An Analysis of Model-Based Heuristic Search Techniques for StarCraft Combat Scenarios

David Churchill
Memorial University, St. John's, A1B 3X5, Canada (dchurchill@mun.ca)

Zeming Lin, Gabriel Synnaeve
Facebook AI Research, New York, 11205, USA (zlin@fb.com, gab@fb.com)

Abstract

Real-Time Strategy games have become a popular test-bed for modern AI system due to their real-time computational constraints, complex multi-unit control problems, and imperfect information. One of the most important aspects of any RTS AI system is the efficient control of units in complex combat scenarios, also known as micromanagement. Recently, a model-based heuristic search technique called Portfolio Greedy Search (PGS) has shown promising performance for providing real-time decision making in RTS combat scenarios, but has so far only been tested in SparCraft: an RTS combat simulator. In this paper we present the first integration of PGS into the StarCraft game engine, and compare its performance to the current state-of-the-art deep reinforcement learning method in several benchmark combat scenarios. We then perform the same experiments within the SparCraft simulator in order to investigate any differences between PGS performance in the simulator and in the actual game. Lastly, we investigate how varying parameters of the SparCraft simulator affect the performance of PGS in the StarCraft game engine. We demonstrate that the performance of PGS relies heavily on the accuracy of the underlying model, outperforming other techniques only for scenarios where the SparCraft simulation model more accurately matches the StarCraft game engine.

Introduction and Background

AI researchers have often used games as a test-bed for evaluating the performance of their artificial intelligence systems. Recently, advances in deep learning techniques, combined with heuristic search, have led to the defeat of professional players of Go and No Limit Texas-Hold-em Poker by the AlphaGo (Silver et al. 2016) and DeepStack (Moravčík et al. 2017) programs, heralding the end of human dominance in traditional two-player games. As their next challenge, many AI researchers have chosen to tackle Real-Time Strategy (RTS) video games, which present more complex problems than traditional board games, with properties such as real-time computational constraints, simultaneous multi-unit control, and imperfect information (Ontanón et al. 2013).

Unit micromanagement in RTS games ("micro") is the problem of making decisions on how to most effectively control the specific movements and actions of units, usually in a combat-related context, and is a key aspect of competitive play. The properties of RTS game combat make it a particularly challenging problem, involving the simultaneous real-time control of dozens of units with varying properties. Each unit on the battlefield can be controlled individually, leading to an exponential number of action combination possibilities that must be chosen from by a player at a given state.

A number of heuristic search based approaches for RTS combat have been introduced in recent years, such as Alpha-Beta Considering Durations (ABCD), UCT Considering Durations (UCT-CD), and Portfolio Greedy Search (PGS). (Churchill and Buro 2013) demonstrated that PGS outperforms other search-based methods in medium and large-scale combat scenarios, and is currently the top performing heuristic search based method for RTS game combat. As PGS is based on heuristic search, it relies on a forward model of the environment, and its only published results so far have been in simulation, not the StarCraft game engine.

In this paper we will present the first experiments which use PGS to control units in the StarCraft game engine, and explore possible issues related to its reliance on an abstract simulator as a forward model. We will compare its performance to the current state-of-the-art technique in deep reinforcement learning as well as several baseline scripted players. By performing these experiments in both the real StarCraft game engine as well as in simulation, we can highlight any differences in results and attempt to draw conclusions about the feasibility of such model-based approaches in real RTS game engines. We begin by first introducing each of the StarCraft combat techniques we will use in our experiments, followed by a description of the combat scenarios used to test them. We then present the results of three separate experiments and discuss our findings.

RTS Unit Micro Scripts

The simplest and most common technique for unit micro in retail RTS games is to use hard-coded scripted behaviours. We define a scripted player to be one that implements a static series of scripted rules, similar to a finite state machine or simple decision tree, but does not perform any sort of forward look-ahead evaluation or learning. In this paper we will use the following common scripts as baseline players to compare the performance of PGS and a reinforcement learning policy:

- c AttackClosest - If a unit is within the attack range of an enemy unit, it will attack the closest enemy unit and wait in the same position until it has reloaded to attack again. If it is not within attack range of any units, it will move toward the closest enemy unit.
- w AttackWeakest - AttackClosest, but will attack the enemy unit which has the lowest current hit points.
- k Kiter - AttackClosest, but moves away from an enemy unit while reloading instead of standing still. This behaviour is very effective when used with long ranged attackers vs. short ranged attackers and is known as 'kiting'.
- h HoldPosition - AttackClosest, but units will never move from their initial positions
- n NoOverkill - Adds a condition to a script that no unit will be assigned to attack an enemy unit which has already been assigned predicted lethal damage on this time step.

For brevity, future references to these scripts in this paper may use the character abbreviation listed before the script name. These scripts can also have their behaviours combined, for example the script $wckn$ would be "attack $w$eakest enemy unit with highest priority, then $c$losest, $k$ite when reloading, with no overkill".

Portfolio Greedy Search

RTS combat game scenarios are difficult for traditional heuristic search techniques to solve, due to the exponential number of possible action combinations to choose from at any state. Portfolio Greedy Search is a heuristic search technique specifically designed for decision making in games with large action spaces. Instead of searching all possible action combinations for units at a give state, PGS reduces the action space generating unit actions from a set of scripted behaviours which we call a *portfolio*. Unlike search algorithms such as Minimax or MCTS, PGS does not build a game tree to search an exponential number of action combinations, but instead uses a hill-climbing approach to reduce the number of actions searched to a linear number. State evaluations in PGS are carried out via game playouts. A full description of the PGS algorithm is available in (Churchill and Buro 2013), but a brief outline is as follows:

1. PGS takes as input an initial game state and a portfolio of scripts, chosen to cover a range of tactical possibilities.
2. A single script is chosen as the initial *seed* script for both players, and is assigned to control each unit of the corresponding player, we call this a *unit-script assignment*.
3. For each unit a player controls, PGS iterates through each script in the portfolio and assigns it to the current unit.
4. A playout is performed to evaluate the current unit-script assignment, simulating the result of combat using the current unit script assignment for some number of turns.
5. The best combination of unit-script assignments is chosen as the one which has the maximum playout value.
6. Steps 3-5 can then be repeated any number of times for both players, improving via self play.
7. When a time limit is reached, the actions produced by the final unit-script assignment are returned.

SparCraft Combat Simulator

As PGS performs playout evaluations to decide on which actions to perform, it requires a forward model of the environment in order to function. The BWAPI programming interface allows for reading the memory of StarCraft and issuing actions to the game, but it does not allow us to directly copy or manipulate local game state instances, which is required for the PGS forward-model. Therefore, in order to perform PGS on combat scenarios in StarCraft, we must use a system which enables us to efficiently simulate the game's combat engine. The model we will use to carry out PGS is SparCraft (Churchill 2016a), a StarCraft combat simulation library written in C++. Specifically designed to be a test-bed for RTS combat algorithms, it models StarCraft combat scenarios in a manner that balances accuracy of simulation with speed of computation, and has the following features:

- It models all StarCraft unit types that are able to attack and move, along with all of their properties such as hit points, weapon damage, speed, and attack range.
- Units in SparCraft are able to perform the following actions: Move to a given (x,y) pixel location, Attack a target enemy unit, and Hold Position for a given duration.
- SparCraft states can be fast-forwarded to the next time step in which a unit is able to act, allowing it to skip many game frames and save significant computation.
- SparCraft actions have set durations, and must be carried to completion with no interruption. For example, a unit must carry out a Move action to its destination and cannot take any other action or stop until it is completed.
- SparCraft does not currently model fog of war, unit acceleration, or spell casting.
- Importantly, SparCraft does not model unit collisions. This greatly reduces the accuracy of the simulation in comparison to the StarCraft game engine, but was necessary to be fast enough to facilitate heuristic search. If unit collisions were simulated, SparCraft would also require a multi-agent path-finding system similar to the StarCraft game engine, which would be computationally expensive, and difficult to accurately engineer.

StarCraft Engine Interface

SparCraft is interfaced with the StarCraft game engine by incorporating it into UAlbertaBot, an open source StarCraft AI agent (Churchill 2016b). In order for SparCraft and PGS to produce actions that can be carried out in the StarCraft game engine, the following process occurs at each time step of a given battle:

1. The current StarCraft game state is read within UAlbertaBot via BWAPI and translated into a corresponding SparCraft game state.
2. The PGS algorithm and all combat simulations are run within the SparCraft combat simulator.
3. When the PGS time limit is reached, the resulting SparCraft actions are returned to UAlbertaBot, translated into their corresponding BWAPI actions, and are issued to the units in the StarCraft game engine via BWAPI.

Deep Reinforcement Learning Player

We now give a brief overview of the deep reinforcement learning model of (Usunier et al. 2017) applied to StarCraft micromanagement, and how it was trained.

The *actions* each unit can take are: movement in the 8 compass directions around the unit, no-op, and all the attack actions to any of the enemy units, totaling $9+\#units$. Units take actions every 9 game frames, chosen by cross validating between 5 and 17. Each unit takes its action "in turn" in the same frame, such that the n-th unit's action is conditioned on the state and the $n-1$ actions that have already been chosen. The units' ordering is randomly sampled each time.

The *model* is a neural network that takes a tuple (actor unit, attack|move, target position) as input and outputs a score. We have as many of these tuples as there are possible actions for each unit. The *state* is converted into a set of numerical features by viewing it as a graph where the units are nodes and the edges denote targeting. Each unit is represented by: its hit points, shield, cooldown, previous command type, type of the current command, attack damage, type, armor, whether it is an enemy, its position relative to the acting unit, distances between the actor and its previous target, and between the actor and the current target. Each such input tuple is fed into a 2 layer neural network. Then, the output of this network is "pooled" to reduce it to a fixed size state embedding vector, via concatenating the average and the max over all dimensions. The state embedding vector is then concatenated with a learned vector representing the type of the *action*, and is fed to another 2 layer neural network. This outputs a scalar that encodes the score of the action for the given unit in the current state.

The model is *trained* via an episodic on-policy algorithm. After each battle, the weights are updated w.r.t. what was predicted in the immediate last battle. The reward signal over an episode is proportional to the total damage inflicted minus the total damage incurred along the episode, normalized for the number of units. The algorithm performs deterministic exploration, by adding noise to the model — *not* by sampling in the actions distribution. For each battle, Gaussian noise (one vector per battle) is added to the parameters of the last layer of the neural network, allowing to compute a zero-order (ZO) estimate (Kiefer, Wolfowitz, and others 1952) of the policy gradient, and updating the weights of this layer. A heuristic is used to compute an approximation of the gradient to backpropagate to the rest of the network. More details are provided in the original paper (Usunier et al. 2017).

TorchCraft

For all the experiments from (Usunier et al. 2017), including scripted players and the graph-based deep model trained through reinforcement learning, we used TorchCraft (Synnaeve et al. 2016), which connects Torch (Collobert, Kavukcuoglu, and Farabet 2011) to StarCraft. It enables deep learning research on StarCraft by making it easy to connect (through ZeroMQ) one training process to several game instances, and to decouple the process with the game simulation from the one with the neural network inference. Currently, TorchCraft provides C++, Lua, and Python clients, and a compression-enabled state serialization format.

Experiments

Three different experiments were carried out: the first to compare the performance of the combat solutions discussed in the previous sections, the second to compare the results of PGS in simulation vs. in the StarCraft engine, and the third to show effects related to varying the parameters of the SparCraft simulation. These experiments used each of the proposed methods to control units within a number of different combat scenarios. Before detailing each experiment, we will first describe the scenarios in which they will take place.

Combat Scenarios

The combat scenarios used were hand-crafted to represent battles of varying complexity, using several types of units with a range of properties, each posing unique tactical challenges in order to be successful. The scenario categories are:

mXvY - Player controls X Terran Marines (slow, ranged, ground unit) vs. Y enemy controlled Terran Marines.

wXvY - Player controls X Terran Wraiths (fast, ranged, flying units) vs. Y enemy controlled Terran Wraiths. Flying units do not incur unit collision, and so more tactical movement can occur than in the mXvY scenario.

dXzY - Both players control X Protoss Dragoons (strong, ranged, ground unit) and Y Protoss Zealots (slow, strong, ground unit). In this scenario, sending the Zealots to the front lines to absorb damage while the Dragoons attack from long range is most effective.

Experiment 1: StarCraft Combat

Each of the combat scenarios were implemented in StarCraft: Broodwar via a custom map file in which the following sequence of events occur in order:

1. Two sets of identical units for each player spawn in separate groups, initially spaced far enough apart such that no units can attack each other.
2. Enemy controlled units approach and attack the player using StarCraft's built-in Attack-Move command.
3. The player's group of units is controlled via their chosen combat algorithm, with commands given via BWAPI.
4. When any player reaches 0 units, the winner is recorded and steps 1-3 are repeated.

For Experiment 1, the following specific scenarios were used: **d2z3**, **m5v5**, **m15v16**, and **m15v17**. For each of these scenarios, a total of 9 players were tested, playing 1000 battles each against the custom map's AttackClosest scripted player: 3 scripted players implemented BWAPI / StarCraft (c, wc, wcn), 1 deep reinforcement learned policy player implemented in TorchCraft, 4 scripted players implemented in SparCraft (c, $wchn$, $wckn$, wcn), and the Portfolio Greedy Search player (PGS). Scripted players were implemented in BWAPI / StarCraft as well as SparCraft in order to highlight any differences in performance that may arise from the conversion of states / actions from StarCraft to SparCraft and vice-versa. Each player also employed a 'protected' command mode that blocks issuing actions that would have interrupted an ongoing attack animation, which is a common occurrence in BWAPI / StarCraft AI development.

Experiment 2: SparCraft Simulated Combat

In addition to performing each of the combat scenarios in Experiment 1 in the StarCraft game engine, each battle using SparCraft players was also carried out entirely within the SparCraft simulator. At the start of each battle in Experiment 1, before any actions were taken in the StarCraft game engine, the battle was simulated entirely within SparCraft and the results were recorded. Enemy unit behaviour was controlled within SparCraft via the AttackClosest script, as this was the most similar behaviour to the enemy units in the custom map in Experiment 1. This experiment was performed in order to highlight the differences between the expected outcome of the SparCraft simulation and the results in the actual StarCraft game engine.

Experiment 3: SparCraft Parameter Variations

One of the main issues with model-based search methods such as PGS is that their result may be very sensitive to the accuracy of the model in which they are simulated. In order to demonstrate this issue, we performed two experiments, each which played PGS in the same setup as Experiment 1, however in each experiment a different parameter of the SparCraft combat simulation engine was varied. The two parameters that varied were:

- **Move Penalty** - In the StarCraft game engine, if a unit is moving in a given direction and then chooses to attack a unit in the opposite direction, it must first play an animation in which it turns to face the enemy unit before firing. Since simulating this turning would add additional computation to the SparCraft engine, it instead abstracts the concept by introducing a Move Penalty in which a unit must wait a specific number of frames after moving before it is able to attack. In this experiment, Move Penalty values from 0 to 16 StarCraft game frames were tested.
- **Range Addition** - In the StarCraft game engine, different types of units have varying attack ranges measured in pixels, and a unit can attack a target unit only if any portion of the target unit's rectangular bounding box is within its attack range. To avoid this bounding box computation, SparCraft abstracts the concept by adding a given number of pixels to the attack range of each unit, simulating the extra distance required. Range Addition values between 0 and 64 StarCraft game pixels were tested. This parameter is particularly important, because a slight difference in attack range between SparCraft and StarCraft may result in a unit being given a move command instead of an attack command, leading to a significant loss in damage output.

PGS Search Environment / Parameters

All experiments involving PGS were performed single-threaded on an Intel(R) Core(TM) i7-6800k CPU @ 3.40GHz running Windows 10. A total of 32GB DDR4 3000mhz RAM was available, however the maximum amount of RAM that used by PGS during the experiments was measured at less than 10MB. PGS and UAlbertaBot were implemented in C++ using BWAPI, and compiled using Visual Studio 2017 Enterprise Edition. The parameters used for Portfolio Greedy Search in all scenarios for Experiment 1 and 2 were as follows:

- Search Time Limit: 10ms
- Improvement Iterations I: 1
- Response Iterations R: 0
- Playout Evaluation Max Turns: 100
- Enemy Seed Script: c (AttackClosest)
- Script Portfolio Used: [wcn, $wckn$, $wchn$]

  wcn - Attack enemy units with priority weakest then closest, with no overkill

  $wckn$ - wcn, but kites enemy unit while reloading

  $wchn$ - wcn, but units hold position and do not move

The 10ms time limit was chosen to simulate real-time performance within a StarCraft AI agent in a competition setting, where bots are allowed a total of 50ms of computation per frame. Higher time limits of up to 40ms were tested, but did not significantly increase performance in the StarCraft game engine, as simulation accuracy appeared to bottleneck performance. The Script Portfolio was constructed such that all three major tactical unit movements are present: straight line movement toward and away from enemy units, as well as holding position. Each of the scripts in the Portfolio were tested in Experiment 1 and 2 along with PGS to demonstrate whether the PGS algorithm was able to outperform each of the scripts by combining their behaviours. PGS parameters for Experiment 3 were identical to the ones for Experiments 1 and 2, with a lowered time limit of 5ms down from 10ms (since many more games had to be played). For Experiments 1 and 2, SparCraft parameters for all scenarios used a MovePenalty of 8 frames, and a RangeAddition of 32 pixels, which were determined empirically via the results of previous experimentation similar to Experiment 3.

Results and Discussion

Experiment 1 and 2

The results of Experiment 1 and 2 can be seen in Table 1, and show the scores of all of the implemented players over 1000 battles in their corresponding scenarios, with score equal to player wins + draws/2. We will discuss the results of each scenario individually:

d2z3 - In this scenario, each player controls 2 ranged Dragoon units and 3 melee Zealot units, with the optimal strategy involving positioning the Zealot units on the front lines while the Dragoons attack from behind. The best result for Experiment 1 in this scenario were obtained by the RL policy, followed by the wc script, then by PGS. Of particular interest for this scenario is the vastly better performance of the BWAPI scripted player c (AttackClosest, column 1) and the same scripted player implemented via SparCraft (column 7). Upon visual inspection of the battles there appear to be two reasons for this difference, both caused by abstractions in the SparCraft simulation. The first reason is due to the notoriously erratic behaviour of Dragoons in StarCraft, which often have issues with path-finding, moving small distances, and attacks being canceled if improperly timed. As SparCraft does not simulate these behaviours accurately, it sometimes issued move commands which interrupted and canceled the Dragoon's attacks, despite having several measures in place

	Experiment 1: StarCraft Game Engine									Experiment 2: Simulation				
	Scripted Players			TorchCraft	SparCraft Players					SparCraft Players				
Scenario	c	wc	wcn	ZO RL	c	$wchn$	$wckn$	wcn	PGS	c	$wchn$	$wckn$	wcn	PGS
d2z3	.67	.83	.50	**.90**	.13	.25	.35	.39	.72	0.5	0.0	0.0	1.0	1.0
m5v5	.94	.96	.83	**1.0**	.42	.79	.08	.53	.88	0.5	0.0	0.0	1.0	1.0
m15v16	**.81**	.10	.68	.79	.55	.08	.01	.26	.22	0.0	0.0	1.0	1.0	1.0
w15v17	.20	.02	.12	.49	.10	.06	.88	.23	**.94**	0.0	0.0	1.0	1.0	1.0

Table 1: Experiment 1 and 2 scores (wins + draws/2) over 1000 battles for each of the scenarios, for all methods and for scripted baselines. Scores in columns under the Experiment 1 header are for battles in the StarCraft game engine. Scores in columns under the Experiment 2 header are for battles carried out entirely within the SparCraft combat simulator vs. an AttackClosest scripted player. The best result for each scenario in Experiment 1 is in bold.

to attempt to stop this. The second issue was due to SparCraft not modeling unit collisions, it would command the player's Zealot units to walk through an enemy unit to attack a lower health target, and get stuck. Despite this, PGS outperformed of all of its individual portfolio scripts, indicating that the search did in fact work well.

The results for Experiment 2 can be seen in the last 5 columns, in which the same SparCraft players performed the battle entirely in the SparCraft simulator vs. the AttackClosest (c) script. Since the battle was symmetric, the results for script c were all ties vs. itself in the simulator. The hold position ($wchn$) and kiting ($wckn$) scripts lost all of their battles, and wcn and PGS won all of their battles. In fact for all scenarios, both wcn and PGS win all of their battles in simulation but do not win all games in the StarCraft game engine.

m5v5 - This simpler scenario is also symmetric, with each player controlling 5 ranged Terran Marine units. The best results come from the RL policy with a near perfect score, with two scripted players c and wc coming just behind them. As this is a simpler scenario consisting of the same unit type, simply attacking the closest unit performs quite well, due to the fact that the enemy player starts by walking all of their units toward the player's units. We can again see a large difference between the BWAPI and SparCraft versions of the scripted player Attack Closest (c). Again, as SparCraft does not model unit collisions, there are a number of times when the initial movement of the units results in a collision, providing a small disadvantage, but enough for the enemy to win.

The results for Experiment 2 are identical to the first scenario, where SparCraft simulated that it should have won all of the battles. As we continue to see this trend, it becomes more obvious that the differences in the SparCraft and StarCraft engine are a major source of error, and leading to worse performance.

m15v16 - In this battle, the player is placed at a disadvantage, having one less Marine than the enemy player. Since both players units spawn in a cluster, it is difficult for clever movement to play much of a role in success due to unit collisions. In this scenario, the best performance is seen by the BWAPI scripted Attack Closest (c) player, followed closely behind by the RL policy. PGS performs worse than its version of the Attack Closest script, again because of issues related to unit collisions. PGS simulates incorrectly that it can move its units in on top of each other to attack the enemy, resulting in reduced performance.

The results for Experiment 2 in this scenario are interesting, as it believes that it should be able to win every battle in simulation by kiting the opponent ($wckn$), but in the SparCraft engine it ends up losing nearly every battle. On visual inspection of the games in simulation, we can see that kiting the marines back and forth has the effect of constantly cycling low hit point units to the back lines, taking them out of range of the enemy Attack Closest script, causing units to stay alive much longer. However, when it tries to implement this tactic in the StarCraft game engine, the units collide and performance suffers.

w15v17 - In this scenario, the player is again at a disadvantage, controlling 15 Wraiths vs. 17. The most important aspect of this scenario is that flying units in StarCraft do not incur unit collisions, and the Wraith unit does not need to stop moving to fire its rockets, meaning that the SparCraft simulation of the scenario is much more accurate than the ground unit scenarios. We see this reflected in the results, with PGS outperforming all other methods.

We can draw several conclusions from these experiments:

- The reinforcement-learned ZO policy outperforms PGS in smaller combat scenarios with less units.
- The reinforcement-learned ZO policy outperforms PGS in situations where the SparCraft simulation is less accurate, particularly in ground unit battles where many unit collisions can happen.
- For large ground unit battles in the StarCraft game engine, it seems difficult to outperform simple scripted players, as unit collisions tend to hinder clever tactical movements, and simulations are less accurate.
- PGS outperforms all other methods in scenarios where SparCraft more accurately simulates the battle, especially with flying units which do not incur collisions.
- PGS performs well in simulation, outperforming any scripted players that it faced, however the actions produced by PGS are often blunders when performed in the StarCraft game engine, due to inaccuracies in the simulation.

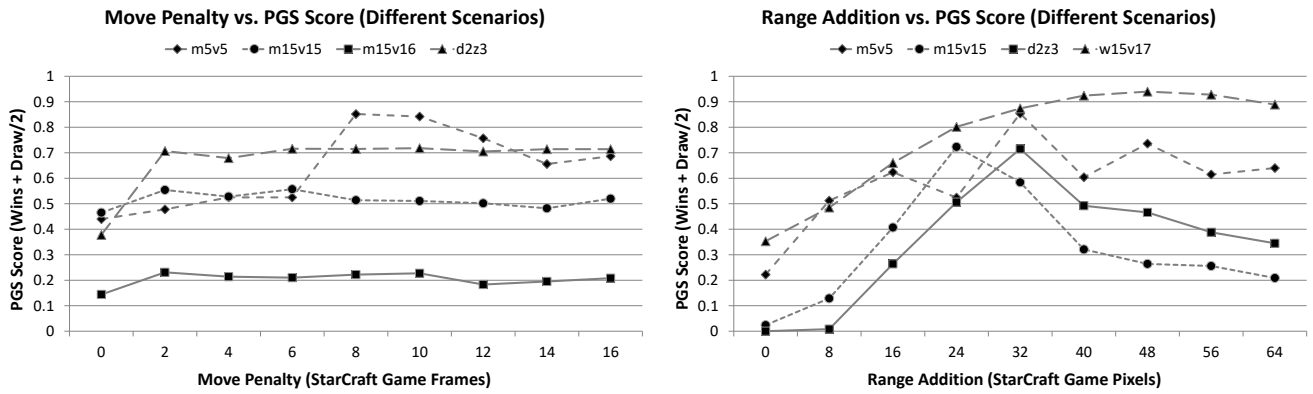

Figure 1: Results of Experiment 3 showing the effects on PGS results with varying parameters of the SparCraft simulation. Shwon are results for varying the Move Penalty (left) and Range Addition (right) parameters within SparCraft, while all other parameters remain fixed. 1000 battles were played for each parameter/scenario combination.

Experiment 3

The results for Experiment 3 can be seen in Figure 1, and show the effects on the PGS score in various scenarios when varying two different parameters of the SparCraft combat simulation, the Movement Penalty (left) and the Range Addition (right). From these results, we can see that varying these parameters can have drastic effects on PGS score for some scenarios, but little effect on other scenarios. For example, a Move Penalty difference of just 2 frames (from 6 to 8) yields a difference of 0.32 in the m5v5 scenario, while yielding no noticeable difference in any other scenario. Intuitively, raising the value for the Move Penalty has the effect of discouraging movement-based tactics, as each movement will delay the next time a unit can attack. On visual inspection of the battles for d2z3, with a Move Penalty of 0 frames, the PGS controlled Dragoons attempted to Kite the enemy units for the entire duration of the battle due to there being no movement penalty, which led to them dealing significantly less overall damage due to the unit turning animation in the StarCraft game engine which was not compensated for.

The Range Addition parameter has even more dramatic effects, with every scenario yielding very low scores for a Range Addition of 0, while hitting a range of good scores between 24 and 40 pixels. Intuitively, a Range Addition value that is too small may force units to move too close to an enemy unit before firing, incurring incoming damage while they move. A value that is too high will result in units attempting to attack units that are too far away, possibly passing by enemy units en route, again incurring additional damage. We believe that to obtain a higher level of accuracy, the Range Addition parameter must be set individually for each unit type, as well as each target unit type, leading to a very difficult parameter tuning problem.

This experiment demonstrates the fragility of model-based methods, whose results can vary dramatically based on the accuracy of the simulation involved. Without access to the source code of a game engine, accurately creating a simulator such as SparCraft becomes an exercise in reverse engineering, data mining, and parameter tuning, none of which are required for systems such as deep learning which do not rely on existing environment models.

Conclusion and Future Work

In this paper we have presented the first implementation of PGS in the StarCraft engine, and compared its performance directly with a state-of-the-art deep reinforcement learning technique for StarCraft unit micro. By running both methods through the same set of StarCraft combat scenarios, we were able to explore the strengths and weaknesses of each. We have shown that the reinforcement learned policy outperforms PGS for small combat scenarios, and for scenarios in which the SparCraft simulator used by PGS is inaccurate. We have also shown that for scenarios in which the SparCraft simulator is more accurate, such as for flying units, PGS outperforms all other methods, and may be suitable for use in such situations in a StarCraft AI competition agent because of its low running time of only 10ms. We also observed that for large scenarios involving many ground units, simple scripted players such as AttackClosest perform surprisingly well in the StarCraft game engine.

We can conclude that neither method is *currently* a clear winner for all scenarios, and both offer unique strengths and weaknesses. The reinforcement learned policy offers good performance for small battle sizes without the need for an environment model, however it requires a significant training period to achieve this level of performance. Heuristic search techniques offer good performance in situations where an accurate, efficient model is available, and require no training period, however creating an accurate model is time consuming, may require parameter optimization, and in many scenarios (such as real world problems) may be impossible to construct. It is the opinion of the authors that if learned micro policies continue to improve to surpass the capabilities of heuristic search in all scenarios, they would be preferable in situations where offline training periods are acceptable. In the end, we do not necessarily view search and learning techniques as strictly competing, but as techniques that can be used together to create systems that are more powerful than either could be alone. In the future, we hope to combine heuristic search and deep learning to create the next generation of RTS game combat systems.

References

Churchill, D., and Buro, M. 2013. Portfolio greedy search and simulation for large-scale combat in StarCraft. In *IEEE Conference on Computational Intelligence in Games (CIG)*, 1–8. IEEE.

Churchill, D. 2016a. SparCraft: Open Source StarCraft Combat Simulation. `https://github.com/davechurchill/ualbertabot/wiki/SparCraft-Home`.

Churchill, D. 2016b. UAlbertaBot. `https://github.com/davechurchill/ualbertabot/`.

Collobert, R.; Kavukcuoglu, K.; and Farabet, C. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376.

Kiefer, J.; Wolfowitz, J.; et al. 1952. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics* 23(3):462–466.

Moravčík, M.; Schmid, M.; Burch, N.; Lisỳ, V.; Morrill, D.; Bard, N.; Davis, T.; Waugh, K.; Johanson, M.; and Bowling, M. 2017. Deepstack: Expert-level artificial intelligence in no-limit poker. *arXiv preprint arXiv:1701.01724*.

Ontanón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2013. A survey of real-time strategy game AI research and competition in StarCraft. *TCIAIG*.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529(7587):484–489.

Synnaeve, G.; Nardelli, N.; Auvolat, A.; Chintala, S.; Lacroix, T.; Lin, Z.; Richoux, F.; and Usunier, N. 2016. Torchcraft: a library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*.

Usunier, N.; Synnaeve, G.; Lin, Z.; and Chintala, S. 2017. Episodic exploration for deep deterministic policies: An application to starcraft micromanagement tasks. In *ICLR*.