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

Jeffrey Lu*

Abstract—This project examines overfitting and generalization error with Monte Carlo Tree Search in the real-time strategy video game StarCraft and provides a basis for evaluating the specific scenarios of StarCraft based on their estimated generalization error. 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

A multi-billion dollar industry, video games provide a continuing source of exploration in creating "smart" intelligent AI that will keep the player engaged and challenged. Nevertheless, many game artificial intelligences are comprised of predefined actions and states created through finite-state machines or behavior trees. Developers value control of their AI creations over adaptive search algorithms in order to better balance gameplay. Specifically, 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. It is very possible to tune MCTS to very specific scenarios in an RTS game, but those same parameters may not perform very well on the larger distribution of scenarios and game modes that compose the typical commercial RTS game.

This research examines overfitting of MCTS to the RTS video game StarCraft by tuning the exploration parameter of UCT, a specific implementation of MCTS. Then, the generalization error of overfitting is examined and estimated with Rademacher complexity. These estimates provide a basis for evaluating specific cases of StarCraft.

II. WHAT IS A REAL-TIME STRATEGY?

A. Overview

A real-time strategy (RTS) is a genre of strategy games that progresses continuously rather than in a series of turns [2]. 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 units and structures under their control in order to achieve one or more objectives, most commonly the destruction of enemy units and structures (Conquest). Other common objectives include holding a position on the map (King of the Hill) or destroying a certain unit or structure (Regicide). As the available resources to achieve these objectives are usually limited, strategy is an important element of gameplay.

B. Gameplay

RTS matches typically start by gathering resources and building structures to supply an army. Players create offensive units and build defensive structures. Throughout the match, players order their armies to attack each other and take strategically important positions on the map, such as additional resources. These skirmish matchups require minute attention and maneuvering. Players must balance between maintaining a base of operations and controlling their armies in battle. Therefore, 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.

In professional tournaments, there are many agreed upon rules for macromanagement. In fact, the fundamentals of macromanagement are arguably completely based on timing of resource gathering speeds in early stages of a game. For the scope of this project, MCTS will be applied to the micromanagement aspect of gameplay. We will test MCTS to across an variety of unit matchups in StarCraft.

III. STARCraft

Blizzard’s StarCraft (1998) is a revolutionary entry into the RTS genre and is considered by many to be the definitive RTS game [3]. Praised for its wide array of units, StarCraft is renowned for its well balanced gameplay, engaging game storyline, and fast-paced action. The game’s synopsis pits three distinct races against each other in a space opera where the only allies are enemies.

A. Races

From a gameplay perspective, StarCraft features three well-balanced races:

- Terran: a mechanized human society focused on average-cost units with strong ranged attributes
- Protoss: a technologically advanced alien race featuring expensive and durable units
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:

\[ v_i + C \cdot \ln \left( \frac{N}{n_i} \right) \] (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 exploration 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

This research examines overfitting of the exploration C-Value parameter to StarCraft. In order to achieve this, 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 close as possible.

A. Brood War API

Brood War API (BWAPI) is a powerful framework that provides a way for interacting with StarCraft [7]. BWAPI works by taking over the main game loop. The API allows retrieval of StarCraft engines variables and injection of custom AI scripts. 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

SparCraft is a combat simulation package for StarCraft, utilizing a sophisticated frame-fast forwarding system which allows thousands of unit moves to be executed per second [8]. It was designed to provide a testing ground for AI algorithms by accurately models units statistics and behaviors. However, due to the closed source nature of StarCraft, SparCraft only provides a best guess of the engine’s inner workings. Other limitations include lack of collisions, fog-of-war, and implementation of spell-casters and air 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 UCT’s current units at time \( t \). The aggregate score for each is equal to:

\[ S_t = \frac{\sum_{u \in U} hp(u) \cdot dps(u) - \sum_{e \in E} hp(e) \cdot \sum_{u \in U} hp_0(u)}{\sum_{u \in U} hp_0(u)} \] (2)

where \( U \) is the set of current UCT units, \( E \) is the set of current enemy units. We normalize by dividing by the initial start health \( hp_0 \) of UCT’s units.

Using the algorithm of MCTS, if a node did not end the game, it would generate its children nodes. Move nodes alternated between UCT’s own 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.

VI. TESTING AND RESULTS

Two test sets were created for the purpose of examining overgeneralization. Each individual matchup is composed of two different types of units pitted against each other, similar to a skirmish encountered in competitive StarCraft. It was important to create balanced matchups, and part of the challenge was finding 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:

$$ W = \frac{U}{G} $$

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 the win ratio. Then, using that C-Value on the test set, UCT was again matched against the default AI, and the resulting difference in win ratio gave the generalization error.

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 important for UCT to exploit nodes with high reward rather than explore nodes with less visits. Therefore, the highest C-Value was set to one, giving exploration and exploitation equal priority. The specific C-Values were: 0.075, 0.2, 0.4, 0.6, 0.8, 1.0.

Training set 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 across those 3 training distributions was calculated and compared to the average of their respective win ratio when applied to the larger test set. The graph below summarizes the results.

Fig. 3. A typical individual matchup: MCTS Marines (left) vs. Default AI Zerglings and Hydralisk (right)

Fig. 4. Results for Test Set 1

When training set size is equal to 1, UCT can be fit C-Value so that the win ratio is almost 100 percent. However, when that same C-Value is tested against the test set, the win ratio 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 win ratio and test set win ratio almost decrease to 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 amount of units. In contrast to Test Set 1, Test Set 2 favored more exploration. Therefore, the highest C-Value was set to four, which places a higher 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 win ratio across those 5 training distributions was calculated and compared to the average of their respective win ratio when applied to the larger test set. The graph below summarizes the results.

![Graph](image)

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

It is important to note that UCT only wins against the default AI 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

As we can see, there is generalization error 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 \( \sigma \). 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:

\[
R_S = E[\sup_{f \in S} \frac{1}{m} \sum_{i=1}^{m} \sigma_i f_i]
\]

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 \( m \):

\[
R_m = E[R_S]
\]

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

\[
E_D \leq E_S + 2R_m + \sqrt{\ln \frac{1}{\delta}} m
\]

We apply the Rademacher complexity bounds to our two test sets. The Rademacher complexity for training set \( m \) was smoothed by averaging over several 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 bottom dotted line representing the worst case generalization error As we can see, the test set performance lies in between the bounds. Additionally, it is important to note that the generalization bounds begin to tighten as training size increases.

![Graph](image)

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 Test Set 1, the bounds are tighter.
Again, the generalization bounds begin to tighten as training size increases.

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 can equal approximately 100 percent. In general, smaller training set result in worse bounds than larger training set sizes.

The most important take-away 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. Lastly, this also provides a basis for evaluating a group of matchup scenarios to 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 an equation. Different weights can be used to tuned for specific matchups. Lastly, this project did not explore air units or spell-caster units in StarCraft. Future work should include different matchups with air, ground, and spell-caster units. The potential of MCTS to RTS games exists, it’s just a matter of finding the correct scenario.

**ACKNOWLEDGMENT**

A very special thanks to Professor Michael Littman for mentoring and teaching me all the artificial intelligence knowledge I needed to complete this project and for pointing me in the right direction.

Thanks to Professor Stefanie Tellex for reading and teaching me CS141, which peaked 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 countless 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**


**ACKNOWLEDGMENT**

A very special thanks to Professor Michael Littman for mentoring and teaching me all the artificial intelligence knowledge I needed to complete this project and for pointing me in the right direction.

Thanks to Professor Stefanie Tellex for reading and teaching me CS141, which peaked 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 countless 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**


