or join actions are assigned until all friendly groups are utilized. At this point, simulation occurs. The remaining groups are reassigned actions, and simulation resumes until one side has no remaining units.

In RTS applications, there are a sufficiently large number variables affecting the outcome of the game that executing the same sequence actions from the same starting position does not always produce the same results. The design of the game state representation must account for this. We did this by decoupling the game state of the current rollout from the current UCT node. Each node keeps a list of all actions that could possibly be executed. When a decision needs to be made, the list of actions is filtered to only contain valid actions given the game state of the current rollout. This allows us to account for the multiple potential configurations produced by a single action while still retaining the information learned from previous runs.

As indicated above, the UCT node is simplified and only keeps track of all possible actions and the number of times it has been visited. The actions for each node are all generated based on the parameters of the game state at the root node. We can make this assumption in our case because executing actions can only reduce the number of possible actions. If one group attacks or joins another group, this reduces the total number of groups, and thus, the total number of possible actions.

The game state consists of all of the pertinent information about the units, their status, and their location on the board. Specifically, we track each friendly and enemy group. Within a group, we maintain a list of each associated unit, its type, and its health. We treat each group as a point particle and only keep track of its position which is initialized with the groups geometric center.

C. Implementation

Given a set of friendly units and enemy units, the first decision that must be made is how to partition these units into different groups. While an interesting problem in its own right, we do not attempt to tackle the issue in these experiments. Instead, the scenarios are designed to use a particular number of groups. We use a simple bottom-up agglomerative hierarchical clustering based on spatial position to form the groups for both friendly and enemy groups.

Because all players in the game act simultaneously, we may intend to complete a particular action but be interrupted by the opponent before we can finish. Because of this, we handle these interruptions by engaging the enemy group. For example, for the *Attack(f,e)* action, our group will move to confront the enemy group, e, but will stop and fight if intercepted by a different enemy group. The same applies to the *Join(G)* action as the set of groups are moving towards the centroid location.

When the groups have joined together, the new group is idle and triggers a new round of UCT runs to determine the action it will take. If one or more of the joining groups is destroyed before the join is complete, then the action is nullified and the remaining groups are set to be idle so that they can be assigned new actions. Whenever an enemy group has been destroyed, the attacking friendly group becomes idle and is assigned a new action. This continues until either all friendly groups or all enemy groups have been defeated.

After traversing an edge, the current game state is updated to represent that the groups involved in the chosen action are no longer idle. This does not actually progress in time in the game but is more of a bookkeeping step. The list of all possible actions is retrieved from the UCT node trimmed down to only actions involving idle groups. Because of the non-deterministic nature of our rollouts, each action is explored a set number of times before its value estimation is considered. After the initial exploration period, the action with the highest value is selected.

The learning update rule is as follows:

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

The term $Q(s,a)$ represents the current value estimate for state s and action a while $n(s)$ indicates the number of times the current state has been visited and $n(s,a)$ the number of times the action a has been selected in state s.

Once we have reached a state where all groups have been assigned actions, we use simulation to generate the rest of the tree. The simulation for the current run continues until either all of the enemy groups or friendly groups have been eliminated. At this leaf node, we calculate the final reward, R.

$$R = \sum_{f \in \text{friendly}} \frac{\text{fh}_f \text{rv}_f}{\text{ih}_f} - \sum_{e \in \text{enemy}} \frac{\text{fh}_e \text{rv}_e}{\text{ih}_e} \quad (1)$$

The values fh_u, ih_u, and rv_u represent unit u's final health, initial health, and resource value, respectively. Essentially, this sums the production cost of each remaining friendly unit weighted by its remaining health and subtracts the same calculation for the enemy units. This means that simulations in which the enemy wins result in a negative reward which helps UCT guide the search towards actions that are closer to victory.

After the reward is calculated, it is propagated back up the tree such that R is added to the total reward for each executed action and the visit counts of each node and action are incremented. The $Q(s,a)$ value for each node is simply the mean value of all of the rewards it has earned. This process continues for a set number of rollouts. Once a sufficient number of rollouts have been completed, we are ready to gather the actions with the best estimated values. Starting at the root node, we pick each action with the highest $Q(s,a)$ value until we reach a state where there are no idle friendly groups. This set of actions is assigned to each associated friendly group and the game continues.

D. Reward Model

The main purpose of the offline training is to learn a reward model for confrontations between different unit types to be used by the UCT rollout simulation. The reward model is stored in the form of ratios derived by testing every StarCraft unit type of the Terran Race against all of the unit types from each of the three races.

We created a Campaign Editor and Testbed map for learning the reward model. The Testbed design was primarily developed using the StarCraft Campaign Editor which is an application that allows a user to create or manipulate a StarCraft battle map. For this Testbed, a unique map was developed that divided a single large map space into smaller separate grid areas. These areas are designed to prevent the units from leaving their specified area, as well as prevent other units from entering another unit's area. The areas are also designed to allow the Testbed to be modified depending on the type of tests desired (such as higher/lower ground advantages). This design allows the Testbed to be very accurate, while allowing for a very flexible space to create different environments.

With the Testbed in place, the next stage was the creation of the Data Gatherer to retain relevant data. The Data Gatherer monitors each of the battles within the Testbed for changes. When one of the units in the battle is eliminated or the round timer ends, then the Data Gatherer would evaluate and compile data from the specific match-ups. The data that is retained for the simulation is the ratio between enemy health loss and friend health loss, which can be combined with the unit resource value to generate rollout rewards.

E. Simulation

Once we have reached a state in the UCT search where actions have been assigned to all friendly groups, we need to simulate execution of those actions. Because battles are simulated in a non-deterministic fashion, simulating the same actions can potentially produce different final results. This is how we try to model the uncertainty that is present when executing actions in the real game.

Groups are reduced to point particles represented by the list of units in the group and the centroid location. To model the actions of enemy groups, we assume each one will attack the closest friendly group. The simulation is executed at roughly two seconds of game time per step. During each step, all groups are first moved by interpolating between their current and target position using the average speed of the group. Next, if opposing groups are within a set distance of each other, an attack is simulated, and if joining groups are within a set distance of each other, the groups are combined into one large group at their center.

Attacks between opposing groups are simulated by randomly pairing a unit from each group against each other. This attempts to model the randomness of the army formation as it travels across the map. Based on the unit type information that we gained from offline training, we compute the damage that will be done to each unit. A slight advantage is also given to the unit that is part of the larger group. Battles continue until one group no longer has any units. Once one or more friendly groups become idle or we reach a leaf node, the simulation ends and returns to the UCT search.

Fig. 3. The 4v4 Mixed Map is an example evaluation scenario that commonly occurs within StarCraft in which there is a large battle with opposing groups intermixed.

VI. Evaluation

To evaluate the performance of our learned unit micromanagement model, we compare our UCT implementation against a variety of competing approaches in several different StarCraft scenarios. We also show the performance of the final version of our *Adjutant* bot which attempts to encode some of the micromanagement insights derived from the learned model within the MilitaryManager logic.

A. Setup

We created a variety of different combat scenarios that resemble different types of battles in StarCraft. To keep them relatively simple, all scenarios take place on flat ground with no obstacles. At the beginning of the round, the units for each group are generated at the same location. The round is over when one team has no units left. The metric used to evaluate each baseline is the same final resource value formula as described in [12]. Each scenario was run 50 times for every baseline.

The TeamWork map pits two friendly groups against two enemy groups of the exact same unit type. The greatest reward is provided when the teams work together to defeat the enemy. The 3v3RockPaperScissors scenario involves three groups on each team each composed of a different unit type. As the name suggests, it was designed so that certain friendly groups are more effective at battling particular enemy groups. In 3v4Surrounded, a variety of units are used on both sides and the friendly groups are outnumbered. The 4v2SplitUp map has roughly equal teams but splits the friendly units up into more groups. The 4v4Mixed setup simulates a large battle with opposing groups intermixed and a typical screen shot of this map can be seen in Figure 3. Finally, the 4v4SplitUp map sets up a large battle between opposing groups that are placed on opposite sides of the map.

For these test scenarios, the UCT search is run with 5000 rollouts and each action encountered is explored at least 10 times. Increasing the rollout and action budget beyond these values did not produce significantly better results.

TABLE I. Win rates against the StarCraft build-in AI

Race	Win Rate
Protoss	86%
Terran	73%
Zerg	90%

B. Benchmarks

We use a variety of benchmarks to evaluate the performance of UCT:

- Closest: for each friendly group, only attacks the closest enemy group
- Random: picks a random action from the possible join and attack actions
- Isolated: determines the enemy group that is farthest away from the others and sends all groups to attack it
- Adjutant: the final version of the low-level logic that tries to attack as one large tightly-packed group
- StarCraft AI: the built-in artificial intelligence that controls computer opponents in StarCraft

C. Results

We evaluated the efficacy of each of the six benchmarks by running them in the different game scenarios. Figure 5 shows the adjusted final resource value that was computed after averaging the scores that each benchmark received over 50 trials. The UCT algorithm was able to score higher than all of the other algorithms in 3 out of the 6 scenarios. In the other scenarios, it was able to score among the best 3 algorithms, only being outperformed by the *Adjutant* and Isolated benchmark. Of the scenarios that UCT scored the best in, two were designed to test specific tactics in battles on a smaller scale and the other was a large open map that allows the chosen actions to have a larger impact on the final outcome of the game. Of the scenarios where UCT scored the worst, two have very high standard deviations which could indicate that the randomness of the situation had an impact on the outcome.

Figure 4 shows the results from each baseline averaged across all of the different game scenarios that were tested. This data indicates that the UCT algorithm outperformed all other heuristics except for the final version of our StarCraft bot *Adjutant*. This indicates that our UCT implementation can outperform a wide variety of heuristics and the built-in game AI, but that the performance can be duplicated with a hand-tuned logic model.

Table I shows the win rates of our bot against the StarCraft build-in AI, based on 500 games on a selection of maps from the AIIDE Starcraft competition. Table II shows the win rates against other competitors, including the earlier version of Adjutant. Looking at our agent's results against other bots, we can see that our current implementation has improved over the first iteration, but there is still room for improvement. We were able to beat the previous version of our bot, Adjutant 1.0, 93% of the time. Our agent was also able to win the majority of games against many of the other opponents, but rarely defeated the most complex agents like Skynet and UAlbertaBot.

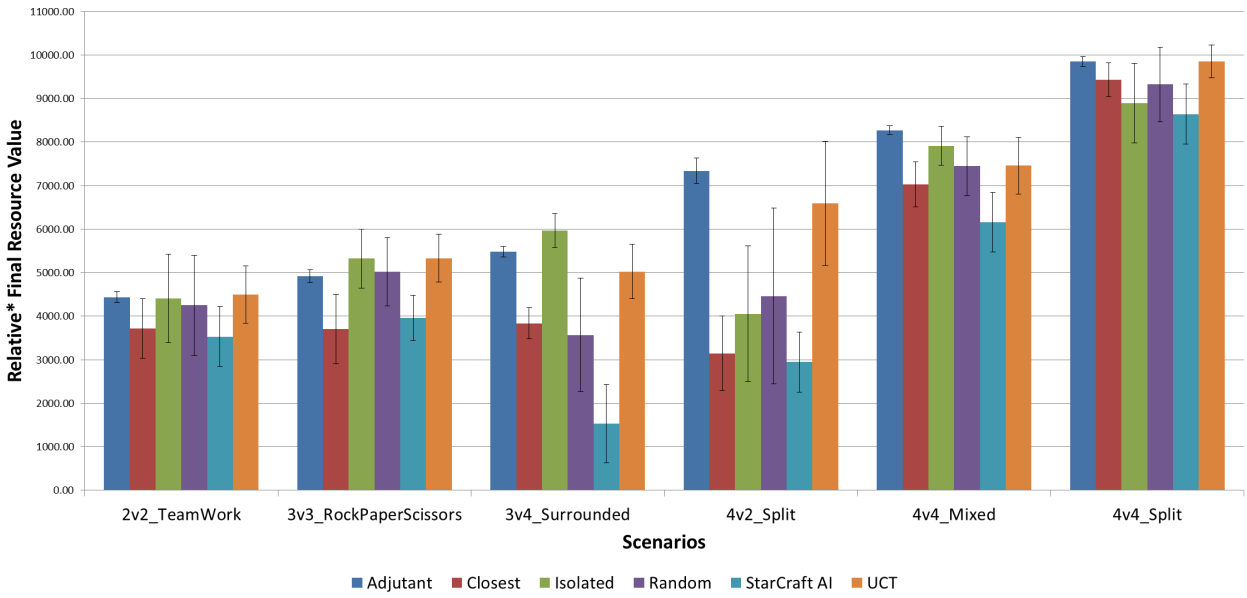

Fig. 5. Final resource value results for all baselines

TABLE II. WIN RATES AGAINST OTHER STARCRAFT COMPETITORS

Agent	Race	Win Rate
Adjutant1.0	Terran	93%
Aiur	Protoss	53%
bigbrother	Zerg	93%
Cromulent	Terran	93%
Nova	Terran	40%
Quorum	Terran	100%
Skynet	Protoss	0%
UAlbertaBot	Protoss	7%

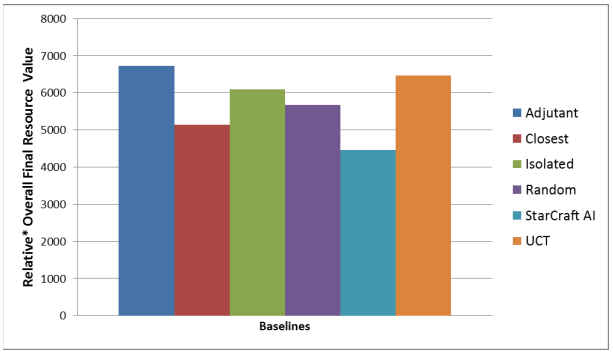

Fig. 4. Final resource value averaged across all scenarios. The grouping tactics learned by UCT and the hand-tuned final *Adjutant* logic outperform the heuristics and the StarCraft AI at reducing unnecessary attrition.

D. Conclusion and Future Work

While the results show that our hand-tuned *Adjutant* bot remains better, UCT is a consistently reliable performer that can outperform both the built-in StarCraft AI and a wide range of micromanagement heuristics. This means that in cases where hand-tuning is not an option, UCT is a good choice for unit micromanagement. For our purposes, it was more efficient to encode the core micromanagement insights directly within the *Adjutant* MilitaryManager logic rather than to run UCT during gameplay.

There are many interesting avenues for future work, with the most promising being unit grouping and targeting. The current Offline Training module gathers and retains enough data about specific unit types to yield insights as to what unit types would most likely win in a match-up. One idea would be to modify the possible actions in the UCT search to consider that data when selecting actions. This data could be used to create specialized unit groups and to target the specific enemy units that the friendly unit group has the greatest chance of defeating.

Another future work idea involves the better use of the battle maps. The Testbed is designed to allow freedom in testing for different situations. Learning how different unit types react while in different areas of the map could produce greater efficiency in the UCT runs enabling more accurate action selection during simulations. For example, units that are located in a chokepoint or on an elevated platform have a greater chance of defeating enemy units. In general, refining the UCT simulations in a data driven fashion promises to improve search performance. Interestingly, human players often perform poorly at micro-management tasks but salvage the situation through innovative high-level strategies, while most of the top bot competitors are quite good at this aspect of the competition. Hence this approach may not yield deep cognitive insights of how humans approach complex planning and resource management problems, but remains an important part of bot vs. bot competitions.

VII. ACKNOWLEDGMENTS

This research was supported in part by DARPA award D13AP00002 and NSF IIS-08451.

REFERENCES

[1] E. Lowe, "For want of a nail," *Analysis*, vol. 40, no. 1, pp. 50–52, 1980.

[2] L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo planning," in *European Conference on Machine Learning (ECML)*, 2006, pp. 282–293.

[3] N. Sturtevant, "An analysis of UCT in multi-player games," in *Proceedings of the Conference on Computers and Games*, 2008.

[4] H. Simon, "Bounded rationality and organizational learning," *Organization Science*, vol. 2, no. 1, pp. 125–134, 1991.

[5] T. Mahlmann and M. Preuss, "CIG 2012 Starcraft Competition," http://ls11-www.cs.uni-dortmund.de/rts-competition/starcraft-cig2012.

[6] D. Churchill and M. Buro, "Build order optimization in Starcraft," in *Proceedings of Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE)*, 2011, pp. 14–19.

[7] G. Synnaeve and P. Bessiere, "A Bayesian model for plan recognition in RTS games applied to Starcraft," in *Proceedings of Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE)*, 2011, pp. 79–84.

[8] ——, "A Bayesian model for opening prediction in RTS games with application to Starcraft," in *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 2011.

[9] B. Weber, M. Mateas, and A. Jhala, "Applying goal-driven autonomy to StarCraft," in *Proceedings of Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE)*, 2011, pp. 101–106.

[10] A. Uriarte and S. Ontanon, "Kiting in RTS games using influence maps," in *Proceedings of the AIIDE Workshop on AI in Adversarial Real-time Games*, 2012, pp. 31–36.

[11] S. Wender and I. Watson, "Applying reinforcement learning to small-scale combat in the real-time strategy game Broodwar," in *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG)*, 2012.

[12] R. Balla and A. Fern, "UCT for tactical assault planning in real-time strategy games," in *Proceedings of International Joint Conference on Artificial Intelligence*, 2009.

[13] K. Laviers and G. Sukthankar, "A real-time opponent modeling system for Rush football," in *Proceedings of the International Joint Conference on Artificial Intelligence*, Barcelona, Spain, July 2011, pp. 2476–2481.