Avoiding Parameter Overfitting in a Monte Carlo Approach to Real-Time Strategy Games

Jeffrey Lu*

Abstract— **This project examines overfitting and generalization error by applying Monte Carlo Tree Search (MCTS), a heuristic decision-making search algorithm, to the real-time strategy video game *StarCraft* and provides a basis for evaluating specific in-game scenarios of *StarCraft* based on their estimated generalization errors. How can we tell if we are overfitting the input parameters to specific scenarios in *StarCraft*? How well does a specific scenario of *StarCraft* generalize to the entire game?**

I. INTRODUCTION

Generating $22.41 billion in revenue in 2014, video games provide an entertaining and economically relevant source of exploration in the creation of ”smart” artificial intelligence (AI) [10]. For a video game AI, the typical end goal is to keep the player engaged and challenged over many replays. However, excepting social factors the continued popularity of playing against human players (multiplayer) in video games is perhaps a testament to the present superiority of human players as compared to AI players in terms of providing challenging yet entertaining gameplay. In contrast to human players, video game AIs are often comprised of predefined actions and states created through finite-state machines or behavior trees. Video game developers also generally value control of their AI creations over adaptive search algorithms in order to better balance gameplay towards targeted consumer groups. Nevertheless, the usage of Monte Carlo Tree Search (MCTS) in real-time strategy (RTS) games has been the topic of recent research, which has yielded promising results [1]. However, current research has only applied MCTS to specific cases of *StarCraft*, a popular RTS game released in 1998 with a notably active professional competition circuit in South Korea to this day. While MCTS may be readily tailored to very specific scenarios in a RTS game, those same parameters may perform significantly less well on the larger distribution of scenarios and game modes that comprise the typical commercial RTS game.

This research examines overfitting of MCTS to *StarCraft* by modulating the exploration parameter of ”Upper Confidence Bound 1 applied to trees” (UCT), a specific implementation of MCTS designed to balance exploration and exploitation in gameplay decisions [6]. The generalization error of overfitting is then examined and estimated with Rademacher complexity. These estimates provide a basis for evaluating the effectiveness of MCTS for specific cases of *StarCraft*.

*email: jslu@cs.brown.edu

II. WHAT IS A REAL-TIME STRATEGY?

A. Overview

A real-time strategy (RTS) video game is an electronic form of structured play that progresses continuously (real-time) rather than incrementally (turn-based) and that emphasizes skillful planning to achieve victory conditions against one or more opponents [2]. More simply, a RTS is a form of wargame. Players are generally presented with a playing field (map) with conditions of uncertainty (limited map view, i.e. ”fog of war”) and are tasked with managing mobile units, static structures, and economic resources in order to achieve one or more objectives, most commonly the destruction of enemy units and structures (Conquest). Other common objectives include capturing and/or holding a key position on the map (King of the Hill) or destroying a certain unit or structure (Regicide). As the units, structures, and resources available to achieve these objectives are usually limited, strategy is an important element of gameplay.

B. Gameplay

RTS gameplay usually consists of matches between two or more players and may be divided into early, middle, and late phases. For instance, Conquest matches typically start with players having a small initial force. In the early phase, players gather resources, scout the map, and build more units and structures. Players often follow a build order aimed at achieving a particular goal, such as mass production of a specific unit. If fighting occurs, it is with basic units and structures. Gameplay enters middle phase as surviving players build more advanced forces, secure their respective areas on the map, and launch more substantial attacks. By late phase, surviving players generally have access to the most advanced forces available, a large population of these forces, and better awareness of enemy forces and dispositions.

Of course, depending on player skill and scenario conditions a RTS match need not follow this progression. Players may, for instance, elect to start a match with unlimited resources and the most advanced forces available in order to accelerate gameplay. In the typical Conquest match as described, though, players must collect resources, build both offensive and defensive units and structures, locate strategic points and enemy positions on the map, and destroy enemy forces while preserving their own. These complicated demands require minute attention and maneuvering. In broad terms, players must balance between maintaining a base of operations and controlling their forces in battle against enemy forces, and as such gameplay can be divided into two areas:

- Macromanagement: Focuses on gathering resources, building structures for unit creation, and researching to improve structures, units, and abilities. For example, the player may try to secure resource-rich areas on the map as soon as possible in order to gain an economical advantage and deny the same to the enemy.
- Micromanagement: Focuses on addressing minute details to maximize the benefits of macromanagement actions. For example, a unit with low health may be moved away from the battlefield to ensure survival. Ineffective micromanagement can prove disastrous. The same unit with low health, for instance, can also lure poorly managed attackers into a trap.

Balancing effective macromanagement and micromanagement is a major aspect of both the appeal and challenge of playing RTS games. They are of significant interest in professional tournaments, where the fundamentals of macromanagement are arguably based heavily on the timing of micromanagement actions that usually aim to establish an economic advantage in the early stages of a match. For the scope of this project, MCTS will be applied to the micromanagement aspect of gameplay in the form of unit-versus-unit matchups in *StarCraft*.

III. STARCRAFT

Blizzard's *StarCraft* (1998) is a critically acclaimed entry into the RTS genre and considered by many to be the definitive RTS game [3]. Praised for its wide array of units, *StarCraft* is also renowned for its well-balanced gameplay, engaging military science-fiction storyline, and fast-paced action. The game is set in the 25th century in the Koprulu Sector of the Milky Way galaxy, where three species fight for dominance in a space opera where the only allies are enemies.

A. Races

StarCraft features three well-balanced yet distinct species (races) that encourage creative gameplay:

- Terran: a mechanized human society focused on versatile, average-cost units with strong ranged attributes
- Protoss: a technologically advanced alien race featuring durable but expensive units with energy shields
- Zerg: a hive-mind insectoid alien race producing fast, inexpensive, but relatively weak biological units

Fig. 1. Basic infantry units from left to right: Protoss Zealot, Zerg Zergling, Terran Marine

This project will examine the application of MCTS to *StarCraft* units from all three races in unit-versus-unit matchups.

IV. MONTE CARLO TREE SEARCH

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm for making optimal decisions in artificial intelligence problems, typically move planning in combinatoric games [4]. It combines the generality of random simulation with the precision of tree search. By using the Monte Carlo method, Remi Coulom described MCTS by outlining its application to game tree search [5]. The algorithm has shown promise to solving difficult problems, including application outside games.

A. Algorithm Overview

MCTS starts with generating a search tree according to possible playout outcomes. The process for selecting the best move (node) is as follows:

1. Begin at the root node and continue selecting optimal child nodes (see Node Selection) until a leaf node is reached.
2. If the leaf node does not end the game, create additional child nodes and select one to continue.
3. Run simulated playout for the child node, stopping when a result is reached.
4. Propagate backwards and update the move sequence with the results of the simulation.

B. Move Selection

Moves are selected by maximizing a defined reward or payout. Usually, a upper confidence bound (UCB) formula is utilized to calculate payouts:

$$v_i + C * \ln\left(\frac{N}{n_i}\right) \tag{1}$$

where v_i is the best guess of the reward of that node, n_i is the number of visits to that node, N is the number of visits to the node's parent, and C is the adjustable exploration parameter. The first part of the equation v_i represents exploitation and will be higher for nodes with high reward. The second part of the equation represents exploration and will be higher for nodes with few traversals.

When UCB is applied to MCTS, the resulting algorithm is termed *Upper Confidence Bound applied to trees*, or UCT [6].

V. IMPLEMENTATION

As this research examines overfitting of the exploration C-Value parameter to *StarCraft*, the UCT must be implemented specifically with *StarCraft* gameplay. Additionally, since MCTS requires simulated playouts of potential move choices, the *StarCraft* game mechanics should be simulated as closely as possible.

A. Brood War API

Brood War API (BWAPI) is a powerful framework for interacting with *StarCraft* [7]. BWAPI works by taking over the main game loop to allow retrieval of variables from and injection of custom AI scripts into *StarCraft's* game engine. On each frame of *StarCraft*, the UCT algorithm will execute many simulations and return the best move. The best move will then be executed by BWAPI.

B. SparCraft

UCT expands the game search tree by executing many simulated playouts to game completion. At the end of the simulated game, the results of each move are recorded so that better moves have a higher chance of selection in future playouts. Therefore, the *StarCraft* engine must also be simulated by UCT.

SparCraft is a combat simulation package for *StarCraft* utilizing a sophisticated frame-fast forwarding system that allows thousands of unit moves to be executed per second [8]. It was designed to provide a testing ground for AI algorithms by accurately modeling unit statistics and behaviors. However, due to the closed source nature of *StarCraft*, SparCraft only provides a best-guess approximation. Other limitations include a lack of collisions, fog-of-war, and ability to implement spell-casters and flying units.

An implementation of UCT was included in SparCraft. Potential moves are generated from the simulated game state. Each node holds potential move actions of all UCT-controlled units.

We can evaluate a node's result based on the current health (hp) and total damage per second (dps) of the UCT's current units at time t. The aggregate score for each is equal to:

$$S_t = \frac{\sum_{u \in U} hp(u) * dps(u) - \sum_{e \in E} hp(e) * dps(e)}{\sum_{u \in U} hp_0(u)} \quad (2)$$

where U is the set of current UCT units and E is the set of current enemy units. We normalize by dividing this by the starting health hp_0 of the UCT units.

Using the MCTS algorithm, if a node did not end the game, it would generate its children nodes. Move nodes alternated between UCT's units and prediction of the default AI's units. Figure 2 shows the alternating pattern where S is the node's score and V is the node's number of visits.

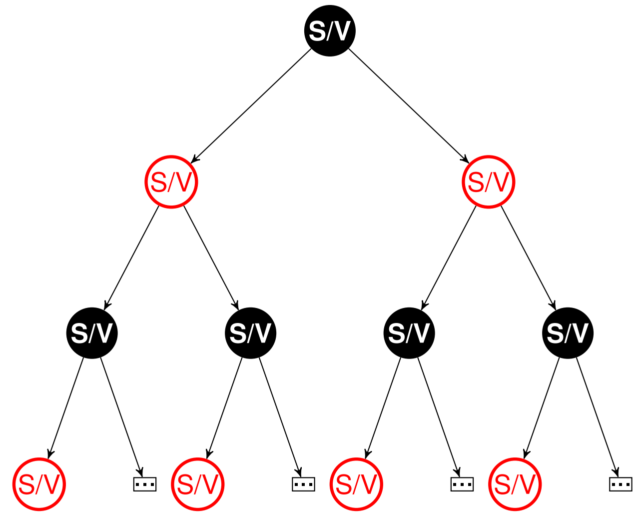

Fig. 2. The alternating move ordering allows for UCT to best react to the enemy's movements.

From the root node, UCT was set to traverse 5000 times with 5 children per move node. Because simulations are very time-consuming, if UCT did not complete in 10 milliseconds, the algorithm would return the best move node's actions. These actions are forwarded by BWAPI for execution in the *StarCraft* game engine.

VI. TESTING AND RESULTS

Two test sets were created to examine overgeneralization. Each individual matchup was composed of two different types of units pitted against each other, similar to a skirmish encountered in typical *StarCraft* gameplay. It was important to create balanced matchups, and part of this project's challenge was determining a fair balance between units of varying abilities and statistics. After each test set was created, a smaller training set of matchups was randomly selected from each distribution. UCT was run against the default *StarCraft* AI with the training set. The win ratio W for a given matchup is equal to:

$$W = \frac{U}{G} \quad (3)$$

where U represents the number of games won by UCT and G represents the number of total games played.

The best C-Value was selected based on W. Then, using that C-Value on the test set, UCT was again matched against the default AI, and the resulting difference in W gave the generalization error.

Fig. 3. A typical unit-versus-unit matchup: MCTS Terran Marines (left) vs. Default AI Zerg Hydralisks (right)

A. Test Set 1

This test set was composed of 9 matchups mirroring early-game unit battles. Since early game units have less advanced movement, it was more advantageous for UCT to exploit nodes with high reward rather than explore nodes with fewer visits. Therefore, the highest C-Value was set to 1.0, giving exploration and exploitation equal priority. The specific C-Values were: 0.075, 0.2, 0.4, 0.6, 0.8, 1.0.

Training sets of size 1 to size 3 were tested. Each C-Value was run 10 times on a matchup for a total of 90 runs for a single C-Value across all 9 matchups.

Each training set was tested 3 times with randomly selected matchup compositions. The average win ratio W across those 3 training distributions was calculated and compared to the average of their respective W when applied to the larger test set. The graph below summarizes the results.

Fig. 4. Results for Test Set 1

When training set size is equal to 1, UCT can be fitted to the C-Value so that W is almost 100 percent. However, when that same C-Value is tested against the test set, W decreases significantly. When we increase sample size, the C-Value will be fitted more accurately across the larger test set. Therefore, we see that the generalization error for between the training set W and test set W decrease to nearly zero.

B. Test Set 2

This test set was composed of 49 matchups mirroring mid-to-late-game unit battles. In order to balance the varying unit abilities, each matchup was composed of a greater number of units when compared to those of Test Set 1. On average, each side was given twice the number of units. In contrast to Test Set 1, Test Set 2 favored more exploration. Therefore the highest C-Value was set to 4.0, which places greater value on exploration versus exploitation. The specific C-Values were: 0.075, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0.

Training sets of size 1 to size 5 were tested. Each C-Value was run 10 times on a matchup for a total of 490 runs for a single C-Value across all 49 matchups.

Each training set was tested 5 times with randomly selected matchup compositions. The average W across those 5 training distributions was calculated and compared to the average of their respective W when applied to the larger test set. The graph below summarizes the results.

Fig. 5. Results for Test Set 2

For a very small training size, UCT can be fitted to the C-Value so that W is almost 100 percent. However, when that same C-Value is tested against the much larger test set, W decreases significantly, almost by 50 percent in extreme cases. When training size is increased, the C-Value will be fitted more accurately across the larger test set. The difference or generalization error in W between sample size and test set therefore decreases to nearly zero.

It is important to note that UCT only wins against the default AI about half of the time. This implies that UCT's performance decreases across the larger distribution of *StarCraft*'s colorful array of units.

VII. RADEMACHER COMPLEXITY

Generalization error is seen when fitting UCT to training sets. We can estimate this generalization error by bounding it with Rademacher complexity [9].

Rademacher complexity is based on using Rademacher variables σ. That is, given a training set $S = (f_1, f_2, f_3, ..., f_m)$ where each f is an individual matchup win ratio and m is the size of the training set, the empirical Rademacher complexity R_s of S is given by:

$$R_s = E_\sigma[\sup_{f \in S} \frac{1}{m} \sum_{i=1}^{m} \sigma_i f_i] \tag{4}$$

where $\sigma = -1$ or $\sigma = 1$ and $P(\sigma = -1) = 0.5$ or $P(\sigma = 1) = 0.5$.

This gives us on average how well an individual matchup correlates with random noise over the sample S. Because we want to measure the correlation of the matchup with respect to the overall distribution, we take the expected value of the matchup over all samples of size m:

$$R_m = E[R_s] \tag{5}$$

By using Rademacher complexity in the above equations, we can bound the generalization error for uniform convergence with confidence δ :

$$E_D \leq E_s + 2R_m + \sqrt{\frac{\ln 1/\delta}{2m}} \tag{6}$$

We apply the Rademacher complexity bounds to our two test sets. The Rademacher complexity for training set m was smoothed by averaging over several times for R_m training set size m.

Figure 6 shows the application of the bounds to Test Set 1. The shaded gray area represents the Rademacher bounds. The lower dotted line represents the worst-case generalization error. The test set performance lies between the bounds. Notably, the generalization bounds begin to tighten as training size increases.

Fig. 6. Test Set 1 results with generalization error bounds

Figure 7 shows the application of the bounds to Test Set 2. Again, the shaded gray area represents the Rademacher bounds. Similarly, the lower dotted line shows the worst-case generalization error. Because the largest training set size of Test Set 2 is greater than that of Test Set 1, the bounds are tighter. Again, the generalization bounds begin to tighten as training size increases.

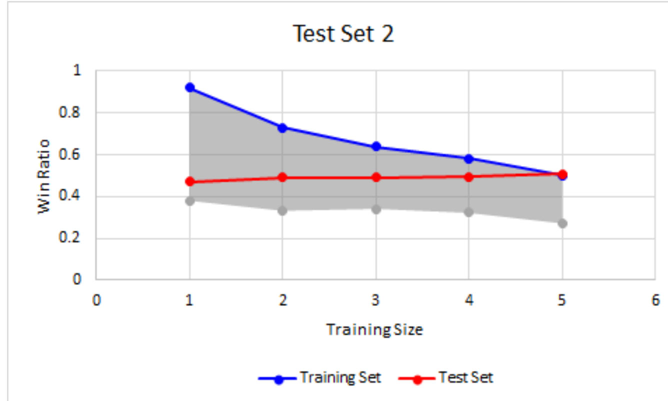

Fig. 7. Test Set 2 results with generalization error bounds

Comparing the two test sets, we can see some interesting trends. Both test sets show that C-Value can be fit such that in one specific matchup, the win ratio W can equal approximately 100 percent. In general, smaller training sets result in worse bounds than larger training sets.

The most important finding is that we can use these bounds to select a set of best individual matchups to generalize to the larger test set. The tighter the resulting calculated bound, the better the matchup generalizes. This provides a basis for fitting UCT to StarCraft.

VIII. CONCLUSION

In conclusion, we have shown that MCTS, specifically UCT, can be overfitted to certain matchups in *StarCraft* and that overfitting can be reduced with sufficient training size. We have provided a method of calculating bounds for the generalization error, thereby eliminating the need to exhaustively run parameters on the true test set population. This also provides a basis for evaluating a group of unit-versus-unit matchup scenarios in relation the general game of *StarCraft*.

IX. FUTURE WORK

Future work includes tuning various different parameters other than C-Value. The current implementation of UCT allows for tuning of the number of children per node. Nodes can also be selected based on the number of visits instead of best reward. Move ordering can also be adjusted based on the matchup.

Additionally, a dynamic equation can be created for calculating C-Value, and the parameters of that equation can be tuned. For example, a C-Value can be dynamically calculated by weighing the number of units, the unit type, and unit health into a single equation. Different weights can be used to tune for specific matchups. Lastly, this project did not explore flying units or spell-caster units in *StarCraft*. Future work should include different matchups with flying, ground, and spell-caster units. As this project has shown, the potential of MCTS to RTS games exists, and it is now just a matter of finding the correct scenarios.

ACKNOWLEDGMENT

A very special thanks to Professor Michael Littman for guiding and mentoring me, not to mention teaching me all the artificial intelligence knowledge I needed to complete this project.

Thanks to Professor Stefanie Tellex for reading and teaching me CS141, which piqued my interest in artificial intelligence.

Thanks to David Churchill for writing the awesome simulation package Sparcraft, and thanks to the BWAPI team for creating a system of interacting with *StarCraft*.

Shoutouts to my older brother Frederic Lu for spending many happy hours playing video games with me (*StarCraft* being one of them) and thanks to my Mom and Dad for buying them for the both of us.

REFERENCES

[1] W. Zhe, K. Q. Nguyen, R. Thawonmas, and F. Rinaldo. (2012, February 24). Using Monte-Carlo Planning for Micro-Management in StarCraft [Online]. Available: http://www.ice.ci.ritsumei.ac.jp/ ruck/PAP/gameonasia12-wang.pdf

[2] Bruce Geryk. (2008. March 31). A History of Real-Time Strategy Games [Online]. Available: http://web.archive.org/web/20110427052656/http://gamespot.com/gamespot/features/all/real_time/

[3] Blizzard Entertainment. (2008). StarCraft's 10-Year Anniversary: A Retrospective [Online]. Available: http://web.archive.org/web/20080402134120/http://www.blizzard.com/us/press/10-years-starcraft.html

[4] C. Browne. (2010). Monte Carlo Tree Search [Online]. Available: http://mcts.ai/?q=mcts

[5] R. Coulom. (2006). Efficient selectivity and backup operators in Monte-Carlo tree search [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.81.6817

[6] L. Kocsis and C. Szepesvári. (2006). Bandit based Monte-Carlo Planning [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.102.1296

[7] A. Heinerm. (2008). BWAPI [Online]. Available: https://code.google.com/p/bwapi/

[8] D. Churchill. (2012). SparCraft [Online]. Available: https://code.google.com/p/sparcraft/

[9] R. Schapire. (2013. March 5). Rademacher Complexity [Online]. Available: https://www.cs.princeton.edu/courses/archive/spring13/cos511/scribe_notes/0305.pdf

[10] Entertainment Software Association. (2014). Essential Facts About the Computer and Video Game Industry [Online]. Available: http://www.theesa.com/wp-content/uploads/2014/10/ESA_EF_2014.pdf